Reverse Engineering for Beginners

Dennis Yurichev
<dennis@yurichev.com>

©2013-2014, Dennis Yurichev.
This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/3.0/.
Text version (May 8, 2014).
There is probably a newer version of this text, and also Russian language version also accessible at http://yurichev.com/RE-book.html. A4-format version is also available on the page.
You may also subscribe to my twitter, to get information about updates of this text, etc: @yurichev [1], or to subscribe to mailing list [2].

[1] https://twitter.com/yurichev
[2] http://yurichev.com/mailing_lists.html

Please donate!

I worked more than year on this book, here are more than 600 pages, and it's free. Same level books has price tag from $20 to $50.

More about it: 0.1.

Short contents

I Code patterns 1

II Important fundamentals 458

III Finding important/interesting stuff in the code 463

IV OS-specific 496

V Tools 570

VI More examples 576

VII Other things 726

VIII Books/blogs worth reading 754

IX Exercises 758

Afterword 831

Appendix 833

Acronyms used 882

Contents

0.1 Preface xxi

I Code patterns **1**

1 Short introduction to the CPU **3**

2 Hello, world! **4**
2.1 x86 4
2.1.1 MSVC—x86 4
2.1.2 GCC—x86 6
2.1.3 GCC: AT&T syntax 7
2.2 x86-64 9
2.2.1 MSVC—x86-64 9
2.2.2 GCC—x86-64 10
2.3 ARM 11
2.3.1 Non-optimizing Keil + ARM mode 12
2.3.2 Non-optimizing Keil: thumb mode 14
2.3.3 Optimizing Xcode (LLVM) + ARM mode 15
2.3.4 Optimizing Xcode (LLVM) + thumb-2 mode 16

3 Function prologue and epilogue **19**
3.1 Recursion 19

4 Stack **21**
4.1 Why does the stack grow backward? 22
4.2 What is the stack used for? 22
4.2.1 Save the return address where a function must return control after execution 22
4.2.2 Passing function arguments 24
4.2.3 Local variable storage 25
4.2.4 x86: alloca() function 26
4.2.5 (Windows) SEH 29

4.2.6 Buffer overflow protection . . . 29
4.3 Typical stack layout . . . 29

5 `printf()` with several arguments **30**
5.1 x86: 3 arguments . . . 30
5.1.1 MSVC . . . 30
5.1.2 MSVC and OllyDbg . . . 31
5.1.3 GCC . . . 35
5.1.4 GCC and GDB . . . 36
5.2 x64: 8 arguments . . . 39
5.2.1 MSVC . . . 40
5.2.2 GCC . . . 40
5.2.3 GCC + GDB . . . 41
5.3 ARM: 3 arguments . . . 44
5.3.1 Non-optimizing Keil + ARM mode . . . 44
5.3.2 Optimizing Keil + ARM mode . . . 45
5.3.3 Optimizing Keil + thumb mode . . . 45
5.4 ARM: 8 arguments . . . 46
5.4.1 Optimizing Keil: ARM mode . . . 46
5.4.2 Optimizing Keil: thumb mode . . . 48
5.4.3 Optimizing Xcode (LLVM): ARM mode . . . 48
5.4.4 Optimizing Xcode (LLVM): thumb-2 mode . . . 49
5.5 By the way . . . 50

6 scanf() **51**
6.1 About pointers . . . 51
6.2 x86 . . . 52
6.2.1 MSVC . . . 52
6.2.2 MSVC + OllyDbg . . . 54
6.2.3 GCC . . . 56
6.3 x64 . . . 56
6.3.1 MSVC . . . 56
6.3.2 GCC . . . 57
6.4 ARM . . . 58
6.4.1 Optimizing Keil + thumb mode . . . 58
6.5 Global variables . . . 59
6.5.1 MSVC: x86 . . . 59
6.5.2 MSVC: x86 + OllyDbg . . . 61
6.5.3 GCC: x86 . . . 62
6.5.4 MSVC: x64 . . . 63
6.5.5 ARM: Optimizing Keil + thumb mode . . . 63
6.6 scanf() result checking . . . 65
6.6.1 MSVC: x86 . . . 66
6.6.2 MSVC: x86: IDA . . . 67

6.6.3 MSVC: x86 + OllyDbg . . . 71
6.6.4 MSVC: x86 + Hiew . . . 72
6.6.5 GCC: x86 . . . 74
6.6.6 MSVC: x64 . . . 74
6.6.7 ARM: Optimizing Keil + thumb mode . . . 75

7 Accessing passed arguments 77
7.1 x86 . . . 77
7.1.1 MSVC . . . 77
7.1.2 MSVC + OllyDbg . . . 79
7.1.3 GCC . . . 79
7.2 x64 . . . 80
7.2.1 MSVC . . . 80
7.2.2 GCC . . . 82
7.2.3 GCC: uint64_t instead int . . . 84
7.3 ARM . . . 85
7.3.1 Non-optimizing Keil + ARM mode . . . 85
7.3.2 Optimizing Keil + ARM mode . . . 86
7.3.3 Optimizing Keil + thumb mode . . . 86

8 One more word about results returning. 87

9 Pointers 91
9.1 Global variables example . . . 91
9.2 Local variables example . . . 95
9.3 Conclusion . . . 97

10 Conditional jumps 98
10.1 x86 . . . 99
10.1.1 x86 + MSVC . . . 99
10.1.2 x86 + MSVC + OllyDbg . . . 101
10.1.3 x86 + MSVC + Hiew . . . 103
10.1.4 Non-optimizing GCC . . . 105
10.1.5 Optimizing GCC . . . 105
10.2 ARM . . . 107
10.2.1 Optimizing Keil + ARM mode . . . 107
10.2.2 Optimizing Keil + thumb mode . . . 109

11 switch()/case/default 111
11.1 Few number of cases . . . 111
11.1.1 x86 . . . 111
11.1.2 ARM: Optimizing Keil + ARM mode . . . 114
11.1.3 ARM: Optimizing Keil + thumb mode . . . 115
11.2 A lot of cases . . . 116

11.2.1 x86 . . . 116
11.2.2 ARM: Optimizing Keil + ARM mode . . . 120
11.2.3 ARM: Optimizing Keil + thumb mode . . . 122

12 Loops **125**
12.1 x86 . . . 125
12.1.1 OllyDbg . . . 130
12.1.2 tracer . . . 131
12.2 ARM . . . 133
12.2.1 Non-optimizing Keil + ARM mode . . . 133
12.2.2 Optimizing Keil + thumb mode . . . 134
12.2.3 Optimizing Xcode (LLVM) + thumb-2 mode . . . 134
12.3 One more thing . . . 135

13 strlen() **136**
13.1 x86 . . . 136
13.1.1 Non-optimizing MSVC . . . 136
13.1.2 Non-optimizing GCC . . . 138
13.1.3 Optimizing MSVC . . . 139
13.1.4 Optimizing MSVC + OllyDbg . . . 140
13.1.5 Optimizing GCC . . . 141
13.2 ARM . . . 143
13.2.1 Non-optimizing Xcode (LLVM) + ARM mode . . . 143
13.2.2 Optimizing Xcode (LLVM) + thumb mode . . . 144
13.2.3 Optimizing Keil + ARM mode . . . 145

14 Division by 9 **147**
14.1 x86 . . . 147
14.2 ARM . . . 149
14.2.1 Optimizing Xcode (LLVM) + ARM mode . . . 149
14.2.2 Optimizing Xcode (LLVM) + thumb-2 mode . . . 150
14.2.3 Non-optimizing Xcode (LLVM) and Keil . . . 150
14.3 How it works . . . 150
14.4 Getting divisor . . . 152
14.4.1 Variant #1 . . . 152
14.4.2 Variant #2 . . . 153

15 Working with FPU **155**
15.1 Simple example . . . 156
15.1.1 x86 . . . 156
15.1.2 ARM: Optimizing Xcode (LLVM) + ARM mode . . . 159
15.1.3 ARM: Optimizing Keil + thumb mode . . . 160
15.2 Passing floating point number via arguments . . . 161
15.2.1 x86 . . . 161

15.2.2 ARM + Non-optimizing Xcode (LLVM) + thumb-2 mode 162
15.2.3 ARM + Non-optimizing Keil + ARM mode 163
15.3 Comparison example 164
15.3.1 x86 164
15.3.2 Now let's compile it with MSVC 2010 with optimization option /Ox 166
15.3.3 GCC 4.4.1 167
15.3.4 GCC 4.4.1 with -O3 optimization turned on 170
15.3.5 ARM + Optimizing Xcode (LLVM) + ARM mode 171
15.3.6 ARM + Optimizing Xcode (LLVM) + thumb-2 mode 171
15.3.7 ARM + Non-optimizing Xcode (LLVM) + ARM mode 173
15.3.8 ARM + Optimizing Keil + thumb mode 174
15.4 x64 175

16 Arrays **176**
16.1 Simple example 176
16.1.1 x86 176
16.1.2 ARM + Non-optimizing Keil + ARM mode 179
16.1.3 ARM + Optimizing Keil + thumb mode 180
16.2 Buffer overflow 182
16.3 Buffer overflow protection methods 187
16.3.1 Optimizing Xcode (LLVM) + thumb-2 mode 190
16.4 One more word about arrays 193
16.5 Multidimensional arrays 193
16.5.1 x86 194
16.5.2 ARM + Non-optimizing Xcode (LLVM) + thumb mode 196
16.5.3 ARM + Optimizing Xcode (LLVM) + thumb mode 197

17 Bit fields **199**
17.1 Specific bit checking 199
17.1.1 x86 199
17.1.2 ARM 203
17.2 Specific bit setting/clearing 205
17.2.1 x86 205
17.2.2 ARM + Optimizing Keil + ARM mode 208
17.2.3 ARM + Optimizing Keil + thumb mode 208
17.2.4 ARM + Optimizing Xcode (LLVM) + ARM mode 209
17.3 Shifts 209
17.3.1 x86 210
17.3.2 ARM + Optimizing Xcode (LLVM) + ARM mode 213
17.3.3 ARM + Optimizing Xcode (LLVM) + thumb-2 mode 213
17.4 CRC32 calculation example 214

18 Structures **220**
18.1 SYSTEMTIME example . . . 220
18.2 Let's allocate space for structure using malloc() . . . 223
18.3 struct tm . . . 226
18.3.1 Linux . . . 226
18.3.2 ARM + Optimizing Keil + thumb mode . . . 231
18.3.3 ARM + Optimizing Xcode (LLVM) + thumb-2 mode . . . 232
18.4 Fields packing in structure . . . 234
18.4.1 x86 . . . 234
18.4.2 ARM + Optimizing Keil + thumb mode . . . 236
18.4.3 ARM + Optimizing Xcode (LLVM) + thumb-2 mode . . . 237
18.5 Nested structures . . . 238
18.6 Bit fields in structure . . . 239
18.6.1 CPUID example . . . 239
18.6.2 Working with the float type as with a structure . . . 244

19 Unions **249**
19.1 Pseudo-random number generator example . . . 249

20 Pointers to functions **253**
20.1 MSVC . . . 254
20.1.1 MSVC + OllyDbg . . . 257
20.1.2 MSVC + tracer . . . 258
20.1.3 MSVC + tracer (code coverage) . . . 260
20.2 GCC . . . 260
20.2.1 GCC + GDB (with source code) . . . 262
20.2.2 GCC + GDB (no source code) . . . 263

21 64-bit values in 32-bit environment **268**
21.1 Arguments passing, addition, subtraction . . . 268
21.2 Multiplication, division . . . 270
21.3 Shifting right . . . 273
21.4 Converting of 32-bit value into 64-bit one . . . 274

22 SIMD **277**
22.1 Vectorization . . . 278
22.1.1 Intel C++ . . . 279
22.1.2 GCC . . . 284
22.2 SIMD `strlen()` implementation . . . 287

23 64 bits **292**
23.1 x86-64 . . . 292
23.2 ARM . . . 302
23.3 Float point numbers . . . 303

24 Working with float point numbers using SIMD in x64 **304**
24.1 Simple example . . . 304
24.2 Passing floating point number via arguments . . . 306
24.3 Comparison example . . . 307
24.4 Summary . . . 308

25 Temperature converting **310**
25.1 Integer values . . . 310
25.1.1 MSVC 2012 x86 /Ox . . . 311
25.1.2 MSVC 2012 x64 /Ox . . . 313
25.2 Float point values . . . 313

26 C99 restrict **318**

27 Inline functions **322**

28 Incorrectly disassembled code **326**
28.1 Disassembling started incorrectly (x86) . . . 326
28.2 How random noise looks disassembled? . . . 327
28.3 Information entropy of average code . . . 350
28.3.1 x86 . . . 350
28.3.2 ARM (Thumb) . . . 351
28.3.3 ARM (ARM mode) . . . 351
28.3.4 MIPS (little endian) . . . 351

29 C++ **352**
29.1 Classes . . . 352
29.1.1 Simple example . . . 352
29.1.2 Class inheritance . . . 361
29.1.3 Encapsulation . . . 366
29.1.4 Multiple inheritance . . . 368
29.1.5 Virtual methods . . . 373
29.2 ostream . . . 377
29.3 References . . . 379
29.4 STL . . . 380
29.4.1 std::string . . . 380
29.4.2 std::list . . . 390
29.4.3 std::vector . . . 405
29.4.4 std::map and std::set . . . 417

30 Obfuscation **435**
30.1 Text strings . . . 435
30.2 Executable code . . . 436
30.2.1 Inserting garbage . . . 436

30.2.2 Replacing instructions to bloated equivalents . . . 436
30.2.3 Always executed/never executed code . . . 437
30.2.4 Making a lot of mess . . . 437
30.2.5 Using indirect pointers . . . 438
30.3 Virtual machine / pseudo-code . . . 438
30.4 Other thing to mention . . . 439

31 Windows 16-bit **440**
31.1 Example#1 . . . 440
31.2 Example #2 . . . 441
31.3 Example #3 . . . 442
31.4 Example #4 . . . 444
31.5 Example #5 . . . 447
31.6 Example #6 . . . 453
31.6.1 Global variables . . . 455

II Important fundamentals **458**

32 Signed number representations **459**
32.1 Integer overflow . . . 460

33 Endianness **461**
33.1 Big-endian . . . 461
33.2 Little-endian . . . 461
33.3 Bi-endian . . . 462
33.4 Converting data . . . 462

III Finding important/interesting stuff in the code **463**

34 Identification of executable files **465**
34.1 Microsoft Visual C++ . . . 465
34.1.1 Name mangling . . . 465
34.2 GCC . . . 466
34.2.1 Name mangling . . . 466
34.2.2 Cygwin . . . 466
34.2.3 MinGW . . . 466
34.3 Intel FORTRAN . . . 466
34.4 Watcom, OpenWatcom . . . 466
34.4.1 Name mangling . . . 466
34.5 Borland . . . 466
34.5.1 Delphi . . . 467
34.6 Other known DLLs . . . 470

35 Communication with the outer world (win32) 471
35.1 Often used functions in Windows API . . . 472
35.2 tracer: Intercepting all functions in specific module . . . 473

36 Strings 475
36.1 Text strings . . . 475
36.1.1 Unicode . . . 476
36.2 Error/debug messages . . . 480

37 Calls to assert() 481

38 Constants 483
38.1 Magic numbers . . . 484
38.1.1 DHCP . . . 484
38.2 Constant searching . . . 485

39 Finding the right instructions 486

40 Suspicious code patterns 489
40.1 XOR instructions . . . 489
40.2 Hand-written assembly code . . . 489

41 Using magic numbers while tracing 491

42 Other things 493

43 Old-school techniques, nevertheless, interesting to know 494
43.1 Memory "snapshots" comparing . . . 494
43.1.1 Windows registry . . . 495

IV OS-specific 496

44 Arguments passing methods (calling conventions) 497
44.1 cdecl . . . 497
44.2 stdcall . . . 497
44.2.1 Variable arguments number functions . . . 499
44.3 fastcall . . . 499
44.3.1 GCC regparm . . . 500
44.3.2 Watcom/OpenWatcom . . . 501
44.4 thiscall . . . 501
44.5 x86-64 . . . 501
44.5.1 Windows x64 . . . 501
44.5.2 Linux x64 . . . 504
44.6 Returning values of *float* and *double* type . . . 505

44.7 Modifying arguments . . . 506

45 Thread Local Storage **507**

46 System calls (syscall-s) **508**
46.1 Linux . . . 508
46.2 Windows . . . 509

47 Linux **510**
47.1 Position-independent code . . . 510
47.1.1 Windows . . . 514
47.2 *LD_PRELOAD* hack in Linux . . . 514

48 Windows NT **518**
48.1 CRT (win32) . . . 518
48.2 Win32 PE . . . 523
48.2.1 Terminology . . . 523
48.2.2 Base address . . . 524
48.2.3 Subsystem . . . 525
48.2.4 OS version . . . 525
48.2.5 Sections . . . 525
48.2.6 Relocations (relocs) . . . 527
48.2.7 Exports and imports . . . 527
48.2.8 Resources . . . 531
48.2.9 .NET . . . 531
48.2.10 TLS . . . 531
48.2.11 Tools . . . 532
48.2.12 Further reading . . . 532
48.3 Windows SEH . . . 532
48.3.1 Let's forget about MSVC . . . 532
48.3.2 Now let's get back to MSVC . . . 539
48.3.3 Windows x64 . . . 561
48.3.4 Read more about SEH . . . 566
48.4 Windows NT: Critical section . . . 567

V Tools **570**

49 Disassembler **571**
49.1 IDA . . . 571

50 Debugger **572**
50.1 tracer . . . 572
50.2 OllyDbg . . . 572
50.3 GDB . . . 572

51 System calls tracing **573**
51.0.1 strace / dtruss 573

52 Decompilers **574**

53 Other tools **575**

VI More examples **576**

54 Hand decompiling + using Z3 SMT solver for defeating amateur cryptography **577**
54.1 Hand decompiling 578
54.2 Now let's use Z3 SMT solver 583

55 Dongles **590**
55.1 Example #1: MacOS Classic and PowerPC 590
55.2 Example #2: SCO OpenServer 601
55.2.1 Decrypting error messages 613
55.3 Example #3: MS-DOS 617

56 "QR9": Rubik's cube inspired amateur crypto-algorithm **626**

57 SAP **673**
57.1 About SAP client network traffic compression 673
57.2 SAP 6.0 password checking functions 690

58 Oracle RDBMS **697**
58.1 `V$VERSION` table in the Oracle RDBMS 697
58.2 `X$KSMLRU` table in Oracle RDBMS 710
58.3 `V$TIMER` table in Oracle RDBMS 713

59 Handwritten assembly code **719**
59.1 EICAR test file 719

60 Demos **721**
60.1 10 PRINT CHR$(205.5+RND(1)); : GOTO 10 721
60.1.1 Trixter's 42 byte version 722
60.1.2 My attempt to reduce Trixter's version: 27 bytes 723
60.1.3 Take a random memory garbage as a source of randomness . . 724
60.1.4 Conclusion 725

VII Other things 726

61 npad 727

62 Executable files patching 730
62.1 Text strings . . . 730
62.2 x86 code . . . 730

63 Compiler intrinsic 732

64 Compiler's anomalies 733

65 OpenMP 735
65.1 MSVC . . . 738
65.2 GCC . . . 741

66 Itanium 744

67 8086 memory model 749

68 Basic blocks reordering 751
68.1 Profile-guided optimization . . . 751

VIII Books/blogs worth reading 754

69 Books 755
69.1 Windows . . . 755
69.2 C/C++ . . . 755
69.3 x86 / x86-64 . . . 755
69.4 ARM . . . 755

70 Blogs 756
70.1 Windows . . . 756

71 Other 757

IX Exercises 758

72 Level 1 760
72.1 Exercise 1.1 . . . 760
72.1.1 MSVC 2012 x64 + /Ox . . . 760
72.1.2 Keil (ARM) . . . 760
72.1.3 Keil (thumb) . . . 760
72.2 Exercise 1.2 . . . 761

72.3 Exercise 1.3 . . . 761
72.4 Exercise 1.4 . . . 761

73 Level 2 762
73.1 Exercise 2.1 . . . 762
73.1.1 MSVC 2010 . . . 762
73.1.2 GCC 4.4.1 + -O3 . . . 763
73.1.3 Keil (ARM) + -O3 . . . 763
73.1.4 Keil (thumb) + -O3 . . . 763
73.2 Exercise 2.2 . . . 764
73.2.1 MSVC 2010 + /Ox . . . 764
73.2.2 GCC 4.4.1 . . . 765
73.2.3 Keil (ARM) + -O3 . . . 767
73.2.4 Keil (thumb) + -O3 . . . 768
73.3 Exercise 2.3 . . . 769
73.3.1 MSVC 2010 + /Ox . . . 769
73.3.2 GCC 4.4.1 . . . 770
73.3.3 Keil (ARM) + -O3 . . . 771
73.3.4 Keil (thumb) + -O3 . . . 772
73.4 Exercise 2.4 . . . 772
73.4.1 MSVC 2010 + /Ox . . . 772
73.4.2 GCC 4.4.1 . . . 774
73.4.3 Keil (ARM) + -O3 . . . 775
73.4.4 Keil (thumb) + -O3 . . . 776
73.5 Exercise 2.5 . . . 777
73.5.1 MSVC 2010 + /Ox . . . 777
73.6 Exercise 2.6 . . . 778
73.6.1 MSVC 2010 + /Ox . . . 778
73.6.2 Keil (ARM) + -O3 . . . 780
73.6.3 Keil (thumb) + -O3 . . . 781
73.7 Exercise 2.7 . . . 782
73.7.1 MSVC 2010 + /Ox . . . 782
73.7.2 Keil (ARM) + -O3 . . . 783
73.7.3 Keil (thumb) + -O3 . . . 785
73.8 Exercise 2.8 . . . 788
73.8.1 MSVC 2010 + /O1 . . . 788
73.8.2 Keil (ARM) + -O3 . . . 788
73.8.3 Keil (thumb) + -O3 . . . 789
73.9 Exercise 2.9 . . . 790
73.9.1 MSVC 2010 + /O1 . . . 790
73.9.2 Keil (ARM) + -O3 . . . 791
73.9.3 Keil (thumb) + -O3 . . . 792
73.10 Exercise 2.10 . . . 794
73.11 Exercise 2.11 . . . 794

73.12 Exercise 2.12 . . . 795
73.12.1 MSVC 2012 x64 + /Ox . . . 795
73.12.2 Keil (ARM) . . . 796
73.12.3 Keil (thumb) . . . 797
73.13 Exercise 2.13 . . . 798
73.13.1 MSVC 2012 + /Ox . . . 798
73.13.2 Keil (ARM) . . . 799
73.13.3 Keil (thumb) . . . 799
73.14 Exercise 2.14 . . . 800
73.14.1 MSVC 2012 . . . 800
73.14.2 Keil (ARM mode) . . . 801
73.14.3 GCC 4.6.3 for Raspberry Pi (ARM mode) . . . 802
73.15 Exercise 2.15 . . . 803
73.15.1 MSVC 2012 x64 /Ox . . . 803
73.15.2 GCC 4.4.6 -O3 x64 . . . 808
73.15.3 GCC 4.8.1 -O3 x86 . . . 809
73.15.4 Keil (ARM mode): Cortex-R4F CPU as target . . . 811
73.16 Exercise 2.16 . . . 812
73.16.1 MSVC 2012 x64 /Ox . . . 812
73.16.2 Keil (ARM) -O3 . . . 813
73.16.3 Keil (thumb) -O3 . . . 813
73.17 Exercise 2.17 . . . 814
73.18 Exercise 2.18 . . . 814
73.19 Exercise 2.19 . . . 815

74 Level 3 **816**
74.1 Exercise 3.1 . . . 816
74.2 Exercise 3.2 . . . 825
74.3 Exercise 3.3 . . . 826
74.4 Exercise 3.4 . . . 826
74.5 Exercise 3.5 . . . 826
74.6 Exercise 3.6 . . . 827
74.7 Exercise 3.7 . . . 827
74.8 Exercise 3.8 . . . 827

75 crackme / keygenme **829**

Afterword **831**

76 Questions? **831**

Appendix **833**

A Common terminology **833**

B x86 **834**
B.1 Terminology . . . 834
B.2 General purpose registers . . . 834
B.2.1 RAX/EAX/AX/AL . . . 835
B.2.2 RBX/EBX/BX/BL . . . 835
B.2.3 RCX/ECX/CX/CL . . . 835
B.2.4 RDX/EDX/DX/DL . . . 835
B.2.5 RSI/ESI/SI/SIL . . . 836
B.2.6 RDI/EDI/DI/DIL . . . 836
B.2.7 R8/R8D/R8W/R8L . . . 836
B.2.8 R9/R9D/R9W/R9L . . . 836
B.2.9 R10/R10D/R10W/R10L . . . 836
B.2.10 R11/R11D/R11W/R11L . . . 837
B.2.11 R12/R12D/R12W/R12L . . . 837
B.2.12 R13/R13D/R13W/R13L . . . 837
B.2.13 R14/R14D/R14W/R14L . . . 837
B.2.14 R15/R15D/R15W/R15L . . . 837
B.2.15 RSP/ESP/SP/SPL . . . 838
B.2.16 RBP/EBP/BP/BPL . . . 838
B.2.17 RIP/EIP/IP . . . 838
B.2.18 CS/DS/ES/SS/FS/GS . . . 839
B.2.19 Flags register . . . 839
B.3 FPU-registers . . . 840
B.3.1 Control Word . . . 840
B.3.2 Status Word . . . 840
B.3.3 Tag Word . . . 841
B.4 SIMD-registers . . . 842
B.4.1 MMX-registers . . . 842
B.4.2 SSE and AVX-registers . . . 842
B.5 Debugging registers . . . 842
B.5.1 DR6 . . . 842
B.5.2 DR7 . . . 843
B.6 Instructions . . . 844
B.6.1 Prefixes . . . 844
B.6.2 Most frequently used instructions . . . 844
B.6.3 Less frequently used instructions . . . 851
B.6.4 FPU instructions . . . 858
B.6.5 SIMD instructions . . . 860
B.6.6 Instructions having printable ASCII opcode . . . 860

C ARM **863**
C.1 Terminology . . . 863
C.2 General purpose registers . . . 863
C.3 Current Program Status Register (CPSR) . . . 864
C.4 VFP (floating point) and NEON registers . . . 864

D Some GCC library functions **866**

E Some MSVC library functions **867**

F Cheatsheets **868**
F.1 IDA . . . 868
F.2 OllyDbg . . . 869
F.3 MSVC . . . 870
F.4 GCC . . . 870
F.5 GDB . . . 870

G Exercise solutions **872**
G.1 Level 1 . . . 872
G.1.1 Exercise 1.1 . . . 872
G.1.2 Exercise 1.4 . . . 872
G.2 Level 2 . . . 872
G.2.1 Exercise 2.1 . . . 872
G.2.2 Exercise 2.2 . . . 873
G.2.3 Exercise 2.3 . . . 873
G.2.4 Exercise 2.4 . . . 874
G.2.5 Exercise 2.5 . . . 874
G.2.6 Exercise 2.6 . . . 875
G.2.7 Exercise 2.7 . . . 876
G.2.8 Exercise 2.8 . . . 877
G.2.9 Exercise 2.9 . . . 878
G.2.10 Exercise 2.11 . . . 878
G.2.11 Exercise 2.12 . . . 878
G.2.12 Exercise 2.13 . . . 878
G.2.13 Exercise 2.14 . . . 879
G.2.14 Exercise 2.15 . . . 879
G.2.15 Exercise 2.16 . . . 879
G.2.16 Exercise 2.17 . . . 879
G.2.17 Exercise 2.18 . . . 879
G.2.18 Exercise 2.19 . . . 879
G.3 Level 3 . . . 880
G.3.1 Exercise 3.1 . . . 880
G.3.2 Exercise 3.2 . . . 880
G.3.3 Exercise 3.3 . . . 880

G.3.4 Exercise 3.4 880
G.3.5 Exercise 3.5 880
G.3.6 Exercise 3.6 880
G.3.7 Exercise 3.8 880

Acronyms used **882**

Bibliography **887**

Glossary **890**

Index **893**

0.1 Preface

Here are some of my notes about reverse engineering in English language for those beginners who would like to learn to understand x86 (which accounts for almost all executable software in the world) and ARM code created by C/C++ compilers.

There are several popular meanings of the term "reverse engineering": 1) reverse engineering of software: researching of compiled programs; 2) 3D model scanning and reworking in order to make a copy of it; 3) recreating DBMS[3] structure. These notes are related to the first meaning.

Topics discussed

x86, ARM.

Topics touched

Oracle RDBMS (58), Itanium (66), copy-protection dongles (55), LD_PRELOAD (47.2), stack overflow, ELF[4], win32 PE file format (48.2), x86-64 (23.1), critical sections (48.4), syscalls (46), TLS[5], position-independent code (PIC[6]) (47.1), profile-guided optimization (68.1), C++ STL (29.4), OpenMP (65), SEH ().

Mini-FAQ[7]

- Q: Should one learn to understand assembly language these days?
  A: Yes: in order to have deeper understanding of the internals and to debug your software better and faster.
- Q: Should one learn to write in assembly language these days?
  A: Unless one writes low-level OS[8] code, probably no.
- Q: But what about writing highly optimized routines?
  A: No, modern C/C++ compilers do this job better.
- Q: Should I learn microprocessor internals?
  A: Modern CPU[9]-s are very complex. If you do not plan to write highly optimized code or if you do not work on compiler's code generator then you may still learn internals in bare outlines. [10]. At the same time, in order to understand and

[3] Database management systems
[4] Executable file format widely used in *NIX system including Linux
[5] Thread Local Storage
[6] Position Independent Code: 47.1
[7] Frequently Asked Questions
[8] Operating System
[9] Central processing unit
[10] Very good text about it: [10]

analyze compiled code it is enough to know only ISA[11], register's descriptions, i.e., the "outside" part of a CPU that is available to an application programmer.

- Q: So why should I learn assembly language anyway?
  A: Mostly to better understand what is going on while debugging and for reverse engineering without source code, including, but not limited to, malware.
- Q: How would I search for a reverse engineering job?
  A: There are hiring threads that appear from time to time on reddit devoted to RE[12] (2013 Q3, 2014). Try to take a look there.

About the author

Dennis Yurichev is an experienced reverse engineer and programmer. His CV is available on his website[13].

Thanks

Andrey "herm1t" Baranovich, Slava "Avid" Kazakov, Stanislav "Beaver" Bobrytskyy, Alexander Lysenko, Alexander "Lstar" Chernenkiy, Andrew Zubinski, Vladimir Botov, Mark "Logxen" Cooper, Shell Rocket, Yuan Jochen Kang, Arnaud Patard (rtp on #debian-arm IRC), and all the folks on github.com who have contributed notes and corrections.

A lot of LaTeX packages were used: I would thank their authors as well.

Praise for *Reverse Engineering for Beginners*

- "It's very well done .. and for free .. amazing."[14] Daniel Bilar, Siege Technologies, LLC.

[11] Instruction Set Architecture
[12] http://www.reddit.com/r/ReverseEngineering/
[13] http://yurichev.com/Dennis_Yurichev.pdf
[14] https://twitter.com/daniel_bilar/status/436578617221742593

- "...excellent and free"[15] Pete Finnigan, Oracle RDBMS security guru.
- "... book is interesting, great job!" Michael Sikorski, author of *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*.
- "... my compliments for the very nice tutorial!" Herbert Bos, full professor at the Vrije Universiteit Amsterdam.
- "... It is amazing and unbelievable." Luis Rocha, CISSP / ISSAP, Technical Manager, Network & Information Security at Verizon Business.
- "Thanks for the great work and your book." Joris van de Vis, SAP Netweaver & Security specialist.

Donate

As it turns out, (technical) writing takes a lot of effort and work.

This book is free, available freely and available in source code form [16] (LaTeX), and it will be so forever.

My current plan for this book is to add lots of information about: PLANS[17].

If you want me to continue writing on all these topics you may consider donating.

I worked more than year on this book [18], there are more than 500 pages. There are ≈ 300 TEX-files, ≈ 90 C/C++ source codes, ≈ 350 various listings.

Price of other books on the same subject varies between $20 and $50 on amazon.com.

Ways to donate are available on the page: http://yurichev.com/donate.html

Every donor's name will be included in the book! Donors also have a right to ask me to rearrange items in my writing plan.

Why not try to publish? Because it's technical literature which, as I believe, cannot be finished or frozen in paper state. Such technical references akin to Wikipedia or MSDN[19] library. They can evolve and grow indefinitely. Someone can sit down and write everything from the begin to the end, publish it and forget about it. As it turns out, it's not me. I have everyday thoughts like "that was written badly and can be rewritten better", "that was a bad example, I know a better one", "that is also a thing I can explain better and shorter", etc. As you may see in commit history of this book's source code, I make a lot of small changes almost every day: https://github.com/dennis714/RE-for-beginners/commits/master.

So the book will probably be a "rolling release" as they say about Linux distros like Gentoo. No fixed releases (and deadlines) at all, but continuous development. I

[15] https://twitter.com/petefinnigan/status/400551705797869568

[16] https://github.com/dennis714/RE-for-beginners

[17] https://github.com/dennis714/RE-for-beginners/blob/master/PLANS

[18] Initial git commit from March 2013:
https://github.com/dennis714/RE-for-beginners/tree/1e57ef540d827c7f7a92fcb3a4626af3e13c7ee4

[19] Microsoft Developer Network

don't know how long it will take to write all I know. Maybe 10 years or more. Of course, it is not very convenient for readers who want something stable, but all I can offer is a ChangeLog [20] file serving as a "what's new" section. Those who are interested may check it from time to time, or my blog/twitter [21] .

Donors

6 * anonymous, 2 * Oleg Vygovsky, Daniel Bilar, James Truscott, Luis Rocha, Joris van de Vis, Richard S Shultz, Jang Minchang, Shade Atlas, Yao Xiao.

About illustrations

Those readers who are used to read a lot in the Internet, expects seeing illustrations at the places where they should be. It's because there are no pages at all, only single one. It's not possible to place illustrations in the book at the suitable context. So, in this book, illustrations can be at the end of section, and a referenceses in the text may be present, like "fig.1.1".

This is A5-format version for e-book readers. Although, it mostly the same, but illustrations are resized and probably not readable. Sorry for this! You can always view them in A4-format version here: http://yurichev.com/RE-book.html.

[20] https://github.com/dennis714/RE-for-beginners/blob/master/ChangeLog
[21] http://blog.yurichev.com/ https://twitter.com/yurichev

Part I

Code patterns

When I first learned C and then C++, I wrote small pieces of code, compiled them, and saw what was produced in the assembly language. This was easy for me. I did it many times and the relation between the C/C++ code and what the compiler produced was imprinted in my mind so deep that I can quickly understand what was in the original C code when I look at produced x86 code. Perhaps this technique may be helpful for someone else so I will try to describe some examples here.

Chapter 1

Short introduction to the CPU

The CPU is the unit which executes all of the programs.
Short glossary:

Instruction : a primitive command to the CPU. Simplest examples: moving data between registers, working with memory, arithmetic primitives. As a rule, each CPU has its own instruction set architecture (ISA).

Machine code : code for the CPU. Each instruction is usually encoded by several bytes.

Assembly language : mnemonic code and some extensions like macros which are intended to make a programmer's life easier.

CPU register : Each CPU has a fixed set of general purpose registers (GPR[1]). ≈ 8 in x86, ≈ 16 in x86-64, ≈ 16 in ARM. The easiest way to understand a register is to think of it as an untyped temporary variable. Imagine you are working with a high-level PL[2] and you have only 8 32-bit variables. A lot of things can be done using only these!

What is the difference between machine code and a PL? It is much easier for humans to use a high-level PL like C/C++, Java, Python, etc., but it is easier for a CPU to use a much lower level of abstraction. Perhaps, it would be possible to invent a CPU which can execute high-level PL code, but it would be much more complex. On the contrary, it is very inconvenient for humans to use assembly language due to its low-levelness. Besides, it is very hard to do it without making a huge amount of annoying mistakes. The program which converts high-level PL code into assembly is called a *compiler*.

[1] General Purpose Registers
[2] Programming language

Chapter 2

Hello, world!

Let's start with the famous example from the book "The C programming Language" [17]:

```
#include <stdio.h>

int main()
{
    printf("hello, world");
    return 0;
};
```

2.1 x86

2.1.1 MSVC—x86

Let's compile it in MSVC 2010:

```
cl 1.cpp /Fa1.asm
```

(/Fa option means generate assembly listing file)

Listing 2.1: MSVC 2010

```
CONST   SEGMENT
$SG3830 DB      'hello, world', 00H
CONST   ENDS
PUBLIC  _main
EXTRN   _printf:PROC
; Function compile flags: /Odtp
_TEXT   SEGMENT
```

```
_main   PROC
        push    ebp
        mov     ebp, esp
        push    OFFSET $SG3830
        call    _printf
        add     esp, 4
        xor     eax, eax
        pop     ebp
        ret     0
_main   ENDP
_TEXT   ENDS
```

MSVC produces assembly listings in Intel-syntax. The difference between Intel-syntax and AT&T-syntax will be discussed hereafter.

The compiler generated `1.obj` file will be linked into `1.exe`.

In our case, the file contain two segments: `CONST` (for data constants) and `_TEXT` (for code).

The string ‘`hello, world`’ in C/C++ has type `const char*`, however it does not have its own name.

The compiler needs to deal with the string somehow so it defines the internal name `$SG3830` for it.

So the example may be rewritten as:

```
#include <stdio.h>

const char *$SG3830="hello, world";

int main()
{
    printf($SG3830);
    return 0;
};
```

Let’s back to the assembly listing. As we can see, the string is terminated by a zero byte which is standard for C/C++ strings. More about C strings: 36.1.

In the code segment, `_TEXT`, there is only one function so far: `main()`.

The function `main()` starts with prologue code and ends with epilogue code (like almost any function) [1].

After the function prologue we see the call to the `printf()` function: `CALL _printf`.

Before the call the string address (or a pointer to it) containing our greeting is placed on the stack with the help of the `PUSH` instruction.

When the `printf()` function returns flow control to the `main()` function, string address (or pointer to it) is still in stack.

[1] Read more about it in section about function prolog and epilog (3).

Since we do not need it anymore the stack pointer (the ESP register) needs to be corrected.

ADD ESP, 4 means add 4 to the value in the ESP register.

Why 4? Since it is 32-bit code we need exactly 4 bytes for address passing through the stack. It is 8 bytes in x64-code.

''ADD ESP, 4'' is effectively equivalent to ''POP register'' but without using any register[2].

Some compilers (like Intel C++ Compiler) in the same situation may emit POP ECX instead of ADD (e.g. such a pattern can be observed in the Oracle RDBMS code as it is compiled by Intel C++ compiler). This instruction has almost the same effect but the ECX register contents will be rewritten.

The Intel C++ compiler probably uses POP ECX since this instruction's opcode is shorter then ADD ESP, x (1 byte against 3).

Read more about the stack in section (4).

After the call to printf(), in the original C/C++ code was return 0 —return 0 as the result of the main() function.

In the generated code this is implemented by instruction XOR EAX, EAX

XOR is in fact, just "eXclusive OR" [3] but compilers often use it instead of MOV EAX, 0 —again because it is a slightly shorter opcode (2 bytes against 5).

Some compilers emit SUB EAX, EAX, which means *SUBtract the value in the EAX from the value in EAX*, which in any case will result zero.

The last instruction RET returns control flow to the caller. Usually, it is C/C++ CRT[4] code which in turn returns control to the OS.

2.1.2 GCC—x86

Now let's try to compile the same C/C++ code in the GCC 4.4.1 compiler in Linux: gcc 1.c -o 1

Next, with the assistance of the IDA[5] disassembler, let's see how the main() function was created.

(IDA, like MSVC, shows code in Intel-syntax).

N.B. We could also have GCC produce assembly listings in Intel-syntax by applying the options -S -masm=intel

Listing 2.2: GCC

```
main            proc near

var_10          = dword ptr -10h

                push    ebp
```

[2] CPU flags, however, are modified

[3] http://en.wikipedia.org/wiki/Exclusive_or

[4] C runtime library: sec:CRT

[5] Interactive Disassembler

```
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 10h
                mov     eax, offset aHelloWorld ; "hello, world"
                mov     [esp+10h+var_10], eax
                call    _printf
                mov     eax, 0
                leave
                retn
main            endp
```

The result is almost the same. The address of the "hello, world" string (stored in the data segment) is saved in the EAX register first and then it is stored on the stack. Also in the function prologue we see AND ESP, 0FFFFFFF0h —this instruction aligns the value in the ESP register on a 16-byte boundary. This results in all values in the stack being aligned. (The CPU performs better if the values it is dealing with are located in memory at addresses aligned on a 4- or 16-byte boundary)[6].

SUB ESP, 10h allocates 16 bytes on the stack. Although, as we can see hereafter, only 4 are necessary here.

This is because the size of the allocated stack is also aligned on a 16-byte boundary.

The string address (or a pointer to the string) is then written directly onto the stack space without using the PUSH instruction. *var_10* —is a local variable and is also an argument for printf(). Read about it below.

Then the printf() function is called.

Unlike MSVC, when GCC is compiling without optimization turned on, it emits MOV EAX, 0 instead of a shorter opcode.

The last instruction, LEAVE —is the equivalent of the MOV ESP, EBP and POP EBP instruction pair —in other words, this instruction sets the stack pointer (ESP) back and restores the EBP register to its initial state.

This is necessary since we modified these register values (ESP and EBP) at the beginning of the function (executing MOV EBP, ESP / AND ESP, ...).

2.1.3 GCC: AT&T syntax

Let's see how this can be represented in the AT&T syntax of assembly language. This syntax is much more popular in the UNIX-world.

Listing 2.3: let's compile in GCC 4.7.3

```
gcc -S 1_1.c
```

We get this:

[6] Wikipedia: Data structure alignment

Listing 2.4: GCC 4.7.3

```
        .file   "1_1.c"
        .section        .rodata
.LC0:
        .string "hello, world"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushl   %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp
        .cfi_def_cfa_register 5
        andl    $-16, %esp
        subl    $16, %esp
        movl    $.LC0, (%esp)
        call    printf
        movl    $0, %eax
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu/Linaro 4.7.3-1ubuntu1) 4.7.3"
        .section        .note.GNU-stack,"",@progbits
```

There are a lot of macros (beginning with dot). These are not very interesting to us so far. For now, for the sake of simplification, we can ignore them (except the *.string* macro which encodes a null-terminated character sequence just like a C-string). Then we'll see this [7]:

Listing 2.5: GCC 4.7.3

```
.LC0:
        .string "hello, world"
main:
        pushl   %ebp
        movl    %esp, %ebp
```

[7] This GCC option can be used to eliminate "unnecessary" macros: *-fno-asynchronous-unwind-tables*

```
        andl    $-16, %esp
        subl    $16, %esp
        movl    $.LC0, (%esp)
        call    printf
        movl    $0, %eax
        leave
        ret
```

Some of the major differences between Intel and AT&T syntax are:

- Operands are written backwards.

  In Intel-syntax: <instruction> <destination operand> <source operand>.

  In AT&T syntax: <instruction> <source operand> <destination operand>.

  Here is a way to think about them: when you deal with Intel-syntax, you can put in equality sign (=) in your mind between operands and when you deal with AT&T-syntax put in a right arrow (→) [8].

- AT&T: Before register names a percent sign must be written (%) and before numbers a dollar sign ($). Parentheses are used instead of brackets.

- AT&T: A special symbol is to be added to each instruction defining the type of data:

  - l — long (32 bits)
  - w — word (16 bits)
  - b — byte (8 bits)

Let's go back to the compiled result: it is identical to what we saw in IDA. With one subtle difference: `0FFFFFFF0h` is written as `$-16`. It is the same: 16 in the decimal system is `0x10` in hexadecimal. `-0x10` is equal to `0xFFFFFFF0` (for a 32-bit data type).

One more thing: the return value is to be set to 0 by using usual `MOV`, not `XOR`. `MOV` just loads value to a register. Its name is not intuitive (data is not moved). In other architectures, this instruction has the name "load" or something like that.

2.2 x86-64

2.2.1 MSVC—x86-64

Let's also try 64-bit MSVC:

[8] By the way, in some C standard functions (e.g., memcpy(), strcpy()) arguments are listed in the same way as in Intel-syntax: pointer to destination memory block at the beginning and then pointer to source memory block.

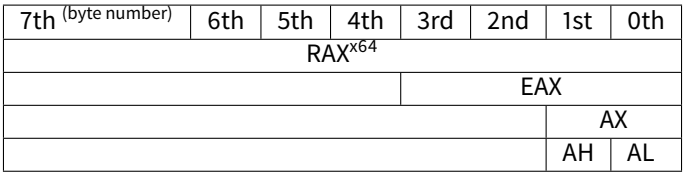
Listing 2.6: MSVC 2012 x64

```
$SG2989 DB      'hello, world', 00H

main    PROC
        sub     rsp, 40
        lea     rcx, OFFSET FLAT:$SG2923
        call    printf
        xor     eax, eax
        add     rsp, 40
        ret     0
main    ENDP
```

As of x86-64, all registers were extended to 64-bit and now have a R- prefix. In order to use the stack less often (in other words, to access external memory less often), there exists a popular way to pass function arguments via registers (fastcall: 44.3). I.e., one part of function arguments are passed in registers, other part—via stack. In Win64, 4 function arguments are passed in RCX, RDX, R8, R9 registers. That is what we see here: a pointer to the string for `printf()` is now passed not in stack, but in the RCX register.

Pointers are 64-bit now, so they are passed in the 64-bit part of registers (which have the R- prefix). But for backward compatibility, it is still possible to access 32-bit parts, using the E- prefix.

This is how RAX/EAX/AX/AL looks like in 64-bit x86-compatible CPUs:



The `main()` function returns an *int*-typed value, which is, in the C PL, for better backward compatibility and portability, still 32-bit, so that is why the EAX register is cleared at the function end (i.e., 32-bit part of register) instead of RAX.

2.2.2 GCC—x86-64

Let's also try GCC in 64-bit Linux:

Listing 2.7: GCC 4.4.6 x64

```
.string "hello, world"
main:
        sub     rsp, 8
        mov     edi, OFFSET FLAT:.LC0 ; "hello, world"
        xor     eax, eax  ; number of vector registers passed
```

```
        call    printf
        xor     eax, eax
        add     rsp, 8
        ret
```

A method to pass function arguments in registers is also used in Linux, *BSD and Mac OS X [21]. The first 6 arguments are passed in the RDI, RSI, RDX, RCX, R8, R9 registers, and others—via stack.

So the pointer to the string is passed in EDI (32-bit part of register). But why not use the 64-bit part, RDI?

It is important to keep in mind that all MOV instructions in 64-bit mode writing something into the lower 32-bit register part, also clear the higher 32-bits [14]. I.e., the MOV EAX, 011223344h will write a value correctly into RAX, since the higher bits will be cleared.

If we open the compiled object file (.o), we will also see all instruction's opcodes [9]:

Listing 2.8: GCC 4.4.6 x64

```
.text:00000000004004D0                  main  proc near
.text:00000000004004D0 48 83 EC 08      sub     rsp, 8
.text:00000000004004D4 BF E8 05 40 00   mov     edi, offset ↙
    ↳ format ; "hello, world"
.text:00000000004004D9 31 C0            xor     eax, eax
.text:00000000004004DB E8 D8 FE FF FF   call    _printf
.text:00000000004004E0 31 C0            xor     eax, eax
.text:00000000004004E2 48 83 C4 08      add     rsp, 8
.text:00000000004004E6 C3               retn
.text:00000000004004E6                  main  endp
```

As we can see, the instruction writing into EDI at 0x4004D4 occupies 5 bytes. The same instruction writing a 64-bit value into RDI will occupy 7 bytes. Apparently, GCC is trying to save some space. Besides, it can be sure that the data segment containing the string will not be allocated at the addresses higher than 4GiB.

We also see EAX register clearance before printf() function call. This is done because a number of used vector registers is passed in EAX by standard: "with variable arguments passes information about the number of vector registers used" [21].

2.3 ARM

For my experiments with ARM processors I chose two compilers: popular in the embedded area Keil Release 6/2013 and Apple Xcode 4.6.3 IDE (with LLVM-GCC 4.2 com-

[9] This should be enabled in Options → Disassembly → Number of opcode bytes

piler), which produces code for ARM-compatible processors and SOC[10] in iPod/iPhone/iPad, Windows 8 and Window RT tables[11] and also such devices as Raspberry Pi.

32-bit ARM code is used in all cases in this book, if not mentioned otherwise.

2.3.1 Non-optimizing Keil + ARM mode

Let's start by compiling our example in Keil:

```
armcc.exe --arm --c90 -O0 1.c
```

The *armcc* compiler produces assembly listings in Intel-syntax but it has high-level ARM-processor related macros[12], but it is more important for us to see the instructions "as is" so let's see the compiled result in IDA.

Listing 2.9: Non-optimizing Keil + ARM mode + IDA

```
.text:00000000                 main
.text:00000000 10 40 2D E9     STMFD   SP!, {R4,LR}
.text:00000004 1E 0E 8F E2     ADR     R0, aHelloWorld ; "hello, ↙
    ↳ world"
.text:00000008 15 19 00 EB     BL      __2printf
.text:0000000C 00 00 A0 E3     MOV     R0, #0
.text:00000010 10 80 BD E8     LDMFD   SP!, {R4,PC}

.text:000001EC 68 65 6C 6C+aHelloWorld  DCB "hello, world",0    ↙
    ↳ ; DATA XREF: main+4
```

Here are a couple of ARM-related facts that we should know in order to proceed. An ARM processor has at least two major modes: ARM mode and thumb mode. In the first (ARM) mode, all instructions are enabled and each is 32 bits (4 bytes) in size. In the second (thumb) mode each instruction is 16 bits (2 bytes) in size [13]. Thumb mode may look attractive because programs that use it may 1) be compact and 2) execute faster on microcontrollers having a 16-bit memory datapath. Nothing comes for free. In thumb mode, there is a reduced instruction set, only 8 registers are accessible and one needs several thumb instructions for doing some operations when you only need one in ARM mode.

Starting from ARMv7 the thumb-2 instruction set is also available. This is an extended thumb mode that supports a much larger instruction set. There is a common misconception that thumb-2 is a mix of ARM and thumb. This is not correct. Rather, thumb-2 was extended to fully support processor features so it could compete with ARM mode. A program for the ARM processor may be a mix of procedures compiled

[10] System on Chip

[11] http://en.wikipedia.org/wiki/List_of_Windows_8_and_RT_tablet_devices

[12] e.g. ARM mode lacks PUSH/POP instructions

[13] By the way, fixed-length instructions are handy in a way that one can calculate the next (or previous) instruction's address without effort. This feature will be discussed in switch() (11.2.2) section.

for both modes. The majority of iPod/iPhone/iPad applications are compiled for the thumb-2 instruction set because Xcode does this by default.

In the example we can easily see each instruction has a size of 4 bytes. Indeed, we compiled our code for ARM mode, not for thumb.

The very first instruction, ‘‘STMFD SP!, {R4,LR}’’[14], works as an x86 PUSH instruction, writing the values of two registers (R4 and LR[15]) into the stack. Indeed, in the output listing from the *armcc* compiler, for the sake of simplification, actually shows the ‘‘PUSH {r4,lr}’’ instruction. But it is not quite correct. The PUSH instruction is only available in thumb mode. So, to make things less messy, that is why I suggested working in IDA.

This instruction first decrements SP[16] so it will point to the place in the stack that is free for new entries, then it writes the values of the R4 and LR registers at the address in changed SP.

This instruction (like the PUSH instruction in thumb mode) is able to save several register values at once and this may be useful. By the way, there is no such thing in x86. It can also be noted that the STMFD instruction is a generalization of the PUSH instruction (extending its features), since it can work with any register, not just with SP, and this can be very useful.

The ‘‘ADR R0, aHelloWorld’’ instruction adds the value in the PC[17] register to the offset where the *“hello, world”* string is located. How is the PC register used here, one might ask? This is so-called “position-independent code”. [18] It is intended to be executed at a non-fixed address in memory. In the opcode of the ADR instruction, the difference between the address of this instruction and the place where the string is located is encoded. The difference will always be the same, independent of the address where the code is loaded by the OS. That’s why all we need is to add the address of the current instruction (from PC) in order to get the absolute address of our C-string in memory.

‘‘BL __2printf’’[19] instruction calls the printf() function. Here’s how this instruction works:

- write the address following the BL instruction (0xC) into the LR;
- then pass control flow into printf() by writing its address into the PC register.

When printf() finishes its work it must have information about where it must return control. That’s why each function passes control to the address stored in the LR register.

[14] Store Multiple Full Descending
[15] Link Register
[16] stack pointer. SP/ESP/RSP in x86/x64. SP in ARM.
[17] Program Counter. IP/EIP/RIP in x86/64. PC in ARM.
[18] Read more about it in relevant section (47.1)
[19] Branch with Link

That is the difference between "pure" RISC[20]-processors like ARM and CISC[21]-processors like x86, where the return address is stored on the stack[22].

By the way, an absolute 32-bit address or offset cannot be encoded in the 32-bit BL instruction because it only has space for 24 bits. It is also worth noting all ARM-mode instructions have a size of 4 bytes (32 bits). Hence they can only be located on 4-byte boundary addresses. This means the the last 2 bits of the instruction address (which are always zero bits) may be omitted. In summary, we have 26 bits for offset encoding. This is enough to represent offset $\pm \approx 32M$.

Next, the "MOV R0, #0"[23] instruction just writes 0 into the R0 register. That's because our C-function returns 0 and the return value is to be placed in the R0 register.

The last instruction "LDMFD SP!, R4,PC"[24] is an inverse instruction of STMFD. It loads values from the stack in order to save them into R4 and PC, and increments the stack pointer SP. It can be said that it is similar to POP. N.B. The very first instruction STMFD saves the R4 and LR registers pair on the stack, but R4 and PC are *restored* during execution of LDMFD.

As I wrote before, the address of the place to where each function must return control is usually saved in the LR register. The very first function saves its value in the stack because our `main()` function will use the register in order to call `printf()`. In the function end this value can be written to the PC register, thus passing control to where our function was called. Since our `main()` function is usually the primary function in C/C++, control will be returned to the OS loader or to a point in CRT, or something like that.

DCB is an assembly language directive defining an array of bytes or ASCII strings, akin to the DB directive in x86-assembly language.

2.3.2 Non-optimizing Keil: thumb mode

Let's compile the same example using Keil in thumb mode:

```
armcc.exe --thumb --c90 -O0 1.c
```

We will get (in IDA):

Listing 2.10: Non-optimizing Keil + thumb mode + IDA

```
.text:00000000                 main
.text:00000000 10 B5              PUSH    {R4,LR}
.text:00000002 C0 A0              ADR     R0, aHelloWorld ; "hello, ↙
    ↳ world"
.text:00000004 06 F0 2E F9        BL      __2printf
```

[20] Reduced instruction set computing
[21] Complex instruction set computing
[22] Read more about this in next section (4)
[23] MOVe
[24] Load Multiple Full Descending

```
.text:00000008 00 20          MOVS    R0, #0
.text:0000000A 10 BD          POP     {R4,PC}

.text:00000304 68 65 6C 6C+aHelloWorld  DCB "hello, world",0     ↙
    ↳ ; DATA XREF: main+2
```

We can easily spot the 2-byte (16-bit) opcodes. This is, as I mentioned, thumb. The BL instruction however consists of two 16-bit instructions. This is because it is impossible to load an offset for the `printf()` function into PC while using the small space in one 16-bit opcode. That's why the first 16-bit instruction loads the higher 10 bits of the offset and the second instruction loads the lower 11 bits of the offset. As I mentioned, all instructions in thumb mode have a size of 2 bytes (or 16 bits). This means it is impossible for a thumb-instruction to be at an odd address whatsoever. Given the above, the last address bit may be omitted while encoding instructions. In summary, in the BL thumb-instruction $\pm \approx 2M$ can be encoded as the offset from the current address.

As for the other instructions in the function: PUSH and POP work just like the described STMFD/LDMFD but the SP register is not mentioned explicitly here. ADR works just like in previous example. MOVS writes 0 into the R0 register in order to return zero.

2.3.3 Optimizing Xcode (LLVM) + ARM mode

Xcode 4.6.3 without optimization turned on produces a lot of redundant code so we'll study the version where the instruction count is as small as possible: -O3.

Listing 2.11: Optimizing Xcode (LLVM) + ARM mode

```
__text:000028C4              _hello_world
__text:000028C4 80 40 2D E9   STMFD           SP!, {R7,LR}
__text:000028C8 86 06 01 E3   MOV             R0, #0x1686
__text:000028CC 0D 70 A0 E1   MOV             R7, SP
__text:000028D0 00 00 40 E3   MOVT            R0, #0
__text:000028D4 00 00 8F E0   ADD             R0, PC, R0
__text:000028D8 C3 05 00 EB   BL              _puts
__text:000028DC 00 00 A0 E3   MOV             R0, #0
__text:000028E0 80 80 BD E8   LDMFD           SP!, {R7,PC}

__cstring:00003F62 48 65 6C 6C+aHelloWorld_0  DCB "Hello world↙
    ↳ !",0
```

The instructions STMFD and LDMFD are familiar to us.

The MOV instruction just writes the number 0x1686 into the R0 register. This is the offset pointing to the "Hello world!" string.

The R7 register as it is standardized in [2] is a frame pointer. More on it below.

The MOVT R0, #0 instruction writes 0 into higher 16 bits of the register. The issue here is that the generic MOV instruction in ARM mode may write only the lower 16 bits of the register. Remember, all instruction opcodes in ARM mode are limited in size to 32 bits. Of course, this limitation is not related to moving between registers. That's why an additional instruction MOVT exists for writing into the higher bits (from 16 to 31 inclusive). However, its usage here is redundant because the "MOV R0, #0x1686" instruction above cleared the higher part of the register. This is probably a shortcoming of the compiler.

The "ADD R0, PC, R0" instruction adds the value in the PC to the value in the R0, to calculate absolute address of the "Hello world!" string. As we already know, it is "position-independent code" so this correction is essential here.

The BL instruction calls the puts() function instead of printf().

GCC replaced the first printf() call with puts(). Indeed: printf() with a sole argument is almost analogous to puts().

Almost because we need to be sure the string will not contain printf-control statements starting with %: then the effect of these two functions would be different [25].

Why did the compiler replace the printf() with puts()? Because puts() is faster [26].

puts() works faster because it just passes characters to stdout without comparing each to the % symbol.

Next, we see the familiar "MOV R0, #0" instruction intended to set the R0 register to 0.

2.3.4 Optimizing Xcode (LLVM) + thumb-2 mode

By default Xcode 4.6.3 generates code for thumb-2 in this manner:

Listing 2.12: Optimizing Xcode (LLVM) + thumb-2 mode

```
__text:00002B6C                     _hello_world
__text:00002B6C 80 B5               PUSH            {R7,LR}
__text:00002B6E 41 F2 D8 30         MOVW            R0, #0x13D8
__text:00002B72 6F 46               MOV             R7, SP
__text:00002B74 C0 F2 00 00         MOVT.W          R0, #0
__text:00002B78 78 44               ADD             R0, PC
__text:00002B7A 01 F0 38 EA         BLX             _puts
__text:00002B7E 00 20               MOVS            R0, #0
__text:00002B80 80 BD               POP             {R7,PC}

...
```

[25] It should also be noted the puts() does not require a '\n' new line symbol at the end of a string, so we do not see it here.

[26] http://www.ciselant.de/projects/gcc_printf/gcc_printf.html

```
__cstring:00003E70 48 65 6C 6C 6F 20+aHelloWorld  DCB "Hello ↙
    ↳ world!",0xA,0
```

The BL and BLX instructions in thumb mode, as we recall, are encoded as a pair of 16-bit instructions. In thumb-2 these *surrogate* opcodes are extended in such a way so that new instructions may be encoded here as 32-bit instructions. That's easily observable —opcodes of thumb-2 instructions also begin with 0xFx or 0xEx. But in the IDA listings two opcode bytes are swapped (for thumb and thumb-2 modes). For instructions in ARM mode, the order is the fourth byte, then the third, then the second and finally the first (due to different endianness). So as we can see, the MOVW, MOVT.W and BLX instructions begin with 0xFx.

One of the thumb-2 instructions is "MOVW R0, #0x13D8" —it writes a 16-bit value into the lower part of the R0 register.

Also, "MOVT.W R0, #0" works just like MOVT from the previous example but it works in thumb-2.

Among other differences, here the BLX instruction is used instead of BL. The difference is that, besides saving the RA[27] in the LR register and passing control to the puts() function, the processor is also switching from thumb mode to ARM (or back). This instruction is placed here since the instruction to which control is passed looks like (it is encoded in ARM mode):

```
__symbolstub1:00003FEC _puts           ; CODE XREF: _hello_world↙
    ↳ +E
__symbolstub1:00003FEC 44 F0 9F E5     LDR  PC, =__imp__puts
```

So, the observant reader may ask: why not call puts() right at the point in the code where it is needed?

Because it is not very space-efficient.

Almost any program uses external dynamic libraries (like DLL in Windows, .so in *NIX or .dylib in Mac OS X). Often-used library functions are stored in dynamic libraries, including the standard C-function puts().

In an executable binary file (Windows PE .exe, ELF or Mach-O) an import section is present. This is a list of symbols (functions or global variables) being imported from external modules along with the names of these modules.

The OS loader loads all modules it needs and, while enumerating import symbols in the primary module, determines the correct addresses of each symbol.

In our case, *__imp__puts* is a 32-bit variable where the OS loader will write the correct address of the function in an external library. Then the LDR instruction just takes the 32-bit value from this variable and writes it into the PC register, passing control to it.

So, in order to reduce the time that an OS loader needs for doing this procedure, it is good idea for it to write the address of each symbol only once to a specially-allocated place just for it.

[27] Return Address

Besides, as we have already figured out, it is impossible to load a 32-bit value into a register while using only one instruction without a memory access. So, it is optimal to allocate a separate function working in ARM mode with only one goal —to pass control to the dynamic library and then to jump to this short one-instruction function (the so-called thunk function) from thumb-code.

By the way, in the previous example (compiled for ARM mode) control passed by the `BL` instruction goes to the same thunk function. However the processor mode is not switched (hence the absence of an "X" in the instruction mnemonic).

Chapter 3

Function prologue and epilogue

A function prologue is a sequence of instructions at the start of a function. It often looks something like the following code fragment:

```
push    ebp
mov     ebp, esp
sub     esp, X
```

What these instruction do: saves the value in the EBP register, sets the value of the EBP register to the value of the ESP and then allocates space on the stack for local variables.

The value in the EBP is fixed over a period of function execution and it is to be used for local variables and arguments access. One can use ESP, but it is changing over time and it is not convenient.

The function epilogue frees allocated space in the stack, returns the value in the EBP register back to initial state and returns the control flow to callee:

```
mov     esp, ebp
pop     ebp
ret     0
```

Function prologues and epilogues are usually detected in disassemblers for function delimitation from each other.

3.1 Recursion

Epilogues and prologues can make recursion performance worse.

For example, once upon a time I wrote a function to seek the correct node in a binary tree. As a recursive function it would look stylish but since additional time is to be spend at each function call for the prologue/epilogue, it was working a couple of times slower than an iterative (recursion-free) implementation.

By the way, that is the reason compilers use tail call.

Chapter 4

Stack

A stack is one of the most fundamental data structures in computer science [1].

Technically, it is just a block of memory in process memory along with the ESP or RSP register in x86 or x64, or the SP register in ARM, as a pointer within the block.

The most frequently used stack access instructions are PUSH and POP (in both x86 and ARM thumb-mode). PUSH subtracts 4 in 32-bit mode (or 8 in 64-bit mode) from ESP/RSP/SP and then writes the contents of its sole operand to the memory address pointed to by ESP/RSP/SP.

POP is the reverse operation: get the data from memory pointed to by SP, put it in the operand (often a register) and then add 4 (or 8) to the stack pointer.

After stack allocation the stack pointer points to the end of stack. PUSH decreases the stack pointer and POP increases it. The end of the stack is actually at the beginning of the memory allocated for the stack block. It seems strange, but that's the way it is.

Nevertheless ARM not only has instructions supporting descending stacks but also ascending stacks.

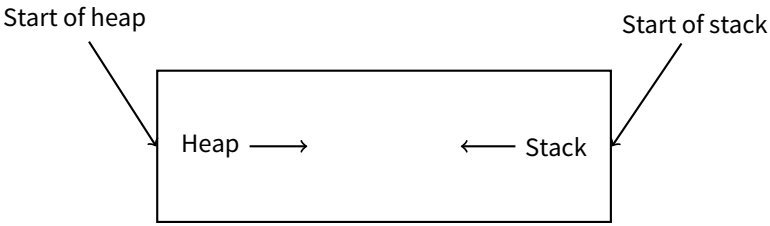
For example the STMFD[2]/LDMFD[3], STMED[4]/LDMED[5] instructions are intended to deal with a descending stack. The STMFA[6]/LMDFA[7], STMEA[8]/LDMEA[9] instructions are intended to deal with an ascending stack.

[1] http://en.wikipedia.org/wiki/Call_stack
[2] Store Multiple Full Descending
[3] Load Multiple Full Descending
[4] Store Multiple Empty Descending
[5] Load Multiple Empty Descending
[6] Store Multiple Full Ascending
[7] Load Multiple Full Ascending
[8] Store Multiple Empty Ascending
[9] Load Multiple Empty Ascending

4.1 Why does the stack grow backward?

Intuitively, we might think that, like any other data structure, the stack may grow upward, i.e., towards higher addresses.

The reason the stack grows backward is probably historical. When computers were big and occupied a whole room, it was easy to divide memory into two parts, one for the heap and one for the stack. Of course, it was unknown how big the heap and the stack would be during program execution, so this solution was the simplest possible.



In [26] we can read:

> The user-core part of an image is divided into three logical segments. The program text segment begins at location 0 in the virtual address space. During execution, this segment is write-protected and a single copy of it is shared among all processes executing the same program. At the first 8K byte boundary above the program text segment in the virtual address space begins a nonshared, writable data segment, the size of which may be extended by a system call. Starting at the highest address in the virtual address space is a stack segment, which automatically grows downward as the hardware's stack pointer fluctuates.

4.2 What is the stack used for?

4.2.1 Save the return address where a function must return control after execution

x86

While calling another function with a `CALL` instruction the address of the point exactly after the `CALL` instruction is saved to the stack and then an unconditional jump to the address in the CALL operand is executed.

The CALL instruction is equivalent to a PUSH address_after_call / JMP operand instruction pair.

RET fetches a value from the stack and jumps to it —it is equivalent to a POP tmp / JMP tmp instruction pair.

Overflowing the stack is straightforward. Just run eternal recursion:

```
void f()
{
        f();
};
```

MSVC 2008 reports the problem:

```
c:\tmp6>cl ss.cpp /Fass.asm
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version ↙
    ↳ 15.00.21022.08 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

ss.cpp
c:\tmp6\ss.cpp(4) : warning C4717: 'f' : recursive on all ↙
    ↳ control paths, function will cause runtime stack overflow
```

...but generates the right code anyway:

```
?f@@YAXXZ PROC                                          ; f
; File c:\tmp6\ss.cpp
; Line 2
        push    ebp
        mov     ebp, esp
; Line 3
        call    ?f@@YAXXZ                               ; f
; Line 4
        pop     ebp
        ret     0
?f@@YAXXZ ENDP                                          ; f
```

...Also if we turn on optimization (/Ox option) the optimized code will not overflow the stack but instead will work *correctly*[10]:

```
?f@@YAXXZ PROC                                          ; f
; File c:\tmp6\ss.cpp
; Line 2
$LL3@f:
; Line 3
        jmp     SHORT $LL3@f
```

[10] irony here

```
?f@@YAXXZ ENDP                                          ; f
```

GCC 4.4.1 generates similar code in both cases, although without issuing any warning about the problem.

ARM

ARM programs also use the stack for saving return addresses, but differently. As mentioned in "Hello, world!" (2.3), the RA is saved to the LR (link register). However, if one needs to call another function and use the LR register one more time its value should be saved. Usually it is saved in the function prologue. Often, we see instructions like "`PUSH R4-R7,LR`" along with this instruction in epilogue "`POP R4-R7,PC`" —thus register values to be used in the function are saved in the stack, including LR.

Nevertheless, if a function never calls any other function, in ARM terminology it is called a *leaf function*[11]. As a consequence, leaf functions do not use the LR register. If this function is small and uses a small number of registers, it may not use the stack at all. Thus, it is possible to call leaf functions without using the stack. This can be faster than on x86 because external RAM is not used for the stack [12]. It can be useful for such situations when memory for the stack is not yet allocated or not available.

4.2.2 Passing function arguments

The most popular way to pass parameters in x86 is called "cdecl":

```
push arg3
push arg2
push arg1
call f
add esp, 4*3
```

Callee functions get their arguments via the stack pointer.

Consequently, this is how values will be located in the stack before execution of the very first instruction of the f() function:

ESP	return address
ESP+4	argument#1, marked in IDA as `arg_0`
ESP+8	argument#2, marked in IDA as `arg_4`
ESP+0xC	argument#3, marked in IDA as `arg_8`
...	...

[11] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka13785.html

[12] Some time ago, on PDP-11 and VAX, the CALL instruction (calling other functions) was expensive; up to 50% of execution time might be spent on it, so it was common sense that big number of small function is anti-pattern [25, Chapter 4, Part II].

See also the section about other calling conventions (44). It is worth noting that nothing obliges programmers to pass arguments through the stack. It is not a requirement. One could implement any other method without using the stack at all.

For example, it is possible to allocate a space for arguments in the heap, fill it and pass it to a function via a pointer to this block in the EAX register. This will work. [13]. However, it is a convenient custom in x86 and ARM to use the stack for this.

By the way, the callee function does not have any information about how many arguments were passed. Functions with a variable number of arguments (like `printf()`) determine the number by specifiers (which begin with a % sign) in the format string. If we write something like

```
printf("%d %d %d", 1234);
```

`printf()` will dump 1234, and then also two random numbers, which were laying near it in the stack, by chance.
That's why it is not very important how we declare the `main()` function: as `main()`, `main(int argc, char *argv[])` or `main(int argc, char *argv[], char *envp[])`.

In fact, CRT-code is calling `main()` roughly as:

```
push envp
push argv
push argc
call main
...
```

If you declare `main()` as `main()` without arguments, they are, nevertheless, still present in the stack, but not used. If you declare `main()` as `main(int argc, char *argv[])`, you will use two arguments, and third will remain "invisible" for your function. Even more than that, it is possible to declare `main(int argc)`, and it will work.

4.2.3 Local variable storage

A function could allocate space in the stack for its local variables just by shifting the stack pointer towards the stack bottom.

It is also not a requirement. You could store local variables wherever you like, but traditionally this is how it's done.

[13] For example, in the "The Art of Computer Programming" book by Donald Knuth, in section 1.4.1 dedicated to subroutines [18, section 1.4.1], we can read about one way to supply arguments to a subroutine is simply to list them after the JMP instruction passing control to subroutine. Knuth writes this method was particularly convenient on System/360.

4.2.4 x86: alloca() function

It is worth noting the `alloca()` function.[14].

This function works like `malloc()` but allocates memory just on the stack.

The allocated memory chunk does not need to be freed via a `free()` function call since the function epilogue (3) will return `ESP` back to its initial state and the allocated memory will be just annulled.

It is worth noting how `alloca()` is implemented.

In simple terms, this function just shifts `ESP` downwards toward the stack bottom by the number of bytes you need and sets `ESP` as a pointer to the *allocated* block. Let's try:

```
#ifdef __GNUC__
#include <alloca.h> // GCC
#else
#include <malloc.h> // MSVC
#endif
#include <stdio.h>

void f()
{
    char *buf=(char*)alloca (600);
#ifdef __GNUC__
    snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // GCC
#else
    _snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // MSVC
#endif

    puts (buf);
};
```

(`_snprintf()` function works just like `printf()`, but instead of dumping the result into stdout (e.g., to terminal or console), it writes to the `buf` buffer. `puts()` copies `buf` contents to stdout. Of course, these two function calls might be replaced by one `printf()` call, but I would like to illustrate small buffer usage.)

MSVC

Let's compile (MSVC 2010):

Listing 4.1: MSVC 2010

```
...
```

[14]In MSVC, the function implementation can be found in `alloca16.asm` and `chkstk.asm` in `C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\crt\src\intel`

```
    mov     eax, 600          ; 00000258H
    call    __alloca_probe_16
    mov     esi, esp

    push    3
    push    2
    push    1
    push    OFFSET $SG2672
    push    600               ; 00000258H
    push    esi
    call    __snprintf

    push    esi
    call    _puts
    add     esp, 28           ; 0000001cH

...
```

The sole `alloca()` argument is passed via `EAX` (instead of pushing into stack) [15]. After the `alloca()` call, `ESP` points to the block of 600 bytes and we can use it as memory for the `buf` array.

GCC + Intel syntax

GCC 4.4.1 can do the same without calling external functions:

Listing 4.2: GCC 4.7.3

```
.LC0:
        .string "hi! %d, %d, %d\n"
f:
        push    ebp
        mov     ebp, esp
        push    ebx
        sub     esp, 660
        lea     ebx, [esp+39]
        and     ebx, -16                                ; align ↙
    ↳ pointer by 16-bit border
        mov     DWORD PTR [esp], ebx                    ; s
        mov     DWORD PTR [esp+20], 3
```

[15] It is because alloca() is rather compiler intrinsic (63) than usual function.
One of the reason there is a separate function instead of couple instructions just in the code, because MSVC[16] implementation of the alloca() function also has a code which reads from the memory just allocated, in order to let OS to map physical memory to this VM[17] region.

```
        mov     DWORD PTR [esp+16], 2
        mov     DWORD PTR [esp+12], 1
        mov     DWORD PTR [esp+8], OFFSET FLAT:.LC0 ; "hi! %d, %↙
    ↪ d, %d\n"
        mov     DWORD PTR [esp+4], 600              ; maxlen
        call    _snprintf
        mov     DWORD PTR [esp], ebx                ; s
        call    puts
        mov     ebx, DWORD PTR [ebp-4]
        leave
        ret
```

GCC + AT&T syntax

Let's see the same code, but in AT&T syntax:

Listing 4.3: GCC 4.7.3

```
.LC0:
        .string "hi! %d, %d, %d\n"
f:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ebx
        subl    $660, %esp
        leal    39(%esp), %ebx
        andl    $-16, %ebx
        movl    %ebx, (%esp)
        movl    $3, 20(%esp)
        movl    $2, 16(%esp)
        movl    $1, 12(%esp)
        movl    $.LC0, 8(%esp)
        movl    $600, 4(%esp)
        call    _snprintf
        movl    %ebx, (%esp)
        call    puts
        movl    -4(%ebp), %ebx
        leave
        ret
```

The code is the same as in the previous listing.

N.B. E.g. `movl $3, 20(%esp)` is analogous to `mov DWORD PTR [esp+20], 3` in Intel-syntax —when addressing memory in form *register+offset*, it is written in AT&T syntax as `offset(%register)`.

4.2.5 (Windows) SEH

SEH[18] records are also stored on the stack (if they present)..
Read more about it: (48.3).

4.2.6 Buffer overflow protection

More about it here (16.2).

4.3 Typical stack layout

A very typical stack layout in a 32-bit environment at the start of a function:

...	...
ESP-0xC	local variable #2, marked in IDA as `var_8`
ESP-8	local variable #1, marked in IDA as `var_4`
ESP-4	saved value of EBP
ESP	return address
ESP+4	argument#1, marked in IDA as `arg_0`
ESP+8	argument#2, marked in IDA as `arg_4`
ESP+0xC	argument#3, marked in IDA as `arg_8`
...	...

[18] Structured Exception Handling: 48.3

Chapter 5

printf() with several arguments

Now let's extend the *Hello, world!* (2) example, replacing `printf()` in the `main()` function body by this:

```
#include <stdio.h>

int main()
{
        printf("a=%d; b=%d; c=%d", 1, 2, 3);
        return 0;
};
```

5.1 x86: 3 arguments

5.1.1 MSVC

Let's compile it by MSVC 2010 Express and we got:

```
$SG3830 DB      'a=%d; b=%d; c=%d', 00H

...

        push    3
        push    2
        push    1
        push    OFFSET $SG3830
        call    _printf
```

```
        add     esp, 16                                         ; ↙
    ↳ 00000010H
```

Almost the same, but now we can see the `printf()` arguments are pushed onto the stack in reverse order. The first argument is pushed last.

By the way, variables of *int* type in 32-bit environment have 32-bit width, that is 4 bytes.

So, we have here 4 arguments. $4 * 4 = 16$ —they occupy exactly 16 bytes in the stack: a 32-bit pointer to a string and 3 numbers of type *int*.

When the stack pointer (`ESP` register) is corrected by ‘`ADD ESP, X`’ instruction after a function call, often, the number of function arguments can be deduced here: just divide X by 4.

Of course, this is related only to *cdecl* calling convention.

See also the section about calling conventions (44).

It is also possible for the compiler to merge several ‘`ADD ESP, X`’ instructions into one, after the last call:

```
push a1
push a2
call ...
...
push a1
call ...
...
push a1
push a2
push a3
call ...
add esp, 24
```

5.1.2 MSVC and OllyDbg

Now let’s try to load this example in OllyDbg. It is one of the most popular userland win32 debugger. We can try to compile our example in MSVC 2012 with `/MD` option, meaning, to link against `MSVCR*.DLL`, so we will able to see imported functions clearly in debugger.

Then load executable in OllyDbg. The very first breakpoint is in `ntdll.dll`, press F9 (run). The second breakpoint is in CRT-code. Now we should find `main()` function.

Find this code by scrolling the code to the very bottom (MSVC allocates `main()` function at the very beginning of the code section): fig. 5.3.

Click on `PUSH EBP` instruction, press F2 (set breakpoint) and press F9 (run). We need to do these manupulations in order to skip CRT-code, because, we don’t really interesting in it yet.

Press F8 (step over) 6 times, i.e., skip 6 instructions: fig. 5.4.

Now the PC points to the `CALL printf` instruction. OllyDbg, like other debuggers, highlights value of registers which were changed. So each time you press F8, `EIP` is changing and its value looking red. `ESP` is changing as well, because values are pushed into the stack.

Where are the values in the stack? Take a look into right/bottom window of debugger:

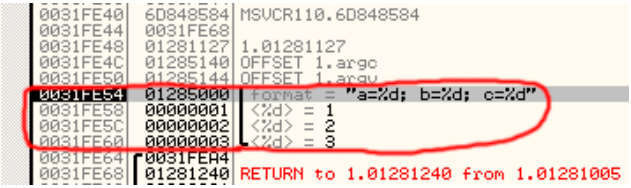


Figure 5.1: OllyDbg: stack after values pushed (I made round red mark here in graphics editor)

So we can see there 3 columns: address in the stack, value in the stack and some additional OllyDbg comments. OllyDbg understands `printf()`-like strings, so it reports the string here and 3 values *attached* to it.

It is possible to right-click on the format string, click on "Follow in dump", and the format string will appear in the window at the left-bottom part, where some memory part is always seen. These memory values can be edited. It is possible to change the format string, and then the result of our example will be different. It is probably not very useful now, but it's very good idea for doing it as exercise, to get feeling how everything is works here.

Press F8 (step over).

In the console we'll see the output:

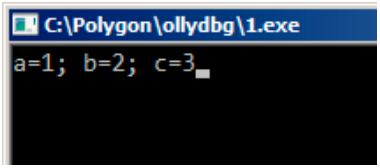


Figure 5.2: `printf()` function executed

Let's see how registers and stack state are changed: fig. 5.5.

`EAX` register now contains `0xD` (13). That's correct, `printf()` returns number of characters printed. `EIP` value is changed: indeed, now there is address of the in-

struction after CALL printf. ECX and EDX values are changed as well. Apparently, printf() function's hidden machinery used them for its own needs.

A very important thing is that ESP value is not changed. And stack state too! We clearly see that format string and corresponding 3 values are still there. Indeed, that's *cdecl* calling convention, calling function doesn't clear arguments in stack. It's caller's duty to do so.

Press F8 again to execute ADD ESP, 10 instruction: fig. 5.6.

ESP is changed, but values are still in the stack! Yes, of course, no one needs to fill these values by zero or something like that. Because, everything above stack pointer (SP) is *noise* or *garbage*, it has no value at all. It would be time consuming to clear unused stack entries, besides, no one really needs to.

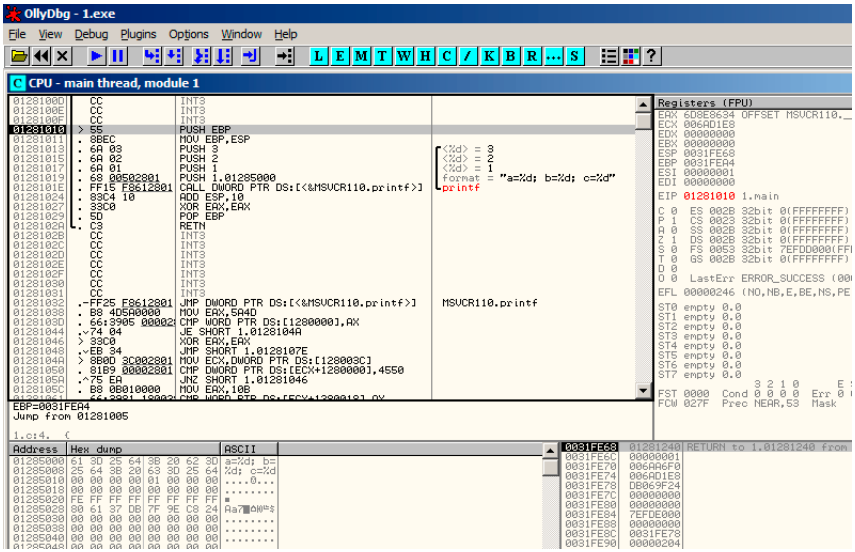


Figure 5.3: OllyDbg: the very start of the main() function

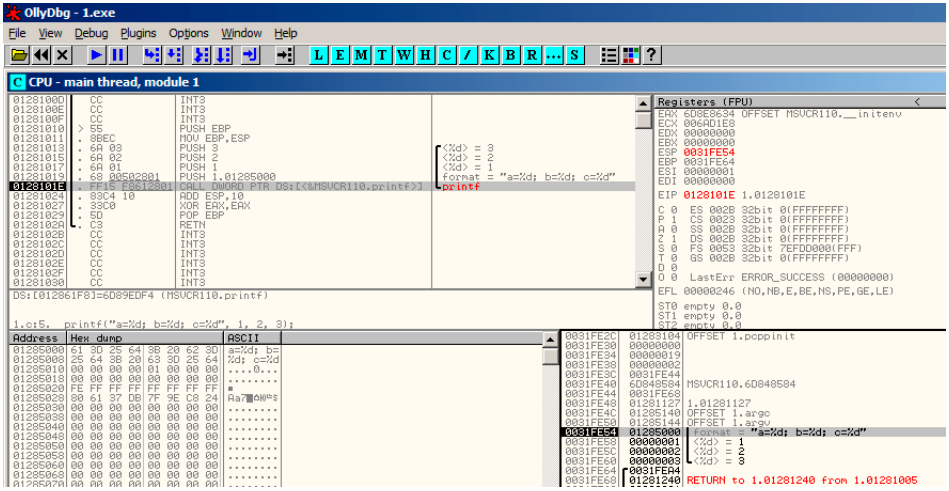


Figure 5.4: OllyDbg: before `printf()` execution

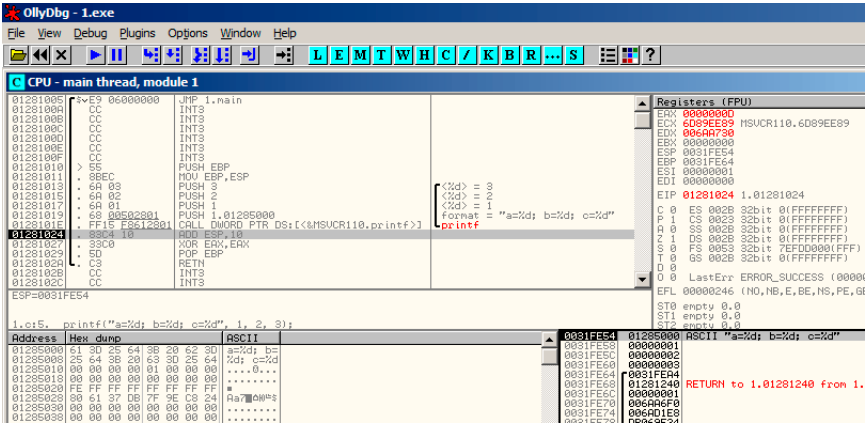


Figure 5.5: OllyDbg: after `printf()` execution

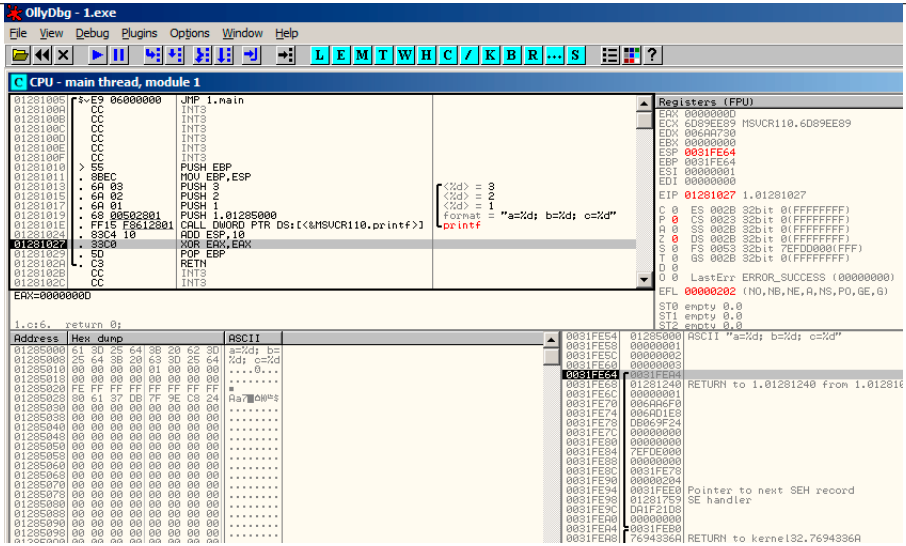


Figure 5.6: OllyDbg: after ADD ESP, 10 instruction execution

5.1.3 GCC

Now let's compile the same program in Linux using GCC 4.4.1 and take a look in IDA what we got:

```
main            proc near

var_10          = dword ptr -10h
var_C           = dword ptr -0Ch
var_8           = dword ptr -8
var_4           = dword ptr -4

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 10h
                mov     eax, offset aADBDCD ; "a=%d; b=%d; c=%d"
                mov     [esp+10h+var_4], 3
                mov     [esp+10h+var_8], 2
                mov     [esp+10h+var_C], 1
                mov     [esp+10h+var_10], eax
                call    _printf
                mov     eax, 0
                leave
```

```
                retn
main            endp
```

It can be said that the difference between code from MSVC and code from GCC is only in the method of placing arguments on the stack. Here GCC is working directly with the stack without PUSH/POP.

5.1.4 GCC and GDB

Let's try this example also in GDB[1] in Linux.

-g mean produce debug information into executable file.

```
$ gcc 1.c -g -o 1
```

```
$ gdb 1
GNU gdb (GDB) 7.6.1-ubuntu
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/↙
    ↳ licenses/gpl.html>
This is free software: you are free to change and redistribute ↙
    ↳ it.
There is NO WARRANTY, to the extent permitted by law.  Type "↙
    ↳ show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/dennis/polygon/1...done.
```

Listing 5.1: let's set breakpoint on printf()

```
(gdb) b printf
Breakpoint 1 at 0x80482f0
```

Run. There are no printf() function source code here, so GDB can't show its source, but may do so.

```
(gdb) run
Starting program: /home/dennis/polygon/1

Breakpoint 1, __printf (format=0x80484f0 "a=%d; b=%d; c=%d") at ↙
    ↳ printf.c:29
29      printf.c: No such file or directory.
```

[1] GNU debugger

Print 10 stack elements. Left column is an address in stack.

```
(gdb) x/10w $esp
0xbffff11c:     0x0804844a      0x080484f0      0x00000001      ↙
    ↳ 0x00000002
0xbffff12c:     0x00000003      0x08048460      0x00000000      ↙
    ↳ 0x00000000
0xbffff13c:     0xb7e29905      0x00000001
```

The very first element is RA (0x0804844a). We can be sure in it by disassembling the memory at this address:

```
(gdb) x/5i 0x0804844a
   0x804844a <main+45>: mov    $0x0,%eax
   0x804844f <main+50>: leave
   0x8048450 <main+51>: ret
   0x8048451:   xchg   %ax,%ax
   0x8048453:   xchg   %ax,%ax
```

Two XCHG instructions, apparently, is some random garbage, which we can ignore so far.

The second element (0x080484f0) is an address of format string:

```
(gdb) x/s 0x080484f0
0x80484f0:      "a=%d; b=%d; c=%d"
```

Other 3 elements (1, 2, 3) are printf() arguments. Other elements may be just "garbage" present in stack, but also may be values from other functions, their local variables, etc. We can ignore it yet.

Execute "finish". This mean, execute till function end. Here it means: execute till the finish of printf().

```
(gdb) finish
Run till exit from #0  __printf (format=0x80484f0 "a=%d; b=%d; c↙
    ↳ =%d") at printf.c:29
main () at 1.c:6
6               return 0;
Value returned is $2 = 13
```

GDB shows what printf() returned in EAX (13). This is number of characters printed, just like in the example with OllyDbg.

We also see "return 0;" and the information that this expression is in the 1.c file at the line 6. Indeed, the 1.c file is located in the current directory, and GDB finds the string there. How GDB knows, which C-code line is being executed now? This is related to the fact that compiler, while generating debugging information, also saves a table of relations between source code line numbers and instruction addresses. GDB is source-level debugger, after all.

Let's examine registers. 13 in EAX:

```
(gdb) info registers
eax            0xd      13
ecx            0x0      0
edx            0x0      0
ebx            0xb7fc0000       -1208221696
esp            0xbffff120       0xbffff120
ebp            0xbffff138       0xbffff138
esi            0x0      0
edi            0x0      0
eip            0x804844a        0x804844a <main+45>
...
```

Let's disassemble current instructions. Arrow points to the instruction being executed next.

```
(gdb) disas
Dump of assembler code for function main:
   0x0804841d <+0>:     push   %ebp
   0x0804841e <+1>:     mov    %esp,%ebp
   0x08048420 <+3>:     and    $0xfffffff0,%esp
   0x08048423 <+6>:     sub    $0x10,%esp
   0x08048426 <+9>:     movl   $0x3,0xc(%esp)
   0x0804842e <+17>:    movl   $0x2,0x8(%esp)
   0x08048436 <+25>:    movl   $0x1,0x4(%esp)
   0x0804843e <+33>:    movl   $0x80484f0,(%esp)
   0x08048445 <+40>:    call   0x80482f0 <printf@plt>
=> 0x0804844a <+45>:    mov    $0x0,%eax
   0x0804844f <+50>:    leave
   0x08048450 <+51>:    ret
End of assembler dump.
```

GDB shows disassembly in AT&T syntax by default. It's possible to switch to Intel syntax:

```
(gdb) set disassembly-flavor intel
(gdb) disas
Dump of assembler code for function main:
   0x0804841d <+0>:     push   ebp
   0x0804841e <+1>:     mov    ebp,esp
   0x08048420 <+3>:     and    esp,0xfffffff0
   0x08048423 <+6>:     sub    esp,0x10
   0x08048426 <+9>:     mov    DWORD PTR [esp+0xc],0x3
   0x0804842e <+17>:    mov    DWORD PTR [esp+0x8],0x2
   0x08048436 <+25>:    mov    DWORD PTR [esp+0x4],0x1
```

```
   0x0804843e <+33>:    mov    DWORD PTR [esp],0x80484f0
   0x08048445 <+40>:    call   0x80482f0 <printf@plt>
=> 0x0804844a <+45>:    mov    eax,0x0
   0x0804844f <+50>:    leave
   0x08048450 <+51>:    ret
End of assembler dump.
```

Execute next instruction. GDB shows ending bracket, meaning, this is ending block of function.

```
(gdb) step
7       };
```

Let's see registers after `MOV EAX, 0` instruction execution. `EAX` here is zero indeed.

```
(gdb) info registers
eax            0x0      0
ecx            0x0      0
edx            0x0      0
ebx            0xb7fc0000       -1208221696
esp            0xbffff120       0xbffff120
ebp            0xbffff138       0xbffff138
esi            0x0      0
edi            0x0      0
eip            0x804844f        0x804844f <main+50>
...
```

5.2 x64: 8 arguments

To see how other arguments will be passed via the stack, let's change our example again by increasing the number of arguments to be passed to 9 (`printf()` format string + 8 *int* variables):

```
#include <stdio.h>

int main()
{
        printf("a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n↙
    ↳ ", 1, 2, 3, 4, 5, 6, 7, 8);
        return 0;
};
```

5.2.1 MSVC

As we saw before, the first 4 arguments are passed in the RCX, RDX, R8, R9 registers in Win64, while all the rest—via the stack. That is what we see here. However, the MOV instruction, instead of PUSH, is used for preparing the stack, so the values are written to the stack in a straightforward manner.

Listing 5.2: MSVC 2012 x64

```
$SG2923 DB      'a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d↙
    ↳ ', 0aH, 00H

main    PROC
        sub     rsp, 88

        mov     DWORD PTR [rsp+64], 8
        mov     DWORD PTR [rsp+56], 7
        mov     DWORD PTR [rsp+48], 6
        mov     DWORD PTR [rsp+40], 5
        mov     DWORD PTR [rsp+32], 4
        mov     r9d, 3
        mov     r8d, 2
        mov     edx, 1
        lea     rcx, OFFSET FLAT:$SG2923
        call    printf

        ; return 0
        xor     eax, eax

        add     rsp, 88
        ret     0
main    ENDP
_TEXT   ENDS
END
```

5.2.2 GCC

In *NIX OS-es, it's the same story for x86-64, except that the first 6 arguments are passed in the RDI, RSI, RDX, RCX, R8, R9 registers. All the rest—via the stack. GCC generates the code writing string pointer into EDI instead if RDI—we saw this thing before: 2.2.2.

We also saw before the EAX register being cleared before a printf() call: 2.2.2.

Listing 5.3: GCC 4.4.6 -O3 x64

```
.LC0:
        .string "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\↙
    ↳ n"

main:
        sub     rsp, 40

        mov     r9d, 5
        mov     r8d, 4
        mov     ecx, 3
        mov     edx, 2
        mov     esi, 1
        mov     edi, OFFSET FLAT:.LC0
        xor     eax, eax ; number of vector registers passed
        mov     DWORD PTR [rsp+16], 8
        mov     DWORD PTR [rsp+8], 7
        mov     DWORD PTR [rsp], 6
        call    printf

        ; return 0

        xor     eax, eax
        add     rsp, 40
        ret
```

5.2.3 GCC + GDB

Let's try this example in GDB.

```
$ gcc -g 2.c -o 2
```

```
$ gdb 2
GNU gdb (GDB) 7.6.1-ubuntu
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/↙
    ↳ licenses/gpl.html>
This is free software: you are free to change and redistribute ↙
    ↳ it.
There is NO WARRANTY, to the extent permitted by law.  Type "↙
    ↳ show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
```

```
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/dennis/polygon/2...done.
```

Listing 5.4: let's set breakpoint to `printf()`, and run

```
(gdb) b printf
Breakpoint 1 at 0x400410
(gdb) run
Starting program: /home/dennis/polygon/2

Breakpoint 1, __printf (format=0x400628 "a=%d; b=%d; c=%d; d=%d;↙
    ↳  e=%d; f=%d; g=%d; h=%d\n") at printf.c:29
29      printf.c: No such file or directory.
```

Registers RSI/RDX/RCX/R8/R9 has the values which are should be there. RIP has an address of the very first instruction of the `printf()` function.

```
(gdb) info registers
rax            0x0      0
rbx            0x0      0
rcx            0x3      3
rdx            0x2      2
rsi            0x1      1
rdi            0x400628 4195880
rbp            0x7fffffffdf60   0x7fffffffdf60
rsp            0x7fffffffdf38   0x7fffffffdf38
r8             0x4      4
r9             0x5      5
r10            0x7fffffffdce0   140737488346336
r11            0x7ffff7a65f60   140737348263776
r12            0x400440 4195392
r13            0x7fffffffe040   140737488347200
r14            0x0      0
r15            0x0      0
rip            0x7ffff7a65f60   0x7ffff7a65f60 <__printf>
...
```

Listing 5.5: let's inspect format string

```
(gdb) x/s $rdi
0x400628:       "a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\↙
    ↳ n"
```

Let's dump stack with x/g command this time—g means *giant words*, i.e., 64-bit words.

```
(gdb) x/10g $rsp
0x7fffffffdf38: 0x0000000000400576      0x0000000000000006
0x7fffffffdf48: 0x0000000000000007      0x00007fff00000008
0x7fffffffdf58: 0x0000000000000000      0x0000000000000000
0x7fffffffdf68: 0x00007ffff7a33de5      0x0000000000000000
0x7fffffffdf78: 0x00007fffffffe048      0x0000000100000000
```

The very first stack element, just like in previous case, is RA. 3 values are also passed in stack: 6, 7, 8. We also see that 8 is passed with high 32-bits not cleared: 0x00007fff00000008. That's OK, because, values has *int* type, which is 32-bit type. So, high register or stack element part may contain "random garbage".

If to take a look, where control flow will return after `printf()` execution, GDB will show the whole `main()` function:

```
(gdb) set disassembly-flavor intel
(gdb) disas 0x0000000000400576
Dump of assembler code for function main:
   0x000000000040052d <+0>:     push   rbp
   0x000000000040052e <+1>:     mov    rbp,rsp
   0x0000000000400531 <+4>:     sub    rsp,0x20
   0x0000000000400535 <+8>:     mov    DWORD PTR [rsp+0x10],0x8
   0x000000000040053d <+16>:    mov    DWORD PTR [rsp+0x8],0x7
   0x0000000000400545 <+24>:    mov    DWORD PTR [rsp],0x6
   0x000000000040054c <+31>:    mov    r9d,0x5
   0x0000000000400552 <+37>:    mov    r8d,0x4
   0x0000000000400558 <+43>:    mov    ecx,0x3
   0x000000000040055d <+48>:    mov    edx,0x2
   0x0000000000400562 <+53>:    mov    esi,0x1
   0x0000000000400567 <+58>:    mov    edi,0x400628
   0x000000000040056c <+63>:    mov    eax,0x0
   0x0000000000400571 <+68>:    call   0x400410 <printf@plt>
   0x0000000000400576 <+73>:    mov    eax,0x0
   0x000000000040057b <+78>:    leave
   0x000000000040057c <+79>:    ret
End of assembler dump.
```

Let's finish `printf()` execution, execute the instruction zeroing `EAX`, take a notice that `EAX` register has exactly zero. `RIP` now points to the `LEAVE` instruction, i.e., penultimate in `main()` function.

```
(gdb) finish
Run till exit from #0  __printf (format=0x400628 "a=%d; b=%d; c↙
    ↳ =%d; d=%d; e=%d; f=%d; g=%d; h=%d\n") at printf.c:29
a=1; b=2; c=3; d=4; e=5; f=6; g=7; h=8
```

```
main () at 2.c:6
6               return 0;
Value returned is $1 = 39
(gdb) next
7       };
(gdb) info registers
rax            0x0      0
rbx            0x0      0
rcx            0x26     38
rdx            0x7ffff7dd59f0   140737351866864
rsi            0x7fffffd9       2147483609
rdi            0x0      0
rbp            0x7fffffffdf60   0x7fffffffdf60
rsp            0x7fffffffdf40   0x7fffffffdf40
r8             0x7ffff7dd26a0   140737351853728
r9             0x7ffff7a60134   140737348239668
r10            0x7fffffffd5b0   140737488344496
r11            0x7ffff7a95900   140737348458752
r12            0x400440 4195392
r13            0x7fffffffe040   140737488347200
r14            0x0      0
r15            0x0      0
rip            0x40057b 0x40057b <main+78>
...
```

5.3 ARM: 3 arguments

Traditionally, ARM's scheme for passing arguments (calling convention) is as follows: the first 4 arguments are passed in the R0-R3 registers; the remaining arguments, via the stack. This resembles the arguments passing scheme in fastcall (44.3) or win64 (44.5.1).

5.3.1 Non-optimizing Keil + ARM mode

Listing 5.6: Non-optimizing Keil + ARM mode

```
.text:00000014                 printf_main1
.text:00000014 10 40 2D E9     STMFD   SP!, {R4,LR}
.text:00000018 03 30 A0 E3     MOV     R3, #3
.text:0000001C 02 20 A0 E3     MOV     R2, #2
.text:00000020 01 10 A0 E3     MOV     R1, #1
.text:00000024 1D 0E 8F E2     ADR     R0, aADBDCD     ; "a=%d; b↙
    ↳ =%d; c=%d\n"
```

```
.text:00000028 0D 19 00 EB   BL      __2printf
.text:0000002C 10 80 BD E8   LDMFD   SP!, {R4,PC}
```

So, the first 4 arguments are passed via the R0-R3 registers in this order: a pointer to the `printf()` format string in R0, then 1 in R1, 2 in R2 and 3 in R3.

There is nothing unusual so far.

5.3.2 Optimizing Keil + ARM mode

Listing 5.7: Optimizing Keil + ARM mode

```
.text:00000014                EXPORT printf_main1
.text:00000014              printf_main1
.text:00000014 03 30 A0 E3    MOV     R3, #3
.text:00000018 02 20 A0 E3    MOV     R2, #2
.text:0000001C 01 10 A0 E3    MOV     R1, #1
.text:00000020 1E 0E 8F E2    ADR     R0, aADBDCD     ; "a=%d; b↙
    ↳ =%d; c=%d\n"
.text:00000024 CB 18 00 EA    B       __2printf
```

This is optimized (-O3) version for ARM mode and here we see B as the last instruction instead of the familiar BL. Another difference between this optimized version and the previous one (compiled without optimization) is also in the fact that there is no function prologue and epilogue (instructions that save R0 and LR registers values). The B instruction just jumps to another address, without any manipulation of the LR register, that is, it is analogous to JMP in x86. Why does it work? Because this code is, in fact, effectively equivalent to the previous. There are two main reasons: 1) neither the stack nor SP, the stack pointer, is modified; 2) the call to `printf()` is the last instruction, so there is nothing going on after it. After finishing, the `printf()` function will just return control to the address stored in LR. But the address of the point from where our function was called is now in LR! Consequently, control from `printf()` will be returned to that point. As a consequence, we do not need to save LR since we do not need to modify LR. We do not need to modify LR since there are no other function calls except `printf()`. Furthermore, after this call we do not to do anything! That's why this optimization is possible.

Another similar example was described in "switch()/case/default" section, here (11.1.1).

5.3.3 Optimizing Keil + thumb mode

Listing 5.8: Optimizing Keil + thumb mode

```
.text:0000000C              printf_main1
.text:0000000C 10 B5          PUSH    {R4,LR}
.text:0000000E 03 23          MOVS    R3, #3
```

```
.text:00000010 02 22          MOVS    R2, #2
.text:00000012 01 21          MOVS    R1, #1
.text:00000014 A4 A0          ADR     R0, aADBDCD     ; "a=%d; b↙
    ↳ =%d; c=%d\n"
.text:00000016 06 F0 EB F8    BL      __2printf
.text:0000001A 10 BD          POP     {R4,PC}
```

There is no significant difference from the non-optimized code for ARM mode.

5.4 ARM: 8 arguments

Let's use again the example with 9 arguments from the previous section: 5.2.

```
void printf_main2()
{
    printf("a=%d; b=%d; c=%d; d=%d; e=%d; f=%d; g=%d; h=%d\n",
        1, 2, 3, 4, 5, 6, 7, 8);
};
```

5.4.1 Optimizing Keil: ARM mode

```
.text:00000028              printf_main2
.text:00000028
.text:00000028              var_18 = -0x18
.text:00000028              var_14 = -0x14
.text:00000028              var_4  = -4
.text:00000028
.text:00000028 04 E0 2D E5  STR     LR, [SP,#var_4]!
.text:0000002C 14 D0 4D E2  SUB     SP, SP, #0x14
.text:00000030 08 30 A0 E3  MOV     R3, #8
.text:00000034 07 20 A0 E3  MOV     R2, #7
.text:00000038 06 10 A0 E3  MOV     R1, #6
.text:0000003C 05 00 A0 E3  MOV     R0, #5
.text:00000040 04 C0 8D E2  ADD     R12, SP, #0x18+var_14
.text:00000044 0F 00 8C E8  STMIA   R12, {R0-R3}
.text:00000048 04 00 A0 E3  MOV     R0, #4
.text:0000004C 00 00 8D E5  STR     R0, [SP,#0x18+var_18]
.text:00000050 03 30 A0 E3  MOV     R3, #3
.text:00000054 02 20 A0 E3  MOV     R2, #2
.text:00000058 01 10 A0 E3  MOV     R1, #1
.text:0000005C 6E 0F 8F E2  ADR     R0, aADBDCDDDEDFDGD ; "a=%d; ↙
    ↳ b=%d; c=%d; d=%d; e=%d; f=%d; g=%"...
```

```
.text:00000060 BC 18 00 EB  BL      __2printf
.text:00000064 14 D0 8D E2  ADD     SP, SP, #0x14
.text:00000068 04 F0 9D E4  LDR     PC, [SP+4+var_4],#4
```

This code can be divided into several parts:

- Function prologue:

  The very first ‘‘STR LR, [SP,#var_4]!’’ instruction saves LR on the stack, because we will use this register for the printf() call.

  The second ‘‘SUB SP, SP, #0x14’’ instruction decreases SP, the stack pointer, in order to allocate 0x14 (20) bytes on the stack. Indeed, we need to pass 5 32-bit values via the stack to the printf() function, and each one occupies 4 bytes, that is $5 * 4 = 20$ —exactly. The other 4 32-bit values will be passed in registers.

- Passing 5, 6, 7 and 8 via stack:

  Then, the values 5, 6, 7 and 8 are written to the R0, R1, R2 and R3 registers respectively. Then, the ‘‘ADD R12, SP, #0x18+var_14’’ instruction writes an address of the point in the stack, where these 4 variables will be written, into the R12 register. *var_14* is an assembly macro, equal to $-0x14$, such macros are created by IDA to succinctly denote code accessing the stack. *var_?* macros created by IDA reflecting local variables in the stack. So, $SP+4$ will be written into the R12 register. The next ‘‘STMIA R12, R0-R3’’ instruction writes R0-R3 registers contents at the point in memory to which R12 pointing. STMIA instruction meaning *Store Multiple Increment After*. *Increment After* means that R12 will be increased by 4 after each register value is written.

- Passing 4 via stack: 4 is stored in R0 and then, this value, with the help of ‘‘STR R0, [SP,#0x18+var_18]’’ instruction, is saved on the stack. *var_18* is $-0x18$, offset will be 0, so, the value from the R0 register (4) will be written to the point where SP is pointing to.

- Passing 1, 2 and 3 via registers:

  Values of the first 3 numbers (a, b, c) (1, 2, 3 respectively) are passed in the R1, R2 and R3 registers right before the printf() call, and the other 5 values are passed via the stack:

- printf() call:

- Function epilogue:

  The ‘‘ADD SP, SP, #0x14’’ instruction returns the SP pointer back to its former point, thus cleaning the stack. Of course, what was written on the stack will stay there, but it all will be rewritten during the execution of subsequent functions.

The ‘‘LDR PC, [SP+4+var_4],#4’’ instruction loads the saved LR value from the stack into the PC register, thus causing the function to exit.

5.4.2 Optimizing Keil: thumb mode

```
.text:0000001C                printf_main2
.text:0000001C
.text:0000001C                var_18 = -0x18
.text:0000001C                var_14 = -0x14
.text:0000001C                var_8  = -8
.text:0000001C
.text:0000001C 00 B5           PUSH    {LR}
.text:0000001E 08 23           MOVS    R3, #8
.text:00000020 85 B0           SUB     SP, SP, #0x14
.text:00000022 04 93           STR     R3, [SP,#0x18+var_8]
.text:00000024 07 22           MOVS    R2, #7
.text:00000026 06 21           MOVS    R1, #6
.text:00000028 05 20           MOVS    R0, #5
.text:0000002A 01 AB           ADD     R3, SP, #0x18+var_14
.text:0000002C 07 C3           STMIA   R3!, {R0-R2}
.text:0000002E 04 20           MOVS    R0, #4
.text:00000030 00 90           STR     R0, [SP,#0x18+var_18]
.text:00000032 03 23           MOVS    R3, #3
.text:00000034 02 22           MOVS    R2, #2
.text:00000036 01 21           MOVS    R1, #1
.text:00000038 A0 A0           ADR     R0, aADBDCDDDEDFDGD ; "a=%d;↙
    ↳  b=%d; c=%d; d=%d; e=%d; f=%d; g=%"...
.text:0000003A 06 F0 D9 F8     BL      __2printf
.text:0000003E
.text:0000003E                loc_3E   ; CODE XREF: example13_f+16
.text:0000003E 05 B0           ADD     SP, SP, #0x14
.text:00000040 00 BD           POP     {PC}
```

Almost same as in previous example, however, this is thumb code and values are packed into stack differently: 8 for the first time, then 5, 6, 7 for the second and 4 for the third.

5.4.3 Optimizing Xcode (LLVM): ARM mode

```
__text:0000290C                _printf_main2
__text:0000290C
__text:0000290C                var_1C = -0x1C
__text:0000290C                var_C  = -0xC
```

```
__text:0000290C
__text:0000290C 80 40 2D E9   STMFD  SP!, {R7,LR}
__text:00002910 0D 70 A0 E1   MOV    R7, SP
__text:00002914 14 D0 4D E2   SUB    SP, SP, #0x14
__text:00002918 70 05 01 E3   MOV    R0, #0x1570
__text:0000291C 07 C0 A0 E3   MOV    R12, #7
__text:00002920 00 00 40 E3   MOVT   R0, #0
__text:00002924 04 20 A0 E3   MOV    R2, #4
__text:00002928 00 00 8F E0   ADD    R0, PC, R0
__text:0000292C 06 30 A0 E3   MOV    R3, #6
__text:00002930 05 10 A0 E3   MOV    R1, #5
__text:00002934 00 20 8D E5   STR    R2, [SP,#0x1C+var_1C]
__text:00002938 0A 10 8D E9   STMFA  SP, {R1,R3,R12}
__text:0000293C 08 90 A0 E3   MOV    R9, #8
__text:00002940 01 10 A0 E3   MOV    R1, #1
__text:00002944 02 20 A0 E3   MOV    R2, #2
__text:00002948 03 30 A0 E3   MOV    R3, #3
__text:0000294C 10 90 8D E5   STR    R9, [SP,#0x1C+var_C]
__text:00002950 A4 05 00 EB   BL     _printf
__text:00002954 07 D0 A0 E1   MOV    SP, R7
__text:00002958 80 80 BD E8   LDMFD  SP!, {R7,PC}
```

Almost the same what we already figured out, with the exception of `STMFA` (Store Multiple Full Ascending) instruction, it is synonym to `STMIB` (Store Multiple Increment Before) instruction. This instruction increasing value in the SP register and only then writing next register value into memory, but not vice versa.

Another thing we easily spot is the instructions are ostensibly located randomly. For instance, value in the `R0` register is prepared in three places, at addresses `0x2918`, `0x2920` and `0x2928`, when it would be possible to do it in one single point. However, optimizing compiler has its own reasons about how to place instructions better. Usually, processor attempts to simultaneously execute instructions located side-by-side. For example, instructions like "`MOVT R0, #0`" and "`ADD R0, PC, R0`" cannot be executed simultaneously since they both modifying the `R0` register. On the other hand, "`MOVT R0, #0`" and "`MOV R2, #4`" instructions can be executed simultaneously since effects of their execution are not conflicting with each other. Presumably, compiler tries to generate code in such a way, where it is possible, of course.

5.4.4 Optimizing Xcode (LLVM): thumb-2 mode

```
__text:00002BA0                _printf_main2
__text:00002BA0
__text:00002BA0                var_1C = -0x1C
__text:00002BA0                var_18 = -0x18
```

```
__text:00002BA0                var_C  = -0xC
__text:00002BA0
__text:00002BA0 80 B5           PUSH     {R7,LR}
__text:00002BA2 6F 46           MOV      R7, SP
__text:00002BA4 85 B0           SUB      SP, SP, #0x14
__text:00002BA6 41 F2 D8 20     MOVW     R0, #0x12D8
__text:00002BAA 4F F0 07 0C     MOV.W    R12, #7
__text:00002BAE C0 F2 00 00     MOVT.W   R0, #0
__text:00002BB2 04 22           MOVS     R2, #4
__text:00002BB4 78 44           ADD      R0, PC  ; char *
__text:00002BB6 06 23           MOVS     R3, #6
__text:00002BB8 05 21           MOVS     R1, #5
__text:00002BBA 0D F1 04 0E     ADD.W    LR, SP, #0x1C+var_18
__text:00002BBE 00 92           STR      R2, [SP,#0x1C+var_1C]
__text:00002BC0 4F F0 08 09     MOV.W    R9, #8
__text:00002BC4 8E E8 0A 10     STMIA.W  LR, {R1,R3,R12}
__text:00002BC8 01 21           MOVS     R1, #1
__text:00002BCA 02 22           MOVS     R2, #2
__text:00002BCC 03 23           MOVS     R3, #3
__text:00002BCE CD F8 10 90     STR.W    R9, [SP,#0x1C+var_C]
__text:00002BD2 01 F0 0A EA     BLX      _printf
__text:00002BD6 05 B0           ADD      SP, SP, #0x14
__text:00002BD8 80 BD           POP      {R7,PC}
```

Almost the same as in previous example, with the exception the thumb-instructions are used here instead.

5.5 By the way

By the way, this difference between passing arguments in x86, x64, fastcall and ARM is a good illustration the CPU is not aware of how arguments is passed to functions. It is also possible to create hypothetical compiler which is able to pass arguments via a special structure not using stack at all.

Chapter 6

scanf()

Now let's use scanf().

```
#include <stdio.h>

int main()
{
        int x;
        printf ("Enter X:\n");

        scanf ("%d", &x);

        printf ("You entered %d...\n", x);

        return 0;
};
```

OK, I agree, it is not clever to use `scanf()` today. But I wanted to illustrate passing pointer to *int*.

6.1 About pointers

It is one of the most fundamental things in computer science. Often, large array, structure or object, it is too costly to pass to other function, while passing its address is much easier. More than that: if calling function must modify something in the large array or structure, to return it as a whole is absurdly as well. So the simplest thing to do is to pass an address of array or structure to function, and let it change what must be changed.

In C/C++ it is just an address of some point in memory.

In x86, address is represented as 32-bit number (i.e., occupying 4 bytes), while in x86–64 it is 64-bit number (occupying 8 bytes). By the way, that is the reason of some people's indignation related to switching to x86-64 —all pointers on x64-architecture will require twice as more space.

With some effort, it is possible to work only with untyped pointers; e.g. standard C function `memcpy()`, copying a block from one place in memory to another, takes 2 pointers of `void*` type on input, since it is impossible to predict block type you would like to copy. And it is not even important to know, only block size is important.

Also pointers are widely used when function needs to return more than one value (we will back to this in future (9)). *scanf()* is just that case. In addition to the function's need to show how many values were read successfully, it also should return all these values.

In C/C++ pointer type is needed only for type checking on compiling stage. Internally, in compiled code, there is no information about pointers types.

6.2 x86

6.2.1 MSVC

What we got after compiling in MSVC 2010:

```
CONST    SEGMENT
$SG3831    DB    'Enter X:', 0aH, 00H
$SG3832    DB    '%d', 00H
$SG3833    DB    'You entered %d...', 0aH, 00H
CONST    ENDS
PUBLIC    _main
EXTRN    _scanf:PROC
EXTRN    _printf:PROC
; Function compile flags: /Odtp
_TEXT    SEGMENT
_x$ = -4                        ; size = 4
_main    PROC
    push   ebp
    mov    ebp, esp
    push   ecx
    push   OFFSET $SG3831 ; 'Enter X:'
    call   _printf
    add    esp, 4
    lea    eax, DWORD PTR _x$[ebp]
    push   eax
    push   OFFSET $SG3832 ; '%d'
    call   _scanf
```

```
    add     esp, 8
    mov     ecx, DWORD PTR _x$[ebp]
    push    ecx
    push    OFFSET $SG3833 ; 'You entered %d...'
    call    _printf
    add     esp, 8

    ; return 0
    xor     eax, eax
    mov     esp, ebp
    pop     ebp
    ret     0
_main    ENDP
_TEXT    ENDS
```

Variable x is local.

C/C++ standard tell us it must be visible only in this function and not from any other point. Traditionally, local variables are placed in the stack. Probably, there could be other ways, but in x86 it is so.

Next instruction after function prologue, PUSH ECX, has not a goal to save ECX state (notice absence of corresponding POP ECX at the function end).

In fact, this instruction just allocates 4 bytes on the stack for x variable storage.

x will be accessed with the assistance of the _x$ macro (it equals to -4) and the EBP register pointing to current frame.

Over a span of function execution, EBP is pointing to current stack frame and it is possible to have an access to local variables and function arguments via EBP+offset.

It is also possible to use ESP, but it is often changing and not very convenient. So it can be said, the value of the EBP is *frozen state* of the value of the ESP at the moment of function execution start.

A very typical stack frame layout in 32-bit environment is:

...	...
EBP-8	local variable #2, marked in IDA as var_8
EBP-4	local variable #1, marked in IDA as var_4
EBP	saved value of EBP
EBP+4	return address
EBP+8	argument#1, marked in IDA as arg_0
EBP+0xC	argument#2, marked in IDA as arg_4
EBP+0x10	argument#3, marked in IDA as arg_8
...	...

Function scanf() in our example has two arguments.

First is pointer to the string containing "%d" and second —address of variable x.

First of all, address of the x variable is placed into the EAX register by lea eax, DWORD PTR _x$[ebp] instruction

LEA meaning *load effective address* but over a time it changed its primary application (B.6.2).

It can be said, LEA here just stores sum of the value in the EBP register and _x$ macro to the EAX register.

It is the same as lea eax, [ebp-4].

So, 4 subtracting from value in the EBP register and result is placed to the EAX register. And then value in the EAX register is pushing into stack and scanf() is called.

After that, printf() is called. First argument is pointer to string: "You entered %d...\n".

Second argument is prepared as: mov ecx, [ebp-4], this instruction places to the ECX not address of the x variable, but its contents.

After, value in the ECX is placed on the stack and the last printf() called.

6.2.2 MSVC + OllyDbg

Let's try this example in OllyDbg. Let's load, press F8 (step over) until we get into our executable file instead of ntdll.dll. Scroll up until main() appears. Let's click on the first instruction (PUSH EBP), press F2, then F9 (Run) and breakpoint triggers on the main() begin.

Let's trace to the place where the address of x variable is prepared: fig. 6.2.

It is possible to right-click on EAX in registers window and then "Follow in stack". This address will appear in stack window. Look, this is a variable in the local stack. I drawed a red arrow there. And there are some garbage (0x77D478). Now address of the stack element, with the help of PUSH, will be written to the same stack, nearly. Let's trace by F8 until scanf() execution finished. During the moment of scanf() execution, we enter, for example, 123, in the console window:

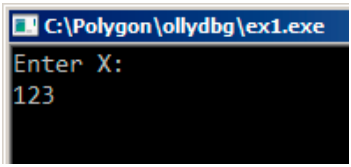


Figure 6.1: Console output

scanf() executed here: fig. 6.3. scanf() returns 1 in EAX, which means, it have read one value successfully. The element of stack of our attention now contain 0x7B (123).

Further, this value is copied from the stack to the ECX register and passed into printf(): fig. 6.4.

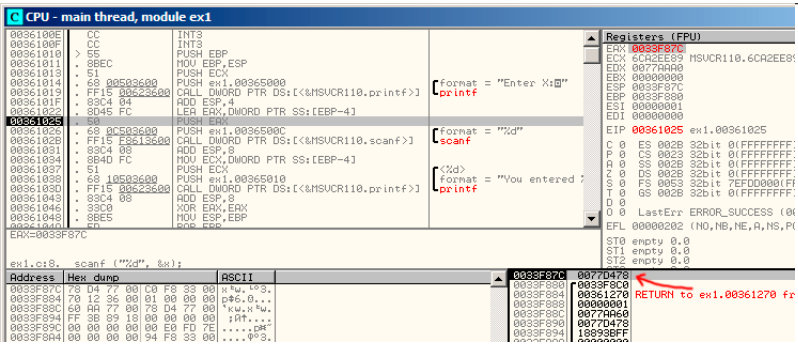


Figure 6.2: OllyDbg: address of the local variable is computed

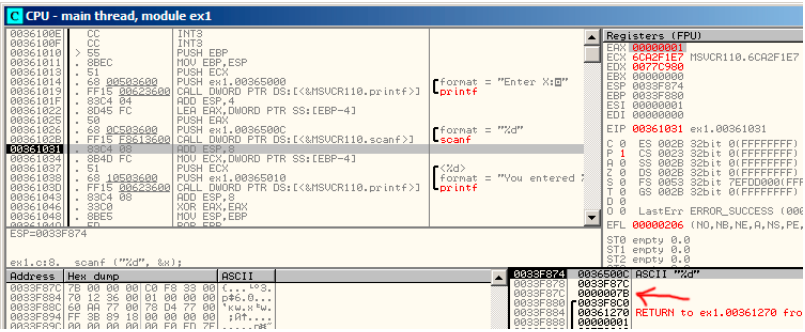


Figure 6.3: OllyDbg: `scanf()` executed

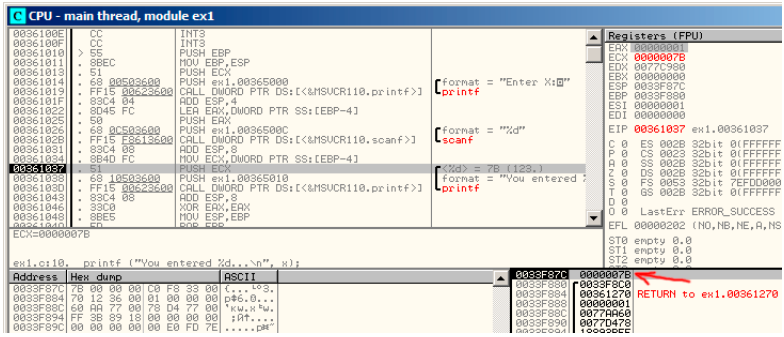


Figure 6.4: OllyDbg: preparing the value for passing into `printf()`

6.2.3 GCC

Let's try to compile this code in GCC 4.4.1 under Linux:

```
main            proc near

var_20          = dword ptr -20h
var_1C          = dword ptr -1Ch
var_4           = dword ptr -4

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 20h
                mov     [esp+20h+var_20], offset aEnterX ; "↙
    ↳ Enter X:"
                call    _puts
                mov     eax, offset aD  ; "%d"
                lea     edx, [esp+20h+var_4]
                mov     [esp+20h+var_1C], edx
                mov     [esp+20h+var_20], eax
                call    ___isoc99_scanf
                mov     edx, [esp+20h+var_4]
                mov     eax, offset aYouEnteredD___ ; "You ↙
    ↳ entered %d...\n"
                mov     [esp+20h+var_1C], edx
                mov     [esp+20h+var_20], eax
                call    _printf
                mov     eax, 0
                leave
                retn
main            endp
```

GCC replaced first the `printf()` call to the `puts()`, it was already described (2.3.3) why it was done.

As before —arguments are placed on the stack by `MOV` instruction.

6.3 x64

All the same, but registers are used instead of stack for arguments passing.

6.3.1 MSVC

Listing 6.1: MSVC 2012 x64

```
_DATA   SEGMENT
$SG1289 DB      'Enter X:', 0aH, 00H
$SG1291 DB      '%d', 00H
$SG1292 DB      'You entered %d...', 0aH, 00H
_DATA   ENDS

_TEXT   SEGMENT
x$ = 32
main    PROC
$LN3:
        sub     rsp, 56
        lea     rcx, OFFSET FLAT:$SG1289 ; 'Enter X:'
        call    printf
        lea     rdx, QWORD PTR x$[rsp]
        lea     rcx, OFFSET FLAT:$SG1291 ; '%d'
        call    scanf
        mov     edx, DWORD PTR x$[rsp]
        lea     rcx, OFFSET FLAT:$SG1292 ; 'You entered %d...'
        call    printf

        ; return 0
        xor     eax, eax
        add     rsp, 56
        ret     0
main    ENDP
_TEXT   ENDS
```

6.3.2 GCC

Listing 6.2: GCC 4.4.6 -O3 x64

```
.LC0:
        .string "Enter X:"
.LC1:
        .string "%d"
.LC2:
        .string "You entered %d...\n"

main:
        sub     rsp, 24
        mov     edi, OFFSET FLAT:.LC0 ; "Enter X:"
        call    puts
```

```
        lea     rsi, [rsp+12]
        mov     edi, OFFSET FLAT:.LC1 ; "%d"
        xor     eax, eax
        call    __isoc99_scanf
        mov     esi, DWORD PTR [rsp+12]
        mov     edi, OFFSET FLAT:.LC2 ; "You entered %d...\n"
        xor     eax, eax
        call    printf

        ; return 0
        xor     eax, eax
        add     rsp, 24
        ret
```

6.4 ARM

6.4.1 Optimizing Keil + thumb mode

```
.text:00000042             scanf_main
.text:00000042
.text:00000042             var_8           = -8
.text:00000042
.text:00000042 08 B5                       PUSH    {R3,LR}
.text:00000044 A9 A0                       ADR     R0, aEnterX  ↙
    ↳     ; "Enter X:\n"
.text:00000046 06 F0 D3 F8                 BL      __2printf
.text:0000004A 69 46                       MOV     R1, SP
.text:0000004C AA A0                       ADR     R0, aD       ↙
    ↳     ; "%d"
.text:0000004E 06 F0 CD F8                 BL      __0scanf
.text:00000052 00 99                       LDR     R1, [SP,#8+↙
    ↳ var_8]
.text:00000054 A9 A0                       ADR     R0, ↙
    ↳ aYouEnteredD___ ; "You entered %d...\n"
.text:00000056 06 F0 CB F8                 BL      __2printf
.text:0000005A 00 20                       MOVS    R0, #0
.text:0000005C 08 BD                       POP     {R3,PC}
```

A pointer to a *int*-typed variable must be passed to a `scanf()` so it can return value via it. *int* is 32-bit value, so we need 4 bytes for storing it somewhere in memory, and it fits exactly in 32-bit register. A place for the local variable `x` is allocated in the stack and IDA named it *var_8*, however, it is not necessary to allocate it since

SP stack pointer is already pointing to the space may be used instantly. So, SP stack pointer value is copied to the R1 register and, together with format-string, passed into scanf(). Later, with the help of the LDR instruction, this value is moved from stack into the R1 register in order to be passed into printf().

Examples compiled for ARM-mode and also examples compiled with Xcode LLVM are not differ significantly from what we saw here, so they are omitted.

6.5 Global variables

What if x variable from previous example will not be local but global variable? Then it will be accessible from any point, not only from function body. Global variables are considered as anti-pattern, but for the sake of experiment we could do this.

```
#include <stdio.h>

int x;

int main()
{
        printf ("Enter X:\n");

        scanf ("%d", &x);

        printf ("You entered %d...\n", x);

        return 0;
};
```

6.5.1 MSVC: x86

```
_DATA    SEGMENT
COMM    _x:DWORD
$SG2456    DB    'Enter X:', 0aH, 00H
$SG2457    DB    '%d', 00H
$SG2458    DB    'You entered %d...', 0aH, 00H
_DATA    ENDS
PUBLIC    _main
EXTRN    _scanf:PROC
EXTRN    _printf:PROC
; Function compile flags: /Odtp
_TEXT    SEGMENT
_main    PROC
```

```
    push   ebp
    mov    ebp, esp
    push   OFFSET $SG2456
    call   _printf
    add    esp, 4
    push   OFFSET _x
    push   OFFSET $SG2457
    call   _scanf
    add    esp, 8
    mov    eax, DWORD PTR _x
    push   eax
    push   OFFSET $SG2458
    call   _printf
    add    esp, 8
    xor    eax, eax
    pop    ebp
    ret    0
_main    ENDP
_TEXT    ENDS
```

Now x variable is defined in the _DATA segment. Memory in local stack is not allocated anymore. All accesses to it are not via stack but directly to process memory. Not initialized global variables takes no place in the executable file (indeed, why we should allocate a place in the executable file for initially zeroed variables?), but when someone will access this place in memory, OS will allocate a block of zeroes there[1].

Now let's assign value to variable explicitly:

```
int x=10; // default value
```

We got:

```
_DATA   SEGMENT
_x      DD      0aH

...
```

Here we see value 0xA of DWORD type (DD meaning DWORD = 32 bit).

If you will open compiled .exe in IDA, you will see the *x* variable placed at the beginning of the _DATA segment, and after you'll see text strings.

If you will open compiled .exe in IDA from previous example where *x* value is not defined, you'll see something like this:

```
.data:0040FA80 _x              dd ?                    ; DATA ↙
    ↳ XREF: _main+10
```

[1]That is how VM behaves

```
.data:0040FA80                                          ; _main↙
    ↳ +22
.data:0040FA84 dword_40FA84    dd ?                     ; DATA ↙
    ↳ XREF: _memset+1E
.data:0040FA84                                          ; ↙
    ↳ unknown_libname_1+28
.data:0040FA88 dword_40FA88    dd ?                     ; DATA ↙
    ↳ XREF: ___sbh_find_block+5
.data:0040FA88                                          ; ↙
    ↳ ___sbh_free_block+2BC
.data:0040FA8C ; LPVOID lpMem
.data:0040FA8C lpMem           dd ?                     ; DATA ↙
    ↳ XREF: ___sbh_find_block+B
.data:0040FA8C                                          ; ↙
    ↳ ___sbh_free_block+2CA
.data:0040FA90 dword_40FA90    dd ?                     ; DATA ↙
    ↳ XREF: _V6_HeapAlloc+13
.data:0040FA90                                          ; ↙
    ↳ __calloc_impl+72
.data:0040FA94 dword_40FA94    dd ?                     ; DATA ↙
    ↳ XREF: ___sbh_free_block+2FE
```

_x marked as ? among other variables not required to be initialized. This means that after loading .exe to memory, a space for all these variables will be allocated and a random garbage will be here. But in an .exe file these not initialized variables are not occupy anything. E.g. it is suitable for large arrays.

6.5.2 MSVC: x86 + OllyDbg

Things are even simpler here: fig. 6.5. Variable is located in the data segment. By the way, after PUSH instruction, pushing x address, is executed, the address will appear in stack, and it is possible to right-click on that element and select "Follow in dump". And the variable will appear in the memory window at left.

After we enter 123 in the console, 0x7B will appear here.

But why the very first byte is 7B? Thinking logically, a 00 00 00 7B should be there. This is what called endianness, and *little-endian* is used in x86. This mean that lowest byte is written first, and highest written last. More about it: 33.

Some time after, 32-bit value from this place of memory is loaded into EAX and passed into printf().

x variable address in the memory is 0xDC3390. In OllyDbg we can see process memory map (Alt-M) and we will see that this address is inside of .data PE-segment of our program: fig. 6.6.

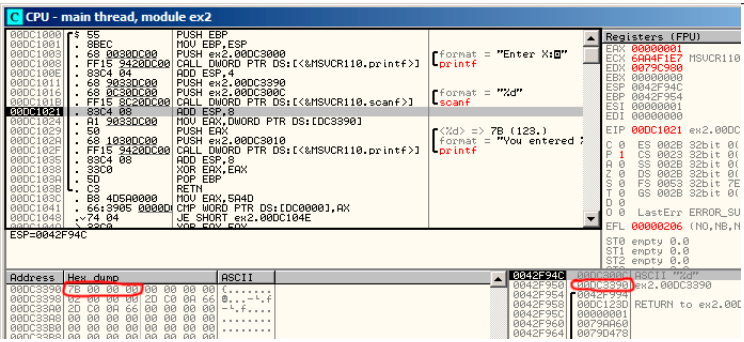


Figure 6.5: OllyDbg: after `scanf()` execution

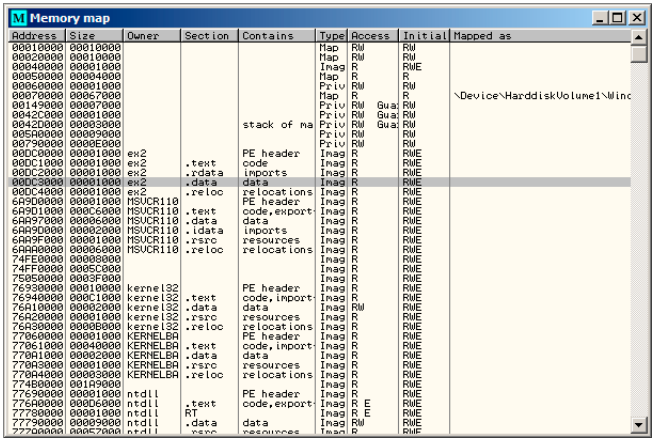


Figure 6.6: OllyDbg: process memory map

6.5.3 GCC: x86

It is almost the same in Linux, except segment names and properties: not initialized variables are located in the `_bss` segment. In ELF file format this segment has such attributes:

```
; Segment type: Uninitialized
; Segment permissions: Read/Write
```

If to statically assign a value to variable, e.g. 10, it will be placed in the `_data` segment, this is segment with the following attributes:

```
; Segment type: Pure data
```

```
; Segment permissions: Read/Write
```

6.5.4 MSVC: x64

Listing 6.3: MSVC 2012 x64

```
_DATA   SEGMENT
COMM    x:DWORD
$SG2924 DB      'Enter X:', 0aH, 00H
$SG2925 DB      '%d', 00H
$SG2926 DB      'You entered %d...', 0aH, 00H
_DATA   ENDS

_TEXT   SEGMENT
main    PROC
$LN3:
        sub     rsp, 40

        lea     rcx, OFFSET FLAT:$SG2924 ; 'Enter X:'
        call    printf
        lea     rdx, OFFSET FLAT:x
        lea     rcx, OFFSET FLAT:$SG2925 ; '%d'
        call    scanf
        mov     edx, DWORD PTR x
        lea     rcx, OFFSET FLAT:$SG2926 ; 'You entered %d...'
        call    printf

        ; return 0
        xor     eax, eax

        add     rsp, 40
        ret     0
main    ENDP
_TEXT   ENDS
```

Almost the same code as in x86. Take a notice that x variable address is passed to `scanf()` using `LEA` instruction, while the value of variable is passed to the second `printf()` using `MOV` instruction. ‘`DWORD PTR`’—is a part of assembly language (no related to machine codes), showing that the variable data type is 32-bit and the `MOV` instruction should be encoded accordingly.

6.5.5 ARM: Optimizing Keil + thumb mode

```
.text:00000000 ; Segment type: Pure code
.text:00000000                 AREA .text, CODE
...
.text:00000000 main
.text:00000000                 PUSH    {R4,LR}
.text:00000002                 ADR     R0, aEnterX     ; "Enter ↙
    ↳ X:\n"
.text:00000004                 BL      __2printf
.text:00000008                 LDR     R1, =x
.text:0000000A                 ADR     R0, aD          ; "%d"
.text:0000000C                 BL      __0scanf
.text:00000010                 LDR     R0, =x
.text:00000012                 LDR     R1, [R0]
.text:00000014                 ADR     R0, aYouEnteredD___ ; "↙
    ↳ You entered %d...\n"
.text:00000016                 BL      __2printf
.text:0000001A                 MOVS    R0, #0
.text:0000001C                 POP     {R4,PC}
...
.text:00000020 aEnterX         DCB "Enter X:",0xA,0    ; DATA ↙
    ↳ XREF: main+2
.text:0000002A                 DCB    0
.text:0000002B                 DCB    0
.text:0000002C off_2C          DCD x                   ; DATA ↙
    ↳ XREF: main+8
.text:0000002C                                         ; main+10
.text:00000030 aD              DCB "%d",0              ; DATA ↙
    ↳ XREF: main+A
.text:00000033                 DCB    0
.text:00000034 aYouEnteredD___ DCB "You entered %d...",0xA,0 ; ↙
    ↳ DATA XREF: main+14
.text:00000047                 DCB 0
.text:00000047 ; .text         ends
.text:00000047
...
.data:00000048 ; Segment type: Pure data
.data:00000048                 AREA .data, DATA
.data:00000048                 ; ORG 0x48
.data:00000048                 EXPORT x
.data:00000048 x               DCD 0xA                 ; DATA ↙
    ↳ XREF: main+8
.data:00000048                                         ; main+10
.data:00000048 ; .data         ends
```

So, x variable is now global and somehow, it is now located in another segment, namely data segment (*.data*). One could ask, why text strings are located in code segment (*.text*) and x can be located right here? Since this is variable, and by its definition, it can be changed. And probably, can be changed very often. Segment of code not infrequently can be located in microcontroller ROM (remember, we now deal with embedded microelectronics, and memory scarcity is common here), and changeable variables —in RAM[2]. It is not very economically to store constant variables in RAM when one have ROM. Furthermore, data segment with constants in RAM must be initialized before, since after RAM turning on, obviously, it contain random information.

Onwards, we see, in code segment, a pointer to the x (off_2C) variable, and all operations with variable occurred via this pointer. This is because x variable can be located somewhere far from this code fragment, so its address must be saved somewhere in close proximity to the code. LDR instruction in thumb mode can address only variable in range of 1020 bytes from the point it is located. Same instruction in ARM-mode —variables in range ± 4095 bytes, this, address of the x variable must be located somewhere in close proximity, because, there is no guarantee the linker will able to place this variable near the code, it could be even in external memory chip!

One more thing: if variable will be declared as *const*, Keil compiler shall allocate it in the .constdata segment. Perhaps, thereafter, linker will able to place this segment in ROM too, along with code segment.

6.6 scanf() result checking

As I noticed before, it is slightly old-fashioned to use scanf() today. But if we have to, we need at least check if scanf() finished correctly without error.

```
#include <stdio.h>

int main()
{
        int x;
        printf ("Enter X:\n");

        if (scanf ("%d", &x)==1)
                printf ("You entered %d...\n", x);
        else
                printf ("What you entered? Huh?\n");

        return 0;
};
```

[2] Random-access memory

By standard, `scanf()`[3] function returns number of fields it successfully read.

In our case, if everything went fine and user entered a number, `scanf()` will return 1 or 0 or EOF in case of error.

I added C code for `scanf()` result checking and printing error message in case of error.

This works predictably:

```
C:\...>ex3.exe
Enter X:
123
You entered 123...

C:\...>ex3.exe
Enter X:
ouch
What you entered? Huh?
```

6.6.1 MSVC: x86

What we got in assembly language (MSVC 2010):

```
        lea     eax, DWORD PTR _x$[ebp]
        push    eax
        push    OFFSET $SG3833 ; '%d', 00H
        call    _scanf
        add     esp, 8
        cmp     eax, 1
        jne     SHORT $LN2@main
        mov     ecx, DWORD PTR _x$[ebp]
        push    ecx
        push    OFFSET $SG3834 ; 'You entered %d...', 0aH, 00H
        call    _printf
        add     esp, 8
        jmp     SHORT $LN1@main
$LN2@main:
        push    OFFSET $SG3836 ; 'What you entered? Huh?', 0aH, ↙
    ↳ 00H
        call    _printf
        add     esp, 4
$LN1@main:
        xor     eax, eax
```

[3] MSDN: scanf, wscanf: http://msdn.microsoft.com/en-us/library/9y6s16x1(VS.71).aspx

Caller function (`main()`) must have access to the result of callee function (`scanf()`), so callee leaves this value in the `EAX` register.

After, we check it with the help of instruction `CMP EAX, 1` (*CoMPare*), in other words, we compare value in the `EAX` register with `1`.

`JNE` conditional jump follows `CMP` instruction. `JNE` means *Jump if Not Equal*.

So, if value in the `EAX` register not equals to `1`, then the processor will pass execution to the address mentioned in operand of `JNE`, in our case it is `$LN2@main`. Passing control to this address, CPU will execute function `printf()` with argument ''`What you entered? Huh?`''. But if everything is fine, conditional jump will not be taken, and another `printf()` call will be executed, with two arguments: '`You entered %d...`' and value of variable `x`.

Since second subsequent `printf()` not needed to be executed, there is `JMP` after (unconditional jump), it will pass control to the point after second `printf()` and before `XOR EAX, EAX` instruction, which implement `return 0`.

So, it can be said that comparing a value with another is *usually* implemented by `CMP`/`Jcc` instructions pair, where *cc* is *condition code*. `CMP` comparing two values and set processor flags[4]. `Jcc` check flags needed to be checked and pass control to mentioned address (or not pass).

But in fact, this could be perceived paradoxical, but `CMP` instruction is in fact `SUB` (subtract). All arithmetic instructions set processor flags too, not only `CMP`. If we compare 1 and 1, $1 - 1$ will be 0 in result, `ZF` flag will be set (meaning the last result was 0). There is no any other circumstances when it is possible except when operands are equal. `JNE` checks only `ZF` flag and jumping only if it is not set. `JNE` is in fact a synonym of `JNZ` (*Jump if Not Zero*) instruction. Assembler translating both `JNE` and `JNZ` instructions into one single opcode. So, `CMP` instruction can be replaced to `SUB` instruction and almost everything will be fine, but the difference is in the `SUB` alter the value of the first operand. `CMP` is *"SUB without saving result"*.

6.6.2 MSVC: x86: IDA

It's time to run IDA and try to do something in it. By the way, it is good idea to use `/MD` option in MSVC for beginners: this mean that all these standard functions will not be linked with executable file, but will be imported from the `MSVCR*.DLL` file instead. Thus it will be easier to see which standard function used and where.

While analysing code in IDA, it is very advisable to do notes for oneself (and others). For example, analysing this example, we see that `JNZ` will be triggered in case of error. So it's possible to move cursor to the label, press "n" and rename it to "error". Another label—into "exit". What I've got:

```
.text:00401000 _main proc near
.text:00401000
.text:00401000 var_4 = dword ptr -4
```

[4] About x86 flags, see also: http://en.wikipedia.org/wiki/FLAGS_register_(computing).

```
.text:00401000 argc  = dword ptr  8
.text:00401000 argv  = dword ptr  0Ch
.text:00401000 envp  = dword ptr  10h
.text:00401000
.text:00401000       push    ebp
.text:00401001       mov     ebp, esp
.text:00401003       push    ecx
.text:00401004       push    offset Format ; "Enter X:\n"
.text:00401009       call    ds:printf
.text:0040100F       add     esp, 4
.text:00401012       lea     eax, [ebp+var_4]
.text:00401015       push    eax
.text:00401016       push    offset aD ; "%d"
.text:0040101B       call    ds:scanf
.text:00401021       add     esp, 8
.text:00401024       cmp     eax, 1
.text:00401027       jnz     short error
.text:00401029       mov     ecx, [ebp+var_4]
.text:0040102C       push    ecx
.text:0040102D       push    offset aYou ; "You entered %d...\n"
.text:00401032       call    ds:printf
.text:00401038       add     esp, 8
.text:0040103B       jmp     short exit
.text:0040103D
.text:0040103D error: ; CODE XREF: _main+27
.text:0040103D       push    offset aWhat ; "What you entered? ↙
    ↳ Huh?\n"
.text:00401042       call    ds:printf
.text:00401048       add     esp, 4
.text:0040104B
.text:0040104B exit: ; CODE XREF: _main+3B
.text:0040104B       xor     eax, eax
.text:0040104D       mov     esp, ebp
.text:0040104F       pop     ebp
.text:00401050       retn
.text:00401050 _main endp
```

Now it's slightly easier to understand the code. However, it's not good idea to comment every instruction excessively.

A part of function can also be hidden in IDA: a block should be marked, then "-" on numerical pad is pressed and text to be entered.

I've hide two parts and gave names to them:

```
.text:00401000 _text segment para public 'CODE' use32
```

```
.text:00401000       assume cs:_text
.text:00401000       ;org 401000h
.text:00401000 ; ask for X
.text:00401012 ; get X
.text:00401024       cmp  eax, 1
.text:00401027       jnz  short error
.text:00401029 ; print result
.text:0040103B       jmp  short exit
.text:0040103D
.text:0040103D error: ; CODE XREF: _main+27
.text:0040103D       push offset aWhat ; "What you entered? Huh↙
    ↳ ?\n"
.text:00401042       call ds:printf
.text:00401048       add  esp, 4
.text:0040104B
.text:0040104B exit: ; CODE XREF: _main+3B
.text:0040104B       xor  eax, eax
.text:0040104D       mov  esp, ebp
.text:0040104F       pop  ebp
.text:00401050       retn
.text:00401050 _main endp
```

To unhide these parts, “+” on numerical pad can be used.

By pressing “space”, we can see how IDA can represent a function as a graph: fig. 6.7. There are two arrows after each conditional jump: green and red. Green arrow pointing to the block which will be executed if jump is triggered, and red if otherwise.

It is possible to fold nodes is this mode and give them names as well (“group nodes”). I did it for 3 blocks: fig. 6.8.

It’s very useful. It can be said, a very important part of reverse engineer’s job is to reduce information he/she have.

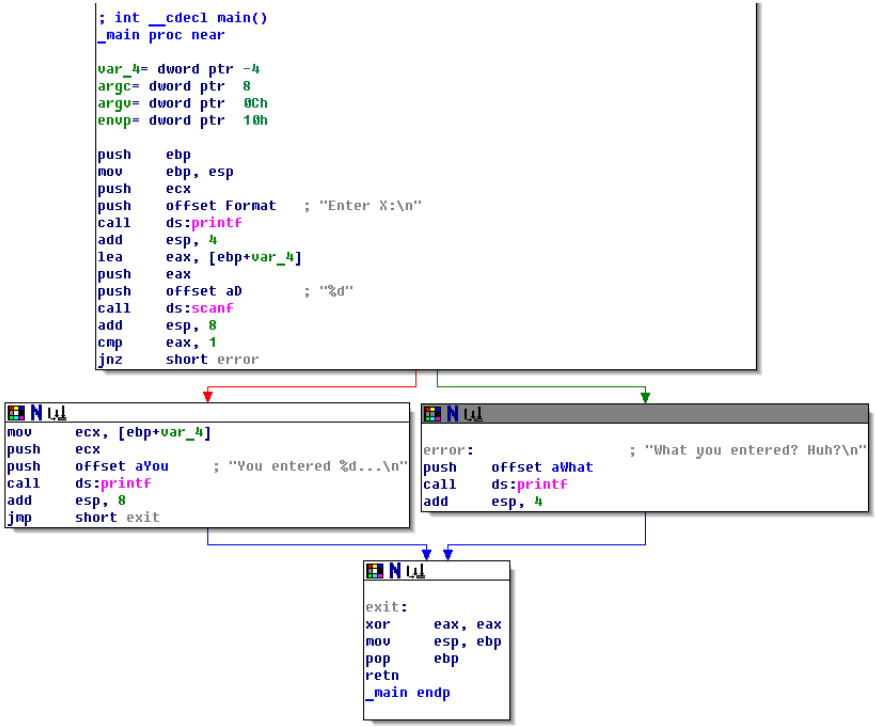


Figure 6.7: Graph mode in IDA

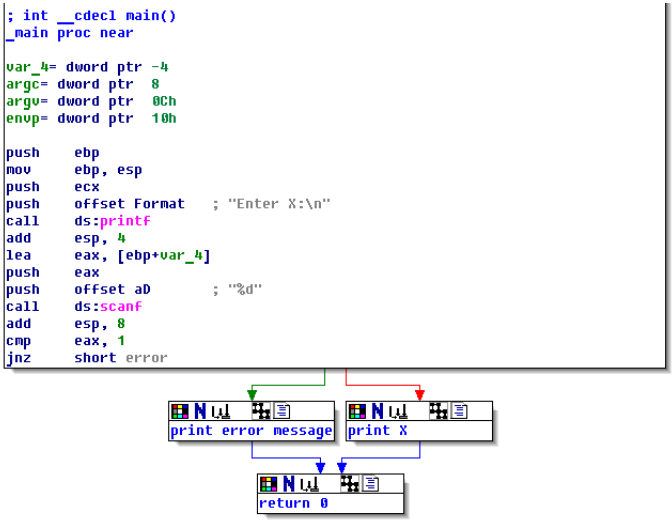


Figure 6.8: Graph mode in IDA with 3 nodes folded

6.6.3 MSVC: x86 + OllyDbg

Let's try to hack our program in OllyDbg, forcing it to think `scanf()` working always without error.

When address of local variable is passed into `scanf()`, initially this variable contain some random garbage, that is `0x4CD478` in case: fig. 6.10.

When `scanf()` is executing, I enter in the console something definitely not a number, like "asdasd". `scanf()` finishing with 0 in `EAX`, which mean, an error occurred: fig. 6.11.

We can also see to the local variable in the stack and notice that it's not changed. Indeed, what `scanf()` would write there? It just did nothing except returning zero.

Now let's try to "hack" our program. Let's right-click on `EAX`, there will also be "Set to 1" among other options. This is what we need.

1 now in `EAX`, so the following check will executed as we need, and `printf()` will print value of variable in the stack.

Let's run (F9) and we will see this in console window:

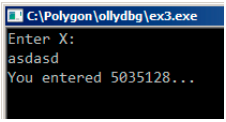


Figure 6.9: console window

Indeed, 5035128 is a decimal representation of the number in stack (0x4CD478)!

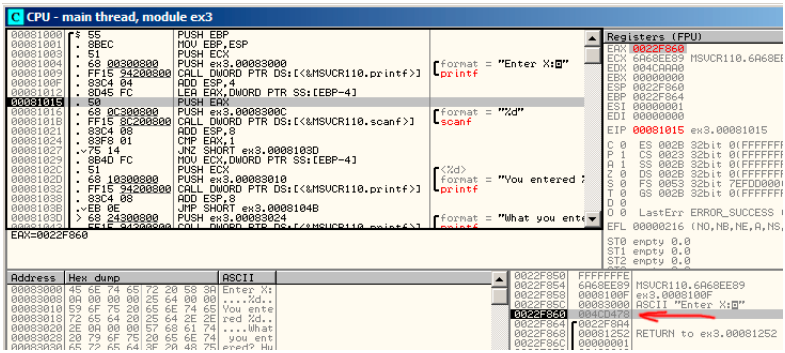


Figure 6.10: OllyDbg: passing variable address into `scanf()`

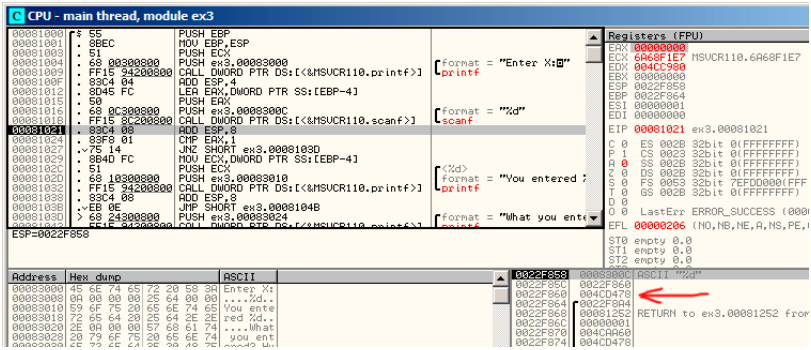


Figure 6.11: OllyDbg: `scanf()` returning error

6.6.4 MSVC: x86 + Hiew

This can be also a simple example of executable file patching. We may try to patch executable, so the program will always print numbers, no matter what we entered.

Assuming the executable compiled against external `MSVCR*.DLL` (i.e., with `/MD` option)[5], we may find `main()` function at the very beginning of `.text` section. Let's open executable in Hiew, find the very beginning of `.text` section (Enter, F8, F6, Enter, Enter).

We will see this: fig. 6.12.

Hiew finds ASCIIZ[6] strings and displays them, as well as imported function names.

[5] that's what also called "dynamic linking"

[6] ASCII Zero (null-terminated ASCII string)

Move cursor to the address .00401027 (with the JNZ instruction we should bypass), press F3, and then type "9090" (meaning two NOP[7]-s): fig. 6.13.

Then F9 (update). Now the executable saved to disk. It will behave as we wanted.

Two NOP-s are probably not quite æsthetically as it could be. Other way to patch this instruction is to write just 0 to the second opcode byte (jump offset), so that JNZ will always jump to the next instruction.

We can do the opposite: replace first byte to EB while not touching the second byte (jump offset). We'll got here always triggered unconditional jump. The error message will be printed always, no matter what number was entered.

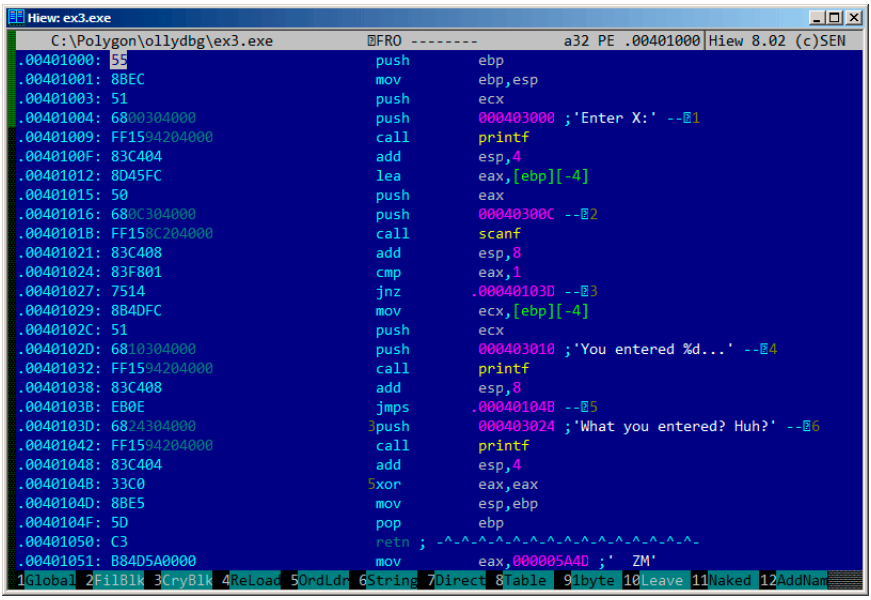


Figure 6.12: Hiew: main() function

[7]No OPeration

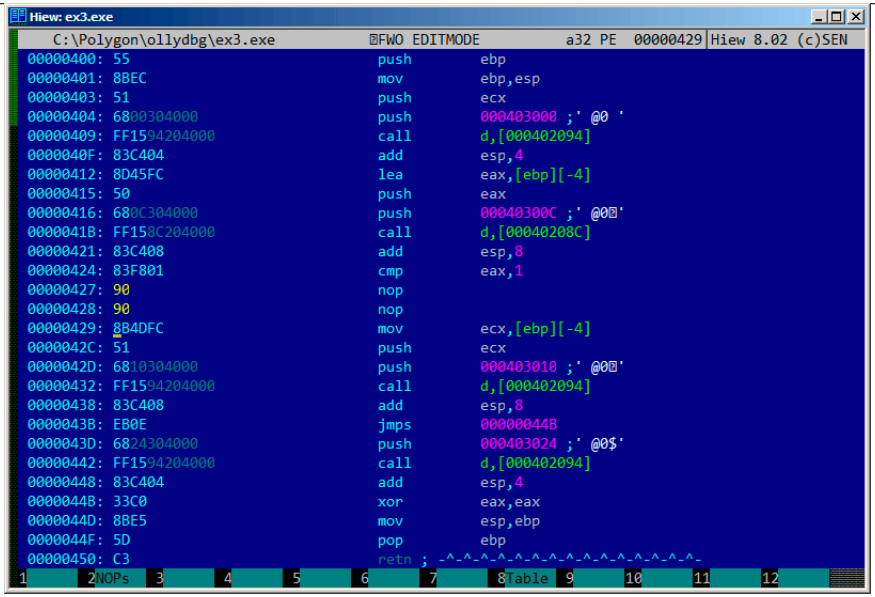


Figure 6.13: Hiew: replacing JNZ by two NOP-s

6.6.5 GCC: x86

Code generated by GCC 4.4.1 in Linux is almost the same, except differences we already considered.

6.6.6 MSVC: x64

Since we work here with *int*-typed variables, which are still 32-bit in x86-64, we see how 32-bit part of registers (prefixed with E-) are used here as well. While working with pointers, however, 64-bit register parts are used, prefied with R-.

Listing 6.4: MSVC 2012 x64

```
_DATA   SEGMENT
$SG2924 DB      'Enter X:', 0aH, 00H
$SG2926 DB      '%d', 00H
$SG2927 DB      'You entered %d...', 0aH, 00H
$SG2929 DB      'What you entered? Huh?', 0aH, 00H
_DATA   ENDS

_TEXT   SEGMENT
x$ = 32
main    PROC
```

```
$LN5:
        sub     rsp, 56
        lea     rcx, OFFSET FLAT:$SG2924 ; 'Enter X:'
        call    printf
        lea     rdx, QWORD PTR x$[rsp]
        lea     rcx, OFFSET FLAT:$SG2926 ; '%d'
        call    scanf
        cmp     eax, 1
        jne     SHORT $LN2@main
        mov     edx, DWORD PTR x$[rsp]
        lea     rcx, OFFSET FLAT:$SG2927 ; 'You entered %d...'
        call    printf
        jmp     SHORT $LN1@main
$LN2@main:
        lea     rcx, OFFSET FLAT:$SG2929 ; 'What you entered? ↙
    ↳ Huh?'
        call    printf
$LN1@main:
        ; return 0
        xor     eax, eax
        add     rsp, 56
        ret     0
main    ENDP
_TEXT   ENDS
END
```

6.6.7 ARM: Optimizing Keil + thumb mode

Listing 6.5: Optimizing Keil + thumb mode

```
var_8           = -8

                PUSH    {R3,LR}
                ADR     R0, aEnterX     ; "Enter X:\n"
                BL      __2printf
                MOV     R1, SP
                ADR     R0, aD          ; "%d"
                BL      __0scanf
                CMP     R0, #1
                BEQ     loc_1E
                ADR     R0, aWhatYouEntered ; "What you entered?↙
    ↳  Huh?\n"
                BL      __2printf
```

```
loc_1A                                  ; CODE XREF: main+26
                MOVS    R0, #0
                POP     {R3,PC}

loc_1E                                  ; CODE XREF: main+12
                LDR     R1, [SP,#8+var_8]
                ADR     R0, aYouEnteredD___ ; "You entered %d↙
    ↳ ...\n"
                BL      __2printf
                B       loc_1A
```

New instructions here are CMP and BEQ[8].

CMP is akin to the x86 instruction bearing the same name, it subtracts one argument from another and saves flags.

BEQ is jumping to another address if operands while comparing were equal to each other, or, if result of last computation was 0, or if Z flag is 1. Same thing as JZ in x86.

Everything else is simple: execution flow is forking into two branches, then the branches are converging at the point where 0 is written into the R0, as a value returned from the function, and then function finishing.

[8] (PowerPC, ARM) Branch if Equal

Chapter 7

Accessing passed arguments

Now we figured out the caller function passing arguments to the callee via stack. But how callee access them?

Listing 7.1: simple example

```
#include <stdio.h>

int f (int a, int b, int c)
{
        return a*b+c;
};

int main()
{
        printf ("%d\n", f(1, 2, 3));
        return 0;
};
```

7.1 x86

7.1.1 MSVC

What we have after compilation (MSVC 2010 Express):

Listing 7.2: MSVC 2010 Express

```
_TEXT   SEGMENT
_a$ = 8                                                 ; size =↙
    ↳  4
```

```
_b$ = 12                                                    ; size =↙
    ↳ 4
_c$ = 16                                                    ; size =↙
    ↳ 4
_f      PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _a$[ebp]
        imul    eax, DWORD PTR _b$[ebp]
        add     eax, DWORD PTR _c$[ebp]
        pop     ebp
        ret     0
_f      ENDP

_main   PROC
        push    ebp
        mov     ebp, esp
        push    3 ; 3rd argument
        push    2 ; 2nd argument
        push    1 ; 1st argument
        call    _f
        add     esp, 12
        push    eax
        push    OFFSET $SG2463 ; '%d', 0aH, 00H
        call    _printf
        add     esp, 8
        ; return 0
        xor     eax, eax
        pop     ebp
        ret     0
_main   ENDP
```

What we see is the 3 numbers are pushing to stack in function `main()` and `f(int,int,int)` is called then. Argument access inside `f()` is organized with the help of macros like: `_a$ = 8`, in the same way as local variables accessed, but the difference in that these offsets are positive (addressed with *plus* sign). So, adding `_a$` macro to the value in the `EBP` register, *outer* side of stack frame is addressed.

Then `a` value is stored into `EAX`. After `IMUL` instruction execution, value in the `EAX` is a product[1] of value in `EAX` and what is stored in `_b`. After `IMUL` execution, `ADD` is summing value in `EAX` and what is stored in `_c`. Value in the `EAX` is not needed to be moved: it is already in place it must be. Now return to caller —it will take value from the `EAX` and used it as `printf()` argument.

[1] result of multiplication

7.1.2 MSVC + OllyDbg

Let's illustrate this in OllyDbg. When we trace until the very first instruction in `f()` that uses one of the arguments (first one), we see that EBP is pointing to the stack frame, I marked its begin with red arrow. The first element of stack frame is saved EBP value, second is RA, third is first function argument, then second argument and third one. To access the first function argument, one need to add exactly 8 (2 32-bit words) to EBP.

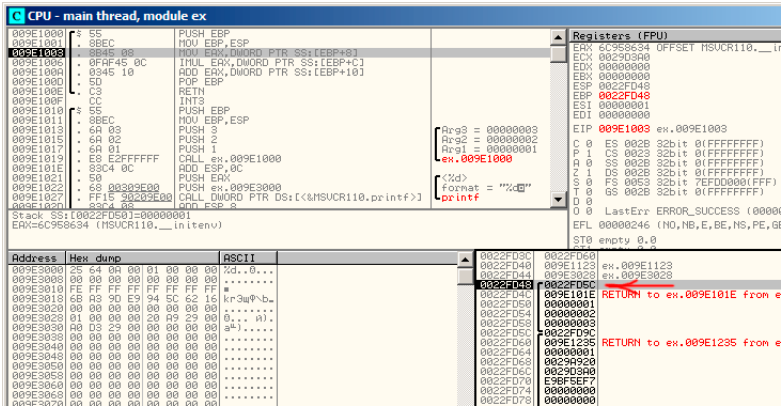


Figure 7.1: OllyDbg: inside of f() function

7.1.3 GCC

Let's compile the same in GCC 4.4.1 and let's see results in IDA:

Listing 7.3: GCC 4.4.1

```
                public f
f               proc near

arg_0           = dword ptr  8
arg_4           = dword ptr  0Ch
arg_8           = dword ptr  10h

                push    ebp
                mov     ebp, esp
                mov     eax, [ebp+arg_0] ; 1st argument
                imul    eax, [ebp+arg_4] ; 2nd argument
                add     eax, [ebp+arg_8] ; 3rd argument
                pop     ebp
                retn
```

```
f               endp

                public main
main            proc near

var_10          = dword ptr -10h
var_C           = dword ptr -0Ch
var_8           = dword ptr -8

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 10h
                mov     [esp+10h+var_8], 3 ; 3rd argument
                mov     [esp+10h+var_C], 2 ; 2nd argument
                mov     [esp+10h+var_10], 1 ; 1st argument
                call    f
                mov     edx, offset aD  ; "%d\n"
                mov     [esp+10h+var_C], eax
                mov     [esp+10h+var_10], edx
                call    _printf
                mov     eax, 0
                leave
                retn
main            endp
```

Almost the same result.

The stack pointer is not returning back after both function exeuction, because penultimate `LEAVE` (B.6.2) instruction will do this, at the end.

7.2 x64

The story is a bit different in x86-64, function arguments (4 or 6) are passed in registers, and a callee reading them from there instead of stack accessing.

7.2.1 MSVC

Optimizing MSVC:

Listing 7.4: MSVC 2012 /Ox x64

```
$SG2997 DB      '%d', 0aH, 00H

main    PROC
```

```
        sub     rsp, 40
        mov     edx, 2
        lea     r8d, QWORD PTR [rdx+1] ; R8D=3
        lea     ecx, QWORD PTR [rdx-1] ; ECX=1
        call    f
        lea     rcx, OFFSET FLAT:$SG2997 ; '%d'
        mov     edx, eax
        call    printf
        xor     eax, eax
        add     rsp, 40
        ret     0
main    ENDP

f       PROC
        ; ECX - 1st argument
        ; EDX - 2nd argument
        ; R8D - 3rd argument
        imul    ecx, edx
        lea     eax, DWORD PTR [r8+rcx]
        ret     0
f       ENDP
```

As we can see, very compact `f()` function takes arguments right from the registers. `LEA` instruction is used here for addition, apparently, compiler considered this instruction here faster then `ADD`. `LEA` is also used in `main()` for the first and third arguments preparing, apparently, compiler thinks that it will work faster than usual value loading to the register using `MOV` instruction.

Let's try to take a look on output of non-optimizing MSVC:

Listing 7.5: MSVC 2012 x64

```
f               proc near

; shadow space:
arg_0           = dword ptr  8
arg_8           = dword ptr  10h
arg_10          = dword ptr  18h

                ; ECX - 1st argument
                ; EDX - 2nd argument
                ; R8D - 3rd argument
                mov     [rsp+arg_10], r8d
                mov     [rsp+arg_8], edx
                mov     [rsp+arg_0], ecx
                mov     eax, [rsp+arg_0]
```

```
                imul    eax, [rsp+arg_8]
                add     eax, [rsp+arg_10]
                retn
f               endp

main            proc near
                sub     rsp, 28h
                mov     r8d, 3 ; 3rd argument
                mov     edx, 2 ; 2nd argument
                mov     ecx, 1 ; 1st argument
                call    f
                mov     edx, eax
                lea     rcx, $SG2931    ; "%d\n"
                call    printf

                ; return 0
                xor     eax, eax
                add     rsp, 28h
                retn
main            endp
```

Somewhat puzzling: all 3 arguments from registers are saved to the stack for some reason. This is called "shadow space" [2]: every Win64 may (but not required to) save all 4 register values there. This is done by two reasons: 1) it is too lavish to allocate the whole register (or even 4 registers) for the input argument, so it will be accessed via stack; 2) debugger is always aware where to find function arguments at a break [3].

It is duty of caller to allocate "shadow space" in stack.

7.2.2 GCC

Optimizing GCC does more or less understanable code:

Listing 7.6: GCC 4.4.6 -O3 x64

```
f:
        ; EDI - 1st argument
        ; ESI - 2nd argument
        ; EDX - 3rd argument
        imul    esi, edi
        lea     eax, [rdx+rsi]
        ret
```

[2] http://msdn.microsoft.com/en-us/library/zthk2dkh(v=vs.80).aspx
[3] http://msdn.microsoft.com/en-us/library/ew5tede7(v=VS.90).aspx

```
main:
        sub     rsp, 8
        mov     edx, 3
        mov     esi, 2
        mov     edi, 1
        call    f
        mov     edi, OFFSET FLAT:.LC0 ; "%d\n"
        mov     esi, eax
        xor     eax, eax  ; number of vector registers passed
        call    printf
        xor     eax, eax
        add     rsp, 8
        ret
```

Non-optimizing GCC:

Listing 7.7: GCC 4.4.6 x64

```
f:
        ; EDI - 1st argument
        ; ESI - 2nd argument
        ; EDX - 3rd argument
        push    rbp
        mov     rbp, rsp
        mov     DWORD PTR [rbp-4], edi
        mov     DWORD PTR [rbp-8], esi
        mov     DWORD PTR [rbp-12], edx
        mov     eax, DWORD PTR [rbp-4]
        imul    eax, DWORD PTR [rbp-8]
        add     eax, DWORD PTR [rbp-12]
        leave
        ret
main:
        push    rbp
        mov     rbp, rsp
        mov     edx, 3
        mov     esi, 2
        mov     edi, 1
        call    f
        mov     edx, eax
        mov     eax, OFFSET FLAT:.LC0 ; "%d\n"
        mov     esi, edx
        mov     rdi, rax
        mov     eax, 0  ; number of vector registers passed
```

```
        call    printf
        mov     eax, 0
        leave
        ret
```

There are no "shadow space" requirement in System V *NIX [21], but callee may need to save arguments somewhere, because, again, it may be regsiters shortage.

7.2.3 GCC: uint64_t instead int

Our example worked with 32-bit *int*, that is why 32-bit register parts were used (prefixed by E-).

It can be altered slightly in order to use 64-bit values:

```
#include <stdio.h>
#include <stdint.h>

uint64_t f (uint64_t a, uint64_t b, uint64_t c)
{
        return a*b+c;
};

int main()
{
        printf ("%lld\n", f(0x1122334455667788,
                            0x1111111122222222,
                            0x3333333344444444));
        return 0;
};
```

Listing 7.8: GCC 4.4.6 -O3 x64

```
f               proc near
                imul    rsi, rdi
                lea     rax, [rdx+rsi]
                retn
f               endp

main            proc near
                sub     rsp, 8
                mov     rdx, 3333333344444444h ; 3rd argument
                mov     rsi, 1111111122222222h ; 2nd argument
                mov     rdi, 1122334455667788h ; 1st argument
                call    f
                mov     edi, offset format ; "%lld\n"
```

```
                mov     rsi, rax
                xor     eax, eax ; number of vector registers ↙
    ↳ passed
                call    _printf
                xor     eax, eax
                add     rsp, 8
                retn
main            endp
```

The code is very same, but registers (prefixed by R-) are *used as a whole*.

7.3 ARM

7.3.1 Non-optimizing Keil + ARM mode

```
.text:000000A4 00 30 A0 E1                  MOV     R3, R0
.text:000000A8 93 21 20 E0                  MLA     R0, R3, R1, ↙
    ↳ R2
.text:000000AC 1E FF 2F E1                  BX      LR
...
.text:000000B0                 main
.text:000000B0 10 40 2D E9                  STMFD   SP!, {R4,LR}
.text:000000B4 03 20 A0 E3                  MOV     R2, #3
.text:000000B8 02 10 A0 E3                  MOV     R1, #2
.text:000000BC 01 00 A0 E3                  MOV     R0, #1
.text:000000C0 F7 FF FF EB                  BL      f
.text:000000C4 00 40 A0 E1                  MOV     R4, R0
.text:000000C8 04 10 A0 E1                  MOV     R1, R4
.text:000000CC 5A 0F 8F E2                  ADR     R0, aD_0      ↙
    ↳     ; "%d\n"
.text:000000D0 E3 18 00 EB                  BL      __2printf
.text:000000D4 00 00 A0 E3                  MOV     R0, #0
.text:000000D8 10 80 BD E8                  LDMFD   SP!, {R4,PC}
```

In `main()` function, two other functions are simply called, and three values are passed to the first one (`f`).

As I mentioned before, in ARM, first 4 values are usually passed in first 4 registers (R0-R3).

`f`function, as it seems, use first 3 registers (R0-R2) as arguments.

`MLA` (*Multiply Accumulate*) instruction multiplicates two first operands (R3 and R1), adds third operand (R2) to product and places result into zeroth operand (R0), via which, by standard, values are returned from functions.

Multiplication and addition at once[4] (*Fused multiply–add*) is very useful operation, by the way, there is no such instruction in x86, if not to count new FMA-instruction[5] in SIMD.

The very first `MOV R3, R0`, instruction, apparently, redundant (single `MLA` instruction could be used here instead), compiler was not optimized it, since this is non-optimizing compilation.

`BX` instruction returns control to the address stored in the LR register and, if it is necessary, switches processor mode from thumb to ARM or vice versa. This can be necessary since, as we can see, `f` function is not aware, from which code it may be called, from ARM or thumb. This, if it will be called from thumb code, `BX` will not only return control to the calling function, but also will switch processor mode to thumb mode. Or not switch, if the function was called from ARM code.

7.3.2 Optimizing Keil + ARM mode

```
.text:00000098                 f
.text:00000098 91 20 20 E0                 MLA     R0, R1, R0, ↙
    ↳ R2
.text:0000009C 1E FF 2F E1                 BX      LR
```

And here is `f` function compiled by Keil compiler in full optimization mode (`-O3`). `MOV` instruction was optimized (or reduced) and now `MLA` uses all input registers and also places result right into `R0`, exactly where calling function will read it and use.

7.3.3 Optimizing Keil + thumb mode

```
.text:0000005E 48 43                       MULS    R0, R1
.text:00000060 80 18                       ADDS    R0, R0, R2
.text:00000062 70 47                       BX      LR
```

`MLA` instruction is not available in thumb mode, so, compiler generates the code doing these two operations separately. First `MULS` instruction multiply `R0` by `R1` leaving result in the `R1` register. Second (`ADDS`) instruction adds result and `R2` leaving result in the `R0` register.

[4] wikipedia: Multiply–accumulate operation

[5] https://en.wikipedia.org/wiki/FMA_instruction_set

Chapter 8

One more word about results returning.

As of x86, function execution result is usually returned [1] in the EAX register. If it is byte type or character (*char*) —then in the lowest register EAX part —AL. If function returns *float* number, the FPU register ST(0) is to be used instead. In ARM, result is usually returned in the R0 register.

By the way, what if returning value of the main() function will be declared not as *int* but as *void*?

so-called startup-code is calling main() roughly as:

```
push envp
push argv
push argc
call main
push eax
call exit
```

In other words:

```
exit(main(argc,argv,envp));
```

If you declare main() as *void* and nothing will be returned explicitly (by *return* statement), then something random, that was stored in the EAX register at the moment of the main() finish, will come into the sole exit() function argument. Most likely, there will be a random value, leaved from your function execution. So, exit

[1] See also: MSDN: Return Values (C++): http://msdn.microsoft.com/en-us/library/7572ztz4.aspx

code of program will be pseudorandom.

I can illustrate this fact. Please notice, the `main()` function has *void* type:

```
#include <stdio.h>

void main()
{
        printf ("Hello, world!\n");
};
```

Let's compile it in Linux.

GCC 4.8.1 replaced `printf()` to `puts()` (we saw this before: 2.3.3), but that's OK, since `puts()` returns number of characters printed, just like `printf()`. Please notice that `EAX` is not zeroed before `main()` finish. This means, `EAX` value at the `main()` finish will contain what `puts()` leaved there.

Listing 8.1: GCC 4.8.1

```
.LC0:
        .string "Hello, world!"
main:
        push    ebp
        mov     ebp, esp
        and     esp, -16
        sub     esp, 16
        mov     DWORD PTR [esp], OFFSET FLAT:.LC0
        call    puts
        leave
        ret
```

Let' s write bash script, showing exit status:

Listing 8.2: tst.sh

```
#!/bin/sh
./hello_world
echo $?
```

And run it:

```
$ tst.sh
Hello, world!
14
```

14 is a number of characters printed.

Let's back to the fact the returning value is leaved in the EAX register. That is why old C compilers cannot create functions capable of returning something not fitting in one register (usually type *int*) but if one needs it, one should return information via pointers passed in function arguments. Now it is possible, to return, let's say, whole structure, but still it is not very popular. If function must return a large structure, caller must allocate it and pass pointer to it via first argument, transparently for programmer. That is almost the same as to pass pointer in first argument manually, but compiler hide this.

Small example:

```
struct s
{
    int a;
    int b;
    int c;
};

struct s get_some_values (int a)
{
    struct s rt;

    rt.a=a+1;
    rt.b=a+2;
    rt.c=a+3;

    return rt;
};
```

...what we got (MSVC 2010 /Ox):

```
$T3853 = 8                          ; size = 4
_a$ = 12                            ; size = 4
?get_some_values@@YA?AUs@@H@Z PROC            ; get_some_values
    mov    ecx, DWORD PTR _a$[esp-4]
    mov    eax, DWORD PTR $T3853[esp-4]
    lea    edx, DWORD PTR [ecx+1]
    mov    DWORD PTR [eax], edx
    lea    edx, DWORD PTR [ecx+2]
    add    ecx, 3
    mov    DWORD PTR [eax+4], edx
    mov    DWORD PTR [eax+8], ecx
    ret    0
?get_some_values@@YA?AUs@@H@Z ENDP            ; get_some_values
```

Macro name for internal variable passing pointer to structure is $T3853 here.

This example can be rewritten using C99 language extensions:

```
struct s
{
    int a;
    int b;
    int c;
};

struct s get_some_values (int a)
{
    return (struct s){.a=a+1, .b=a+2, .c=a+3};
};
```

Listing 8.3: GCC 4.8.1

```
_get_some_values proc near

ptr_to_struct   = dword ptr  4
a               = dword ptr  8

                mov     edx, [esp+a]
                mov     eax, [esp+ptr_to_struct]
                lea     ecx, [edx+1]
                mov     [eax], ecx
                lea     ecx, [edx+2]
                add     edx, 3
                mov     [eax+4], ecx
                mov     [eax+8], edx
                retn
_get_some_values endp
```

As we may see, the function is just filling fields in the structure, allocated by caller function. So there are no performance drawbacks.

Chapter 9

Pointers

Pointers are often used to return values from function (recall `scanf()` case (6)). For example, when function should return two values.

9.1 Global variables example

```
#include <stdio.h>

void f1 (int x, int y, int *sum, int *product)
{
        *sum=x+y;
        *product=x*y;
};

int sum, product;

void main()
{
        f1(123, 456, &sum, &product);
        printf ("sum=%d, product=%d\n", sum, product);
};
```

This compiling into:

Listing 9.1: Optimizing MSVC 2010 (/Ox /Ob0)

```
COMM    _product:DWORD
COMM    _sum:DWORD
$SG2803 DB      'sum=%d, product=%d', 0aH, 00H
```

```
_x$ = 8                                                 ; size =↙
    ↳ 4
_y$ = 12                                                ; size =↙
    ↳ 4
_sum$ = 16                                              ; size =↙
    ↳ 4
_product$ = 20                                          ; size =↙
    ↳ 4
_f1     PROC
        mov     ecx, DWORD PTR _y$[esp-4]
        mov     eax, DWORD PTR _x$[esp-4]
        lea     edx, DWORD PTR [eax+ecx]
        imul    eax, ecx
        mov     ecx, DWORD PTR _product$[esp-4]
        push    esi
        mov     esi, DWORD PTR _sum$[esp]
        mov     DWORD PTR [esi], edx
        mov     DWORD PTR [ecx], eax
        pop     esi
        ret     0
_f1     ENDP

_main   PROC
        push    OFFSET _product
        push    OFFSET _sum
        push    456                                     ; 000001↙
    ↳ c8H
        push    123                                     ; ↙
    ↳ 0000007bH
        call    _f1
        mov     eax, DWORD PTR _product
        mov     ecx, DWORD PTR _sum
        push    eax
        push    ecx
        push    OFFSET $SG2803
        call    DWORD PTR __imp__printf
        add     esp, 28                                 ; ↙
    ↳ 0000001cH
        xor     eax, eax
        ret     0
_main   ENDP
```

Let's see this in OllyDbg: fig. 9.1. At first, global variables addresses are passed into `f1()`. We can click "Follow in dump" on the stack element, and we will see a place in

data segment allocated for two variables. These variables are cleared, because non-initialized data (BSS[1]) are cleared before execution begin. They are residing in data segment, we can be sure it is so, by pressing Alt-M and seeing memory map: fig. 9.5.

Let's trace (F7) until execution of `f1()` fig. 9.2. Two values are seen in the stack 456 (0x1C8) and 123 (0x7B), and two global variables addresses as well.

Let's trace until the end of `f1()`. At the window at left we see how calculation results are appeared in the gloval variables fig. 9.3.

Now values of global variables are loaded into registers for passing into `printf()`: fig. 9.4.

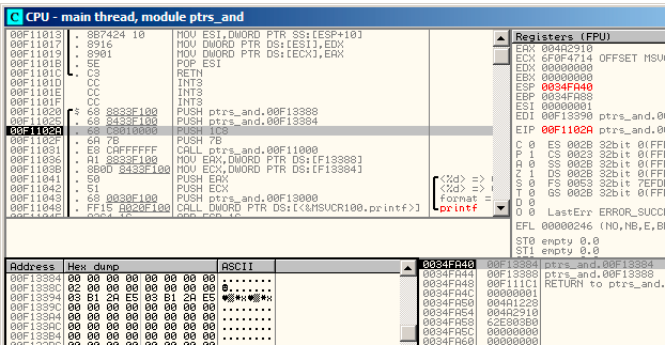


Figure 9.1: OllyDbg: global variables addresses are passing into `f1()`

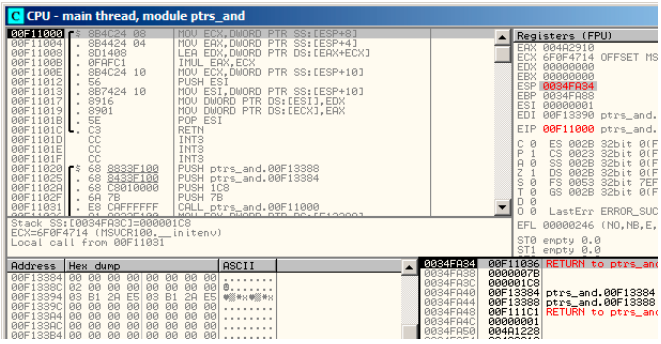


Figure 9.2: OllyDbg: `f1()` is started

[1]Block Started by Symbol

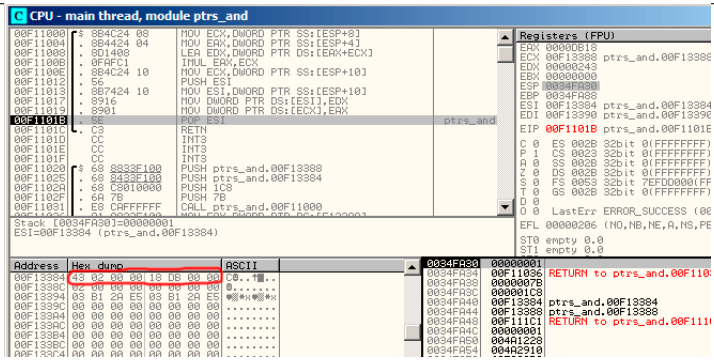


Figure 9.3: OllyDbg: `f1()` finishes

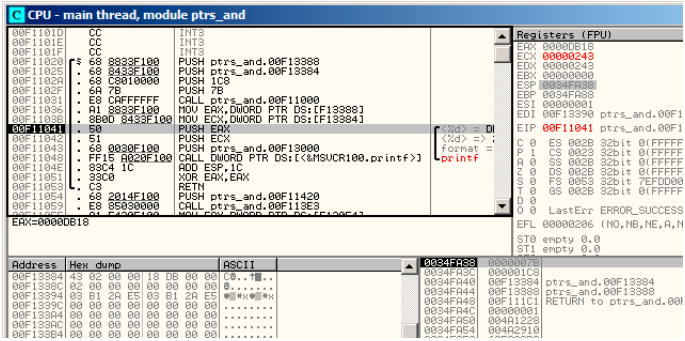


Figure 9.4: OllyDbg: global variables addresses are passed into `printf()`

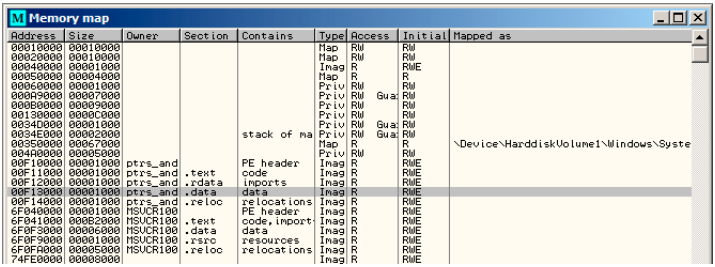


Figure 9.5: OllyDbg: memory map

9.2 Local variables example

Let's rework our example slightly:

Listing 9.2: now variables are local

```
void main()
{
        int sum, product; // now variables are here

        f1(123, 456, &sum, &product);
        printf ("sum=%d, product=%d\n", sum, product);
};
```

`f1()` function code will not changed. Only `main()` code will:

Listing 9.3: Optimizing MSVC 2010 (/Ox /Ob0)

```
_product$ = -8                                          ; size =↙
    ↳  4
_sum$ = -4                                              ; size =↙
    ↳  4
_main   PROC
; Line 10
        sub     esp, 8
; Line 13
        lea     eax, DWORD PTR _product$[esp+8]
        push    eax
        lea     ecx, DWORD PTR _sum$[esp+12]
        push    ecx
        push    456                                     ; 000001↙
    ↳ c8H
        push    123                                     ; ↙
    ↳ 0000007bH
        call    _f1
; Line 14
        mov     edx, DWORD PTR _product$[esp+24]
        mov     eax, DWORD PTR _sum$[esp+24]
        push    edx
        push    eax
        push    OFFSET $SG2803
        call    DWORD PTR __imp__printf
; Line 15
        xor     eax, eax
        add     esp, 36                                 ; ↙
    ↳ 00000024H
```

```
        ret     0
```

Let's again take a look into OllyDbg. Local variable addresses in the stack are 0x35FCF4 and 0x35FCF8. We see how these are pushed into the stack: fig. 9.6.

f1() is started. Random garbage are at 0x35FCF4 and 0x35FCF8 so far fig. 9.7.

f1() finished. There are 0xDB18 and 0x243 now at 0x35FCF4 and 0x35FCF8 addresses, these values are f1() function result.

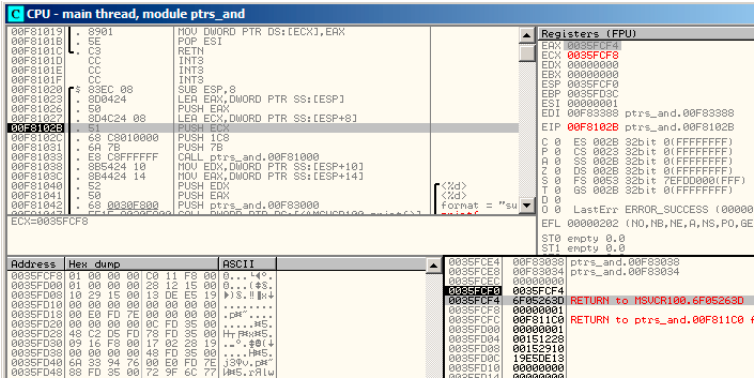


Figure 9.6: OllyDbg: addresses of local variables are pushed into the stack

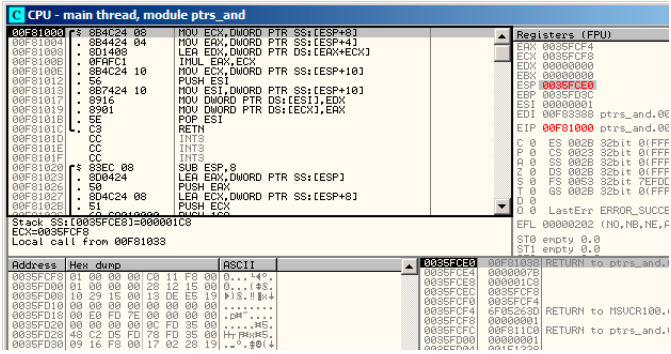


Figure 9.7: OllyDbg: f1() starting

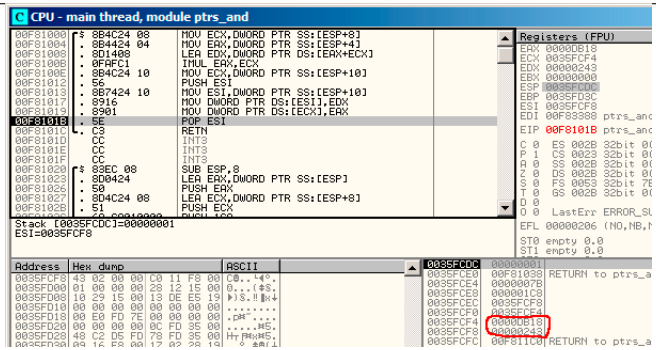


Figure 9.8: OllyDbg: `f1()` finished

9.3 Conclusion

`f1()` can return results to any place in memory, located anywhere. This is essence and usefulness of pointers.

By the way, C++ *references* works just in the same way. Read more about them: (29.3).

Chapter 10

Conditional jumps

Now about conditional jumps.

```
#include <stdio.h>

void f_signed (int a, int b)
{
    if (a>b)
        printf ("a>b\n");
    if (a==b)
        printf ("a==b\n");
    if (a<b)
        printf ("a<b\n");
};

void f_unsigned (unsigned int a, unsigned int b)
{
    if (a>b)
        printf ("a>b\n");
    if (a==b)
        printf ("a==b\n");
    if (a<b)
        printf ("a<b\n");
};

int main()
{
    f_signed(1, 2);
    f_unsigned(1, 2);
    return 0;
```

```
};
```

10.1 x86

10.1.1 x86 + MSVC

What we have in the `f_signed()` function:

Listing 10.1: Non-optimizing MSVC 2010

```
_a$ = 8
_b$ = 12
_f_signed PROC
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _a$[ebp]
    cmp    eax, DWORD PTR _b$[ebp]
    jle    SHORT $LN3@f_signed
    push   OFFSET $SG737        ; 'a>b'
    call   _printf
    add    esp, 4
$LN3@f_signed:
    mov    ecx, DWORD PTR _a$[ebp]
    cmp    ecx, DWORD PTR _b$[ebp]
    jne    SHORT $LN2@f_signed
    push   OFFSET $SG739        ; 'a==b'
    call   _printf
    add    esp, 4
$LN2@f_signed:
    mov    edx, DWORD PTR _a$[ebp]
    cmp    edx, DWORD PTR _b$[ebp]
    jge    SHORT $LN4@f_signed
    push   OFFSET $SG741        ; 'a<b'
    call   _printf
    add    esp, 4
$LN4@f_signed:
    pop    ebp
    ret    0
_f_signed ENDP
```

First instruction JLE means *Jump if Less or Equal*. In other words, if second operand is larger than first or equal, control flow will be passed to address or label mentioned in instruction. But if this condition will not trigger (second operand less than first), control flow will not be altered and first `printf()` will be called. The second check

is JNE: *Jump if Not Equal*. Control flow will not altered if operands are equals to each other. The third check is JGE: *Jump if Greater or Equal*—jump if the first operand is larger than the second or if they are equals to each other. By the way, if all three conditional jumps are triggered, no `printf()` will be called whatsoever. But, without special intervention, it is nearly impossible.

`f_unsigned()` function is likewise, with the exception the JBE and JAE instructions are used here instead of JLE and JGE, see below about it:

Now let's take a look to the `f_unsigned()` function

Listing 10.2: GCC

```
_a$ = 8   ; size = 4
_b$ = 12  ; size = 4
_f_unsigned PROC
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _a$[ebp]
    cmp    eax, DWORD PTR _b$[ebp]
    jbe    SHORT $LN3@f_unsigned
    push   OFFSET $SG2761    ; 'a>b'
    call   _printf
    add    esp, 4
$LN3@f_unsigned:
    mov    ecx, DWORD PTR _a$[ebp]
    cmp    ecx, DWORD PTR _b$[ebp]
    jne    SHORT $LN2@f_unsigned
    push   OFFSET $SG2763    ; 'a==b'
    call   _printf
    add    esp, 4
$LN2@f_unsigned:
    mov    edx, DWORD PTR _a$[ebp]
    cmp    edx, DWORD PTR _b$[ebp]
    jae    SHORT $LN4@f_unsigned
    push   OFFSET $SG2765    ; 'a<b'
    call   _printf
    add    esp, 4
$LN4@f_unsigned:
    pop    ebp
    ret    0
_f_unsigned ENDP
```

Almost the same, with exception of instructions: JBE—*Jump if Below or Equal* and JAE—*Jump if Above or Equal*. These instructions (JA/JAE/JBE/JBE) are distinct from JG/JGE/JL/JLE in that way, they works with unsigned numbers.

See also section about signed number representations (32). So, where we see usage of JG/JL instead of JA/JBE or otherwise, we can almost be sure about signed or unsigned type of variable.

Here is also main() function, where nothing much new to us:

Listing 10.3: main()

```
_main   PROC
        push    ebp
        mov     ebp, esp
        push    2
        push    1
        call    _f_signed
        add     esp, 8
        push    2
        push    1
        call    _f_unsigned
        add     esp, 8
        xor     eax, eax
        pop     ebp
        ret     0
_main   ENDP
```

10.1.2 x86 + MSVC + OllyDbg

We can see how flags are set by running this example in OllyDbg. Let's begin with f_unsigned() function, which works with unsigned number. CMP executed thrice here, but for the same arguments, so flags will be the same each time.

First comparison results: fig. 10.1. So, the flags are: C=1, P=1, A=1, Z=0, S=1, T=0, D=0, O=0. Flags are named by one characters in OllyDbg for brevity.

OllyDbg gives a hint that (JBE) jump will be triggered. Indeed, if to take a look into [14], we will read there that JBE will trigger if CF=1 or ZF=1. Condition is true here, so jump is triggered.

The next conditional jump:fig. 10.2. OllyDbg gives a hint that JNZ will trigger. Indeed, JNZ will trigger if ZF=0 (zero flag).

The third conditional jump JNB: fig. 10.3. In [14] we may find that JNB will trigger if CF=0 (carry flag). It's not true in our case, so the third printf() will execute.

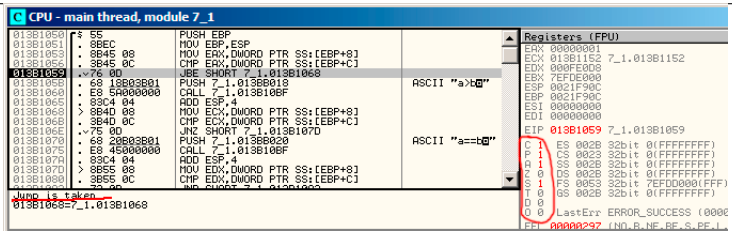


Figure 10.1: OllyDbg: `f_unsigned()`: first conditional jump

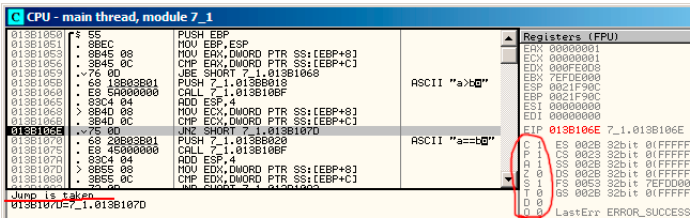


Figure 10.2: OllyDbg: `f_unsigned()`: second conditional jump

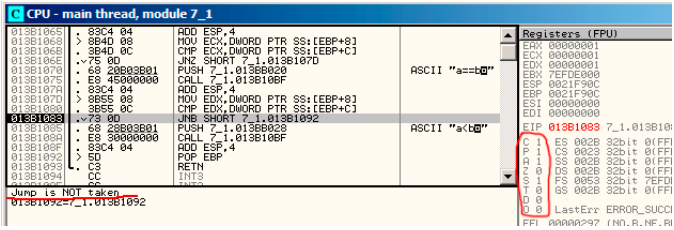


Figure 10.3: OllyDbg: `f_unsigned()`: third conditional jump

Now we can try in OllyDbg the `f_signed()` function working with signed values.

Flags are set in the same way: C=1, P=1, A=1, Z=0, S=1, T=0, D=0, O=0.

The first conditional jump `JLE` will trigger. fig. 10.4. In [14] we may find that this instruction is triggering if ZF=1 or SF≠OF. SF≠OF in our case, so jump is triggering.

The next `JNZ` conditional jump will trigger: it does if ZF=0 (zero flag): fig. 10.5.

The third conditional jump `JGE` will not trigger because it will only if SF=OF, and that is not true in our case: fig. 10.6.

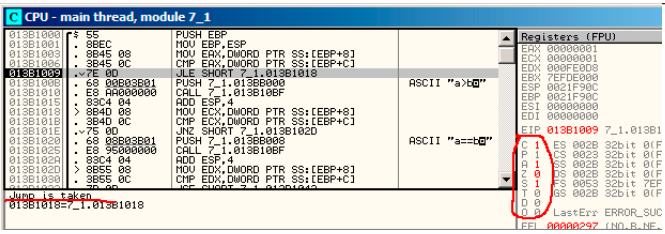


Figure 10.4: OllyDbg: `f_unsigned()`: first conditional jump

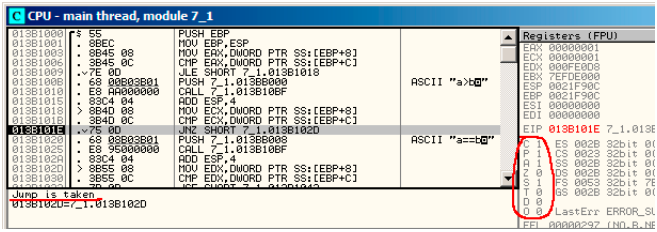


Figure 10.5: OllyDbg: `f_unsigned()`: second conditional jump

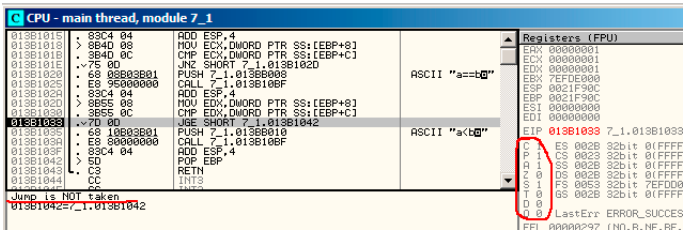


Figure 10.6: OllyDbg: `f_unsigned()`: third conditional jump

10.1.3 x86 + MSVC + Hiew

We can try patch executable file in that way, that `f_unsigned()` function will always print “a==b”, for any input values.

Here is how it looks in Hiew: fig. 10.7.

Essentially, we’ve got three tasks:

- force first jump to be always triggered;
- force second jump to be never triggered;
- force third jump to be always triggered.

Thus we can point code flow into the second `printf()`, and it always print "a==b". Three instructions (or bytes) should be patched:

- The first jump will now be `JMP`, but jump offset will be same.
- The second jump may be triggered sometimes, but in any case it will jump to the next instruction, because, we set jump offset to 0. Jump offset is just to be added to the address of the next instruction in these instructions. So if offset is 0, jump will be done to the next instruction.
- The third jump we convert into `JMP` just as the first one, so it will be triggered always.

That's what we do: fig. 10.8.

If we could forget about any of these jumps, then several `printf()` calls may execute, but this behaviour is not we're need.

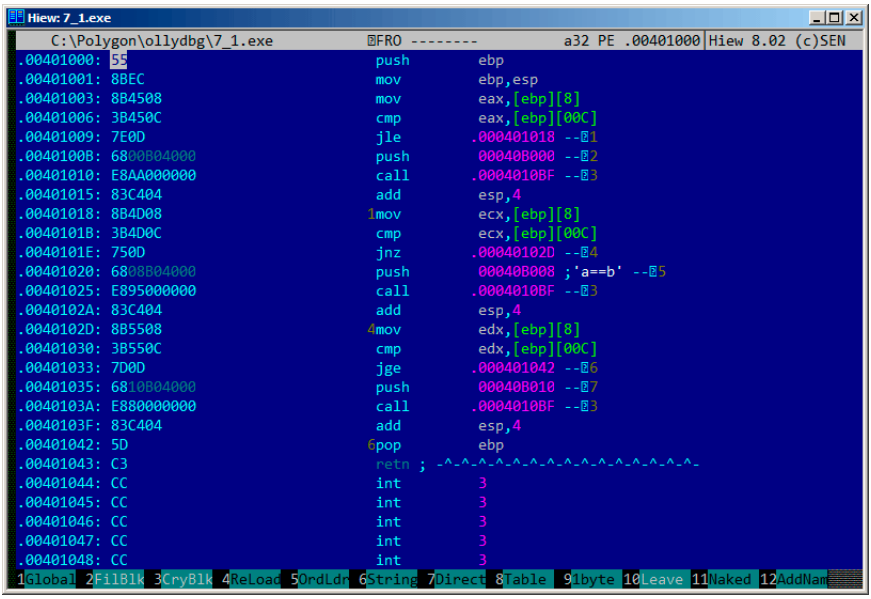


Figure 10.7: Hiew: `f_unsigned()` function

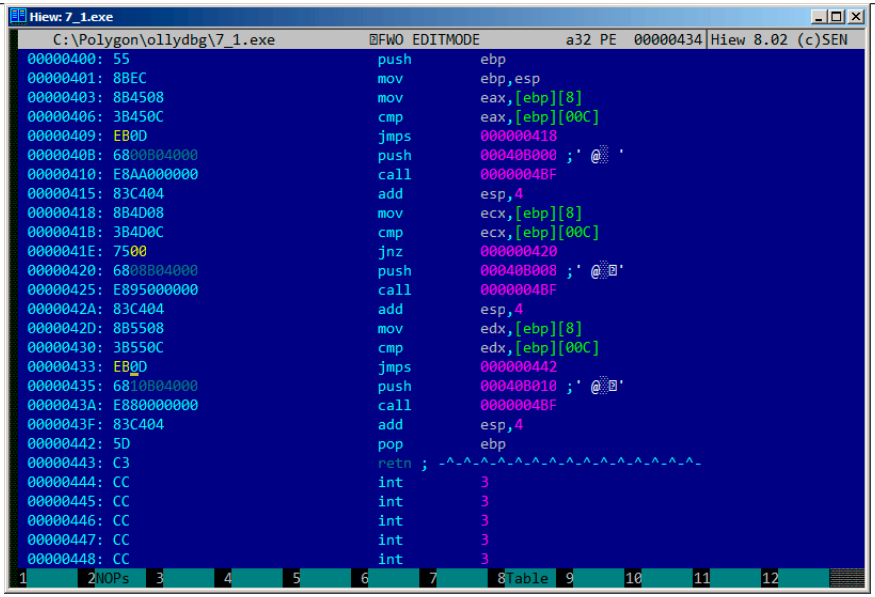


Figure 10.8: Hiew: let's modify `f_unsigned()` function

10.1.4 Non-optimizing GCC

Non-optimizing GCC 4.4.1 produce almost the same code, but with `puts()` (2.3.3) instead of `printf()`.

10.1.5 Optimizing GCC

Observant reader may ask, why to execute `CMP` so many times, if flags are same all the time? Perhaps, optimizing MSVC can't do this, but optimizing GCC 4.8.1 can do deep optimization:

Listing 10.4: GCC 4.8.1 f_signed()

```
f_signed:
        mov     eax, DWORD PTR [esp+8]
        cmp     DWORD PTR [esp+4], eax
        jg      .L6
        je      .L7
        jge     .L1
        mov     DWORD PTR [esp+4], OFFSET FLAT:.LC2 ; "a<b"
        jmp     puts
.L6:
        mov     DWORD PTR [esp+4], OFFSET FLAT:.LC0 ; "a>b"
```

```
        jmp     puts
.L1:
        rep ret
.L7:
        mov     DWORD PTR [esp+4], OFFSET FLAT:.LC1 ; "a==b"
        jmp     puts
```

We also see `JMP puts` here instead of `CALL puts / RETN`. This kind of trick will be described later: 11.1.1.

Needless to say, that type of x86 code is rare. MSVC 2012, as it seems, can't do that. On the other case, assembly language programmers are fully aware of the fact that `Jcc` instructions can be stacked. So if you see it somewhere, it may be a good probability that the code is hand-written.

`f_unsigned()` function is not that æsthetically short:

Listing 10.5: GCC 4.8.1 f_unsigned()

```
f_unsigned:
        push    esi
        push    ebx
        sub     esp, 20
        mov     esi, DWORD PTR [esp+32]
        mov     ebx, DWORD PTR [esp+36]
        cmp     esi, ebx
        ja      .L13
        cmp     esi, ebx        ; instruction may be removed
        je      .L14
.L10:
        jb      .L15
        add     esp, 20
        pop     ebx
        pop     esi
        ret
.L15:
        mov     DWORD PTR [esp+32], OFFSET FLAT:.LC2 ; "a<b"
        add     esp, 20
        pop     ebx
        pop     esi
        jmp     puts
.L13:
        mov     DWORD PTR [esp], OFFSET FLAT:.LC0 ; "a>b"
        call    puts
        cmp     esi, ebx
        jne     .L10
.L14:
```

```
        mov     DWORD PTR [esp+32], OFFSET FLAT:.LC1 ; "a==b"
        add     esp, 20
        pop     ebx
        pop     esi
        jmp     puts
```

So, optimization algorithms of GCC 4.8.1 are probably not always perfect yet.

10.2 ARM

10.2.1 Optimizing Keil + ARM mode

Listing 10.6: Optimizing Keil + ARM mode

```
.text:000000B8                         EXPORT f_signed
.text:000000B8             f_signed                ; CODE XREF: ↙
    ↳ main+C
.text:000000B8 70 40 2D E9         STMFD   SP!, {R4-R6,LR}
.text:000000BC 01 40 A0 E1         MOV     R4, R1
.text:000000C0 04 00 50 E1         CMP     R0, R4
.text:000000C4 00 50 A0 E1         MOV     R5, R0
.text:000000C8 1A 0E 8F C2         ADRGT   R0, aAB         ; "a>↙
    ↳ b\n"
.text:000000CC A1 18 00 CB         BLGT    __2printf
.text:000000D0 04 00 55 E1         CMP     R5, R4
.text:000000D4 67 0F 8F 02         ADREQ   R0, aAB_0       ; "a↙
    ↳ ==b\n"
.text:000000D8 9E 18 00 0B         BLEQ    __2printf
.text:000000DC 04 00 55 E1         CMP     R5, R4
.text:000000E0 70 80 BD A8         LDMGEFD SP!, {R4-R6,PC}
.text:000000E4 70 40 BD E8         LDMFD   SP!, {R4-R6,LR}
.text:000000E8 19 0E 8F E2         ADR     R0, aAB_1       ; "a<↙
    ↳ b\n"
.text:000000EC 99 18 00 EA         B       __2printf
.text:000000EC             ; End of function f_signed
```

A lot of instructions in ARM mode can be executed only when specific flags are set. E.g. this is often used while numbers comparing.

For instance, ADD instruction is ADDAL internally in fact, where AL meaning *Always*, i.e., execute always. Predicates are encoded in 4 high bits of 32-bit ARM instructions (*condition field*). B instruction of unconditional jump is in fact conditional and encoded just like any other conditional jumps, but has AL in the *condition field*, and what it means, executing always, ignoring flags.

ADRGT instructions works just like ADR but will execute only in the case when previous CMP instruction, while comparing two numbers, found one number greater than another (*Greater Than*).

The next BLGT instruction behaves exactly as BL and will be triggered only if result of comparison was the same (*Greater Than*). ADRGT writes a pointer to the string ''a>b\n'', into R0 and BLGT calls printf(). Consequently, these instructions with -GT suffix, will be executed only in the case when value in the R0 (a is there) was bigger than value in the R4 (b is there).

Then we see ADREQ and BLEQ instructions. They behave just like ADR and BL but is to be executed only in the case when operands were equal to each other while comparison. Another CMP is before them (since printf() call may tamper state of flags).

Then we see LDMGEFD, this instruction works just like LDMFD[1], but will be triggered only in the case when one value was greater or equal to another while comparison (*Greater or Equal*).

The sense of ''LDMGEFD SP!, {R4-R6,PC}'' instruction is that is like function epilogue, but it will be triggered only if $a >= b$, only then function execution will be finished. But if it is not true, i.e., $a < b$, then control flow come to next ''LDMFD SP!, {R4-R6,LR}'' instruction, this is one more function epilogue, this instruction restores R4-R6 registers state, but also LR instead of PC, thus, it does not returns from function. Last two instructions calls printf() with the string «a<b\n» as sole argument. Unconditional jump to the printf() function instead of function return, is what we already examined in «printf() with several arguments» section, here (5.3.2).

f_unsigned is likewise, but ADRHI, BLHI, and LDMCSFD instructions are used there, these predicates (*HI = Unsigned higher, CS = Carry Set (greater than or equal)*) are analogical to those examined before, but serving for unsigned values.

There is not much new in the main() function for us:

Listing 10.7: main()

```
.text:00000128                 EXPORT main
.text:00000128             main
.text:00000128 10 40 2D E9      STMFD   SP!, {R4,LR}
.text:0000012C 02 10 A0 E3      MOV     R1, #2
.text:00000130 01 00 A0 E3      MOV     R0, #1
.text:00000134 DF FF FF EB      BL      f_signed
.text:00000138 02 10 A0 E3      MOV     R1, #2
.text:0000013C 01 00 A0 E3      MOV     R0, #1
.text:00000140 EA FF FF EB      BL      f_unsigned
.text:00000144 00 00 A0 E3      MOV     R0, #0
.text:00000148 10 80 BD E8      LDMFD   SP!, {R4,PC}
.text:00000148             ; End of function main
```

That's how to get rid of conditional jumps in ARM mode.

[1] Load Multiple Full Descending

Why it is so good? Since ARM is RISC-processor with pipeline for instructions executing. In short, pipelined processor is not very good on jumps at all, so that is why branch predictor units are critical here. It is very good if the program has as few jumps as possible, conditional and unconditional, so that is why, predicated instructions can help in reducing conditional jumps count.

There is no such feature in x86, if not to count CMOVcc instruction, it is the same as MOV, but triggered only when specific flags are set, usually set while value comparison by CMP.

10.2.2 Optimizing Keil + thumb mode

Listing 10.8: Optimizing Keil + thumb mode

```
.text:00000072             f_signed ; CODE XREF: main+6
.text:00000072 70 B5         PUSH    {R4-R6,LR}
.text:00000074 0C 00         MOVS    R4, R1
.text:00000076 05 00         MOVS    R5, R0
.text:00000078 A0 42         CMP     R0, R4
.text:0000007A 02 DD         BLE     loc_82
.text:0000007C A4 A0         ADR     R0, aAB         ; "a>b\n"
.text:0000007E 06 F0 B7 F8   BL      __2printf
.text:00000082
.text:00000082             loc_82 ; CODE XREF: f_signed+8
.text:00000082 A5 42         CMP     R5, R4
.text:00000084 02 D1         BNE     loc_8C
.text:00000086 A4 A0         ADR     R0, aAB_0       ; "a==b\n"
.text:00000088 06 F0 B2 F8   BL      __2printf
.text:0000008C
.text:0000008C             loc_8C ; CODE XREF: f_signed+12
.text:0000008C A5 42         CMP     R5, R4
.text:0000008E 02 DA         BGE     locret_96
.text:00000090 A3 A0         ADR     R0, aAB_1       ; "a<b\n"
.text:00000092 06 F0 AD F8   BL      __2printf
.text:00000096
.text:00000096             locret_96 ; CODE XREF: f_signed+1C
.text:00000096 70 BD         POP     {R4-R6,PC}
.text:00000096             ; End of function f_signed
```

Only B instructions in thumb mode may be supplemented by *condition codes*, so the thumb code looks more ordinary.

BLE is usual conditional jump *Less than or Equal*, BNE—*Not Equal*, BGE—*Greater than or Equal*.

f_unsigned function is just likewise, but other instructions are used while dealing with unsigned values:BLS (*Unsigned lower or same*) and BCS (*Carry Set (Greater*

than or equal)).

Chapter 11

switch()/case/default

11.1 Few number of cases

```
void f (int a)
{
    switch (a)
    {
    case 0: printf ("zero\n"); break;
    case 1: printf ("one\n"); break;
    case 2: printf ("two\n"); break;
    default: printf ("something unknown\n"); break;
    };
};
```

11.1.1 x86

Result (MSVC 2010):

Listing 11.1: MSVC 2010

```
tv64 = -4 ; size = 4
_a$ = 8   ; size = 4
_f    PROC
    push   ebp
    mov    ebp, esp
    push   ecx
    mov    eax, DWORD PTR _a$[ebp]
    mov    DWORD PTR tv64[ebp], eax
    cmp    DWORD PTR tv64[ebp], 0
```

```
    je      SHORT $LN4@f
    cmp     DWORD PTR tv64[ebp], 1
    je      SHORT $LN3@f
    cmp     DWORD PTR tv64[ebp], 2
    je      SHORT $LN2@f
    jmp     SHORT $LN1@f
$LN4@f:
    push    OFFSET $SG739 ; 'zero', 0aH, 00H
    call    _printf
    add     esp, 4
    jmp     SHORT $LN7@f
$LN3@f:
    push    OFFSET $SG741 ; 'one', 0aH, 00H
    call    _printf
    add     esp, 4
    jmp     SHORT $LN7@f
$LN2@f:
    push    OFFSET $SG743 ; 'two', 0aH, 00H
    call    _printf
    add     esp, 4
    jmp     SHORT $LN7@f
$LN1@f:
    push    OFFSET $SG745 ; 'something unknown', 0aH, 00H
    call    _printf
    add     esp, 4
$LN7@f:
    mov     esp, ebp
    pop     ebp
    ret     0
_f    ENDP
```

Out function with a few cases in switch(), in fact, is analogous to this construction:

```
void f (int a)
{
    if (a==0)
        printf ("zero\n");
    else if (a==1)
        printf ("one\n");
    else if (a==2)
        printf ("two\n");
    else
        printf ("something unknown\n");
};
```

When few cases in switch(), and we see such code, it is impossible to say with certainty, was it switch() in source code, or just pack of if(). This means, switch() is syntactic sugar for large number of nested checks constructed using if().

Nothing especially new to us in generated code, with the exception the compiler moving input variable a to temporary local variable tv64.

If to compile the same in GCC 4.4.1, we'll get almost the same, even with maximal optimization turned on (-O3 option).

Now let's turn on optimization in MSVC (/Ox): `cl 1.c /Fa1.asm /Ox`

Listing 11.2: MSVC

```
_a$ = 8 ; size = 4
_f    PROC
    mov    eax, DWORD PTR _a$[esp-4]
    sub    eax, 0
    je     SHORT $LN4@f
    sub    eax, 1
    je     SHORT $LN3@f
    sub    eax, 1
    je     SHORT $LN2@f
    mov    DWORD PTR _a$[esp-4], OFFSET $SG791 ; 'something ↙
    ↳ unknown', 0aH, 00H
    jmp    _printf
$LN2@f:
    mov    DWORD PTR _a$[esp-4], OFFSET $SG789 ; 'two', 0aH, 00H
    jmp    _printf
$LN3@f:
    mov    DWORD PTR _a$[esp-4], OFFSET $SG787 ; 'one', 0aH, 00H
    jmp    _printf
$LN4@f:
    mov    DWORD PTR _a$[esp-4], OFFSET $SG785 ; 'zero', 0aH, 00↙
    ↳ H
    jmp    _printf
_f    ENDP
```

Here we can see even dirty hacks.

First: the value of the a variable is placed into EAX and 0 subtracted from it. Sounds absurdly, but it may needs to check if 0 was in the EAX register before? If yes, flag ZF will be set (this also means that subtracting from 0 is 0) and first conditional jump JE (*Jump if Equal* or synonym JZ —*Jump if Zero*) will be triggered and control flow passed to the $LN4@f label, where 'zero' message is begin printed. If first jump was not triggered, 1 subtracted from the input value and if at some stage 0 will be resulted, corresponding jump will be triggered.

And if no jump triggered at all, control flow passed to the printf() with argument 'something unknown'.

Second: we see unusual thing for us: string pointer is placed into the a variable, and then printf() is called not via CALL, but via JMP. This could be explained simply. Caller pushing to stack a value and calling our function via CALL. CALL itself pushing returning address to stack and do unconditional jump to our function address. Our function at any point of execution (since it do not contain any instruction moving stack pointer) has the following stack layout:

- ESP—pointing to RA
- ESP+4—pointing to the a variable

On the other side, when we need to call printf() here, we need exactly the same stack layout, except of first printf() argument pointing to string. And that is what our code does.

It replaces function's first argument to different and jumping to the printf(), as if not our function f() was called firstly, but immediately printf(). printf() printing a string to stdout and then execute RET instruction, which POPping RA from stack and control flow is returned not to f() but to the f()'s callee, escaping f().

All this is possible since printf() is called right at the end of the f() function in any case. In some way, it is all similar to the longjmp()[1] function. And of course, it is all done for the sake of speed.

Similar case with ARM compiler described in "printf() with several arguments", section, here (5.3.2).

11.1.2 ARM: Optimizing Keil + ARM mode

```
.text:0000014C             f1
.text:0000014C 00 00 50 E3   CMP     R0, #0
.text:00000150 13 0E 8F 02   ADREQ   R0, aZero ; "zero\n"
.text:00000154 05 00 00 0A   BEQ     loc_170
.text:00000158 01 00 50 E3   CMP     R0, #1
.text:0000015C 4B 0F 8F 02   ADREQ   R0, aOne  ; "one\n"
.text:00000160 02 00 00 0A   BEQ     loc_170
.text:00000164 02 00 50 E3   CMP     R0, #2
.text:00000168 4A 0F 8F 12   ADRNE   R0, aSomethingUnkno ; "↙
    ↳ something unknown\n"
.text:0000016C 4E 0F 8F 02   ADREQ   R0, aTwo  ; "two\n"
.text:00000170
.text:00000170             loc_170 ; CODE XREF: f1+8
.text:00000170                     ; f1+14
.text:00000170 78 18 00 EA   B       __2printf
```

[1] http://en.wikipedia.org/wiki/Setjmp.h

Again, by investigating this code, we cannot say, was it switch() in the original source code, or pack of if() statements.

Anyway, we see here predicated instructions again (like ADREQ (*Equal*)) which will be triggered only in $R0 = 0$ case, and the, address of the *«zero\n»* string will be loaded into the R0. The next instruction BEQ will redirect control flow to loc_170, if $R0 = 0$. By the way, astute reader may ask, will BEQ triggered right since ADREQ before it is already filled the R0 register with another value. Yes, it will since BEQ checking flags set by CMP instruction, and ADREQ not modifying flags at all.

By the way, there is *-S* suffix for some instructions in ARM, indicating the instruction will set the flags according to the result, and without it —the flags will not be touched. For example ADD unlike ADDS will add two numbers, but flags will not be touched. Such instructions are convenient to use between CMP where flags are set and, e.g. conditional jumps, where flags are used.

Other instructions are already familiar to us. There is only one call to printf(), at the end, and we already examined this trick here (5.3.2). There are three paths to printf() at the end.

Also pay attention to what is going on if $a = 2$ and if a is not in range of constants it is comparing against. ''CMP R0, #2'' instruction is needed here to know, if $a = 2$ or not. If it is not true, then ADRNE will load pointer to the string *«something unknown \n»* into R0 since a was already checked before to be equal to 0 or 1, so we can be assured the a variable is not equal to these numbers at this point. And if $R0 = 2$, a pointer to string *«two\n»* will be loaded by ADREQ into R0.

11.1.3 ARM: Optimizing Keil + thumb mode

```
.text:000000D4                f1
.text:000000D4 10 B5          PUSH    {R4,LR}
.text:000000D6 00 28          CMP     R0, #0
.text:000000D8 05 D0          BEQ     zero_case
.text:000000DA 01 28          CMP     R0, #1
.text:000000DC 05 D0          BEQ     one_case
.text:000000DE 02 28          CMP     R0, #2
.text:000000E0 05 D0          BEQ     two_case
.text:000000E2 91 A0          ADR     R0, aSomethingUnkno ; "↙
    ↳ something unknown\n"
.text:000000E4 04 E0          B       default_case

.text:000000E6                zero_case ; CODE XREF: f1+4
.text:000000E6 95 A0          ADR     R0, aZero ; "zero\n"
.text:000000E8 02 E0          B       default_case

.text:000000EA                one_case ; CODE XREF: f1+8
.text:000000EA 96 A0          ADR     R0, aOne ; "one\n"
```

```
.text:000000EC 00 E0         B       default_case

.text:000000EE               two_case ; CODE XREF: f1+C
.text:000000EE 97 A0         ADR     R0, aTwo ; "two\n"
.text:000000F0               default_case ; CODE XREF: f1+10
.text:000000F0                                        ; f1+14
.text:000000F0 06 F0 7E F8   BL      __2printf
.text:000000F4 10 BD         POP     {R4,PC}
.text:000000F4               ; End of function f1
```

As I already mentioned, there is no feature of *connecting* predicates to majority of instructions in thumb mode, so the thumb-code here is somewhat similar to the easily understandable x86 CISC-code

11.2 A lot of cases

If `switch()` statement contain a lot of case's, it is not very convenient for compiler to emit too large code with a lot `JE`/`JNE` instructions.

```
void f (int a)
{
    switch (a)
    {
    case 0: printf ("zero\n"); break;
    case 1: printf ("one\n"); break;
    case 2: printf ("two\n"); break;
    case 3: printf ("three\n"); break;
    case 4: printf ("four\n"); break;
    default: printf ("something unknown\n"); break;
    };
};
```

11.2.1 x86

We got (MSVC 2010):

Listing 11.3: MSVC 2010

```
tv64 = -4  ; size = 4
_a$ = 8    ; size = 4
_f    PROC
    push   ebp
    mov    ebp, esp
    push   ecx
```

```
    mov     eax, DWORD PTR _a$[ebp]
    mov     DWORD PTR tv64[ebp], eax
    cmp     DWORD PTR tv64[ebp], 4
    ja      SHORT $LN1@f
    mov     ecx, DWORD PTR tv64[ebp]
    jmp     DWORD PTR $LN11@f[ecx*4]
$LN6@f:
    push    OFFSET $SG739 ; 'zero', 0aH, 00H
    call    _printf
    add     esp, 4
    jmp     SHORT $LN9@f
$LN5@f:
    push    OFFSET $SG741 ; 'one', 0aH, 00H
    call    _printf
    add     esp, 4
    jmp     SHORT $LN9@f
$LN4@f:
    push    OFFSET $SG743 ; 'two', 0aH, 00H
    call    _printf
    add     esp, 4
    jmp     SHORT $LN9@f
$LN3@f:
    push    OFFSET $SG745 ; 'three', 0aH, 00H
    call    _printf
    add     esp, 4
    jmp     SHORT $LN9@f
$LN2@f:
    push    OFFSET $SG747 ; 'four', 0aH, 00H
    call    _printf
    add     esp, 4
    jmp     SHORT $LN9@f
$LN1@f:
    push    OFFSET $SG749 ; 'something unknown', 0aH, 00H
    call    _printf
    add     esp, 4
$LN9@f:
    mov     esp, ebp
    pop     ebp
    ret     0
    npad    2
$LN11@f:
    DD      $LN6@f ; 0
    DD      $LN5@f ; 1
```

```
    DD    $LN4@f ; 2
    DD    $LN3@f ; 3
    DD    $LN2@f ; 4
_f    ENDP
```

OK, what we see here is: there is a set of the `printf()` calls with various arguments. All they has not only addresses in process memory, but also internal symbolic labels assigned by compiler. Besides, all these labels are also places into `$LN11@f` internal table.

At the function beginning, if `a` is greater than 4, control flow is passed to label `$LN1@f`, where `printf()` with argument `'something unknown'` is called.

And if `a` value is less or equals to 4, let's multiply it by 4 and add `$LN1@f` table address. That is how address inside of table is constructed, pointing exactly to the element we need. For example, let's say `a` is equal to 2. $2 * 4 = 8$ (all table elements are addresses within 32-bit process that is why all elements contain 4 bytes). Address of the `$LN11@f` table + 8 —it will be table element where `$LN4@f` label is stored. `JMP` fetches `$LN4@f` address from the table and jump to it.

This table called sometimes *jumptable*.

Then corresponding `printf()` is called with argument `'two'`. Literally, `jmp DWORD PTR $LN11@f[ecx*4]` instruction means *jump to DWORD, which is stored at address* `$LN11@f + ecx * 4`.

`npad` (61) is assembly language macro, aligning next label so that it will be stored at address aligned on a 4 byte (or 16 byte) border. This is very suitable for processor since it is able to fetch 32-bit values from memory through memory bus, cache memory, etc, in much effective way if it is aligned.

Let's see what GCC 4.4.1 generates:

Listing 11.4: GCC 4.4.1

```
        public f
f       proc near ; CODE XREF: main+10

var_18 = dword ptr -18h
arg_0  = dword ptr  8

        push    ebp
        mov     ebp, esp
        sub     esp, 18h
        cmp     [ebp+arg_0], 4
        ja      short loc_8048444
        mov     eax, [ebp+arg_0]
        shl     eax, 2
        mov     eax, ds:off_804855C[eax]
        jmp     eax
```

```
loc_80483FE: ; DATA XREF: .rodata:off_804855C
        mov     [esp+18h+var_18], offset aZero ; "zero"
        call    _puts
        jmp     short locret_8048450

loc_804840C: ; DATA XREF: .rodata:08048560
        mov     [esp+18h+var_18], offset aOne ; "one"
        call    _puts
        jmp     short locret_8048450

loc_804841A: ; DATA XREF: .rodata:08048564
        mov     [esp+18h+var_18], offset aTwo ; "two"
        call    _puts
        jmp     short locret_8048450

loc_8048428: ; DATA XREF: .rodata:08048568
        mov     [esp+18h+var_18], offset aThree ; "three"
        call    _puts
        jmp     short locret_8048450

loc_8048436: ; DATA XREF: .rodata:0804856C
        mov     [esp+18h+var_18], offset aFour ; "four"
        call    _puts
        jmp     short locret_8048450

loc_8048444: ; CODE XREF: f+A
        mov     [esp+18h+var_18], offset aSomethingUnkno ; "↙
    ↳ something unknown"
        call    _puts

locret_8048450: ; CODE XREF: f+26
                ; f+34...
        leave
        retn
f       endp

off_804855C dd offset loc_80483FE   ; DATA XREF: f+12
            dd offset loc_804840C
            dd offset loc_804841A
            dd offset loc_8048428
            dd offset loc_8048436
```

It is almost the same, except little nuance: argument `arg_0` is multiplied by 4 with by shifting it to left by 2 bits (it is almost the same as multiplication by 4) (17.3.1). Then

label address is taken from `off_804855C` array, address calculated and stored into `EAX`, then ''`JMP EAX`'' do actual jump.

11.2.2 ARM: Optimizing Keil + ARM mode

```
00000174             f2
00000174 05 00 50 E3     CMP     R0, #5           ; switch 5 ↙
    ↳ cases
00000178 00 F1 8F 30     ADDCC   PC, PC, R0,LSL#2 ; switch jump
0000017C 0E 00 00 EA     B       default_case     ; jumptable ↙
    ↳ 00000178 default case

00000180
00000180             loc_180 ; CODE XREF: f2+4
00000180 03 00 00 EA     B       zero_case        ; jumptable ↙
    ↳ 00000178 case 0

00000184
00000184             loc_184 ; CODE XREF: f2+4
00000184 04 00 00 EA     B       one_case         ; jumptable ↙
    ↳ 00000178 case 1

00000188
00000188             loc_188 ; CODE XREF: f2+4
00000188 05 00 00 EA     B       two_case         ; jumptable ↙
    ↳ 00000178 case 2

0000018C
0000018C             loc_18C ; CODE XREF: f2+4
0000018C 06 00 00 EA     B       three_case       ; jumptable ↙
    ↳ 00000178 case 3

00000190
00000190             loc_190 ; CODE XREF: f2+4
00000190 07 00 00 EA     B       four_case        ; jumptable ↙
    ↳ 00000178 case 4

00000194
00000194             zero_case ; CODE XREF: f2+4
00000194                       ; f2:loc_180
00000194 EC 00 8F E2     ADR     R0, aZero        ; jumptable ↙
    ↳ 00000178 case 0
00000198 06 00 00 EA     B       loc_1B8
```

```
0000019C
0000019C             one_case ; CODE XREF: f2+4
0000019C                      ; f2:loc_184
0000019C EC 00 8F E2     ADR     R0, aOne        ; jumptable ↙
    ↳ 00000178 case 1
000001A0 04 00 00 EA     B       loc_1B8

000001A4
000001A4             two_case ; CODE XREF: f2+4
000001A4                      ; f2:loc_188
000001A4 01 0C 8F E2     ADR     R0, aTwo        ; jumptable ↙
    ↳ 00000178 case 2
000001A8 02 00 00 EA     B       loc_1B8

000001AC
000001AC             three_case ; CODE XREF: f2+4
000001AC                       ; f2:loc_18C
000001AC 01 0C 8F E2     ADR     R0, aThree      ; jumptable ↙
    ↳ 00000178 case 3
000001B0 00 00 00 EA     B       loc_1B8

000001B4
000001B4             four_case ; CODE XREF: f2+4
000001B4                      ; f2:loc_190
000001B4 01 0C 8F E2     ADR     R0, aFour       ; jumptable ↙
    ↳ 00000178 case 4
000001B8
000001B8             loc_1B8   ; CODE XREF: f2+24
000001B8                      ; f2+2C
000001B8 66 18 00 EA     B       __2printf

000001BC
000001BC             default_case ; CODE XREF: f2+4
000001BC                          ; f2+8
000001BC D4 00 8F E2     ADR     R0, aSomethingUnkno ; jumptable↙
    ↳  00000178 default case
000001C0 FC FF FF EA     B       loc_1B8
000001C0             ; End of function f2
```

This code makes use of the ARM feature in which all instructions in the ARM mode has size of 4 bytes.

Let's keep in mind the maximum value for a is 4 and any greater value must cause *«something unknown\n»* string printing.

The very first ‘‘CMP R0, #5’’ instruction compares a input value with 5.

The next ‘‘ADDCC PC, PC, R0,LSL#2’’ [2] instruction will execute only if $R0 < 5$ (*CC=Carry clear / Less than*). Consequently, if ADDCC will not trigger (it is a $R0 \geq 5$ case), a jump to *default_case*label will be occurred.

But if $R0 < 5$ and ADDCC will trigger, following events will happen:

Value in the R0 is multiplied by 4. In fact, LSL#2 at the instruction's ending means “shift left by 2 bits”. But as we will see later (17.3.1) in “Shifts” section, shift left by 2 bits is just equivalently to multiplying by 4.

Then, $R0 * 4$ value we got, is added to current value in the PC, thus jumping to one of B (*Branch*) instructions located below.

At the moment of ADDCC execution, value in the PC is 8 bytes ahead (0x180) than address at which ADDCC instruction is located (0x178), or, in other words, 2 instructions ahead.

This is how ARM processor pipeline works: when ADDCC instruction is executed, the processor at the moment is beginning to process instruction after the next one, so that is why PC pointing there.

If $a = 0$, then nothing will be added to the value in the PC, and actual value in the PC is to be written into the PC (which is 8 bytes ahead) and jump to the label *loc_180* will happen, this is 8 bytes ahead of the point where ADDCC instruction is.

In case of $a = 1$, then $PC + 8 + a * 4 = PC + 8 + 1 * 4 = PC + 16 = 0x184$will be written to the PC, this is the address of the *loc_184* label.

With every 1 added to a, resulting PC increasing by 4. 4 is also instruction length in ARM mode and also, length of each B instruction length, there are 5 of them in row.

Each of these five B instructions passing control further, where something is going on, what was programmed in *switch()*. Pointer loading to corresponding string occurring there, etc.

11.2.3 ARM: Optimizing Keil + thumb mode

```
000000F6                        EXPORT f2
000000F6                    f2
000000F6 10 B5                  PUSH    {R4,LR}
000000F8 03 00                  MOVS    R3, R0
000000FA 06 F0 69 F8            BL      ↙
    ↳ __ARM_common_switch8_thumb ; switch 6 cases

000000FE 05                     DCB 5
000000FF 04 06 08 0A 0C 10      DCB 4, 6, 8, 0xA, 0xC, 0x10 ; ↙
    ↳ jump table for switch statement
00000105 00                     ALIGN 2
00000106
```

[2] ADD—addition

```
00000106                    zero_case ; CODE XREF: f2+4
00000106 8D A0                  ADR     R0, aZero ; jumptable ↙
    ↳ 000000FA case 0
00000108 06 E0                  B       loc_118

0000010A
0000010A                    one_case ; CODE XREF: f2+4
0000010A 8E A0                  ADR     R0, aOne ; jumptable ↙
    ↳ 000000FA case 1
0000010C 04 E0                  B       loc_118

0000010E
0000010E                    two_case ; CODE XREF: f2+4
0000010E 8F A0                  ADR     R0, aTwo ; jumptable ↙
    ↳ 000000FA case 2
00000110 02 E0                  B       loc_118

00000112
00000112                    three_case ; CODE XREF: f2+4
00000112 90 A0                  ADR     R0, aThree ; jumptable ↙
    ↳ 000000FA case 3
00000114 00 E0                  B       loc_118

00000116
00000116                    four_case ; CODE XREF: f2+4
00000116 91 A0                  ADR     R0, aFour ; jumptable ↙
    ↳ 000000FA case 4
00000118
00000118                    loc_118 ; CODE XREF: f2+12
00000118                            ; f2+16
00000118 06 F0 6A F8            BL      __2printf
0000011C 10 BD                  POP     {R4,PC}

0000011E
0000011E                    default_case ; CODE XREF: f2+4
0000011E 82 A0                  ADR     R0, aSomethingUnkno ; ↙
    ↳ jumptable 000000FA default case
00000120 FA E7                  B       loc_118


000061D0                        EXPORT __ARM_common_switch8_thumb
000061D0                    __ARM_common_switch8_thumb ; CODE ↙
    ↳ XREF: example6_f2+4
```

```
000061D0 78 47                  BX      PC

000061D2 00 00                  ALIGN 4
000061D2                    ; End of function ↙
    ↳ __ARM_common_switch8_thumb
000061D2
000061D4                    __32__ARM_common_switch8_thumb ; CODE↙
    ↳  XREF: __ARM_common_switch8_thumb
000061D4 01 C0 5E E5            LDRB    R12, [LR,#-1]
000061D8 0C 00 53 E1            CMP     R3, R12
000061DC 0C 30 DE 27            LDRCSB  R3, [LR,R12]
000061E0 03 30 DE 37            LDRCCB  R3, [LR,R3]
000061E4 83 C0 8E E0            ADD     R12, LR, R3,LSL#1
000061E8 1C FF 2F E1            BX      R12
000061E8                    ; End of function ↙
    ↳ __32__ARM_common_switch8_thumb
```

One cannot be sure all instructions in thumb and thumb-2 modes will have same size. It is even can be said that in these modes instructions has variable length, just like in x86.

So there is a special table added, containing information about how much cases are there, not including default-case, and offset, for each, each encoding a label, to which control must be passed in corresponding case.

A special function here present in order to deal with the table and pass control, named *__ARM_common_switch8_thumb*. It is beginning with "`BX PC`" instruction, which function is to switch processor to ARM-mode. Then you may see the function for table processing. It is too complex for describing it here now, so I will omit elaborations.

But it is interesting to note the function uses LR register as a pointer to the table. Indeed, after this function calling, LR will contain address after "`BL __ARM_common_switch8_thumb`" instruction, and the table is beginning right there.

It is also worth noting the code is generated as a separate function in order to reuse it, in similar places, in similar cases, for *switch()* processing, so compiler will not generate same code at each point.

IDA successfully perceived it as a service function and table, automatically, and added commentaries to labels like `jumptable 000000FA case 0`.

Chapter 12

Loops

12.1 x86

There is a special LOOP instruction in x86 instruction set, it is checking value in the ECX register and if it is not 0, do ECX decrement and pass control flow to the label mentioned in the LOOP operand. Probably, this instruction is not very convenient, so, I did not ever see any modern compiler emit it automatically. So, if you see the instruction somewhere in code, it is most likely this is manually written piece of assembly code.

By the way, as home exercise, you could try to explain, why this instruction is not very convenient.

In C/C++ loops are constructed using `for()`, `while()`, `do/while()` statements.

Let's start with `for()`.

This statement defines loop initialization (set loop counter to initial value), loop condition (is counter is bigger than a limit?), what is done at each iteration (increment/decrement) and of course loop body.

```
for (initialization; condition; at each iteration)
{
  loop_body;
}
```

So, generated code will be consisted of four parts too.

Let's start with simple example:

```
#include <stdio.h>

void f(int i)
{
        printf ("f(%d)\n", i);
};
```

```
int main()
{
        int i;

        for (i=2; i<10; i++)
                f(i);

        return 0;
};
```

Result (MSVC 2010):

Listing 12.1: MSVC 2010

```
_i$ = -4
_main    PROC
    push   ebp
    mov    ebp, esp
    push   ecx
    mov    DWORD PTR _i$[ebp], 2   ; loop initialization
    jmp    SHORT $LN3@main
$LN2@main:
    mov    eax, DWORD PTR _i$[ebp] ; here is what we do after ↙
    ↘ each iteration:
    add    eax, 1                  ; add 1 to i value
    mov    DWORD PTR _i$[ebp], eax
$LN3@main:
    cmp    DWORD PTR _i$[ebp], 10  ; this condition is checked *↙
    ↘ before* each iteration
    jge    SHORT $LN1@main         ; if i is biggest or equals ↙
    ↘ to 10, let's finish loop
    mov    ecx, DWORD PTR _i$[ebp] ; loop body: call f(i)
    push   ecx
    call   _f
    add    esp, 4
    jmp    SHORT $LN2@main         ; jump to loop begin
$LN1@main:                         ; loop end
    xor    eax, eax
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP
```

Nothing very special, as we see.

GCC 4.4.1 emits almost the same code, with one subtle difference:

Listing 12.2: GCC 4.4.1

```
main            proc near               ; DATA XREF: _start+17

var_20          = dword ptr -20h
var_4           = dword ptr -4

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 20h
                mov     [esp+20h+var_4], 2  ; i initializing
                jmp     short loc_8048476

loc_8048465:
                mov     eax, [esp+20h+var_4]
                mov     [esp+20h+var_20], eax
                call    f
                add     [esp+20h+var_4], 1  ; i increment

loc_8048476:
                cmp     [esp+20h+var_4], 9
                jle     short loc_8048465   ; if i<=9, continue ↙
    ↳ loop
                mov     eax, 0
                leave
                retn
main            endp
```

Now let's see what we will get if optimization is turned on (`/Ox`):

Listing 12.3: Optimizing MSVC

```
_main    PROC
    push   esi
    mov    esi, 2
$LL3@main:
    push   esi
    call   _f
    inc    esi
    add    esp, 4
    cmp    esi, 10     ; 0000000aH
    jl     SHORT $LL3@main
    xor    eax, eax
    pop    esi
    ret    0
```

```
_main    ENDP
```

What is going on here is: space for the *i* variable is not allocated in local stack anymore, but even individual register: the ESI. This is possible in such small functions where not so many local variables are present.

One very important property is the f() function must not change the value in the ESI. Our compiler is sure here. And if compiler decided to use the ESI register in f() too, its value would be saved then at the f() function's prologue and restored at the f() function's epilogue. Almost like in our listing: please note PUSH ESI/POP ESI at the function begin and end.

Let's try GCC 4.4.1 with maximal optimization turned on (-O3 option):

Listing 12.4: Optimizing GCC 4.4.1

```
main            proc near

var_10          = dword ptr -10h

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 10h
                mov     [esp+10h+var_10], 2
                call    f
                mov     [esp+10h+var_10], 3
                call    f
                mov     [esp+10h+var_10], 4
                call    f
                mov     [esp+10h+var_10], 5
                call    f
                mov     [esp+10h+var_10], 6
                call    f
                mov     [esp+10h+var_10], 7
                call    f
                mov     [esp+10h+var_10], 8
                call    f
                mov     [esp+10h+var_10], 9
                call    f
                xor     eax, eax
                leave
                retn
main            endp
```

Huh, GCC just unwind our loop.

Loop unwinding has advantage in these cases when there is not so much iterations and we could economy some execution speed by removing all loop supporting instructions. On the other side, resulting code is obviously larger.

OK, let's increase maximal value of the *i* variable to 100 and try again. GCC resulting:

Listing 12.5: GCC

```
                public main
main            proc near

var_20          = dword ptr -20h

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                push    ebx
                mov     ebx, 2    ; i=2
                sub     esp, 1Ch
                nop  ; aligning label loc_80484D0 (loop body ↙
    ↳ begin) by 16-byte border

loc_80484D0:
                mov     [esp+20h+var_20], ebx ; pass i as first ↙
    ↳ argument to f()
                add     ebx, 1    ; i++
                call    f
                cmp     ebx, 64h  ; i==100?
                jnz     short loc_80484D0 ; if not, continue
                add     esp, 1Ch
                xor     eax, eax  ; return 0
                pop     ebx
                mov     esp, ebp
                pop     ebp
                retn
main            endp
```

It is quite similar to what MSVC 2010 with optimization (`/Ox`) produce. With the exception the `EBX` register will be fixed to the `i` variable. GCC is sure this register will not be modified inside of the `f()` function, and if it will, it will be saved at the function prologue and restored at epilogue, just like here in the `main()` function.

12.1.1 OllyDbg

Let's compile our example in MSVC 2010 with `/Ox` and `/Ob0` options and load it into OllyDbg.

It seems, OllyDbg is able to detect simple loops and show them in square brackets, for convenience: fig. 12.1.

By tracing (F8 (step over)) we see how `ESI` incrementing. Here, for instance, `ESI` =i=6: fig. 12.2.

9 is a last loop value. That's why `JL` will not trigger after increment, and function finishing: fig. 12.3.

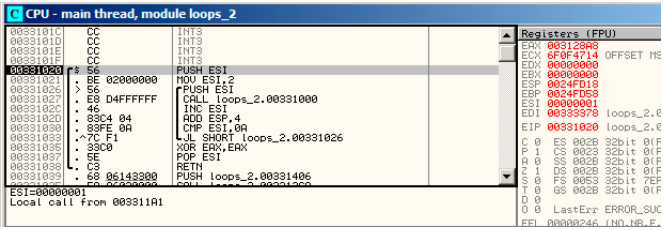


Figure 12.1: OllyDbg: `main()` begin

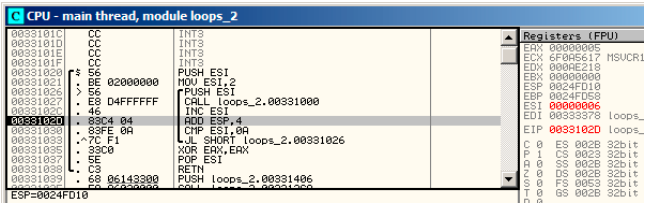


Figure 12.2: OllyDbg: loop body just executed with i=6

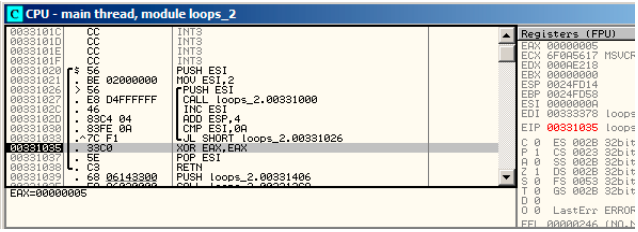


Figure 12.3: OllyDbg: `ESI` =10, loop end

12.1.2 tracer

As we might see, it is not very convenient to trace in debugger manually. That's one of the reasons I write tracer for myself.

I open compiled example in IDA, I find the address of the instruction `PUSH ESI` (passing sole argument into `f()`) and this is `0x401026` for me and I run tracer:

```
tracer.exe -l:loops_2.exe bpx=loops_2.exe!0x00401026
```

BPX just sets breakpoint at address and then will print registers state.

In the `tracer.log` I see after running:

```
PID=12884|New process loops_2.exe
(0) loops_2.exe!0x401026
EAX=0x00a328c8 EBX=0x00000000 ECX=0x6f0f4714 EDX=0x00000000
ESI=0x00000002 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=PF ZF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000003 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000004 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000005 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000006 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000007 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
```

```
ESI=0x00000008 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF AF SF IF
(0) loops_2.exe!0x401026
EAX=0x00000005 EBX=0x00000000 ECX=0x6f0a5617 EDX=0x000ee188
ESI=0x00000009 EDI=0x00333378 EBP=0x0024fbfc ESP=0x0024fbb8
EIP=0x00331026
FLAGS=CF PF AF SF IF
PID=12884|Process loops_2.exe exited. ExitCode=0 (0x0)
```

We see how value of ESI register is changed from 2 to 9.

Even more than that, tracer can collect register values on all addresses within function. This is called *trace* there. Each instruction is being traced, all interesting register values are noticed and collected. .idc-script for IDA is then generated. So, in the IDA I've learned that `main()` function address is `0x00401020` and I run:

```
tracer.exe -l:loops_2.exe bpf=loops_2.exe!0x00401020,trace:cc
```

BPF mean set breakpoint on function.

As a result, I have got `loops_2.exe.idc` and `loops_2.exe_clear.idc` scripts. I'm loading `loops_2.exe.idc` into IDA and I see: fig. 12.4

We see that ESI can be from 2 to 9 at the begin of loop body, but from 3 to 0xA (10) after increment. We can also see that `main()` is finishing with 0 in EAX.

tracer also generates `loops_2.exe.txt`, containing information about how many times each instruction was executed and register values:

Listing 12.6: loops_2.exe.txt

```
0x401020 (.text+0x20), e=       1 [PUSH ESI] ESI=1
0x401021 (.text+0x21), e=       1 [MOV ESI, 2]
0x401026 (.text+0x26), e=       8 [PUSH ESI] ESI=2..9
0x401027 (.text+0x27), e=       8 [CALL 8D1000h] tracing nested ↙
    ↳ maximum level (1) reached, skipping this CALL 8D1000h=0↙
    ↳ x8d1000
0x40102c (.text+0x2c), e=       8 [INC ESI] ESI=2..9
0x40102d (.text+0x2d), e=       8 [ADD ESP, 4] ESP=0x38fcbc
0x401030 (.text+0x30), e=       8 [CMP ESI, 0Ah] ESI=3..0xa
0x401033 (.text+0x33), e=       8 [JL 8D1026h] SF=false,true OF=↙
    ↳ false
0x401035 (.text+0x35), e=       1 [XOR EAX, EAX]
0x401037 (.text+0x37), e=       1 [POP ESI]
0x401038 (.text+0x38), e=       1 [RETN] EAX=0
```

grep can be used here.

```
.text:00401020
.text:00401020 ; =============== S U B R O U T I N E =======================================
.text:00401020
.text:00401020
.text:00401020 ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:00401020 _main           proc near               ; CODE XREF: ___tmainCRTStartup+11D↓p
.text:00401020
.text:00401020 argc            = dword ptr  4
.text:00401020 argv            = dword ptr  8
.text:00401020 envp            = dword ptr  0Ch
.text:00401020
.text:00401020                 push    esi             ; ESI=1
.text:00401021                 mov     esi, 2
.text:00401026
.text:00401026 loc_401026:                             ; CODE XREF: _main+13↓j
.text:00401026                 push    esi             ; ESI=2..9
.text:00401027                 call    sub_401000      ; tracing nested maximum level (1) reached,
.text:0040102C                 inc     esi             ; ESI=2..9
.text:0040102D                 add     esp, 4          ; ESP=0x38fcbc
.text:00401030                 cmp     esi, 0Ah        ; ESI=3..0xa
.text:00401033                 jl      short loc_401026 ; SF=false,true OF=false
.text:00401035                 xor     eax, eax
.text:00401037                 pop     esi
.text:00401038                 retn                    ; EAX=0
.text:00401038 _main           endp
```

Figure 12.4: IDA with .idc-script loaded

12.2 ARM

12.2.1 Non-optimizing Keil + ARM mode

```
main
        STMFD   SP!, {R4,LR}
        MOV     R4, #2
        B       loc_368
loc_35C  ; CODE XREF: main+1C
        MOV     R0, R4
        BL      f
        ADD     R4, R4, #1

loc_368  ; CODE XREF: main+8
        CMP     R4, #0xA
        BLT     loc_35C
        MOV     R0, #0
        LDMFD   SP!, {R4,PC}
```

Iteration counter `i` is to be stored in the `R4` register.

''`MOV R4, #2`'' instruction just initializing *i*.

''`MOV R0, R4`'' and ''`BL f`'' instructions are compose loop body, the first instruction preparing argument for `f()` function and the second is calling it.

''`ADD R4, R4, #1`'' instruction is just adding 1 to the *i* variable during each iteration.

"CMP R4, #0xA" comparing i with 0xA (10). Next instruction BLT (*Branch Less Than*) will jump if *i* is less than 10.

Otherwise, 0 will be written into R0 (since our function returns 0) and function execution ended.

12.2.2 Optimizing Keil + thumb mode

```
_main
                PUSH    {R4,LR}
                MOVS    R4, #2

loc_132                                 ; CODE XREF: _main+E
                MOVS    R0, R4
                BL      example7_f
                ADDS    R4, R4, #1
                CMP     R4, #0xA
                BLT     loc_132
                MOVS    R0, #0
                POP     {R4,PC}
```

Practically, the same.

12.2.3 Optimizing Xcode (LLVM) + thumb-2 mode

```
_main
        PUSH            {R4,R7,LR}
        MOVW            R4, #0x1124 ; "%d\n"
        MOVS            R1, #2
        MOVT.W          R4, #0
        ADD             R7, SP, #4
        ADD             R4, PC
        MOV             R0, R4
        BLX             _printf
        MOV             R0, R4
        MOVS            R1, #3
        BLX             _printf
        MOV             R0, R4
        MOVS            R1, #4
        BLX             _printf
        MOV             R0, R4
        MOVS            R1, #5
        BLX             _printf
        MOV             R0, R4
```

```
        MOVS            R1, #6
        BLX             _printf
        MOV             R0, R4
        MOVS            R1, #7
        BLX             _printf
        MOV             R0, R4
        MOVS            R1, #8
        BLX             _printf
        MOV             R0, R4
        MOVS            R1, #9
        BLX             _printf
        MOVS            R0, #0
        POP             {R4,R7,PC}
```

In fact, this was in my `f()` function:

```
void f(int i)
{
    // do something here
    printf ("%d\n", i);
};
```

So, LLVM not just *unrolled* the loop, but also represented my very simple function `f()` as *inlined*, and inserted its body 8 times instead of loop. This is possible when function is so primitive (like mine) and when it is called not many times (like here).

12.3 One more thing

On the code generated we can see: after *i* initialization, loop body will not be executed, but *i* condition checked first, and only after loop body is to be executed. And that is correct. Because, if loop condition is not met at the beginning, loop body must not be executed. For example, this is possible in the following case:

```
for (i=0; i<total_entries_to_process; i++)
    loop_body;
```

If *total_entries_to_process* equals to 0, loop body must not be executed whatsoever. So that is why condition checked before loop body execution.

However, optimizing compiler may swap condition check and loop body, if it sure that the situation described here is not possible (like in case of our very simple example and Keil, Xcode (LLVM), MSVC in optimization mode).

Chapter 13

strlen()

Now let's talk about loops one more time. Often, `strlen()` function[1] is implemented using `while()` statement. Here is how it is done in MSVC standard libraries:

```
int my_strlen (const char * str)
{
        const char *eos = str;

        while( *eos++ ) ;

        return( eos - str - 1 );
}

int main()
{
        // test
        return my_strlen("hello!");
};
```

13.1 x86

13.1.1 Non-optimizing MSVC

Let's compile:

```
_eos$ = -4                              ; size = 4
_str$ = 8                               ; size = 4
_strlen PROC
```

[1] counting characters in string in C language

```
    push    ebp
    mov     ebp, esp
    push    ecx
    mov     eax, DWORD PTR _str$[ebp]  ; place pointer to string↙
    ↳  from str
    mov     DWORD PTR _eos$[ebp], eax  ; place it to local ↙
    ↳ varuable eos
$LN2@strlen_:
    mov     ecx, DWORD PTR _eos$[ebp]  ; ECX=eos

    ; take 8-bit byte from address in ECX and place it as 32-bit↙
    ↳  value to EDX with sign extension

    movsx   edx, BYTE PTR [ecx]
    mov     eax, DWORD PTR _eos$[ebp]  ; EAX=eos
    add     eax, 1                     ; increment EAX
    mov     DWORD PTR _eos$[ebp], eax  ; place EAX back to eos
    test    edx, edx                   ; EDX is zero?
    je      SHORT $LN1@strlen_         ; yes, then finish loop
    jmp     SHORT $LN2@strlen_         ; continue loop
$LN1@strlen_:

    ; here we calculate the difference between two pointers

    mov    eax, DWORD PTR _eos$[ebp]
    sub    eax, DWORD PTR _str$[ebp]
    sub    eax, 1                     ; subtract 1 and return ↙
    ↳ result
    mov    esp, ebp
    pop    ebp
    ret    0
_strlen_ ENDP
```

Two new instructions here: MOVSX (13.1.1) and TEST.

About first: MOVSX (13.1.1) is intended to take byte from a point in memory and store value in a 32-bit register. MOVSX (13.1.1) meaning *MOV with Sign-Extent*. Rest bits starting at 8th till 31th MOVSX (13.1.1) will set to 1 if source byte in memory has *minus* sign or to 0 if *plus*.

And here is why all this.

C/C++ standard defines *char* type as signed. If we have two values, one is *char* and another is *int*, (*int* is signed too), and if first value contain -2 (it is coded as 0xFE) and we just copying this byte into *int* container, there will be 0x000000FE, and this, from the point of signed *int* view is 254, but not -2. In signed int, -2 is coded as 0xFFFFFFFE. So if we need to transfer 0xFE value from variable of *char* type to *int*,

we need to identify its sign and extend it. That is what MOVSX (13.1.1) does.

See also in section "*Signed number representations*" (32).

I'm not sure if the compiler needs to store *char* variable in the EDX, it could take 8-bit register part (let's say DL). Apparently, compiler's register allocator works like that.

Then we see TEST EDX, EDX. About TEST instruction, read more in section about bit fields (17). But here, this instruction just checking value in the EDX, if it is equals to 0.

13.1.2 Non-optimizing GCC

Let's try GCC 4.4.1:

```
                public strlen
strlen          proc near

eos             = dword ptr -4
arg_0           = dword ptr  8

                push    ebp
                mov     ebp, esp
                sub     esp, 10h
                mov     eax, [ebp+arg_0]
                mov     [ebp+eos], eax

loc_80483F0:
                mov     eax, [ebp+eos]
                movzx   eax, byte ptr [eax]
                test    al, al
                setnz   al
                add     [ebp+eos], 1
                test    al, al
                jnz     short loc_80483F0
                mov     edx, [ebp+eos]
                mov     eax, [ebp+arg_0]
                mov     ecx, edx
                sub     ecx, eax
                mov     eax, ecx
                sub     eax, 1
                leave
                retn
strlen          endp
```

The result almost the same as MSVC did, but here we see MOVZX instead of MOVSX (13.1.1). MOVZX means *MOV with Zero-Extent*. This instruction copies 8-bit or 16-bit value into 32-bit register and sets the rest bits to 0. In fact, this instruction is convenient only since it enable us to replace two instructions at once: `xor eax, eax / mov al, [...]`.

On the other hand, it is obvious to us the compiler could produce the code: `mov al, byte ptr [eax] / test al, al` —it is almost the same, however, the highest EAX register bits will contain random noise. But let's think it is compiler's drawback —it cannot produce more understandable code. Strictly speaking, compiler is not obliged to emit understandable (to humans) code at all.

Next new instruction for us is SETNZ. Here, if AL contain not zero, `test al, al` will set 0 to the ZF flag, but SETNZ, if ZF==0 (*NZ* means *not zero*) will set 1 to the AL. Speaking in natural language, *if AL is not zero, let's jump to loc_80483F0*. Compiler emitted slightly redundant code, but let's not forget the optimization is turned off.

13.1.3 Optimizing MSVC

Now let's compile all this in MSVC 2012, with optimization turned on (/Ox):

Listing 13.1: MSVC 2012 /Ox /Ob0

```
_str$ = 8                       ; size = 4
_strlen PROC
        mov     edx, DWORD PTR _str$[esp-4] ; EDX -> pointer to ↙
    ↳ the string
        mov     eax, edx                    ; move to EAX
$LL2@strlen:
        mov     cl, BYTE PTR [eax]          ; CL = *EAX
        inc     eax                         ; EAX++
        test    cl, cl                      ; CL==0?
        jne     SHORT $LL2@strlen           ; no, continue loop
        sub     eax, edx                    ; calculate pointers↙
    ↳  difference
        dec     eax                         ; decrement EAX
        ret     0
_strlen ENDP
```

Now it is all simpler. But it is needless to say the compiler could use registers such efficiently only in small functions with small number of local variables.

INC/DEC—are increment/decrement instruction, in other words: add 1 to variable or subtract.

13.1.4 Optimizing MSVC + OllyDbg

We may try this (optimized) example in OllyDbg. Here is a very first iteration: fig. 13.1. We see that OllyDbg found a loop and, for convenience, *wrapped* its instructions in bracket. By clicking right button on EAX, we can choose "Follow in Dump" and the memory window position will scroll to the right place. We can see here a string "hello!" in memory. There are at least once zero byte after it and then random garbage. If OllyDbg sees that a register has an address pointing to a string, it will show it.

Let's press F8 (step over) enough time so the current address will be at the loop body begin again: fig. 13.2. We see that EAX contain address of the second character in the string.

We will press F8 enough times in order to escape from the loop: fig. 13.3. We will see that EAX now contain address of zeroth byte, placed right after the string. Meanwhile, EDX wasn't changed, so it still pointing to the string begin. Difference between these two addresses will be calculated now.

SUB instruction was just executed: fig. 13.4. Difference in the EAX—7. Indeed, the "hello!" string length—6, but with zeroth byte included—7. But the `strlen()` must return non-zero characters in the string. So the decrement will processed now and then return from the function.

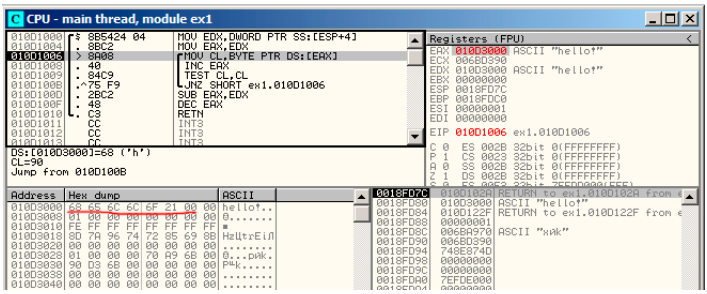


Figure 13.1: OllyDbg: first iteration begin

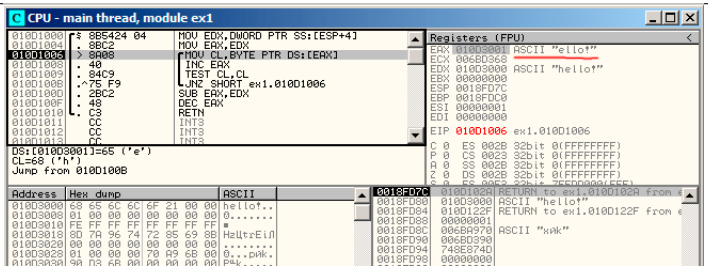


Figure 13.2: OllyDbg: second iteration begin

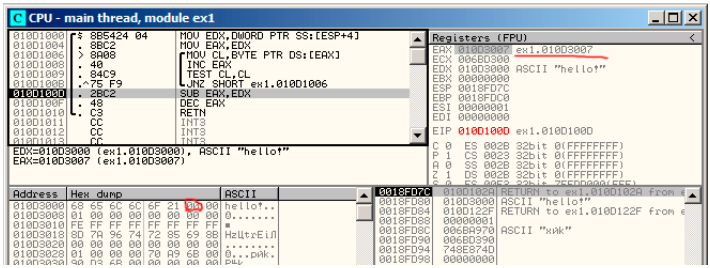


Figure 13.3: OllyDbg: pointers difference to be calculated now

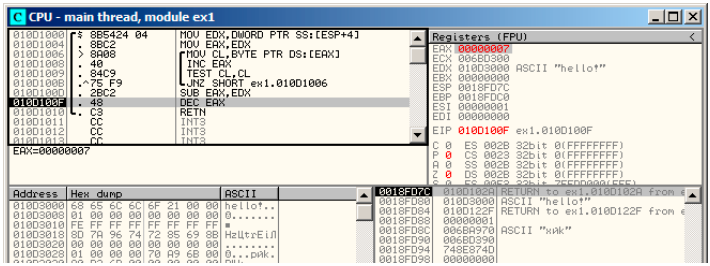


Figure 13.4: OllyDbg: EAX to be decremented now

13.1.5 Optimizing GCC

Let's check GCC 4.4.1 with optimization turned on (-O3 key):

```
                public strlen
strlen          proc near

arg_0           = dword ptr  8
```

```
                push    ebp
                mov     ebp, esp
                mov     ecx, [ebp+arg_0]
                mov     eax, ecx

loc_8048418:
                movzx   edx, byte ptr [eax]
                add     eax, 1
                test    dl, dl
                jnz     short loc_8048418
                not     ecx
                add     eax, ecx
                pop     ebp
                retn
strlen          endp
```

Here GCC is almost the same as MSVC, except of `MOVZX` presence.

However, `MOVZX` could be replaced here to `mov dl, byte ptr [eax]`.

Probably, it is simpler for GCC compiler's code generator to *remember* the whole register is allocated for *char* variable and it can be sure the highest bits will not contain any noise at any point.

After, we also see new instruction `NOT`. This instruction inverts all bits in operand. It can be said, it is synonym to the `XOR ECX, 0ffffffffh` instruction. `NOT` and following `ADD` calculating pointer difference and subtracting 1. At the beginning `ECX`, where pointer to *str* is stored, inverted and 1 is subtracted from it.

See also: "Signed number representations" (32).

In other words, at the end of function, just after loop body, these operations are executed:

```
ecx=str;
eax=eos;
ecx=(-ecx)-1;
eax=eax+ecx
return eax
```

...and this is effectively equivalent to:

```
ecx=str;
eax=eos;
eax=eax-ecx;
eax=eax-1;
return eax
```

Why GCC decided it would be better? I cannot be sure. But I'm sure the both variants are effectively equivalent in efficiency sense.

13.2 ARM

13.2.1 Non-optimizing Xcode (LLVM) + ARM mode

Listing 13.2: Non-optimizing Xcode (LLVM) + ARM mode

```
_strlen

eos  = -8
str  = -4

     SUB    SP, SP, #8 ; allocate 8 bytes for local variables
     STR    R0, [SP,#8+str]
     LDR    R0, [SP,#8+str]
     STR    R0, [SP,#8+eos]

loc_2CB8 ; CODE XREF: _strlen+28
     LDR    R0, [SP,#8+eos]
     ADD    R1, R0, #1
     STR    R1, [SP,#8+eos]
     LDRSB  R0, [R0]
     CMP    R0, #0
     BEQ    loc_2CD4
     B      loc_2CB8
loc_2CD4 ; CODE XREF: _strlen+24
     LDR    R0, [SP,#8+eos]
     LDR    R1, [SP,#8+str]
     SUB    R0, R0, R1 ; R0=eos-str
     SUB    R0, R0, #1 ; R0=R0-1
     ADD    SP, SP, #8 ; deallocate 8 bytes for local variables
     BX     LR
```

Non-optimizing LLVM generates too much code, however, here we can see how function works with local variables in the stack. There are only two local variables in our function, *eos* and *str*.

In this listing, generated by IDA, I renamed *var_8* and *var_4* into *eos* and *str* manually.

So, first instructions are just saves input value in *str* and *eos*.

Loop body is beginning at *loc_2CB8* label.

First three instruction in loop body (`LDR`, `ADD`, `STR`) loads *eos* value into `R0`, then value is incremented and it is saved back into *eos* local variable located in the stack.

The next "`LDRSB R0, [R0]`" (*Load Register Signed Byte*) instruction loading byte from memory at `R0` address and sign-extends it to 32-bit. This is similar to `MOVSX` (13.1.1)

instruction in x86. The compiler treating this byte as signed since *char* type in C standard is signed. I already wrote about it (13.1.1) in this section, but related to x86.

It is should be noted, it is impossible in ARM to use 8-bit part or 16-bit part of 32-bit register separately of the whole register, as it is in x86. Apparently, it is because x86 has a huge history of compatibility with its ancestors like 16-bit 8086 and even 8-bit 8080, but ARM was developed from scratch as 32-bit RISC-processor. Consequently, in order to process separate bytes in ARM, one have to use 32-bit registers anyway.

So, LDRSB loads symbol from string into R0, one by one. Next CMP and BEQ instructions checks, if loaded symbol is 0. If not 0, control passing to loop body begin. And if 0, loop is finishing.

At the end of function, a difference between *eos* and *str* is calculated, 1 is also subtracting, and resulting value is returned via R0.

N.B. Registers was not saved in this function. That's because by ARM calling convention, R0-R3 registers are "scratch registers", they are intended for arguments passing, its values may not be restored upon function exit since calling function will not use them anymore. Consequently, they may be used for anything we want. Other registers are not used here, so that is why we have nothing to save on the stack. Thus, control may be returned back to calling function by simple jump (BX), to address in the LR register.

13.2.2 Optimizing Xcode (LLVM) + thumb mode

Listing 13.3: Optimizing Xcode (LLVM) + thumb mode

```
_strlen
            MOV         R1, R0

loc_2DF6                            ; CODE XREF: _strlen+8
            LDRB.W      R2, [R1],#1
            CMP         R2, #0
            BNE         loc_2DF6
            MVNS        R0, R0
            ADD         R0, R1
            BX          LR
```

As optimizing LLVM concludes, space on the stack for *eos* and *str* may not be allocated, and these variables may always be stored right in registers. Before loop body beginning, *str* will always be in R0, and *eos*—in R1.

''LDRB.W R2, [R1],#1'' instruction loads byte from memory at the address R1 into R2, sign-extending it to 32-bit value, but not only that. #1 at the instruction's end calling "Post-indexed addressing", this means, 1 is to be added to the R1 after byte load. That's convenient when accessing arrays.

There is no such addressing mode in x86, but it is present in some other processors, even on PDP-11. There is a legend the pre-increment, post-increment, pre-

decrement and post-decrement modes in PDP-11, were "guilty" in appearance such C language (which developed on PDP-11) constructs as *ptr++, *++ptr, *ptr--, *--ptr. By the way, this is one of hard to memorize C feature. This is how it is:

C term	ARM term	C statement	how it works
Post-increment	post-indexed addressing	`*ptr++`	use `*ptr` value, then increment `ptr` pointer
Post-decrement	post-indexed addressing	`*ptr--`	use `*ptr` value, then decrement `ptr` pointer
Pre-increment	pre-indexed addressing	`*++ptr`	increment `ptr` pointer, then use `*ptr` value
Pre-decrement	post-indexed addressing	`*--ptr`	decrement `ptr` pointer, then use `*ptr` value

Dennis Ritchie (one of C language creators) mentioned that it is, probably, was invented by Ken Thompson (another C creator) because this processor feature was present in PDP-7 [28] [29]. Thus, C language compilers may use it, if it is present in target processor.

Then one may spot `CMP` and BNE[2] in loop body, these instructions continue operation until 0 will be met in string.

`MVNS`[3] (inverting all bits, `NOT` in x86 analogue) instructions and `ADD` computes $eos - str - 1$. In fact, these two instructions computes $R0 = str + eos$, which is effectively equivalent to what was in source code, and why it is so, I already described here (13.1.5).

Apparently, LLVM, just like GCC, concludes this code will be shorter, or faster.

13.2.3 Optimizing Keil + ARM mode

Listing 13.4: Optimizing Keil + ARM mode

```
_strlen
                MOV     R1, R0

loc_2C8 ; CODE XREF: _strlen+14
                LDRB    R2, [R1],#1
                CMP     R2, #0
                SUBEQ   R0, R1, R0
                SUBEQ   R0, R0, #1
                BNE     loc_2C8
                BX      LR
```

[2] (PowerPC, ARM) Branch if Not Equal
[3] MoVe Not

Almost the same what we saw before, with the exception the $str - eos - 1$ expression may be computed not at the function's end, but right in loop body. -EQsuffix, as we may recall, means the instruction will be executed only if operands in executed before CMP were equal to each other. Thus, if 0 will be in the R0 register, both SUBEQ instructions are to be executed and result is leaving in the R0 register.

Chapter 14

Division by 9

Very simple function:

```
int f(int a)
{
        return a/9;
};
```

14.1 x86

…is compiled in a very predictable way:

Listing 14.1: MSVC

```
_a$ = 8             ; size = 4
_f    PROC
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _a$[ebp]
    cdq            ; sign extend EAX to EDX:EAX
    mov    ecx, 9
    idiv   ecx
    pop    ebp
    ret    0
_f  ENDP
```

IDIV divides 64-bit number stored in the EDX:EAX register pair by value in the ECX register. As a result, EAX will contain quotient[1], and EDX —remainder. Result is

[1] result of division

returning from the f() function in the EAX register, so, the value is not moved anymore after division operation, it is in right place already. Since IDIV requires value in the EDX:EAX register pair, CDQ instruction (before IDIV) extending value in the EAX to 64-bit value taking value sign into account, just as MOVSX (13.1.1) does. If we turn optimization on (/Ox), we got:

Listing 14.2: Optimizing MSVC

```
_a$ = 8                         ; size = 4
_f    PROC

    mov    ecx, DWORD PTR _a$[esp-4]
    mov    eax, 954437177   ; 38e38e39H
    imul   ecx
    sar    edx, 1
    mov    eax, edx
    shr    eax, 31          ; 0000001fH
    add    eax, edx
    ret    0
_f    ENDP
```

This is —division by multiplication. Multiplication operation works much faster. And it is possible to use the trick [2] to produce a code which is effectively equivalent and faster.

This is also called “strength reduction” in compiler optimization.

GCC 4.4.1 even without optimization turned on, generates almost the same code as MSVC with optimization turned on:

Listing 14.3: Non-optimizing GCC 4.4.1

```
        public f
f       proc near

arg_0   = dword ptr  8

        push    ebp
        mov     ebp, esp
        mov     ecx, [ebp+arg_0]
        mov     edx, 954437177 ; 38E38E39h
        mov     eax, ecx
        imul    edx
        sar     edx, 1
        mov     eax, ecx
        sar     eax, 1Fh
```

[2] Read more about division by multiplication in [35, 10-3]

```
        mov     ecx, edx
        sub     ecx, eax
        mov     eax, ecx
        pop     ebp
        retn
f       endp
```

14.2 ARM

ARM processor, just like in any other "pure" RISC-processors, lacks division instruction It lacks also a single instruction for multiplication by 32-bit constant. By taking advantage of the one clever trick (or *hack*), it is possible to do division using only three instructions: addition, subtraction and bit shifts (17).

Here is an example of 32-bit number division by 10 from [20, 3.3 Division by a Constant]. Quotient and remainder on output.

```
; takes argument in a1
; returns quotient in a1, remainder in a2
; cycles could be saved if only divide or remainder is required
    SUB    a2, a1, #10             ; keep (x-10) for later
    SUB    a1, a1, a1, lsr #2
    ADD    a1, a1, a1, lsr #4
    ADD    a1, a1, a1, lsr #8
    ADD    a1, a1, a1, lsr #16
    MOV    a1, a1, lsr #3
    ADD    a3, a1, a1, asl #2
    SUBS   a2, a2, a3, asl #1      ; calc (x-10) - (x/10)*10
    ADDPL  a1, a1, #1              ; fix-up quotient
    ADDMI  a2, a2, #10             ; fix-up remainder
    MOV    pc, lr
```

14.2.1 Optimizing Xcode (LLVM) + ARM mode

```
__text:00002C58 39 1E 08 E3 E3 18 43 E3  MOV     R1, 0x38E38E39
__text:00002C60 10 F1 50 E7              SMMUL   R0, R0, R1
__text:00002C64 C0 10 A0 E1              MOV     R1, R0,ASR#1
__text:00002C68 A0 0F 81 E0              ADD     R0, R1, R0,LSR↙
    ↳ #31
__text:00002C6C 1E FF 2F E1              BX      LR
```

This code is mostly the same to what was generated by optimizing MSVC and GCC. Apparently, LLVM use the same algorithm for constants generating.

Observant reader may ask, how MOV writes 32-bit value in register, while this is not possible in ARM mode. it is impossible indeed, but, as we see, there are 8 bytes per instruction instead of standard 4, in fact, there are two instructions. First instruction loading 0x8E39 value into low 16 bit of register and second instruction is in fact MOVT, it loading 0x383E into high 16-bit of register. IDA is aware of such sequences, and for the sake of compactness, reduced it to one single "pseudo-instruction".

SMMUL (*Signed Most Significant Word Multiply*) instruction multiply numbers treating them as signed numbers, and leaving high 32-bit part of result in the R0 register, dropping low 32-bit part of result.

''MOV R1, R0,ASR#1'' instruction is arithmetic shift right by one bit.

''ADD R0, R1, R0,LSR#31'' is $R0 = R1 + R0 >> 31$

As a matter of fact, there is no separate shifting instruction in ARM mode. Instead, an instructions like (MOV, ADD, SUB, RSB)[3] may be supplied by option, is the second operand must be shifted, if yes, by what value and how. ASR meaning *Arithmetic Shift Right*, LSR—*Logican Shift Right*.

14.2.2 Optimizing Xcode (LLVM) + thumb-2 mode

```
MOV             R1, 0x38E38E39
SMMUL.W         R0, R0, R1
ASRS            R1, R0, #1
ADD.W           R0, R1, R0,LSR#31
BX              LR
```

There are separate instructions for shifting in thumb mode, and one of them is used here—ASRS (arithmetic shift right).

14.2.3 Non-optimizing Xcode (LLVM) and Keil

Non-optimizing LLVM does not generate code we saw before in this section, but inserts a call to library function ___*divsi3* instead.

What about Keil: it inserts call to library function __*aeabi_idivmod* in all cases.

14.3 How it works

That's how division can be replaced by multiplication and division by 2^n numbers:

$$result = \frac{input}{divisor} = \frac{input \cdot \frac{2^n}{divisor}}{2^n} = \frac{input \cdot M}{2^n}$$

Where M is *magic*-coefficient.

That's how M can be computed:

[3] These instructions are also called "data processing instructions"

$$M = \frac{2^n}{divisor}$$

So these code snippets are usually have this form:

$$result = \frac{input \cdot M}{2^n}$$

n can be arbitrary number, it may be 32 (then high part of multiplication result is taked from EDX or RDX register), or 31 (then high part of multiplication result is shifted right additionally).

n is choosen in order to minimize error.

When doing signed division, sign of multiplication result also added to the output result.

Take a look at the difference:

```
int f3_32_signed(int a)
{
        return a/3;
};

unsigned int f3_32_unsigned(unsigned int a)
{
        return a/3;
};
```

In the unsigned version of function, *magic*-coefficient is 0xAAAAAAAB and multiplication result is divided by $2^3 3$.

In the signed version of function, *magic*-coefficient is 0x55555556 and multiplication result is divided by 2^{32}. Sign also taken from multiplication result: high 32 bits of result is shifted by 31 (leaving sign in least significant bit of EAX). 1 is added to the final result if sign is negative.

Listing 14.4: MSVC 2012 /Ox

```
_f3_32_unsigned PROC
        mov     eax, -1431655765                        ; ↙
    ↳ aaaaaaabH
        mul     DWORD PTR _a$[esp-4] ; unsigned multiply
        shr     edx, 1
        mov     eax, edx
        ret     0
_f3_32_unsigned ENDP

_f3_32_signed PROC
        mov     eax, 1431655766                         ; ↙
    ↳ 55555556H
```

```
        imul    DWORD PTR _a$[esp-4] ; signed multiply
        mov     eax, edx
        shr     eax, 31                                 ; ↙
    ↳ 0000001fH
        add     eax, edx ; add 1 if sign is negative
        ret     0
_f3_32_signed ENDP
```

Read more about it in [35, 10-3].

14.4 Getting divisor

14.4.1 Variant #1

Often, the code has a form of:

```
        mov     eax, MAGICAL CONSTANT
        imul    input value
        sar     edx, SHIFTING COEFFICIENT ; signed division by ↙
    ↳ 2^x using arithmetic shift right
        mov     eax, edx
        shr     eax, 31
        add     eax, edx
```

Let's denote 32-bit *magic*-coefficient as M, shifting coefficient by C and divisor by D.

The divisor we need to get is:

$$D = \frac{2^{32+C}}{M}$$

For example:

Listing 14.5: Optimizing MSVC 2012

```
        mov     eax, 2021161081                         ; ↙
    ↳ 78787879H
        imul    DWORD PTR _a$[esp-4]
        sar     edx, 3
        mov     eax, edx
        shr     eax, 31                                 ; ↙
    ↳ 0000001fH
        add     eax, edx
```

This is:

$$D = \frac{2^{32+3}}{2021161081}$$

Numbers are larger than 32-bit ones, so I use Wolfram Mathematica for convenience:

Listing 14.6: Wolfram Mathematica

```
In[1]:=N[2^(32+3)/2021161081]
Out[1]:=17.
```

So the divisor from the code I used for example is 17.

As of x64 division, things are the same, but 2^{64} should be used instead of 2^{32}:

```
uint64_t f1234(uint64_t a)
{
        return a/1234;
};
```

Listing 14.7: MSVC 2012 x64 /Ox

```
f1234   PROC
        mov     rax, 7653754429286296943                ; 6↙
    ↳ a37991a23aead6fH
        mul     rcx
        shr     rdx, 9
        mov     rax, rdx
        ret     0
f1234   ENDP
```

Listing 14.8: Wolfram Mathematica

```
In[1]:=N[2^(64+9)/16^^6a37991a23aead6f]
Out[1]:=1234.
```

14.4.2 Variant #2

A variant with omitted arithmetic shift is also exist:

```
                mov     eax, 55555556h ; 1431655766
                imul    ecx
                mov     eax, edx
                shr     eax, 1Fh
```

The method of getting divisor is simplified:

$$D = \frac{2^{32}}{M}$$

As of my example, this is:

$$D = \frac{2^{32}}{1431655766}$$

And again I use Wolfram Mathematica:

Listing 14.9: Wolfram Mathematica

```
In[1]:=N[2^32/16^^55555556]
Out[1]:=3.
```

The divisor is 3.

Chapter 15

Working with FPU

FPU[1]—is a device within main CPU specially designed to deal with floating point numbers.

It was called coprocessor in past. It stay aside of the main CPU and looks like programmable calculator in some way and.

It is worth to study stack machines[2] before FPU studying, or learn Forth language basics[3].

It is interesting to know that in past (before 80486 CPU) coprocessor was a separate chip and it was not always settled on motherboard. It was possible to buy it separately and install [4].

Starting at 80486 DX CPU, FPU is always present in it.

`FWAIT` instruction may remind us that fact—it switches CPU to waiting state, so it can wait until FPU finishes its work. Another rudiment is the fact that FPU-instruction opcodes are started with so called "escape"-opcodes (`D8..DF`), i.e., opcodes passed into FPU.

FPU has a stack capable to hold 8 80-bit registers, each register can hold a number in IEEE 754[5]format.

[1]Floating-point unit

[2]http://en.wikipedia.org/wiki/Stack_machine

[3]http://en.wikipedia.org/wiki/Forth_(programming_language)

[4]For example, John Carmack used fixed-point arithmetic values in his Doom video game, stored in 32-bit GPR registers (16 bit for intergral part and another 16 bit for fractional part), so the Doom could work on 32-bit computer without FPU, i.e., 80386 and 80486 SX

[5]http://en.wikipedia.org/wiki/IEEE_754-2008

C/C++ language offer at least two floating number types, *float* (*single-precision*[6], 32 bits) [7] and *double* (*double-precision*[8], 64 bits).

GCC also supports *long double* type (*extended precision*[9], 80 bit) but MSVC is not.

float type requires the same number of bits as *int* type in 32-bit environment, but number representation is completely different.

Number consisting of sign, significand (also called *fraction*) and exponent.

Function having *float* or *double* among argument list is getting the value via stack. If function returns *float* or *double* value, it leaves the value in the `ST(0)` register —at top of FPU stack.

15.1 Simple example

Let's consider simple example:

```
double f (double a, double b)
{
        return a/3.14 + b*4.1;
};
```

15.1.1 x86

Compile it in MSVC 2010:

Listing 15.1: MSVC 2010

```
CONST    SEGMENT
__real@4010666666666666 DQ 04010666666666666r    ; 4.1
CONST    ENDS
CONST    SEGMENT
__real@40091eb851eb851f DQ 040091eb851eb851fr    ; 3.14
CONST    ENDS
_TEXT    SEGMENT
_a$ = 8          ; size = 8
_b$ = 16         ; size = 8
```

[6] http://en.wikipedia.org/wiki/Single-precision_floating-point_format

[7] single precision float numbers format is also addressed in the *Working with the float type as with a structure* (18.6.2) section

[8] http://en.wikipedia.org/wiki/Double-precision_floating-point_format

[9] http://en.wikipedia.org/wiki/Extended_precision

```
_f  PROC
    push   ebp
    mov    ebp, esp
    fld    QWORD PTR _a$[ebp]

; current stack state: ST(0) = _a

    fdiv   QWORD PTR __real@40091eb851eb851f

; current stack state: ST(0) = result of _a divided by 3.13

    fld    QWORD PTR _b$[ebp]

; current stack state: ST(0) = _b; ST(1) = result of _a divided ↙
    ↳ by 3.13

    fmul   QWORD PTR __real@4010666666666666

; current stack state: ST(0) = result of _b * 4.1; ST(1) = ↙
    ↳ result of _a divided by 3.13

    faddp  ST(1), ST(0)

; current stack state: ST(0) = result of addition

    pop    ebp
    ret    0
_f  ENDP
```

FLD takes 8 bytes from stack and load the number into the ST(0) register, automatically converting it into internal 80-bit format *extended precision*).

FDIV divides value in the ST(0) register by number storing at address __real@40091eb851 3.14 value is coded there. Assembler syntax missing floating point numbers, so, what we see here is hexadecimal representation of *3.14* number in 64-bit IEEE 754 encoded.

After FDIV execution, ST(0) will hold quotient[10].

By the way, there is also FDIVP instruction, which divides ST(1) by ST(0), popping both these values from stack and then pushing result. If you know Forth language[11], you will quickly understand that this is stack machine[12].

Next FLD instruction pushing *b* value into stack.

After that, quotient is placed to the ST(1) register, and the ST(0) will hold *b* value.

[10] result of division

[11] http://en.wikipedia.org/wiki/Forth_(programming_language)

[12] http://en.wikipedia.org/wiki/Stack_machine

Next FMUL instruction do multiplication: *b* from the ST(0) register by value at __real@4010666666666666 (4.1 number is there) and leaves result in the ST(0) register.

Very last FADDP instruction adds two values at top of stack, storing result to the ST(1) register and then popping value at ST(1), hereby leaving result at top of stack in the ST(0).

The function must return result in the ST(0) register, so, after FADDP there are no any other instructions except of function epilogue.

GCC 4.4.1 (with -O3 option) emits the same code, however, slightly different:

Listing 15.2: Optimizing GCC 4.4.1

```
                public f
f               proc near

arg_0           = qword ptr  8
arg_8           = qword ptr  10h

                push    ebp
                fld     ds:dbl_8048608 ; 3.14

; stack state now: ST(0) = 3.13

                mov     ebp, esp
                fdivr   [ebp+arg_0]

; stack state now: ST(0) = result of division

                fld     ds:dbl_8048610 ; 4.1

; stack state now: ST(0) = 4.1, ST(1) = result of division

                fmul    [ebp+arg_8]

; stack state now: ST(0) = result of multiplication, ST(1) = ↙
    ↳ result of division

                pop     ebp
                faddp   st(1), st

; stack state now: ST(0) = result of addition

                retn
f               endp
```

The difference is that, first of all, 3.14 is pushed to stack (into ST(0)), and then value in arg_0 is divided by value in the ST(0) register.

FDIVR meaning *Reverse Divide* —to divide with divisor and dividend swapped with each other. There is no likewise instruction for multiplication since multiplication is commutative operation, so we have just FMUL without its -R counterpart.

FADDP adding two values but also popping one value from stack. After that operation, ST(0) holds the sum.

This fragment of disassembled code was produced using IDA which named the ST(0) register as ST for short.

15.1.2 ARM: Optimizing Xcode (LLVM) + ARM mode

Until ARM has floating standardized point support, several processor manufacturers may add their own instructions extensions. Then, VFP (*Vector Floating Point*) was standardized.

One important difference from x86, there you working with FPU-stack, but here, in ARM, there are no any stack, you work just with registers.

```
f
                VLDR            D16, =3.14
                VMOV            D17, R0, R1 ; load a
                VMOV            D18, R2, R3 ; load b
                VDIV.F64        D16, D17, D16 ; a/3.14
                VLDR            D17, =4.1
                VMUL.F64        D17, D18, D17 ; b*4.1
                VADD.F64        D16, D17, D16 ; +
                VMOV            R0, R1, D16
                BX              LR

dbl_2C98        DCFD 3.14               ; DATA XREF: f
dbl_2CA0        DCFD 4.1                ; DATA XREF: f+10
```

So, we see here new registers used, with D prefix. These are 64-bit registers, there are 32 of them, and these can be used both for floating-point numbers (double) but also for SIMD (it is called NEON here in ARM).

There are also 32 32-bit S-registers, they are intended to be used for single precision floating pointer numbers (float).

It is easy to remember: D-registers are intended for double precision numbers, while S-registers —for single precision numbers.

Both (3.14 and 4.1) constants are stored in memory in IEEE 754 form.

VLDR and VMOV instructions, as it can be easily deduced, are analogous to the LDR and MOV instructions, but they works with D-registers. It should be noted that these instructions, just like D-registers, are intended not only for floating point numbers, but can be also used for SIMD (NEON) operations and this will also be revealed soon.

Arguments are passed to function in common way, via R-registers, however, each number having double precision has size 64-bits, so, for passing each, two R-registers are needed.

‘‘VMOV D17, R0, R1’’ at the very beginning, composing two 32-bit values from R0 and R1 into one 64-bit value and saves it to D17.

‘‘VMOV R0, R1, D16’’ is inverse operation, what was in D16 leaving in two R0 and R1 registers, since double-precision number, needing 64 bits for storage, is returning in the R0 and R1 registers pair.

VDIV, VMUL and VADD, are instruction for floating point numbers processing, computing, quotient[13], product[14] and sum[15], respectively.

The code for thumb-2 is same.

15.1.3 ARM: Optimizing Keil + thumb mode

```
f
                PUSH    {R3-R7,LR}
                MOVS    R7, R2
                MOVS    R4, R3
                MOVS    R5, R0
                MOVS    R6, R1
                LDR     R2, =0x66666666
                LDR     R3, =0x40106666
                MOVS    R0, R7
                MOVS    R1, R4
                BL      __aeabi_dmul
                MOVS    R7, R0
                MOVS    R4, R1
                LDR     R2, =0x51EB851F
                LDR     R3, =0x40091EB8
                MOVS    R0, R5
                MOVS    R1, R6
                BL      __aeabi_ddiv
                MOVS    R2, R7
                MOVS    R3, R4
                BL      __aeabi_dadd
                POP     {R3-R7,PC}

dword_364       DCD 0x66666666          ; DATA XREF: f+A
dword_368       DCD 0x40106666          ; DATA XREF: f+C
dword_36C       DCD 0x51EB851F          ; DATA XREF: f+1A
```

[13] result of division
[14] result of multiplication
[15] result of addition

```
dword_370       DCD 0x40091EB8          ; DATA XREF: f+1C
```

Keil generates for processors not supporting FPU or NEON. So, double-precision floating numbers are passed via generic R-registers, and instead of FPU-instructions, service library functions are called (like `__aeabi_dmul`, `__aeabi_ddiv`, `__aeabi_dadd`) which emulates multiplication, division and addition floating-point numbers. Of course, that is slower than FPU-coprocessor, but it is better than nothing.

By the way, similar FPU-emulating libraries were very popular in x86 world when coprocessors were rare and expensive, and were installed only on expensive computers.

FPU-coprocessor emulating called *soft float* or *armel* in ARM world, while using coprocessor's FPU-instructions called *hard float* or *armhf*.

For example, Linux kernel for Raspberry Pi is compiled in two variants. In *soft float* case, arguments will be passed via R-registers, and in *hard float* case —via D-registers.

And that is what do not let you use e.g. armhf-libraries from armel-code or vice versa, so that is why all code in Linux distribution must be compiled according to the chosen calling convention.

15.2 Passing floating point number via arguments

```
#include <math.h>
#include <stdio.h>

int main ()
{
        printf ("32.01 ^ 1.54 = %lf\n", pow (32.01,1.54));

        return 0;
}
```

15.2.1 x86

Let's see what we got in (MSVC 2010):

Listing 15.3: MSVC 2010

```
CONST    SEGMENT
__real@40400147ae147ae1 DQ 040400147ae147ae1r    ; 32.01
__real@3ff8a3d70a3d70a4 DQ 03ff8a3d70a3d70a4r    ; 1.54
CONST    ENDS

_main    PROC
    push   ebp
```

```
    mov    ebp, esp
    sub    esp, 8  ; allocate place for the first variable
    fld    QWORD PTR __real@3ff8a3d70a3d70a4
    fstp   QWORD PTR [esp]
    sub    esp, 8  ; allocate place for the second variable
    fld    QWORD PTR __real@40400147ae147ae1
    fstp   QWORD PTR [esp]
    call   _pow
    add    esp, 8  ; "return back" place of one variable.

; in local stack here 8 bytes still reserved for us.
; result now in ST(0)

    fstp   QWORD PTR [esp] ; move result from ST(0) to local ↙
    ↪ stack for printf()
    push   OFFSET $SG2651
    call   _printf
    add    esp, 12
    xor    eax, eax
    pop    ebp
    ret    0
_main    ENDP
```

FLD and FSTP are moving variables from/to data segment to FPU stack. pow()[16] taking both values from FPU-stack and returns result in the ST(0) register. printf() takes 8 bytes from local stack and interpret them as *double* type variable.

15.2.2 ARM + Non-optimizing Xcode (LLVM) + thumb-2 mode

```
_main

var_C           = -0xC

                PUSH            {R7,LR}
                MOV             R7, SP
                SUB             SP, SP, #4
                VLDR            D16, =32.01
                VMOV            R0, R1, D16
                VLDR            D16, =1.54
                VMOV            R2, R3, D16
                BLX             _pow
                VMOV            D16, R0, R1
```

[16] standard C function, raises a number to the given power

```
                MOV             R0, 0xFC1 ; "32.01 ^ 1.54 = %lf\↙
    ↳ n"
                ADD             R0, PC
                VMOV            R1, R2, D16
                BLX             _printf
                MOVS            R1, 0
                STR             R0, [SP,#0xC+var_C]
                MOV             R0, R1
                ADD             SP, SP, #4
                POP             {R7,PC}

dbl_2F90        DCFD 32.01              ; DATA XREF: _main+6
dbl_2F98        DCFD 1.54               ; DATA XREF: _main+E
```

As I wrote before, 64-bit floating pointer numbers passing in R-registers pairs. This is code is redundant for a little (certainly because optimization is turned off), because, it is actually possible to load values into R-registers straightforwardly without touching D-registers.

So, as we see, `_pow` function receiving first argument in `R0` and `R1`, and the second one in `R2` and `R3`. Function leaves result in `R0` and `R1`. Result of `_pow` is moved into `D16`, then in `R1` and `R2` pair, from where `printf()` will take this number.

15.2.3 ARM + Non-optimizing Keil + ARM mode

```
_main
                STMFD   SP!, {R4-R6,LR}
                LDR     R2, =0xA3D70A4  ; y
                LDR     R3, =0x3FF8A3D7
                LDR     R0, =0xAE147AE1 ; x
                LDR     R1, =0x40400147
                BL      pow
                MOV     R4, R0
                MOV     R2, R4
                MOV     R3, R1
                ADR     R0, a32_011_54Lf ; "32.01 ^ 1.54 = %lf\n↙
    ↳ "
                BL      __2printf
                MOV     R0, #0
                LDMFD   SP!, {R4-R6,PC}

y               DCD 0xA3D70A4           ; DATA XREF: _main+4
dword_520       DCD 0x3FF8A3D7          ; DATA XREF: _main+8
; double x
```

```
x               DCD 0xAE147AE1          ; DATA XREF: _main+C
dword_528       DCD 0x40400147          ; DATA XREF: _main+10
a32_011_54Lf    DCB "32.01 ^ 1.54 = %lf",0xA,0
                                        ; DATA XREF: _main+24
```

D-registers are not used here, only R-register pairs are used.

15.3 Comparison example

Let's try this:

```
double d_max (double a, double b)
{
        if (a>b)
                return a;

        return b;
};
```

15.3.1 x86

Despite simplicity of the function, it will be harder to understand how it works.

MSVC 2010 generated:

Listing 15.4: MSVC 2010

```
PUBLIC    _d_max
_TEXT    SEGMENT
_a$ = 8                 ; size = 8
_b$ = 16                ; size = 8
_d_max     PROC
    push   ebp
    mov    ebp, esp
    fld    QWORD PTR _b$[ebp]

; current stack state: ST(0) = _b
; compare _b (ST(0)) and _a, and pop register

    fcomp  QWORD PTR _a$[ebp]

; stack is empty here

    fnstsw ax
    test   ah, 5
```

```
    jp      SHORT $LN1@d_max

; we are here only if a>b

    fld     QWORD PTR _a$[ebp]
    jmp     SHORT $LN2@d_max
$LN1@d_max:
    fld     QWORD PTR _b$[ebp]
$LN2@d_max:
    pop     ebp
    ret     0
_d_max      ENDP
```

So, FLD loading _b into the ST(0) register.

FCOMP compares the value in the ST(0) register with what is in _a value and set C3/C2/C0 bits in FPU status word register. This is 16-bit register reflecting current state of FPU.

For now C3/C2/C0 bits are set, but unfortunately, CPU before Intel P6 [17] has not any conditional jumps instructions which are checking these bits. Probably, it is a matter of history (remember: FPU was separate chip in past). Modern CPU starting at Intel P6 has FCOMI/FCOMIP/FUCOMI/FUCOMIP instructions —which does the same, but modifies CPU flags ZF/PF/CF.

After bits are set, the FCOMP instruction popping one variable from stack. This is what distinguish it from FCOM, which is just comparing values, leaving the stack at the same state.

FNSTSW copies FPU status word register to the AX. Bits C3/C2/C0 are placed at positions 14/10/8, they will be at the same positions in the AX register and all they are placed in high part of the AX —AH.

- If b>a in our example, then C3/C2/C0 bits will be set as following: 0, 0, 0.
- If a>b, then bits will be set: 0, 0, 1.
- If a=b, then bits will be set: 1, 0, 0.

After `test ah, 5` execution, bits C3 and C1 will be set to 0, but at positions 0 and 2 (in the AH registers) C0 and C2 bits will be leaved.

Now let's talk about parity flag. Another notable epoch rudiment:

> One common reason to test the parity flag actually has nothing to do with parity. The FPU has four condition flags (C0 to C3), but

[17] Intel P6 is Pentium Pro, Pentium II, etc

they can not be tested directly, and must instead be first copied to the flags register. When this happens, C0 is placed in the carry flag, C2 in the parity flag and C3 in the zero flag. The C2 flag is set when e.g. incomparable floating point values (NaN or unsupported format) are compared with the FUCOM instructions.[18]

This flag is to be set to 1 if ones number is even. And to 0 if odd.

Thus, PF flag will be set to 1 if both C0 and C2 are set to 0 or both are 1. And then following JP (*jump if PF==1*) will be triggered. If we recall values of the C3/C2/C0 for various cases, we will see the conditional jump JP will be triggered in two cases: if b>a or a==b (C3 bit is already not considering here since it was cleared while execution of the `test ah, 5` instruction).

It is all simple thereafter. If conditional jump was triggered, FLD will load the _b value to the ST(0) register, and if it is not triggered, the value of the _a variable will be loaded.

But it is not over yet!

15.3.2 Now let's compile it with MSVC 2010 with optimization option /Ox

Listing 15.5: Optimizing MSVC 2010

```
_a$ = 8                 ; size = 8
_b$ = 16                ; size = 8
_d_max    PROC
    fld    QWORD PTR _b$[esp-4]
    fld    QWORD PTR _a$[esp-4]

; current stack state: ST(0) = _a, ST(1) = _b

    fcom    ST(1) ; compare _a and ST(1) = (_b)
    fnstsw    ax
    test    ah, 65                    ; 00000041H
    jne    SHORT $LN5@d_max
    fstp    ST(1)  ; copy ST(0) to ST(1) and pop register, leave↙
    ↳  (_a) on top

; current stack state: ST(0) = _a

    ret    0
$LN5@d_max:
```

```
    fstp    ST(0)  ; copy ST(0) to ST(0) and pop register, leave↙
    ↪  (_b) on top

; current stack state: ST(0) = _b

    ret    0
_d_max    ENDP
```

FCOM is distinguished from FCOMP in that sense that it just comparing values and leaves FPU stack in the same state. Unlike previous example, operands here in reversed order. And that is why result of comparison in the C3/C2/C0 will be different:

- If a>b in our example, then C3/C2/C0 bits will be set as: 0, 0, 0.
- If b>a, then bits will be set as: 0, 0, 1.
- If a=b, then bits will be set as: 1, 0, 0.

It can be said, `test ah, 65` instruction just leaves two bits —C3 and C0. Both will be zeroes if a>b: in that case JNE jump will not be triggered. Then `FSTP ST(1)` is following —this instruction copies value in the `ST(0)` into operand and popping one value from FPU stack. In other words, the instruction copies `ST(0)` (where _a value is now) into the `ST(1)`. After that, two values of the _a are at the top of stack now. After that, one value is popping. After that, `ST(0)` will contain _a and function is finished.

Conditional jump JNE is triggered in two cases: of b>a or a==b. `ST(0)` into `ST(0)` will be copied, it is just like idle (NOP) operation, then one value is popping from stack and top of stack (`ST(0)`) will contain what was in the `ST(1)` before (that is _b). Then function finishes. The instruction used here probably since FPU has no instruction to pop value from stack and not to store it anywhere.

Well, but it is still not over.

15.3.3 GCC 4.4.1

Listing 15.6: GCC 4.4.1

```
d_max proc near

b               = qword ptr -10h
a               = qword ptr -8
a_first_half    = dword ptr  8
a_second_half   = dword ptr  0Ch
b_first_half    = dword ptr  10h
b_second_half   = dword ptr  14h

    push    ebp
```

```
    mov     ebp, esp
    sub     esp, 10h

; put a and b to local stack:

    mov     eax, [ebp+a_first_half]
    mov     dword ptr [ebp+a], eax
    mov     eax, [ebp+a_second_half]
    mov     dword ptr [ebp+a+4], eax
    mov     eax, [ebp+b_first_half]
    mov     dword ptr [ebp+b], eax
    mov     eax, [ebp+b_second_half]
    mov     dword ptr [ebp+b+4], eax

; load a and b to FPU stack:

    fld     [ebp+a]
    fld     [ebp+b]

; current stack state: ST(0) - b; ST(1) - a

    fxch    st(1) ; this instruction swapping ST(1) and ST(0)

; current stack state: ST(0) - a; ST(1) - b

    fucompp    ; compare a and b and pop two values from stack, ↙
    ↳ i.e., a and b
    fnstsw  ax ; store FPU status to AX
    sahf       ; load SF, ZF, AF, PF, and CF flags state from AH
    setnbe  al ; store 1 to AL if CF=0 and ZF=0
    test    al, al             ; AL==0 ?
    jz      short loc_8048453 ; yes
    fld     [ebp+a]
    jmp     short locret_8048456

loc_8048453:
    fld     [ebp+b]

locret_8048456:
    leave
    retn
d_max endp
```

FUCOMPP —is almost like FCOM, but popping both values from stack and handling

"not-a-numbers" differently.

More about *not-a-numbers*:

FPU is able to deal with a special values which are *not-a-numbers* or NaNs[19]. These are infinity, result of dividing by 0, etc. Not-a-numbers can be "quiet" and "signaling". It is possible to continue to work with "quiet" NaNs, but if one try to do any operation with "signaling" NaNs —an exception will be raised.

FCOM will raise exception if any operand —NaN. FUCOM will raise exception only if any operand —signaling NaN (SNaN).

The following instruction is SAHF —this is rare instruction in the code which is not use FPU. 8 bits from AH is movinto into lower 8 bits of CPU flags in the following order: SF:ZF:-:AF:-:PF:-:CF <- AH.

Let's remember the FNSTSW is moving interesting for us bits C3/C2/C0 into the AH and they will be in positions 6, 2, 0 in the AH register.

In other words, fnstsw ax / sahf instruction pair is moving C3/C2/C0 into ZF, PF, CF CPU flags.

Now let's also recall, what values of the C3/C2/C0 bits will be set:

- If a is greater than b in our example, then C3/C2/C0 bits will be set as: 0, 0, 0.
- if a is less than b, then bits will be set as: 0, 0, 1.
- If a=b, then bits will be set: 1, 0, 0.

In other words, after FUCOMPP/FNSTSW/SAHF instructions, we will have these CPU flags states:

- If a>b, CPU flags will be set as: ZF=0, PF=0, CF=0.
- If a<b, then CPU flags will be set as: ZF=0, PF=0, CF=1.
- If a=b, then CPU flags will be set as: ZF=1, PF=0, CF=0.

How SETNBE instruction will store 1 or 0 to AL: it is depends of CPU flags. It is almost JNBE instruction counterpart, with the exception the SETcc[20] is storing 1 or 0 to the AL, but Jcc do actual jump or not. SETNBE store 1 only if CF=0 and ZF=0. If it is not true, 0 will be stored into AL.

Both CF is 0 and ZF is 0 simultaneously only in one case: if a>b.

Then one will be stored to the AL and the following JZ will not be triggered and function will return _a. In all other cases, _b will be returned.

But it is still not over.

[19] http://en.wikipedia.org/wiki/NaN

[20] *cc* is *condition code*

15.3.4 GCC 4.4.1 with -O3 optimization turned on

Listing 15.7: Optimizing GCC 4.4.1

```
                public d_max
d_max           proc near

arg_0           = qword ptr  8
arg_8           = qword ptr  10h

                push    ebp
                mov     ebp, esp
                fld     [ebp+arg_0] ; _a
                fld     [ebp+arg_8] ; _b

; stack state now: ST(0) = _b, ST(1) = _a
                fxch    st(1)

; stack state now: ST(0) = _a, ST(1) = _b
                fucom   st(1) ; compare _a and _b
                fnstsw  ax
                sahf
                ja      short loc_8048448

; store ST(0) to ST(0) (idle operation), pop value at top of ↙
    ↳ stack, leave _b at top
                fstp    st
                jmp     short loc_804844A

loc_8048448:
; store _a to ST(0), pop value at top of stack, leave _a at top
                fstp    st(1)

loc_804844A:
                pop     ebp
                retn
d_max           endp
```

It is almost the same except one: JA usage instead of SAHF. Actually, conditional jump instructions checking "larger", "lesser" or "equal" for unsigned number comparison (JA, JAE, JBE, JBE, JE/JZ, JNA, JNAE, JNB, JNBE, JNE/JNZ) are checking only CF and ZF flags. And C3/C2/C0 bits after comparison are moving into these flags exactly in the same fashion so conditional jumps will work here. JA will work if both CF are ZF zero.

Thereby, conditional jumps instructions listed here can be used after FNSTSW/SAHF instructions pair.

It seems, FPU C3/C2/C0 status bits was placed there intentionally so to map them to base CPU flags without additional permutations.

15.3.5 ARM + Optimizing Xcode (LLVM) + ARM mode

Listing 15.8: Optimizing Xcode (LLVM) + ARM mode

```
VMOV            D16, R2, R3 ; b
VMOV            D17, R0, R1 ; a
VCMPE.F64       D17, D16
VMRS            APSR_nzcv, FPSCR
VMOVGT.F64      D16, D17 ; copy b to D16
VMOV            R0, R1, D16
BX              LR
```

A very simple case. Input values are placed into the D17 and D16 registers and then compared with the help of VCMPE instruction. Just like in x86 coprocessor, ARM coprocessor has its own status and flags register, (FPSCR), since there is a need to store coprocessor-specific flags.

And just like in x86, there are no conditional jump instruction in ARM, checking bits in coprocessor status register, so there is VMRS instruction, copying 4 bits (N, Z, C, V) from the coprocessor status word into bits of *general* status (APSR register).

VMOVGT is analogue of MOVGT, instruction, to be executed if one operand is greater than other while comparing (*GT—Greater Than*).

If it will be executed, b value will be written into D16, stored at the moment in D17.

And if it will not be triggered, then a value will stay in the D16 register.

Penultimate instruction VMOV will prepare value in the D16 register for returning via R0 and R1 registers pair.

15.3.6 ARM + Optimizing Xcode (LLVM) + thumb-2 mode

Listing 15.9: Optimizing Xcode (LLVM) + thumb-2 mode

```
VMOV            D16, R2, R3 ; b
VMOV            D17, R0, R1 ; a
VCMPE.F64       D17, D16
VMRS            APSR_nzcv, FPSCR
IT GT
VMOVGT.F64      D16, D17
VMOV            R0, R1, D16
BX              LR
```

Almost the same as in previous example, howeverm slightly different. As a matter of fact, many instructions in ARM mode can be supplied by condition predicate, and the instruction is to be executed if condition is true.

But there is no such thing in thumb mode. There is no place in 16-bit instructions for spare 4 bits where condition can be encoded.

However, thumb-2 was extended to make possible to specify predicates to old thumb instructions.

Here, is the IDA-generated listing, we see `VMOVGT` instruction, the same as in previous example.

But in fact, usual `VMOV` is encoded there, but IDA added `-GT` suffix to it, since there is ‘‘`IT GT`’’ instruction placed right before.

`IT` instruction defines so-called *if-then block*. After the instruction, it is possible to place up to 4 instructions, to which predicate suffix will be added. In our example, ‘‘`IT GT`’’ meaning, the next instruction will be executed, if *GT* (*Greater Than*) condition is true.

Now more complex code fragment, by the way, from “Angry Birds” (for iOS):

Listing 15.10: Angry Birds Classic

```
ITE NE
VMOVNE          R2, R3, D16
VMOVEQ          R2, R3, D17
```

`ITE` meaning *if-then-else* and it encode suffixes for two next instructions. First instruction will execute if condition encoded in `ITE` (*NE, not equal*) will be true at the moment, and the second —if the condition will not be true. (Inverse condition of `NE` is `EQ` (*equal*)).

Slightly harder, and this fragment from “Angry Birds” as well:

Listing 15.11: Angry Birds Classic

```
ITTTT EQ
MOVEQ           R0, R4
ADDEQ           SP, SP, #0x20
POPEQ.W         {R8,R10}
POPEQ           {R4-R7,PC}
```

4 “T” symbols in instruction mnemonic means the 4 next instructions will be executed if condition is true. That’s why IDA added `-EQ` suffix to each 4 instructions.

And if there will be e.g. `ITEEE EQ` (*if-then-else-else-else*), then suffixes will be set as follows:

```
-EQ
-NE
-NE
-NE
```

Another fragment from “Angry Birds”:

Listing 15.12: Angry Birds Classic

```
CMP.W           R0, #0xFFFFFFFF
ITTE LE
SUBLE.W         R10, R0, #1
NEGLE           R0, R0
MOVGT           R10, R0
```

ITTE (*if-then-then-else*) means the 1st and 2nd instructions will be executed, if LE (*Less or Equal*) condition is true, and 3rd—if inverse condition (GT—*Greater Than*) is true.

Compilers usually are not generating all possible combinations. For example, it mentioned “Angry Birds” game (*classic* version for iOS) only these cases of IT instruction are used: IT, ITE, ITT, ITTE, ITTT, ITTTT. How I learnt this? In IDA it is possible to produce listing files, so I did it, but I also set in options to show 4 bytes of each opcodes . Then, knowing the high part of 16-bit opcode IT is 0xBF, I did this with grep:

```
cat AngryBirdsClassic.lst | grep " BF" | grep "IT" > results.lst
```

By the way, if to program in ARM assembly language manually for thumb-2 mode, with adding conditional suffixes, assembler will add IT instructions automatically, with respectable flags, where it is necessary.

15.3.7 ARM + Non-optimizing Xcode (LLVM) + ARM mode

Listing 15.13: Non-optimizing Xcode (LLVM) + ARM mode

```
b               = -0x20
a               = -0x18
val_to_return   = -0x10
saved_R7        = -4

                STR             R7, [SP,#saved_R7]!
                MOV             R7, SP
                SUB             SP, SP, #0x1C
                BIC             SP, SP, #7
                VMOV            D16, R2, R3
                VMOV            D17, R0, R1
                VSTR            D17, [SP,#0x20+a]
                VSTR            D16, [SP,#0x20+b]
                VLDR            D16, [SP,#0x20+a]
                VLDR            D17, [SP,#0x20+b]
```

```
                VCMPE.F64       D16, D17
                VMRS            APSR_nzcv, FPSCR
                BLE             loc_2E08
                VLDR            D16, [SP,#0x20+a]
                VSTR            D16, [SP,#0x20+val_to_return]
                B               loc_2E10

loc_2E08
                VLDR            D16, [SP,#0x20+b]
                VSTR            D16, [SP,#0x20+val_to_return]

loc_2E10
                VLDR            D16, [SP,#0x20+val_to_return]
                VMOV            R0, R1, D16
                MOV             SP, R7
                LDR             R7, [SP+0x20+b],#4
                BX              LR
```

Almost the same we already saw, but too much redundant code because of a and b variables storage in local stack, as well as returning value.

15.3.8 ARM + Optimizing Keil + thumb mode

Listing 15.14: Optimizing Keil + thumb mode

```
                PUSH    {R3-R7,LR}
                MOVS    R4, R2
                MOVS    R5, R3
                MOVS    R6, R0
                MOVS    R7, R1
                BL      __aeabi_cdrcmple
                BCS     loc_1C0
                MOVS    R0, R6
                MOVS    R1, R7
                POP     {R3-R7,PC}

loc_1C0
                MOVS    R0, R4
                MOVS    R1, R5
                POP     {R3-R7,PC}
```

Keil not generates special instruction for float numbers comparing since it cannot rely it will be supported on the target CPU, and it cannot be done by straightforward bitwise comparing. So there is called external library function for comparing:

`__aeabi_cdrcmple`. N.B. Comparison result is to be leaved in flags, so the following `BCS` (*Carry set - Greater than or equal*) instruction may work without any additional code.

15.4 x64

Read more here24 about how float point numbers are processed in x86-64.

Chapter 16

Arrays

Array is just a set of variables in memory, always lying next to each other, always has same type [1].

16.1 Simple example

```
#include <stdio.h>

int main()
{
    int a[20];
    int i;

    for (i=0; i<20; i++)
        a[i]=i*2;

    for (i=0; i<20; i++)
        printf ("a[%d]=%d\n", i, a[i]);

    return 0;
};
```

16.1.1 x86

Let's compile:

[1] AKA[2] "homogeneous container"

Listing 16.1: MSVC

```
_TEXT    SEGMENT
_i$ = -84                       ; size = 4
_a$ = -80                       ; size = 80
_main      PROC
    push   ebp
    mov    ebp, esp
    sub    esp, 84        ; 00000054H
    mov    DWORD PTR _i$[ebp], 0
    jmp    SHORT $LN6@main
$LN5@main:
    mov    eax, DWORD PTR _i$[ebp]
    add    eax, 1
    mov    DWORD PTR _i$[ebp], eax
$LN6@main:
    cmp    DWORD PTR _i$[ebp], 20   ; 00000014H
    jge    SHORT $LN4@main
    mov    ecx, DWORD PTR _i$[ebp]
    shl    ecx, 1
    mov    edx, DWORD PTR _i$[ebp]
    mov    DWORD PTR _a$[ebp+edx*4], ecx
    jmp    SHORT $LN5@main
$LN4@main:
    mov    DWORD PTR _i$[ebp], 0
    jmp    SHORT $LN3@main
$LN2@main:
    mov    eax, DWORD PTR _i$[ebp]
    add    eax, 1
    mov    DWORD PTR _i$[ebp], eax
$LN3@main:
    cmp    DWORD PTR _i$[ebp], 20    ; 00000014H
    jge    SHORT $LN1@main
    mov    ecx, DWORD PTR _i$[ebp]
    mov    edx, DWORD PTR _a$[ebp+ecx*4]
    push   edx
    mov    eax, DWORD PTR _i$[ebp]
    push   eax
    push   OFFSET $SG2463
    call   _printf
    add    esp, 12        ; 0000000cH
    jmp    SHORT $LN2@main
$LN1@main:
    xor    eax, eax
```

```
    mov     esp, ebp
    pop     ebp
    ret     0
_main       ENDP
```

Nothing very special, just two loops: first is filling loop and second is printing loop. `shl ecx, 1` instruction is used for value multiplication by 2 in the ECX, more about below 17.3.1.

80 bytes are allocated on the stack for array, that is 20 elements of 4 bytes.

Here is what GCC 4.4.1 does:

Listing 16.2: GCC 4.4.1

```
                public main
main            proc near               ; DATA XREF: _start+17

var_70          = dword ptr -70h
var_6C          = dword ptr -6Ch
var_68          = dword ptr -68h
i_2             = dword ptr -54h
i               = dword ptr -4

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 70h
                mov     [esp+70h+i], 0          ; i=0
                jmp     short loc_804840A

loc_80483F7:
                mov     eax, [esp+70h+i]
                mov     edx, [esp+70h+i]
                add     edx, edx                ; edx=i*2
                mov     [esp+eax*4+70h+i_2], edx
                add     [esp+70h+i], 1          ; i++

loc_804840A:
                cmp     [esp+70h+i], 13h
                jle     short loc_80483F7
                mov     [esp+70h+i], 0
                jmp     short loc_8048441

loc_804841B:
                mov     eax, [esp+70h+i]
                mov     edx, [esp+eax*4+70h+i_2]
```

```
                mov     eax, offset aADD ; "a[%d]=%d\n"
                mov     [esp+70h+var_68], edx
                mov     edx, [esp+70h+i]
                mov     [esp+70h+var_6C], edx
                mov     [esp+70h+var_70], eax
                call    _printf
                add     [esp+70h+i], 1

loc_8048441:
                cmp     [esp+70h+i], 13h
                jle     short loc_804841B
                mov     eax, 0
                leave
                retn
main            endp
```

By the way, *a* variable has *int** type (the pointer to *int*) —you can try to pass a pointer to array to another function, but it much correctly to say the pointer to the first array element is passed (addresses of another element's places are calculated in obvious way). If to index this pointer as *a[idx]*, *idx* just to be added to the pointer and the element placed there (to which calculated pointer is pointing) returned.

An interesting example: string of characters like *"string"* is array of characters and it has *const char** type.Index can be applied to this pointer. And that is why it is possible to write like ‘‘`string`’’`[i]` —this is correct C/C++ expression!

16.1.2 ARM + Non-optimizing Keil + ARM mode

```
            EXPORT _main
_main
            STMFD   SP!, {R4,LR}
            SUB     SP, SP, #0x50      ; allocate place for 20 ↙
    ↳ int variables

; first loop

            MOV     R4, #0             ; i
            B       loc_4A0
loc_494
            MOV     R0, R4,LSL#1       ; R0=R4*2
            STR     R0, [SP,R4,LSL#2]  ; store R0 to SP+R4<<2 (↙
    ↳ same as SP+R4*4)
            ADD     R4, R4, #1         ; i=i+1
```

```
loc_4A0
            CMP     R4, #20             ; i<20?
            BLT     loc_494             ; yes, run loop body ↙
    ↳ again

; second loop

            MOV     R4, #0              ; i
            B       loc_4C4
loc_4B0
            LDR     R2, [SP,R4,LSL#2]   ; (second printf argument↙
    ↳ ) R2=*(SP+R4<<4) (same as *(SP+R4*4))
            MOV     R1, R4              ; (first printf argument)↙
    ↳  R1=i
            ADR     R0, aADD            ; "a[%d]=%d\n"
            BL      __2printf
            ADD     R4, R4, #1          ; i=i+1

loc_4C4
            CMP     R4, #20             ; i<20?
            BLT     loc_4B0             ; yes, run loop body ↙
    ↳ again
            MOV     R0, #0              ; value to return
            ADD     SP, SP, #0x50       ; deallocate place, ↙
    ↳ allocated for 20 int variables
            LDMFD   SP!, {R4,PC}
```

int type requires 32 bits for storage, or 4 bytes, so for storage of 20 *int* variables, 80 (0x50) bytes are needed, so that is why ‘‘SUB SP, SP, #0x50’’ instruction in function epilogue allocates exactly this amount of space in local stack.

In both first and second loops, i loop iterator will be placed in the R4 register.

A number to be written into array, is calculating as $i * 2$ which is effectively equivalent to shifting left by one bit, so ‘‘MOV R0, R4,LSL#1’’ instruction do this.

‘‘STR R0, [SP,R4,LSL#2]’’ writes R0 contents into array. Here is how a pointer to array element is to be calculated: SP pointing to array begin, R4 is i. So shift i left by 2 bits, that is effectively equivalent to multiplication by 4 (since each array element has size of 4 bytes) and add it to address of array begin.

The second loop has inverse ‘‘LDR R2, [SP,R4,LSL#2]’’, instruction, it loads from array value we need, and the pointer to it is calculated likewise.

16.1.3 ARM + Optimizing Keil + thumb mode

```
_main
```

```
          PUSH    {R4,R5,LR}
; allocate place for 20 int variables + one more variable
          SUB     SP, SP, #0x54

; first loop

          MOVS    R0, #0          ; i
          MOV     R5, SP          ; pointer to first array ↙
    ↳ element

loc_1CE
          LSLS    R1, R0, #1      ; R1=i<<1 (same as i*2)
          LSLS    R2, R0, #2      ; R2=i<<2 (same as i*4)
          ADDS    R0, R0, #1      ; i=i+1
          CMP     R0, #20         ; i<20?
          STR     R1, [R5,R2]     ; store R1 to *(R5+R2) (same ↙
    ↳ R5+i*4)
          BLT     loc_1CE         ; yes, i<20, run loop body ↙
    ↳ again

; second loop

          MOVS    R4, #0          ; i=0
loc_1DC
          LSLS    R0, R4, #2      ; R0=i<<2 (same as i*4)
          LDR     R2, [R5,R0]     ; load from *(R5+R0) (same as ↙
    ↳ R5+i*4)
          MOVS    R1, R4
          ADR     R0, aADD        ; "a[%d]=%d\n"
          BL      __2printf
          ADDS    R4, R4, #1      ; i=i+1
          CMP     R4, #20         ; i<20?
          BLT     loc_1DC         ; yes, i<20, run loop body ↙
    ↳ again
          MOVS    R0, #0          ; value to return
; deallocate place, allocated for 20 int variables + one more ↙
    ↳ variable
          ADD     SP, SP, #0x54
          POP     {R4,R5,PC}
```

Thumb code is very similar. Thumb mode has special instructions for bit shifting (like LSLS), which calculates value to be written into array and address of each element in array as well.

Compiler allocates slightly more space in local stack, however, last 4 bytes are not used.

16.2 Buffer overflow

So, array indexing is just *array[index]*. If you study generated code closely, you'll probably note missing index bounds checking, which could check index, *if it is less than 20*. What if index will be greater than 20? That's the one C/C++ feature it is often blamed for.

Here is a code successfully compiling and working:

```
#include <stdio.h>

int main()
{
        int a[20];
        int i;

        for (i=0; i<20; i++)
                a[i]=i*2;

        printf ("a[100]=%d\n", a[100]);

        return 0;
};
```

Compilation results (MSVC 2010):

```
_TEXT    SEGMENT
_i$ = -84                       ; size = 4
_a$ = -80                       ; size = 80
_main      PROC
    push   ebp
    mov    ebp, esp
    sub    esp, 84            ; 00000054H
    mov    DWORD PTR _i$[ebp], 0
    jmp    SHORT $LN3@main
$LN2@main:
    mov    eax, DWORD PTR _i$[ebp]
    add    eax, 1
    mov    DWORD PTR _i$[ebp], eax
$LN3@main:
    cmp    DWORD PTR _i$[ebp], 20 ; 00000014H
    jge    SHORT $LN1@main
```

```
    mov     ecx, DWORD PTR _i$[ebp]
    shl     ecx, 1
    mov     edx, DWORD PTR _i$[ebp]
    mov     DWORD PTR _a$[ebp+edx*4], ecx
    jmp     SHORT $LN2@main
$LN1@main:
    mov     eax, DWORD PTR _a$[ebp+400]
    push    eax
    push    OFFSET $SG2460
    call    _printf
    add     esp, 8
    xor     eax, eax
    mov     esp, ebp
    pop     ebp
    ret     0
_main       ENDP
```

I'm running it, and I got:

```
a[100]=760826203
```

It is just *something*, occasionally lying in the stack near to array, 400 bytes from its first element.

Indeed, how it could be done differently? Compiler may generate some additional code for checking index value to be always in array's bound (like in higher-level programming languages[3]) but this makes running code slower.

OK, we read some values from the stack *illegally* but what if we could write something to it?

Here is what we will write:

```
#include <stdio.h>

int main()
{
        int a[20];
        int i;

        for (i=0; i<30; i++)
                a[i]=i;

        return 0;
};
```

And what we've got:

[3] Java, Python, etc

```
_TEXT    SEGMENT
_i$ = -84                ; size = 4
_a$ = -80                ; size = 80
_main   PROC
 push   ebp
 mov    ebp, esp
 sub    esp, 84                    ; 00000054H
 mov    DWORD PTR _i$[ebp], 0
 jmp    SHORT $LN3@main
$LN2@main:
 mov    eax, DWORD PTR _i$[ebp]
 add    eax, 1
 mov    DWORD PTR _i$[ebp], eax
$LN3@main:
 cmp    DWORD PTR _i$[ebp], 30      ; 0000001eH
 jge    SHORT $LN1@main
 mov    ecx, DWORD PTR _i$[ebp]
 mov    edx, DWORD PTR _i$[ebp]      ; that instruction is ↙
    ↳ obviously redundant
 mov    DWORD PTR _a$[ebp+ecx*4], edx ; ECX could be used as ↙
    ↳ second operand here instead
 jmp    SHORT $LN2@main
$LN1@main:
 xor    eax, eax
 mov    esp, ebp
 pop    ebp
 ret    0
_main   ENDP
```

Run compiled program and its crashing. No wonder. Let's see, where exactly it is crashing.

I'm not using debugger anymore since I tried to run it each time, move mouse, etc, when I need just to spot a register's state at the specific point. That's why I wrote very minimalistic tool for myself, tracer, which is enough for my tasks.

I can also use it just to see, where debuggee is crashed. So let's see:

```
generic tracer 0.4 (WIN32), http://conus.info/gt

New process: C:\PRJ\...\1.exe, PID=7988
EXCEPTION_ACCESS_VIOLATION: 0x15 (<symbol (0x15) is in unknown ↙
    ↳ module>), ExceptionInformation[0]=8
EAX=0x00000000 EBX=0x7EFDE000 ECX=0x0000001D EDX=0x0000001D
ESI=0x00000000 EDI=0x00000000 EBP=0x00000014 ESP=0x0018FF48
EIP=0x00000015
```

```
FLAGS=PF ZF IF RF
PID=7988|Process exit, return code -1073740791
```

Now please keep your eyes on registers.

Exception occurred at address 0x15. It is not legal address for code —at least for win32 code! We trapped there somehow against our will. It is also interesting fact the EBP register contain 0x14, ECX and EDX —0x1D.

Let's study stack layout more.

After control flow was passed into `main()`, the value in the EBP register was saved on the stack. Then, 84 bytes was allocated for array and *i* variable. That's `(20+1)*sizeof(int` The ESP pointing now to the `_i` variable in the local stack and after execution of next `PUSH something`, *something* will be appeared next to `_i`.

That's stack layout while control is inside `main()`:

ESP	4 bytes for *i*
ESP+4	80 bytes for `a[20]` array
ESP+84	saved EBP value
ESP+88	returning address

Instruction `a[19]=something` writes last *int* in array bounds (in bounds so far!)

Instruction `a[20]=something` writes *something* to the place where value from the EBP is saved.

Please take a look at registers state at the crash moment. In our case, number 20 was written to 20th element. By the function ending, function epilogue restores original EBP value. (20 in decimal system is 0x14 in hexadecimal). Then, RET instruction was executed, which is effectively equivalent to `POP EIP` instruction.

RET instruction taking returning address from the stack (that is the address inside of CRT), which was called `main()`), and 21 was stored there (0x15 in hexadecimal). The CPU trapped at the address 0x15, but there is no executable code, so exception was raised.

Welcome! It is called *buffer overflow*[4].

Replace *int* array by string (*char* array), create a long string deliberately, and pass it to the program, to the function which is not checking string length and copies it to short buffer, and you'll able to point to a program an address to which it must jump. Not that simple in reality, but that is how it was emerged [5]

Let's try the same code in GCC 4.4.1. We got:

```
                public main
main            proc near

a               = dword ptr -54h
i               = dword ptr -4
```

[4] http://en.wikipedia.org/wiki/Stack_buffer_overflow

[5] Classic article about it: [22].

```
                push    ebp
                mov     ebp, esp
                sub     esp, 60h
                mov     [ebp+i], 0
                jmp     short loc_80483D1
loc_80483C3:
                mov     eax, [ebp+i]
                mov     edx, [ebp+i]
                mov     [ebp+eax*4+a], edx
                add     [ebp+i], 1
loc_80483D1:
                cmp     [ebp+i], 1Dh
                jle     short loc_80483C3
                mov     eax, 0
                leave
                retn
main            endp
```

Running this in Linux will produce: `Segmentation fault`.

If we run this in GDB debugger, we getting this:

```
(gdb) r
Starting program: /home/dennis/RE/1

Program received signal SIGSEGV, Segmentation fault.
0x00000016 in ?? ()
(gdb) info registers
eax            0x0      0
ecx            0xd2f96388       -755407992
edx            0x1d     29
ebx            0x26eff4 2551796
esp            0xbffff4b0       0xbffff4b0
ebp            0x15     0x15
esi            0x0      0
edi            0x0      0
eip            0x16     0x16
eflags         0x10202  [ IF RF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
```

```
(gdb)
```

Register values are slightly different then in win32 example since stack layout is slightly different too.

16.3 Buffer overflow protection methods

There are several methods to protect against it, regardless of C/C++ programmers' negligence. MSVC has options like[6]:

```
/RTCs Stack Frame runtime checking
/GZ Enable stack checks (/RTCs)
```

One of the methods is to write random value among local variables to stack at function prologue and to check it in function epilogue before function exiting. And if value is not the same, do not execute last instruction `RET`, but halt (or hang). Process will hang, but that is much better then remote attack to your host.

This random value is called "canary" sometimes, it is related to miner's canary[7], they were used by miners in these days, in order to detect poisonous gases quickly. Canaries are very sensetive to mine gases, they become very agitated in case of danger, or even dead.

If to compile our very simple array example (16.1) in MSVC with RTC1 and RTCs option, you will see call to `@_RTC_CheckStackVars@8` function at the function end, checking "canary" correctness.

Let's see how GCC handles this. Let's take `alloca()` (4.2.4) example:

```
#ifdef __GNUC__
#include <alloca.h> // GCC
#else
#include <malloc.h> // MSVC
#endif
#include <stdio.h>

void f()
{
    char *buf=(char*)alloca (600);
#ifdef __GNUC__
    snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // GCC
#else
    _snprintf (buf, 600, "hi! %d, %d, %d\n", 1, 2, 3); // MSVC
#endif
```

[6] Wikipedia: compiler-side buffer overflow protection methods: http://en.wikipedia.org/wiki/Buffer_overflow_protection

[7] http://en.wikipedia.org/wiki/Domestic_Canary#Miner.27s_canary

```
    puts (buf);
};
```

By default, without any additional options, GCC 4.7.3 will insert "canary" check into code:

Listing 16.3: GCC 4.7.3

```
.LC0:
        .string "hi! %d, %d, %d\n"
f:
        push    ebp
        mov     ebp, esp
        push    ebx
        sub     esp, 676
        lea     ebx, [esp+39]
        and     ebx, -16
        mov     DWORD PTR [esp+20], 3
        mov     DWORD PTR [esp+16], 2
        mov     DWORD PTR [esp+12], 1
        mov     DWORD PTR [esp+8], OFFSET FLAT:.LC0  ; "hi! %d, ↙
    ↳ %d, %d\n"
        mov     DWORD PTR [esp+4], 600
        mov     DWORD PTR [esp], ebx
        mov     eax, DWORD PTR gs:20                 ; canary
        mov     DWORD PTR [ebp-12], eax
        xor     eax, eax
        call    _snprintf
        mov     DWORD PTR [esp], ebx
        call    puts
        mov     eax, DWORD PTR [ebp-12]
        xor     eax, DWORD PTR gs:20                 ; canary
        jne     .L5
        mov     ebx, DWORD PTR [ebp-4]
        leave
        ret
.L5:
        call    __stack_chk_fail
```

Random value is located in `gs:20`. It is to be written on the stack and then, at the function end, value in the stack is compared with correct "canary" in `gs:20`. If values are not equal to each other, `__stack_chk_fail` function will be called and we will see something like that in console (Ubuntu 13.04 x86):

```
*** buffer overflow detected ***: ./2_1 terminated
```

```
======= Backtrace: =========
/lib/i386-linux-gnu/libc.so.6(__fortify_fail+0x63)[0xb7699bc3]
/lib/i386-linux-gnu/libc.so.6(+0x10593a)[0xb769893a]
/lib/i386-linux-gnu/libc.so.6(+0x105008)[0xb7698008]
/lib/i386-linux-gnu/libc.so.6(_IO_default_xsputn+0x8c)[0↙
    ↳ xb7606e5c]
/lib/i386-linux-gnu/libc.so.6(_IO_vfprintf+0x165)[0xb75d7a45]
/lib/i386-linux-gnu/libc.so.6(__vsprintf_chk+0xc9)[0xb76980d9]
/lib/i386-linux-gnu/libc.so.6(__sprintf_chk+0x2f)[0xb7697fef]
./2_1[0x8048404]
/lib/i386-linux-gnu/libc.so.6(__libc_start_main+0xf5)[0xb75ac935↙
    ↳ ]
======= Memory map: ========
08048000-08049000 r-xp 00000000 08:01 2097586    /home/dennis/2↙
    ↳ _1
08049000-0804a000 r--p 00000000 08:01 2097586    /home/dennis/2↙
    ↳ _1
0804a000-0804b000 rw-p 00001000 08:01 2097586    /home/dennis/2↙
    ↳ _1
094d1000-094f2000 rw-p 00000000 00:00 0          [heap]
b7560000-b757b000 r-xp 00000000 08:01 1048602    /lib/i386-linux↙
    ↳ -gnu/libgcc_s.so.1
b757b000-b757c000 r--p 0001a000 08:01 1048602    /lib/i386-linux↙
    ↳ -gnu/libgcc_s.so.1
b757c000-b757d000 rw-p 0001b000 08:01 1048602    /lib/i386-linux↙
    ↳ -gnu/libgcc_s.so.1
b7592000-b7593000 rw-p 00000000 00:00 0
b7593000-b7740000 r-xp 00000000 08:01 1050781    /lib/i386-linux↙
    ↳ -gnu/libc-2.17.so
b7740000-b7742000 r--p 001ad000 08:01 1050781    /lib/i386-linux↙
    ↳ -gnu/libc-2.17.so
b7742000-b7743000 rw-p 001af000 08:01 1050781    /lib/i386-linux↙
    ↳ -gnu/libc-2.17.so
b7743000-b7746000 rw-p 00000000 00:00 0
b775a000-b775d000 rw-p 00000000 00:00 0
b775d000-b775e000 r-xp 00000000 00:00 0          [vdso]
b775e000-b777e000 r-xp 00000000 08:01 1050794    /lib/i386-linux↙
    ↳ -gnu/ld-2.17.so
b777e000-b777f000 r--p 0001f000 08:01 1050794    /lib/i386-linux↙
    ↳ -gnu/ld-2.17.so
b777f000-b7780000 rw-p 00020000 08:01 1050794    /lib/i386-linux↙
    ↳ -gnu/ld-2.17.so
bff35000-bff56000 rw-p 00000000 00:00 0          [stack]
```

```
Aborted (core dumped)
```

gs—is so-called segment register, these registers were used widely in MS-DOS and DOS-extenders times. Today, its function is different. If to say briefly, the `gs` register in Linux is always pointing to the TLS (45) —various information specific to thread is stored there (by the way, in win32 environment, the `fs` register plays the same role, it pointing to TIB[8] [9]).

More information can be found in Linux source codes (at least in 3.11 version), in *arch/x86/include/asm/stackprotector.h* file this variable is described in comments.

16.3.1 Optimizing Xcode (LLVM) + thumb-2 mode

Let's back to our simple array example (16.1), again, now we can see how LLVM will check "canary" correctness:

```
_main

var_64          = -0x64
var_60          = -0x60
var_5C          = -0x5C
var_58          = -0x58
var_54          = -0x54
var_50          = -0x50
var_4C          = -0x4C
var_48          = -0x48
var_44          = -0x44
var_40          = -0x40
var_3C          = -0x3C
var_38          = -0x38
var_34          = -0x34
var_30          = -0x30
var_2C          = -0x2C
var_28          = -0x28
var_24          = -0x24
var_20          = -0x20
var_1C          = -0x1C
var_18          = -0x18
canary          = -0x14
var_10          = -0x10

    PUSH     {R4-R7,LR}
    ADD      R7, SP, #0xC
```

[8] Thread Information Block

[9] https://en.wikipedia.org/wiki/Win32_Thread_Information_Block

```
    STR.W    R8, [SP,#0xC+var_10]!
    SUB      SP, SP, #0x54
    MOVW     R0, #aObjc_methtype ; "objc_methtype"
    MOVS     R2, #0
    MOVT.W   R0, #0
    MOVS     R5, #0
    ADD      R0, PC
    LDR.W    R8, [R0]
    LDR.W    R0, [R8]
    STR      R0, [SP,#0x64+canary]
    MOVS     R0, #2
    STR      R2, [SP,#0x64+var_64]
    STR      R0, [SP,#0x64+var_60]
    MOVS     R0, #4
    STR      R0, [SP,#0x64+var_5C]
    MOVS     R0, #6
    STR      R0, [SP,#0x64+var_58]
    MOVS     R0, #8
    STR      R0, [SP,#0x64+var_54]
    MOVS     R0, #0xA
    STR      R0, [SP,#0x64+var_50]
    MOVS     R0, #0xC
    STR      R0, [SP,#0x64+var_4C]
    MOVS     R0, #0xE
    STR      R0, [SP,#0x64+var_48]
    MOVS     R0, #0x10
    STR      R0, [SP,#0x64+var_44]
    MOVS     R0, #0x12
    STR      R0, [SP,#0x64+var_40]
    MOVS     R0, #0x14
    STR      R0, [SP,#0x64+var_3C]
    MOVS     R0, #0x16
    STR      R0, [SP,#0x64+var_38]
    MOVS     R0, #0x18
    STR      R0, [SP,#0x64+var_34]
    MOVS     R0, #0x1A
    STR      R0, [SP,#0x64+var_30]
    MOVS     R0, #0x1C
    STR      R0, [SP,#0x64+var_2C]
    MOVS     R0, #0x1E
    STR      R0, [SP,#0x64+var_28]
    MOVS     R0, #0x20
    STR      R0, [SP,#0x64+var_24]
```

```
    MOVS     R0, #0x22
    STR      R0, [SP,#0x64+var_20]
    MOVS     R0, #0x24
    STR      R0, [SP,#0x64+var_1C]
    MOVS     R0, #0x26
    STR      R0, [SP,#0x64+var_18]
    MOV      R4, 0xFDA ; "a[%d]=%d\n"
    MOV      R0, SP
    ADDS     R6, R0, #4
    ADD      R4, PC
    B        loc_2F1C

; second loop begin

loc_2F14
    ADDS     R0, R5, #1
    LDR.W    R2, [R6,R5,LSL#2]
    MOV      R5, R0

loc_2F1C
    MOV      R0, R4
    MOV      R1, R5
    BLX      _printf
    CMP      R5, #0x13
    BNE      loc_2F14
    LDR.W    R0, [R8]
    LDR      R1, [SP,#0x64+canary]
    CMP      R0, R1
    ITTTT EQ            ; canary still correct?
    MOVEQ    R0, #0
    ADDEQ    SP, SP, #0x54
    LDREQ.W  R8, [SP+0x64+var_64],#4
    POPEQ    {R4-R7,PC}
    BLX      ___stack_chk_fail
```

First of all, as we see, LLVM made loop "unrolled" and all values are written into array one-by-one, already calculated since LLVM concluded it will be faster. By the way, ARM mode instructions may help to do this even faster, and finding this way could be your homework.

At the function end wee see "canaries" comparison —that laying in local stack and correct one, to which the R8 register pointing. If they are equal to each other, 4-instruction block is triggered by "ITTTT EQ", it is writing 0 into R0, function epilogue and exit. If "canaries" are not equal, block will not be triggered, and jump to ___stack_chk_fail function will be occurred, which, as I suppose, will halt execu-

tion.

16.4 One more word about arrays

Now we understand, why it is impossible to write something like that in C/C++ code [10]:

```
void f(int size)
{
    int a[size];
...
};
```

That's just because compiler must know exact array size to allocate space for it in local stack layout or in data segment (in case of global variable) on compiling stage.

If you need array of arbitrary size, allocate it by `malloc()`, then access allocated memory block as array of variables of type you need. Or use C99 standard feature [15, 6.7.5/2], but it will be looks like alloca() (4.2.4) internally.

16.5 Multidimensional arrays

Internally, multidimensional array is essentially the same thing as linear array.

Since computer memory in linear, it is one-dimensional array. But this one-dimensional array can be easily represented as multidimensional for convenience.

For example, that is how $a[3][4]$ array elements will be placed in one-dimensional array of 12 cells:

[0][0]
[0][1]
[0][2]
[0][3]
[1][0]
[1][1]
[1][2]
[1][3]
[2][0]
[2][1]
[2][2]
[2][3]

[10] However, it is possible in C99 standard [15, 6.7.5/2]: GCC is actually do this by allocating array dynamically on the stack (like alloca() (4.2.4))

That is how two-dimensional array with one-dimensional (memory) array index numbers can be represented:

0	1	2	3
4	5	6	7
8	9	10	11

So, in order to address elements we need, first multiply first index by 4 (matrix width) and then add second index. That's called *row-major order*, and this method of arrays and matrices representation is used in at least in C/C++, Python. *row-major order* term in plain English language means: "first, write elements of first row, then second row ... and finally elements of last row".

Another method of representation called *column-major order* (array indices used in reverse order) and it is used at least in FORTRAN, MATLAB, R. *column-major order* term in plain English language means: "first, write elements of first column, then second column ... and finally elements of last column".

Same thing about multidimensional arrays.

Let's see:

Listing 16.4: simple example

```
#include <stdio.h>

int a[10][20][30];

void insert(int x, int y, int z, int value)
{
        a[x][y][z]=value;
};
```

16.5.1 x86

We got (MSVC 2010):

Listing 16.5: MSVC 2010

```
_DATA    SEGMENT
COMM     _a:DWORD:01770H
_DATA    ENDS
PUBLIC   _insert
_TEXT    SEGMENT
_x$ = 8                 ; size = 4
_y$ = 12                ; size = 4
_z$ = 16                ; size = 4
_value$ = 20            ; size = 4
```

```
_insert    PROC
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _x$[ebp]
    imul   eax, 2400                    ; eax=600*4*x
    mov    ecx, DWORD PTR _y$[ebp]
    imul   ecx, 120                     ; ecx=30*4*y
    lea    edx, DWORD PTR _a[eax+ecx]   ; edx=a + 600*4*x + ↙
    ↘ 30*4*y
    mov    eax, DWORD PTR _z$[ebp]
    mov    ecx, DWORD PTR _value$[ebp]
    mov    DWORD PTR [edx+eax*4], ecx   ; *(edx+z*4)=value
    pop    ebp
    ret    0
_insert    ENDP
_TEXT      ENDS
```

Nothing special. For index calculation, three input arguments are multiplying by formula $address = 600 \cdot 4 \cdot x + 30 \cdot 4 \cdot y + 4z$ to represent array as multidimensional. Do not forget the *int* type is 32-bit (4 bytes), so all coefficients must be multiplied by 4.

Listing 16.6: GCC 4.4.1

```
                public insert
insert          proc near

x               = dword ptr  8
y               = dword ptr  0Ch
z               = dword ptr  10h
value           = dword ptr  14h

                push    ebp
                mov     ebp, esp
                push    ebx
                mov     ebx, [ebp+x]
                mov     eax, [ebp+y]
                mov     ecx, [ebp+z]
                lea     edx, [eax+eax]              ; edx=y*2
                mov     eax, edx                    ; eax=y*2
                shl     eax, 4                      ; eax=(y*2)↙
    ↘ <<4 = y*2*16 = y*32
                sub     eax, edx                    ; eax=y*32 -↙
    ↘  y*2=y*30
                imul    edx, ebx, 600               ; edx=x*600
```

```
                add     eax, edx                        ; eax=eax+↙
    ↳ edx=y*30 + x*600
                lea     edx, [eax+ecx]                  ; edx=y*30 +↙
    ↳  x*600 + z
                mov     eax, [ebp+value]
                mov     dword ptr ds:a[edx*4], eax      ; *(a+edx*4)↙
    ↳ =value
                pop     ebx
                pop     ebp
                retn
insert          endp
```

GCC compiler does it differently. For one of operations calculating $(30y)$, GCC produced a code without multiplication instruction. This is how it done: $(y+y) \ll 4-(y+y) = (2y) \ll 4-2y = 2 \cdot 16 \cdot y - 2y = 32y - 2y = 30y$. Thus, for $30y$ calculation, only one addition operation used, one bitwise shift operation and one subtraction operation. That works faster.

16.5.2 ARM + Non-optimizing Xcode (LLVM) + thumb mode

Listing 16.7: Non-optimizing Xcode (LLVM) + thumb mode

```
_insert

value   = -0x10
z       = -0xC
y       = -8
x       = -4

; allocate place in local stack for 4 values of int type
SUB     SP, SP, #0x10
MOV     R9, 0xFC2 ; a
ADD     R9, PC
LDR.W   R9, [R9]
STR     R0, [SP,#0x10+x]
STR     R1, [SP,#0x10+y]
STR     R2, [SP,#0x10+z]
STR     R3, [SP,#0x10+value]
LDR     R0, [SP,#0x10+value]
LDR     R1, [SP,#0x10+z]
LDR     R2, [SP,#0x10+y]
LDR     R3, [SP,#0x10+x]
MOV     R12, 2400
MUL.W   R3, R3, R12
```

```
ADD     R3, R9
MOV     R9, 120
MUL.W   R2, R2, R9
ADD     R2, R3
LSLS    R1, R1, #2 ; R1=R1<<2
ADD     R1, R2
STR     R0, [R1]   ; R1 - address of array element
; deallocate place in local stack, allocated for 4 values of int↙
    ↳ type
ADD     SP, SP, #0x10
BX      LR
```

Non-optimizing LLVM saves all variables in local stack, however, it is redundant. Address of array element is calculated by formula we already figured out.

16.5.3 ARM + Optimizing Xcode (LLVM) + thumb mode

Listing 16.8: Optimizing Xcode (LLVM) + thumb mode

```
_insert
MOVW    R9, #0x10FC
MOV.W   R12, #2400
MOVT.W  R9, #0
RSB.W   R1, R1, R1,LSL#4  ; R1 - y. R1=y<<4 - y = y*16 - y = y↙
    ↳ *15
ADD     R9, PC            ; R9 = pointer to a array
LDR.W   R9, [R9]
MLA.W   R0, R0, R12, R9   ; R0 - x, R12 - 2400, R9 - pointer to ↙
    ↳ a. R0=x*2400 + ptr to a
ADD.W   R0, R0, R1,LSL#3  ; R0 = R0+R1<<3 = R0+R1*8 = x*2400 + ↙
    ↳ ptr to a + y*15*8 =
                          ; ptr to a + y*30*4 + x*600*4
STR.W   R3, [R0,R2,LSL#2] ; R2 - z, R3 - value. address=R0+z*4 =
                          ; ptr to a + y*30*4 + x*600*4 + z*4
BX      LR
```

Here is used tricks for replacing multiplication by shift, addition and subtraction we already considered.

Here we also see new instruction for us: RSB (*Reverse Subtract*). It works just as SUB, but swapping operands with each other. Why? SUB, RSB, are those instructions, to the second operand of which shift coefficient may be applied: (LSL#4). But this coefficient may be applied only to second operand. That's fine for commutative operations like addition or multiplication, operands may be swapped there without result affecting. But subtraction is non-commutative operation, so, for these cases, RSB

exist.

‘‘LDR.W R9, [R9]’’ works like LEA (B.6.2) in x86, but here it does nothing, it is redundant. Apparently, compiler not optimized it.

Chapter 17

Bit fields

A lot of functions defining input flags in arguments using bit fields. Of course, it could be substituted by *bool*-typed variables set, but it is not frugally.

17.1 Specific bit checking

17.1.1 x86

Win32 API example:

```
        HANDLE fh;

        fh=CreateFile ("file", GENERIC_WRITE | GENERIC_READ, ↙
    ↳ FILE_SHARE_READ, NULL, OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL,↙
    ↳  NULL);
```

We got (MSVC 2010):

Listing 17.1: MSVC 2010

```
        push    0
        push    128                                     ; ↙
    ↳ 00000080H
        push    4
        push    0
        push    1
        push    -1073741824                             ; ↙
    ↳ c0000000H
        push    OFFSET $SG78813
        call    DWORD PTR __imp__CreateFileA@28
        mov     DWORD PTR _fh$[ebp], eax
```

Let's take a look into WinNT.h:

Listing 17.2: WinNT.h

```
#define GENERIC_READ                     (0x80000000L)
#define GENERIC_WRITE                    (0x40000000L)
#define GENERIC_EXECUTE                  (0x20000000L)
#define GENERIC_ALL                      (0x10000000L)
```

Everything is clear, `GENERIC_READ | GENERIC_WRITE = 0x80000000 | 0x40000000 = 0xC0000000`, and that is value is used as the second argument for `CreateFile()`[1] function.

How `CreateFile()` will check flags?

Let's take a look into KERNEL32.DLL in Windows XP SP3 x86 and we'll find this fragment of code in the function `CreateFileW`:

Listing 17.3: KERNEL32.DLL (Windows XP SP3 x86)

```
.text:7C83D429                 test    byte ptr [ebp+↙
    ↳ dwDesiredAccess+3], 40h
.text:7C83D42D                 mov     [ebp+var_8], 1
.text:7C83D434                 jz      short loc_7C83D417
.text:7C83D436                 jmp     loc_7C810817
```

Here we see `TEST` instruction, it takes, however, not the whole second argument, but only most significant byte (`ebp+dwDesiredAccess+3`) and checks it for 0x40 flag (meaning `GENERIC_WRITE` flag here)

`TEST` is merely the same instruction as `AND`, but without result saving (recall the fact `CMP` instruction is merely the same as `SUB`, but without result saving (6.6.1)).

This fragment of code logic is as follows:

```
if ((dwDesiredAccess&0x40000000) == 0) goto loc_7C83D417
```

If `AND` instruction leaving this bit, `ZF` flag is to be cleared and `JZ` conditional jump will not be triggered. Conditional jump will be triggered only if `0x40000000` bit is absent in the `dwDesiredAccess` variable —then `AND` result will be 0, `ZF` flag will be set and conditional jump is to be triggered.

Let's try GCC 4.4.1 and Linux:

```
#include <stdio.h>
#include <fcntl.h>

void main()
{
        int handle;
```

[1] MSDN: CreateFile function

```
        handle=open ("file", O_RDWR | O_CREAT);
};
```

We got:

Listing 17.4: GCC 4.4.1

```
                public main
main            proc near

var_20          = dword ptr -20h
var_1C          = dword ptr -1Ch
var_4           = dword ptr -4

                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF0h
                sub     esp, 20h
                mov     [esp+20h+var_1C], 42h
                mov     [esp+20h+var_20], offset aFile ; "file"
                call    _open
                mov     [esp+20h+var_4], eax
                leave
                retn
main            endp
```

Let's take a look into `open()` function in the `libc.so.6` library, but there is only syscall calling:

Listing 17.5: open() (libc.so.6)

```
.text:000BE69B                  mov     edx, [esp+4+mode] ; mode
.text:000BE69F                  mov     ecx, [esp+4+flags] ; ↙
    ↳ flags
.text:000BE6A3                  mov     ebx, [esp+4+filename] ; ↙
    ↳ filename
.text:000BE6A7                  mov     eax, 5
.text:000BE6AC                  int     80h             ; LINUX -↙
    ↳  sys_open
```

So, `open()` bit fields apparently checked somewhere in Linux kernel.

Of course, it is easily to download both Glibc and Linux kernel source code, but we are interesting to understand the matter without it.

So, as of Linux 2.6, when `sys_open` syscall is called, control eventually passed into `do_sys_open` kernel function. From there —to the `do_filp_open()` function (this function located in kernel source tree in the file `fs/namei.c`).

N.B. Aside from common passing arguments via stack, there is also a method of passing some of them via registers. This is also called fastcall (44.3). This works faster since CPU not needed to access a stack in memory to read argument values. GCC has option *regparm*[2], and it is possible to set a number of arguments which might be passed via registers.

Linux 2.6 kernel compiled with `-mregparm=3` option [3] [4].

What it means to us, the first 3 arguments will be passed via `EAX`, `EDX` and `ECX` registers, the rest ones via stack. Of course, if arguments number is less than 3, only part of registers are to be used.

So, let's download Linux Kernel 2.6.31, compile it in Ubuntu: `make vmlinux`, open it in IDA, find the `do_filp_open()` function. At the beginning, we will see (comments are mine):

Listing 17.6: do_filp_open() (linux kernel 2.6.31)

```
do_filp_open    proc near
...
                push    ebp
                mov     ebp, esp
                push    edi
                push    esi
                push    ebx
                mov     ebx, ecx
                add     ebx, 1
                sub     esp, 98h
                mov     esi, [ebp+arg_4] ; acc_mode (5th arg)
                test    bl, 3
                mov     [ebp+var_80], eax ; dfd (1th arg)
                mov     [ebp+var_7C], edx ; pathname (2th arg)
                mov     [ebp+var_78], ecx ; open_flag (3th arg)
                jnz     short loc_C01EF684
                mov     ebx, ecx        ; ebx <- open_flag
```

GCC saves first 3 arguments values in local stack. Otherwise, if compiler would not touch these registers, it would be too tight environment for compiler's register allocator.

Let's find this fragment of code:

Listing 17.7: do_filp_open() (linux kernel 2.6.31)

```
loc_C01EF6B4:                           ; CODE XREF: ↙
    ↳ do_filp_open+4F
                test    bl, 40h         ; O_CREAT
```

[2] http://ohse.de/uwe/articles/gcc-attributes.html#func-regparm

[3] http://kernelnewbies.org/Linux_2_6_20#head-042c62f290834eb1fe0a1942bbf5bb9a4accbc8f

[4] See also `arch\x86\include\asm\calling.h` file in kernel tree

```
                jnz     loc_C01EF810
                mov     edi, ebx
                shr     edi, 11h
                xor     edi, 1
                and     edi, 1
                test    ebx, 10000h
                jz      short loc_C01EF6D3
                or      edi, 2
```

0x40 —is what O_CREAT macro equals to. open_flag checked for 0x40 bit presence, and if this bit is 1, next JNZ instruction is triggered.

17.1.2 ARM

O_CREAT bit is checked differently in Linux kernel 3.8.0.

Listing 17.8: linux kernel 3.8.0

```
struct file *do_filp_open(int dfd, struct filename *pathname,
                const struct open_flags *op)
{
...
        filp = path_openat(dfd, pathname, &nd, op, flags | ↙
    ↳ LOOKUP_RCU);
...
}

static struct file *path_openat(int dfd, struct filename *↙
    ↳ pathname,
                struct nameidata *nd, const struct open_flags *↙
    ↳ op, int flags)
{
...
        error = do_last(nd, &path, file, op, &opened, pathname);
...
}



static int do_last(struct nameidata *nd, struct path *path,
                   struct file *file, const struct open_flags *↙
    ↳ op,
                   int *opened, struct filename *name)
{
...
        if (!(open_flag & O_CREAT)) {
```

```
    ...
                error = lookup_fast(nd, path, &inode);
    ...
        } else {
    ...
                error = complete_walk(nd);
        }
...
}
```

Here is how kernel compiled for ARM mode looks like in IDA:

Listing 17.9: do_last() (vmlinux)

```
...
.text:C0169EA8                  MOV             R9, R3  ; R3 - (4↙
    ↳ th argument) open_flag
...
.text:C0169ED4                  LDR             R6, [R9] ; R6 - ↙
    ↳ open_flag
...
.text:C0169F68                  TST             R6, #0x40 ; ↙
    ↳ jumptable C0169F00 default case
.text:C0169F6C                  BNE             loc_C016A128
.text:C0169F70                  LDR             R2, [R4,#0x10]
.text:C0169F74                  ADD             R12, R4, #8
.text:C0169F78                  LDR             R3, [R4,#0xC]
.text:C0169F7C                  MOV             R0, R4
.text:C0169F80                  STR             R12, [R11,#var_50↙
    ↳ ]
.text:C0169F84                  LDRB            R3, [R2,R3]
.text:C0169F88                  MOV             R2, R8
.text:C0169F8C                  CMP             R3, #0
.text:C0169F90                  ORRNE           R1, R1, #3
.text:C0169F94                  STRNE           R1, [R4,#0x24]
.text:C0169F98                  ANDS            R3, R6, #0x200000
.text:C0169F9C                  MOV             R1, R12
.text:C0169FA0                  LDRNE           R3, [R4,#0x24]
.text:C0169FA4                  ANDNE           R3, R3, #1
.text:C0169FA8                  EORNE           R3, R3, #1
.text:C0169FAC                  STR             R3, [R11,#var_54]
.text:C0169FB0                  SUB             R3, R11, #-var_38
.text:C0169FB4                  BL              lookup_fast
...
```

```
.text:C016A128 loc_C016A128                                    ; CODE ↙
    ↳ XREF: do_last.isra.14+DC
.text:C016A128                  MOV             R0, R4
.text:C016A12C                  BL              complete_walk
...
```

TST is analogical to a TEST instruction in x86.

We can "spot" visually this code fragment by the fact the `lookup_fast()` will be executed in one case and the `complete_walk()` in another case. This is corresponding to the `do_last()` function source code.

O_CREAT macro is equals to 0x40 here too.

17.2 Specific bit setting/clearing

For example:

```
#define IS_SET(flag, bit)       ((flag) & (bit))
#define SET_BIT(var, bit)       ((var) |= (bit))
#define REMOVE_BIT(var, bit)    ((var) &= ~(bit))

int f(int a)
{
    int rt=a;

    SET_BIT (rt, 0x4000);
    REMOVE_BIT (rt, 0x200);

    return rt;
};
```

17.2.1 x86

We got (MSVC 2010):

Listing 17.10: MSVC 2010

```
_rt$ = -4          ; size = 4
_a$ = 8            ; size = 4
_f  PROC
    push   ebp
    mov    ebp, esp
    push   ecx
    mov    eax, DWORD PTR _a$[ebp]
    mov    DWORD PTR _rt$[ebp], eax
```

```
    mov     ecx, DWORD PTR _rt$[ebp]
    or      ecx, 16384                ; 00004000H
    mov     DWORD PTR _rt$[ebp], ecx
    mov     edx, DWORD PTR _rt$[ebp]
    and     edx, -513                 ; fffffdffH
    mov     DWORD PTR _rt$[ebp], edx
    mov     eax, DWORD PTR _rt$[ebp]
    mov     esp, ebp
    pop     ebp
    ret     0
_f  ENDP
```

OR instruction adds one more bit to value while ignoring the rest ones.

AND resetting one bit. It can be said, AND just copies all bits except one. Indeed, in the second AND operand only those bits are set, which are needed to be saved, except one bit we would not like to copy (which is 0 in bitmask). It is easier way to memorize the logic.

If we compile it in MSVC with optimization turned on (/Ox), the code will be even shorter:

Listing 17.11: Optimizing MSVC

```
_a$ = 8                       ; size = 4
_f    PROC
    mov     eax, DWORD PTR _a$[esp-4]
    and     eax, -513         ; fffffdffH
    or      eax, 16384        ; 00004000H
    ret     0
_f    ENDP
```

Let's try GCC 4.4.1 without optimization:

Listing 17.12: Non-optimizing GCC

```
                public f
f               proc near

var_4           = dword ptr -4
arg_0           = dword ptr  8

                push    ebp
                mov     ebp, esp
                sub     esp, 10h
                mov     eax, [ebp+arg_0]
                mov     [ebp+var_4], eax
                or      [ebp+var_4], 4000h
```

```
                and     [ebp+var_4], 0FFFFFDFFh
                mov     eax, [ebp+var_4]
                leave
                retn
f               endp
```

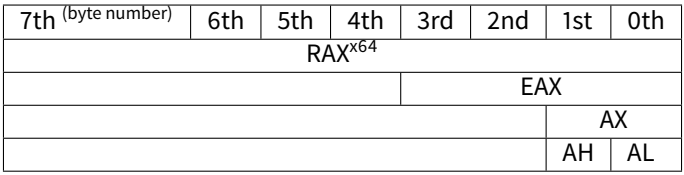
There is a redundant code present, however, it is shorter then MSVC version without optimization.

Now let's try GCC with optimization turned on `-O3`:

Listing 17.13: Optimizing GCC

```
                public f
f               proc near

arg_0           = dword ptr  8

                push    ebp
                mov     ebp, esp
                mov     eax, [ebp+arg_0]
                pop     ebp
                or      ah, 40h
                and     ah, 0FDh
                retn
f               endp
```

That's shorter. It is worth noting the compiler works with the `EAX` register part via the `AH` register —that is the `EAX` register part from 8th to 15th bits inclusive.



N.B. 16-bit CPU 8086 accumulator was named `AX` and consisted of two 8-bit halves — `AL` (lower byte) and `AH` (higher byte). In 80386 almost all registers were extended to 32-bit, accumulator was named `EAX`, but for the sake of compatibility, its *older parts* may be still accessed as `AX`/`AH`/`AL` registers.

Since all x86 CPUs are 16-bit 8086 CPU successors, these *older* 16-bit opcodes are shorter than newer 32-bit opcodes. That's why "`or ah, 40h`" instruction occupying only 3 bytes. It would be more logical way to emit here "`or eax, 04000h`" but that is 5 bytes, or even 6 (in case if register in first operand is not `EAX`).

It would be even shorter if to turn on `-O3` optimization flag and also set `regparm=3`.

Listing 17.14: Optimizing GCC

```
                public f
f               proc near
                push    ebp
                or      ah, 40h
                mov     ebp, esp
                and     ah, 0FDh
                pop     ebp
                retn
f               endp
```

Indeed —first argument is already loaded into EAX, so it is possible to work with it in-place. It is worth noting that both function prologue (‘‘push ebp / mov ebp,esp’’) and epilogue (‘‘pop ebp’’) can easily be omitted here, but GCC probably is not good enough for such code size optimizations. However, such short functions are better to be *inlined functions* (27).

17.2.2 ARM + Optimizing Keil + ARM mode

Listing 17.15: Optimizing Keil + ARM mode

```
02 0C C0 E3          BIC     R0, R0, #0x200
01 09 80 E3          ORR     R0, R0, #0x4000
1E FF 2F E1          BX      LR
```

BIC is “logical and”, analogical to AND in x86. ORR is “logical or”, analogical to OR in x86.

So far, so easy.

17.2.3 ARM + Optimizing Keil + thumb mode

Listing 17.16: Optimizing Keil + thumb mode

```
01 21 89 03          MOVS    R1, 0x4000
08 43                ORRS    R0, R1
49 11                ASRS    R1, R1, #5   ; generate 0x200 and ↙
    ↳ place to R1
88 43                BICS    R0, R1
70 47                BX      LR
```

Apparently, Keil concludes the code in thumb mode, making 0x200 from 0x4000, will be more compact than code, writing 0x200 to arbitrary register.

So that is why, with the help of ASRS (arithmetic shift right), this value is calculating as $0x4000 \gg 5$.

17.2.4 ARM + Optimizing Xcode (LLVM) + ARM mode

Listing 17.17: Optimizing Xcode (LLVM) + ARM mode

```
42 0C C0 E3          BIC             R0, R0, #0x4200
01 09 80 E3          ORR             R0, R0, #0x4000
1E FF 2F E1          BX              LR
```

The code was generated by LLVM, in source code form, in fact, could be looks like:

```
    REMOVE_BIT (rt, 0x4200);
    SET_BIT (rt, 0x4000);
```

And it does exactly the same we need. But why 0x4200? Perhaps, that is the LLVM optimizer's artifact [5]. Probably, compiler's optimizer error, but generated code works correct anyway.

More about compiler's anomalies, read here (64).

For thumb mode, Optimizing Xcode (LLVM) generates likewise code.

17.3 Shifts

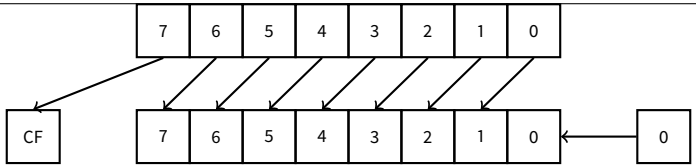
Bit shifts in C/C++ are implemented via $\ll$ and $\gg$ operators.

Here is a simple example of function, calculating number of 1 bits in input variable:

```
#define IS_SET(flag, bit)       ((flag) & (bit))

int f(unsigned int a)
{
    int i;
    int rt=0;

    for (i=0; i<32; i++)
        if (IS_SET (a, 1<<i))
            rt++;

    return rt;
};
```

In this loop, iteration count value *i* counting from 0 to 31, $1 \ll i$ statement will be counting from 1 to 0x80000000. Describing this operation in natural language, we would say *shift* 1 *by n bits left*. In other words, $1 \ll i$ statement will consequently produce all possible bit positions in 32-bit number. By the way, freed bit at right is always cleared. IS_SET macro is checking bit presence in the a.

[5] It was LLVM build 2410.2.00 bundled with Apple Xcode 4.6.3



The IS_SET macro is in fact logical and operation (*AND*) and it returns 0 if specific bit is absent there, or bit mask, if the bit is present. *if()* operator triggered in C/C++ if expression in it is not a zero, it might be even 123456, that is why it always working correctly.

17.3.1 x86

Let's compile (MSVC 2010):

Listing 17.18: MSVC 2010

```
_rt$ = -8               ; size = 4
_i$ = -4                ; size = 4
_a$ = 8                 ; size = 4
_f  PROC
    push   ebp
    mov    ebp, esp
    sub    esp, 8
    mov    DWORD PTR _rt$[ebp], 0
    mov    DWORD PTR _i$[ebp], 0
    jmp    SHORT $LN4@f
$LN3@f:
    mov    eax, DWORD PTR _i$[ebp]   ; increment of 1
    add    eax, 1
    mov    DWORD PTR _i$[ebp], eax
$LN4@f:
    cmp    DWORD PTR _i$[ebp], 32    ; 00000020H
    jge    SHORT $LN2@f              ; loop finished?
    mov    edx, 1
    mov    ecx, DWORD PTR _i$[ebp]
    shl    edx, cl                   ; EDX=EDX<<CL
    and    edx, DWORD PTR _a$[ebp]
    je     SHORT $LN1@f              ; result of AND instruction↙
    ↳  was 0?
                                     ; then skip next ↙
    ↳ instructions
    mov    eax, DWORD PTR _rt$[ebp]  ; no, not zero
    add    eax, 1                    ; increment rt
    mov    DWORD PTR _rt$[ebp], eax
```

```
$LN1@f:
    jmp     SHORT $LN3@f
$LN2@f:
    mov     eax, DWORD PTR _rt$[ebp]
    mov     esp, ebp
    pop     ebp
    ret     0
_f    ENDP
```

That's how SHL (*SHift Left*) working.
Let's compile it in GCC 4.4.1:

Listing 17.19: GCC 4.4.1

```
                public f
f               proc near

rt              = dword ptr -0Ch
i               = dword ptr -8
arg_0           = dword ptr  8

                push    ebp
                mov     ebp, esp
                push    ebx
                sub     esp, 10h
                mov     [ebp+rt], 0
                mov     [ebp+i], 0
                jmp     short loc_80483EF
loc_80483D0:
                mov     eax, [ebp+i]
                mov     edx, 1
                mov     ebx, edx
                mov     ecx, eax
                shl     ebx, cl
                mov     eax, ebx
                and     eax, [ebp+arg_0]
                test    eax, eax
                jz      short loc_80483EB
                add     [ebp+rt], 1
loc_80483EB:
                add     [ebp+i], 1
loc_80483EF:
                cmp     [ebp+i], 1Fh
                jle     short loc_80483D0
                mov     eax, [ebp+rt]
```

```
                add     esp, 10h
                pop     ebx
                pop     ebp
                retn
f               endp
```

Shift instructions are often used in division and multiplications by power of two numbers (1, 2, 4, 8, etc).

For example:

```
unsigned int f(unsigned int a)
{
        return a/4;
};
```

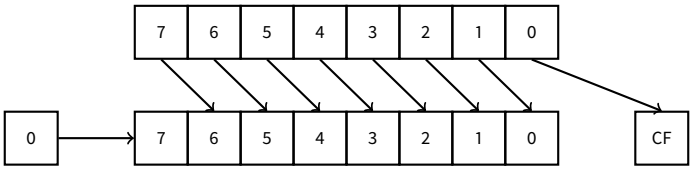
We got (MSVC 2010):

Listing 17.20: MSVC 2010

```
_a$ = 8                                                 ; size =↙
    ↳ 4
_f      PROC
        mov     eax, DWORD PTR _a$[esp-4]
        shr     eax, 2
        ret     0
_f      ENDP
```

SHR (*SHift Right*) instruction in this example is shifting a number by 2 bits right. Two freed bits at left (e.g., two most significant bits) are set to zero. Two least significant bits are dropped. In fact, these two dropped bits —division operation remainder.

SHR instruction works just like as SHL but in other direction.



It can be easily understood if to imagine decimal numeral system and number 23. 23 can be easily divided by 10 just by dropping last digit (3 —is division remainder). 2 is leaving after operation as a quotient.

The same story about multiplication. Multiplication by 4 is just shifting the number to the left by 2 bits, while inserting 2 zero bits at right (as the last two bits). It is just like to multiply 3 by 100 —we need just to add two zeroes at the right.

17.3.2 ARM + Optimizing Xcode (LLVM) + ARM mode

Listing 17.21: Optimizing Xcode (LLVM) + ARM mode

```
                MOV             R1, R0
                MOV             R0, #0
                MOV             R2, #1
                MOV             R3, R0
loc_2E54
                TST             R1, R2,LSL R3 ; set flags ↙
    ↳ according to R1 & (R2<<R3)
                ADD             R3, R3, #1    ; R3++
                ADDNE           R0, R0, #1    ; if ZF flag is ↙
    ↳ cleared by TST, R0++
                CMP             R3, #32
                BNE             loc_2E54
                BX              LR
```

TST is the same things as TEST in x86.

As I mentioned before (14.2.1), there are no separate shifting instructions in ARM mode. However, there are modifiers LSL (*Logical Shift Left*), LSR (*Logical Shift Right*), ASR (*Arithmetic Shift Right*), ROR (*Rotate Right*) and RRX (*Rotate Right with Extend*) , which may be added to such instructions as MOV, TST, CMP, ADD, SUB, RSB[6].

These modificators are defines, how to shift second operand and by how many bits.

Thus ‘‘TST R1, R2,LSL R3’’ instruction works here as $R1 \wedge (R2 \ll R3)$.

17.3.3 ARM + Optimizing Xcode (LLVM) + thumb-2 mode

Almost the same, but here are two LSL.W/TST instructions are used instead of single TST, because, in thumb mode, it is not possible to define LSL modifier right in TST.

```
                MOV             R1, R0
                MOVS            R0, #0
                MOV.W           R9, #1
                MOVS            R3, #0
loc_2F7A
                LSL.W           R2, R9, R3
                TST             R2, R1
                ADD.W           R3, R3, #1
                IT NE
                ADDNE           R0, #1
                CMP             R3, #32
```

[6]These instructions are also called “data processing instructions”

```
                BNE             loc_2F7A
                BX              LR
```

17.4 CRC32 calculation example

This is very popular table-based CRC32 hash calculation technique[7].

```
/* By Bob Jenkins, (c) 2006, Public Domain */

#include <stdio.h>
#include <stddef.h>
#include <string.h>

typedef  unsigned long  ub4;
typedef  unsigned char  ub1;

static const ub4 crctab[256] = {
  0x00000000, 0x77073096, 0xee0e612c, 0x990951ba, 0x076dc419, 0↙
    ↳ x706af48f,
  0xe963a535, 0x9e6495a3, 0x0edb8832, 0x79dcb8a4, 0xe0d5e91e, 0↙
    ↳ x97d2d988,
  0x09b64c2b, 0x7eb17cbd, 0xe7b82d07, 0x90bf1d91, 0x1db71064, 0↙
    ↳ x6ab020f2,
  0xf3b97148, 0x84be41de, 0x1adad47d, 0x6ddde4eb, 0xf4d4b551, 0↙
    ↳ x83d385c7,
  0x136c9856, 0x646ba8c0, 0xfd62f97a, 0x8a65c9ec, 0x14015c4f, 0↙
    ↳ x63066cd9,
  0xfa0f3d63, 0x8d080df5, 0x3b6e20c8, 0x4c69105e, 0xd56041e4, 0↙
    ↳ xa2677172,
  0x3c03e4d1, 0x4b04d447, 0xd20d85fd, 0xa50ab56b, 0x35b5a8fa, 0↙
    ↳ x42b2986c,
  0xdbbbc9d6, 0xacbcf940, 0x32d86ce3, 0x45df5c75, 0xdcd60dcf, 0↙
    ↳ xabd13d59,
  0x26d930ac, 0x51de003a, 0xc8d75180, 0xbfd06116, 0x21b4f4b5, 0↙
    ↳ x56b3c423,
  0xcfba9599, 0xb8bda50f, 0x2802b89e, 0x5f058808, 0xc60cd9b2, 0↙
    ↳ xb10be924,
  0x2f6f7c87, 0x58684c11, 0xc1611dab, 0xb6662d3d, 0x76dc4190, 0↙
    ↳ x01db7106,
  0x98d220bc, 0xefd5102a, 0x71b18589, 0x06b6b51f, 0x9fbfe4a5, 0↙
    ↳ xe8b8d433,
```

[7] Source code was taken here: http://burtleburtle.net/bob/c/crc.c

```
0x7807c9a2, 0x0f00f934, 0x9609a88e, 0xe10e9818, 0x7f6a0dbb, 0↙
  ↳ x086d3d2d,
0x91646c97, 0xe6635c01, 0x6b6b51f4, 0x1c6c6162, 0x856530d8, 0↙
  ↳ xf262004e,
0x6c0695ed, 0x1b01a57b, 0x8208f4c1, 0xf50fc457, 0x65b0d9c6, 0↙
  ↳ x12b7e950,
0x8bbeb8ea, 0xfcb9887c, 0x62dd1ddf, 0x15da2d49, 0x8cd37cf3, 0↙
  ↳ xfbd44c65,
0x4db26158, 0x3ab551ce, 0xa3bc0074, 0xd4bb30e2, 0x4adfa541, 0↙
  ↳ x3dd895d7,
0xa4d1c46d, 0xd3d6f4fb, 0x4369e96a, 0x346ed9fc, 0xad678846, 0↙
  ↳ xda60b8d0,
0x44042d73, 0x33031de5, 0xaa0a4c5f, 0xdd0d7cc9, 0x5005713c, 0↙
  ↳ x270241aa,
0xbe0b1010, 0xc90c2086, 0x5768b525, 0x206f85b3, 0xb966d409, 0↙
  ↳ xce61e49f,
0x5edef90e, 0x29d9c998, 0xb0d09822, 0xc7d7a8b4, 0x59b33d17, 0↙
  ↳ x2eb40d81,
0xb7bd5c3b, 0xc0ba6cad, 0xedb88320, 0x9abfb3b6, 0x03b6e20c, 0↙
  ↳ x74b1d29a,
0xead54739, 0x9dd277af, 0x04db2615, 0x73dc1683, 0xe3630b12, 0↙
  ↳ x94643b84,
0x0d6d6a3e, 0x7a6a5aa8, 0xe40ecf0b, 0x9309ff9d, 0x0a00ae27, 0↙
  ↳ x7d079eb1,
0xf00f9344, 0x8708a3d2, 0x1e01f268, 0x6906c2fe, 0xf762575d, 0↙
  ↳ x806567cb,
0x196c3671, 0x6e6b06e7, 0xfed41b76, 0x89d32be0, 0x10da7a5a, 0↙
  ↳ x67dd4acc,
0xf9b9df6f, 0x8ebeeff9, 0x17b7be43, 0x60b08ed5, 0xd6d6a3e8, 0↙
  ↳ xa1d1937e,
0x38d8c2c4, 0x4fdff252, 0xd1bb67f1, 0xa6bc5767, 0x3fb506dd, 0↙
  ↳ x48b2364b,
0xd80d2bda, 0xaf0a1b4c, 0x36034af6, 0x41047a60, 0xdf60efc3, 0↙
  ↳ xa867df55,
0x316e8eef, 0x4669be79, 0xcb61b38c, 0xbc66831a, 0x256fd2a0, 0↙
  ↳ x5268e236,
0xcc0c7795, 0xbb0b4703, 0x220216b9, 0x5505262f, 0xc5ba3bbe, 0↙
  ↳ xb2bd0b28,
0x2bb45a92, 0x5cb36a04, 0xc2d7ffa7, 0xb5d0cf31, 0x2cd99e8b, 0↙
  ↳ x5bdeae1d,
0x9b64c2b0, 0xec63f226, 0x756aa39c, 0x026d930a, 0x9c0906a9, 0↙
  ↳ xeb0e363f,
0x72076785, 0x05005713, 0x95bf4a82, 0xe2b87a14, 0x7bb12bae, 0↙
```

```
    ↪ x0cb61b38,
  0x92d28e9b, 0xe5d5be0d, 0x7cdcefb7, 0x0bdbdf21, 0x86d3d2d4, 0↙
    ↪ xf1d4e242,
  0x68ddb3f8, 0x1fda836e, 0x81be16cd, 0xf6b9265b, 0x6fb077e1, 0↙
    ↪ x18b74777,
  0x88085ae6, 0xff0f6a70, 0x66063bca, 0x11010b5c, 0x8f659eff, 0↙
    ↪ xf862ae69,
  0x616bffd3, 0x166ccf45, 0xa00ae278, 0xd70dd2ee, 0x4e048354, 0↙
    ↪ x3903b3c2,
  0xa7672661, 0xd06016f7, 0x4969474d, 0x3e6e77db, 0xaed16a4a, 0↙
    ↪ xd9d65adc,
  0x40df0b66, 0x37d83bf0, 0xa9bcae53, 0xdebb9ec5, 0x47b2cf7f, 0↙
    ↪ x30b5ffe9,
  0xbdbdf21c, 0xcabac28a, 0x53b39330, 0x24b4a3a6, 0xbad03605, 0↙
    ↪ xcdd70693,
  0x54de5729, 0x23d967bf, 0xb3667a2e, 0xc4614ab8, 0x5d681b02, 0↙
    ↪ x2a6f2b94,
  0xb40bbe37, 0xc30c8ea1, 0x5a05df1b, 0x2d02ef8d,
};

/* how to derive the values in crctab[] from polynomial 0↙
    ↪ xedb88320 */
void build_table()
{
  ub4 i, j;
  for (i=0; i<256; ++i) {
    j = i;
    j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
    j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
    j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
    j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
    j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
    j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
    j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
    j = (j>>1) ^ ((j&1) ? 0xedb88320 : 0);
    printf("0x%.8lx, ", j);
    if (i%6 == 5) printf("\n");
  }
}


/* the hash function */
ub4 crc(const void *key, ub4 len, ub4 hash)
```

```
{
  ub4  i;
  const ub1 *k = key;
  for (hash=len, i=0; i<len; ++i)
    hash = (hash >> 8) ^ crctab[(hash & 0xff) ^ k[i]];
  return hash;
}

/* To use, try "gcc -O crc.c -o crc; crc < crc.c" */
int main()
{
  char s[1000];
  while (gets(s)) printf("%.8lx\n", crc(s, strlen(s), 0));
  return 0;
}
```

We are interesting in the `crc()` function only. By the way, pay attention to two loop initializers in the `for()` statement: `hash=len, i=0`. C/C++ standard allows this, of course. Emitted code will contain two operations in loop initialization part instead of usual one.

Let's compile it in MSVC with optimization (`/Ox`). For the sake of brevity, only `crc()` function is listed here, with my comments.

```
_key$ = 8                ; size = 4
_len$ = 12               ; size = 4
_hash$ = 16              ; size = 4
_crc    PROC
    mov    edx, DWORD PTR _len$[esp-4]
    xor    ecx, ecx ; i will be stored in ECX
    mov    eax, edx
    test   edx, edx
    jbe    SHORT $LN1@crc
    push   ebx
    push   esi
    mov    esi, DWORD PTR _key$[esp+4] ; ESI = key
    push   edi
$LL3@crc:

; work with bytes using only 32-bit registers. byte from address↙
    ↳  key+i we store into EDI

    movzx  edi, BYTE PTR [ecx+esi]
    mov    ebx, eax ; EBX = (hash = len)
    and    ebx, 255 ; EBX = hash & 0xff
```

```
; XOR EDI, EBX (EDI=EDI^EBX) - this operation uses all 32 bits ↙
    ↳ of each register
; but other bits (8-31) are cleared all time, so it's OK
; these are cleared because, as for EDI, it was done by MOVZX ↙
    ↳ instruction above
; high bits of EBX was cleared by AND EBX, 255 instruction above↙
    ↳  (255 = 0xff)

    xor    edi, ebx

; EAX=EAX>>8; bits 24-31 taken "from nowhere" will be cleared
    shr    eax, 8

; EAX=EAX^crctab[EDI*4] - choose EDI-th element from crctab[] ↙
    ↳ table
    xor    eax, DWORD PTR _crctab[edi*4]
    inc    ecx             ; i++
    cmp    ecx, edx        ; i<len ?
    jb     SHORT $LL3@crc ; yes
    pop    edi
    pop    esi
    pop    ebx
$LN1@crc:
    ret    0
_crc    ENDP
```

Let's try the same in GCC 4.4.1 with -O3 option:

```
                public crc
crc             proc near

key             = dword ptr  8
hash            = dword ptr  0Ch

                push    ebp
                xor     edx, edx
                mov     ebp, esp
                push    esi
                mov     esi, [ebp+key]
                push    ebx
                mov     ebx, [ebp+hash]
                test    ebx, ebx
                mov     eax, ebx
```

```
                jz      short loc_80484D3
                nop                     ; padding
                lea     esi, [esi+0]    ; padding; ESI doesn't ↙
    ↳ changing here

loc_80484B8:
                mov     ecx, eax        ; save previous state of↙
    ↳  hash to ECX
                xor     al, [esi+edx]   ; AL=*(key+i)
                add     edx, 1          ; i++
                shr     ecx, 8          ; ECX=hash>>8
                movzx   eax, al         ; EAX=*(key+i)
                mov     eax, dword ptr ds:crctab[eax*4] ; EAX=↙
    ↳ crctab[EAX]
                xor     eax, ecx        ; hash=EAX^ECX
                cmp     ebx, edx
                ja      short loc_80484B8

loc_80484D3:
                pop     ebx
                pop     esi
                pop     ebp
                retn
crc             endp
\
```

GCC aligned loop start on a 8-byte boundary by adding `NOP` and `lea esi, [esi+0]` (that is the *idle operation* too). Read more about it in npad section (61).

Chapter 18

Structures

It can be defined the C/C++ structure, with some assumptions, just a set of variables, always stored in memory together, not necessary of the same type [1].

18.1 SYSTEMTIME example

Let's take SYSTEMTIME[2] win32 structure describing time.

That's how it is defined:

Listing 18.1: WinBase.h

```
typedef struct _SYSTEMTIME {
  WORD wYear;
  WORD wMonth;
  WORD wDayOfWeek;
  WORD wDay;
  WORD wHour;
  WORD wMinute;
  WORD wSecond;
  WORD wMilliseconds;
} SYSTEMTIME, *PSYSTEMTIME;
```

Let's write a C function to get current time:

```
#include <windows.h>
#include <stdio.h>

void main()
```

[1] AKA "heterogeneous container"

[2] MSDN: SYSTEMTIME structure

```
{
    SYSTEMTIME t;
    GetSystemTime (&t);

    printf ("%04d-%02d-%02d %02d:%02d:%02d\n",
        t.wYear, t.wMonth, t.wDay,
        t.wHour, t.wMinute, t.wSecond);

    return;
};
```

We got (MSVC 2010):

Listing 18.2: MSVC 2010

```
_t$ = -16 ; size = 16
_main      PROC
    push   ebp
    mov    ebp, esp
    sub    esp, 16
    lea    eax, DWORD PTR _t$[ebp]
    push   eax
    call   DWORD PTR __imp__GetSystemTime@4
    movzx  ecx, WORD PTR _t$[ebp+12] ; wSecond
    push   ecx
    movzx  edx, WORD PTR _t$[ebp+10] ; wMinute
    push   edx
    movzx  eax, WORD PTR _t$[ebp+8] ; wHour
    push   eax
    movzx  ecx, WORD PTR _t$[ebp+6] ; wDay
    push   ecx
    movzx  edx, WORD PTR _t$[ebp+2] ; wMonth
    push   edx
    movzx  eax, WORD PTR _t$[ebp] ; wYear
    push   eax
    push   OFFSET $SG78811 ; '%04d-%02d-%02d %02d:%02d:%02d', 0↙
    ↳ aH, 00H
    call   _printf
    add    esp, 28
    xor    eax, eax
    mov    esp, ebp
    pop    ebp
    ret    0
_main      ENDP
```

16 bytes are allocated for this structure in local stack —that is exactly `sizeof(WORD)*8` (there are 8 WORD variables in the structure).

Pay attention to the fact the structure beginning with `wYear` field. It can be said, an pointer to SYSTEMTIME structure is passed to the `GetSystemTime()`[3], but it is also can be said, pointer to the `wYear` field is passed, and that is the same! `GetSystemTime()` writes current year to the WORD pointer pointing to, then shifts 2 bytes ahead, then writes current month, etc, etc.

The fact the structure fields are just variables located side-by-side, I can demonstrate by the following technique. Keeping in ming SYSTEMTIME structure description, I can rewrite this simple example like this:

```
#include <windows.h>
#include <stdio.h>

void main()
{
    WORD array[8];
    GetSystemTime (array);

    printf ("%04d-%02d-%02d %02d:%02d:%02d\n",
        array[0] /* wYear */, array[1] /* wMonth */, array[3] /*↙
    ↪  wDay */,
        array[4] /* wHour */, array[5] /* wMinute */, array[6] ↙
    ↪ /* wSecond */);

    return;
};
```

Compiler will grumble for a little:

```
systemtime2.c(7) : warning C4133: 'function' : incompatible ↙
    ↪ types - from 'WORD [8]' to 'LPSYSTEMTIME'
```

But nevertheless, it will produce this code:

Listing 18.3: MSVC 2010

```
$SG78573 DB     '%04d-%02d-%02d %02d:%02d:%02d', 0aH, 00H

_array$ = -16   ; size = 16
_main   PROC
        push    ebp
        mov     ebp, esp
        sub     esp, 16
```

[3] MSDN: SYSTEMTIME structure

```
        lea     eax, DWORD PTR _array$[ebp]
        push    eax
        call    DWORD PTR __imp__GetSystemTime@4
        movzx   ecx, WORD PTR _array$[ebp+12] ; wSecond
        push    ecx
        movzx   edx, WORD PTR _array$[ebp+10] ; wMinute
        push    edx
        movzx   eax, WORD PTR _array$[ebp+8] ; wHoure
        push    eax
        movzx   ecx, WORD PTR _array$[ebp+6] ; wDay
        push    ecx
        movzx   edx, WORD PTR _array$[ebp+2] ; wMonth
        push    edx
        movzx   eax, WORD PTR _array$[ebp] ; wYear
        push    eax
        push    OFFSET $SG78573
        call    _printf
        add     esp, 28
        xor     eax, eax
        mov     esp, ebp
        pop     ebp
        ret     0
_main   ENDP
```

And it works just as the same!

It is very interesting fact the result in assembly form cannot be distinguished from the result of previous compilation. So by looking at this code, one cannot say for sure, was there structure declared, or just pack of variables.

Nevertheless, no one will do it in sane state of mind. Since it is not convenient. Also structure fields may be changed by developers, swapped, etc.

18.2 Let's allocate space for structure using malloc()

However, sometimes it is simpler to place structures not in local stack, but in heap:

```
#include <windows.h>
#include <stdio.h>

void main()
{
    SYSTEMTIME *t;

    t=(SYSTEMTIME *)malloc (sizeof (SYSTEMTIME));
```

```
    GetSystemTime (t);

    printf ("%04d-%02d-%02d %02d:%02d:%02d\n",
        t->wYear, t->wMonth, t->wDay,
        t->wHour, t->wMinute, t->wSecond);

    free (t);

    return;
};
```

Let's compile it now with optimization (/Ox) so to easily see what we need.

Listing 18.4: Optimizing MSVC

```
_main      PROC
    push   esi
    push   16
    call   _malloc
    add    esp, 4
    mov    esi, eax
    push   esi
    call   DWORD PTR __imp__GetSystemTime@4
    movzx  eax, WORD PTR [esi+12] ; wSecond
    movzx  ecx, WORD PTR [esi+10] ; wMinute
    movzx  edx, WORD PTR [esi+8] ; wHour
    push   eax
    movzx  eax, WORD PTR [esi+6] ; wDay
    push   ecx
    movzx  ecx, WORD PTR [esi+2] ; wMonth
    push   edx
    movzx  edx, WORD PTR [esi] ; wYear
    push   eax
    push   ecx
    push   edx
    push   OFFSET $SG78833
    call   _printf
    push   esi
    call   _free
    add    esp, 32
    xor    eax, eax
    pop    esi
    ret    0
_main      ENDP
```

So, `sizeof(SYSTEMTIME) = 16`, that is exact number of bytes to be allocated by `malloc()`. It returns the pointer to freshly allocated memory block in the `EAX` register, which is then moved into the `ESI` register. `GetSystemTime()` win32 function undertake to save value in the `ESI`, and that is why it is not saved here and continue to be used after `GetSystemTime()` call.

New instruction —`MOVZX` (*Move with Zero eXtent*). It may be used almost in those cases as `MOVSX` (13.1.1), but, it clears other bits to 0. That's because `printf()` requires 32-bit *int*, but we got WORD in structure —that is 16-bit unsigned type. That's why by copying value from WORD into *int*, bits from 16 to 31 must also be cleared, because there will be random noise otherwise, leaved there from previous operations on registers.

In this example, I can represent structure as array of WORD-s:

```
#include <windows.h>
#include <stdio.h>

void main()
{
    WORD *t;

    t=(WORD *)malloc (16);

    GetSystemTime (t);

    printf ("%04d-%02d-%02d %02d:%02d:%02d\n",
        t[0] /* wYear */, t[1] /* wMonth */, t[3] /* wDay */,
        t[4] /* wHour */, t[5] /* wMinute */, t[6] /* wSecond ↙
    ↘ */);

    free (t);

    return;
};
```

We got:

Listing 18.5: Optimizing MSVC

```
$SG78594 DB      '%04d-%02d-%02d %02d:%02d:%02d', 0aH, 00H

_main   PROC
        push    esi
        push    16
        call    _malloc
        add     esp, 4
```

```
        mov     esi, eax
        push    esi
        call    DWORD PTR __imp__GetSystemTime@4
        movzx   eax, WORD PTR [esi+12]
        movzx   ecx, WORD PTR [esi+10]
        movzx   edx, WORD PTR [esi+8]
        push    eax
        movzx   eax, WORD PTR [esi+6]
        push    ecx
        movzx   ecx, WORD PTR [esi+2]
        push    edx
        movzx   edx, WORD PTR [esi]
        push    eax
        push    ecx
        push    edx
        push    OFFSET $SG78594
        call    _printf
        push    esi
        call    _free
        add     esp, 32
        xor     eax, eax
        pop     esi
        ret     0
_main   ENDP
```

Again, we got the code cannot be distinguished from the previous. And again I should note, one should not do this in practice.

18.3 struct tm

18.3.1 Linux

As of Linux, let's take `tm` structure from `time.h` for example:

```
#include <stdio.h>
#include <time.h>

void main()
{
    struct tm t;
    time_t unix_time;

    unix_time=time(NULL);
```

```
    localtime_r (&unix_time, &t);

    printf ("Year: %d\n", t.tm_year+1900);
    printf ("Month: %d\n", t.tm_mon);
    printf ("Day: %d\n", t.tm_mday);
    printf ("Hour: %d\n", t.tm_hour);
    printf ("Minutes: %d\n", t.tm_min);
    printf ("Seconds: %d\n", t.tm_sec);
};
```

Let's compile it in GCC 4.4.1:

Listing 18.6: GCC 4.4.1

```
main proc near
     push    ebp
     mov     ebp, esp
     and     esp, 0FFFFFFF0h
     sub     esp, 40h
     mov     dword ptr [esp], 0 ; first argument for time()
     call    time
     mov     [esp+3Ch], eax
     lea     eax, [esp+3Ch]  ; take pointer to what time() ↙
    ↳ returned
     lea     edx, [esp+10h]  ; at ESP+10h struct tm will begin
     mov     [esp+4], edx    ; pass pointer to the structure ↙
    ↳ begin
     mov     [esp], eax      ; pass pointer to result of time()
     call    localtime_r
     mov     eax, [esp+24h]  ; tm_year
     lea     edx, [eax+76Ch] ; edx=eax+1900
     mov     eax, offset format ; "Year: %d\n"
     mov     [esp+4], edx
     mov     [esp], eax
     call    printf
     mov     edx, [esp+20h]      ; tm_mon
     mov     eax, offset aMonthD ; "Month: %d\n"
     mov     [esp+4], edx
     mov     [esp], eax
     call    printf
     mov     edx, [esp+1Ch]     ; tm_mday
     mov     eax, offset aDayD  ; "Day: %d\n"
     mov     [esp+4], edx
     mov     [esp], eax
     call    printf
```

```
    mov     edx, [esp+18h]      ; tm_hour
    mov     eax, offset aHourD ; "Hour: %d\n"
    mov     [esp+4], edx
    mov     [esp], eax
    call    printf
    mov     edx, [esp+14h]         ; tm_min
    mov     eax, offset aMinutesD ; "Minutes: %d\n"
    mov     [esp+4], edx
    mov     [esp], eax
    call    printf
    mov     edx, [esp+10h]
    mov     eax, offset aSecondsD ; "Seconds: %d\n"
    mov     [esp+4], edx           ; tm_sec
    mov     [esp], eax
    call    printf
    leave
    retn
main endp
```

Somehow, IDA did not created local variables names in local stack. But since we already experienced reverse engineers :-) we may do it without this information in this simple example.

Please also pay attention to the `lea edx, [eax+76Ch]` —this instruction just adding `0x76C` to value in the `EAX`, but not modifies any flags. See also relevant section about `LEA` (B.6.2).

In order to illustrate the structure is just variables laying side-by-side in one place, let's rework example, while looking at the file *time.h*:

Listing 18.7: time.h

```
struct tm
{
  int   tm_sec;
  int   tm_min;
  int   tm_hour;
  int   tm_mday;
  int   tm_mon;
  int   tm_year;
  int   tm_wday;
  int   tm_yday;
  int   tm_isdst;
};
```

```
#include <stdio.h>
#include <time.h>
```

```
void main()
{
    int tm_sec, tm_min, tm_hour, tm_mday, tm_mon, tm_year, ↙
    ↘ tm_wday, tm_yday, tm_isdst;
    time_t unix_time;

    unix_time=time(NULL);

    localtime_r (&unix_time, &tm_sec);

    printf ("Year: %d\n", tm_year+1900);
    printf ("Month: %d\n", tm_mon);
    printf ("Day: %d\n", tm_mday);
    printf ("Hour: %d\n", tm_hour);
    printf ("Minutes: %d\n", tm_min);
    printf ("Seconds: %d\n", tm_sec);
};
```

N.B. The pointer to the exactly `tm_sec` field is passed into `localtime_r`, i.e., to the first "structure" element.

Compiler will warn us:

Listing 18.8: GCC 4.7.3

```
GCC_tm2.c: In function 'main':
GCC_tm2.c:11:5: warning: passing argument 2 of 'localtime_r' ↙
    ↘ from incompatible pointer type [enabled by default]
In file included from GCC_tm2.c:2:0:
/usr/include/time.h:59:12: note: expected 'struct tm *' but ↙
    ↘ argument is of type 'int *'
```

But nevertheless, will generate this:

Listing 18.9: GCC 4.7.3

```
main      proc near

var_30    = dword ptr -30h
var_2C    = dword ptr -2Ch
unix_time = dword ptr -1Ch
tm_sec    = dword ptr -18h
tm_min    = dword ptr -14h
tm_hour   = dword ptr -10h
tm_mday   = dword ptr -0Ch
tm_mon    = dword ptr -8
```

```
tm_year   = dword ptr -4

          push    ebp
          mov     ebp, esp
          and     esp, 0FFFFFFF0h
          sub     esp, 30h
          call    __main
          mov     [esp+30h+var_30], 0 ; arg 0
          call    time
          mov     [esp+30h+unix_time], eax
          lea     eax, [esp+30h+tm_sec]
          mov     [esp+30h+var_2C], eax
          lea     eax, [esp+30h+unix_time]
          mov     [esp+30h+var_30], eax
          call    localtime_r
          mov     eax, [esp+30h+tm_year]
          add     eax, 1900
          mov     [esp+30h+var_2C], eax
          mov     [esp+30h+var_30], offset aYearD ; "Year: %d\n"
          call    printf
          mov     eax, [esp+30h+tm_mon]
          mov     [esp+30h+var_2C], eax
          mov     [esp+30h+var_30], offset aMonthD ; "Month: %d\↙
    ↳ n"
          call    printf
          mov     eax, [esp+30h+tm_mday]
          mov     [esp+30h+var_2C], eax
          mov     [esp+30h+var_30], offset aDayD ; "Day: %d\n"
          call    printf
          mov     eax, [esp+30h+tm_hour]
          mov     [esp+30h+var_2C], eax
          mov     [esp+30h+var_30], offset aHourD ; "Hour: %d\n"
          call    printf
          mov     eax, [esp+30h+tm_min]
          mov     [esp+30h+var_2C], eax
          mov     [esp+30h+var_30], offset aMinutesD ; "Minutes:↙
    ↳  %d\n"
          call    printf
          mov     eax, [esp+30h+tm_sec]
          mov     [esp+30h+var_2C], eax
          mov     [esp+30h+var_30], offset aSecondsD ; "Seconds:↙
    ↳  %d\n"
          call    printf
```

```
        leave
        retn
main    endp
```

This code is identical to what we saw previously and it is not possible to say, was it structure in original source code or just pack of variables.

And this works. However, it is not recommended to do this in practice. Usually, compiler allocated variables in local stack in the same order as they were declared in function. Nevertheless, there is no any guarantee.

By the way, some other compiler may warn the `tm_year`, `tm_mon`, `tm_mday`, `tm_hour`, `tm_min` variables, but not `tm_sec` are used without being initialized. Indeed, compiler do not know these will be filled when calling to `localtime_r()`.

I chose exactly this example for illustration, since all structure fields has *int* type, and `SYSTEMTIME` structure fields —16-bit `WORD`, and if to declare them as a local variables, they will be aligned on a 32-bit border, and nothing will work (because `GetSystemTime()` will fill them incorrectly). Read more about it in next section: "Fields packing in structure".

So, structure is just variables pack laying on one place, side-by-side. I could say the structure is a syntactic sugar, directing compiler to hold them in one place. However, I'm not programming languages expert, so, most likely, I'm wrong with this term. By the way, there were a times, in very early C versions (before 1972), in which there were no structures at all [29].

18.3.2 ARM + Optimizing Keil + thumb mode

Same example:

Listing 18.10: Optimizing Keil + thumb mode

```
var_38 = -0x38
var_34 = -0x34
var_30 = -0x30
var_2C = -0x2C
var_28 = -0x28
var_24 = -0x24
timer  = -0xC

        PUSH    {LR}
        MOVS    R0, #0          ; timer
        SUB     SP, SP, #0x34
        BL      time
        STR     R0, [SP,#0x38+timer]
        MOV     R1, SP          ; tp
        ADD     R0, SP, #0x38+timer ; timer
        BL      localtime_r
```

```
        LDR     R1, =0x76C
        LDR     R0, [SP,#0x38+var_24]
        ADDS    R1, R0, R1
        ADR     R0, aYearD      ; "Year: %d\n"
        BL      __2printf
        LDR     R1, [SP,#0x38+var_28]
        ADR     R0, aMonthD     ; "Month: %d\n"
        BL      __2printf
        LDR     R1, [SP,#0x38+var_2C]
        ADR     R0, aDayD       ; "Day: %d\n"
        BL      __2printf
        LDR     R1, [SP,#0x38+var_30]
        ADR     R0, aHourD      ; "Hour: %d\n"
        BL      __2printf
        LDR     R1, [SP,#0x38+var_34]
        ADR     R0, aMinutesD   ; "Minutes: %d\n"
        BL      __2printf
        LDR     R1, [SP,#0x38+var_38]
        ADR     R0, aSecondsD   ; "Seconds: %d\n"
        BL      __2printf
        ADD     SP, SP, #0x34
        POP     {PC}
```

18.3.3 ARM + Optimizing Xcode (LLVM) + thumb-2 mode

IDA "get to know" `tm` structure (because IDA "knows" argument types of library functions like `localtime_r()`), so it shows here structure elements accesses and also names are assigned to them.

Listing 18.11: Optimizing Xcode (LLVM) + thumb-2 mode

```
var_38 = -0x38
var_34 = -0x34

        PUSH {R7,LR}
        MOV  R7, SP
        SUB  SP, SP, #0x30
        MOVS R0, #0  ; time_t *
        BLX  _time
        ADD  R1, SP, #0x38+var_34 ; struct tm *
        STR  R0, [SP,#0x38+var_38]
        MOV  R0, SP  ; time_t *
        BLX  _localtime_r
        LDR  R1, [SP,#0x38+var_34.tm_year]
```

```
        MOV  R0, 0xF44 ; "Year: %d\n"
        ADD  R0, PC  ; char *
        ADDW R1, R1, #0x76C
        BLX  _printf
        LDR  R1, [SP,#0x38+var_34.tm_mon]
        MOV  R0, 0xF3A ; "Month: %d\n"
        ADD  R0, PC  ; char *
        BLX  _printf
        LDR  R1, [SP,#0x38+var_34.tm_mday]
        MOV  R0, 0xF35 ; "Day: %d\n"
        ADD  R0, PC  ; char *
        BLX  _printf
        LDR  R1, [SP,#0x38+var_34.tm_hour]
        MOV  R0, 0xF2E ; "Hour: %d\n"
        ADD  R0, PC  ; char *
        BLX  _printf
        LDR  R1, [SP,#0x38+var_34.tm_min]
        MOV  R0, 0xF28 ; "Minutes: %d\n"
        ADD  R0, PC  ; char *
        BLX  _printf
        LDR  R1, [SP,#0x38+var_34]
        MOV  R0, 0xF25 ; "Seconds: %d\n"
        ADD  R0, PC  ; char *
        BLX  _printf
        ADD  SP, SP, #0x30
        POP  {R7,PC}

...

00000000 tm         struc ; (sizeof=0x2C, standard type)
00000000 tm_sec     DCD ?
00000004 tm_min     DCD ?
00000008 tm_hour    DCD ?
0000000C tm_mday    DCD ?
00000010 tm_mon     DCD ?
00000014 tm_year    DCD ?
00000018 tm_wday    DCD ?
0000001C tm_yday    DCD ?
00000020 tm_isdst   DCD ?
00000024 tm_gmtoff  DCD ?
00000028 tm_zone    DCD ? ; offset
0000002C tm         ends
```

18.4 Fields packing in structure

One important thing is fields packing in structures[4].

Let's take a simple example:

```
#include <stdio.h>

struct s
{
    char a;
    int b;
    char c;
    int d;
};

void f(struct s s)
{
    printf ("a=%d; b=%d; c=%d; d=%d\n", s.a, s.b, s.c, s.d);
};
```

As we see, we have two *char* fields (each is exactly one byte) and two more —*int* (each - 4 bytes).

18.4.1 x86

That's all compiling into:

```
_s$ = 8 ; size = 16
?f@@YAXUs@@@Z PROC ; f
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _s$[ebp+12]
    push   eax
    movsx  ecx, BYTE PTR _s$[ebp+8]
    push   ecx
    mov    edx, DWORD PTR _s$[ebp+4]
    push   edx
    movsx  eax, BYTE PTR _s$[ebp]
    push   eax
    push   OFFSET $SG3842
    call   _printf
    add    esp, 20
    pop    ebp
```

[4] See also: Wikipedia: Data structure alignment

```
    ret     0
?f@@YAXUs@@@Z ENDP ; f
_TEXT    ENDS
```

As we can see, each field's address is aligned on a 4-bytes border. That's why each *char* occupies 4 bytes here (like *int*). Why? Thus it is easier for CPU to access memory at aligned addresses and to cache data from it.

However, it is not very economical in size sense.

Let's try to compile it with option (/Zp1) (*/Zp[n] pack structures on n-byte boundary*).

Listing 18.12: MSVC /Zp1

```
_TEXT    SEGMENT
_s$ = 8 ; size = 10
?f@@YAXUs@@@Z PROC    ; f
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _s$[ebp+6]
    push   eax
    movsx  ecx, BYTE PTR _s$[ebp+5]
    push   ecx
    mov    edx, DWORD PTR _s$[ebp+1]
    push   edx
    movsx  eax, BYTE PTR _s$[ebp]
    push   eax
    push   OFFSET $SG3842
    call   _printf
    add    esp, 20
    pop    ebp
    ret    0
?f@@YAXUs@@@Z ENDP    ; f
```

Now the structure takes only 10 bytes and each *char* value takes 1 byte. What it give to us? Size economy. And as drawback —CPU will access these fields without maximal performance it can.

As it can be easily guessed, if the structure is used in many source and object files, all these must be compiled with the same convention about structures packing.

Aside from MSVC /Zp option which set how to align each structure field, here is also `#pragma pack` compiler option, it can be defined right in source code. It is available in both MSVC[5] and GCC[6].

Let's back to the SYSTEMTIME structure consisting in 16-bit fields. How our compiler know to pack them on 1-byte alignment boundary?

[5] MSDN: Working with Packing Structures

[6] Structure-Packing Pragmas

WinNT.h file has this:

Listing 18.13: WinNT.h

```
#include "pshpack1.h"
```

And this:

Listing 18.14: WinNT.h

```
#include "pshpack4.h"                    // 4 byte packing is the↙
    ↳ default
```

The file PshPack1.h looks like:

Listing 18.15: PshPack1.h

```
#if ! (defined(lint) || defined(RC_INVOKED))
#if ( _MSC_VER >= 800 && !defined(_M_I86)) || defined(↙
    ↳ _PUSHPOP_SUPPORTED)
#pragma warning(disable:4103)
#if !(defined( MIDL_PASS )) || defined( __midl )
#pragma pack(push,1)
#else
#pragma pack(1)
#endif
#else
#pragma pack(1)
#endif
#endif /* ! (defined(lint) || defined(RC_INVOKED)) */
```

That's how compiler will pack structures defined after `#pragma pack`.

18.4.2 ARM + Optimizing Keil + thumb mode

Listing 18.16: Optimizing Keil + thumb mode

```
.text:0000003E             exit ; CODE XREF: f+16
.text:0000003E 05 B0               ADD     SP, SP, #0x14
.text:00000040 00 BD               POP     {PC}



.text:00000280             f
.text:00000280
.text:00000280             var_18 = -0x18
.text:00000280             a      = -0x14
.text:00000280             b      = -0x10
```

```
.text:00000280                c       = -0xC
.text:00000280                d       = -8
.text:00000280
.text:00000280 0F B5                  PUSH    {R0-R3,LR}
.text:00000282 81 B0                  SUB     SP, SP, #4
.text:00000284 04 98                  LDR     R0, [SP,#16]    ; d
.text:00000286 02 9A                  LDR     R2, [SP,#8]     ; b
.text:00000288 00 90                  STR     R0, [SP]
.text:0000028A 68 46                  MOV     R0, SP
.text:0000028C 03 7B                  LDRB    R3, [R0,#12]    ; c
.text:0000028E 01 79                  LDRB    R1, [R0,#4]     ; a
.text:00000290 59 A0                  ADR     R0, aADBDCDDD   ; "a=%↙
    ↳ d; b=%d; c=%d; d=%d\n"
.text:00000292 05 F0 AD FF            BL      __2printf
.text:00000296 D2 E6                  B       exit
```

As we may recall, here a structure passed instead of pointer to structure, and since first 4 function arguments in ARM are passed via registers, so then structure fields are passed via R0-R3.

LDRB loads one byte from memory and extending it to 32-bit, taking into account its sign. This is akin to MOVSX (13.1.1) instruction in x86. Here it is used for loading fields a and c from structure.

One more thing we spot easily, instead of function epilogue, here is jump to another function's epilogue! Indeed, that was quite different function, not related in any way to our function, however, it has exactly the same epilogue (probably because, it hold 5 local variables too ($5 * 4 = 0x14$)). Also it is located nearly (take a look on addresses). Indeed, there is no difference, which epilogue to execute, if it works just as we need. Apparently, Keil decides to reuse a part of another function by a reason of economy. Epilogue takes 4 bytes while jump —only 2.

18.4.3 ARM + Optimizing Xcode (LLVM) + thumb-2 mode

Listing 18.17: Optimizing Xcode (LLVM) + thumb-2 mode

```
var_C = -0xC

      PUSH    {R7,LR}
      MOV     R7, SP
      SUB     SP, SP, #4
      MOV     R9, R1 ; b
      MOV     R1, R0 ; a
      MOVW    R0, #0xF10 ; "a=%d; b=%d; c=%d; d=%d\n"
      SXTB    R1, R1  ; prepare a
      MOVT.W R0, #0
```

```
        STR     R3, [SP,#0xC+var_C] ; place d to stack for printf()
        ADD     R0, PC  ; format-string
        SXTB    R3, R2  ; prepare c
        MOV     R2, R9  ; b
        BLX     _printf
        ADD     SP, SP, #4
        POP     {R7,PC}
```

SXTB (*Signed Extend Byte*) is analogous to MOVSX (13.1.1) in x86 as well, but works not with memory, but with register. All the rest —just the same.

18.5 Nested structures

Now what about situations when one structure defines another structure inside?

```
#include <stdio.h>

struct inner_struct
{
    int a;
    int b;
};

struct outer_struct
{
    char a;
    int b;
    struct inner_struct c;
    char d;
    int e;
};

void f(struct outer_struct s)
{
    printf ("a=%d; b=%d; c.a=%d; c.b=%d; d=%d; e=%d\n",
        s.a, s.b, s.c.a, s.c.b, s.d, s.e);
};
```

...in this case, both inner_struct fields will be placed between a,b and d,e fields of outer_struct.

Let's compile (MSVC 2010):

Listing 18.18: MSVC 2010

```
_s$ = 8 ; size = 24
```

```
_f  PROC
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _s$[ebp+20] ; e
    push   eax
    movsx  ecx, BYTE PTR _s$[ebp+16] ; d
    push   ecx
    mov    edx, DWORD PTR _s$[ebp+12] ; c.b
    push   edx
    mov    eax, DWORD PTR _s$[ebp+8] ; c.a
    push   eax
    mov    ecx, DWORD PTR _s$[ebp+4] ; b
    push   ecx
    movsx  edx, BYTE PTR _s$[ebp] ;a
    push   edx
    push   OFFSET $SG2466
    call   _printf
    add    esp, 28
    pop    ebp
    ret    0
_f  ENDP
```

One curious point here is that by looking onto this assembly code, we do not even see that another structure was used inside of it! Thus, we would say, nested structures are finally unfolds into *linear* or *one-dimensional* structure.

Of course, if to replace `struct inner_struct c;` declaration to `struct inner_struct *c;` (thus making a pointer here) situation will be quite different.

18.6 Bit fields in structure

18.6.1 CPUID example

C/C++ language allow to define exact number of bits for each structure fields. It is very useful if one needs to save memory space. For example, one bit is enough for variable of *bool* type. But of course, it is not rational if speed is important.

Let's consider `CPUID`[7] instruction example. This instruction returning information about current CPU and its features.

If the `EAX` is set to 1 before instruction execution, `CPUID` will return this information packed into the `EAX` register:

[7] http://en.wikipedia.org/wiki/CPUID

3:0	Stepping
7:4	Model
11:8	Family
13:12	Processor Type
19:16	Extended Model
27:20	Extended Family

MSVC 2010 has CPUID macro, but GCC 4.4.1 —has not. So let's make this function by yourself for GCC with the help of its built-in assembler[8].

```
#include <stdio.h>

#ifdef __GNUC__
static inline void cpuid(int code, int *a, int *b, int *c, int *↙
    ↳ d) {
  asm volatile("cpuid":"=a"(*a),"=b"(*b),"=c"(*c),"=d"(*d):"a"(↙
    ↳ code));
}
#endif

#ifdef _MSC_VER
#include <intrin.h>
#endif

struct CPUID_1_EAX
{
    unsigned int stepping:4;
    unsigned int model:4;
    unsigned int family_id:4;
    unsigned int processor_type:2;
    unsigned int reserved1:2;
    unsigned int extended_model_id:4;
    unsigned int extended_family_id:8;
    unsigned int reserved2:4;
};

int main()
{
    struct CPUID_1_EAX *tmp;
    int b[4];

#ifdef _MSC_VER
    __cpuid(b,1);
```

[8] More about internal GCC assembler

```
#endif

#ifdef __GNUC__
    cpuid (1, &b[0], &b[1], &b[2], &b[3]);
#endif

    tmp=(struct CPUID_1_EAX *)&b[0];

    printf ("stepping=%d\n", tmp->stepping);
    printf ("model=%d\n", tmp->model);
    printf ("family_id=%d\n", tmp->family_id);
    printf ("processor_type=%d\n", tmp->processor_type);
    printf ("extended_model_id=%d\n", tmp->extended_model_id);
    printf ("extended_family_id=%d\n", tmp->extended_family_id);

    return 0;
};
```

After CPUID will fill EAX/EBX/ECX/EDX, these registers will be reflected in the b[] array. Then, we have a pointer to the CPUID_1_EAX structure and we point it to the value in the EAX from b[] array.

In other words, we treat 32-bit *int* value as a structure.

Then we read from the stucture.

Let's compile it in MSVC 2008 with /Ox option:

Listing 18.19: Optimizing MSVC 2008

```
_b$ = -16  ; size = 16
_main    PROC
    sub    esp, 16
    push   ebx

    xor    ecx, ecx
    mov    eax, 1
    cpuid
    push   esi
    lea    esi, DWORD PTR _b$[esp+24]
    mov    DWORD PTR [esi], eax
    mov    DWORD PTR [esi+4], ebx
    mov    DWORD PTR [esi+8], ecx
    mov    DWORD PTR [esi+12], edx

    mov    esi, DWORD PTR _b$[esp+24]
    mov    eax, esi
    and    eax, 15
```

```
    push   eax
    push   OFFSET $SG15435 ; 'stepping=%d', 0aH, 00H
    call   _printf

    mov    ecx, esi
    shr    ecx, 4
    and    ecx, 15
    push   ecx
    push   OFFSET $SG15436 ; 'model=%d', 0aH, 00H
    call   _printf

    mov    edx, esi
    shr    edx, 8
    and    edx, 15
    push   edx
    push   OFFSET $SG15437 ; 'family_id=%d', 0aH, 00H
    call   _printf

    mov    eax, esi
    shr    eax, 12
    and    eax, 3
    push   eax
    push   OFFSET $SG15438 ; 'processor_type=%d', 0aH, 00H
    call   _printf

    mov    ecx, esi
    shr    ecx, 16
    and    ecx, 15
    push   ecx
    push   OFFSET $SG15439 ; 'extended_model_id=%d', 0aH, 00H
    call   _printf

    shr    esi, 20
    and    esi, 255
    push   esi
    push   OFFSET $SG15440 ; 'extended_family_id=%d', 0aH, 00H
    call   _printf
    add    esp, 48
    pop    esi

    xor    eax, eax
    pop    ebx
```

```
    add     esp, 16
    ret     0
_main    ENDP
```

SHR instruction shifting value in the EAX register by number of bits must be *skipped*, e.g., we ignore a bits *at right*.

AND instruction clears bits not needed *at left*, or, in other words, leaves only those bits in the EAX register we need now.

Let's try GCC 4.4.1 with -O3 option.

Listing 18.20: Optimizing GCC 4.4.1

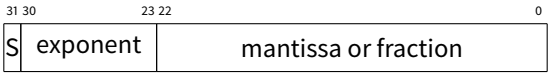
```
main            proc near ; DATA XREF: _start+17
    push    ebp
    mov     ebp, esp
    and     esp, 0FFFFFFF0h
    push    esi
    mov     esi, 1
    push    ebx
    mov     eax, esi
    sub     esp, 18h
    cpuid
    mov     esi, eax
    and     eax, 0Fh
    mov     [esp+8], eax
    mov     dword ptr [esp+4], offset aSteppingD ; "stepping=%d\↙
    ↳ n"
    mov     dword ptr [esp], 1
    call    ___printf_chk
    mov     eax, esi
    shr     eax, 4
    and     eax, 0Fh
    mov     [esp+8], eax
    mov     dword ptr [esp+4], offset aModelD ; "model=%d\n"
    mov     dword ptr [esp], 1
    call    ___printf_chk
    mov     eax, esi
    shr     eax, 8
    and     eax, 0Fh
    mov     [esp+8], eax
    mov     dword ptr [esp+4], offset aFamily_idD ; "family_id=%↙
    ↳ d\n"
    mov     dword ptr [esp], 1
    call    ___printf_chk
    mov     eax, esi
```

```
    shr     eax, 0Ch
    and     eax, 3
    mov     [esp+8], eax
    mov     dword ptr [esp+4], offset aProcessor_type ; "↙
    ↳ processor_type=%d\n"
    mov     dword ptr [esp], 1
    call    ___printf_chk
    mov     eax, esi
    shr     eax, 10h
    shr     esi, 14h
    and     eax, 0Fh
    and     esi, 0FFh
    mov     [esp+8], eax
    mov     dword ptr [esp+4], offset aExtended_model ; "↙
    ↳ extended_model_id=%d\n"
    mov     dword ptr [esp], 1
    call    ___printf_chk
    mov     [esp+8], esi
    mov     dword ptr [esp+4], offset unk_80486D0
    mov     dword ptr [esp], 1
    call    ___printf_chk
    add     esp, 18h
    xor     eax, eax
    pop     ebx
    pop     esi
    mov     esp, ebp
    pop     ebp
    retn
main            endp
```

Almost the same. The only thing worth noting is the GCC somehow united calculation of `extended_model_id` and `extended_family_id` into one block, instead of calculating them separately, before corresponding each `printf()` call.

18.6.2 Working with the float type as with a structure

As it was already noted in section about FPU (15), both *float* and *double* types consisted of sign, significand (or fraction) and exponent. But will we able to work with these fields directly? Let's try with *float*.



(S—sign)

```
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <memory.h>

struct float_as_struct
{
    unsigned int fraction : 23; // fractional part
    unsigned int exponent : 8;  // exponent + 0x3FF
    unsigned int sign : 1;      // sign bit
};

float f(float _in)
{
    float f=_in;
    struct float_as_struct t;

    assert (sizeof (struct float_as_struct) == sizeof (float));

    memcpy (&t, &f, sizeof (float));

    t.sign=1; // set negative sign
    t.exponent=t.exponent+2; // multiple d by 2^n (n here is 2)

    memcpy (&f, &t, sizeof (float));

    return f;
};

int main()
{
    printf ("%f\n", f(1.234));
};
```

`float_as_struct` structure occupies as much space is memory as *float*, e.g., 4 bytes or 32 bits.

Now we setting negative sign in input value and also by adding 2 to exponent we thereby multiplicating the whole number by 2^2, e.g., by 4.

Let's compile in MSVC 2008 without optimization:

Listing 18.21: Non-optimizing MSVC 2008

```
_t$ = -8   ; size = 4
_f$ = -4   ; size = 4
```

```
__in$ = 8  ; size = 4
?f@@YAMM@Z PROC ; f
    push   ebp
    mov    ebp, esp
    sub    esp, 8

    fld    DWORD PTR __in$[ebp]
    fstp   DWORD PTR _f$[ebp]

    push   4
    lea    eax, DWORD PTR _f$[ebp]
    push   eax
    lea    ecx, DWORD PTR _t$[ebp]
    push   ecx
    call   _memcpy
    add    esp, 12

    mov    edx, DWORD PTR _t$[ebp]
    or     edx, -2147483648 ; 80000000H - set minus sign
    mov    DWORD PTR _t$[ebp], edx

    mov    eax, DWORD PTR _t$[ebp]
    shr    eax, 23          ; 00000017H - drop significand
    and    eax, 255         ; 000000ffH - leave here only ↙
    ↳ exponent
    add    eax, 2           ; add 2 to it
    and    eax, 255         ; 000000ffH
    shl    eax, 23          ; 00000017H - shift result to place ↙
    ↳ of bits 30:23
    mov    ecx, DWORD PTR _t$[ebp]
    and    ecx, -2139095041 ; 807fffffH - drop exponent
    or     ecx, eax         ; add original value without ↙
    ↳ exponent with new calculated exponent
    mov    DWORD PTR _t$[ebp], ecx

    push   4
    lea    edx, DWORD PTR _t$[ebp]
    push   edx
    lea    eax, DWORD PTR _f$[ebp]
    push   eax
    call   _memcpy
    add    esp, 12
```

```
    fld     DWORD PTR _f$[ebp]

    mov     esp, ebp
    pop     ebp
    ret     0
?f@@YAMM@Z ENDP   ; f
```

Redundant for a bit. If it is compiled with `/Ox` flag there is no `memcpy()` call, `f` variable is used directly. But it is easier to understand it all considering unoptimized version.

What GCC 4.4.1 with `-O3` will do?

Listing 18.22: Optimizing GCC 4.4.1

```
; f(float)
        public _Z1ff
_Z1ff   proc near

var_4   = dword ptr -4
arg_0   = dword ptr  8

        push    ebp
        mov     ebp, esp
        sub     esp, 4
        mov     eax, [ebp+arg_0]
        or      eax, 80000000h  ; set minus sign
        mov     edx, eax
        and     eax, 807FFFFFh  ; leave only significand and ↙
    ↳ exponent in EAX
        shr     edx, 23         ; prepare exponent
        add     edx, 2          ; add 2
        movzx   edx, dl         ; clear all bits except 7:0 in ↙
    ↳ EAX
        shl     edx, 23         ; shift new calculated exponent ↙
    ↳ to its place
        or      eax, edx        ; add new exponent and original ↙
    ↳ value without exponent
        mov     [ebp+var_4], eax
        fld     [ebp+var_4]
        leave
        retn
_Z1ff   endp

        public main
main    proc near
```

```
        push    ebp
        mov     ebp, esp
        and     esp, 0FFFFFFF0h
        sub     esp, 10h
        fld     ds:dword_8048614 ; -4.936
        fstp    qword ptr [esp+8]
        mov     dword ptr [esp+4], offset asc_8048610 ; "%f\n"
        mov     dword ptr [esp], 1
        call    ___printf_chk
        xor     eax, eax
        leave
        retn
main    endp
```

The `f()` function is almost understandable. However, what is interesting, GCC was able to calculate `f(1.234)` result during compilation stage despite all this hodgepodge with structure fields and prepared this argument to the `printf()` as precalculated!

Chapter 19

Unions

19.1 Pseudo-random number generator example

If we need float random numbers from 0 to 1, the most simplest thing is to use PRNG[1] like Mersenne twister produces random 32-bit values in DWORD form, transform this value to *float* and then dividing it by `RAND_MAX` (`0xFFFFFFFF` in our case) —value we got will be in 0..1 interval.

But as we know, division operation is slow. Will it be possible to get rid of it, as in case of division by multiplication? (14)

Let's recall what float number consisted of: sign bit, significand bits and exponent bits. We need just to store random bits to all significand bits for getting random float number!

Exponent cannot be zero (number will be denormalized in this case), so we will store `01111111` to exponent —this means exponent will be 1. Then fill significand with random bits, set sign bit to 0 (which means positive number) and voilà. Generated numbers will be in 1 to 2 interval, so we also must subtract 1 from it.

Very simple linear congruential random numbers generator is used in my example[2], produces 32-bit numbers. The PRNG initializing by current time in UNIX-style.

Then, *float* type represented as *union* —it is the C/C++ construction enabling us to interpret piece of memory as differently typed. In our case, we are able to create a variable of `union` type and then access to it as it is *float* or as it is *uint32_t*. It can be said, it is just a hack. A dirty one.

```
#include <stdio.h>
#include <stdint.h>
#include <time.h>
```

[1] Pseudorandom number generator

[2] idea was taken from: http://xor0110.wordpress.com/2010/09/24/how-to-generate-floating-point-random-numbers-efficiently

```
union uint32_t_float
{
    uint32_t i;
    float f;
};

// from the Numerical Recipes book
const uint32_t RNG_a=1664525;
const uint32_t RNG_c=1013904223;

int main()
{
    uint32_t_float tmp;

    uint32_t RNG_state=time(NULL); // initial seed
    for (int i=0; i<100; i++)
    {
        RNG_state=RNG_state*RNG_a+RNG_c;
        tmp.i=RNG_state & 0x007fffff | 0x3F800000;
        float x=tmp.f-1;
        printf ("%f\n", x);
    };
    return 0;
};
```

Listing 19.1: MSVC 2010 (/Ox)

```
$SG4232    DB    '%f', 0aH, 00H

__real@3ff0000000000000 DQ 03ff0000000000000r    ; 1

tv140 = -4                          ; size = 4
_tmp$ = -4                          ; size = 4
_main    PROC
    push   ebp
    mov    ebp, esp
    and    esp, -64                 ; ffffffc0H
    sub    esp, 56                  ; 00000038H
    push   esi
    push   edi
    push   0
    call   __time64
    add    esp, 4
```

```
    mov    esi, eax
    mov    edi, 100                ; 00000064H
$LN3@main:

; let's generate random 32-bit number

    imul   esi, 1664525            ; 0019660dH
    add    esi, 1013904223         ; 3c6ef35fH
    mov    eax, esi

; leave bits for significand only

    and    eax, 8388607            ; 007fffffH

; set exponent to 1

    or     eax, 1065353216         ; 3f800000H

; store this value as int

    mov    DWORD PTR _tmp$[esp+64], eax
    sub    esp, 8

; load this value as float

    fld    DWORD PTR _tmp$[esp+72]

; subtract one from it

    fsub   QWORD PTR __real@3ff0000000000000
    fstp   DWORD PTR tv140[esp+72]
    fld    DWORD PTR tv140[esp+72]
    fstp   QWORD PTR [esp]
    push   OFFSET $SG4232
    call   _printf
    add    esp, 12                 ; 0000000cH
    dec    edi
    jne    SHORT $LN3@main
    pop    edi
    xor    eax, eax
    pop    esi
    mov    esp, ebp
    pop    ebp
```

```
    ret     0
_main    ENDP
_TEXT    ENDS
END
```

GCC produces very similar code.

Chapter 20

Pointers to functions

Pointer to function, as any other pointer, is just an address of function beginning in its code segment.

It is often used in callbacks [1].

Well-known examples are:

- `qsort()`[2], `atexit()`[3] from the standard C library;
- signals in *NIX OS[4];
- thread starting: `CreateThread()` (win32), `pthread_create()` (POSIX);
- a lot of win32 functions, e.g. `EnumChildWindows()`[5].

So, `qsort()` function is a C/C++ standard library quicksort implementation. The functions is able to sort anything, any types of data, if you have a function for two elements comparison and `qsort()` is able to call it.

The comparison function can be defined as:

```
int (*compare)(const void *, const void *)
```

Let's use slightly modified example I found here:

```
1 /* ex3 Sorting ints with qsort */
2
3 #include <stdio.h>
4 #include <stdlib.h>
5
```

[1] http://en.wikipedia.org/wiki/Callback_(computer_science)
[2] http://en.wikipedia.org/wiki/Qsort_(C_standard_library)
[3] http://www.opengroup.org/onlinepubs/009695399/functions/atexit.html
[4] http://en.wikipedia.org/wiki/Signal.h
[5] http://msdn.microsoft.com/en-us/library/ms633494(VS.85).aspx

```
6  int comp(const void * _a, const void * _b)
7  {
8    const int *a=(const int *)_a;
9    const int *b=(const int *)_b;
10
11   if (*a==*b)
12     return 0;
13   else
14     if (*a < *b)
15         return -1;
16      else
17       return 1;
18 }
19
20 int main(int argc, char* argv[])
21 {
22    int numbers↙
       ↳ [10]={1892,45,200,-98,4087,5,-12345,1087,88,-100000};
23    int i;
24
25   /* Sort the array */
26   qsort(numbers,10,sizeof(int),comp) ;
27   for (i=0;i<9;i++)
28     printf("Number = %d\n",numbers[ i ]) ;
29   return 0;
30 }
```

20.1 MSVC

Let's compile it in MSVC 2010 (I omitted some parts for the sake of brevity) with `/Ox` option:

Listing 20.1: Optimizing MSVC 2010: /Ox /GS- /MD

```
__a$ = 8                                                ; size =↙
    ↳ 4
__b$ = 12                                               ; size =↙
    ↳ 4
_comp   PROC
        mov     eax, DWORD PTR __a$[esp-4]
        mov     ecx, DWORD PTR __b$[esp-4]
        mov     eax, DWORD PTR [eax]
        mov     ecx, DWORD PTR [ecx]
```

```
        cmp     eax, ecx
        jne     SHORT $LN4@comp
        xor     eax, eax
        ret     0
$LN4@comp:
        xor     edx, edx
        cmp     eax, ecx
        setge   dl
        lea     eax, DWORD PTR [edx+edx-1]
        ret     0
_comp   ENDP


_numbers$ = -40                                         ; size =↙
    ↳  40
_argc$ = 8                                              ; size =↙
    ↳  4
_argv$ = 12                                             ; size =↙
    ↳  4
_main   PROC
        sub     esp, 40                                 ; ↙
    ↳ 00000028H
        push    esi
        push    OFFSET _comp
        push    4
        lea     eax, DWORD PTR _numbers$[esp+52]
        push    10                                      ; ↙
    ↳ 0000000aH
        push    eax
        mov     DWORD PTR _numbers$[esp+60], 1892       ; ↙
    ↳ 00000764H
        mov     DWORD PTR _numbers$[esp+64], 45         ; ↙
    ↳ 0000002dH
        mov     DWORD PTR _numbers$[esp+68], 200        ; 000000↙
    ↳ c8H
        mov     DWORD PTR _numbers$[esp+72], -98        ; ↙
    ↳ ffffff9eH
        mov     DWORD PTR _numbers$[esp+76], 4087       ; 00000↙
    ↳ ff7H
        mov     DWORD PTR _numbers$[esp+80], 5
        mov     DWORD PTR _numbers$[esp+84], -12345     ; ↙
    ↳ ffffcfc7H
        mov     DWORD PTR _numbers$[esp+88], 1087       ; ↙
```

```
    ↪ 0000043fH
        mov     DWORD PTR _numbers$[esp+92], 88         ; ↙
    ↪ 00000058H
        mov     DWORD PTR _numbers$[esp+96], -100000    ; ↙
    ↪ fffe7960H
        call    _qsort
        add     esp, 16                                 ; ↙
    ↪ 00000010H

...
```

Nothing surprising so far. As a fourth argument, an address of label `_comp` is passed, that is just a place where function `comp()` located.

How `qsort()` calling it?

Let's take a look into this function located in MSVCR80.DLL (a MSVC DLL module with C standard library functions):

Listing 20.2: MSVCR80.DLL

```
.text:7816CBF0 ; void __cdecl qsort(void *, unsigned int, ↙
    ↪ unsigned int, int (__cdecl *)(const void *, const void *))
.text:7816CBF0                  public _qsort
.text:7816CBF0 _qsort           proc near
.text:7816CBF0
.text:7816CBF0 lo               = dword ptr -104h
.text:7816CBF0 hi               = dword ptr -100h
.text:7816CBF0 var_FC           = dword ptr -0FCh
.text:7816CBF0 stkptr           = dword ptr -0F8h
.text:7816CBF0 lostk            = dword ptr -0F4h
.text:7816CBF0 histk            = dword ptr -7Ch
.text:7816CBF0 base             = dword ptr  4
.text:7816CBF0 num              = dword ptr  8
.text:7816CBF0 width            = dword ptr  0Ch
.text:7816CBF0 comp             = dword ptr  10h
.text:7816CBF0
.text:7816CBF0                  sub     esp, 100h

....

.text:7816CCE0 loc_7816CCE0:                            ; CODE ↙
    ↪ XREF: _qsort+B1
.text:7816CCE0                  shr     eax, 1
.text:7816CCE2                  imul    eax, ebp
.text:7816CCE5                  add     eax, ebx
.text:7816CCE7                  mov     edi, eax
```

```
.text:7816CCE9                  push    edi
.text:7816CCEA                  push    ebx
.text:7816CCEB                  call    [esp+118h+comp]
.text:7816CCF2                  add     esp, 8
.text:7816CCF5                  test    eax, eax
.text:7816CCF7                  jle     short loc_7816CD04
```

comp—is fourth function argument. Here the control is just passed to the address in the comp argument. Before it, two arguments prepared for comp(). Its result is checked after its execution.

That's why it is dangerous to use pointers to functions. First of all, if you call qsort() with incorrect pointer to function, qsort() may pass control to incorrect point, a process may crash and this bug will be hard to find.

Second reason is the callback function types must comply strictly, calling wrong function with wrong arguments of wrong types may lead to serious problems, however, process crashing is not a big problem —big problem is to determine a reason of crashing —because compiler may be silent about potential trouble while compiling.

20.1.1 MSVC + OllyDbg

Let's load our example into OllyDbg and set breakpoint on comp() function.

How values are compared we can see at the very first comp() call: fig. 20.1. OllyDbg shows compared values in the window under code window, for convenience. We can also see that the SP pointing to RA where the place in qsort() function is (actually located in MSVCR100.DLL).

By tracing (F8) until RETN instruction, and pressing F8 one more time, we returning into qsort() function: fig. 20.2. That was a call to comparison function.

Here is also screenshot of the moment of the second call of comp()—now values to be compared are different: fig. 20.3.

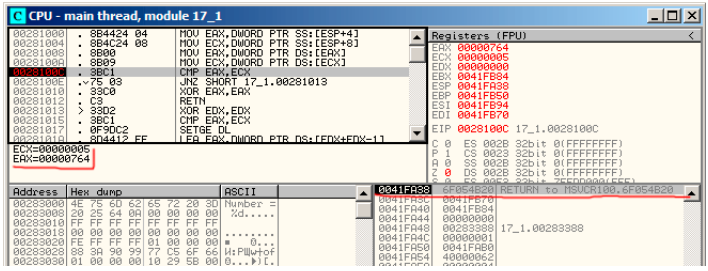


Figure 20.1: OllyDbg: first call of comp()

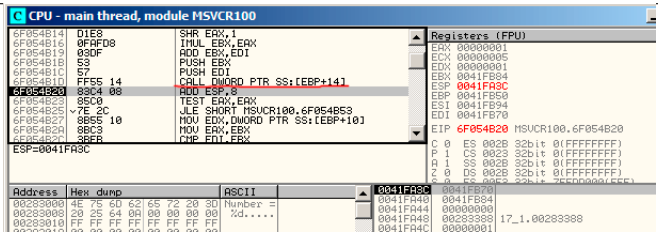


Figure 20.2: OllyDbg: the code in `qsort()` right after `comp()` call

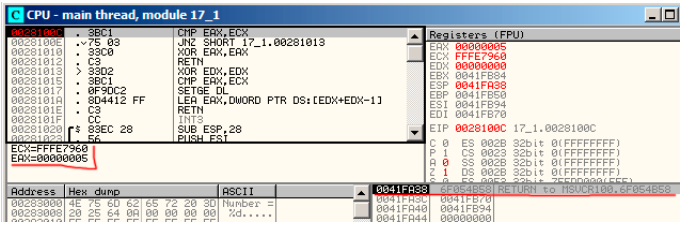


Figure 20.3: OllyDbg: second call of `comp()`

20.1.2 MSVC + tracer

Let's also see, which pairs are compared. These 10 numbers are being sorted: 1892, 45, 200, -98, 4087, 5, -12345, 1087, 88, -100000.

I found the address of the first `CMP` instruction in `comp()`, it is `0x0040100C` and I'm setting breakpoint on it:

```
tracer.exe -l:17_1.exe bpx=17_1.exe!0x0040100C
```

I'm getting information about registers at breakpoint:

```
PID=4336|New process 17_1.exe
(0) 17_1.exe!0x40100c
EAX=0x00000764 EBX=0x0051f7c8 ECX=0x00000005 EDX=0x00000000
ESI=0x0051f7d8 EDI=0x0051f7b4 EBP=0x0051f794 ESP=0x0051f67c
EIP=0x0028100c
FLAGS=IF
(0) 17_1.exe!0x40100c
EAX=0x00000005 EBX=0x0051f7c8 ECX=0xfffe7960 EDX=0x00000000
ESI=0x0051f7d8 EDI=0x0051f7b4 EBP=0x0051f794 ESP=0x0051f67c
EIP=0x0028100c
FLAGS=PF ZF IF
(0) 17_1.exe!0x40100c
```

```
EAX=0x00000764 EBX=0x0051f7c8 ECX=0x00000005 EDX=0x00000000
ESI=0x0051f7d8 EDI=0x0051f7b4 EBP=0x0051f794 ESP=0x0051f67c
EIP=0x0028100c
FLAGS=CF PF ZF IF
...
```

I filtered out EAX and ECX and got:

```
EAX=0x00000764 ECX=0x00000005
EAX=0x00000005 ECX=0xfffe7960
EAX=0x00000764 ECX=0x00000005
EAX=0x0000002d ECX=0x00000005
EAX=0x00000058 ECX=0x00000005
EAX=0x0000043f ECX=0x00000005
EAX=0xffffcfc7 ECX=0x00000005
EAX=0x000000c8 ECX=0x00000005
EAX=0xffffff9e ECX=0x00000005
EAX=0x00000ff7 ECX=0x00000005
EAX=0x00000ff7 ECX=0x00000005
EAX=0xffffff9e ECX=0x00000005
EAX=0xffffff9e ECX=0x00000005
EAX=0xffffcfc7 ECX=0xfffe7960
EAX=0x00000005 ECX=0xffffcfc7
EAX=0xffffff9e ECX=0x00000005
EAX=0xffffcfc7 ECX=0xfffe7960
EAX=0xffffff9e ECX=0xffffcfc7
EAX=0xffffcfc7 ECX=0xfffe7960
EAX=0x000000c8 ECX=0x00000ff7
EAX=0x0000002d ECX=0x00000ff7
EAX=0x0000043f ECX=0x00000ff7
EAX=0x00000058 ECX=0x00000ff7
EAX=0x00000764 ECX=0x00000ff7
EAX=0x000000c8 ECX=0x00000764
EAX=0x0000002d ECX=0x00000764
EAX=0x0000043f ECX=0x00000764
EAX=0x00000058 ECX=0x00000764
EAX=0x000000c8 ECX=0x00000058
EAX=0x0000002d ECX=0x000000c8
EAX=0x0000043f ECX=0x000000c8
EAX=0x000000c8 ECX=0x00000058
EAX=0x0000002d ECX=0x000000c8
EAX=0x0000002d ECX=0x00000058
```

That's 34 pairs. Therefore, quick sort algorithm needs 34 comparison operations for sorting these 10 numbers.

20.1.3 MSVC + tracer (code coverage)

We can also use tracer's feature to collect all possible register's values and show them in IDA.

Let's trace all instructions in `comp()` function:

```
tracer.exe -l:17_1.exe bpf=17_1.exe!0x00401000,trace:cc
```

We getting .idc-script for loading into IDA and load it: fig. 20.4.

IDA gave the function name (PtFuncCompare) —it seems, because IDA sees that pointer to this function is passed into `qsort()`.

We see that a and b pointers are points to various places in array, but step between points is 4—indeed, 32-bit values are stored in the array.

We see that the instructions at `0x401010` and `0x401012` was never executed (so they leaved as white): indeed, `comp()` was never returned 0, because there no equal elements.

```
.text:00401000
.text:00401000 ; int __cdecl PtFuncCompare(const void *, const void *)
.text:00401000 PtFuncCompare   proc near               ; DATA XREF: _main+5↓o
.text:00401000
.text:00401000 arg_0           = dword ptr  4
.text:00401000 arg_4           = dword ptr  8
.text:00401000
.text:00401000                 mov     eax, [esp+arg_0] ; [ESP+4]=0x45f7ec..0x45f810(step=4), L"?\x04?
.text:00401004                 mov     ecx, [esp+arg_4] ; [ESP+8]=0x45f7ec..0x45f7f4(step=4), 0x45f7fc
.text:00401008                 mov     eax, [eax]       ; [EAX]=5, 0x2d, 0x58, 0xc8, 0x43f, 0x764, 0xff
.text:0040100A                 mov     ecx, [ecx]       ; [ECX]=5, 0x58, 0xc8, 0x764, 0xff7, 0xfffe7960
.text:0040100C                 cmp     eax, ecx         ; EAX=5, 0x2d, 0x58, 0xc8, 0x43f, 0x764, 0xff7,
.text:0040100E                 jnz     short loc_401013 ; ZF=false
.text:00401010                 xor     eax, eax
.text:00401012                 retn
.text:00401013 ; ---------------------------------------------------------------------------
.text:00401013
.text:00401013 loc_401013:                              ; CODE XREF: PtFuncCompare+E↑j
.text:00401013                 xor     edx, edx
.text:00401015                 cmp     eax, ecx         ; EAX=5, 0x2d, 0x58, 0xc8, 0x43f, 0x764, 0xff7,
.text:00401017                 setnl   dl               ; SF=false,true OF=false
.text:0040101A                 lea     eax, [edx+edx-1]
.text:0040101E                 retn                     ; EAX=1, 0xffffffff
.text:0040101E PtFuncCompare   endp
.text:0040101F
```

Figure 20.4: tracer and IDA. N.B.: some values are cutted at right

20.2 GCC

Not a big difference:

Listing 20.3: GCC

```
        lea     eax, [esp+40h+var_28]
        mov     [esp+40h+var_40], eax
        mov     [esp+40h+var_28], 764h
        mov     [esp+40h+var_24], 2Dh
        mov     [esp+40h+var_20], 0C8h
        mov     [esp+40h+var_1C], 0FFFFFF9Eh
```

```
                mov     [esp+40h+var_18], 0FF7h
                mov     [esp+40h+var_14], 5
                mov     [esp+40h+var_10], 0FFFFCFC7h
                mov     [esp+40h+var_C], 43Fh
                mov     [esp+40h+var_8], 58h
                mov     [esp+40h+var_4], 0FFFE7960h
                mov     [esp+40h+var_34], offset comp
                mov     [esp+40h+var_38], 4
                mov     [esp+40h+var_3C], 0Ah
                call    _qsort
```

`comp()` function:

```
                public comp
comp            proc near

arg_0           = dword ptr  8
arg_4           = dword ptr  0Ch

                push    ebp
                mov     ebp, esp
                mov     eax, [ebp+arg_4]
                mov     ecx, [ebp+arg_0]
                mov     edx, [eax]
                xor     eax, eax
                cmp     [ecx], edx
                jnz     short loc_8048458
                pop     ebp
                retn
loc_8048458:
                setnl   al
                movzx   eax, al
                lea     eax, [eax+eax-1]
                pop     ebp
                retn
comp            endp
```

`qsort()` implementation is located in the `libc.so.6` and it is in fact just a wrapper [6] for `qsort_r()`.

It will call then `quicksort()`, where our defined function will be called via passed pointer:

Listing 20.4: (file libc.so.6, glibc version—2.10.1)

[6] a concept like thunk function

```
.text:0002DDF6                  mov     edx, [ebp+arg_10]
.text:0002DDF9                  mov     [esp+4], esi
.text:0002DDFD                  mov     [esp], edi
.text:0002DE00                  mov     [esp+8], edx
.text:0002DE04                  call    [ebp+arg_C]
...
```

20.2.1 GCC + GDB (with source code)

Obviously, we have a C-source code of our example (20), so we can set breakpoint (b) on line number (11th—the line where first comparison is occurred). We also need to compile example with debugging information included (-g), so the table with addresses and corresponding line numbers is present. We can also print values by variable name (p): debugging information also has information about which register and/or local stack element contain which variable.

We can also see stack (bt) and find out that there are some intermediate function `msort_with_tmp()` used in Glibc.

Listing 20.5: GDB session

```
dennis@ubuntuvm:~/polygon$ gcc 17_1.c -g
dennis@ubuntuvm:~/polygon$ gdb ./a.out
GNU gdb (GDB) 7.6.1-ubuntu
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/↙
    ↳ licenses/gpl.html>
This is free software: you are free to change and redistribute ↙
    ↳ it.
There is NO WARRANTY, to the extent permitted by law.  Type "↙
    ↳ show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/dennis/polygon/a.out...done.
(gdb) b 17_1.c:11
Breakpoint 1 at 0x804845f: file 17_1.c, line 11.
(gdb) run
Starting program: /home/dennis/polygon/./a.out

Breakpoint 1, comp (_a=0xbffff0f8, _b=_b@entry=0xbffff0fc) at 17↙
    ↳ _1.c:11
11          if (*a==*b)
```

```
(gdb) p *a
$1 = 1892
(gdb) p *b
$2 = 45
(gdb) c
Continuing.

Breakpoint 1, comp (_a=0xbffff104, _b=_b@entry=0xbffff108) at 17↙
    ↳ _1.c:11
11        if (*a==*b)
(gdb) p *a
$3 = -98
(gdb) p *b
$4 = 4087
(gdb) bt
#0  comp (_a=0xbffff0f8, _b=_b@entry=0xbffff0fc) at 17_1.c:11
#1  0xb7e42872 in msort_with_tmp (p=p@entry=0xbffff07c, b=↙
    ↳ b@entry=0xbffff0f8, n=n@entry=2)
    at msort.c:65
#2  0xb7e4273e in msort_with_tmp (n=2, b=0xbffff0f8, p=0↙
    ↳ xbffff07c) at msort.c:45
#3  msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8, ↙
    ↳ n=n@entry=5) at msort.c:53
#4  0xb7e4273e in msort_with_tmp (n=5, b=0xbffff0f8, p=0↙
    ↳ xbffff07c) at msort.c:45
#5  msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8, ↙
    ↳ n=n@entry=10) at msort.c:53
#6  0xb7e42cef in msort_with_tmp (n=10, b=0xbffff0f8, p=0↙
    ↳ xbffff07c) at msort.c:45
#7  __GI_qsort_r (b=b@entry=0xbffff0f8, n=n@entry=10, s=s@entry↙
    ↳ =4, cmp=cmp@entry=0x804844d <comp>,
    arg=arg@entry=0x0) at msort.c:297
#8  0xb7e42dcf in __GI_qsort (b=0xbffff0f8, n=10, s=4, cmp=0↙
    ↳ x804844d <comp>) at msort.c:307
#9  0x0804850d in main (argc=1, argv=0xbffff1c4) at 17_1.c:26
(gdb)
```

20.2.2 GCC + GDB (no source code)

But often there are no source code at all, so we can disassemble `comp()` function (`disas`), find the very first `CMP` instruction and set breakpoint (`b`) at that address. At each breakpoint, we will dump all register contents (`info registers`). Stack information is also available (`bt`), but partial: there are no line number information for

comp() function.

Listing 20.6: GDB session

```
dennis@ubuntuvm:~/polygon$ gcc 17_1.c
dennis@ubuntuvm:~/polygon$ gdb ./a.out
GNU gdb (GDB) 7.6.1-ubuntu
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/↙
    ↳ licenses/gpl.html>
This is free software: you are free to change and redistribute ↙
    ↳ it.
There is NO WARRANTY, to the extent permitted by law.  Type "↙
    ↳ show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/dennis/polygon/a.out...(no debugging ↙
    ↳ symbols found)...done.
(gdb) set disassembly-flavor intel
(gdb) disas comp
Dump of assembler code for function comp:
   0x0804844d <+0>:     push   ebp
   0x0804844e <+1>:     mov    ebp,esp
   0x08048450 <+3>:     sub    esp,0x10
   0x08048453 <+6>:     mov    eax,DWORD PTR [ebp+0x8]
   0x08048456 <+9>:     mov    DWORD PTR [ebp-0x8],eax
   0x08048459 <+12>:    mov    eax,DWORD PTR [ebp+0xc]
   0x0804845c <+15>:    mov    DWORD PTR [ebp-0x4],eax
   0x0804845f <+18>:    mov    eax,DWORD PTR [ebp-0x8]
   0x08048462 <+21>:    mov    edx,DWORD PTR [eax]
   0x08048464 <+23>:    mov    eax,DWORD PTR [ebp-0x4]
   0x08048467 <+26>:    mov    eax,DWORD PTR [eax]
   0x08048469 <+28>:    cmp    edx,eax
   0x0804846b <+30>:    jne    0x8048474 <comp+39>
   0x0804846d <+32>:    mov    eax,0x0
   0x08048472 <+37>:    jmp    0x804848e <comp+65>
   0x08048474 <+39>:    mov    eax,DWORD PTR [ebp-0x8]
   0x08048477 <+42>:    mov    edx,DWORD PTR [eax]
   0x08048479 <+44>:    mov    eax,DWORD PTR [ebp-0x4]
   0x0804847c <+47>:    mov    eax,DWORD PTR [eax]
   0x0804847e <+49>:    cmp    edx,eax
   0x08048480 <+51>:    jge    0x8048489 <comp+60>
   0x08048482 <+53>:    mov    eax,0xffffffff
```

```
   0x08048487 <+58>:    jmp    0x804848e <comp+65>
   0x08048489 <+60>:    mov    eax,0x1
   0x0804848e <+65>:    leave
   0x0804848f <+66>:    ret
End of assembler dump.
(gdb) b *0x08048469
Breakpoint 1 at 0x8048469
(gdb) run
Starting program: /home/dennis/polygon/./a.out

Breakpoint 1, 0x08048469 in comp ()
(gdb) info registers
eax            0x2d     45
ecx            0xbffff0f8       -1073745672
edx            0x764    1892
ebx            0xb7fc0000       -1208221696
esp            0xbfffeeb8       0xbfffeeb8
ebp            0xbfffeec8       0xbfffeec8
esi            0xbffff0fc       -1073745668
edi            0xbffff010       -1073745904
eip            0x8048469        0x8048469 <comp+28>
eflags         0x286    [ PF SF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb) c
Continuing.

Breakpoint 1, 0x08048469 in comp ()
(gdb) info registers
eax            0xff7    4087
ecx            0xbffff104       -1073745660
edx            0xffffff9e       -98
ebx            0xb7fc0000       -1208221696
esp            0xbfffee58       0xbfffee58
ebp            0xbfffee68       0xbfffee68
esi            0xbffff108       -1073745656
edi            0xbffff010       -1073745904
eip            0x8048469        0x8048469 <comp+28>
eflags         0x282    [ SF IF ]
```

```
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb) c
Continuing.

Breakpoint 1, 0x08048469 in comp ()
(gdb) info registers
eax            0xffffff9e       -98
ecx            0xbffff100       -1073745664
edx            0xc8     200
ebx            0xb7fc0000       -1208221696
esp            0xbfffeeb8       0xbfffeeb8
ebp            0xbfffeec8       0xbfffeec8
esi            0xbffff104       -1073745660
edi            0xbffff010       -1073745904
eip            0x8048469        0x8048469 <comp+28>
eflags         0x286    [ PF SF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
(gdb) bt
#0  0x08048469 in comp ()
#1  0xb7e42872 in msort_with_tmp (p=p@entry=0xbffff07c, b=↙
    ↳ b@entry=0xbffff0f8, n=n@entry=2)
    at msort.c:65
#2  0xb7e4273e in msort_with_tmp (n=2, b=0xbffff0f8, p=0↙
    ↳ xbffff07c) at msort.c:45
#3  msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8, ↙
    ↳ n=n@entry=5) at msort.c:53
#4  0xb7e4273e in msort_with_tmp (n=5, b=0xbffff0f8, p=0↙
    ↳ xbffff07c) at msort.c:45
#5  msort_with_tmp (p=p@entry=0xbffff07c, b=b@entry=0xbffff0f8, ↙
    ↳ n=n@entry=10) at msort.c:53
#6  0xb7e42cef in msort_with_tmp (n=10, b=0xbffff0f8, p=0↙
    ↳ xbffff07c) at msort.c:45
#7  __GI_qsort_r (b=b@entry=0xbffff0f8, n=n@entry=10, s=s@entry↙
```

```
    ↪ =4, cmp=cmp@entry=0x804844d <comp>,
    arg=arg@entry=0x0) at msort.c:297
#8  0xb7e42dcf in __GI_qsort (b=0xbffff0f8, n=10, s=4, cmp=0↙
    ↪ x804844d <comp>) at msort.c:307
#9  0x0804850d in main ()
```

Chapter 21

64-bit values in 32-bit environment

In the 32-bit environment GPR's are 32-bit, so 64-bit values are passed as 32-bit value pairs [1].

21.1 Arguments passing, addition, subtraction

```
#include <stdint.h>

uint64_t f1 (uint64_t a, uint64_t b)
{
        return a+b;
};

void f1_test ()
{
#ifdef __GNUC__
        printf ("%lld\n", f1(12345678901234, 23456789012345));
#else
        printf ("%I64d\n", f1(12345678901234, 23456789012345));
#endif
};

uint64_t f2 (uint64_t a, uint64_t b)
{
```

[1] By the way, 32-bit values are passed as pairs in 16-bit environment just as the same

```
        return a-b;
};
```

Listing 21.1: MSVC 2012 /Ox /Ob1

```
_a$ = 8         ; size = 8
_b$ = 16        ; size = 8
_f1     PROC
        mov     eax, DWORD PTR _a$[esp-4]
        add     eax, DWORD PTR _b$[esp-4]
        mov     edx, DWORD PTR _a$[esp]
        adc     edx, DWORD PTR _b$[esp]
        ret     0
_f1     ENDP

_f1_test PROC
        push    5461            ; 00001555H
        push    1972608889      ; 75939f79H
        push    2874            ; 00000b3aH
        push    1942892530      ; 73ce2ff2H
        call    _f1
        push    edx
        push    eax
        push    OFFSET $SG1436 ; '%I64d', 0aH, 00H
        call    _printf
        add     esp, 28
        ret     0
_f1_test ENDP

_f2     PROC
        mov     eax, DWORD PTR _a$[esp-4]
        sub     eax, DWORD PTR _b$[esp-4]
        mov     edx, DWORD PTR _a$[esp]
        sbb     edx, DWORD PTR _b$[esp]
        ret     0
_f2     ENDP
```

We may see in the `f1_test()` function as each 64-bit value is passed by two 32-bit values, high part first, then low part.

Addition and subtraction occurring by pairs as well.

While addition, low 32-bit part are added first. If carry was occurred while addition, CF flag is set. The next `ADC` instruction adds high parts of values, but also adding 1 if

CF=1.

Subtraction is also occurred by pairs. The very first SUB may also turn CF flag on, which will be checked in the subsequent SBB instruction: if carry flag is on, then 1 will also be subtracted from the result.

In a 32-bit environment, 64-bit values are returned from a functions in EDX:EAX registers pair. It is easily can be seen how `f1()` function is then passed to `printf()`.

Listing 21.2: GCC 4.8.1 -O1 -fno-inline

```
_f1:
        mov     eax, DWORD PTR [esp+12]
        mov     edx, DWORD PTR [esp+16]
        add     eax, DWORD PTR [esp+4]
        adc     edx, DWORD PTR [esp+8]
        ret

_f1_test:
        sub     esp, 28
        mov     DWORD PTR [esp+8], 1972608889   ; 75939f79H
        mov     DWORD PTR [esp+12], 5461        ; 00001555H
        mov     DWORD PTR [esp], 1942892530     ; 73ce2ff2H
        mov     DWORD PTR [esp+4], 2874         ; 00000b3aH
        call    _f1
        mov     DWORD PTR [esp+4], eax
        mov     DWORD PTR [esp+8], edx
        mov     DWORD PTR [esp], OFFSET FLAT:LC0 ; "%lld\12\0"
        call    _printf
        add     esp, 28
        ret

_f2:
        mov     eax, DWORD PTR [esp+4]
        mov     edx, DWORD PTR [esp+8]
        sub     eax, DWORD PTR [esp+12]
        sbb     edx, DWORD PTR [esp+16]
        ret
```

GCC code is the same.

21.2 Multiplication, division

```
#include <stdint.h>

uint64_t f3 (uint64_t a, uint64_t b)
{
        return a*b;
};

uint64_t f4 (uint64_t a, uint64_t b)
{
        return a/b;
};

uint64_t f5 (uint64_t a, uint64_t b)
{
        return a % b;
};
```

Listing 21.3: MSVC 2012 /Ox /Ob1

```
_a$ = 8         ; size = 8
_b$ = 16        ; size = 8
_f3     PROC
        push    DWORD PTR _b$[esp]
        push    DWORD PTR _b$[esp]
        push    DWORD PTR _a$[esp+8]
        push    DWORD PTR _a$[esp+8]
        call    __allmul ; long long multiplication
        ret     0
_f3     ENDP

_a$ = 8         ; size = 8
_b$ = 16        ; size = 8
_f4     PROC
        push    DWORD PTR _b$[esp]
        push    DWORD PTR _b$[esp]
        push    DWORD PTR _a$[esp+8]
        push    DWORD PTR _a$[esp+8]
        call    __aulldiv ; unsigned long long division
        ret     0
_f4     ENDP

_a$ = 8         ; size = 8
_b$ = 16        ; size = 8
_f5     PROC
```

```
        push    DWORD PTR _b$[esp]
        push    DWORD PTR _b$[esp]
        push    DWORD PTR _a$[esp+8]
        push    DWORD PTR _a$[esp+8]
        call    __aullrem ; unsigned long long remainder
        ret     0
_f5     ENDP
```

Multiplication and division is more complex operation, so usually, the compiler embedds calls to the library functions doing that.

These functions meaning are here: E.

Listing 21.4: GCC 4.8.1 -O3 -fno-inline

```
_f3:
        push    ebx
        mov     edx, DWORD PTR [esp+8]
        mov     eax, DWORD PTR [esp+16]
        mov     ebx, DWORD PTR [esp+12]
        mov     ecx, DWORD PTR [esp+20]
        imul    ebx, eax
        imul    ecx, edx
        mul     edx
        add     ecx, ebx
        add     edx, ecx
        pop     ebx
        ret

_f4:
        sub     esp, 28
        mov     eax, DWORD PTR [esp+40]
        mov     edx, DWORD PTR [esp+44]
        mov     DWORD PTR [esp+8], eax
        mov     eax, DWORD PTR [esp+32]
        mov     DWORD PTR [esp+12], edx
        mov     edx, DWORD PTR [esp+36]
        mov     DWORD PTR [esp], eax
        mov     DWORD PTR [esp+4], edx
        call    ___udivdi3 ; unsigned division
        add     esp, 28
        ret

_f5:
        sub     esp, 28
```

```
        mov     eax, DWORD PTR [esp+40]
        mov     edx, DWORD PTR [esp+44]
        mov     DWORD PTR [esp+8], eax
        mov     eax, DWORD PTR [esp+32]
        mov     DWORD PTR [esp+12], edx
        mov     edx, DWORD PTR [esp+36]
        mov     DWORD PTR [esp], eax
        mov     DWORD PTR [esp+4], edx
        call    ___umoddi3 ; unsigned modulo
        add     esp, 28
        ret
```

GCC doing almost the same, but multiplication code is inlined right in the function, thinking it could be more efficient. GCC has different library function names: D.

21.3 Shifting right

```
#include <stdint.h>

uint64_t f6 (uint64_t a)
{
        return a>>7;
};
```

Listing 21.5: MSVC 2012 /Ox /Ob1

```
_a$ = 8         ; size = 8
_f6     PROC
        mov     eax, DWORD PTR _a$[esp-4]
        mov     edx, DWORD PTR _a$[esp]
        shrd    eax, edx, 7
        shr     edx, 7
        ret     0
_f6     ENDP
```

Listing 21.6: GCC 4.8.1 -O3 -fno-inline

```
_f6:
        mov     edx, DWORD PTR [esp+8]
        mov     eax, DWORD PTR [esp+4]
        shrd    eax, edx, 7
        shr     edx, 7
```

```
        ret
```

Shifting also occurring in two passes: first lower part is shifting, then higher part. But the lower part is shifting with the help of SHRD instruction, it shifting EDX value by 7 bits, but pulling new bits from EAX, i.e., from the higher part. Higher part is shifting using more popular SHR instruction: indeed, freed bits in the higher part should be just filled with zeroes.

21.4 Converting of 32-bit value into 64-bit one

```
#include <stdint.h>

int64_t f7 (int64_t a, int64_t b, int32_t c)
{
        return a*b+c;
};

int64_t f7_main ()
{
        return f7(12345678901234, 23456789012345, 12345);
};
```

Listing 21.7: MSVC 2012 /Ox /Ob1

```
_a$ = 8         ; size = 8
_b$ = 16        ; size = 8
_c$ = 24        ; size = 4
_f7     PROC
        push    esi
        push    DWORD PTR _b$[esp+4]
        push    DWORD PTR _b$[esp+4]
        push    DWORD PTR _a$[esp+12]
        push    DWORD PTR _a$[esp+12]
        call    __allmul ; long long multiplication
        mov     ecx, eax
        mov     eax, DWORD PTR _c$[esp]
        mov     esi, edx
        cdq     ; input: 32-bit value in EAX; output: 64-bit ↙
    ↳ value in EDX:EAX
        add     eax, ecx
        adc     edx, esi
        pop     esi
        ret     0
```

```
_f7     ENDP

_f7_main PROC
        push    12345           ; 00003039H
        push    5461            ; 00001555H
        push    1972608889      ; 75939f79H
        push    2874            ; 00000b3aH
        push    1942892530      ; 73ce2ff2H
        call    _f7
        add     esp, 20         ; 00000014H
        ret     0
_f7_main ENDP
```

Here we also run into necessity to extend 32-bit signed value from c into 64-bit signed. Unsigned values are converted straightforwardly: all bits in higher part should be set to 0. But it is not appropriate for signed data types: sign should be copied into higher part of resulting number. An instruction CDQ doing that here, it takes input value in EAX, extending value to 64-bit and leaving it in the EDX:EAX registers pair. In other words, CDQ instruction getting number sign in EAX (by getting just most significant bit in EAX), and depending of it, setting all 32-bits in EDX to 0 or 1. Its operation is somewhat similar to the MOVSX (13.1.1) instruction.

Listing 21.8: GCC 4.8.1 -O3 -fno-inline

```
_f7:
        push    edi
        push    esi
        push    ebx
        mov     esi, DWORD PTR [esp+16]
        mov     edi, DWORD PTR [esp+24]
        mov     ebx, DWORD PTR [esp+20]
        mov     ecx, DWORD PTR [esp+28]
        mov     eax, esi
        mul     edi
        imul    ebx, edi
        imul    ecx, esi
        mov     esi, edx
        add     ecx, ebx
        mov     ebx, eax
        mov     eax, DWORD PTR [esp+32]
        add     esi, ecx
        cdq     ; input: 32-bit value in EAX; output: 64-bit ↙
    ↳ value in EDX:EAX
        add     eax, ebx
        adc     edx, esi
```

```
        pop     ebx
        pop     esi
        pop     edi
        ret

_f7_main:
        sub     esp, 28
        mov     DWORD PTR [esp+16], 12345       ; 00003039H
        mov     DWORD PTR [esp+8], 1972608889   ; 75939f79H
        mov     DWORD PTR [esp+12], 5461        ; 00001555H
        mov     DWORD PTR [esp], 1942892530     ; 73ce2ff2H
        mov     DWORD PTR [esp+4], 2874         ; 00000b3aH
        call    _f7
        add     esp, 28
        ret
```

GCC generates just the same code as MSVC, but inlines multiplication code right in the function.

See also: 32-bit values in 16-bit environment: 31.4.

Chapter 22

SIMD

SIMD[1] is just acronym: *Single Instruction, Multiple Data*.

As it is said, it is multiple data processing using only one instruction.

Just as FPU, that CPU subsystem looks like separate processor inside x86.

SIMD began as MMX in x86. 8 new 64-bit registers appeared: MM0-MM7.

Each MMX register may hold 2 32-bit values, 4 16-bit values or 8 bytes. For example, it is possible to add 8 8-bit values (bytes) simultaneously by adding two values in MMX-registers.

One simple example is graphics editor, representing image as a two dimensional array. When user change image brightness, the editor must add a coefficient to each pixel value, or to subtract. For the sake of brevity, our image may be grayscale and each pixel defined by one 8-bit byte, then it is possible to change brightness of 8 pixels simultaneously.

When MMX appeared, these registers was actually located in FPU registers. It was possible to use either FPU or MMX at the same time. One might think, Intel saved on transistors, but in fact, the reason of such symbiosis is simpler —older OS may not aware of additional CPU registers would not save them at the context switching, but will save FPU registers. Thus, MMX-enabled CPU + old OS + process utilizing MMX features = that all will work together.

SSE—is extension of SIMD registers up to 128 bits, now separately from FPU.

AVX—another extension to 256 bits.

Now about practical usage.

Of course, memory copying (`memcpy`), memory comparing (`memcmp`) and so on.

One more example: we got DES encryption algorithm, it takes 64-bit block, 56-bit key, encrypt block and produce 64-bit result. DES algorithm may be considered as a very large electronic circuit, with wires and AND/OR/NOT gates.

Bitslice DES[2] —is an idea of processing group of blocks and keys simultaneously.

[1] Single instruction, multiple data

[2] http://www.darkside.com.au/bitslice/

Let's say, variable of type *unsigned int* on x86 may hold up to 32 bits, so, it is possible to store there intermediate results for 32 blocks-keys pairs simultaneously, using 64+56 variables of *unsigned int* type.

I wrote an utility to brute-force Oracle RDBMS passwords/hashes (ones based on DES), slightly modified bitslice DES algorithm for SSE2 and AVX —now it is possible to encrypt 128 or 256 block-keys pairs simultaneously.

http://conus.info/utils/ops_SIMD/

22.1 Vectorization

Vectorization[3], for example, is when you have a loop taking couple of arrays at input and produces one array. Loop body takes values from input arrays, do something and put result into output array. It is important that there is only one single operation applied to each element. Vectorization —is to process several elements simultaneously.

Vectorization is not very fresh technology: author of this textbook saw it at least on Cray Y-MP supercomputer line from 1988 when played with its "lite" version Cray Y-MP EL [4].

For example:

```
for (i = 0; i < 1024; i++)
{
    C[i] = A[i]*B[i];
}
```

This fragment of code takes elements from A and B, multiplies them and save result into C.

If each array element we have is 32-bit *int*, then it is possible to load 4 elements from A into 128-bit XMM-register, from B to another XMM-registers, and by executing *PMULLD* (*Multiply Packed Signed Dword Integers and Store Low Result*) and `PMULHW` (*Multiply Packed Signed Integers and Store High Result*), it is possible to get 4 64-bit products at once.

Thus, loop body count is $1024/4$ instead of 1024, that is 4 times less and, of course, faster.

Some compilers can do vectorization automatically in a simple cases, e.g., Intel C++[5].

I wrote tiny function:

```
int f (int sz, int *ar1, int *ar2, int *ar3)
{
        for (int i=0; i<sz; i++)
                ar3[i]=ar1[i]+ar2[i];
```

[3] Wikipedia: vectorization

[4] Remotely. It is installed in the museum of supercomputers: http://www.cray-cyber.org

[5] More about Intel C++ automatic vectorization: Excerpt: Effective Automatic Vectorization

```
        return 0;
};
```

22.1.1 Intel C++

Let's compile it with Intel C++ 11.1.051 win32:

```
icl intel.cpp /QaxSSE2 /Faintel.asm /Ox
```

We got (in IDA):

```
; int __cdecl f(int, int *, int *, int *)
                public ?f@@YAHHPAH00@Z
?f@@YAHHPAH00@Z proc near

var_10 = dword ptr -10h
sz     = dword ptr  4
ar1    = dword ptr  8
ar2    = dword ptr  0Ch
ar3    = dword ptr  10h

       push    edi
       push    esi
       push    ebx
       push    esi
       mov     edx, [esp+10h+sz]
       test    edx, edx
       jle     loc_15B
       mov     eax, [esp+10h+ar3]
       cmp     edx, 6
       jle     loc_143
       cmp     eax, [esp+10h+ar2]
       jbe     short loc_36
       mov     esi, [esp+10h+ar2]
       sub     esi, eax
       lea     ecx, ds:0[edx*4]
       neg     esi
       cmp     ecx, esi
       jbe     short loc_55

loc_36: ; CODE XREF: f(int,int *,int *,int *)+21
       cmp     eax, [esp+10h+ar2]
       jnb     loc_143
```

```
        mov     esi, [esp+10h+ar2]
        sub     esi, eax
        lea     ecx, ds:0[edx*4]
        cmp     esi, ecx
        jb      loc_143

loc_55: ; CODE XREF: f(int,int *,int *,int *)+34
        cmp     eax, [esp+10h+ar1]
        jbe     short loc_67
        mov     esi, [esp+10h+ar1]
        sub     esi, eax
        neg     esi
        cmp     ecx, esi
        jbe     short loc_7F

loc_67: ; CODE XREF: f(int,int *,int *,int *)+59
        cmp     eax, [esp+10h+ar1]
        jnb     loc_143
        mov     esi, [esp+10h+ar1]
        sub     esi, eax
        cmp     esi, ecx
        jb      loc_143

loc_7F: ; CODE XREF: f(int,int *,int *,int *)+65
        mov     edi, eax        ; edi = ar1
        and     edi, 0Fh        ; is ar1 16-byte aligned?
        jz      short loc_9A    ; yes
        test    edi, 3
        jnz     loc_162
        neg     edi
        add     edi, 10h
        shr     edi, 2

loc_9A: ; CODE XREF: f(int,int *,int *,int *)+84
        lea     ecx, [edi+4]
        cmp     edx, ecx
        jl      loc_162
        mov     ecx, edx
        sub     ecx, edi
        and     ecx, 3
        neg     ecx
        add     ecx, edx
        test    edi, edi
```

```
        jbe     short loc_D6
        mov     ebx, [esp+10h+ar2]
        mov     [esp+10h+var_10], ecx
        mov     ecx, [esp+10h+ar1]
        xor     esi, esi

loc_C1: ; CODE XREF: f(int,int *,int *,int *)+CD
        mov     edx, [ecx+esi*4]
        add     edx, [ebx+esi*4]
        mov     [eax+esi*4], edx
        inc     esi
        cmp     esi, edi
        jb      short loc_C1
        mov     ecx, [esp+10h+var_10]
        mov     edx, [esp+10h+sz]

loc_D6: ; CODE XREF: f(int,int *,int *,int *)+B2
        mov     esi, [esp+10h+ar2]
        lea     esi, [esi+edi*4] ; is ar2+i*4 16-byte aligned?
        test    esi, 0Fh
        jz      short loc_109   ; yes!
        mov     ebx, [esp+10h+ar1]
        mov     esi, [esp+10h+ar2]

loc_ED: ; CODE XREF: f(int,int *,int *,int *)+105
        movdqu  xmm1, xmmword ptr [ebx+edi*4]
        movdqu  xmm0, xmmword ptr [esi+edi*4] ; ar2+i*4 is not ↙
    ↳ 16-byte aligned, so load it to xmm0
        paddd   xmm1, xmm0
        movdqa  xmmword ptr [eax+edi*4], xmm1
        add     edi, 4
        cmp     edi, ecx
        jb      short loc_ED
        jmp     short loc_127

loc_109: ; CODE XREF: f(int,int *,int *,int *)+E3
        mov     ebx, [esp+10h+ar1]
        mov     esi, [esp+10h+ar2]

loc_111: ; CODE XREF: f(int,int *,int *,int *)+125
        movdqu  xmm0, xmmword ptr [ebx+edi*4]
        paddd   xmm0, xmmword ptr [esi+edi*4]
        movdqa  xmmword ptr [eax+edi*4], xmm0
```

```
        add     edi, 4
        cmp     edi, ecx
        jb      short loc_111

loc_127: ; CODE XREF: f(int,int *,int *,int *)+107
         ; f(int,int *,int *,int *)+164
        cmp     ecx, edx
        jnb     short loc_15B
        mov     esi, [esp+10h+ar1]
        mov     edi, [esp+10h+ar2]

loc_133: ; CODE XREF: f(int,int *,int *,int *)+13F
        mov     ebx, [esi+ecx*4]
        add     ebx, [edi+ecx*4]
        mov     [eax+ecx*4], ebx
        inc     ecx
        cmp     ecx, edx
        jb      short loc_133
        jmp     short loc_15B

loc_143: ; CODE XREF: f(int,int *,int *,int *)+17
         ; f(int,int *,int *,int *)+3A ...
        mov     esi, [esp+10h+ar1]
        mov     edi, [esp+10h+ar2]
        xor     ecx, ecx

loc_14D: ; CODE XREF: f(int,int *,int *,int *)+159
        mov     ebx, [esi+ecx*4]
        add     ebx, [edi+ecx*4]
        mov     [eax+ecx*4], ebx
        inc     ecx
        cmp     ecx, edx
        jb      short loc_14D

loc_15B: ; CODE XREF: f(int,int *,int *,int *)+A
         ; f(int,int *,int *,int *)+129 ...
        xor     eax, eax
        pop     ecx
        pop     ebx
        pop     esi
        pop     edi
        retn
```

```
loc_162: ; CODE XREF: f(int,int *,int *,int *)+8C
         ; f(int,int *,int *,int *)+9F
       xor     ecx, ecx
       jmp     short loc_127
?f@@YAHHPAH00@Z endp
```

SSE2-related instructions are:

- MOVDQU (*Move Unaligned Double Quadword*)—it just load 16 bytes from memory into a XMM-register.
- PADDD (*Add Packed Integers*)—adding 4 pairs of 32-bit numbers and leaving result in first operand. By the way, no exception raised in case of overflow and no flags will be set, just low 32-bit of result will be stored. If one of PADDD operands is address of value in memory, then address must be aligned on a 16-byte boundary. If it is not aligned, exception will be occurred [6].
- MOVDQA (*Move Aligned Double Quadword*)—the same as MOVDQU, but requires address of value in memory to be aligned on a 16-bit border. If it is not aligned, exception will be raised. MOVDQA works faster than MOVDQU, but requires aforesaid.

So, these SSE2-instructions will be executed only in case if there are more 4 pairs to work on plus pointer ar3 is aligned on a 16-byte boundary.

More than that, if ar2 is aligned on a 16-byte boundary as well, this fragment of code will be executed:

```
movdqu  xmm0, xmmword ptr [ebx+edi*4] ; ar1+i*4
paddd   xmm0, xmmword ptr [esi+edi*4] ; ar2+i*4
movdqa  xmmword ptr [eax+edi*4], xmm0 ; ar3+i*4
```

Otherwise, value from ar2 will be loaded into XMM0 using MOVDQU, it does not require aligned pointer, but may work slower:

```
movdqu  xmm1, xmmword ptr [ebx+edi*4] ; ar1+i*4
movdqu  xmm0, xmmword ptr [esi+edi*4] ; ar2+i*4 is not 16-byte ↙
    ↳ aligned, so load it to xmm0
paddd   xmm1, xmm0
movdqa  xmmword ptr [eax+edi*4], xmm1 ; ar3+i*4
```

In all other cases, non-SSE2 code will be executed.

[6] More about data aligning: Wikipedia: Data structure alignment

22.1.2 GCC

GCC may also vectorize in a simple cases[7], if to use -O3 option and to turn on SSE2 support: -msse2.

What we got (GCC 4.4.1):

```
; f(int, int *, int *, int *)
            public _Z1fiPiS_S_
_Z1fiPiS_S_ proc near

var_18      = dword ptr -18h
var_14      = dword ptr -14h
var_10      = dword ptr -10h
arg_0       = dword ptr  8
arg_4       = dword ptr  0Ch
arg_8       = dword ptr  10h
arg_C       = dword ptr  14h

            push    ebp
            mov     ebp, esp
            push    edi
            push    esi
            push    ebx
            sub     esp, 0Ch
            mov     ecx, [ebp+arg_0]
            mov     esi, [ebp+arg_4]
            mov     edi, [ebp+arg_8]
            mov     ebx, [ebp+arg_C]
            test    ecx, ecx
            jle     short loc_80484D8
            cmp     ecx, 6
            lea     eax, [ebx+10h]
            ja      short loc_80484E8

loc_80484C1: ; CODE XREF: f(int,int *,int *,int *)+4B
             ; f(int,int *,int *,int *)+61 ...
            xor     eax, eax
            nop
            lea     esi, [esi+0]

loc_80484C8: ; CODE XREF: f(int,int *,int *,int *)+36
            mov     edx, [edi+eax*4]
```

[7] More about GCC vectorization support: http://gcc.gnu.org/projects/tree-ssa/vectorization.html

```
            add     edx, [esi+eax*4]
            mov     [ebx+eax*4], edx
            add     eax, 1
            cmp     eax, ecx
            jnz     short loc_80484C8

loc_80484D8: ; CODE XREF: f(int,int *,int *,int *)+17
             ; f(int,int *,int *,int *)+A5
            add     esp, 0Ch
            xor     eax, eax
            pop     ebx
            pop     esi
            pop     edi
            pop     ebp
            retn

            align 8

loc_80484E8: ; CODE XREF: f(int,int *,int *,int *)+1F
            test    bl, 0Fh
            jnz     short loc_80484C1
            lea     edx, [esi+10h]
            cmp     ebx, edx
            jbe     loc_8048578

loc_80484F8: ; CODE XREF: f(int,int *,int *,int *)+E0
            lea     edx, [edi+10h]
            cmp     ebx, edx
            ja      short loc_8048503
            cmp     edi, eax
            jbe     short loc_80484C1

loc_8048503: ; CODE XREF: f(int,int *,int *,int *)+5D
            mov     eax, ecx
            shr     eax, 2
            mov     [ebp+var_14], eax
            shl     eax, 2
            test    eax, eax
            mov     [ebp+var_10], eax
            jz      short loc_8048547
            mov     [ebp+var_18], ecx
            mov     ecx, [ebp+var_14]
            xor     eax, eax
```

```
            xor     edx, edx
            nop

loc_8048520: ; CODE XREF: f(int,int *,int *,int *)+9B
            movdqu  xmm1, xmmword ptr [edi+eax]
            movdqu  xmm0, xmmword ptr [esi+eax]
            add     edx, 1
            paddd   xmm0, xmm1
            movdqa  xmmword ptr [ebx+eax], xmm0
            add     eax, 10h
            cmp     edx, ecx
            jb      short loc_8048520
            mov     ecx, [ebp+var_18]
            mov     eax, [ebp+var_10]
            cmp     ecx, eax
            jz      short loc_80484D8

loc_8048547: ; CODE XREF: f(int,int *,int *,int *)+73
            lea     edx, ds:0[eax*4]
            add     esi, edx
            add     edi, edx
            add     ebx, edx
            lea     esi, [esi+0]

loc_8048558: ; CODE XREF: f(int,int *,int *,int *)+CC
            mov     edx, [edi]
            add     eax, 1
            add     edi, 4
            add     edx, [esi]
            add     esi, 4
            mov     [ebx], edx
            add     ebx, 4
            cmp     ecx, eax
            jg      short loc_8048558
            add     esp, 0Ch
            xor     eax, eax
            pop     ebx
            pop     esi
            pop     edi
            pop     ebp
            retn

loc_8048578: ; CODE XREF: f(int,int *,int *,int *)+52
```

```
            cmp     eax, esi
            jnb     loc_80484C1
            jmp     loc_80484F8
_Z1fiPiS_S_ endp
```

Almost the same, however, not as meticulously as Intel C++ doing it.

22.2 SIMD `strlen()` implementation

It should be noted the SIMD-instructions may be inserted into C/C++ code via special macros[8]. As of MSVC, some of them are located in the `intrin.h` file.

It is possible to implement `strlen()` function[9] using SIMD-instructions, working 2-2.5 times faster than common implementation. This function will load 16 characters into a XMM-register and check each against zero.

```
size_t strlen_sse2(const char *str)
{
    register size_t len = 0;
    const char *s=str;
    bool str_is_aligned=(((unsigned int)str)&0xFFFFFFF0) == (↙
    ↳ unsigned int)str;

    if (str_is_aligned==false)
        return strlen (str);

    __m128i xmm0 = _mm_setzero_si128();
    __m128i xmm1;
    int mask = 0;

    for (;;)
    {
        xmm1 = _mm_load_si128((__m128i *)s);
        xmm1 = _mm_cmpeq_epi8(xmm1, xmm0);
        if ((mask = _mm_movemask_epi8(xmm1)) != 0)
        {
            unsigned long pos;
            _BitScanForward(&pos, mask);
            len += (size_t)pos;

            break;
        }
```

[8] MSDN: MMX, SSE, and SSE2 Intrinsics

[9] strlen() —standard C library function for calculating string length

```
        s += sizeof(__m128i);
        len += sizeof(__m128i);
    };

    return len;
}
```

(the example is based on source code from there).

Let's compile in MSVC 2010 with /Ox option:

```
_pos$75552 = -4          ; size = 4
_str$ = 8                ; size = 4
?strlen_sse2@@YAIPBD@Z PROC ; strlen_sse2

    push    ebp
    mov     ebp, esp
    and     esp, -16             ; fffffff0H
    mov     eax, DWORD PTR _str$[ebp]
    sub     esp, 12              ; 0000000cH
    push    esi
    mov     esi, eax
    and     esi, -16             ; fffffff0H
    xor     edx, edx
    mov     ecx, eax
    cmp     esi, eax
    je      SHORT $LN4@strlen_sse
    lea     edx, DWORD PTR [eax+1]
    npad    3
$LL11@strlen_sse:
    mov     cl, BYTE PTR [eax]
    inc     eax
    test    cl, cl
    jne     SHORT $LL11@strlen_sse
    sub     eax, edx
    pop     esi
    mov     esp, ebp
    pop     ebp
    ret     0
$LN4@strlen_sse:
    movdqa  xmm1, XMMWORD PTR [eax]
    pxor    xmm0, xmm0
    pcmpeqb xmm1, xmm0
    pmovmskb eax, xmm1
    test    eax, eax
```

```
    jne     SHORT $LN9@strlen_sse
$LL3@strlen_sse:
    movdqa  xmm1, XMMWORD PTR [ecx+16]
    add     ecx, 16                     ; 00000010H
    pcmpeqb xmm1, xmm0
    add     edx, 16                     ; 00000010H
    pmovmskb eax, xmm1
    test    eax, eax
    je      SHORT $LL3@strlen_sse
$LN9@strlen_sse:
    bsf     eax, eax
    mov     ecx, eax
    mov     DWORD PTR _pos$75552[esp+16], eax
    lea     eax, DWORD PTR [ecx+edx]
    pop     esi
    mov     esp, ebp
    pop     ebp
    ret     0
?strlen_sse2@@YAIPBD@Z ENDP                 ; strlen_sse2
```

First of all, we check `str` pointer, if it is aligned on a 16-byte boundary. If not, let's call generic `strlen()` implementation.

Then, load next 16 bytes into the `XMM1` register using `MOVDQA` instruction.

Observant reader might ask, why `MOVDQU` cannot be used here since it can load data from the memory regardless the fact if the pointer aligned or not.

Yes, it might be done in this way: if pointer is aligned, load data using `MOVDQA`, if not —use slower `MOVDQU`.

But here we are may stick into hard to notice caveat:

In Windows NT line of OS but not limited to it, memory allocated by pages of 4 KiB (4096 bytes). Each win32-process has ostensibly 4 GiB, but in fact, only some parts of address space are connected to real physical memory. If the process accessing to the absent memory block, exception will be raised. That's how virtual memory works[10].

So, a function loading 16 bytes at once, may step over a border of allocated memory block. Let's consider, OS allocated 8192 (0x2000) bytes at the address 0x008c0000. Thus, the block is the bytes starting from address 0x008c0000 to 0x008c1fff inclusive.

After the block, that is, starting from address 0x008c2000 there is nothing at all, e.g., OS not allocated any memory there. Attempt to access a memory starting from the address will raise exception.

And let's consider, the program holding a string containing 5 characters almost at the end of block, and that is not a crime.

[10] http://en.wikipedia.org/wiki/Page_(computer_memory)

0x008c1ff8	'h'
0x008c1ff9	'e'
0x008c1ffa	'l'
0x008c1ffb	'l'
0x008c1ffc	'o'
0x008c1ffd	'\x00'
0x008c1ffe	random noise
0x008c1fff	random noise

So, in common conditions the program calling `strlen()` passing it a pointer to string `'hello'` lying in memory at address 0x008c1ff8. `strlen()` will read one byte at a time until 0x008c1ffd, where zero-byte, and so here it will stop working.

Now if we implement our own `strlen()` reading 16 byte at once, starting at any address, will it be aligned or not, `MOVDQU` may attempt to load 16 bytes at once at address 0x008c1ff8 up to 0x008c2008, and then exception will be raised. That's the situation to be avoided, of course.

So then we'll work only with the addresses aligned on a 16 byte boundary, what in combination with a knowledge of OS page size is usually aligned on a 16-byte boundary too, give us some warranty our function will not read from unallocated memory.

Let's back to our function.

`_mm_setzero_si128()`—is a macro generating `pxor xmm0, xmm0` —instruction just clears the `XMM0` register

`_mm_load_si128()`—is a macro for `MOVDQA`, it just loading 16 bytes from the address in the `XMM1` register.

`_mm_cmpeq_epi8()`—is a macro for `PCMPEQB`, is an instruction comparing two XMM-registers bytewise.

And if some byte was equals to other, there will be `0xff` at this point in the result or 0 if otherwise.

For example.

```
XMM1: 11223344556677880000000000000000
XMM0: 11ab3444007877881111111111111111
```

After `pcmpeqb xmm1, xmm0` execution, the `XMM1` register shall contain:

```
XMM1: ff0000ff0000ffff0000000000000000
```

In our case, this instruction comparing each 16-byte block with the block of 16 zero-bytes, was set in the `XMM0` register by `pxor xmm0, xmm0`.

The next macro is `_mm_movemask_epi8()` —that is `PMOVMSKB` instruction.

It is very useful if to use it with `PCMPEQB`.

`pmovmskb eax, xmm1`

This instruction will set first `EAX` bit into 1 if most significant bit of the first byte in the `XMM1` is 1. In other words, if first byte of the `XMM1` register is `0xff`, first `EAX` bit will be set to 1 too.

If second byte in the XMM1 register is 0xff, then second EAX bit will be set to 1 too. In other words, the instruction is answer to the question *which bytes in the XMM1 are 0xff?* And will prepare 16 bits in the EAX register. Other bits in the EAX register are to be cleared.

By the way, do not forget about this feature of our algorithm:

There might be 16 bytes on input like hello\x00garbage\x00ab

It is a 'hello' string, terminating zero, and also a random noise in memory.

If we load these 16 bytes into XMM1 and compare them with zeroed XMM0, we will get something like (I use here order from MSB[11] to LSB[12]):

```
XMM1: 0000ff00000000000000ff0000000000
```

This means, the instruction found two zero bytes, and that is not surprising.

PMOVMSKB in our case will prepare EAX like (in binary representation): *0010000000100000b*.

Obviously, our function must consider only first zero bit and ignore the rest ones.

The next instruction—BSF (*Bit Scan Forward*). This instruction find first bit set to 1 and stores its position into first operand.

```
EAX=0010000000100000b
```

After bsf eax, eax instruction execution, EAX will contain 5, this means, 1 found at 5th bit position (starting from zero).

MSVC has a macro for this instruction: _BitScanForward.

Now it is simple. If zero byte found, its position added to what we already counted and now we have ready to return result.

Almost all.

By the way, it is also should be noted, MSVC compiler emitted two loop bodies side by side, for optimization.

By the way, SSE 4.2 (appeared in Intel Core i7) offers more instructions where these string manipulations might be even easier: http://www.strchr.com/strcmp_and_strlen_using_sse_4.2

[11] most significant bit

[12] least significant bit

Chapter 23

64 bits

23.1 x86-64

It is a 64-bit extension to x86-architecture.

From the reverse engineer's perspective, most important differences are:

- Almost all registers (except FPU and SIMD) are extended to 64 bits and got r- prefix. 8 additional registers added. Now GPR's are: RAX, RBX, RCX, RDX, RBP, RSP, RSI, RDI, R8, R9, R10, R11, R12, R13, R14, R15.

  It is still possible to access to *older* register parts as usual. For example, it is possible to access lower 32-bit part of the RAX register using EAX.

  New r8-r15 registers also has its *lower parts*: r8d-r15d (lower 32-bit parts), r8w-r15w (lower 16-bit parts), r8b-r15b (lower 8-bit parts).

  SIMD-registers number are doubled: from 8 to 16: XMM0-XMM15.

- In Win64, function calling convention is slightly different, somewhat resembling fastcall (44.3). First 4 arguments stored in the RCX, RDX, R8, R9 registers, others —in the stack. Caller function must also allocate 32 bytes so the callee may save there 4 first arguments and use these registers for own needs. Short functions may use arguments just from registers, but larger may save their values on the stack.

  System V AMD64 ABI (Linux, *BSD, Mac OS X) [21] also somewhat resembling fastcall, it uses 6 registers RDI, RSI, RDX, RCX, R8, R9 for the first 6 arguments. All the rest are passed in the stack.

  See also section about calling conventions (44).

- C *int* type is still 32-bit for compatibility.

- All pointers are 64-bit now.

This provokes irritation sometimes: now one need twice as much memory for storing pointers, including, cache memory, despite the fact x64 CPUs addresses only 48 bits of external RAM.

Since now registers number are doubled, compilers has more space now for maneuvering calling register allocation. What it meanings for us, emitted code will contain less local variables.

For example, function calculating first S-box of DES encryption algorithm, it processing 32/64/128/256 values at once (depending on `DES_type` type (uint32, uint64, SSE2 or AVX)) using bitslice DES method (read more about this technique here (22)):

```
/*
 * Generated S-box files.
 *
 * This software may be modified, redistributed, and used for ↙
    ↳ any purpose,
 * so long as its origin is acknowledged.
 *
 * Produced by Matthew Kwan - March 1998
 */

#ifdef _WIN64
#define DES_type unsigned __int64
#else
#define DES_type unsigned int
#endif

void
s1 (
    DES_type    a1,
    DES_type    a2,
    DES_type    a3,
    DES_type    a4,
    DES_type    a5,
    DES_type    a6,
    DES_type    *out1,
    DES_type    *out2,
    DES_type    *out3,
    DES_type    *out4
) {
    DES_type    x1, x2, x3, x4, x5, x6, x7, x8;
    DES_type    x9, x10, x11, x12, x13, x14, x15, x16;
    DES_type    x17, x18, x19, x20, x21, x22, x23, x24;
    DES_type    x25, x26, x27, x28, x29, x30, x31, x32;
    DES_type    x33, x34, x35, x36, x37, x38, x39, x40;
```

```
    DES_type    x41, x42, x43, x44, x45, x46, x47, x48;
    DES_type    x49, x50, x51, x52, x53, x54, x55, x56;

    x1 = a3 & ~a5;
    x2 = x1 ^ a4;
    x3 = a3 & ~a4;
    x4 = x3 | a5;
    x5 = a6 & x4;
    x6 = x2 ^ x5;
    x7 = a4 & ~a5;
    x8 = a3 ^ a4;
    x9 = a6 & ~x8;
    x10 = x7 ^ x9;
    x11 = a2 | x10;
    x12 = x6 ^ x11;
    x13 = a5 ^ x5;
    x14 = x13 & x8;
    x15 = a5 & ~a4;
    x16 = x3 ^ x14;
    x17 = a6 | x16;
    x18 = x15 ^ x17;
    x19 = a2 | x18;
    x20 = x14 ^ x19;
    x21 = a1 & x20;
    x22 = x12 ^ ~x21;
    *out2 ^= x22;
    x23 = x1 | x5;
    x24 = x23 ^ x8;
    x25 = x18 & ~x2;
    x26 = a2 & ~x25;
    x27 = x24 ^ x26;
    x28 = x6 | x7;
    x29 = x28 ^ x25;
    x30 = x9 ^ x24;
    x31 = x18 & ~x30;
    x32 = a2 & x31;
    x33 = x29 ^ x32;
    x34 = a1 & x33;
    x35 = x27 ^ x34;
    *out4 ^= x35;
    x36 = a3 & x28;
    x37 = x18 & ~x36;
    x38 = a2 | x3;
```

```
    x39 = x37 ^ x38;
    x40 = a3 | x31;
    x41 = x24 & ~x37;
    x42 = x41 | x3;
    x43 = x42 & ~a2;
    x44 = x40 ^ x43;
    x45 = a1 & ~x44;
    x46 = x39 ^ ~x45;
    *out1 ^= x46;
    x47 = x33 & ~x9;
    x48 = x47 ^ x39;
    x49 = x4 ^ x36;
    x50 = x49 & ~x5;
    x51 = x42 | x18;
    x52 = x51 ^ a5;
    x53 = a2 & ~x52;
    x54 = x50 ^ x53;
    x55 = a1 | x54;
    x56 = x48 ^ ~x55;
    *out3 ^= x56;
}
```

There is a lot of local variables. Of course, not all those will be in local stack. Let's compile it with MSVC 2008 with `/Ox` option:

Listing 23.1: Optimizing MSVC 2008

```
PUBLIC    _s1
; Function compile flags: /Ogtpy
_TEXT    SEGMENT
_x6$ = -20          ; size = 4
_x3$ = -16          ; size = 4
_x1$ = -12          ; size = 4
_x8$ = -8           ; size = 4
_x4$ = -4           ; size = 4
_a1$ = 8            ; size = 4
_a2$ = 12           ; size = 4
_a3$ = 16           ; size = 4
_x33$ = 20          ; size = 4
_x7$ = 20           ; size = 4
_a4$ = 20           ; size = 4
_a5$ = 24           ; size = 4
tv326 = 28          ; size = 4
_x36$ = 28          ; size = 4
_x28$ = 28          ; size = 4
```

```
_a6$ = 28               ; size = 4
_out1$ = 32             ; size = 4
_x24$ = 36              ; size = 4
_out2$ = 36             ; size = 4
_out3$ = 40             ; size = 4
_out4$ = 44             ; size = 4
_s1    PROC
    sub    esp, 20                      ; 00000014H
    mov    edx, DWORD PTR _a5$[esp+16]
    push   ebx
    mov    ebx, DWORD PTR _a4$[esp+20]
    push   ebp
    push   esi
    mov    esi, DWORD PTR _a3$[esp+28]
    push   edi
    mov    edi, ebx
    not    edi
    mov    ebp, edi
    and    edi, DWORD PTR _a5$[esp+32]
    mov    ecx, edx
    not    ecx
    and    ebp, esi
    mov    eax, ecx
    and    eax, esi
    and    ecx, ebx
    mov    DWORD PTR _x1$[esp+36], eax
    xor    eax, ebx
    mov    esi, ebp
    or     esi, edx
    mov    DWORD PTR _x4$[esp+36], esi
    and    esi, DWORD PTR _a6$[esp+32]
    mov    DWORD PTR _x7$[esp+32], ecx
    mov    edx, esi
    xor    edx, eax
    mov    DWORD PTR _x6$[esp+36], edx
    mov    edx, DWORD PTR _a3$[esp+32]
    xor    edx, ebx
    mov    ebx, esi
    xor    ebx, DWORD PTR _a5$[esp+32]
    mov    DWORD PTR _x8$[esp+36], edx
    and    ebx, edx
    mov    ecx, edx
    mov    edx, ebx
```

```
    xor     edx, ebp
    or      edx, DWORD PTR _a6$[esp+32]
    not     ecx
    and     ecx, DWORD PTR _a6$[esp+32]
    xor     edx, edi
    mov     edi, edx
    or      edi, DWORD PTR _a2$[esp+32]
    mov     DWORD PTR _x3$[esp+36], ebp
    mov     ebp, DWORD PTR _a2$[esp+32]
    xor     edi, ebx
    and     edi, DWORD PTR _a1$[esp+32]
    mov     ebx, ecx
    xor     ebx, DWORD PTR _x7$[esp+32]
    not     edi
    or      ebx, ebp
    xor     edi, ebx
    mov     ebx, edi
    mov     edi, DWORD PTR _out2$[esp+32]
    xor     ebx, DWORD PTR [edi]
    not     eax
    xor     ebx, DWORD PTR _x6$[esp+36]
    and     eax, edx
    mov     DWORD PTR [edi], ebx
    mov     ebx, DWORD PTR _x7$[esp+32]
    or      ebx, DWORD PTR _x6$[esp+36]
    mov     edi, esi
    or      edi, DWORD PTR _x1$[esp+36]
    mov     DWORD PTR _x28$[esp+32], ebx
    xor     edi, DWORD PTR _x8$[esp+36]
    mov     DWORD PTR _x24$[esp+32], edi
    xor     edi, ecx
    not     edi
    and     edi, edx
    mov     ebx, edi
    and     ebx, ebp
    xor     ebx, DWORD PTR _x28$[esp+32]
    xor     ebx, eax
    not     eax
    mov     DWORD PTR _x33$[esp+32], ebx
    and     ebx, DWORD PTR _a1$[esp+32]
    and     eax, ebp
    xor     eax, ebx
    mov     ebx, DWORD PTR _out4$[esp+32]
```

```
    xor     eax, DWORD PTR [ebx]
    xor     eax, DWORD PTR _x24$[esp+32]
    mov     DWORD PTR [ebx], eax
    mov     eax, DWORD PTR _x28$[esp+32]
    and     eax, DWORD PTR _a3$[esp+32]
    mov     ebx, DWORD PTR _x3$[esp+36]
    or      edi, DWORD PTR _a3$[esp+32]
    mov     DWORD PTR _x36$[esp+32], eax
    not     eax
    and     eax, edx
    or      ebx, ebp
    xor     ebx, eax
    not     eax
    and     eax, DWORD PTR _x24$[esp+32]
    not     ebp
    or      eax, DWORD PTR _x3$[esp+36]
    not     esi
    and     ebp, eax
    or      eax, edx
    xor     eax, DWORD PTR _a5$[esp+32]
    mov     edx, DWORD PTR _x36$[esp+32]
    xor     edx, DWORD PTR _x4$[esp+36]
    xor     ebp, edi
    mov     edi, DWORD PTR _out1$[esp+32]
    not     eax
    and     eax, DWORD PTR _a2$[esp+32]
    not     ebp
    and     ebp, DWORD PTR _a1$[esp+32]
    and     edx, esi
    xor     eax, edx
    or      eax, DWORD PTR _a1$[esp+32]
    not     ebp
    xor     ebp, DWORD PTR [edi]
    not     ecx
    and     ecx, DWORD PTR _x33$[esp+32]
    xor     ebp, ebx
    not     eax
    mov     DWORD PTR [edi], ebp
    xor     eax, ecx
    mov     ecx, DWORD PTR _out3$[esp+32]
    xor     eax, DWORD PTR [ecx]
    pop     edi
    pop     esi
```

```
    xor     eax, ebx
    pop     ebp
    mov     DWORD PTR [ecx], eax
    pop     ebx
    add     esp, 20                     ; 00000014H
    ret     0
_s1    ENDP
```

5 variables was allocated in local stack by compiler.
Now let's try the same thing in 64-bit version of MSVC 2008:

Listing 23.2: Optimizing MSVC 2008

```
a1$ = 56
a2$ = 64
a3$ = 72
a4$ = 80
x36$1$ = 88
a5$ = 88
a6$ = 96
out1$ = 104
out2$ = 112
out3$ = 120
out4$ = 128
s1    PROC
$LN3:
    mov     QWORD PTR [rsp+24], rbx
    mov     QWORD PTR [rsp+32], rbp
    mov     QWORD PTR [rsp+16], rdx
    mov     QWORD PTR [rsp+8], rcx
    push    rsi
    push    rdi
    push    r12
    push    r13
    push    r14
    push    r15
    mov     r15, QWORD PTR a5$[rsp]
    mov     rcx, QWORD PTR a6$[rsp]
    mov     rbp, r8
    mov     r10, r9
    mov     rax, r15
    mov     rdx, rbp
    not     rax
    xor     rdx, r9
    not     r10
```

```
    mov    r11, rax
    and    rax, r9
    mov    rsi, r10
    mov    QWORD PTR x36$1$[rsp], rax
    and    r11, r8
    and    rsi, r8
    and    r10, r15
    mov    r13, rdx
    mov    rbx, r11
    xor    rbx, r9
    mov    r9, QWORD PTR a2$[rsp]
    mov    r12, rsi
    or     r12, r15
    not    r13
    and    r13, rcx
    mov    r14, r12
    and    r14, rcx
    mov    rax, r14
    mov    r8, r14
    xor    r8, rbx
    xor    rax, r15
    not    rbx
    and    rax, rdx
    mov    rdi, rax
    xor    rdi, rsi
    or     rdi, rcx
    xor    rdi, r10
    and    rbx, rdi
    mov    rcx, rdi
    or     rcx, r9
    xor    rcx, rax
    mov    rax, r13
    xor    rax, QWORD PTR x36$1$[rsp]
    and    rcx, QWORD PTR a1$[rsp]
    or     rax, r9
    not    rcx
    xor    rcx, rax
    mov    rax, QWORD PTR out2$[rsp]
    xor    rcx, QWORD PTR [rax]
    xor    rcx, r8
    mov    QWORD PTR [rax], rcx
    mov    rax, QWORD PTR x36$1$[rsp]
    mov    rcx, r14
```

```
    or      rax, r8
    or      rcx, r11
    mov     r11, r9
    xor     rcx, rdx
    mov     QWORD PTR x36$1$[rsp], rax
    mov     r8, rsi
    mov     rdx, rcx
    xor     rdx, r13
    not     rdx
    and     rdx, rdi
    mov     r10, rdx
    and     r10, r9
    xor     r10, rax
    xor     r10, rbx
    not     rbx
    and     rbx, r9
    mov     rax, r10
    and     rax, QWORD PTR a1$[rsp]
    xor     rbx, rax
    mov     rax, QWORD PTR out4$[rsp]
    xor     rbx, QWORD PTR [rax]
    xor     rbx, rcx
    mov     QWORD PTR [rax], rbx
    mov     rbx, QWORD PTR x36$1$[rsp]
    and     rbx, rbp
    mov     r9, rbx
    not     r9
    and     r9, rdi
    or      r8, r11
    mov     rax, QWORD PTR out1$[rsp]
    xor     r8, r9
    not     r9
    and     r9, rcx
    or      rdx, rbp
    mov     rbp, QWORD PTR [rsp+80]
    or      r9, rsi
    xor     rbx, r12
    mov     rcx, r11
    not     rcx
    not     r14
    not     r13
    and     rcx, r9
    or      r9, rdi
```

```
    and     rbx, r14
    xor     r9, r15
    xor     rcx, rdx
    mov     rdx, QWORD PTR a1$[rsp]
    not     r9
    not     rcx
    and     r13, r10
    and     r9, r11
    and     rcx, rdx
    xor     r9, rbx
    mov     rbx, QWORD PTR [rsp+72]
    not     rcx
    xor     rcx, QWORD PTR [rax]
    or      r9, rdx
    not     r9
    xor     rcx, r8
    mov     QWORD PTR [rax], rcx
    mov     rax, QWORD PTR out3$[rsp]
    xor     r9, r13
    xor     r9, QWORD PTR [rax]
    xor     r9, r8
    mov     QWORD PTR [rax], r9
    pop     r15
    pop     r14
    pop     r13
    pop     r12
    pop     rdi
    pop     rsi
    ret     0
s1    ENDP
```

Nothing allocated in local stack by compiler, x36 is synonym for a5.

By the way, we can see here, the function saved RCX and RDX registers in allocated by caller space, but R8 and R9 are not saved but used from the beginning.

By the way, there are CPUs with much more GPR's, e.g. Itanium (128 registers).

23.2 ARM

In ARM, 64-bit instructions are appeared in ARMv8.

23.3 Float point numbers

Read more here24 about how float point numbers are processed in x86-64.

Chapter 24

Working with float point numbers using SIMD in x64

Of course, FPU remained in x86-compatible processors, when x64 extension was added. But at the time, SIMD-extensions (SSE[1], SSE2, etc) were already present, which can work with float point numbers as well. Number format remaining the same (IEEE 754).

So, x86-64 compilers are usually use SIMD-instructions. It can be said, it's a good news, because it's easier to work with them.

We will reuse here examples from the FPU section 15.

24.1 Simple example

```
double f (double a, double b)
{
        return a/3.14 + b*4.1;
};
```

Listing 24.1: MSVC 2012 x64 /Ox

```
__real@4010666666666666 DQ 04010666666666666r   ; 4.1
__real@40091eb851eb851f DQ 040091eb851eb851fr   ; 3.14

a$ = 8
b$ = 16
f       PROC
        divsd   xmm0, QWORD PTR __real@40091eb851eb851f
```

[1] Streaming SIMD Extensions

```
        mulsd   xmm1, QWORD PTR __real@4010666666666666
        addsd   xmm0, xmm1
        ret     0
f       ENDP
```

Input floating point values are passed in XMM0-XMM3 registers, all the rest—via stack [2].

a is passed in XMM0, b—via XMM1. XMM-registers are 128-bit (as we know from the section about SIMD22), but *double* values—64 bit ones, so only lower register half is used.

DIVSD is SSE-instruction, meaning "Divide Scalar Double-Precision Floating-Point Values", it just divides one value of *double* type by another, stored in the lower halves of operands.

Constants are encoded by compiler in IEEE 754 format.

MULSD and ADDSD works just as the same, but doing multiplication and addition.

The result of *double* type the function leaves in XMM0 register.

That is how non-optimizing MSVC works:

Listing 24.2: MSVC 2012 x64

```
__real@4010666666666666 DQ 04010666666666666r   ; 4.1
__real@40091eb851eb851f DQ 040091eb851eb851fr   ; 3.14

a$ = 8
b$ = 16
f       PROC
        movsdx  QWORD PTR [rsp+16], xmm1
        movsdx  QWORD PTR [rsp+8], xmm0
        movsdx  xmm0, QWORD PTR a$[rsp]
        divsd   xmm0, QWORD PTR __real@40091eb851eb851f
        movsdx  xmm1, QWORD PTR b$[rsp]
        mulsd   xmm1, QWORD PTR __real@4010666666666666
        addsd   xmm0, xmm1
        ret     0
f       ENDP
```

Slightly redundant. Input arguments are saved in "shadow space" (7.2.1), but only lower register halves, i.e., only 64-bit values of *double* type.

GCC produces very same code.

[2] MSDN: Parameter Passing

24.2 Passing floating point number via arguments

```
#include <math.h>
#include <stdio.h>

int main ()
{
        printf ("32.01 ^ 1.54 = %lf\n", pow (32.01,1.54));

        return 0;
}
```

They are passed in lower halves of the XMM0-XMM3 registers.

Listing 24.3: MSVC 2012 x64 /Ox

```
$SG1354 DB      '32.01 ^ 1.54 = %lf', 0aH, 00H

__real@40400147ae147ae1 DQ 040400147ae147ae1r   ; 32.01
__real@3ff8a3d70a3d70a4 DQ 03ff8a3d70a3d70a4r   ; 1.54

main    PROC
        sub     rsp, 40                                 ; ↙
    ↳ 00000028H
        movsdx  xmm1, QWORD PTR __real@3ff8a3d70a3d70a4
        movsdx  xmm0, QWORD PTR __real@40400147ae147ae1
        call    pow
        lea     rcx, OFFSET FLAT:$SG1354
        movaps  xmm1, xmm0
        movd    rdx, xmm1
        call    printf
        xor     eax, eax
        add     rsp, 40                                 ; ↙
    ↳ 00000028H
        ret     0
main    ENDP
```

There are no MOVSDX instruction in Intel [14] and AMD [1] manuals, it is called there just MOVSD. So there are two instructions sharing the same name in x86 (about other: B.6.2). Apparently, Microsoft developers wanted to get rid of mess, so they renamed it into MOVSDX. It just loads a value into lower half of XMM-register.

pow() takes arguments from XMM0 and XMM1, and returning result in XMM0. It is then moved into RDX for printf(). Why? Honestly speaking, I don't know, maybe because printf()—is a variable arguments function?

Listing 24.4: GCC 4.4.6 x64 -O3

```
.LC2:
        .string "32.01 ^ 1.54 = %lf\n"
main:
        sub     rsp, 8
        movsd   xmm1, QWORD PTR .LC0[rip]
        movsd   xmm0, QWORD PTR .LC1[rip]
        call    pow
        ; result is now in XMM0
        mov     edi, OFFSET FLAT:.LC2
        mov     eax, 1 ; number of vector registers passed
        call    printf
        xor     eax, eax
        add     rsp, 8
        ret
.LC0:
        .long   171798692
        .long   1073259479
.LC1:
        .long   2920577761
        .long   1077936455
```

GCC making more clear result. Value for `printf()` is passed in `XMM0`. By the way, here is a case when 1 is written into `EAX` for `printf()`—this mean that one argument will be passed in vector registers, just as the standard requires [21].

24.3 Comparison example

```
double d_max (double a, double b)
{
        if (a>b)
                return a;

        return b;
};
```

Listing 24.5: MSVC 2012 x64 /Ox

```
a$ = 8
b$ = 16
d_max   PROC
        comisd  xmm0, xmm1
```

```
        ja      SHORT $LN2@d_max
        movaps  xmm0, xmm1
$LN2@d_max:
        fatret  0
d_max   ENDP
```

Optimizing MSVC generates very easy code to understand.

COMISD is "Compare Scalar Ordered Double-Precision Floating-Point Values and Set EFLAGS". Essentially, that is what it does.

Non-optimizing MSVC generates more redundant code, but it is still not hard to understand:

Listing 24.6: MSVC 2012 x64

```
a$ = 8
b$ = 16
d_max   PROC
        movsdx  QWORD PTR [rsp+16], xmm1
        movsdx  QWORD PTR [rsp+8], xmm0
        movsdx  xmm0, QWORD PTR a$[rsp]
        comisd  xmm0, QWORD PTR b$[rsp]
        jbe     SHORT $LN1@d_max
        movsdx  xmm0, QWORD PTR a$[rsp]
        jmp     SHORT $LN2@d_max
$LN1@d_max:
        movsdx  xmm0, QWORD PTR b$[rsp]
$LN2@d_max:
        fatret  0
d_max   ENDP
```

However, GCC 4.4.6 did more optimizing and used the MAXSD ("Return Maximum Scalar Double-Precision Floating-Point Value") instruction, which just choose maximal value!

Listing 24.7: GCC 4.4.6 x64 -O3

```
d_max:
        maxsd   xmm0, xmm1
        ret
```

24.4 Summary

Only lower half of XMM-registers are used in all examples here, a number in IEEE 754 format is stored there.

Essentially, all instructions prefixed by `-SD` ("Scalar Double-Precision")—are instructions working with float point numbers in IEEE 754 format stored in the lower 64-bit half of XMM-register.

And it is easier than FPU, apparently because SIMD extensions were evolved not as chaotic as FPU in the past. Stack register model is not used.

If you would try to replace *double* to *float* in these examples, the same instructions will be used, but prefixed with `-SS` ("Scalar Single-Precision"), for example, `MOVSS`, `COMISS`, `ADDSS`, etc.

"Scalar" mean that SIMD-register will contain only one value instead of several. Instructions working with several values in a register simultaneously, has "Packed" in the name.

Chapter 25

Temperature converting

Another very popular example in programming books for beginners, is a small program converting Fahrenheit temperature to Celsius or back.

$$C = \frac{5 \cdot (F - 32)}{9}$$

I also added simple error handling: 1) we should check if user enters correct number; 2) we should check if Celsius temperature is not below -273 number (which is below absolute zero, as we may remember from school physics lessons).

`exit()` function terminates program instantly, without returning to the caller function.

25.1 Integer values

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
        int celsius, fahr;
        printf ("Enter temperature in Fahrenheit:\n");
        if (scanf ("%d", &fahr)!=1)
        {
                printf ("Error while parsing your input\n");
                exit(0);
        };

        celsius = 5 * (fahr-32) / 9;
```

```
        if (celsius<-273)
        {
                printf ("Error: incorrect temperature!\n");
                exit(0);
        };
        printf ("Celsius: %d\n", celsius);
};
```

25.1.1 MSVC 2012 x86 /Ox

Listing 25.1: MSVC 2012 x86 /Ox

```
$SG4228 DB      'Enter temperature in Fahrenheit:', 0aH, 00H
$SG4230 DB      '%d', 00H
$SG4231 DB      'Error while parsing your input', 0aH, 00H
$SG4233 DB      'Error: incorrect temperature!', 0aH, 00H
$SG4234 DB      'Celsius: %d', 0aH, 00H

_fahr$ = -4                                             ; size =↙
    ↳ 4
_main   PROC
        push    ecx
        push    esi
        mov     esi, DWORD PTR __imp__printf
        push    OFFSET $SG4228          ; 'Enter temperature in ↙
    ↳ Fahrenheit:'
        call    esi                     ; call printf()
        lea     eax, DWORD PTR _fahr$[esp+12]
        push    eax
        push    OFFSET $SG4230          ; '%d'
        call    DWORD PTR __imp__scanf
        add     esp, 12                                 ; ↙
    ↳ 0000000cH
        cmp     eax, 1
        je      SHORT $LN2@main
        push    OFFSET $SG4231          ; 'Error while parsing ↙
    ↳ your input'
        call    esi                     ; call printf()
        add     esp, 4
        push    0
        call    DWORD PTR __imp__exit
$LN9@main:
```

```
$LN2@main:
        mov     eax, DWORD PTR _fahr$[esp+8]
        add     eax, -32                                ; ↙
    ↳ ffffffe0H
        lea     ecx, DWORD PTR [eax+eax*4]
        mov     eax, 954437177                          ; 38↙
    ↳ e38e39H
        imul    ecx
        sar     edx, 1
        mov     eax, edx
        shr     eax, 31                                 ; ↙
    ↳ 0000001fH
        add     eax, edx
        cmp     eax, -273                               ; ↙
    ↳ fffffeefH
        jge     SHORT $LN1@main
        push    OFFSET $SG4233          ; 'Error: incorrect ↙
    ↳ temperature!'
        call    esi                     ; call printf()
        add     esp, 4
        push    0
        call    DWORD PTR __imp__exit
$LN10@main:
$LN1@main:
        push    eax
        push    OFFSET $SG4234          ; 'Celsius: %d'
        call    esi                     ; call printf()
        add     esp, 8
        ; return 0 - at least by C99 standard
        xor     eax, eax
        pop     esi
        pop     ecx
        ret     0
$LN8@main:
_main   ENDP
```

What we can say about it:

- Address of `printf()` is first loaded into `ESI` register, so the subsequent `printf()` calls are processed just by `CALL ESI` instruction. It's a very popular compiler technique, possible if several consequent calls to the same function are present in the code, and/or, if there are free register which can be used for this.
- We see `ADD EAX, -32` instruction at the place where 32 should be subtracted from the value. $EAX = EAX + (-32)$ is equivalent to $EAX = EAX - 32$ and

somehow, compiler decide to use ADD instead of SUB. Maybe it's worth it.

- LEA instruction is used when value should be multiplied by 5: `lea ecx, DWORD PTR [eax+eax*4]`. Yes, $i + i * 4$ is equivalent to $i * 5$ and LEA works faster then IMUL. By the way, `SHL EAX, 2 / ADD EAX, EAX` instructions pair could be also used here instead— some compilers do it in this way.
- Division by multiplication trick (14) is also used here.
- `main()` function returns 0 while we haven't `return 0` at its end. C99 standard tells us [15, 5.1.2.2.3] that `main()` will return 0 in case of `return` statement absence. This rule works only for `main()` function. Though, MSVC doesn't support C99, but maybe partly it does?

25.1.2 MSVC 2012 x64 /Ox

The code is almost the same, but I've found `INT 3` instructions after each `exit()` call:

```
        xor     ecx, ecx
        call    QWORD PTR __imp_exit
        int     3
```

`INT 3` is a debugger breakpoint.

It is known that `exit()` is one of functions which never can return [1], so if it does, something really odd happens and it's time to load debugger.

25.2 Float point values

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
        double celsius, fahr;
        printf ("Enter temperature in Fahrenheit:\n");
        if (scanf ("%lf", &fahr)!=1)
        {
                printf ("Error while parsing your input\n");
                exit(0);
        };

        celsius = 5 * (fahr-32) / 9;
```

[1] another popular one is `longjmp()`

```
        if (celsius<-273)
        {
                printf ("Error: incorrect temperature!\n");
                exit(0);
        };
        printf ("Celsius: %lf\n", celsius);
};
```

MSVC 2010 x86 use FPU instructions...

Listing 25.2: MSVC 2010 x86 /Ox

```
$SG4038 DB      'Enter temperature in Fahrenheit:', 0aH, 00H
$SG4040 DB      '%lf', 00H
$SG4041 DB      'Error while parsing your input', 0aH, 00H
$SG4043 DB      'Error: incorrect temperature!', 0aH, 00H
$SG4044 DB      'Celsius: %lf', 0aH, 00H

__real@c071100000000000 DQ 0c071100000000000r   ; -273
__real@4022000000000000 DQ 04022000000000000r   ; 9
__real@4014000000000000 DQ 04014000000000000r   ; 5
__real@4040000000000000 DQ 04040000000000000r   ; 32

_fahr$ = -8                                             ; size =↙
    ↳ 8
_main   PROC
        sub     esp, 8
        push    esi
        mov     esi, DWORD PTR __imp__printf
        push    OFFSET $SG4038          ; 'Enter temperature in ↙
    ↳ Fahrenheit:'
        call    esi                     ; call printf
        lea     eax, DWORD PTR _fahr$[esp+16]
        push    eax
        push    OFFSET $SG4040          ; '%lf'
        call    DWORD PTR __imp__scanf
        add     esp, 12                                 ; ↙
    ↳ 0000000cH
        cmp     eax, 1
        je      SHORT $LN2@main
        push    OFFSET $SG4041          ; 'Error while parsing ↙
    ↳ your input'
        call    esi                     ; call printf
        add     esp, 4
```

```
        push    0
        call    DWORD PTR __imp__exit
$LN2@main:
        fld     QWORD PTR _fahr$[esp+12]
        fsub    QWORD PTR __real@4040000000000000 ; 32
        fmul    QWORD PTR __real@4014000000000000 ; 5
        fdiv    QWORD PTR __real@4022000000000000 ; 9
        fld     QWORD PTR __real@c071100000000000 ; -273
        fcomp   ST(1)
        fnstsw  ax
        test    ah, 65                                  ; ↙
    ↳ 00000041H
        jne     SHORT $LN1@main
        push    OFFSET $SG4043          ; 'Error: incorrect ↙
    ↳ temperature!'
        fstp    ST(0)
        call    esi                     ; call printf
        add     esp, 4
        push    0
        call    DWORD PTR __imp__exit
$LN1@main:
        sub     esp, 8
        fstp    QWORD PTR [esp]
        push    OFFSET $SG4044          ; 'Celsius: %lf'
        call    esi
        add     esp, 12                                 ; ↙
    ↳ 0000000cH
        ; return 0
        xor     eax, eax
        pop     esi
        add     esp, 8
        ret     0
$LN10@main:
_main   ENDP
```

... but MSVC from year 2012 use SIMD instructions instead:

Listing 25.3: MSVC 2010 x86 /Ox

```
$SG4228 DB      'Enter temperature in Fahrenheit:', 0aH, 00H
$SG4230 DB      '%lf', 00H
$SG4231 DB      'Error while parsing your input', 0aH, 00H
$SG4233 DB      'Error: incorrect temperature!', 0aH, 00H
$SG4234 DB      'Celsius: %lf', 0aH, 00H
__real@c071100000000000 DQ 0c071100000000000r   ; -273
```

```
__real@4040000000000000 DQ 0404000000000000r   ; 32
__real@4022000000000000 DQ 0402200000000000r   ; 9
__real@4014000000000000 DQ 0401400000000000r   ; 5

_fahr$ = -8                                              ; size =↙
    ↳ 8
_main   PROC
        sub     esp, 8
        push    esi
        mov     esi, DWORD PTR __imp__printf
        push    OFFSET $SG4228          ; 'Enter temperature in ↙
    ↳ Fahrenheit:'
        call    esi                     ; call printf
        lea     eax, DWORD PTR _fahr$[esp+16]
        push    eax
        push    OFFSET $SG4230          ; '%lf'
        call    DWORD PTR __imp__scanf
        add     esp, 12                                  ; ↙
    ↳ 0000000cH
        cmp     eax, 1
        je      SHORT $LN2@main
        push    OFFSET $SG4231          ; 'Error while parsing ↙
    ↳ your input'
        call    esi                     ; call printf
        add     esp, 4
        push    0
        call    DWORD PTR __imp__exit
$LN9@main:
$LN2@main:
        movsd   xmm1, QWORD PTR _fahr$[esp+12]
        subsd   xmm1, QWORD PTR __real@4040000000000000 ; 32
        movsd   xmm0, QWORD PTR __real@c071100000000000 ; -273
        mulsd   xmm1, QWORD PTR __real@4014000000000000 ; 5
        divsd   xmm1, QWORD PTR __real@4022000000000000 ; 9
        comisd  xmm0, xmm1
        jbe     SHORT $LN1@main
        push    OFFSET $SG4233          ; 'Error: incorrect ↙
    ↳ temperature!'
        call    esi                     ; call printf
        add     esp, 4
        push    0
        call    DWORD PTR __imp__exit
$LN10@main:
```

```
$LN1@main:
        sub     esp, 8
        movsd   QWORD PTR [esp], xmm1
        push    OFFSET $SG4234          ; 'Celsius: %lf'
        call    esi                     ; call printf
        add     esp, 12                                 ; ↙
    ↳ 0000000cH
        ; return 0
        xor     eax, eax
        pop     esi
        add     esp, 8
        ret     0
$LN8@main:
_main   ENDP
```

Of course, SIMD instructions are available in x86 mode, including those working with floating point numbers. It's somewhat easier to use them for calculations, so the new Microsoft compiler use them.

We may also notice that -273 value is loaded into XMM0 register too early. And that's OK, because, compiler may emit instructions not in the order they are in source code.

Chapter 26

C99 restrict

Here is a reason why FORTRAN programs, in some cases, works faster than C/C++ ones.

```
void f1 (int* x, int* y, int* sum, int* product, int* ↙
    ↳ sum_product, int* update_me, size_t s)
{
        for (int i=0; i<s; i++)
        {
                sum[i]=x[i]+y[i];
                product[i]=x[i]*y[i];
                update_me[i]=i*123; // some dummy value
                sum_product[i]=sum[i]+product[i];
        };
};
```

That's very simple example with one specific thing in it: pointer to `update_me` array could be a pointer to `sum` array, `product` array, or even `sum_product` array—since it is not a crime in it, right?

Compiler is fully aware about it, so it generates a code with four stages in loop body:

- calculate next `sum[i]`
- calculate next `product[i]`
- calculate next `update_me[i]`
- calculate next `sum_product[i]`—on this stage, we need to load from memory already calculated `sum[i]` and `product[i]`

Is it possible to optimize the last stage? Since already calculated `sum[i]` and `product[i]` are not necessary to load from memory again, because we already calculated them. Yes, but compiler is not sure that nothing was overwritten on 3rd stage! This is called "pointer aliasing", a situation, when compiler cannot be sure that a memory to which a pointer is pointing, was not changed.

restrict in C99 standard [15, 6.7.3/1] is a promise, given by programmer to compiler the function arguments marked by this keyword will always be pointing to different memory locations and never be crossed.

If to be more precise and describe this formally, *restrict* shows that only this pointer is to be used to access an object, with which we are working via this pointer, and no other pointer will be used for it. It can be even said the object will be accessed only via one single pointer, if it is marked as *restrict*.

Let's add this keyword to each argument-pointer:

```
void f2 (int* restrict x, int* restrict y, int* restrict sum, ↙
    ↳ int* restrict product, int* restrict sum_product,
        int* restrict update_me, size_t s)
{
        for (int i=0; i<s; i++)
        {
                sum[i]=x[i]+y[i];
                product[i]=x[i]*y[i];
                update_me[i]=i*123; // some dummy value
                sum_product[i]=sum[i]+product[i];
        };
};
```

Let's see results:

Listing 26.1: GCC x64: f1()

```
f1:
        push    r15 r14 r13 r12 rbp rdi rsi rbx
        mov     r13, QWORD PTR 120[rsp]
        mov     rbp, QWORD PTR 104[rsp]
        mov     r12, QWORD PTR 112[rsp]
        test    r13, r13
        je      .L1
        add     r13, 1
        xor     ebx, ebx
        mov     edi, 1
        xor     r11d, r11d
        jmp     .L4
.L6:
        mov     r11, rdi
```

```
        mov     rdi, rax
.L4:
        lea     rax, 0[0+r11*4]
        lea     r10, [rcx+rax]
        lea     r14, [rdx+rax]
        lea     rsi, [r8+rax]
        add     rax, r9
        mov     r15d, DWORD PTR [r10]
        add     r15d, DWORD PTR [r14]
        mov     DWORD PTR [rsi], r15d         ; store to sum[]
        mov     r10d, DWORD PTR [r10]
        imul    r10d, DWORD PTR [r14]
        mov     DWORD PTR [rax], r10d         ; store to product↙
    ↳ []
        mov     DWORD PTR [r12+r11*4], ebx    ; store to ↙
    ↳ update_me[]
        add     ebx, 123
        mov     r10d, DWORD PTR [rsi]         ; reload sum[i]
        add     r10d, DWORD PTR [rax]         ; reload product[i↙
    ↳ ]
        lea     rax, 1[rdi]
        cmp     rax, r13
        mov     DWORD PTR 0[rbp+r11*4], r10d  ; store to ↙
    ↳ sum_product[]
        jne     .L6
.L1:
        pop     rbx rsi rdi rbp r12 r13 r14 r15
        ret
```

Listing 26.2: GCC x64: f2()

```
f2:
        push    r13 r12 rbp rdi rsi rbx
        mov     r13, QWORD PTR 104[rsp]
        mov     rbp, QWORD PTR 88[rsp]
        mov     r12, QWORD PTR 96[rsp]
        test    r13, r13
        je      .L7
        add     r13, 1
        xor     r10d, r10d
        mov     edi, 1
        xor     eax, eax
        jmp     .L10
.L11:
```

```
        mov     rax, rdi
        mov     rdi, r11
.L10:
        mov     esi, DWORD PTR [rcx+rax*4]
        mov     r11d, DWORD PTR [rdx+rax*4]
        mov     DWORD PTR [r12+rax*4], r10d  ; store to ↙
    ↳ update_me[]
        add     r10d, 123
        lea     ebx, [rsi+r11]
        imul    r11d, esi
        mov     DWORD PTR [r8+rax*4], ebx    ; store to sum[]
        mov     DWORD PTR [r9+rax*4], r11d   ; store to product↙
    ↳ []
        add     r11d, ebx
        mov     DWORD PTR 0[rbp+rax*4], r11d ; store to ↙
    ↳ sum_product[]
        lea     r11, 1[rdi]
        cmp     r11, r13
        jne     .L11
.L7:
        pop     rbx rsi rdi rbp r12 r13
        ret
```

The difference between compiled `f1()` and `f2()` function is as follows: in `f1()`, `sum[i]` and `product[i]` are reloaded in the middle of loop, and in `f2()` there are no such thing, already calculated values are used, since we "promised" to compiler, that no one and nothing will change values in `sum[i]` and `product[i]` while execution of loop body, so it is "sure" the value from memory may not be loaded again. Obviously, second example will work faster.

But what if pointers in function arguments will be crossed somehow? This will be on programmer's conscience, but results will be incorrect.

Let's back to FORTRAN. Compilers from this programming language treats all pointers as such, so when it was not possible to set *restrict*, FORTRAN in these cases may generate faster code.

How practical is it? In the cases when function works with several big blocks in memory. E.g. there are a lot of such in linear algebra. A lot of linear algebra used on supercomputers/HPC[1], probably, that is why, traditionally, FORTRAN is still used there [19].

But when a number of iterations is not very big, certainly, speed boost will not be significant.

[1] High-Performance Computing

Chapter 27

Inline functions

Inlined code is when compiler, instead of placing call instruction to small or tiny function, just placing its body right in-place.

Listing 27.1: Simple example

```
#include <stdio.h>

int celsius_to_fahrenheit (int celsius)
{
        return celsius * 9 / 5 + 32;
};

int main(int argc, char *argv[])
{
        int celsius=atol(argv[1]);
        printf ("%d\n", celsius_to_fahrenheit (celsius));
};
```

... is compiled in very predictable way, however, if to turn on GCC optimization (-O3), we'll see:

Listing 27.2: GCC 4.8.1 -O3

```
_main:
        push    ebp
        mov     ebp, esp
        and     esp, -16
        sub     esp, 16
        call    ___main
        mov     eax, DWORD PTR [ebp+12]
        mov     eax, DWORD PTR [eax+4]
```

```
        mov     DWORD PTR [esp], eax
        call    _atol
        mov     edx, 1717986919
        mov     DWORD PTR [esp], OFFSET FLAT:LC2 ; "%d\12\0"
        lea     ecx, [eax+eax*8]
        mov     eax, ecx
        imul    edx
        sar     ecx, 31
        sar     edx
        sub     edx, ecx
        add     edx, 32
        mov     DWORD PTR [esp+4], edx
        call    _printf
        leave
        ret
```

(Here division is done by multiplication(14).)

Yes, our small function was just placed before `printf()` call. Why? It may be faster than executing this function's code plus calling/returning overhead.

In past, such function must be marked with "inline" keyword in function's declaration, however, in modern times, these functions are chosen automatically by compiler.

Another very common automatic optimization is inlining of string functions like *strcpy()*, *strcmp()*, etc.

Listing 27.3: Another simple example

```
bool is_bool (char *s)
{
        if (strcmp (s, "true")==0)
                return true;
        if (strcmp (s, "false")==0)
                return false;

        assert(0);
};
```

Listing 27.4: GCC 4.8.1 -O3

```
_is_bool:
        push    edi
        mov     ecx, 5
        push    esi
        mov     edi, OFFSET FLAT:LC0 ; "true\0"
        sub     esp, 20
```

```
        mov     esi, DWORD PTR [esp+32]
        repz cmpsb
        je      L3
        mov     esi, DWORD PTR [esp+32]
        mov     ecx, 6
        mov     edi, OFFSET FLAT:LC1 ; "false\0"
        repz cmpsb
        seta    cl
        setb    dl
        xor     eax, eax
        cmp     cl, dl
        jne     L8
        add     esp, 20
        pop     esi
        pop     edi
        ret
```

Here is an example of very frequently seen piece of strcmp() code generated by MSVC:

Listing 27.5: MSVC

```
    mov     dl, [eax]
    cmp     dl, [ecx]
    jnz     short loc_10027FA0
    test    dl, dl
    jz      short loc_10027F9C
    mov     dl, [eax+1]
    cmp     dl, [ecx+1]
    jnz     short loc_10027FA0
    add     eax, 2
    add     ecx, 2
    test    dl, dl
    jnz     short loc_10027F80

loc_10027F9C: ; CODE XREF: f1+448
    xor     eax, eax
    jmp     short loc_10027FA5

loc_10027FA0: ; CODE XREF: f1+444
                ; f1+450
    sbb     eax, eax
    sbb     eax, 0FFFFFFFFh
```

I wrote small IDA script for searching and folding such very frequently seen pieces of inline code:
https://github.com/yurichev/IDA_scripts.

Chapter 28

Incorrectly disassembled code

Practicing reverse engineers often dealing with incorrectly disassembled code.

28.1 Disassembling started incorrectly (x86)

Unlike ARM and MIPS (where any instruction has length of 2 or 4 bytes), x86 instructions has variable size, so, any disassembler, starting at the middle of x86 instruction, may produce incorrect results.

As an example:

```
add     [ebp-31F7Bh], cl
dec     dword ptr [ecx-3277Bh]
dec     dword ptr [ebp-2CF7Bh]
inc     dword ptr [ebx-7A76F33Ch]
fdiv    st(4), st
db 0FFh
dec     dword ptr [ecx-21F7Bh]
dec     dword ptr [ecx-22373h]
dec     dword ptr [ecx-2276Bh]
dec     dword ptr [ecx-22B63h]
dec     dword ptr [ecx-22F4Bh]
dec     dword ptr [ecx-23343h]
jmp     dword ptr [esi-74h]
xchg    eax, ebp
clc
std
db 0FFh
db 0FFh
mov     word ptr [ebp-214h], cs
```

```
mov     word ptr [ebp-238h], ds
mov     word ptr [ebp-23Ch], es
mov     word ptr [ebp-240h], fs
mov     word ptr [ebp-244h], gs
pushf
pop     dword ptr [ebp-210h]
mov     eax, [ebp+4]
mov     [ebp-218h], eax
lea     eax, [ebp+4]
mov     [ebp-20Ch], eax
mov     dword ptr [ebp-2D0h], 10001h
mov     eax, [eax-4]
mov     [ebp-21Ch], eax
mov     eax, [ebp+0Ch]
mov     [ebp-320h], eax
mov     eax, [ebp+10h]
mov     [ebp-31Ch], eax
mov     eax, [ebp+4]
mov     [ebp-314h], eax
call    ds:IsDebuggerPresent
mov     edi, eax
lea     eax, [ebp-328h]
push    eax
call    sub_407663
pop     ecx
test    eax, eax
jnz     short loc_402D7B
```

There are incorrectly disassembled instructions at the beginning, but eventually, disassembler finds right track.

28.2 How random noise looks disassembled?

Common properties which can be easily spotted are:

- Unusually big instruction dispersion. Most frequent x86 instructions are `PUSH`, `MOV`, `CALL`, but here we will see instructions from any instruction group: FPU instructions, `IN`/`OUT` instructions, rare and system instructions, everything messed up in one single place.
- Big and random values, offsets and immediates.
- Jumps having incorrect offsets often jumping into the middle of another instructions.

Listing 28.1: random noise (x86)

```
    mov     bl, 0Ch
    mov     ecx, 0D38558Dh
    mov     eax, ds:2C869A86h
    db      67h
    mov     dl, 0CCh
    insb
    movsb
    push    eax
    xor     [edx-53h], ah
    fcom    qword ptr [edi-45A0EF72h]
    pop     esp
    pop     ss
    in      eax, dx
    dec     ebx
    push    esp
    lds     esp, [esi-41h]
    retf
    rcl     dword ptr [eax], cl
    mov     cl, 9Ch
    mov     ch, 0DFh
    push    cs
    insb
    mov     esi, 0D9C65E4Dh
    imul    ebp, [ecx], 66h
    pushf
    sal     dword ptr [ebp-64h], cl
    sub     eax, 0AC433D64h
    out     8Ch, eax
    pop     ss
    sbb     [eax], ebx
    aas
    xchg    cl, [ebx+ebx*4+14B31Eh]
    jecxz   short near ptr loc_58+1
    xor     al, 0C6h
    inc     edx
    db      36h
    pusha
    stosb
    test    [ebx], ebx
    sub     al, 0D3h ; 'L'
    pop     eax
    stosb
```

```
loc_58: ; CODE XREF: seg000:0000004A
    test    [esi], eax
    inc     ebp
    das
    db      64h
    pop     ecx
    das
    hlt

    pop     edx
    out     0B0h, al
    lodsb
    push    ebx
    cdq
    out     dx, al
    sub     al, 0Ah
    sti
    outsd
    add     dword ptr [edx], 96FCBE4Bh
    and     eax, 0E537EE4Fh
    inc     esp
    stosd
    cdq
    push    ecx
    in      al, 0CBh
    mov     ds:0D114C45Ch, al
    mov     esi, 659D1985h
    enter   6FE8h, 0D9h
    enter   6FE6h, 0D9h
    xchg    eax, esi
    sub     eax, 0A599866Eh
    retn

    pop     eax
    dec     eax
    adc     al, 21h ; '!'
    lahf
    inc     edi
    sub     eax, 9062EE5Bh
    bound   eax, [ebx]

loc_A2: ; CODE XREF: seg000:00000120
```

```
    wait
    iret

    jnb     short loc_D7
    cmpsd
    iret

    jnb     short loc_D7
    sub     ebx, [ecx]
    in      al, 0Ch
    add     esp, esp
    mov     bl, 8Fh
    xchg    eax, ecx
    int     67h
    pop     ds
    pop     ebx
    db      36h
    xor     esi, [ebp-4Ah]
    mov     ebx, 0EB4F980Ch
    repne add bl, dh
    imul    ebx, [ebp+5616E7A5h], 67A4D1EEh
    xchg    eax, ebp
    scasb
    push    esp
    wait
    mov     dl, 11h
    mov     ah, 29h ; ')'
    fist    dword ptr [edx]

loc_D7: ; CODE XREF: seg000:000000A4
        ; seg000:000000A8 ...
    dec     dword ptr [ebp-5D0E0BA4h]
    call    near ptr 622FEE3Eh
    sbb     ax, 5A2Fh
    jmp     dword ptr cs:[ebx]

    xor     ch, [edx-5]
    inc     esp
    push    edi
    xor     esp, [ebx-6779D3B8h]
    pop     eax
    int     3               ; Trap to Debugger
    rcl     byte ptr [ebx-3Eh], cl
```

```
    xor     [edi], bl
    sbb     al, [edx+ecx*4]
    xor     ah, [ecx-1DA4E05Dh]
    push    edi
    xor     ah, cl
    popa
    cmp     dword ptr [edx-62h], 46h ; 'F'
    dec     eax
    in      al, 69h
    dec     ebx
    iret

    or      al, 6
    jns     short near ptr loc_D7+3
    shl     byte ptr [esi], 42h
    repne adc [ebx+2Ch], eax
    icebp
    cmpsd
    leave
    push    esi
    jmp     short loc_A2

    and     eax, 0F2E41FE9h
    push    esi
    loop    loc_14F
    add     ah, fs:[edx]

loc_12D: ; CODE XREF: seg000:00000169
    mov     dh, 0F7h
    add     [ebx+7B61D47Eh], esp
    mov     edi, 79F19525h
    rcl     byte ptr [eax+22015F55h], cl
    cli
    sub     al, 0D2h ; 'T'
    dec     eax
    mov     ds:0A81406F5h, eax
    sbb     eax, 0A7AA179Ah
    in      eax, dx

loc_14F: ; CODE XREF: seg000:00000128
    and     [ebx-4CDFAC74h], ah
    pop     ecx
    push    esi
```

```
    mov     bl, 2Dh ; '-'
    in      eax, 2Ch
    stosd
    inc     edi
    push    esp

locret_15E: ; CODE XREF: seg000:loc_1A0
    retn    0C432h

    and     al, 86h
    cwde
    and     al, 8Fh
    cmp     ebp, [ebp+7]
    jz      short loc_12D
    sub     bh, ch
    or      dword ptr [edi-7Bh], 8A16C0F7h
    db      65h
    insd
    mov     al, ds:0A3A5173Dh
    dec     ecx
    push    ds
    xor     al, cl
    jg      short loc_195
    push    6Eh ; 'n'
    out     0DDh, al
    inc     edi
    sub     eax, 6899BBF1h
    leave
    rcr     dword ptr [ecx-69h], cl
    sbb     ch, [edi+5EDDCB54h]

loc_195: ; CODE XREF: seg000:0000017F
    push    es
    repne sub ah, [eax-105FF22Dh]
    cmc
    and     ch, al

loc_1A0: ; CODE XREF: seg000:00000217
    jnp     short near ptr locret_15E+1
    or      ch, [eax-66h]
    add     [edi+edx-35h], esi
    out     dx, al
    db      2Eh
```

```
    call    far ptr 1AAh:6832F5DDh
    jz      short near ptr loc_1DA+1
    sbb     esp, [edi+2CB02CEFh]
    xchg    eax, edi
    xor     [ebx-766342ABh], edx

loc_1C1: ; CODE XREF: seg000:00000212
    cmp     eax, 1BE9080h
    add     [ecx], edi
    aad     0
    imul    esp, [edx-70h], 0A8990126h
    or      dword ptr [edx+10C33693h], 4Bh
    popf

loc_1DA: ; CODE XREF: seg000:000001B2
    mov     ecx, cs
    aaa
    mov     al, 39h ; '9'
    adc     byte ptr [eax-77F7F1C5h], 0C7h
    add     [ecx], bl
    retn    0DD42h

    db      3Eh
    mov     fs:[edi], edi
    and     [ebx-24h], esp
    db      64h
    xchg    eax, ebp
    push    cs
    adc     eax, [edi+36h]
    mov     bh, 0C7h
    sub     eax, 0A710CBE7h
    xchg    eax, ecx
    or      eax, 51836E42h
    xchg    eax, ebx
    inc     ecx
    jb      short near ptr loc_21E+3
    db      64h
    xchg    eax, esp
    and     dh, [eax-31h]
    mov     ch, 13h
    add     ebx, edx
    jnb     short loc_1C1
    db      65h
```

```
    adc     al, 0C5h
    js      short loc_1A0
    sbb     eax, 887F5BEEh

loc_21E: ; CODE XREF: seg000:00000207
    mov     eax, 888E1FD6h
    mov     bl, 90h
    cmp     [eax], ecx
    rep int 61h             ; reserved for user interrupt
    and     edx, [esi-7EB5C9EAh]
    fisttp  qword ptr [eax+esi*4+38F9BA6h]
    jmp     short loc_27C

    fadd    st, st(2)
    db      3Eh
    mov     edx, 54C03172h
    retn

    db      64h
    pop     ds
    xchg    eax, esi
    rcr     ebx, cl
    cmp     [di+2Eh], ebx
    repne xor [di-19h], dh
    insd
    adc     dl, [eax-0C4579F7h]
    push    ss
    xor     [ecx+edx*4+65h], ecx
    mov     cl, [ecx+ebx-32E8AC51h]
    or      [ebx], ebp
    cmpsb
    lodsb
    iret
```

Listing 28.2: random noise (x86-64)

```
    lea     esi, [rax+rdx*4+43558D29h]

loc_AF3: ; CODE XREF: seg000:0000000000000B46
    rcl     byte ptr [rsi+rax*8+29BB423Ah], 1
    lea     ecx, cs:0FFFFFFFFB2A6780Fh
    mov     al, 96h
    mov     ah, 0CEh
    push    rsp
```

```
    lods    byte ptr [esi]

    db  2Fh ; /

    pop     rsp
    db      64h
    retf    0E993h

    cmp     ah, [rax+4Ah]
    movzx   rsi, dword ptr [rbp-25h]
    push    4Ah
    movzx   rdi, dword ptr [rdi+rdx*8]

    db  9Ah

    rcr     byte ptr [rax+1Dh], cl
    lodsd
    xor     [rbp+6CF20173h], edx
    xor     [rbp+66F8B593h], edx
    push    rbx
    sbb     ch, [rbx-0Fh]
    stosd
    int     87h
    db      46h, 4Ch
    out     33h, rax
    xchg    eax, ebp
    test    ecx, ebp
    movsd
    leave
    push    rsp

    db  16h

    xchg    eax, esi
    pop     rdi

loc_B3D: ; CODE XREF: seg000:0000000000000B5F
    mov     ds:93CA685DF98A90F9h, eax
    jnz     short near ptr loc_AF3+6
    out     dx, eax
    cwde
    mov     bh, 5Dh ; ']'
    movsb
```

```
    pop     rbp

    db  60h ; ‘

    movsxd  rbp, dword ptr [rbp-17h]
    pop     rbx
    out     7Dh, al
    add     eax, 0D79BE769h

    db  1Fh

    retf    0CAB9h

    jl      short near ptr loc_B3D+4
    sal     dword ptr [rbx+rbp+4Dh], 0D3h
    mov     cl, 41h ; ’A’
    imul    eax, [rbp-5B77E717h], 1DDE6E5h
    imul    ecx, ebx, 66359BCCh
    xlat

    db  60h ; ‘

    cmp     bl, [rax]
    and     ebp, [rcx-57h]
    stc
    sub     [rcx+1A533AB4h], al
    jmp     short loc_C05

    db  4Bh ; K

    int     3               ; Trap to Debugger
    xchg    ebx, [rsp+rdx-5Bh]

    db 0D6h

    mov     esp, 0C5BA61F7h
    out     0A3h, al        ; Interrupt Controller #2, 8259A
    add     al, 0A6h
    pop     rbx
    cmp     bh, fs:[rsi]
    and     ch, cl
    cmp     al, 0F3h
```

```
    db  0Eh

    xchg    dh, [rbp+rax*4-4CE9621Ah]
    stosd
    xor     [rdi], ebx
    stosb
    xchg    eax, ecx
    push    rsi
    insd
    fidiv   word ptr [rcx]
    xchg    eax, ecx
    mov     dh, 0C0h ; 'L'
    xchg    eax, esp
    push    rsi
    mov     dh, [rdx+rbp+6918F1F3h]
    xchg    eax, ebp
    out     9Dh, al

loc_BC0: ; CODE XREF: seg000:0000000000000C26
    or      [rcx-0Dh], ch
    int     67h             ;  - LIM EMS
    push    rdx
    sub     al, 43h ; 'C'
    test    ecx, ebp
    test    [rdi+71F372A4h], cl

    db    7

    imul    ebx, [rsi-0Dh], 2BB30231h
    xor     ebx, [rbp-718B6E64h]
    jns     short near ptr loc_C56+1
    ficomp  dword ptr [rcx-1Ah]
    and     eax, 69BEECC7h
    mov     esi, 37DA40F6h
    imul    r13, [rbp+rdi*8+529F33CDh], 0FFFFFFFFF35CDD30h
    or      [rbx], edx
    imul    esi, [rbx-34h], 0CDA42B87h

    db  36h ; 6
    db  1Fh


loc_C05: ; CODE XREF: seg000:0000000000000B86
```

```
    add     dh, [rcx]
    mov     edi, 0DD3E659h
    ror     byte ptr [rdx-33h], cl
    xlat
    db      48h
    sub     rsi, [rcx]

    db  1Fh
    db    6

    xor     [rdi+13F5F362h], bh
    cmpsb
    sub     esi, [rdx]
    pop     rbp
    sbb     al, 62h ; 'b'
    mov     dl, 33h ; '3'

    db  4Dh ; M
    db  17h

    jns     short loc_BC0
    push    0FFFFFFFFFFFFFF86h

loc_C2A: ; CODE XREF: seg000:0000000000000C8F
    sub     [rdi-2Ah], eax

    db 0FEh

    cmpsb
    wait
    rcr     byte ptr [rax+5Fh], cl
    cmp     bl, al
    pushfq
    xchg    ch, cl

    db  4Eh ; N
    db  37h ; 7

    mov     ds:0E43F3CCD3D9AB295h, eax
    cmp     ebp, ecx
    jl      short loc_C87
    retn    8574h
```

```
    out     3, al           ; DMA controller, 8237A-5.
                            ; channel 1 base address and word ↙
    ↳ count

loc_C4C: ; CODE XREF: seg000:0000000000000C7F
    cmp     al, 0A6h
    wait
    push    0FFFFFFFFFFFFFFBEh

    db  82h

    ficom   dword ptr [rbx+r10*8]

loc_C56: ; CODE XREF: seg000:0000000000000BDE
    jnz     short loc_C76
    xchg    eax, edx
    db      26h
    wait
    iret

    push    rcx

    db  48h ; H
    db  9Bh
    db  64h ; d
    db  3Eh ; >
    db  2Fh ; /

    mov     al, ds:8A7490CA2E9AA728h
    stc

    db  60h ; ‘

    test    [rbx+rcx], ebp
    int     3               ; Trap to Debugger
    xlat

loc_C72: ; CODE XREF: seg000:0000000000000CC6
    mov     bh, 98h

    db  2Eh ; .
    db 0DFh
```

```
loc_C76: ; CODE XREF: seg000:loc_C56
    jl      short loc_C91
    sub     ecx, 13A7CCF2h
    movsb
    jns     short near ptr loc_C4C+1
    cmpsd
    sub     ah, ah
    cdq

    db  6Bh ; k
    db  5Ah ; Z


loc_C87: ; CODE XREF: seg000:0000000000000C45
    or      ecx, [rbx+6Eh]
    rep in eax, 0Eh         ; DMA controller, 8237A-5.
                            ; Clear mask registers.
                            ; Any OUT enables all 4 channels.
    cmpsb
    jnb     short loc_C2A

loc_C91: ; CODE XREF: seg000:loc_C76
    scasd
    add     dl, [rcx+5FEF30E6h]
    enter   0FFFFFFFFFFFFC733h, 7Ch
    insd
    mov     ecx, gs
    in      al, dx
    out     2Dh, al
    mov     ds:6599E434E6D96814h, al
    cmpsb
    push    0FFFFFFFFFFFFFFD6h
    popfq
    xor     ecx, ebp
    db      48h
    insb
    test    al, cl
    xor     [rbp-7Bh], cl
    and     al, 9Bh

    db  9Ah
```

```
    push    rsp
    xor     al, 8Fh
    cmp     eax, 924E81B9h
    clc
    mov     bh, 0DEh
    jbe     short near ptr loc_C72+1

    db  1Eh

    retn    8FCAh

    db 0C4h ; -


loc_CCD: ; CODE XREF: seg000:0000000000000D22
    adc     eax, 7CABFBF8h

    db  38h ; 8

    mov     ebp, 9C3E66FCh
    push    rbp
    dec     byte ptr [rcx]
    sahf
    fidivr  word ptr [rdi+2Ch]

    db  1Fh

    db      3Eh
    xchg    eax, esi

loc_CE2: ; CODE XREF: seg000:0000000000000D5E
    mov     ebx, 0C7AFE30Bh
    clc
    in      eax, dx
    sbb     bh, bl
    xchg    eax, ebp

    db  3Fh ; ?

    cmp     edx, 3EC3E4D7h
    push    51h
    db      3Eh
    pushfq
```

```
    jl      short loc_D17
    test    [rax-4CFF0D49h], ebx

    db  2Fh ; /

    rdtsc
    jns     short near ptr loc_D40+4
    mov     ebp, 0B2BB03D8h
    in      eax, dx

    db  1Eh

    fsubr   dword ptr [rbx-0Bh]
    jns     short loc_D70
    scasd
    mov     ch, 0C1h ; '+'
    add     edi, [rbx-53h]

    db 0E7h


loc_D17: ; CODE XREF: seg000:0000000000000CF7
    jp      short near ptr unk_D79
    scasd
    cmc
    sbb     ebx, [rsi]
    fsubr   dword ptr [rbx+3Dh]
    retn

    db    3

    jnp     short near ptr loc_CCD+4
    db      36h
    adc     r14b, r13b

    db  1Fh

    retf

    test    [rdi+rdi*2], ebx
    cdq
    or      ebx, edi
    test    eax, 310B94BCh
```

```
    ffreep  st(7)
    cwde
    sbb     esi, [rdx+53h]
    push    5372CBAAh

loc_D40: ; CODE XREF: seg000:0000000000000D02
    push    53728BAAh
    push    0FFFFFFFF85CF2FCh

    db  0Eh

    retn    9B9Bh

    movzx   r9, dword ptr [rdx]
    adc     [rcx+43h], ebp
    in      al, 31h

    db  37h ; 7

    jl      short loc_DC5
    icebp
    sub     esi, [rdi]
    clc
    pop     rdi
    jb      short near ptr loc_CE2+1
    or      al, 8Fh
    mov     ecx, 770EFF81h
    sub     al, ch
    sub     al, 73h ; 's'
    cmpsd
    adc     bl, al
    out     87h, eax        ; DMA page register 74LS612:
                            ; Channel 0 (address bits 16-23)

loc_D70: ; CODE XREF: seg000:0000000000000D0E
    adc     edi, ebx
    db      49h
    outsb
    enter   33E5h, 97h
    xchg    eax, ebx

unk_D79   db 0FEh ; CODE XREF: seg000:loc_D17
          db 0BEh
```

```
          db 0E1h
          db  82h


loc_D7D: ; CODE XREF: seg000:0000000000000DB3
    cwde

    db    7
    db  5Ch ; \
    db  10h
    db  73h ; s
    db 0A9h
    db  2Bh ; +
    db  9Fh


loc_D85: ; CODE XREF: seg000:0000000000000DD1
    dec     dh
    jnz     short near ptr loc_DD3+3
    mov     ds:7C1758CB282EF9BFh, al
    sal     ch, 91h
    rol     dword ptr [rbx+7Fh], cl
    fbstp   tbyte ptr [rcx+2]
    repne mov al, ds:4BFAB3C3ECF2BE13h
    pushfq
    imul    edx, [rbx+rsi*8+3B484EE9h], 8EDC09C6h
    cmp     [rax], al
    jg      short loc_D7D
    xor     [rcx-638C1102h], edx
    test    eax, 14E3AD7h
    insd

    db  38h ; 8
    db  80h
    db 0C3h


loc_DC5: ; CODE XREF: seg000:0000000000000D57
         ; seg000:0000000000000DD8
    cmp     ah, [rsi+rdi*2+527C01D3h]
    sbb     eax, 5FC631F0h
    jnb     short loc_D85
```

```
loc_DD3: ; CODE XREF: seg000:0000000000000D87
    call    near ptr 0FFFFFFFFC03919C7h
    loope   near ptr loc_DC5+3
    sbb     al, 0C8h
    std
```

Listing 28.3: random noise (ARM in ARM mode)

```
BLNE    0xFE16A9D8
BGE     0x1634D0C
SVCCS   0x450685
STRNVT  R5, [PC],#-0x964
LDCGE   p6, c14, [R0],#0x168
STCCSL  p9, c9, [LR],#0x14C
CMNHIP  PC, R10,LSL#22
FLDMIADNV LR!, {D4}
MCR     p5, 2, R2,c15,c6, 4
BLGE    0x1139558
BLGT    0xFF9146E4
STRNEB  R5, [R4],#0xCA2
STMNEIB R5, {R0,R4,R6,R7,R9-SP,PC}
STMIA   R8, {R0,R2-R4,R7,R8,R10,SP,LR}^
STRB    SP, [R8],PC,ROR#18
LDCCS   p9, c13, [R6,#0x1BC]
LDRGE   R8, [R9,#0x66E]
STRNEB  R5, [R8],#-0x8C3
STCCSL  p15, c9, [R7,#-0x84]
RSBLS   LR, R2, R11,ASR LR
SVCGT   0x9B0362
SVCGT   0xA73173
STMNEDB R11!, {R0,R1,R4-R6,R8,R10,R11,SP}
STR     R0, [R3],#-0xCE4
LDCGT   p15, c8, [R1,#0x2CC]
LDRCCB  R1, [R11],-R7,ROR#30
BLLT    0xFED9D58C
BL      0x13E60F4
LDMVSIB R3!, {R1,R4-R7}^
USATNE  R10, #7, SP,LSL#11
LDRGEB  LR, [R1],#0xE56
STRPLT  R9, [LR],#0x567
LDRLT   R11, [R1],#-0x29B
SVCNV   0x12DB29
MVNNVS  R5, SP,LSL#25
LDCL    p8, c14, [R12,#-0x288]
```

```
STCNEL  p2, c6, [R6,#-0xBC]!
SVCNV   0x2E5A2F
BLX     0x1A8C97E
TEQGE   R3, #0x1100000
STMLSIA R6, {R3,R6,R10,R11,SP}
BICPLS  R12, R2, #0x5800
BNE     0x7CC408
TEQGE   R2, R4,LSL#20
SUBS    R1, R11, #0x28C
BICVS   R3, R12, R7,ASR R0
LDRMI   R7, [LR],R3,LSL#21
BLMI    0x1A79234
STMVCDB R6, {R0-R3,R6,R7,R10,R11}
EORMI   R12, R6, #0xC5
MCRRCS  p1, 0xF, R1,R3,c2
```

Listing 28.4: random noise (ARM in Thumb mode)

```
    LSRS    R3, R6, #0x12
    LDRH    R1, [R7,#0x2C]
    SUBS    R0, #0x55 ; 'U'
    ADR     R1, loc_3C
    LDR     R2, [SP,#0x218]
    CMP     R4, #0x86
    SXTB    R7, R4
    LDR     R4, [R1,#0x4C]
    STR     R4, [R4,R2]
    STR     R0, [R6,#0x20]
    BGT     0xFFFFFF72
    LDRH    R7, [R2,#0x34]
    LDRSH   R0, [R2,R4]
    LDRB    R2, [R7,R2]

    DCB 0x17
    DCB 0xED

    STRB    R3, [R1,R1]
    STR     R5, [R0,#0x6C]
    LDMIA   R3, {R0-R5,R7}
    ASRS    R3, R2, #3
    LDR     R4, [SP,#0x2C4]
    SVC     0xB5
    LDR     R6, [R1,#0x40]
    LDR     R5, =0xB2C5CA32
```

```
    STMIA   R6, {R1-R4,R6}
    LDR     R1, [R3,#0x3C]
    STR     R1, [R5,#0x60]
    BCC     0xFFFFFF70
    LDR     R4, [SP,#0x1D4]
    STR     R5, [R5,#0x40]
    ORRS    R5, R7

loc_3C ; DATA XREF: ROM:00000006
    B       0xFFFFFF98

    ASRS    R4, R1, #0x1E
    ADDS    R1, R3, R0
    STRH    R7, [R7,#0x30]
    LDR     R3, [SP,#0x230]
    CBZ     R6, loc_90
    MOVS    R4, R2
    LSRS    R3, R4, #0x17
    STMIA   R6!, {R2,R4,R5}
    ADDS    R6, #0x42 ; 'B'
    ADD     R2, SP, #0x180
    SUBS    R5, R0, R6
    BCC     loc_B0
    ADD     R2, SP, #0x160
    LSLS    R5, R0, #0x1A
    CMP     R7, #0x45
    LDR     R4, [R4,R5]

    DCB 0x2F ; /
    DCB 0xF4

    B       0xFFFFFD18

    ADD     R4, SP, #0x2C0
    LDR     R1, [SP,#0x14C]
    CMP     R4, #0xEE

    DCB  0xA
    DCB 0xFB

    STRH    R7, [R5,#0xA]
    LDR     R3, loc_78
```

```
    DCB 0xBE ; -
    DCB 0xFC

    MOVS    R5, #0x96

    DCB 0x4F ; O
    DCB 0xEE

    B       0xFFFFFAE6

    ADD     R3, SP, #0x110

loc_78 ; DATA XREF: ROM:0000006C
    STR     R1, [R3,R6]
    LDMIA   R3!, {R2,R5-R7}
    LDRB    R2, [R4,R2]
    ASRS    R4, R0, #0x13
    BKPT    0xD1
    ADDS    R5, R0, R6
    STR     R5, [R3,#0x58]
```

Listing 28.5: random noise(MIPS little endian)

```
    lw      $t9, 0xCB3($t5)
    sb      $t5, 0x3855($t0)
    sltiu   $a2, $a0, -0x657A
    ldr     $t4, -0x4D99($a2)
    daddi   $s0, $s1, 0x50A4
    lw      $s7, -0x2353($s4)
    bgtzl   $a1, 0x17C5C

    .byte 0x17
    .byte 0xED
    .byte 0x4B  # K
    .byte 0x54  # T

    lwc2    $31, 0x66C5($sp)
    lwu     $s1, 0x10D3($a1)
    ldr     $t6, -0x204B($zero)
    lwc1    $f30, 0x4DBE($s2)
    daddiu  $t1, $s1, 0x6BD9
    lwu     $s5, -0x2C64($v1)
    cop0    0x13D642D
    bne     $gp, $t4, 0xFFFF9EF0
```

```
    lh      $ra, 0x1819($s1)
    sdl     $fp, -0x6474($t8)
    jal     0x78C0050
    ori     $v0, $s2, 0xC634
    blez    $gp, 0xFFFEA9D4
    swl     $t8, -0x2CD4($s2)
    sltiu   $a1, $k0, 0x685
    sdc1    $f15, 0x5964($at)
    sw      $s0, -0x19A6($a1)
    sltiu   $t6, $a3, -0x66AD
    lb      $t7, -0x4F6($t3)
    sd      $fp, 0x4B02($a1)

    .byte 0x96
    .byte 0x25  # %
    .byte 0x4F  # O
    .byte 0xEE

    swl     $a0, -0x1AC9($k0)
    lwc2    $4, 0x5199($ra)
    bne     $a2, $a0, 0x17308

    .byte 0xD1
    .byte 0xBE
    .byte 0x85
    .byte 0x19

    swc2    $8, 0x659D($a2)
    swc1    $f8, -0x2691($s6)
    sltiu   $s6, $t4, -0x2691
    sh      $t9, -0x7992($t4)
    bne     $v0, $t0, 0x163A4
    sltiu   $a3, $t2, -0x60DF
    lbu     $v0, -0x11A5($v1)
    pref    0x1B, 0x362($gp)
    pref    7, 0x3173($sp)
    blez    $t1, 0xB678
    swc1    $f3, flt_CE4($zero)
    pref    0x11, -0x704D($t4)
    ori     $k1, $s2, 0x1F67
    swr     $s6, 0x7533($sp)
    swc2    $15, -0x67F4($k0)
    ldl     $s3, 0xF2($t7)
```

```
    bne     $s7, $a3, 0xFFFE973C
    sh      $s1, -0x11AA($a2)
    bnel    $a1, $t6, 0xFFFE566C
    sdr     $s1, -0x4D65($zero)
    sd      $s2, -0x24D7($t8)
    scd     $s4, 0x5C8D($t7)

    .byte 0xA2
    .byte 0xE8
    .byte 0x5C  # \
    .byte 0xED

    bgtz    $t3, 0x189A0
    sd      $t6, 0x5A2F($t9)
    sdc2    $10, 0x3223($k1)
    sb      $s3, 0x5744($t9)
    lwr     $a2, 0x2C48($a0)
    beql    $fp, $s2, 0xFFFF3258
```

It is also important to keep in mind that cleverly constructed unpacking and decrypting code (including self-modifying) may looks like noise as well, nevertheless, it executes correctly.

28.3 Information entropy of average code

ent utility results[1].

(Entropy of ideally compressed (or encrypted) file is 8 bits per byte; of zero file of arbitrary size if 0 bits per byte.)

Here we can see that a code for CPU with 4-byte instructions (ARM in ARM mode and MIPS) is least effective in this sense.

28.3.1 x86

`.text` section of `ntoskrnl.exe` file from Windows 2003:

```
Entropy = 6.662739 bits per byte.

Optimum compression would reduce the size
of this 593920 byte file by 16 percent.
...
```

`.text` section of `ntoskrnl.exe` from Windows 7 x64:

[1] http://www.fourmilab.ch/random/

```
Entropy = 6.549586 bits per byte.

Optimum compression would reduce the size
of this 1685504 byte file by 18 percent.
...
```

28.3.2 ARM (Thumb)

AngryBirds Classic:

```
Entropy = 7.058766 bits per byte.

Optimum compression would reduce the size
of this 3336888 byte file by 11 percent.
...
```

28.3.3 ARM (ARM mode)

Linux Kernel 3.8.0:

```
Entropy = 6.036160 bits per byte.

Optimum compression would reduce the size
of this 6946037 byte file by 24 percent.
...
```

28.3.4 MIPS (little endian)

`.text` section of `user32.dll` from Windows NT 4:

```
Entropy = 6.098227 bits per byte.

Optimum compression would reduce the size
of this 433152 byte file by 23 percent.
....
```

Chapter 29

C++

29.1 Classes

29.1.1 Simple example

Internally, C++ classes representation is almost the same as structures representation.

Let's try an example with two variables, two constructors and one method:

```
#include <stdio.h>

class c
{
private:
    int v1;
    int v2;
public:
    c() // default ctor
    {
        v1=667;
        v2=999;
    };

    c(int a, int b) // ctor
    {
        v1=a;
        v2=b;
    };

    void dump()
```

```
    {
        printf ("%d; %d\n", v1, v2);
    };
};

int main()
{
    class c c1;
    class c c2(5,6);

    c1.dump();
    c2.dump();

    return 0;
};
```

MSVC—x86

Here is how `main()` function looks like translated into assembly language:

Listing 29.1: MSVC

```
_c2$ = -16 ; size = 8
_c1$ = -8  ; size = 8
_main PROC
    push ebp
    mov  ebp, esp
    sub  esp, 16
    lea  ecx, DWORD PTR _c1$[ebp]
    call ??0c@@QAE@XZ ; c::c
    push 6
    push 5
    lea  ecx, DWORD PTR _c2$[ebp]
    call ??0c@@QAE@HH@Z ; c::c
    lea  ecx, DWORD PTR _c1$[ebp]
    call ?dump@c@@QAEXXZ ; c::dump
    lea  ecx, DWORD PTR _c2$[ebp]
    call ?dump@c@@QAEXXZ ; c::dump
    xor  eax, eax
    mov  esp, ebp
    pop  ebp
    ret  0
_main ENDP
```

So what's going on. For each object (instance of class *c*) 8 bytes allocated, that is exactly size of 2 variables storage.

For *c1* a default argumentless constructor `??0c@@QAE@XZ` is called. For *c2* another constructor `??0c@@QAE@HH@Z` is called and two numbers are passed as arguments.

A pointer to object (*this* in C++ terminology) is passed in the `ECX` register. This is called thiscall (29.1.1) —a pointer to object passing method.

MSVC doing it using the `ECX` register. Needless to say, it is not a standardized method, other compilers could do it differently, e.g., via first function argument (like GCC).

Why these functions has so odd names? That's name mangling.

C++ class may contain several methods sharing the same name but having different arguments —that is polymorphism. And of course, different classes may own methods sharing the same name.

Name mangling enable us to encode class name + method name + all method argument types in one ASCII-string, which is to be used as internal function name. That's all because neither linker, nor DLL OS loader (mangled names may be among DLL exports as well) knows nothing about C++ or OOP[1].

`dump()` function called two times after.

Now let's see constructors' code:

Listing 29.2: MSVC

```
_this$ = -4       ; size = 4
??0c@@QAE@XZ PROC ; c::c, COMDAT
; _this$ = ecx
    push ebp
    mov  ebp, esp
    push ecx
    mov  DWORD PTR _this$[ebp], ecx
    mov  eax, DWORD PTR _this$[ebp]
    mov  DWORD PTR [eax], 667
    mov  ecx, DWORD PTR _this$[ebp]
    mov  DWORD PTR [ecx+4], 999
    mov  eax, DWORD PTR _this$[ebp]
    mov  esp, ebp
    pop  ebp
    ret  0
??0c@@QAE@XZ ENDP ; c::c

_this$ = -4 ; size = 4
_a$ = 8     ; size = 4
_b$ = 12    ; size = 4
??0c@@QAE@HH@Z PROC ; c::c, COMDAT
```

[1] Object-Oriented Programming

```
; _this$ = ecx
    push ebp
    mov  ebp, esp
    push ecx
    mov  DWORD PTR _this$[ebp], ecx
    mov  eax, DWORD PTR _this$[ebp]
    mov  ecx, DWORD PTR _a$[ebp]
    mov  DWORD PTR [eax], ecx
    mov  edx, DWORD PTR _this$[ebp]
    mov  eax, DWORD PTR _b$[ebp]
    mov  DWORD PTR [edx+4], eax
    mov  eax, DWORD PTR _this$[ebp]
    mov  esp, ebp
    pop  ebp
    ret  8
??0c@@QAE@HH@Z ENDP ; c::c
```

Constructors are just functions, they use pointer to structure in the ECX, moving the pointer into own local variable, however, it is not necessary.

From the C++ standard [16, 12.1] we know that constructors should not return any values. In fact, internally, constructors are returns pointer to the newly created object, i.e., *this*.

Now `dump()` method:

Listing 29.3: MSVC

```
_this$ = -4          ; size = 4
?dump@c@@QAEXXZ PROC ; c::dump, COMDAT
; _this$ = ecx
    push ebp
    mov  ebp, esp
    push ecx
    mov  DWORD PTR _this$[ebp], ecx
    mov  eax, DWORD PTR _this$[ebp]
    mov  ecx, DWORD PTR [eax+4]
    push ecx
    mov  edx, DWORD PTR _this$[ebp]
    mov  eax, DWORD PTR [edx]
    push eax
    push OFFSET ??_C@_07NJBDCIEC@?$CFd?$DL?5?$CFd?6?$AA@
    call _printf
    add  esp, 12
    mov  esp, ebp
    pop  ebp
    ret  0
```

```
?dump@c@@QAEXXZ ENDP ; c::dump
```

Simple enough: `dump()` taking pointer to the structure containing two *int*'s in the ECX, takes two values from it and passing it into `printf()`.

The code is much shorter if compiled with optimization (`/Ox`):

Listing 29.4: MSVC

```
??0c@@QAE@XZ PROC ; c::c, COMDAT
; _this$ = ecx
    mov  eax, ecx
    mov  DWORD PTR [eax], 667
    mov  DWORD PTR [eax+4], 999
    ret  0
??0c@@QAE@XZ ENDP ; c::c

_a$ = 8  ; size = 4
_b$ = 12 ; size = 4
??0c@@QAE@HH@Z PROC ; c::c, COMDAT
; _this$ = ecx
    mov  edx, DWORD PTR _b$[esp-4]
    mov  eax, ecx
    mov  ecx, DWORD PTR _a$[esp-4]
    mov  DWORD PTR [eax], ecx
    mov  DWORD PTR [eax+4], edx
    ret  8
??0c@@QAE@HH@Z ENDP ; c::c

?dump@c@@QAEXXZ PROC ; c::dump, COMDAT
; _this$ = ecx
    mov  eax, DWORD PTR [ecx+4]
    mov  ecx, DWORD PTR [ecx]
    push eax
    push ecx
    push OFFSET ??_C@_07NJBDCIEC@?$CFd?$DL?5?$CFd?6?$AA@
    call _printf
    add  esp, 12
    ret  0
?dump@c@@QAEXXZ ENDP ; c::dump
```

That's all. One more thing to say is the stack pointer was not corrected with `add esp, X` after constructor called. Withal, constructor has `ret 8` instead of the `RET` at the end.

This is all because here used thiscall (29.1.1) calling convention, the method of passing values through the stack, which is, together with stdcall (44.2) method, offers

to correct stack to callee rather then to caller. `ret x` instruction adding `X` to the value in the `ESP`, then passes control to the caller function.

See also section about calling conventions (44).

It is also should be noted the compiler deciding when to call constructor and destructor —but that is we already know from C++ language basics.

MSVC—x86-64

As we already know, first 4 function arguments in x86-64 are passed in `RCX`, `RDX`, `R8`, `R9` registers, all the rest—via stack. Nevertheless, *this* pointer to the object is passed in `RCX`, first method argument—in `EDX`, etc. We can see this in the `c(int a, int b)` method internals:

Listing 29.5: MSVC 2012 x64 /Ox

```
; void dump()

?dump@c@@QEAAXXZ PROC ; c::dump
    mov    r8d, DWORD PTR [rcx+4]
    mov    edx, DWORD PTR [rcx]
    lea    rcx, OFFSET FLAT:??_C@_07NJBDCIEC@?$CFd?$DL?5?$CFd?6?↙
    ↘ $AA@ ; '%d; %d'
    jmp    printf
?dump@c@@QEAAXXZ ENDP ; c::dump

; c(int a, int b)

??0c@@QEAA@HH@Z PROC ; c::c
    mov    DWORD PTR [rcx], edx   ; 1st argument: a
    mov    DWORD PTR [rcx+4], r8d ; 2nd argument: b
    mov    rax, rcx
    ret    0
??0c@@QEAA@HH@Z ENDP ; c::c

; default ctor

??0c@@QEAA@XZ PROC ; c::c
    mov    DWORD PTR [rcx], 667
    mov    DWORD PTR [rcx+4], 999
    mov    rax, rcx
    ret    0
??0c@@QEAA@XZ ENDP ; c::c
```

int data type is still 32-bit in x64 [2], so that is why 32-bit register's parts are used here.

We also see `JMP printf` instead of `RET` in the `dump()` method, that *hack* we already saw earlier: 11.1.1.

GCC—x86

It is almost the same situation in GCC 4.4.1, with a few exceptions.

Listing 29.6: GCC 4.4.1

```
     public main
main proc near

var_20 = dword ptr -20h
var_1C = dword ptr -1Ch
var_18 = dword ptr -18h
var_10 = dword ptr -10h
var_8  = dword ptr -8

    push ebp
    mov  ebp, esp
    and  esp, 0FFFFFFF0h
    sub  esp, 20h
    lea  eax, [esp+20h+var_8]
    mov  [esp+20h+var_20], eax
    call _ZN1cC1Ev
    mov  [esp+20h+var_18], 6
    mov  [esp+20h+var_1C], 5
    lea  eax, [esp+20h+var_10]
    mov  [esp+20h+var_20], eax
    call _ZN1cC1Eii
    lea  eax, [esp+20h+var_8]
    mov  [esp+20h+var_20], eax
    call _ZN1c4dumpEv
    lea  eax, [esp+20h+var_10]
    mov  [esp+20h+var_20], eax
    call _ZN1c4dumpEv
    mov  eax, 0
    leave
    retn
main endp
```

[2] Apparently, for easier porting of C/C++ 32-bit code to x64

Here we see another *name mangling* style, specific to GNU [3] It is also can be noted the pointer to object is passed as first function argument —transparently from programmer, of course.

First constructor:

```
                public _ZN1cC1Ev ; weak
_ZN1cC1Ev       proc near               ; CODE XREF: main+10

arg_0           = dword ptr  8

                push    ebp
                mov     ebp, esp
                mov     eax, [ebp+arg_0]
                mov     dword ptr [eax], 667
                mov     eax, [ebp+arg_0]
                mov     dword ptr [eax+4], 999
                pop     ebp
                retn
_ZN1cC1Ev       endp
```

What it does is just writes two numbers using pointer passed in first (and single) argument.

Second constructor:

```
                public _ZN1cC1Eii
_ZN1cC1Eii      proc near

arg_0           = dword ptr  8
arg_4           = dword ptr  0Ch
arg_8           = dword ptr  10h

                push    ebp
                mov     ebp, esp
                mov     eax, [ebp+arg_0]
                mov     edx, [ebp+arg_4]
                mov     [eax], edx
                mov     eax, [ebp+arg_0]
                mov     edx, [ebp+arg_8]
                mov     [eax+4], edx
                pop     ebp
                retn
_ZN1cC1Eii      endp
```

This is a function, analog of which could be looks like:

[3] One more document about different compilers name mangling types: [12] standards.

```
void ZN1cC1Eii (int *obj, int a, int b)
{
  *obj=a;
  *(obj+1)=b;
};
```

...and that is completely predictable.

Now `dump()` function:

```
                public _ZN1c4dumpEv
_ZN1c4dumpEv    proc near

var_18          = dword ptr -18h
var_14          = dword ptr -14h
var_10          = dword ptr -10h
arg_0           = dword ptr  8

                push    ebp
                mov     ebp, esp
                sub     esp, 18h
                mov     eax, [ebp+arg_0]
                mov     edx, [eax+4]
                mov     eax, [ebp+arg_0]
                mov     eax, [eax]
                mov     [esp+18h+var_10], edx
                mov     [esp+18h+var_14], eax
                mov     [esp+18h+var_18], offset aDD ; "%d; %d\n↙
    ↳ "
                call    _printf
                leave
                retn
_ZN1c4dumpEv    endp
```

This function in its *internal representation* has sole argument, used as pointer to the object (*this*).

Thus, if to base our judgment on these simple examples, the difference between MSVC and GCC is style of function names encoding (*name mangling*) and passing pointer to object (via the `ECX` register or via the first argument).

GCC—x86-64

The first 6 arguments, as we already know, are passed in the `RDI`, `RSI`, `RDX`, `RCX`, `R8`, `R9` [21] registers, and the pointer to *this* via first one (`RDI`) and that is what we see here. *int* data type is also 32-bit here. `JMP` instead of `RET` *hack* is also used here.

Listing 29.7: GCC 4.4.6 x64

```
; default ctor

_ZN1cC2Ev:
    mov  DWORD PTR [rdi], 667
    mov  DWORD PTR [rdi+4], 999
    ret

; c(int a, int b)

_ZN1cC2Eii:
    mov  DWORD PTR [rdi], esi
    mov  DWORD PTR [rdi+4], edx
    ret

; dump()

_ZN1c4dumpEv:
    mov  edx, DWORD PTR [rdi+4]
    mov  esi, DWORD PTR [rdi]
    xor  eax, eax
    mov  edi, OFFSET FLAT:.LC0 ; "%d; %d\n"
    jmp  printf
```

29.1.2 Class inheritance

It can be said about inherited classes that it is simple structure we already considered, but extending in inherited classes.

Let's take simple example:

```
#include <stdio.h>

class object
{
    public:
        int color;
        object() { };
        object (int color) { this->color=color; };
        void print_color() { printf ("color=%d\n", color); };
};

class box : public object
{
```

```
    private:
        int width, height, depth;
    public:
        box(int color, int width, int height, int depth)
        {
            this->color=color;
            this->width=width;
            this->height=height;
            this->depth=depth;
        };
        void dump()
        {
            printf ("this is box. color=%d, width=%d, height=%d,↙
    ↳  depth=%d\n", color, width, height, depth);
        };
};

class sphere : public object
{
private:
    int radius;
public:
    sphere(int color, int radius)
    {
        this->color=color;
        this->radius=radius;
    };
    void dump()
    {
        printf ("this is sphere. color=%d, radius=%d\n", color, ↙
    ↳ radius);
    };
};

int main()
{
    box b(1, 10, 20, 30);
    sphere s(2, 40);

    b.print_color();
    s.print_color();

    b.dump();
```

```
    s.dump();

    return 0;
};
```

Let's investigate generated code of the `dump()` functions/methods and also `object::prin` let's see memory layout for structures-objects (as of 32-bit code).

So, `dump()` methods for several classes, generated by MSVC 2008 with `/Ox` and `/Ob0` options [4]

Listing 29.8: Optimizing MSVC 2008 /Ob0

```
??_C@_09GCEDOLPA@color?$DN?$CFd?6?$AA@ DB 'color=%d', 0aH, 00H ;↙
    ↳ 'string'
?print_color@object@@QAEXXZ PROC ; object::print_color, COMDAT
; _this$ = ecx
    mov  eax, DWORD PTR [ecx]
    push eax

; 'color=%d', 0aH, 00H
    push OFFSET ??_C@_09GCEDOLPA@color?$DN?$CFd?6?$AA@
    call _printf
    add  esp, 8
    ret  0
?print_color@object@@QAEXXZ ENDP ; object::print_color
```

Listing 29.9: Optimizing MSVC 2008 /Ob0

```
?dump@box@@QAEXXZ PROC ; box::dump, COMDAT
; _this$ = ecx
    mov  eax, DWORD PTR [ecx+12]
    mov  edx, DWORD PTR [ecx+8]
    push eax
    mov  eax, DWORD PTR [ecx+4]
    mov  ecx, DWORD PTR [ecx]
    push edx
    push eax
    push ecx

; 'this is box. color=%d, width=%d, height=%d, depth=%d', 0aH, ↙
    ↳ 00H ; 'string'
    push OFFSET ??_C@_0DG@NCNGAADL@this?5is?5box?4?5color?$DN?↙
    ↳ $CFd?0?5width?$DN?$CFd?0@
```

[4] `/Ob0` options means inline expansion disabling since function inlining right into the code where the function is called will make our experiment harder

```
    call _printf
    add  esp, 20
    ret  0
?dump@box@@QAEXXZ ENDP ; box::dump
```

Listing 29.10: Optimizing MSVC 2008 /Ob0

```
?dump@sphere@@QAEXXZ PROC ; sphere::dump, COMDAT
; _this$ = ecx
    mov  eax, DWORD PTR [ecx+4]
    mov  ecx, DWORD PTR [ecx]
    push eax
    push ecx

; 'this is sphere. color=%d, radius=%d', 0aH, 00H
    push OFFSET ??_C@_0CF@EFEDJLDC@this?5is?5sphere?4?5color?$DN↙
    ↳ ?$CFd?0?5radius@
    call _printf
    add  esp, 12
    ret  0
?dump@sphere@@QAEXXZ ENDP ; sphere::dump
```

So, here is memory layout:
(base class *object*)

offset	description
+0x0	int color

(inherited classes)
box:

offset	description
+0x0	int color
+0x4	int width
+0x8	int height
+0xC	int depth

sphere:

offset	description
+0x0	int color
+0x4	int radius

Let's see `main()` function body:

Listing 29.11: Optimizing MSVC 2008 /Ob0

```
PUBLIC _main
_TEXT SEGMENT
_s$ = -24 ; size = 8
_b$ = -16 ; size = 16
_main PROC
    sub  esp, 24
    push 30
    push 20
    push 10
    push 1
    lea  ecx, DWORD PTR _b$[esp+40]
    call ??0box@@QAE@HHHH@Z ; box::box
    push 40
    push 2
    lea  ecx, DWORD PTR _s$[esp+32]
    call ??0sphere@@QAE@HH@Z ; sphere::sphere
    lea  ecx, DWORD PTR _b$[esp+24]
    call ?print_color@object@@QAEXXZ ; object::print_color
    lea  ecx, DWORD PTR _s$[esp+24]
    call ?print_color@object@@QAEXXZ ; object::print_color
    lea  ecx, DWORD PTR _b$[esp+24]
    call ?dump@box@@QAEXXZ ; box::dump
    lea  ecx, DWORD PTR _s$[esp+24]
    call ?dump@sphere@@QAEXXZ ; sphere::dump
    xor  eax, eax
    add  esp, 24
    ret  0
_main ENDP
```

Inherited classes must always add their fields after base classes' fields, so to make possible for base class methods to work with their fields.

When `object::print_color()` method is called, a pointers to both *box* object and *sphere* object are passed as `this`, it can work with these objects easily since *color* field in these objects is always at the pinned address (at *+0x0* offset).

It can be said, `object::print_color()` method is agnostic in relation to input object type as long as fields will be *pinned* at the same addresses, and this condition is always true.

And if you create inherited class of the e.g. *box* class, compiler will add new fields after *depth* field, leaving *box* class fields at the pinned addresses.

Thus, `box::dump()` method will work fine accessing *color*/*width*/*height*/*depths* fields always pinned on known addresses.

GCC-generated code is almost likewise, with the sole exception of `this` pointer passing (as it was described above, it passing as first argument instead of the `ECX`

registers.

29.1.3 Encapsulation

Encapsulation is data hiding in the *private* sections of class, e.g. to allow access to them only from this class methods, but no more than.

However, are there any marks in code about the fact that some field is private and some other —not?

No, there are no such marks.

Let's try simple example:

```
#include <stdio.h>

class box
{
    private:
        int color, width, height, depth;
    public:
        box(int color, int width, int height, int depth)
        {
            this->color=color;
            this->width=width;
            this->height=height;
            this->depth=depth;
        };
        void dump()
        {
            printf ("this is box. color=%d, width=%d, height=%d,⇙
    ↪  depth=%d\n", color, width, height, depth);
        };
};
```

Let's compile it again in MSVC 2008 with `/Ox` and `/Ob0` options and let's see `box::dump()` method code:

```
?dump@box@@QAEXXZ PROC ; box::dump, COMDAT
; _this$ = ecx
    mov  eax, DWORD PTR [ecx+12]
    mov  edx, DWORD PTR [ecx+8]
    push eax
    mov  eax, DWORD PTR [ecx+4]
    mov  ecx, DWORD PTR [ecx]
    push edx
    push eax
    push ecx
```

```
; 'this is box. color=%d, width=%d, height=%d, depth=%d', 0aH, ↙
    ↳ 00H
    push OFFSET ??_C@_0DG@NCNGAADL@this?5is?5box?4?5color?$DN?↙
    ↳ $CFd?0?5width?$DN?$CFd?0@
    call _printf
    add  esp, 20
    ret  0
?dump@box@@QAEXXZ ENDP ; box::dump
```

Here is a memory layout of the class:

offset	description
+0x0	int color
+0x4	int width
+0x8	int height
+0xC	int depth

All fields are private and not allowed to access from any other functions, but, knowing this layout, can we create a code modifying these fields?

So I added `hack_oop_encapsulation()` function, which, if has the body as follows, will not compile:

```
void hack_oop_encapsulation(class box * o)
{
    o->width=1; // that code can't be compiled:
                // "error C2248: 'box::width' : cannot access ↙
    ↳ private member declared in class 'box'"
};
```

Nevertheless, if to cast *box* type to *pointer to int array*, and if to modify array of the *int*-s we got, then we have success.

```
void hack_oop_encapsulation(class box * o)
{
    unsigned int *ptr_to_object=reinterpret_cast<unsigned int*>(↙
    ↳ o);
    ptr_to_object[1]=123;
};
```

This functions' code is very simple —it can be said, the function taking pointer to array of the *int*-s on input and writing *123* to the second *int*:

```
?hack_oop_encapsulation@@YAXPAVbox@@@Z PROC ; ↙
    ↳ hack_oop_encapsulation
    mov eax, DWORD PTR _o$[esp-4]
    mov DWORD PTR [eax+4], 123
```

```
    ret 0
?hack_oop_encapsulation@@YAXPAVbox@@@Z ENDP ; ↙
    ↳ hack_oop_encapsulation
```

Let's check, how it works:

```
int main()
{
    box b(1, 10, 20, 30);

    b.dump();

    hack_oop_encapsulation(&b);

    b.dump();

    return 0;
};
```

Let's run:

```
this is box. color=1, width=10, height=20, depth=30
this is box. color=1, width=123, height=20, depth=30
```

We see, encapsulation is just class fields protection only on compiling stage. C++ compiler will not allow to generate a code modifying protected fields straightforwardly, nevertheless, it is possible with the help of *dirty hacks*.

29.1.4 Multiple inheritance

Multiple inheritance is a class creation which inherits fields and methods from two or more classes.

Let's write simple example again:

```
#include <stdio.h>

class box
{
    public:
        int width, height, depth;
        box() { };
        box(int width, int height, int depth)
        {
            this->width=width;
            this->height=height;
            this->depth=depth;
```

```
        };
        void dump()
        {
            printf ("this is box. width=%d, height=%d, depth=%d\↙
    ↳ n", width, height, depth);
        };
        int get_volume()
        {
            return width * height * depth;
        };
};

class solid_object
{
    public:
        int density;
        solid_object() { };
        solid_object(int density)
        {
            this->density=density;
        };
        int get_density()
        {
            return density;
        };
        void dump()
        {
            printf ("this is solid_object. density=%d\n", ↙
    ↳ density);
        };
};

class solid_box: box, solid_object
{
    public:
        solid_box (int width, int height, int depth, int density↙
    ↳ )
        {
            this->width=width;
            this->height=height;
            this->depth=depth;
            this->density=density;
        };
```

```
        void dump()
        {
            printf ("this is solid_box. width=%d, height=%d, ↙
    ↳ depth=%d, density=%d\n", width, height, depth, density);
        };
        int get_weight() { return get_volume() * get_density(); ↙
    ↳ };
};

int main()
{
    box b(10, 20, 30);
    solid_object so(100);
    solid_box sb(10, 20, 30, 3);

    b.dump();
    so.dump();
    sb.dump();
    printf ("%d\n", sb.get_weight());

    return 0;
};
```

Let's compile it in MSVC 2008 with `/Ox` and `/Ob0` options and let's see `box::dump()`, `solid_object::dump()` and `solid_box::dump()` methods code:

Listing 29.12: Optimizing MSVC 2008 /Ob0

```
?dump@box@@QAEXXZ PROC ; box::dump, COMDAT
; _this$ = ecx
    mov  eax, DWORD PTR [ecx+8]
    mov  edx, DWORD PTR [ecx+4]
    push eax
    mov  eax, DWORD PTR [ecx]
    push edx
    push eax
; 'this is box. width=%d, height=%d, depth=%d', 0aH, 00H
    push OFFSET ??_C@_0CM@DIKPHDFI@this?5is?5box?4?5width?$DN?↙
    ↳ $CFd?0?5height?$DN?$CFd@
    call _printf
    add  esp, 16
    ret  0
?dump@box@@QAEXXZ ENDP ; box::dump
```

Listing 29.13: Optimizing MSVC 2008 /Ob0

```
?dump@solid_object@@QAEXXZ PROC ; solid_object::dump, COMDAT
; _this$ = ecx
    mov  eax, DWORD PTR [ecx]
    push eax
; 'this is solid_object. density=%d', 0aH
    push OFFSET ??_C@_0CC@KICFJINL@this?5is?5solid_object?4?5↙
    ↳ density?$DN?$CFd@
    call _printf
    add  esp, 8
    ret  0
?dump@solid_object@@QAEXXZ ENDP ; solid_object::dump
```

Listing 29.14: Optimizing MSVC 2008 /Ob0

```
?dump@solid_box@@QAEXXZ PROC ; solid_box::dump, COMDAT
; _this$ = ecx
    mov  eax, DWORD PTR [ecx+12]
    mov  edx, DWORD PTR [ecx+8]
    push eax
    mov  eax, DWORD PTR [ecx+4]
    mov  ecx, DWORD PTR [ecx]
    push edx
    push eax
    push ecx
; 'this is solid_box. width=%d, height=%d, depth=%d, density=%d↙
    ↳ ', 0aH
    push OFFSET ??_C@_0DO@HNCNIHNN@this?5is?5solid_box?4?5width?↙
    ↳ $DN?$CFd?0?5hei@
    call _printf
    add  esp, 20
    ret  0
?dump@solid_box@@QAEXXZ ENDP ; solid_box::dump
```

So, the memory layout for all three classes is:
box class:

offset	description
+0x0	width
+0x4	height
+0x8	depth

solid_object class:

offset	description
+0x0	density

It can be said, *solid_box* class memory layout will be *united*:
solid_box class:

offset	description
+0x0	width
+0x4	height
+0x8	depth
+0xC	density

The code of the `box::get_volume()` and `solid_object::get_density()` methods is trivial:

Listing 29.15: Optimizing MSVC 2008 /Ob0

```
?get_volume@box@@QAEHXZ PROC ; box::get_volume, COMDAT
; _this$ = ecx
    mov  eax, DWORD PTR [ecx+8]
    imul eax, DWORD PTR [ecx+4]
    imul eax, DWORD PTR [ecx]
    ret  0
?get_volume@box@@QAEHXZ ENDP ; box::get_volume
```

Listing 29.16: Optimizing MSVC 2008 /Ob0

```
?get_density@solid_object@@QAEHXZ PROC ; solid_object::↙
    ↳ get_density, COMDAT
; _this$ = ecx
    mov eax, DWORD PTR [ecx]
    ret 0
?get_density@solid_object@@QAEHXZ ENDP ; solid_object::↙
    ↳ get_density
```

But the code of the `solid_box::get_weight()` method is much more interesting:

Listing 29.17: Optimizing MSVC 2008 /Ob0

```
?get_weight@solid_box@@QAEHXZ PROC ; solid_box::get_weight, ↙
    ↳ COMDAT
; _this$ = ecx
    push esi
    mov  esi, ecx
    push edi
    lea  ecx, DWORD PTR [esi+12]
    call ?get_density@solid_object@@QAEHXZ ; solid_object::↙
    ↳ get_density
```

```
    mov  ecx, esi
    mov  edi, eax
    call ?get_volume@box@@QAEHXZ ; box::get_volume
    imul eax, edi
    pop  edi
    pop  esi
    ret  0
?get_weight@solid_box@@QAEHXZ ENDP ; solid_box::get_weight
```

`get_weight()` just calling two methods, but for `get_volume()` it just passing pointer to `this`, and for `get_density()` it passing pointer to `this` shifted by 12 (or 0xC) bytes, and there, in the `solid_box` class memory layout, fields of the `solid_object` class are beginning.

Thus, `solid_object::get_density()` method will believe it is dealing with usual `solid_object` class, and `box::get_volume()` method will work with its three fields, believing this is usual object of the `box` class.

Thus, we can say, an object of a class, inheriting from several other classes, representing in memory *united* class, containing all inherited fields. And each inherited method called with a pointer to corresponding structure's part passed.

29.1.5 Virtual methods

Yet another simple example:

```
#include <stdio.h>

class object
{
    public:
        int color;
        object() { };
        object (int color) { this->color=color; };
        virtual void dump()
        {
            printf ("color=%d\n", color);
        };
};

class box : public object
{
    private:
        int width, height, depth;
    public:
        box(int color, int width, int height, int depth)
```

```
        {
            this->color=color;
            this->width=width;
            this->height=height;
            this->depth=depth;
        };
        void dump()
        {
            printf ("this is box. color=%d, width=%d, height=%d,↙
    ↳  depth=%d\n", color, width, height, depth);
        };
};

class sphere : public object
{
    private:
        int radius;
    public:
        sphere(int color, int radius)
        {
            this->color=color;
            this->radius=radius;
        };
        void dump()
        {
            printf ("this is sphere. color=%d, radius=%d\n", ↙
    ↳ color, radius);
        };
};

int main()
{
    box b(1, 10, 20, 30);
    sphere s(2, 40);

    object *o1=&b;
    object *o2=&s;

    o1->dump();
    o2->dump();
    return 0;
};
```

Class *object* has virtual method `dump()`, being replaced in the *box* and *sphere* class-inheritors.

If in an environment, where it is not known what type has object, as in the `main()` function in example, a virtual method `dump()` is called, somewhere, the information about its type must be stored, so to call relevant virtual method.

Let's compile it in MSVC 2008 with `/Ox` and `/Ob0` options and let's see `main()` function code:

```
_s$ = -32 ; size = 12
_b$ = -20 ; size = 20
_main PROC
    sub  esp, 32
    push 30
    push 20
    push 10
    push 1
    lea  ecx, DWORD PTR _b$[esp+48]
    call ??0box@@QAE@HHHH@Z ; box::box
    push 40
    push 2
    lea  ecx, DWORD PTR _s$[esp+40]
    call ??0sphere@@QAE@HH@Z ; sphere::sphere
    mov  eax, DWORD PTR _b$[esp+32]
    mov  edx, DWORD PTR [eax]
    lea  ecx, DWORD PTR _b$[esp+32]
    call edx
    mov  eax, DWORD PTR _s$[esp+32]
    mov  edx, DWORD PTR [eax]
    lea  ecx, DWORD PTR _s$[esp+32]
    call edx
    xor  eax, eax
    add  esp, 32
    ret  0
_main ENDP
```

Pointer to the `dump()` function is taken somewhere from object. Where the address of new method would be written there? Only somewhere in constructors: there is no other place since nothing more is called in the `main()` function. [5]

Let's see *box* class constructor's code:

```
??_R0?AVbox@@@8 DD FLAT:??_7type_info@@6B@ ; box `RTTI Type ↙
    ↳ Descriptor'
    DD    00H
```

[5] About pointers to functions, read more in relevant section:(20)

```
    DB    '.?AVbox@@', 00H

??_R1A@?0A@EA@box@@8 DD FLAT:??_R0?AVbox@@@8 ; box::‘RTTI Base ↙
    ↳ Class Descriptor at (0,-1,0,64)’
    DD    01H
    DD    00H
    DD    0ffffffffH
    DD    00H
    DD    040H
    DD    FLAT:??_R3box@@8

??_R2box@@8 DD    FLAT:??_R1A@?0A@EA@box@@8 ; box::‘RTTI Base ↙
    ↳ Class Array’
    DD    FLAT:??_R1A@?0A@EA@object@@8

??_R3box@@8 DD    00H ; box::‘RTTI Class Hierarchy Descriptor’
    DD    00H
    DD    02H
    DD    FLAT:??_R2box@@8

??_R4box@@6B@ DD 00H ; box::‘RTTI Complete Object Locator’
    DD    00H
    DD    00H
    DD    FLAT:??_R0?AVbox@@@8
    DD    FLAT:??_R3box@@8

??_7box@@6B@ DD    FLAT:??_R4box@@6B@ ; box::‘vftable’
    DD    FLAT:?dump@box@@UAEXXZ

_color$ = 8   ; size = 4
_width$ = 12  ; size = 4
_height$ = 16 ; size = 4
_depth$ = 20  ; size = 4
??0box@@QAE@HHHH@Z PROC ; box::box, COMDAT
; _this$ = ecx
    push esi
    mov  esi, ecx
    call ??0object@@QAE@XZ ; object::object
    mov  eax, DWORD PTR _color$[esp]
    mov  ecx, DWORD PTR _width$[esp]
    mov  edx, DWORD PTR _height$[esp]
    mov  DWORD PTR [esi+4], eax
    mov  eax, DWORD PTR _depth$[esp]
```

```
    mov  DWORD PTR [esi+16], eax
    mov  DWORD PTR [esi], OFFSET ??_7box@@6B@
    mov  DWORD PTR [esi+8], ecx
    mov  DWORD PTR [esi+12], edx
    mov  eax, esi
    pop  esi
    ret  16
??0box@@QAE@HHHH@Z ENDP ; box::box
```

Here we see slightly different memory layout: the first field is a pointer to some table `box::'vftable'` (name was set by MSVC compiler).

In this table we see a link to the table named `box::'RTTI Complete Object Locator'` and also a link to the `box::dump()` method. So this is named virtual methods table and RTTI[6]. Table of virtual methods contain addresses of methods and RTTI table contain information about types. By the way, RTTI-tables are the tables enumerated while calling to *dynamic_cast* and *typeid* in C++. You can also see here class name as plain text string. Thus, a method of base *object* class may call virtual method *object::dump()*, which in turn, will call a method of inherited class since that information is present right in the object's structure.

Some additional CPU time needed for enumerating these tables and finding right virtual method address, thus virtual methods are widely considered as slightly slower than common methods.

In GCC-generated code RTTI-tables constructed slightly differently.

29.2 ostream

Let's start again with a "hello world" example, but now will use ostream:

```
#include <iostream>

int main()
{
    std::cout << "Hello, world!\n";
}
```

Almost any C++ textbook tells that « operation can be replaced (overloaded) for other types. That is what is done in ostream. We see that `operator«` is called for ostream:

Listing 29.18: MSVC 2012 (reduced listing)

```
$SG37112 DB 'Hello, world!', 0aH, 00H
```

[6] Run-time type information

```
_main PROC
    push OFFSET $SG37112
    push OFFSET ?cout@std@@3V?$basic_ostream@DU?↙
    ↳ $char_traits@D@std@@@1@A ; std::cout
    call ??$?6U?$char_traits@D@std@@@std@@YAAAV?↙
    ↳ $basic_ostream@DU?$char_traits@D@std@@@0@AAV10@PBD@Z ; std↙
    ↳ ::operator<<<std::char_traits<char> >
    add  esp, 8
    xor  eax, eax
    ret  0
_main ENDP
```

Let's modify the example:

```
#include <iostream>

int main()
{
    std::cout << "Hello, " << "world!\n";
}
```

And again, from many C++ textbooks we know that the result of each `operator<<` in ostream is forwarded to the next one. Indeed:

Listing 29.19: MSVC 2012

```
$SG37112 DB 'world!', 0aH, 00H
$SG37113 DB 'Hello, ', 00H

_main PROC
    push OFFSET $SG37113 ; 'Hello, '
    push OFFSET ?cout@std@@3V?$basic_ostream@DU?↙
    ↳ $char_traits@D@std@@@1@A ; std::cout
    call ??$?6U?$char_traits@D@std@@@std@@YAAAV?↙
    ↳ $basic_ostream@DU?$char_traits@D@std@@@0@AAV10@PBD@Z ; std↙
    ↳ ::operator<<<std::char_traits<char> >
    add  esp, 8

    push OFFSET $SG37112 ; 'world!'
    push eax             ; result of previous function execution
    call ??$?6U?$char_traits@D@std@@@std@@YAAAV?↙
    ↳ $basic_ostream@DU?$char_traits@D@std@@@0@AAV10@PBD@Z ; std↙
    ↳ ::operator<<<std::char_traits<char> >
    add  esp, 8

    xor  eax, eax
```

```
    ret  0
_main ENDP
```

If to replace operator« by f(), that code can be rewritten as:

```
f(f(std::cout, "Hello, "), "world!")
```

GCC generates almost the same code as MSVC.

29.3 References

In C++, references are pointers (9) as well, but they are called *safe*, because it is harder to make a mistake while dealing with them [16, 8.3.2]. For example, reference must always be pointing to the object of corresponding type and cannot be NULL [6, 8.6]. Even more than that, reference cannot be changed, it is impossible to point it to another object (reseat) [6, 8.5].

If we will try to change the pointers example (9) to use references instead of pointers:

```
void f2 (int x, int y, int & sum, int & product)
{
        sum=x+y;
        product=x*y;
};
```

Then we'll figure out the compiled code is just the same as in pointers example (9):

Listing 29.20: Optimizing MSVC 2010

```
_x$ = 8                                                 ; size =↙
    ↘  4
_y$ = 12                                                ; size =↙
    ↘  4
_sum$ = 16                                              ; size =↙
    ↘  4
_product$ = 20                                          ; size =↙
    ↘  4
?f2@@YAXHHAAH0@Z PROC                                   ; f2
        mov     ecx, DWORD PTR _y$[esp-4]
        mov     eax, DWORD PTR _x$[esp-4]
        lea     edx, DWORD PTR [eax+ecx]
        imul eax, ecx
        mov ecx, DWORD PTR _product$[esp-4]
        push esi
```

```
        mov     esi, DWORD PTR _sum$[esp]
        mov     DWORD PTR [esi], edx
        mov     DWORD PTR [ecx], eax
        pop     esi
        ret     0
?f2@@YAXHHAAH0@Z ENDP                                   ; f2
```

(A reason why C++ functions has such strange names, is described here: 29.1.1.)

29.4 STL

N.B.: all examples here were checked only in 32-bit environment. x64 wasn't checked.

29.4.1 std::string

Internals

Many string libraries ([37, 2.2]) implements structure containing pointer to the buffer containing string, a variable always containing current string length (that is very convenient for many functions: [37, 2.2.1]) and a variable containing current buffer size. A string in buffer is usually terminated with zero: in order to be able to pass a pointer to a buffer into the functions taking usual C ASCIIZ-string.

It is not specified in the C++ standard ([16]) how std::string should be implemented, however, it is usually implemented as described above.

By standard, std::string is not a class (as QString in Qt, for instance) but template, this is done in order to support various character types: at least char and wchar_t.

There are no assembly listings, because std::string internals in MSVC and GCC can be illustrated without them.

MSVC MSVC implementation may store buffer in place instead of pointer to buffer (if the string is shorter than 16 symbols).

This mean that short string will occupy at least $16 + 4 + 4 = 24$ bytes in 32-bit environment or at least $16 + 8 + 8 = 32$ bytes in 64-bit, and if the string is longer than 16 characters, add also length of the string itself.

Listing 29.21: example for MSVC

```
#include <string>
#include <stdio.h>

struct std_string
{
    union
    {
```

```
        char buf[16];
        char* ptr;
    } u;
    size_t size;     // AKA 'Mysize' in MSVC
    size_t capacity; // AKA 'Myres' in MSVC
};

void dump_std_string(std::string s)
{
    struct std_string *p=(struct std_string*)&s;
    printf ("[%s] size:%d capacity:%d\n", p->size>16 ? p->u.ptr ↙
    ↳ : p->u.buf, p->size, p->capacity);
};

int main()
{
    std::string s1="short string";
    std::string s2="string longer that 16 bytes";

    dump_std_string(s1);
    dump_std_string(s2);

    // that works without using c_str()
    printf ("%s\n", &s1);
    printf ("%s\n", s2);
};
```

Almost everything is clear from the source code.

Couple notes:

If the string is shorter than 16 symbols, a buffer for the string will not be allocated in the heap. This is convenient because in practice, large amount of strings are short indeed. Apparently, Microsoft developers chose 16 characters as a good balance.

Very important thing here is in the end of main() functions: I'm not using c_str() method, nevertheless, if to compile the code and run, both strings will be appeared in the console!

This is why it works.

The string is shorter than 16 characters and buffer with the string is located in the beginning of std::string object (it can be treated just as structure). printf() treats pointer as a pointer to the null-terminated array of characters, hence it works.

Second string (longer than 16 characters) printing is even more dangerous: it is typical programmer's mistake (or typo) to forget to write c_str(). This works because at the moment a pointer to buffer is located at the start of structure. This may left unnoticed for a long span of time: until a longer string will appear there, then a process will crash.

GCC GCC implementation of a structure has one more variable—reference count.

One interesting fact is that a pointer to std::string instance of class points not to beginning of the structure, but to the pointer to buffer. In libstdc++-v3\include\bits\basic_strin we may read that it was made for convenient debugging:

```
   *  The reason you want _M_data pointing to the character %↙
    ↳ array and
   *  not the _Rep is so that the debugger can see the string
   *  contents. (Probably we should add a non-inline member to ↙
    ↳ get
   *  the _Rep for the debugger to use, so users can check the ↙
    ↳ actual
   *  string length.)
```

basic_string.h source code

I considering this in my example:

Listing 29.22: example for GCC

```
#include <string>
#include <stdio.h>

struct std_string
{
    size_t length;
    size_t capacity;
    size_t refcount;
};

void dump_std_string(std::string s)
{
    char *p1=*(char**)&s; // GCC type checking workaround
    struct std_string *p2=(struct std_string*)(p1-sizeof(struct ↙
    ↳ std_string));
    printf ("[%s] size:%d capacity:%d\n", p1, p2->length, p2->↙
    ↳ capacity);
};

int main()
{
    std::string s1="short string";
    std::string s2="string longer that 16 bytes";

    dump_std_string(s1);
    dump_std_string(s2);
```

```
    // GCC type checking workaround:
    printf ("%s\n", *(char**)&s1);
    printf ("%s\n", *(char**)&s2);
};
```

A trickery should be also used to imitate mistake I already wrote above because GCC has stronger type checking, nevertheless, printf() works here without c_str() as well.

More complex example

```
#include <string>
#include <stdio.h>

int main()
{
    std::string s1="Hello, ";
    std::string s2="world!\n";
    std::string s3=s1+s2;

    printf ("%s\n", s3.c_str());
}
```

Listing 29.23: MSVC 2012

```
$SG39512 DB 'Hello, ', 00H
$SG39514 DB 'world!', 0aH, 00H
$SG39581 DB '%s', 0aH, 00H

_s2$ = -72 ; size = 24
_s3$ = -48 ; size = 24
_s1$ = -24 ; size = 24
_main PROC
    sub  esp, 72

    push 7
    push OFFSET $SG39512
    lea  ecx, DWORD PTR _s1$[esp+80]
    mov  DWORD PTR _s1$[esp+100], 15
    mov  DWORD PTR _s1$[esp+96], 0
    mov  BYTE PTR _s1$[esp+80], 0
```

```
    call ?assign@?$basic_string@DU?$char_traits@D@std@@V?↙
    ↳ $allocator@D@2@@std@@QAEAAV12@PBDI@Z ; std::basic_string<↙
    ↳ char,std::char_traits<char>,std::allocator<char> >::assign

    push 7
    push OFFSET $SG39514
    lea  ecx, DWORD PTR _s2$[esp+80]
    mov  DWORD PTR _s2$[esp+100], 15
    mov  DWORD PTR _s2$[esp+96], 0
    mov  BYTE PTR _s2$[esp+80], 0
    call ?assign@?$basic_string@DU?$char_traits@D@std@@V?↙
    ↳ $allocator@D@2@@std@@QAEAAV12@PBDI@Z ; std::basic_string<↙
    ↳ char,std::char_traits<char>,std::allocator<char> >::assign

    lea  eax, DWORD PTR _s2$[esp+72]
    push eax
    lea  eax, DWORD PTR _s1$[esp+76]
    push eax
    lea  eax, DWORD PTR _s3$[esp+80]
    push eax
    call ??$?HDU?$char_traits@D@std@@V?$allocator@D@1@@std@@YA?↙
    ↳ AV?$basic_string@DU?$char_traits@D@std@@V?↙
    ↳ $allocator@D@2@@0@ABV10@0@Z ; std::operator+<char,std::↙
    ↳ char_traits<char>,std::allocator<char> >

    ; inlined c_str() method:
    cmp  DWORD PTR _s3$[esp+104], 16
    lea  eax, DWORD PTR _s3$[esp+84]
    cmovae eax, DWORD PTR _s3$[esp+84]

    push eax
    push OFFSET $SG39581
    call _printf
    add  esp, 20

    cmp  DWORD PTR _s3$[esp+92], 16
    jb   SHORT $LN119@main
    push DWORD PTR _s3$[esp+72]
    call ??3@YAXPAX@Z                ; operator delete
    add  esp, 4
$LN119@main:
    cmp  DWORD PTR _s2$[esp+92], 16
    mov  DWORD PTR _s3$[esp+92], 15
```

```
    mov  DWORD PTR _s3$[esp+88], 0
    mov  BYTE PTR _s3$[esp+72], 0
    jb   SHORT $LN151@main
    push DWORD PTR _s2$[esp+72]
    call ??3@YAXPAX@Z                ; operator delete
    add  esp, 4
$LN151@main:
    cmp  DWORD PTR _s1$[esp+92], 16
    mov  DWORD PTR _s2$[esp+92], 15
    mov  DWORD PTR _s2$[esp+88], 0
    mov  BYTE PTR _s2$[esp+72], 0
    jb   SHORT $LN195@main
    push DWORD PTR _s1$[esp+72]
    call ??3@YAXPAX@Z                ; operator delete
    add  esp, 4
$LN195@main:
    xor  eax, eax
    add  esp, 72
    ret  0
_main ENDP
```

Compiler not constructing strings statically: how it is possible anyway if buffer should be located in the heap? Usual ASCIIZ strings are stored in the data segment instead, and later, at the moment of execution, with the help of "assign" method, s1 and s2 strings are constructed. With the help of `operator+`, s3 string is constructed.

Please note that there are no call to c_str() method, because, its code is tiny enough so compiler inlined it right here: if the string is shorter than 16 characters, a pointer to buffer is leaved in EAX register, and an address of the string buffer located in the heap is fetched otherwise.

Next, we see calls to the 3 destructors, and they are called if string is longer than 16 characters: then a buffers in the heap should be freed. Otherwise, since all three std::string objects are stored in the stack, they are freed automatically, upon function finish.

As a consequence, short strings processing is faster because of lesser heap accesses.

GCC code is even simpler (because GCC way, as I mentioned above, is not to store shorter string right in the structure):

Listing 29.24: GCC 4.8.1

```
.LC0:
    .string "Hello, "
.LC1:
    .string "world!\n"
main:
```

```
    push ebp
    mov  ebp, esp
    push edi
    push esi
    push ebx
    and  esp, -16
    sub  esp, 32
    lea  ebx, [esp+28]
    lea  edi, [esp+20]
    mov  DWORD PTR [esp+8], ebx
    lea  esi, [esp+24]
    mov  DWORD PTR [esp+4], OFFSET FLAT:.LC0
    mov  DWORD PTR [esp], edi

    call _ZNSsC1EPKcRKSaIcE

    mov  DWORD PTR [esp+8], ebx
    mov  DWORD PTR [esp+4], OFFSET FLAT:.LC1
    mov  DWORD PTR [esp], esi

    call _ZNSsC1EPKcRKSaIcE

    mov  DWORD PTR [esp+4], edi
    mov  DWORD PTR [esp], ebx

    call _ZNSsC1ERKSs

    mov  DWORD PTR [esp+4], esi
    mov  DWORD PTR [esp], ebx

    call _ZNSs6appendERKSs

    ; inlined c_str():
    mov  eax, DWORD PTR [esp+28]
    mov  DWORD PTR [esp], eax

    call puts

    mov  eax, DWORD PTR [esp+28]
    lea  ebx, [esp+19]
    mov  DWORD PTR [esp+4], ebx
    sub  eax, 12
    mov  DWORD PTR [esp], eax
```

```
    call _ZNSs4_Rep10_M_disposeERKSaIcE
    mov  eax, DWORD PTR [esp+24]
    mov  DWORD PTR [esp+4], ebx
    sub  eax, 12
    mov  DWORD PTR [esp], eax
    call _ZNSs4_Rep10_M_disposeERKSaIcE
    mov  eax, DWORD PTR [esp+20]
    mov  DWORD PTR [esp+4], ebx
    sub  eax, 12
    mov  DWORD PTR [esp], eax
    call _ZNSs4_Rep10_M_disposeERKSaIcE
    lea  esp, [ebp-12]
    xor  eax, eax
    pop  ebx
    pop  esi
    pop  edi
    pop  ebp
    ret
```

It can be seen that not a pointer to object is passed to destructors, but rather a place 12 bytes (or 3 words) before, i.e., pointer to the real start of the structure.

std::string as a global variable

Experienced C++ programmers may argue: a global variables of STL[7] types are in fact can be defined.

Yes, indeed:

```
#include <stdio.h>
#include <string>

std::string s="a string";

int main()
{
    printf ("%s\n", s.c_str());
};
```

Listing 29.25: MSVC 2012

```
$SG39512 DB 'a string', 00H
$SG39519 DB '%s', 0aH, 00H
```

[7] (C++) Standard Template Library: 29.4

```
_main PROC
    cmp  DWORD PTR ?s@@3V?$basic_string@DU?$char_traits@D@std@@V↙
    ↳ ?$allocator@D@2@@std@@A+20, 16
    mov  eax, OFFSET ?s@@3V?$basic_string@DU?↙
    ↳ $char_traits@D@std@@V?$allocator@D@2@@std@@A ; s
    cmovae eax, DWORD PTR ?s@@3V?$basic_string@DU?↙
    ↳ $char_traits@D@std@@V?$allocator@D@2@@std@@A
    push eax
    push OFFSET $SG39519 ; '%s'
    call _printf
    add  esp, 8
    xor  eax, eax
    ret  0
_main ENDP

??__Es@@YAXXZ PROC ; `dynamic initializer for 's'', COMDAT
    push 8
    push OFFSET $SG39512 ; 'a string'
    mov  ecx, OFFSET ?s@@3V?$basic_string@DU?↙
    ↳ $char_traits@D@std@@V?$allocator@D@2@@std@@A ; s
    call ?assign@?$basic_string@DU?$char_traits@D@std@@V?↙
    ↳ $allocator@D@2@@std@@QAEAAV12@PBDI@Z ; std::basic_string<↙
    ↳ char,std::char_traits<char>,std::allocator<char> >::assign
    push OFFSET ??__Fs@@YAXXZ ; `dynamic atexit destructor for '↙
    ↳ s''
    call _atexit
    pop  ecx
    ret  0
??__Es@@YAXXZ ENDP ; `dynamic initializer for 's''

??__Fs@@YAXXZ PROC ; `dynamic atexit destructor for 's'', COMDAT
    push ecx
    cmp  DWORD PTR ?s@@3V?$basic_string@DU?$char_traits@D@std@@V↙
    ↳ ?$allocator@D@2@@std@@A+20, 16
    jb   SHORT $LN23@dynamic
    push esi
    mov  esi, DWORD PTR ?s@@3V?$basic_string@DU?↙
    ↳ $char_traits@D@std@@V?$allocator@D@2@@std@@A
    lea  ecx, DWORD PTR $T2[esp+8]
    call ??0?$_Wrap_alloc@V?$allocator@D@std@@@std@@QAE@XZ ; std↙
    ↳ ::_Wrap_alloc<std::allocator<char> >::_Wrap_alloc<std::↙
    ↳ allocator<char> >
    push OFFSET ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?↙
```

```
    ↳ $allocator@D@2@@std@@A ; s
    lea  ecx, DWORD PTR $T2[esp+12]
    call ??$destroy@PAD@?$_Wrap_alloc@V?↙
    ↳ $allocator@D@std@@@std@@QAEXPAPAD@Z ; std::_Wrap_alloc<std↙
    ↳ ::allocator<char> >::destroy<char *>
    lea  ecx, DWORD PTR $T1[esp+8]
    call ??0?$_Wrap_alloc@V?$allocator@D@std@@@std@@QAE@XZ ; std↙
    ↳ ::_Wrap_alloc<std::allocator<char> >::_Wrap_alloc<std::↙
    ↳ allocator<char> >
    push esi
    call ??3@YAXPAX@Z ; operator delete
    add  esp, 4
    pop  esi
$LN23@dynamic:
    mov  DWORD PTR ?s@@3V?$basic_string@DU?$char_traits@D@std@@V↙
    ↳ ?$allocator@D@2@@std@@A+20, 15
    mov  DWORD PTR ?s@@3V?$basic_string@DU?$char_traits@D@std@@V↙
    ↳ ?$allocator@D@2@@std@@A+16, 0
    mov  BYTE PTR ?s@@3V?$basic_string@DU?$char_traits@D@std@@V?↙
    ↳ $allocator@D@2@@std@@A, 0
    pop  ecx
    ret  0
??__Fs@@YAXXZ ENDP ; 'dynamic atexit destructor for 's''
```

In fact, a special function with all constructors of global variables is called from CRT, before main(). More than that: with the help of atexit() another function is registered: which contain all destructors of such variables.

GCC works likewise:

Listing 29.26: GCC 4.8.1

```
main:
    push ebp
    mov  ebp, esp
    and  esp, -16
    sub  esp, 16
    mov  eax, DWORD PTR s
    mov  DWORD PTR [esp], eax
    call puts
    xor  eax, eax
    leave
    ret
.LC0:
    .string "a string"
_GLOBAL__sub_I_s:
```

```
    sub  esp, 44
    lea  eax, [esp+31]
    mov  DWORD PTR [esp+8], eax
    mov  DWORD PTR [esp+4], OFFSET FLAT:.LC0
    mov  DWORD PTR [esp], OFFSET FLAT:s
    call _ZNSsC1EPKcRKSaIcE
    mov  DWORD PTR [esp+8], OFFSET FLAT:__dso_handle
    mov  DWORD PTR [esp+4], OFFSET FLAT:s
    mov  DWORD PTR [esp], OFFSET FLAT:_ZNSsD1Ev
    call __cxa_atexit
    add  esp, 44
    ret
.LFE645:
    .size  _GLOBAL__sub_I_s, .-_GLOBAL__sub_I_s
    .section .init_array,"aw"
    .align 4
    .long  _GLOBAL__sub_I_s
    .globl s
    .bss
    .align 4
    .type  s, @object
    .size  s, 4
s:
    .zero  4
    .hidden __dso_handle
```

It even not creates separated functions for this, each destructor is passed to atexit(), one by one.

29.4.2 std::list

This is a well-known doubly-linked list: each element has two pointers, to the previous and the next elements.

This mean that a memory footprint is enlarged by 2 words for each element (8 bytes in 32-bit environment or 16 bytes in 64-bit).

This is also a circular list, meaning that the last element has a pointer to the first and vice versa: first element has a pointer to the last one.

C++ STL just append "next" and "previous" pointers to your existing structure you wish to unite into a list.

Let's work out an example with a simple 2-variable structure we want to store in the list.

Although standard C++ standard [16] does not offer how to implement it, MSVC and GCC implementations are straightforward and similar to each other, so here is only one source code for both:

```
#include <stdio.h>
#include <list>
#include <iostream>

struct a
{
    int x;
    int y;
};

struct List_node
{
    struct List_node* _Next;
    struct List_node* _Prev;
    int x;
    int y;
};

void dump_List_node (struct List_node *n)
{
    printf ("ptr=0x%p _Next=0x%p _Prev=0x%p x=%d y=%d\n",
        n, n->_Next, n->_Prev, n->x, n->y);
};

void dump_List_vals (struct List_node* n)
{
    struct List_node* current=n;

    for (;;)
    {
        dump_List_node (current);
        current=current->_Next;
        if (current==n) // end
            break;
    };
};

void dump_List_val (unsigned int *a)
{
#ifdef _MSC_VER
    // GCC implementation doesn't have "size" field
    printf ("_Myhead=0x%p, _Mysize=%d\n", a[0], a[1]);
#endif
```

```
    dump_List_vals ((struct List_node*)a[0]);
};

int main()
{
    std::list<struct a> l;

    printf ("* empty list:\n");
    dump_List_val((unsigned int*)(void*)&l);

    struct a t1;
    t1.x=1;
    t1.y=2;
    l.push_front (t1);
    t1.x=3;
    t1.y=4;
    l.push_front (t1);
    t1.x=5;
    t1.y=6;
    l.push_back (t1);

    printf ("* 3-elements list:\n");
    dump_List_val((unsigned int*)(void*)&l);

    std::list<struct a>::iterator tmp;
    printf ("node at .begin:\n");
    tmp=l.begin();
    dump_List_node ((struct List_node *)*(void**)&tmp);
    printf ("node at .end:\n");
    tmp=l.end();
    dump_List_node ((struct List_node *)*(void**)&tmp);

    printf ("* let's count from the begin:\n");
    std::list<struct a>::iterator it=l.begin();
    printf ("1st element: %d %d\n", (*it).x, (*it).y);
    it++;
    printf ("2nd element: %d %d\n", (*it).x, (*it).y);
    it++;
    printf ("3rd element: %d %d\n", (*it).x, (*it).y);
    it++;
    printf ("element at .end(): %d %d\n", (*it).x, (*it).y);

    printf ("* let's count from the end:\n");
```

```
    std::list<struct a>::iterator it2=l.end();
    printf ("element at .end(): %d %d\n", (*it2).x, (*it2).y);
    it2--;
    printf ("3rd element: %d %d\n", (*it2).x, (*it2).y);
    it2--;
    printf ("2nd element: %d %d\n", (*it2).x, (*it2).y);
    it2--;
    printf ("1st element: %d %d\n", (*it2).x, (*it2).y);

    printf ("removing last element...\n");
    l.pop_back();
    dump_List_val((unsigned int*)(void*)&l);
};
```

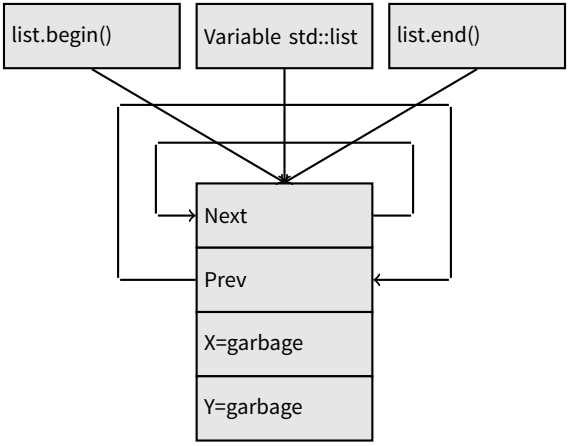
GCC

Let's start with GCC.

When we run the example, we'll see a long dump, let's work with it part by part.

```
* empty list:
ptr=0x0028fe90 _Next=0x0028fe90 _Prev=0x0028fe90 x=3 y=0
```

Here we see an empty list. Despite the fact it is empty, it has one element with garbage in x and y variables. Both "next" and "prev" pointers are pointing to the self node:



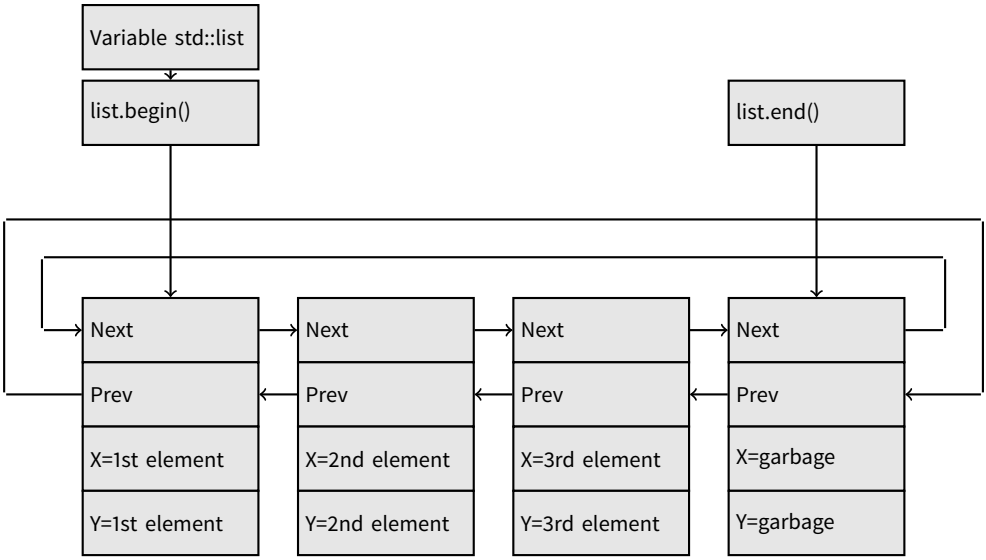
That's is the moment when .begin and .end iterators are equal to each other.

Let's push 3 elements, and the list internally will be:

```
* 3-elements list:
ptr=0x000349a0 _Next=0x00034988 _Prev=0x0028fe90 x=3 y=4
ptr=0x00034988 _Next=0x00034b40 _Prev=0x000349a0 x=1 y=2
ptr=0x00034b40 _Next=0x0028fe90 _Prev=0x00034988 x=5 y=6
ptr=0x0028fe90 _Next=0x000349a0 _Prev=0x00034b40 x=5 y=6
```

The last element is still at 0x0028fe90, it will not be moved until list disposal. It still contain random garbage in x and y fields (5 and 6). By occasion, these values are the same as in the last element, but it doesn't mean they are meaningful.

Here is how 3 elements will be stored in memory:



The variable l is always points to the first node.

.begin() and .end() iterators are not pointing to anything and not present in memory at all, but the pointers to these nodes will be returned when corresponding method is called.

Having a "garbage" element is a very popular practice in implementing doubly-linked lists. Without it, a lot of operations may become slightly more complex and, hence, slower.

Iterator in fact is just a pointer to a node. list.begin() and list.end() are just returning pointers.

```
node at .begin:
ptr=0x000349a0 _Next=0x00034988 _Prev=0x0028fe90 x=3 y=4
node at .end:
ptr=0x0028fe90 _Next=0x000349a0 _Prev=0x00034b40 x=5 y=6
```

The fact the list is circular is very helpful here: having a pointer to the first list element, i.e., that is in the l variable, it is easy to get a pointer to the last one quickly, without need to traverse whole list. Inserting element at the list end is also quick, thanks to this feature.

operator- and operator++ are just set current iterator value to the current_node->prev or current_node->next values. Reverse iterators (.rbegin, .rend) works just as the same, but in inverse way.

operator* of iterator just returns pointer to the point in the node structure, where user's structure is beginning, i.e., pointer to the very first structure element (x).

List insertion and deletion is trivial: just allocate new node (or deallocate) and fix all pointers to be valid.

That's why iterator may become invalid after element deletion: it may still point to the node already deallocated. And of course, the information from the freed node, to which iterator still points, cannot be used anymore.

The GCC implementation (as of 4.8.1) doesn't store current list size: this resulting in slow .size() method: it should traverse the whole list counting elements, because it doesn't have any other way to get the information. This mean this operation is $O(n)$, i.e., it is as slow, as how many elements present in the list.

Listing 29.27: GCC 4.8.1 -O3 -fno-inline-small-functions

```
main proc near
    push ebp
    mov  ebp, esp
    push esi
    push ebx
    and  esp, 0FFFFFFF0h
    sub  esp, 20h
    lea  ebx, [esp+10h]
    mov  dword ptr [esp], offset s ; "* empty list:"
    mov  [esp+10h], ebx
    mov  [esp+14h], ebx
    call puts
    mov  [esp], ebx
    call _Z13dump_List_valPj ; dump_List_val(uint *)
    lea  esi, [esp+18h]
    mov  [esp+4], esi
    mov  [esp], ebx
    mov  dword ptr [esp+18h], 1 ; X for new element
    mov  dword ptr [esp+1Ch], 2 ; Y for new element
    call _ZNSt4listI1aSaIS0_EE10push_frontERKS0_ ; std::list<a,↙
    ↳ std::allocator<a>>::push_front(a const&)
    mov  [esp+4], esi
    mov  [esp], ebx
    mov  dword ptr [esp+18h], 3 ; X for new element
```

```
    mov  dword ptr [esp+1Ch], 4 ; Y for new element
    call _ZNSt4listI1aSaIS0_EE10push_frontERKS0_ ; std::list<a,↙
    ↳ std::allocator<a>>::push_front(a const&)
    mov  dword ptr [esp], 10h
    mov  dword ptr [esp+18h], 5 ; X for new element
    mov  dword ptr [esp+1Ch], 6 ; Y for new element
    call _Znwj          ; operator new(uint)
    cmp  eax, 0FFFFFFF8h
    jz   short loc_80002A6
    mov  ecx, [esp+1Ch]
    mov  edx, [esp+18h]
    mov  [eax+0Ch], ecx
    mov  [eax+8], edx

loc_80002A6: ; CODE XREF: main+86
    mov  [esp+4], ebx
    mov  [esp], eax
    call _ZNSt8__detail15_List_node_base7_M_hookEPS0_ ; std::↙
    ↳ __detail::_List_node_base::_M_hook(std::__detail::↙
    ↳ _List_node_base*)
    mov  dword ptr [esp], offset a3ElementsList ; "* 3-elements ↙
    ↳ list:"
    call puts
    mov  [esp], ebx
    call _Z13dump_List_valPj ; dump_List_val(uint *)
    mov  dword ptr [esp], offset aNodeAt_begin ; "node at .begin↙
    ↳ :"
    call puts
    mov  eax, [esp+10h]
    mov  [esp], eax
    call _Z14dump_List_nodeP9List_node ; dump_List_node(↙
    ↳ List_node *)
    mov  dword ptr [esp], offset aNodeAt_end ; "node at .end:"
    call puts
    mov  [esp], ebx
    call _Z14dump_List_nodeP9List_node ; dump_List_node(↙
    ↳ List_node *)
    mov  dword ptr [esp], offset aLetSCountFromT ; "* let's ↙
    ↳ count from the begin:"
    call puts
    mov  esi, [esp+10h]
    mov  eax, [esi+0Ch]
    mov  [esp+0Ch], eax
```

```
    mov  eax, [esi+8]
    mov  dword ptr [esp+4], offset a1stElementDD ; "1st element: ↙
    ↳  %d %d\n"
    mov  dword ptr [esp], 1
    mov  [esp+8], eax
    call __printf_chk
    mov  esi, [esi] ; operator++: get ->next pointer
    mov  eax, [esi+0Ch]
    mov  [esp+0Ch], eax
    mov  eax, [esi+8]
    mov  dword ptr [esp+4], offset a2ndElementDD ; "2nd element: ↙
    ↳  %d %d\n"
    mov  dword ptr [esp], 1
    mov  [esp+8], eax
    call __printf_chk
    mov  esi, [esi] ; operator++: get ->next pointer
    mov  eax, [esi+0Ch]
    mov  [esp+0Ch], eax
    mov  eax, [esi+8]
    mov  dword ptr [esp+4], offset a3rdElementDD ; "3rd element: ↙
    ↳  %d %d\n"
    mov  dword ptr [esp], 1
    mov  [esp+8], eax
    call __printf_chk
    mov  eax, [esi] ; operator++: get ->next pointer
    mov  edx, [eax+0Ch]
    mov  [esp+0Ch], edx
    mov  eax, [eax+8]
    mov  dword ptr [esp+4], offset aElementAt_endD ; "element at ↙
    ↳  .end(): %d %d\n"
    mov  dword ptr [esp], 1
    mov  [esp+8], eax
    call __printf_chk
    mov  dword ptr [esp], offset aLetSCountFro_0 ; "* let's ↙
    ↳ count from the end:"
    call puts
    mov  eax, [esp+1Ch]
    mov  dword ptr [esp+4], offset aElementAt_endD ; "element at ↙
    ↳  .end(): %d %d\n"
    mov  dword ptr [esp], 1
    mov  [esp+0Ch], eax
    mov  eax, [esp+18h]
    mov  [esp+8], eax
```

```
    call __printf_chk
    mov  esi, [esp+14h]
    mov  eax, [esi+0Ch]
    mov  [esp+0Ch], eax
    mov  eax, [esi+8]
    mov  dword ptr [esp+4], offset a3rdElementDD ; "3rd element: ↙
    ↳ %d %d\n"
    mov  dword ptr [esp], 1
    mov  [esp+8], eax
    call __printf_chk
    mov  esi, [esi+4] ; operator--: get ->prev pointer
    mov  eax, [esi+0Ch]
    mov  [esp+0Ch], eax
    mov  eax, [esi+8]
    mov  dword ptr [esp+4], offset a2ndElementDD ; "2nd element: ↙
    ↳ %d %d\n"
    mov  dword ptr [esp], 1
    mov  [esp+8], eax
    call __printf_chk
    mov  eax, [esi+4] ; operator--: get ->prev pointer
    mov  edx, [eax+0Ch]
    mov  [esp+0Ch], edx
    mov  eax, [eax+8]
    mov  dword ptr [esp+4], offset a1stElementDD ; "1st element: ↙
    ↳ %d %d\n"
    mov  dword ptr [esp], 1
    mov  [esp+8], eax
    call __printf_chk
    mov  dword ptr [esp], offset aRemovingLastEl ; "removing ↙
    ↳ last element..."
    call puts
    mov  esi, [esp+14h]
    mov  [esp], esi
    call _ZNSt8__detail15_List_node_base9_M_unhookEv ; std:: ↙
    ↳ __detail::_List_node_base::_M_unhook(void)
    mov  [esp], esi ; void *
    call _ZdlPv ; operator delete(void *)
    mov  [esp], ebx
    call _Z13dump_List_valPj ; dump_List_val(uint *)
    mov  [esp], ebx
    call _ZNSt10_List_baseI1aSaIS0_EE8_M_clearEv ; std:: ↙
    ↳ _List_base<a,std::allocator<a>>::_M_clear(void)
    lea  esp, [ebp-8]
```

```
    xor  eax, eax
    pop  ebx
    pop  esi
    pop  ebp
    retn
main endp
```

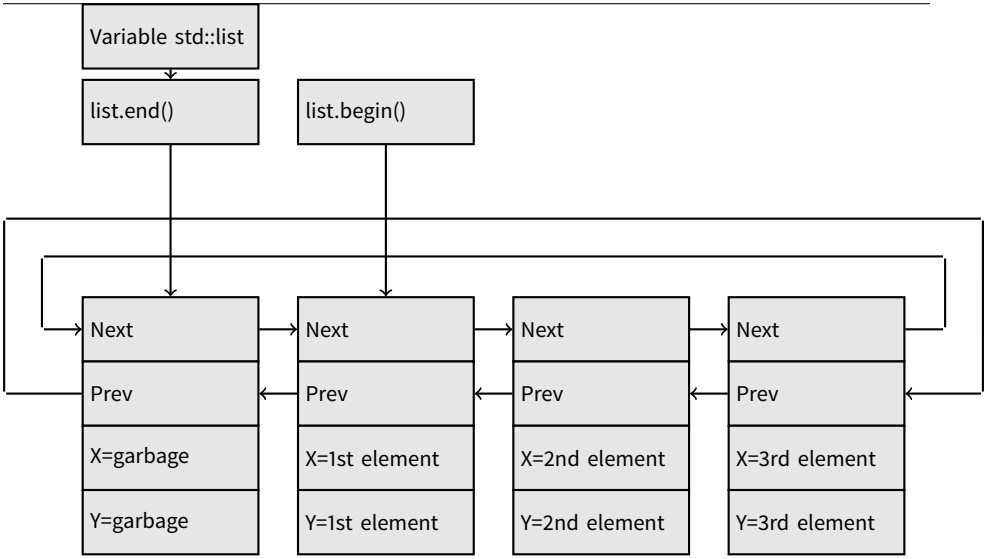
Listing 29.28: The whole output

```
* empty list:
ptr=0x0028fe90 _Next=0x0028fe90 _Prev=0x0028fe90 x=3 y=0
* 3-elements list:
ptr=0x000349a0 _Next=0x00034988 _Prev=0x0028fe90 x=3 y=4
ptr=0x00034988 _Next=0x00034b40 _Prev=0x000349a0 x=1 y=2
ptr=0x00034b40 _Next=0x0028fe90 _Prev=0x00034988 x=5 y=6
ptr=0x0028fe90 _Next=0x000349a0 _Prev=0x00034b40 x=5 y=6
node at .begin:
ptr=0x000349a0 _Next=0x00034988 _Prev=0x0028fe90 x=3 y=4
node at .end:
ptr=0x0028fe90 _Next=0x000349a0 _Prev=0x00034b40 x=5 y=6
* let's count from the begin:
1st element: 3 4
2nd element: 1 2
3rd element: 5 6
element at .end(): 5 6
* let's count from the end:
element at .end(): 5 6
3rd element: 5 6
2nd element: 1 2
1st element: 3 4
removing last element...
ptr=0x000349a0 _Next=0x00034988 _Prev=0x0028fe90 x=3 y=4
ptr=0x00034988 _Next=0x0028fe90 _Prev=0x000349a0 x=1 y=2
ptr=0x0028fe90 _Next=0x000349a0 _Prev=0x00034988 x=5 y=6
```

MSVC

MSVC implementation (2012) is just the same, but it also stores current list size. This mean, .size() method is very fast ($O(1)$): just read one value from memory. On the other way, size variable must be corrected at each insertion/deletion.

MSVC implementation is also slightly different in a way it arrange nodes:



GCC has its "garbage" element at the end of the list, while MSVC at the beginning of it.

Listing 29.29: MSVC 2012 /Fa2.asm /Ox /GS- /Ob1

```
_l$ = -16 ; size = 8
_t1$ = -8 ; size = 8
_main    PROC
    sub  esp, 16
    push ebx
    push esi
    push edi
    push 0
    push 0
    lea  ecx, DWORD PTR _l$[esp+36]
    mov  DWORD PTR _l$[esp+40], 0
    ; allocate first "garbage" element
    call ?_Buynode0@?$_List_alloc@$0A@U?$_List_base_types@Ua@@V?↙
    ↳ $allocator@Ua@@@std@@@std@@@std@@QAEPAU?↙
    ↳ $_List_node@Ua@@PAX@2@PAU32@0@Z ; std::_List_alloc<0,std::↙
    ↳ _List_base_types<a,std::allocator<a> > >::_Buynode0
    mov  edi, DWORD PTR __imp__printf
    mov  ebx, eax
    push OFFSET $SG40685 ; '* empty list:'
    mov  DWORD PTR _l$[esp+32], ebx
    call edi  ; printf
    lea  eax, DWORD PTR _l$[esp+32]
```

```
    push eax
    call ?dump_List_val@@YAXPAI@Z ; dump_List_val
    mov  esi, DWORD PTR [ebx]
    add  esp, 8
    lea  eax, DWORD PTR _t1$[esp+28]
    push eax
    push DWORD PTR [esi+4]
    lea  ecx, DWORD PTR _l$[esp+36]
    push esi
    mov  DWORD PTR _t1$[esp+40], 1 ; data for a new node
    mov  DWORD PTR _t1$[esp+44], 2 ; data for a new node
    ; allocate new node
    call ??$_Buynode@ABUa@@@?$_List_buy@Ua@@V?↙
    ↳ $allocator@Ua@@@std@@@std@@QAEPAU?↙
    ↳ $_List_node@Ua@@PAX@1@PAU21@0ABUa@@@Z ; std::_List_buy<a,↙
    ↳ std::allocator<a> >::_Buynode<a const &>
    mov  DWORD PTR [esi+4], eax
    mov  ecx, DWORD PTR [eax+4]
    mov  DWORD PTR _t1$[esp+28], 3 ; data for a new node
    mov  DWORD PTR [ecx], eax
    mov  esi, DWORD PTR [ebx]
    lea  eax, DWORD PTR _t1$[esp+28]
    push eax
    push DWORD PTR [esi+4]
    lea  ecx, DWORD PTR _l$[esp+36]
    push esi
    mov  DWORD PTR _t1$[esp+44], 4 ; data for a new node
    ; allocate new node
    call ??$_Buynode@ABUa@@@?$_List_buy@Ua@@V?↙
    ↳ $allocator@Ua@@@std@@@std@@QAEPAU?↙
    ↳ $_List_node@Ua@@PAX@1@PAU21@0ABUa@@@Z ; std::_List_buy<a,↙
    ↳ std::allocator<a> >::_Buynode<a const &>
    mov  DWORD PTR [esi+4], eax
    mov  ecx, DWORD PTR [eax+4]
    mov  DWORD PTR _t1$[esp+28], 5 ; data for a new node
    mov  DWORD PTR [ecx], eax
    lea  eax, DWORD PTR _t1$[esp+28]
    push eax
    push DWORD PTR [ebx+4]
    lea  ecx, DWORD PTR _l$[esp+36]
    push ebx
    mov  DWORD PTR _t1$[esp+44], 6 ; data for a new node
    ; allocate new node
```

```
call ??$_Buynode@ABUa@@@?$_List_buy@Ua@@V?↙
↳ $allocator@Ua@@@std@@@std@@QAEPAU?↙
↳ $_List_node@Ua@@PAX@1@PAU21@0ABUa@@@Z ; std::_List_buy<a,↙
↳ std::allocator<a> >::_Buynode<a const &>
mov  DWORD PTR [ebx+4], eax
mov  ecx, DWORD PTR [eax+4]
push OFFSET $SG40689 ; '* 3-elements list:'
mov  DWORD PTR _l$[esp+36], 3
mov  DWORD PTR [ecx], eax
call edi ; printf
lea  eax, DWORD PTR _l$[esp+32]
push eax
call ?dump_List_val@@YAXPAI@Z ; dump_List_val
push OFFSET $SG40831 ; 'node at .begin:'
call edi ; printf
push DWORD PTR [ebx] ; get next field of node $l$ variable ↙
↳ points to
call ?dump_List_node@@YAXPAUList_node@@@Z ; dump_List_node
push OFFSET $SG40835 ; 'node at .end:'
call edi ; printf
push ebx ; pointer to the node $l$ variable points to!
call ?dump_List_node@@YAXPAUList_node@@@Z ; dump_List_node
push OFFSET $SG40839 ; '* let''s count from the begin:'
call edi ; printf
mov  esi, DWORD PTR [ebx] ; operator++: get ->next pointer
push DWORD PTR [esi+12]
push DWORD PTR [esi+8]
push OFFSET $SG40846 ; '1st element: %d %d'
call edi ; printf
mov  esi, DWORD PTR [esi] ; operator++: get ->next pointer
push DWORD PTR [esi+12]
push DWORD PTR [esi+8]
push OFFSET $SG40848 ; '2nd element: %d %d'
call edi ; printf
mov  esi, DWORD PTR [esi] ; operator++: get ->next pointer
push DWORD PTR [esi+12]
push DWORD PTR [esi+8]
push OFFSET $SG40850 ; '3rd element: %d %d'
call edi ; printf
mov  eax, DWORD PTR [esi] ; operator++: get ->next pointer
add  esp, 64
push DWORD PTR [eax+12]
push DWORD PTR [eax+8]
```

```
    push OFFSET $SG40852 ; 'element at .end(): %d %d'
    call edi ; printf
    push OFFSET $SG40853 ; '* let''s count from the end:'
    call edi ; printf
    push DWORD PTR [ebx+12] ; use x and y fields from the node ↙
    ↳ $l$ variable points to
    push DWORD PTR [ebx+8]
    push OFFSET $SG40860 ; 'element at .end(): %d %d'
    call edi ; printf
    mov  esi, DWORD PTR [ebx+4] ; operator--: get ->prev pointer
    push DWORD PTR [esi+12]
    push DWORD PTR [esi+8]
    push OFFSET $SG40862 ; '3rd element: %d %d'
    call edi ; printf
    mov  esi, DWORD PTR [esi+4] ; operator--: get ->prev pointer
    push DWORD PTR [esi+12]
    push DWORD PTR [esi+8]
    push OFFSET $SG40864 ; '2nd element: %d %d'
    call edi ; printf
    mov  eax, DWORD PTR [esi+4] ; operator--: get ->prev pointer
    push DWORD PTR [eax+12]
    push DWORD PTR [eax+8]
    push OFFSET $SG40866 ; '1st element: %d %d'
    call edi ; printf
    add  esp, 64
    push OFFSET $SG40867 ; 'removing last element...'
    call edi ; printf
    mov  edx, DWORD PTR [ebx+4]
    add  esp, 4

    ; prev=next?
    ; it is the only element, "garbage one"?
    ; if yes, do not delete it!
    cmp  edx, ebx
    je   SHORT $LN349@main
    mov  ecx, DWORD PTR [edx+4]
    mov  eax, DWORD PTR [edx]
    mov  DWORD PTR [ecx], eax
    mov  ecx, DWORD PTR [edx]
    mov  eax, DWORD PTR [edx+4]
    push edx
    mov  DWORD PTR [ecx+4], eax
    call ??3@YAXPAX@Z ; operator delete
```

```
    add  esp, 4
    mov  DWORD PTR _l$[esp+32], 2
$LN349@main:
    lea  eax, DWORD PTR _l$[esp+28]
    push eax
    call ?dump_List_val@@YAXPAI@Z ; dump_List_val
    mov  eax, DWORD PTR [ebx]
    add  esp, 4
    mov  DWORD PTR [ebx], ebx
    mov  DWORD PTR [ebx+4], ebx
    cmp  eax, ebx
    je   SHORT $LN412@main
$LL414@main:
    mov  esi, DWORD PTR [eax]
    push eax
    call ??3@YAXPAX@Z ; operator delete
    add  esp, 4
    mov  eax, esi
    cmp  esi, ebx
    jne  SHORT $LL414@main
$LN412@main:
    push  ebx
    call  ??3@YAXPAX@Z ; operator delete
    add   esp, 4
    xor   eax, eax
    pop   edi
    pop   esi
    pop   ebx
    add   esp, 16
    ret   0
_main ENDP
```

Unlike GCC, MSVC code allocates “garbage” element at the function start with “Buynode” function, it is also used for the rest nodes allocations (GCC code allocates the very first element in the local stack).

Listing 29.30: The whole output

```
* empty list:
_Myhead=0x003CC258, _Mysize=0
ptr=0x003CC258 _Next=0x003CC258 _Prev=0x003CC258 x=6226002 y↙
    ↳ =4522072
* 3-elements list:
_Myhead=0x003CC258, _Mysize=3
ptr=0x003CC258 _Next=0x003CC288 _Prev=0x003CC2A0 x=6226002 y↙
```

```
    ↪ =4522072
ptr=0x003CC288 _Next=0x003CC270 _Prev=0x003CC258 x=3 y=4
ptr=0x003CC270 _Next=0x003CC2A0 _Prev=0x003CC288 x=1 y=2
ptr=0x003CC2A0 _Next=0x003CC258 _Prev=0x003CC270 x=5 y=6
node at .begin:
ptr=0x003CC288 _Next=0x003CC270 _Prev=0x003CC258 x=3 y=4
node at .end:
ptr=0x003CC258 _Next=0x003CC288 _Prev=0x003CC2A0 x=6226002 y↙
    ↪ =4522072
* let's count from the begin:
1st element: 3 4
2nd element: 1 2
3rd element: 5 6
element at .end(): 6226002 4522072
* let's count from the end:
element at .end(): 6226002 4522072
3rd element: 5 6
2nd element: 1 2
1st element: 3 4
removing last element...
_Myhead=0x003CC258, _Mysize=2
ptr=0x003CC258 _Next=0x003CC288 _Prev=0x003CC270 x=6226002 y↙
    ↪ =4522072
ptr=0x003CC288 _Next=0x003CC270 _Prev=0x003CC258 x=3 y=4
ptr=0x003CC270 _Next=0x003CC258 _Prev=0x003CC288 x=1 y=2
```

C++11 std::forward_list

The same thing as std::list, but singly-linked one, i.e., having only "next" field at teach node. It require smaller memory footprint, but also don't offer a feature to traverse list back.

29.4.3 std::vector

I would call `std::vector` "safe wrapper" of PODT[8] C array. Internally, it is somewhat similar to `std::string` (29.4.1): it has a pointer to buffer, pointer to the end of array, and a pointer to the end of buffer.

Array elements are lie in memory adjacently to each other, just like in usual array (16). In C++11 there are new method `.data()` appeared, returning a pointer to the buffer, akin to `.c_str()` in `std::string`.

Allocated buffer in heap may be larger than array itself.

[8] (C++) Plain Old Data Type

Both MSVC and GCC implementations are similar, just structure field names are slightly different[9], so here is one source code working for both compilters. Here is again a C-like code for dumping `std::vector` structure:

```
#include <stdio.h>
#include <vector>
#include <algorithm>
#include <functional>

struct vector_of_ints
{
    // MSVC names:
    int *Myfirst;
    int *Mylast;
    int *Myend;

    // GCC structure is the same, names are: _M_start, _M_finish
    ↪ , _M_end_of_storage
};

void dump(struct vector_of_ints *in)
{
    printf ("_Myfirst=%p, _Mylast=%p, _Myend=%p\n", in->Myfirst,
    ↪  in->Mylast, in->Myend);
    size_t size=(in->Mylast-in->Myfirst);
    size_t capacity=(in->Myend-in->Myfirst);
    printf ("size=%d, capacity=%d\n", size, capacity);
    for (size_t i=0; i<size; i++)
        printf ("element %d: %d\n", i, in->Myfirst[i]);
};

int main()
{
    std::vector<int> c;
    dump ((struct vector_of_ints*)(void*)&c);
    c.push_back(1);
    dump ((struct vector_of_ints*)(void*)&c);
    c.push_back(2);
    dump ((struct vector_of_ints*)(void*)&c);
    c.push_back(3);
    dump ((struct vector_of_ints*)(void*)&c);
    c.push_back(4);
```

[9]GCC internals: http://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a01371.html

```
    dump ((struct vector_of_ints*)(void*)&c);
    c.reserve (6);
    dump ((struct vector_of_ints*)(void*)&c);
    c.push_back(5);
    dump ((struct vector_of_ints*)(void*)&c);
    c.push_back(6);
    dump ((struct vector_of_ints*)(void*)&c);
    printf ("%d\n", c.at(5)); // bounds checking
    printf ("%d\n", c[8]); // operator[], no bounds checking
};
```

Here is a sample output if compiled in MSVC:

```
_Myfirst=00000000, _Mylast=00000000, _Myend=00000000
size=0, capacity=0
_Myfirst=0051CF48, _Mylast=0051CF4C, _Myend=0051CF4C
size=1, capacity=1
element 0: 1
_Myfirst=0051CF58, _Mylast=0051CF60, _Myend=0051CF60
size=2, capacity=2
element 0: 1
element 1: 2
_Myfirst=0051C278, _Mylast=0051C284, _Myend=0051C284
size=3, capacity=3
element 0: 1
element 1: 2
element 2: 3
_Myfirst=0051C290, _Mylast=0051C2A0, _Myend=0051C2A0
size=4, capacity=4
element 0: 1
element 1: 2
element 2: 3
element 3: 4
_Myfirst=0051B180, _Mylast=0051B190, _Myend=0051B198
size=4, capacity=6
element 0: 1
element 1: 2
element 2: 3
element 3: 4
_Myfirst=0051B180, _Mylast=0051B194, _Myend=0051B198
size=5, capacity=6
element 0: 1
element 1: 2
element 2: 3
```

```
element 3: 4
element 4: 5
_Myfirst=0051B180, _Mylast=0051B198, _Myend=0051B198
size=6, capacity=6
element 0: 1
element 1: 2
element 2: 3
element 3: 4
element 4: 5
element 5: 6
6
6619158
```

As it can be seen, there is no allocated buffer at the `main()` function start yet. After first `push_back()` call buffer is allocated. And then, after each `push_back()` call, both array size and buffer size (*capacity*) are increased. But buffer address is changed as well, because `push_back()` function reallocates the buffer in the heap each time. It is costly operation, that's why it is very important to predict future array size and reserve a space for it with `.reserve()` method. The very last number is a garbage: there are no array elements at this point, so random number is printed. This is illustration to the fact that `operator[]` of `std::vector` is not checking if the index in the array bounds. `.at()` method, however, does checking and throw `std::out_of_range` exception in case of error.

Let's see the code:

Listing 29.31: MSVC 2012 /GS- /Ob1

```
$SG52650 DB '%d', 0aH, 00H
$SG52651 DB '%d', 0aH, 00H

_this$ = -4 ; size = 4
__Pos$ = 8  ; size = 4
?at@?$vector@HV?$allocator@H@std@@@std@@QAEAAHI@Z PROC ; std::↙
    ↳ vector<int,std::allocator<int> >::at, COMDAT
; _this$ = ecx
    push ebp
    mov  ebp, esp
    push ecx
    mov  DWORD PTR _this$[ebp], ecx
    mov  eax, DWORD PTR _this$[ebp]
    mov  ecx, DWORD PTR _this$[ebp]
    mov  edx, DWORD PTR [eax+4]
    sub  edx, DWORD PTR [ecx]
    sar  edx, 2
    cmp  edx, DWORD PTR __Pos$[ebp]
```

```
    ja   SHORT $LN1@at
    push OFFSET ??_C@_0BM@NMJKDPPO@invalid?5vector?$DMT?$DO?5↙
    ↳ subscript?$AA@
    call DWORD PTR __imp_?_Xout_of_range@std@@YAXPBD@Z
$LN1@at:
    mov  eax, DWORD PTR _this$[ebp]
    mov  ecx, DWORD PTR [eax]
    mov  edx, DWORD PTR __Pos$[ebp]
    lea  eax, DWORD PTR [ecx+edx*4]
$LN3@at:
    mov  esp, ebp
    pop  ebp
    ret  4
?at@?$vector@HV?$allocator@H@std@@@std@@QAEAAHI@Z ENDP ; std::↙
    ↳ vector<int,std::allocator<int> >::at

_c$ = -36 ; size = 12
$T1 = -24 ; size = 4
$T2 = -20 ; size = 4
$T3 = -16 ; size = 4
$T4 = -12 ; size = 4
$T5 = -8  ; size = 4
$T6 = -4  ; size = 4
_main PROC
    push ebp
    mov  ebp, esp
    sub  esp, 36
    mov  DWORD PTR _c$[ebp], 0     ; Myfirst
    mov  DWORD PTR _c$[ebp+4], 0   ; Mylast
    mov  DWORD PTR _c$[ebp+8], 0   ; Myend
    lea  eax, DWORD PTR _c$[ebp]
    push eax
    call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
    add  esp, 4
    mov  DWORD PTR $T6[ebp], 1
    lea  ecx, DWORD PTR $T6[ebp]
    push ecx
    lea  ecx, DWORD PTR _c$[ebp]
    call ?push_back@?$vector@HV?↙
    ↳ $allocator@H@std@@@std@@QAEX$$QAH@Z ; std::vector<int,std↙
    ↳ ::allocator<int> >::push_back
    lea  edx, DWORD PTR _c$[ebp]
    push edx
```

```
    call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
    add  esp, 4
    mov  DWORD PTR $T5[ebp], 2
    lea  eax, DWORD PTR $T5[ebp]
    push eax
    lea  ecx, DWORD PTR _c$[ebp]
    call ?push_back@?$vector@HV?↙
    ↳ $allocator@H@std@@@std@@QAEX$$QAH@Z ; std::vector<int,std↙
    ↳ ::allocator<int> >::push_back
    lea  ecx, DWORD PTR _c$[ebp]
    push ecx
    call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
    add  esp, 4
    mov  DWORD PTR $T4[ebp], 3
    lea  edx, DWORD PTR $T4[ebp]
    push edx
    lea  ecx, DWORD PTR _c$[ebp]
    call ?push_back@?$vector@HV?↙
    ↳ $allocator@H@std@@@std@@QAEX$$QAH@Z ; std::vector<int,std↙
    ↳ ::allocator<int> >::push_back
    lea  eax, DWORD PTR _c$[ebp]
    push eax
    call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
    add  esp, 4
    mov  DWORD PTR $T3[ebp], 4
    lea  ecx, DWORD PTR $T3[ebp]
    push ecx
    lea  ecx, DWORD PTR _c$[ebp]
    call ?push_back@?$vector@HV?↙
    ↳ $allocator@H@std@@@std@@QAEX$$QAH@Z ; std::vector<int,std↙
    ↳ ::allocator<int> >::push_back
    lea  edx, DWORD PTR _c$[ebp]
    push edx
    call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
    add  esp, 4
    push 6
    lea  ecx, DWORD PTR _c$[ebp]
    call ?reserve@?$vector@HV?$allocator@H@std@@@std@@QAEXI@Z ; ↙
    ↳ std::vector<int,std::allocator<int> >::reserve
    lea  eax, DWORD PTR _c$[ebp]
    push eax
    call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
    add  esp, 4
```

```
    mov  DWORD PTR $T2[ebp], 5
    lea  ecx, DWORD PTR $T2[ebp]
    push ecx
    lea  ecx, DWORD PTR _c$[ebp]
    call ?push_back@?$vector@HV?↙
    ↳ $allocator@H@std@@@std@@QAEX$$QAH@Z ; std::vector<int,std↙
    ↳ ::allocator<int> >::push_back
    lea  edx, DWORD PTR _c$[ebp]
    push edx
    call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
    add  esp, 4
    mov  DWORD PTR $T1[ebp], 6
    lea  eax, DWORD PTR $T1[ebp]
    push eax
    lea  ecx, DWORD PTR _c$[ebp]
    call ?push_back@?$vector@HV?↙
    ↳ $allocator@H@std@@@std@@QAEX$$QAH@Z ; std::vector<int,std↙
    ↳ ::allocator<int> >::push_back
    lea  ecx, DWORD PTR _c$[ebp]
    push ecx
    call ?dump@@YAXPAUvector_of_ints@@@Z ; dump
    add  esp, 4
    push 5
    lea  ecx, DWORD PTR _c$[ebp]
    call ?at@?$vector@HV?$allocator@H@std@@@std@@QAEAAHI@Z ; std↙
    ↳ ::vector<int,std::allocator<int> >::at
    mov  edx, DWORD PTR [eax]
    push edx
    push OFFSET $SG52650 ; '%d'
    call DWORD PTR __imp__printf
    add  esp, 8
    mov  eax, 8
    shl  eax, 2
    mov  ecx, DWORD PTR _c$[ebp]
    mov  edx, DWORD PTR [ecx+eax]
    push edx
    push OFFSET $SG52651 ; '%d'
    call DWORD PTR __imp__printf
    add  esp, 8
    lea  ecx, DWORD PTR _c$[ebp]
    call ?_Tidy@?$vector@HV?$allocator@H@std@@@std@@IAEXXZ ; std↙
    ↳ ::vector<int,std::allocator<int> >::_Tidy
    xor  eax, eax
```

```
    mov  esp, ebp
    pop  ebp
    ret  0
_main ENDP
```

We see how `.at()` method check bounds and throw exception in case of error. The number of the last `printf()` call is just to be taken from a memory, without any checks.

One may ask, why not to use variables like "size" and "capacity", like it was done in `std::string`. I suppose, that was done for the faster bounds checking. But I'm not sure.

The code GCC generates is almost the same on the whole, but `.at()` method is inlined:

Listing 29.32: GCC 4.8.1 -fno-inline-small-functions -O1

```
main proc near
     push ebp
     mov  ebp, esp
     push edi
     push esi
     push ebx
     and  esp, 0FFFFFFF0h
     sub  esp, 20h
     mov  dword ptr [esp+14h], 0
     mov  dword ptr [esp+18h], 0
     mov  dword ptr [esp+1Ch], 0
     lea  eax, [esp+14h]
     mov  [esp], eax
     call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
     mov  dword ptr [esp+10h], 1
     lea  eax, [esp+10h]
     mov  [esp+4], eax
     lea  eax, [esp+14h]
     mov  [esp], eax
     call _ZNSt6vectorIiSaIiEE9push_backERKi ; std::vector<int,↙
    ↪ std::allocator<int>>::push_back(int  const&)
     lea  eax, [esp+14h]
     mov  [esp], eax
     call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
     mov  dword ptr [esp+10h], 2
     lea  eax, [esp+10h]
     mov  [esp+4], eax
     lea  eax, [esp+14h]
     mov  [esp], eax
```

```
call _ZNSt6vectorIiSaIiEE9push_backERKi ; std::vector<int,↙
↳ std::allocator<int>>::push_back(int  const&)
lea  eax, [esp+14h]
mov  [esp], eax
call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
mov  dword ptr [esp+10h], 3
lea  eax, [esp+10h]
mov  [esp+4], eax
lea  eax, [esp+14h]
mov  [esp], eax
call _ZNSt6vectorIiSaIiEE9push_backERKi ; std::vector<int,↙
↳ std::allocator<int>>::push_back(int  const&)
lea  eax, [esp+14h]
mov  [esp], eax
call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
mov  dword ptr [esp+10h], 4
lea  eax, [esp+10h]
mov  [esp+4], eax
lea  eax, [esp+14h]
mov  [esp], eax
call _ZNSt6vectorIiSaIiEE9push_backERKi ; std::vector<int,↙
↳ std::allocator<int>>::push_back(int  const&)
lea  eax, [esp+14h]
mov  [esp], eax
call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
mov  ebx, [esp+14h]
mov  eax, [esp+1Ch]
sub  eax, ebx
cmp  eax, 17h
ja   short loc_80001CF
mov  edi, [esp+18h]
sub  edi, ebx
sar  edi, 2
mov  dword ptr [esp], 18h
call _Znwj            ; operator new(uint)
mov  esi, eax
test edi, edi
jz   short loc_80001AD
lea  eax, ds:0[edi*4]
mov  [esp+8], eax     ; n
mov  [esp+4], ebx     ; src
mov  [esp], esi       ; dest
call memmove
```

```
loc_80001AD: ; CODE XREF: main+F8
     mov  eax, [esp+14h]
     test eax, eax
     jz   short loc_80001BD
     mov  [esp], eax      ; void *
     call _ZdlPv          ; operator delete(void *)

loc_80001BD: ; CODE XREF: main+117
     mov  [esp+14h], esi
     lea  eax, [esi+edi*4]
     mov  [esp+18h], eax
     add  esi, 18h
     mov  [esp+1Ch], esi

loc_80001CF: ; CODE XREF: main+DD
     lea  eax, [esp+14h]
     mov  [esp], eax
     call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
     mov  dword ptr [esp+10h], 5
     lea  eax, [esp+10h]
     mov  [esp+4], eax
     lea  eax, [esp+14h]
     mov  [esp], eax
     call _ZNSt6vectorIiSaIiEE9push_backERKi ; std::vector<int,↙
    ↪ std::allocator<int>>::push_back(int  const&)
     lea  eax, [esp+14h]
     mov  [esp], eax
     call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
     mov  dword ptr [esp+10h], 6
     lea  eax, [esp+10h]
     mov  [esp+4], eax
     lea  eax, [esp+14h]
     mov  [esp], eax
     call _ZNSt6vectorIiSaIiEE9push_backERKi ; std::vector<int,↙
    ↪ std::allocator<int>>::push_back(int  const&)
     lea  eax, [esp+14h]
     mov  [esp], eax
     call _Z4dumpP14vector_of_ints ; dump(vector_of_ints *)
     mov  eax, [esp+14h]
     mov  edx, [esp+18h]
     sub  edx, eax
     cmp  edx, 17h
```

```
     ja   short loc_8000246
     mov  dword ptr [esp], offset aVector_m_range ; "vector::↙
    ↳ _M_range_check"
     call _ZSt20__throw_out_of_rangePKc ; std::↙
    ↳ __throw_out_of_range(char  const*)

loc_8000246:                            ; CODE XREF: main+19C
     mov  eax, [eax+14h]
     mov  [esp+8], eax
     mov  dword ptr [esp+4], offset aD ; "%d\n"
     mov  dword ptr [esp], 1
     call __printf_chk
     mov  eax, [esp+14h]
     mov  eax, [eax+20h]
     mov  [esp+8], eax
     mov  dword ptr [esp+4], offset aD ; "%d\n"
     mov  dword ptr [esp], 1
     call __printf_chk
     mov  eax, [esp+14h]
     test eax, eax
     jz   short loc_80002AC
     mov  [esp], eax      ; void *
     call _ZdlPv          ; operator delete(void *)
     jmp  short loc_80002AC

     mov  ebx, eax
     mov  edx, [esp+14h]
     test edx, edx
     jz   short loc_80002A4
     mov  [esp], edx      ; void *
     call _ZdlPv          ; operator delete(void *)

loc_80002A4: ; CODE XREF: main+1FE
     mov  [esp], ebx
     call _Unwind_Resume


loc_80002AC: ; CODE XREF: main+1EA
             ; main+1F4
     mov  eax, 0
     lea  esp, [ebp-0Ch]
     pop  ebx
     pop  esi
```

```
     pop  edi
     pop  ebp

locret_80002B8: ; DATA XREF: .eh_frame:08000510
                ; .eh_frame:080005BC
     retn
main endp
```

`.reserve()` method is inlined as well. It calls `new()` if buffer is too small for new size, call `memmove()` to copy buffer contents, and call `delete()` to free old buffer.

Let's also see what the compiled program outputs if compiled by GCC:

```
_Myfirst=0x(nil), _Mylast=0x(nil), _Myend=0x(nil)
size=0, capacity=0
_Myfirst=0x8257008, _Mylast=0x825700c, _Myend=0x825700c
size=1, capacity=1
element 0: 1
_Myfirst=0x8257018, _Mylast=0x8257020, _Myend=0x8257020
size=2, capacity=2
element 0: 1
element 1: 2
_Myfirst=0x8257028, _Mylast=0x8257034, _Myend=0x8257038
size=3, capacity=4
element 0: 1
element 1: 2
element 2: 3
_Myfirst=0x8257028, _Mylast=0x8257038, _Myend=0x8257038
size=4, capacity=4
element 0: 1
element 1: 2
element 2: 3
element 3: 4
_Myfirst=0x8257040, _Mylast=0x8257050, _Myend=0x8257058
size=4, capacity=6
element 0: 1
element 1: 2
element 2: 3
element 3: 4
_Myfirst=0x8257040, _Mylast=0x8257054, _Myend=0x8257058
size=5, capacity=6
element 0: 1
element 1: 2
element 2: 3
element 3: 4
```

```
element 4: 5
_Myfirst=0x8257040, _Mylast=0x8257058, _Myend=0x8257058
size=6, capacity=6
element 0: 1
element 1: 2
element 2: 3
element 3: 4
element 4: 5
element 5: 6
6
0
```

We can spot that buffer size grows in different way that in MSVC.

Simple experimentation shows that MSVC implementation buffer grows by ~50% each time it needs to be enlarged, while GCC code enlarges it by 100% each time, i.e., doubles it each time.

29.4.4 std::map and std::set

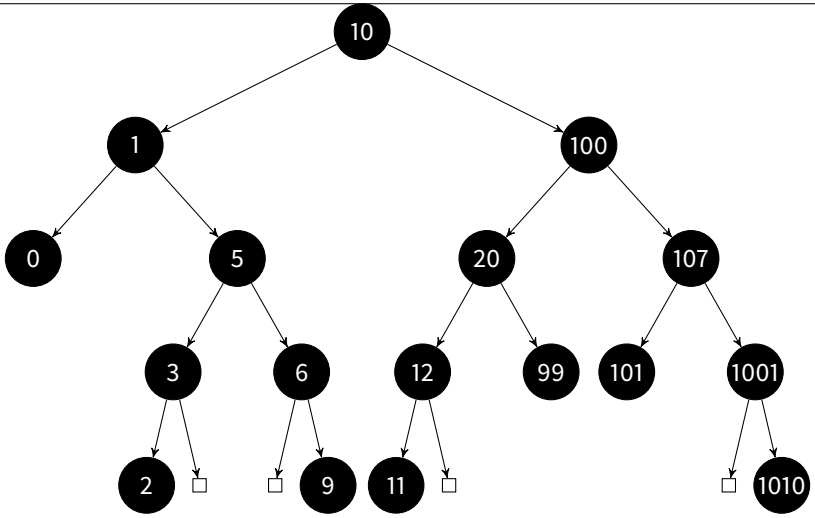
Binary tree is another fundamental data structure. As it states, this is a tree, but each node has at most 2 links to other nodes. Each node have key and/or value.

Binary trees are usually the structure used in "dictionaries" of key-values (AKA "associative arrays") implementations.

There are at least three important properties binary trees has:

- All keys are stored in always sorted form.
- Keys of any types can be stored easily. Binary tree algorithms are unaware of key type, only key comparison function is required.
- Finding needed key is relatively fast in comparison with lists and arrays.

Here is a very simple example: let's store these numbers in binary tree: 0, 1, 2, 3, 5, 6, 9, 10, 11, 12, 20, 99, 100, 101, 107, 1001, 1010.



All keys lesser than node key value is stored on the left side. All keys greater than node key value is stored on the right side.

Hence, finding algorithm is straightforward: if the value you looking for is lesser than current node's key value: move left, if it is greater: move right, stop if the value required is equals to the node's key value. That is why searching algorithm may search for numbers, text strings, etc, using only key comparison function.

All keys has unique values.

Having that, one need $\approx \log_2 n$ steps in order to find a key in the balanced binary tree of n keys. It is ≈ 10 steps for ≈ 1000 keys, or ≈ 13 steps for ≈ 10000 keys. Not bad, but tree should always be balanced for this: i.e., keys should be distributed evenly on all tiers. Insertion and removal operations do some maintenance to keep tree in balanced state.

There are several popular balancing algorithms available, including AVL tree and red-black tree. The latter extends a node by a "color" value for simplifying balancing process, hence, each node may be "red" or "black".

Both GCC and MSVC `std::map` and `std::set` template implementations use red-black trees.

`std::set` contain only keys. `std::map` is "extended" version of set: it also has a value at each node.

MSVC

```
#include <map>
#include <set>
#include <string>
#include <iostream>
```

```
// structure is not packed!
struct tree_node
{
    struct tree_node *Left;
    struct tree_node *Parent;
    struct tree_node *Right;
    char Color; // 0 - Red, 1 - Black
    char Isnil;
    //std::pair Myval;
    unsigned int first; // called Myval in std::set
    const char *second; // not present in std::set
};

struct tree_struct
{
    struct tree_node *Myhead;
    size_t Mysize;
};

void dump_tree_node (struct tree_node *n, bool is_set, bool ↙
    ↳ traverse)
{
    printf ("ptr=0x%p Left=0x%p Parent=0x%p Right=0x%p Color=%d ↙
    ↳ Isnil=%d\n",
            n, n->Left, n->Parent, n->Right, n->Color, n->Isnil)↙
    ↳ ;
    if (n->Isnil==0)
    {
        if (is_set)
            printf ("first=%d\n", n->first);
        else
            printf ("first=%d second=[%s]\n", n->first, n->↙
    ↳ second);
    }

    if (traverse)
    {
        if (n->Isnil==1)
            dump_tree_node (n->Parent, is_set, true);
        else
        {
            if (n->Left->Isnil==0)
```

```
                dump_tree_node (n->Left, is_set, true);
            if (n->Right->Isnil==0)
                dump_tree_node (n->Right, is_set, true);
        };
    };
};

const char* ALOT_OF_TABS="\t\t\t\t\t\t\t\t\t\t\t\t";

void dump_as_tree (int tabs, struct tree_node *n, bool is_set)
{
    if (is_set)
        printf ("%d\n", n->first);
    else
        printf ("%d [%s]\n", n->first, n->second);
    if (n->Left->Isnil==0)
    {
        printf ("%.*sL-------", tabs, ALOT_OF_TABS);
        dump_as_tree (tabs+1, n->Left, is_set);
    };
    if (n->Right->Isnil==0)
    {
        printf ("%.*sR-------", tabs, ALOT_OF_TABS);
        dump_as_tree (tabs+1, n->Right, is_set);
    };
};

void dump_map_and_set(struct tree_struct *m, bool is_set)
{
    printf ("ptr=0x%p, Myhead=0x%p, Mysize=%d\n", m, m->Myhead, ↙
    ↳ m->Mysize);
    dump_tree_node (m->Myhead, is_set, true);
    printf ("As a tree:\n");
    printf ("root----");
    dump_as_tree (1, m->Myhead->Parent, is_set);
};

int main()
{
    // map

    std::map<int, const char*> m;
```

```
    m[10]="ten";
    m[20]="twenty";
    m[3]="three";
    m[101]="one hundred one";
    m[100]="one hundred";
    m[12]="twelve";
    m[107]="one hundred seven";
    m[0]="zero";
    m[1]="one";
    m[6]="six";
    m[99]="ninety-nine";
    m[5]="five";
    m[11]="eleven";
    m[1001]="one thousand one";
    m[1010]="one thousand ten";
    m[2]="two";
    m[9]="nine";
    printf ("dumping m as map:\n");
    dump_map_and_set ((struct tree_struct *)(void*)&m, false);

    std::map<int, const char*>::iterator it1=m.begin();
    printf ("m.begin():\n");
    dump_tree_node ((struct tree_node *)*(void**)&it1, false, ↙
    ↳ false);
    it1=m.end();
    printf ("m.end():\n");
    dump_tree_node ((struct tree_node *)*(void**)&it1, false, ↙
    ↳ false);

    // set

    std::set<int> s;
    s.insert(123);
    s.insert(456);
    s.insert(11);
    s.insert(12);
    s.insert(100);
    s.insert(1001);
    printf ("dumping s as set:\n");
    dump_map_and_set ((struct tree_struct *)(void*)&s, true);
    std::set<int>::iterator it2=s.begin();
    printf ("s.begin():\n");
```

```
    dump_tree_node ((struct tree_node *)*(void**)&it2, true, ↙
    ↳ false);
    it2=s.end();
    printf ("s.end():\n");
    dump_tree_node ((struct tree_node *)*(void**)&it2, true, ↙
    ↳ false);
};
```

Listing 29.33: MSVC 2012

```
dumping m as map:
ptr=0x0020FE04, Myhead=0x005BB3A0, Mysize=17
ptr=0x005BB3A0 Left=0x005BB4A0 Parent=0x005BB3C0 Right=0↙
    ↳ x005BB580 Color=1 Isnil=1
ptr=0x005BB3C0 Left=0x005BB4C0 Parent=0x005BB3A0 Right=0↙
    ↳ x005BB440 Color=1 Isnil=0
first=10 second=[ten]
ptr=0x005BB4C0 Left=0x005BB4A0 Parent=0x005BB3C0 Right=0↙
    ↳ x005BB520 Color=1 Isnil=0
first=1 second=[one]
ptr=0x005BB4A0 Left=0x005BB3A0 Parent=0x005BB4C0 Right=0↙
    ↳ x005BB3A0 Color=1 Isnil=0
first=0 second=[zero]
ptr=0x005BB520 Left=0x005BB400 Parent=0x005BB4C0 Right=0↙
    ↳ x005BB4E0 Color=0 Isnil=0
first=5 second=[five]
ptr=0x005BB400 Left=0x005BB5A0 Parent=0x005BB520 Right=0↙
    ↳ x005BB3A0 Color=1 Isnil=0
first=3 second=[three]
ptr=0x005BB5A0 Left=0x005BB3A0 Parent=0x005BB400 Right=0↙
    ↳ x005BB3A0 Color=0 Isnil=0
first=2 second=[two]
ptr=0x005BB4E0 Left=0x005BB3A0 Parent=0x005BB520 Right=0↙
    ↳ x005BB5C0 Color=1 Isnil=0
first=6 second=[six]
ptr=0x005BB5C0 Left=0x005BB3A0 Parent=0x005BB4E0 Right=0↙
    ↳ x005BB3A0 Color=0 Isnil=0
first=9 second=[nine]
ptr=0x005BB440 Left=0x005BB3E0 Parent=0x005BB3C0 Right=0↙
    ↳ x005BB480 Color=1 Isnil=0
first=100 second=[one hundred]
ptr=0x005BB3E0 Left=0x005BB460 Parent=0x005BB440 Right=0↙
    ↳ x005BB500 Color=0 Isnil=0
first=20 second=[twenty]
```

```
ptr=0x005BB460 Left=0x005BB540 Parent=0x005BB3E0 Right=0↙
    ↳ x005BB3A0 Color=1 Isnil=0
first=12 second=[twelve]
ptr=0x005BB540 Left=0x005BB3A0 Parent=0x005BB460 Right=0↙
    ↳ x005BB3A0 Color=0 Isnil=0
first=11 second=[eleven]
ptr=0x005BB500 Left=0x005BB3A0 Parent=0x005BB3E0 Right=0↙
    ↳ x005BB3A0 Color=1 Isnil=0
first=99 second=[ninety-nine]
ptr=0x005BB480 Left=0x005BB420 Parent=0x005BB440 Right=0↙
    ↳ x005BB560 Color=0 Isnil=0
first=107 second=[one hundred seven]
ptr=0x005BB420 Left=0x005BB3A0 Parent=0x005BB480 Right=0↙
    ↳ x005BB3A0 Color=1 Isnil=0
first=101 second=[one hundred one]
ptr=0x005BB560 Left=0x005BB3A0 Parent=0x005BB480 Right=0↙
    ↳ x005BB580 Color=1 Isnil=0
first=1001 second=[one thousand one]
ptr=0x005BB580 Left=0x005BB3A0 Parent=0x005BB560 Right=0↙
    ↳ x005BB3A0 Color=0 Isnil=0
first=1010 second=[one thousand ten]
As a tree:
root----10 [ten]
        L-------1 [one]
                L-------0 [zero]
                R-------5 [five]
                        L-------3 [three]
                                L-------2 [two]
                        R-------6 [six]
                                R-------9 [nine]
        R-------100 [one hundred]
                L-------20 [twenty]
                        L-------12 [twelve]
                                L-------11 [eleven]
                        R-------99 [ninety-nine]
                R-------107 [one hundred seven]
                        L-------101 [one hundred one]
                        R-------1001 [one thousand one]
                                R-------1010 [one thousand ten]
m.begin():
ptr=0x005BB4A0 Left=0x005BB3A0 Parent=0x005BB4C0 Right=0↙
    ↳ x005BB3A0 Color=1 Isnil=0
first=0 second=[zero]
```

```
m.end():
ptr=0x005BB3A0 Left=0x005BB4A0 Parent=0x005BB3C0 Right=0↙
    ↳ x005BB580 Color=1 Isnil=1

dumping s as set:
ptr=0x0020FDFC, Myhead=0x005BB5E0, Mysize=6
ptr=0x005BB5E0 Left=0x005BB640 Parent=0x005BB600 Right=0↙
    ↳ x005BB6A0 Color=1 Isnil=1
ptr=0x005BB600 Left=0x005BB660 Parent=0x005BB5E0 Right=0↙
    ↳ x005BB620 Color=1 Isnil=0
first=123
ptr=0x005BB660 Left=0x005BB640 Parent=0x005BB600 Right=0↙
    ↳ x005BB680 Color=1 Isnil=0
first=12
ptr=0x005BB640 Left=0x005BB5E0 Parent=0x005BB660 Right=0↙
    ↳ x005BB5E0 Color=0 Isnil=0
first=11
ptr=0x005BB680 Left=0x005BB5E0 Parent=0x005BB660 Right=0↙
    ↳ x005BB5E0 Color=0 Isnil=0
first=100
ptr=0x005BB620 Left=0x005BB5E0 Parent=0x005BB600 Right=0↙
    ↳ x005BB6A0 Color=1 Isnil=0
first=456
ptr=0x005BB6A0 Left=0x005BB5E0 Parent=0x005BB620 Right=0↙
    ↳ x005BB5E0 Color=0 Isnil=0
first=1001
As a tree:
root----123
        L-------12
                L-------11
                R-------100
        R-------456
                R-------1001
s.begin():
ptr=0x005BB640 Left=0x005BB5E0 Parent=0x005BB660 Right=0↙
    ↳ x005BB5E0 Color=0 Isnil=0
first=11
s.end():
ptr=0x005BB5E0 Left=0x005BB640 Parent=0x005BB600 Right=0↙
    ↳ x005BB6A0 Color=1 Isnil=1
```

Structure is not packed, so both *char* type values occupy 4 bytes each.

As for `std::map`, `first` and `second` can be viewed as a single value of `std::pair` type. `std::set` has only one value at this point in the structure instead.

Current size of tree is always present, as in case of `std::list` MSVC implementation (29.4.2).

As in case of `std::list`, iterators are just pointers to the nodes. `.begin()` iterator pointing to the minimal key. That pointer is not stored somewhere (as in lists), minimal key of tree is to be found each time. `operator-` and `operator++` moves pointer to the current node to predecessor and successor respectively, i.e., nodes which has previous and next key. The algorithms for all these operations are described in [7].

`.end()` iterator pointing to the root node, it has 1 in `Isnil`, meaning, the node has no key and/or value. So it can be viewed as a "landing zone" in HDD[10].

GCC

```
#include <stdio.h>
#include <map>
#include <set>
#include <string>
#include <iostream>

struct map_pair
{
    int key;
    const char *value;
};

struct tree_node
{
    int M_color; // 0 - Red, 1 - Black
    struct tree_node *M_parent;
    struct tree_node *M_left;
    struct tree_node *M_right;
};

struct tree_struct
{
    int M_key_compare;
    struct tree_node M_header;
    size_t M_node_count;
};

void dump_tree_node (struct tree_node *n, bool is_set, bool ↙
    ↪ traverse, bool dump_keys_and_values)
```

[10] Hard disk drive

```
{
    printf ("ptr=0x%p M_left=0x%p M_parent=0x%p M_right=0x%p ↙
    ↳ M_color=%d\n",
            n, n->M_left, n->M_parent, n->M_right, n->M_color);

    void *point_after_struct=((char*)n)+sizeof(struct tree_node)↙
    ↳ ;

    if (dump_keys_and_values)
    {
        if (is_set)
            printf ("key=%d\n", *(int*)point_after_struct);
        else
        {
            struct map_pair *p=(struct map_pair *)↙
    ↳ point_after_struct;
            printf ("key=%d value=[%s]\n", p->key, p->value);
        };
    };

    if (traverse==false)
        return;

    if (n->M_left)
        dump_tree_node (n->M_left, is_set, traverse, ↙
    ↳ dump_keys_and_values);
    if (n->M_right)
        dump_tree_node (n->M_right, is_set, traverse, ↙
    ↳ dump_keys_and_values);
};

const char* ALOT_OF_TABS="\t\t\t\t\t\t\t\t\t\t\t\t";

void dump_as_tree (int tabs, struct tree_node *n, bool is_set)
{
    void *point_after_struct=((char*)n)+sizeof(struct tree_node)↙
    ↳ ;

    if (is_set)
        printf ("%d\n", *(int*)point_after_struct);
    else
    {
```

```
        struct map_pair *p=(struct map_pair *)point_after_struct↙
    ↳ ;
        printf ("%d [%s]\n", p->key, p->value);
    }

    if (n->M_left)
    {
        printf ("%.*sL-------", tabs, ALOT_OF_TABS);
        dump_as_tree (tabs+1, n->M_left, is_set);
    };
    if (n->M_right)
    {
        printf ("%.*sR-------", tabs, ALOT_OF_TABS);
        dump_as_tree (tabs+1, n->M_right, is_set);
    };
};

void dump_map_and_set(struct tree_struct *m, bool is_set)
{
    printf ("ptr=0x%p, M_key_compare=0x%x, M_header=0x%p, ↙
    ↳ M_node_count=%d\n",
        m, m->M_key_compare, &m->M_header, m->M_node_count);
    dump_tree_node (m->M_header.M_parent, is_set, true, true);
    printf ("As a tree:\n");
    printf ("root----");
    dump_as_tree (1, m->M_header.M_parent, is_set);
};

int main()
{
    // map

    std::map<int, const char*> m;

    m[10]="ten";
    m[20]="twenty";
    m[3]="three";
    m[101]="one hundred one";
    m[100]="one hundred";
    m[12]="twelve";
    m[107]="one hundred seven";
    m[0]="zero";
    m[1]="one";
```

```
    m[6]="six";
    m[99]="ninety-nine";
    m[5]="five";
    m[11]="eleven";
    m[1001]="one thousand one";
    m[1010]="one thousand ten";
    m[2]="two";
    m[9]="nine";

    printf ("dumping m as map:\n");
    dump_map_and_set ((struct tree_struct *)(void*)&m, false);

    std::map<int, const char*>::iterator it1=m.begin();
    printf ("m.begin():\n");
    dump_tree_node ((struct tree_node *)*(void**)&it1, false, ↙
    ↳ false, true);
    it1=m.end();
    printf ("m.end():\n");
    dump_tree_node ((struct tree_node *)*(void**)&it1, false, ↙
    ↳ false, false);

    // set

    std::set<int> s;
    s.insert(123);
    s.insert(456);
    s.insert(11);
    s.insert(12);
    s.insert(100);
    s.insert(1001);
    printf ("dumping s as set:\n");
    dump_map_and_set ((struct tree_struct *)(void*)&s, true);
    std::set<int>::iterator it2=s.begin();
    printf ("s.begin():\n");
    dump_tree_node ((struct tree_node *)*(void**)&it2, true, ↙
    ↳ false, true);
    it2=s.end();
    printf ("s.end():\n");
    dump_tree_node ((struct tree_node *)*(void**)&it2, true, ↙
    ↳ false, false);
};
```

Listing 29.34: GCC 4.8.1

```
dumping m as map:
ptr=0x0028FE3C, M_key_compare=0x402b70, M_header=0x0028FE40, ↙
    ↳ M_node_count=17
ptr=0x007A4988 M_left=0x007A4C00 M_parent=0x0028FE40 M_right=0↙
    ↳ x007A4B80 M_color=1
key=10 value=[ten]
ptr=0x007A4C00 M_left=0x007A4BE0 M_parent=0x007A4988 M_right=0↙
    ↳ x007A4C60 M_color=1
key=1 value=[one]
ptr=0x007A4BE0 M_left=0x00000000 M_parent=0x007A4C00 M_right=0↙
    ↳ x00000000 M_color=1
key=0 value=[zero]
ptr=0x007A4C60 M_left=0x007A4B40 M_parent=0x007A4C00 M_right=0↙
    ↳ x007A4C20 M_color=0
key=5 value=[five]
ptr=0x007A4B40 M_left=0x007A4CE0 M_parent=0x007A4C60 M_right=0↙
    ↳ x00000000 M_color=1
key=3 value=[three]
ptr=0x007A4CE0 M_left=0x00000000 M_parent=0x007A4B40 M_right=0↙
    ↳ x00000000 M_color=0
key=2 value=[two]
ptr=0x007A4C20 M_left=0x00000000 M_parent=0x007A4C60 M_right=0↙
    ↳ x007A4D00 M_color=1
key=6 value=[six]
ptr=0x007A4D00 M_left=0x00000000 M_parent=0x007A4C20 M_right=0↙
    ↳ x00000000 M_color=0
key=9 value=[nine]
ptr=0x007A4B80 M_left=0x007A49A8 M_parent=0x007A4988 M_right=0↙
    ↳ x007A4BC0 M_color=1
key=100 value=[one hundred]
ptr=0x007A49A8 M_left=0x007A4BA0 M_parent=0x007A4B80 M_right=0↙
    ↳ x007A4C40 M_color=0
key=20 value=[twenty]
ptr=0x007A4BA0 M_left=0x007A4C80 M_parent=0x007A49A8 M_right=0↙
    ↳ x00000000 M_color=1
key=12 value=[twelve]
ptr=0x007A4C80 M_left=0x00000000 M_parent=0x007A4BA0 M_right=0↙
    ↳ x00000000 M_color=0
key=11 value=[eleven]
ptr=0x007A4C40 M_left=0x00000000 M_parent=0x007A49A8 M_right=0↙
    ↳ x00000000 M_color=1
key=99 value=[ninety-nine]
ptr=0x007A4BC0 M_left=0x007A4B60 M_parent=0x007A4B80 M_right=0↙
```

```
    ↳ x007A4CA0 M_color=0
key=107 value=[one hundred seven]
ptr=0x007A4B60 M_left=0x00000000 M_parent=0x007A4BC0 M_right=0↙
    ↳ x00000000 M_color=1
key=101 value=[one hundred one]
ptr=0x007A4CA0 M_left=0x00000000 M_parent=0x007A4BC0 M_right=0↙
    ↳ x007A4CC0 M_color=1
key=1001 value=[one thousand one]
ptr=0x007A4CC0 M_left=0x00000000 M_parent=0x007A4CA0 M_right=0↙
    ↳ x00000000 M_color=0
key=1010 value=[one thousand ten]
As a tree:
root----10 [ten]
        L-------1 [one]
                L-------0 [zero]
                R-------5 [five]
                        L-------3 [three]
                                L-------2 [two]
                        R-------6 [six]
                                R-------9 [nine]
        R-------100 [one hundred]
                L-------20 [twenty]
                        L-------12 [twelve]
                                L-------11 [eleven]
                        R-------99 [ninety-nine]
                R-------107 [one hundred seven]
                        L-------101 [one hundred one]
                        R-------1001 [one thousand one]
                                R-------1010 [one thousand ten]
m.begin():
ptr=0x007A4BE0 M_left=0x00000000 M_parent=0x007A4C00 M_right=0↙
    ↳ x00000000 M_color=1
key=0 value=[zero]
m.end():
ptr=0x0028FE40 M_left=0x007A4BE0 M_parent=0x007A4988 M_right=0↙
    ↳ x007A4CC0 M_color=0

dumping s as set:
ptr=0x0028FE20, M_key_compare=0x8, M_header=0x0028FE24, ↙
    ↳ M_node_count=6
ptr=0x007A1E80 M_left=0x01D5D890 M_parent=0x0028FE24 M_right=0↙
    ↳ x01D5D850 M_color=1
key=123
```

```
ptr=0x01D5D890 M_left=0x01D5D870 M_parent=0x007A1E80 M_right=0↙
    ↳ x01D5D8B0 M_color=1
key=12
ptr=0x01D5D870 M_left=0x00000000 M_parent=0x01D5D890 M_right=0↙
    ↳ x00000000 M_color=0
key=11
ptr=0x01D5D8B0 M_left=0x00000000 M_parent=0x01D5D890 M_right=0↙
    ↳ x00000000 M_color=0
key=100
ptr=0x01D5D850 M_left=0x00000000 M_parent=0x007A1E80 M_right=0↙
    ↳ x01D5D8D0 M_color=1
key=456
ptr=0x01D5D8D0 M_left=0x00000000 M_parent=0x01D5D850 M_right=0↙
    ↳ x00000000 M_color=0
key=1001
As a tree:
root----123
        L-------12
                L-------11
                R-------100
        R-------456
                R-------1001
s.begin():
ptr=0x01D5D870 M_left=0x00000000 M_parent=0x01D5D890 M_right=0↙
    ↳ x00000000 M_color=0
key=11
s.end():
ptr=0x0028FE24 M_left=0x01D5D870 M_parent=0x007A1E80 M_right=0↙
    ↳ x01D5D8D0 M_color=0
```

GCC implementation is very similar [11]. The only difference is absence of `Isnil` field, so the structure occupy slightly less space in memory than as it is implemented in MSVC. Root node is also used as a place `.end()` iterator pointing to and also has no key and/or value.

Rebalancing demo (GCC)

Here is also a demo showing us how tree is rebalanced after insertions.

Listing 29.35: GCC

```
#include <stdio.h>
#include <map>
```

[11] http://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.1/stl__tree_8h-source.html

```
#include <set>
#include <string>
#include <iostream>

struct map_pair
{
    int key;
    const char *value;
};

struct tree_node
{
    int M_color; // 0 - Red, 1 - Black
    struct tree_node *M_parent;
    struct tree_node *M_left;
    struct tree_node *M_right;
};

struct tree_struct
{
    int M_key_compare;
    struct tree_node M_header;
    size_t M_node_count;
};

const char* ALOT_OF_TABS="\t\t\t\t\t\t\t\t\t\t\t\t";

void dump_as_tree (int tabs, struct tree_node *n)
{
    void *point_after_struct=((char*)n)+sizeof(struct tree_node)↙
    ↳ ;

    printf ("%d\n", *(int*)point_after_struct);

    if (n->M_left)
    {
        printf ("%.*sL-------", tabs, ALOT_OF_TABS);
        dump_as_tree (tabs+1, n->M_left);
    };
    if (n->M_right)
    {
        printf ("%.*sR-------", tabs, ALOT_OF_TABS);
        dump_as_tree (tabs+1, n->M_right);
```

```
    };
};

void dump_map_and_set(struct tree_struct *m)
{
    printf ("root----");
    dump_as_tree (1, m->M_header.M_parent);
};

int main()
{
    std::set<int> s;
    s.insert(123);
    s.insert(456);
    printf ("123, 456 are inserted\n");
    dump_map_and_set ((struct tree_struct *)(void*)&s);
    s.insert(11);
    s.insert(12);
    printf ("\n");
    printf ("11, 12 are inserted\n");
    dump_map_and_set ((struct tree_struct *)(void*)&s);
    s.insert(100);
    s.insert(1001);
    printf ("\n");
    printf ("100, 1001 are inserted\n");
    dump_map_and_set ((struct tree_struct *)(void*)&s);
    s.insert(667);
    s.insert(1);
    s.insert(4);
    s.insert(7);
    printf ("\n");
    printf ("667, 1, 4, 7 are inserted\n");
    dump_map_and_set ((struct tree_struct *)(void*)&s);
    printf ("\n");
};
```

Listing 29.36: GCC 4.8.1

```
123, 456 are inserted
root----123
        R-------456

11, 12 are inserted
root----123
```

```
        L-------11
                R-------12
        R-------456

100, 1001 are inserted
root----123
        L-------12
                L-------11
                R-------100
        R-------456
                R-------1001

667, 1, 4, 7 are inserted
root----12
        L-------4
                L-------1
                R-------11
                        L-------7
        R-------123
                L-------100
                R-------667
                        L-------456
                        R-------1001
```

Chapter 30

Obfuscation

Obfuscation is an attempt to hide the code (or its meaning) from reverse engineer.

30.1 Text strings

As I revealed in (36) text strings may be utterly helpful. Programmers who aware of this, may try to hide them resulting unableness to find the string in IDA or any hex editor.

Here is the simpliest method.

That is how the string may be constructed:

```
mov     byte ptr [ebx], 'h'
mov     byte ptr [ebx+1], 'e'
mov     byte ptr [ebx+2], 'l'
mov     byte ptr [ebx+3], 'l'
mov     byte ptr [ebx+4], 'o'
mov     byte ptr [ebx+5], ' '
mov     byte ptr [ebx+6], 'w'
mov     byte ptr [ebx+7], 'o'
mov     byte ptr [ebx+8], 'r'
mov     byte ptr [ebx+9], 'l'
mov     byte ptr [ebx+10], 'd'
```

The string is also can be compared with another like:

```
mov     ebx, offset username
cmp     byte ptr [ebx], 'j'
jnz     fail
cmp     byte ptr [ebx+1], 'o'
jnz     fail
```

```
cmp     byte ptr [ebx+2], 'h'
jnz     fail
cmp     byte ptr [ebx+3], 'n'
jnz     fail
jz      it_is_john
```

In both cases, it is impossible to find these strings straightforwardly in hex editor.

By the way, it is a chance to work with the strings when it is impossible to allocate it in data segment, for example, in PIC or in shellcode.

Another method I once saw is to use `sprintf()` for constructing:

```
sprintf(buf, "%s%c%s%c%s", "hel",'l',"o w",'o',"rld");
```

The code looks weird, but as a simpliest anti-reversing measure it may be helpul.

Text strings may also be present in encrypted form, then all string usage will precede string decrypting routine.

30.2 Executable code

30.2.1 Inserting garbage

Executable code obfuscation mean inserting random garbage code between real one, which executes but not doing anything useful.

Simple example is:

```
add     eax, ebx
mul     ecx
```

Listing 30.1: obfuscated code

```
xor     esi, 011223344h ; garbage
add     esi, eax        ; garbage
add     eax, ebx
mov     edx, eax        ; garbage
shl     edx, 4          ; garbage
mul     ecx
xor     esi, ecx        ; garbage
```

Here garbage code uses registers which are not used in the real code (`ESI` and `EDX`). However, intermediate results produced by the real code may be used by garbage instructions for extra mess—why not?

30.2.2 Replacing instructions to bloated equivalents

- `MOV op1, op2` can be replaced by `PUSH op2 / POP op1` pair.

- `JMP label` can be replaced by `PUSH label / RET` pair. IDA will not show references to the label.
- `CALL label` can be replaced by `PUSH label_after_CALL_instruction / PUSH label / RET` triplet.
- `PUSH op` may also be replaced by `SUB ESP, 4 (or 8) / MOV [ESP], op` pair.

30.2.3 Always executed/never executed code

If the developer is sure that ESI at the point is always 0:

```
mov     esi, 1
...     ; some code not touching ESI
dec     esi
...     ; some code not touching ESI
cmp     esi, 0
jz      real_code
; fake luggage
real_code:
```

Reverse engineer need some time to get into it.
This is also called *opaque predicate*.
Another example (and again, developer is sure that ESI—is always zero):

```
add     eax, ebx        ; real code
mul     ecx             ; real code
add     eax, esi        ; opaque predicate.
```

30.2.4 Making a lot of mess

```
instruction 1
instruction 2
instruction 3
```

Can be replaced to:

```
begin:          jmp     ins1_label

ins2_label:     instruction 2
                jmp     ins3_label

ins3_label:     instruction 3
                jmp     exit:
```

```
ins1_label:     instruction 1
                jmp     ins2_label
exit:
```

30.2.5 Using indirect pointers

```
dummy_data1     db      100h dup (0)
message1        db      'hello world',0

dummy_data2     db      200h dup (0)
message2        db      'another message',0

func            proc
                ...
                mov     eax, offset dummy_data1 ; PE or ELF ↙
    ↳ reloc here
                add     eax, 100h
                push    eax
                call    dump_string
                ...
                mov     eax, offset dummy_data2 ; PE or ELF ↙
    ↳ reloc here
                add     eax, 200h
                push    eax
                call    dump_string
                ...
func            endp
```

IDA will show references only to `dummy_data1` and `dummy_data2`, but not to the text strings.

Global variables and even functions may be accessed like that.

30.3 Virtual machine / pseudo-code

Programmer may construct his/her own PL or ISA and interpreter for it. (Like pre-5.0 Visual Basic, .NET, Java machine). Reverse engineerwill have to spend some time to understand meaning and details of all ISA instructions. Probably, he/she will also need to write a disassembler/decompiler of some sort.

30.4 Other thing to mention

My own (yet weak) attempt to patch Tiny C compiler to produce obfuscated code: http://blog.yurichev.com/node/58.

Using `MOV` instruction for really complicated things: [8].

Chapter 31

Windows 16-bit

16-bit Windows program are rare nowadays, but in the sense of retrocomputing, or dongle hacking (55), I sometimes digging into these.

16-bit Windows versions were up to 3.11. 96/98/ME also support 16-bit code, as well as 32-bit versions of Windows NT line. 64-bit versions of Windows NT line are not support 16-bit executable code at all.

The code is resembling MS-DOS one.

Executable files has not MZ-type, nor PE-type, they are NE-type (so-called "new executable").

All examples considered here were compiled by OpenWatcom 1.9 compiler, using these switches:

```
wcl.exe -i=C:/WATCOM/h/win/ -s -os -bt=windows -bcl=windows example.c
```

31.1 Example#1

```
#include <windows.h>

int PASCAL WinMain( HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow )
{
        MessageBeep(MB_ICONEXCLAMATION);
        return 0;
};
```

```
WinMain         proc near
                push    bp
```

```
                mov     bp, sp
                mov     ax, 30h ; '0'   ; MB_ICONEXCLAMATION ↙
    ↳ constant
                push    ax
                call    MESSAGEBEEP
                xor     ax, ax          ; return 0
                pop     bp
                retn    0Ah
WinMain         endp
```

Seems to be easy, so far.

31.2 Example #2

```
#include <windows.h>

int PASCAL WinMain( HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow )
{
        MessageBox (NULL, "hello, world", "caption", ↙
    ↳ MB_YESNOCANCEL);
        return 0;
};
```

```
WinMain         proc near
                push    bp
                mov     bp, sp
                xor     ax, ax          ; NULL
                push    ax
                push    ds
                mov     ax, offset aHelloWorld ; 0x18. "hello, ↙
    ↳ world"
                push    ax
                push    ds
                mov     ax, offset aCaption ; 0x10. "caption"
                push    ax
                mov     ax, 3           ; MB_YESNOCANCEL
                push    ax
                call    MESSAGEBOX
                xor     ax, ax          ; return 0
```

```
                pop     bp
                retn    0Ah
WinMain         endp

dseg02:0010 aCaption        db 'caption',0
dseg02:0018 aHelloWorld     db 'hello, world',0
```

Couple important things here: PASCAL calling convention dictates passing the last argument first (MB_YESNOCANCEL), and the first argument—last (NULL). This convention also tells callee to restore stack pointer: hence RETN instruction has 0Ah argument, meaning pointer should be shifted above by 10 bytes upon function exit.

Pointers are passed by pairs: a segment of data is first passed, then the pointer inside of segment. Here is only one segment in this example, so DS is always pointing to data segment of executable.

31.3 Example #3

```
#include <windows.h>

int PASCAL WinMain( HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow )
{
    int result=MessageBox (NULL, "hello, world", "caption", ↙
    ↳ MB_YESNOCANCEL);

    if (result==IDCANCEL)
        MessageBox (NULL, "you pressed cancel", "caption", MB_OK↙
    ↳ );
    else if (result==IDYES)
        MessageBox (NULL, "you pressed yes", "caption", MB_OK);
    else if (result==IDNO)
        MessageBox (NULL, "you pressed no", "caption", MB_OK);

    return 0;
};
```

```
WinMain         proc near
                push    bp
                mov     bp, sp
                xor     ax, ax      ; NULL
```

```
                push    ax
                push    ds
                mov     ax, offset aHelloWorld ; "hello, world"
                push    ax
                push    ds
                mov     ax, offset aCaption ; "caption"
                push    ax
                mov     ax, 3           ; MB_YESNOCANCEL
                push    ax
                call    MESSAGEBOX
                cmp     ax, 2           ; IDCANCEL
                jnz     short loc_2F
                xor     ax, ax
                push    ax
                push    ds
                mov     ax, offset aYouPressedCanc ; "you ↙
    ↳ pressed cancel"
                jmp     short loc_49
loc_2F:
                cmp     ax, 6           ; IDYES
                jnz     short loc_3D
                xor     ax, ax
                push    ax
                push    ds
                mov     ax, offset aYouPressedYes ; "you pressed↙
    ↳  yes"
                jmp     short loc_49
loc_3D:
                cmp     ax, 7           ; IDNO
                jnz     short loc_57
                xor     ax, ax
                push    ax
                push    ds
                mov     ax, offset aYouPressedNo ; "you pressed ↙
    ↳ no"
loc_49:
                push    ax
                push    ds
                mov     ax, offset aCaption ; "caption"
                push    ax
                xor     ax, ax
                push    ax
                call    MESSAGEBOX
```

```
loc_57:
                xor     ax, ax
                pop     bp
                retn    0Ah
WinMain         endp
```

Somewhat extended example from the previous section.

31.4 Example #4

```
#include <windows.h>

int PASCAL func1 (int a, int b, int c)
{
        return a*b+c;
};

long PASCAL func2 (long a, long b, long c)
{
        return a*b+c;
};

long PASCAL func3 (long a, long b, long c, int d)
{
        return a*b+c-d;
};

int PASCAL WinMain( HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow )
{
        func1 (123, 456, 789);
        func2 (600000, 700000, 800000);
        func3 (600000, 700000, 800000, 123);
        return 0;
};
```

```
func1           proc near

c               = word ptr  4
b               = word ptr  6
```

```
a               = word ptr  8

                push    bp
                mov     bp, sp
                mov     ax, [bp+a]
                imul    [bp+b]
                add     ax, [bp+c]
                pop     bp
                retn    6
func1           endp

func2           proc near

arg_0           = word ptr  4
arg_2           = word ptr  6
arg_4           = word ptr  8
arg_6           = word ptr  0Ah
arg_8           = word ptr  0Ch
arg_A           = word ptr  0Eh

                push    bp
                mov     bp, sp
                mov     ax, [bp+arg_8]
                mov     dx, [bp+arg_A]
                mov     bx, [bp+arg_4]
                mov     cx, [bp+arg_6]
                call    sub_B2 ; long 32-bit multiplication
                add     ax, [bp+arg_0]
                adc     dx, [bp+arg_2]
                pop     bp
                retn    12
func2           endp

func3           proc near

arg_0           = word ptr  4
arg_2           = word ptr  6
arg_4           = word ptr  8
arg_6           = word ptr  0Ah
arg_8           = word ptr  0Ch
arg_A           = word ptr  0Eh
arg_C           = word ptr  10h
```

```
                push    bp
                mov     bp, sp
                mov     ax, [bp+arg_A]
                mov     dx, [bp+arg_C]
                mov     bx, [bp+arg_6]
                mov     cx, [bp+arg_8]
                call    sub_B2 ; long 32-bit multiplication
                mov     cx, [bp+arg_2]
                add     cx, ax
                mov     bx, [bp+arg_4]
                adc     bx, dx          ; BX=high part, CX=low ↙
    ↳ part
                mov     ax, [bp+arg_0]
                cwd                     ; AX=low part d, DX=high↙
    ↳  part d
                sub     cx, ax
                mov     ax, cx
                sbb     bx, dx
                mov     dx, bx
                pop     bp
                retn    14
func3           endp

WinMain         proc near
                push    bp
                mov     bp, sp
                mov     ax, 123
                push    ax
                mov     ax, 456
                push    ax
                mov     ax, 789
                push    ax
                call    func1
                mov     ax, 9       ; high part of 600000
                push    ax
                mov     ax, 27C0h   ; low part of 600000
                push    ax
                mov     ax, 0Ah     ; high part of 700000
                push    ax
                mov     ax, 0AE60h ; low part of 700000
                push    ax
                mov     ax, 0Ch     ; high part of 800000
                push    ax
```

```
                mov     ax, 3500h  ; low part of 800000
                push    ax
                call    func2
                mov     ax, 9      ; high part of 600000
                push    ax
                mov     ax, 27C0h  ; low part of 600000
                push    ax
                mov     ax, 0Ah    ; high part of 700000
                push    ax
                mov     ax, 0AE60h ; low part of 700000
                push    ax
                mov     ax, 0Ch    ; high part of 800000
                push    ax
                mov     ax, 3500h  ; low part of 800000
                push    ax
                mov     ax, 7Bh    ; 123
                push    ax
                call    func3
                xor     ax, ax     ; return 0
                pop     bp
                retn    0Ah
WinMain         endp
```

32-bit values (`long` data type mean 32-bit, while *int* is fixed on 16-bit data type) in 16-bit code (both MS-DOS and Win16) are passed by pairs. It is just like 64-bit values are used in 32-bit environment (21).

`sub_B2 here` is a library function written by compiler developers, doing "long multiplication", i.e., multiplies two 32-bit values. Other compiler functions doing the same are listed here: E, D.

`ADD`/`ADC` instruction pair is used for addition of compound values: `ADD` may set/clear `CF` carry flag, `ADC` will use it. `SUB`/`SBB` instruction pair is used for subtraction: `SUB` may set/clear `CF` flag, `SBB` will use it.

32-bit values are returned from functions in `DX`:`AX` register pair.

Constant also passed by pairs in `WinMain()` here.

int-typed 123 constant is first converted respecting its sign into 32-bit value using `CWD` instruction.

31.5 Example #5

```
#include <windows.h>

int PASCAL string_compare (char *s1, char *s2)
```

```
{
        while (1)
        {
                if (*s1!=*s2)
                        return 0;
                if (*s1==0 || *s2==0)
                        return 1; // end of string
                s1++;
                s2++;
        };

};

int PASCAL string_compare_far (char far *s1, char far *s2)
{
        while (1)
        {
                if (*s1!=*s2)
                        return 0;
                if (*s1==0 || *s2==0)
                        return 1; // end of string
                s1++;
                s2++;
        };

};

void PASCAL remove_digits (char *s)
{
        while (*s)
        {
                if (*s>='0' && *s<='9')
                        *s='-';
                s++;
        };
};

char str[]="hello 1234 world";

int PASCAL WinMain( HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow )
```

```
{
        string_compare ("asd", "def");
        string_compare_far ("asd", "def");
        remove_digits (str);
        MessageBox (NULL, str, "caption", MB_YESNOCANCEL);
        return 0;
};
```

```
string_compare  proc near

arg_0 = word ptr  4
arg_2 = word ptr  6

      push    bp
      mov     bp, sp
      push    si
      mov     si, [bp+arg_0]
      mov     bx, [bp+arg_2]

loc_12: ; CODE XREF: string_compare+21j
      mov     al, [bx]
      cmp     al, [si]
      jz      short loc_1C
      xor     ax, ax
      jmp     short loc_2B


loc_1C: ; CODE XREF: string_compare+Ej
      test    al, al
      jz      short loc_22
      jnz     short loc_27

loc_22: ; CODE XREF: string_compare+16j
      mov     ax, 1
      jmp     short loc_2B


loc_27: ; CODE XREF: string_compare+18j
      inc     bx
      inc     si
      jmp     short loc_12
```

```
loc_2B: ; CODE XREF: string_compare+12j
        ; string_compare+1Dj
      pop     si
      pop     bp
      retn    4
string_compare  endp

string_compare_far proc near ; CODE XREF: WinMain+18p

arg_0 = word ptr  4
arg_2 = word ptr  6
arg_4 = word ptr  8
arg_6 = word ptr  0Ah

      push    bp
      mov     bp, sp
      push    si
      mov     si, [bp+arg_0]
      mov     bx, [bp+arg_4]

loc_3A: ; CODE XREF: string_compare_far+35j
      mov     es, [bp+arg_6]
      mov     al, es:[bx]
      mov     es, [bp+arg_2]
      cmp     al, es:[si]
      jz      short loc_4C
      xor     ax, ax
      jmp     short loc_67


loc_4C: ; CODE XREF: string_compare_far+16j
      mov     es, [bp+arg_6]
      cmp     byte ptr es:[bx], 0
      jz      short loc_5E
      mov     es, [bp+arg_2]
      cmp     byte ptr es:[si], 0
      jnz     short loc_63

loc_5E: ; CODE XREF: string_compare_far+23j
      mov     ax, 1
      jmp     short loc_67
```

```
loc_63: ; CODE XREF: string_compare_far+2Cj
      inc     bx
      inc     si
      jmp     short loc_3A


loc_67: ; CODE XREF: string_compare_far+1Aj
        ; string_compare_far+31j
      pop     si
      pop     bp
      retn    8
string_compare_far endp

remove_digits   proc near ; CODE XREF: WinMain+1Fp

arg_0 = word ptr  4

      push    bp
      mov     bp, sp
      mov     bx, [bp+arg_0]

loc_72: ; CODE XREF: remove_digits+18j
      mov     al, [bx]
      test    al, al
      jz      short loc_86
      cmp     al, 30h ; '0'
      jb      short loc_83
      cmp     al, 39h ; '9'
      ja      short loc_83
      mov     byte ptr [bx], 2Dh ; '-'

loc_83: ; CODE XREF: remove_digits+Ej
        ; remove_digits+12j
      inc     bx
      jmp     short loc_72


loc_86: ; CODE XREF: remove_digits+Aj
      pop     bp
      retn    2
remove_digits   endp

WinMain proc near ; CODE XREF: start+EDp
```

```
        push    bp
        mov     bp, sp
        mov     ax, offset aAsd ; "asd"
        push    ax
        mov     ax, offset aDef ; "def"
        push    ax
        call    string_compare
        push    ds
        mov     ax, offset aAsd ; "asd"
        push    ax
        push    ds
        mov     ax, offset aDef ; "def"
        push    ax
        call    string_compare_far
        mov     ax, offset aHello1234World ; "hello 1234 world"
        push    ax
        call    remove_digits
        xor     ax, ax
        push    ax
        push    ds
        mov     ax, offset aHello1234World ; "hello 1234 world"
        push    ax
        push    ds
        mov     ax, offset aCaption ; "caption"
        push    ax
        mov     ax, 3 ; MB_YESNOCANCEL
        push    ax
        call    MESSAGEBOX
        xor     ax, ax
        pop     bp
        retn    0Ah
WinMain endp
```

Here we see a difference between so-called “near” pointers and “far” pointers: another weird artefact of segmented memory of 16-bit 8086.

Read more about it: 67.

“near” pointers are those which points within current data segment. Hence, `string_compa` function takes only two 16-bit pointers, and accesses data as it is located in the segment `DS` pointing to (`mov al, [bx]` instruction actually works like `mov al, ds:[bx]` — `DS` is implicitly used here).

“far” pointers are those which may point to data in another segment memory. Hence `string_compare_far()` takes 16-bit pair as a pointer, loads high part of it to `ES` segment register and accessing data through it (`mov al, es:[bx]`). “far” pointers are also used in my `MessageBox()` win16 example: 31.2. Indeed, Windows kernel is

not aware which data segment to use when accessing text strings, so it need more complete information.

The reason for this distinction is that compact program may use just one 64kb data segment, so it doesn't need to pass high part of the address, which is always the same. Bigger program may use several 64kb data segments, so it needs to specify each time, in which segment data is located.

The same story for code segments. Compact program may have all executable code within one 64kb-segment, then all functions will be called in it using `CALL NEAR` instruction, and code flow will be returned using `RETN`. But if there are several code segments, then the address of the function will be specified by pair, it will be called using `CALL FAR` instruction, and the code flow will be returned using `RETF`.

This is what to be set in compiler by specifying "memory model".

Compilers targeting MS-DOS and Win16 has specific libraries for each memory model: they were differ by pointer types for code and data.

31.6 Example #6

```
#include <windows.h>
#include <time.h>
#include <stdio.h>

char strbuf[256];

int PASCAL WinMain( HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow )
{

        struct tm *t;
        time_t unix_time;

        unix_time=time(NULL);

        t=localtime (&unix_time);

        sprintf (strbuf, "%04d-%02d-%02d %02d:%02d:%02d", t->↙
    ↳ tm_year+1900, t->tm_mon, t->tm_mday,
                t->tm_hour, t->tm_min, t->tm_sec);

        MessageBox (NULL, strbuf, "caption", MB_OK);
        return 0;
```

```
};
```

```
WinMain         proc near

var_4           = word ptr -4
var_2           = word ptr -2

                push    bp
                mov     bp, sp
                push    ax
                push    ax
                xor     ax, ax
                call    time_
                mov     [bp+var_4], ax   ; low part of UNIX time
                mov     [bp+var_2], dx   ; high part of UNIX ↙
    ↳ time
                lea     ax, [bp+var_4]   ; take a pointer of ↙
    ↳ high part
                call    localtime_
                mov     bx, ax           ; t
                push    word ptr [bx]    ; second
                push    word ptr [bx+2]  ; minute
                push    word ptr [bx+4]  ; hour
                push    word ptr [bx+6]  ; day
                push    word ptr [bx+8]  ; month
                mov     ax, [bx+0Ah]     ; year
                add     ax, 1900
                push    ax
                mov     ax, offset a04d02d02d02d02 ; "%04d-%02d↙
    ↳ -%02d %02d:%02d:%02d"
                push    ax
                mov     ax, offset strbuf
                push    ax
                call    sprintf_
                add     sp, 10h
                xor     ax, ax            ; NULL
                push    ax
                push    ds
                mov     ax, offset strbuf
                push    ax
                push    ds
                mov     ax, offset aCaption ; "caption"
                push    ax
```

```
                xor     ax, ax          ; MB_OK
                push    ax
                call    MESSAGEBOX
                xor     ax, ax
                mov     sp, bp
                pop     bp
                retn    0Ah
WinMain         endp
```

UNIX time is 32-bit value, so it is returned in DX:AX register pair and stored into two local 16-bit variables. Then a pointer to the pair is passed to `localtime()` function. The `localtime()` function has `struct tm` allocated somewhere in guts of the C library, so only pointer to it is returned. By the way, this is also means that the function cannot be called again until its results are used.

For the `time()` and `localtime()` functions, a Watcom calling convention is used here: first four arguments are passed in AX, DX, BX and CX, registers, all the rest arguments are via stack. Functions used this convention are also marked by underscore at the end of name.

`sprintf()` does not use PASCAL calling convention, nor Watcom one, so the arguments are passed in usual *cdecl* way (44.1).

31.6.1 Global variables

This is the same example, but now these variables are global:

```
#include <windows.h>
#include <time.h>
#include <stdio.h>

char strbuf[256];
struct tm *t;
time_t unix_time;

int PASCAL WinMain( HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow )
{

        unix_time=time(NULL);

        t=localtime (&unix_time);
```

```
        sprintf (strbuf, "%04d-%02d-%02d %02d:%02d:%02d", t->↙
    ↪ tm_year+1900, t->tm_mon, t->tm_mday,
                t->tm_hour, t->tm_min, t->tm_sec);

        MessageBox (NULL, strbuf, "caption", MB_OK);
        return 0;
};
```

```
unix_time_low   dw 0
unix_time_high  dw 0
t               dw 0

WinMain         proc near
                push    bp
                mov     bp, sp
                xor     ax, ax
                call    time_
                mov     unix_time_low, ax
                mov     unix_time_high, dx
                mov     ax, offset unix_time_low
                call    localtime_
                mov     bx, ax
                mov     t, ax               ; will not be used ↙
    ↪ in future...
                push    word ptr [bx]       ; seconds
                push    word ptr [bx+2]     ; minutes
                push    word ptr [bx+4]     ; hour
                push    word ptr [bx+6]     ; day
                push    word ptr [bx+8]     ; month
                mov     ax, [bx+0Ah]        ; year
                add     ax, 1900
                push    ax
                mov     ax, offset a04d02d02d02d02 ; "%04d-%02d↙
    ↪ -%02d %02d:%02d:%02d"
                push    ax
                mov     ax, offset strbuf
                push    ax
                call    sprintf_
                add     sp, 10h
                xor     ax, ax          ; NULL
                push    ax
                push    ds
                mov     ax, offset strbuf
```

```
                push    ax
                push    ds
                mov     ax, offset aCaption ; "caption"
                push    ax
                xor     ax, ax          ; MB_OK
                push    ax
                call    MESSAGEBOX
                xor     ax, ax          ; return 0
                pop     bp
                retn    0Ah
WinMain         endp
```

t will not be used, but compiler emitted the code which stores the value. Because it is not sure, maybe that value will be eventually used somewhere.

Part II

Important fundamentals

Chapter 32

Signed number representations

There are several methods of representing signed numbers[1], but in x86 architecture used "two's complement".

binary	hexadecimal	unsigned	signed (2's complement)
01111111	0x7f	127	127
01111110	0x7e	126	126
...			
00000010	0x2	2	2
00000001	0x1	1	1
00000000	0x0	0	0
11111111	0xff	255	-1
11111110	0xfe	254	-2
...			
10000010	0x82	130	-126
10000001	0x81	129	-127
10000000	0x80	128	-128

The difference between signed and unsigned numbers is that if we represent `0xFFFFFFFE` and `0x0000002` as unsigned, then first number (4294967294) is bigger than second (2). If to represent them both as signed, first will be -2, and it is lesser than second (2). That is the reason why conditional jumps (10) are present both for signed (e.g. `JG`, `JL`) and unsigned (`JA`, `JBE`) operations.

For the sake of simplicity, that is what one need to know:

- Number can be signed or unsigned.

[1] http://en.wikipedia.org/wiki/Signed_number_representations

- C/C++ signed types: *int* (-2147483646..2147483647 or `0x80000000..0x7FFFFFFF`), *char* (-127..128 or `0x7F..0x80`). Unsigned: `unsigned int` (0..4294967295 or `0..0xFFFFFFFF`), `unsigned char` (0..255 or `0..0xFF`), `size_t`.
- Signed types has sign in the most significant bit: 1 mean "minus", 0 mean "plus".
- Addition and subtraction operations are working well for both signed and unsigned values. But for multiplication and division operations, x86 has different instructions: `IDIV`/`IMUL` for signed and `DIV`/`MUL` for unsigned.
- More instructions working with signed numbers: `CBW`/`CWD`/`CWDE`/`CDQ`/`CDQE` (B.6.3), `MOVSX` (13.1.1), `SAR` (B.6.3).

32.1 Integer overflow

It is worth noting that incorrect representation of number can lead integer overflow vulnerability.

For example, we have a network service, it receives network packets. In the packets there is also a field where subpacket length is coded. It is 32-bit value. After network packet received, service checking the field, and if it is larger than, e.g. some `MAX_PACKET_SIZE` (let's say, 10 kilobytes), the packet is rejected as incorrect. Comparison is signed. Intruder set this value to the `0xFFFFFFFF`. While comparison, this number is considered as signed -1 and it is lesser than 10 kilobytes. No error here. Service would like to copy the subpacket to another place in memory and call `memcpy (dst, src, 0xFFFFFFFF)` function: this operation, rapidly garbling a lot of inside of process memory.

More about it: [3].

Chapter 33

Endianness

Endianness is a way of representing values in memory.

33.1 Big-endian

A 0x12345678 value will be represented in memory as:

address in memory	byte value
+0	0x12
+1	0x34
+2	0x56
+3	0x78

Big-endian CPUs are including Motorola 68k, IBM POWER.

33.2 Little-endian

A 0x12345678 value will be represented in memory as:

address in memory	byte value
+0	0x78
+1	0x56
+2	0x34
+3	0x12

Little-endian CPUs are including Intel x86.

33.3 Bi-endian

CPUs which may switch between endianness are ARM, PowerPC, SPARC, MIPS, IA64[1], etc.

33.4 Converting data

TCP/IP network data packets are used big-endian conventions, so that is why a program working on little-endian architecture should convert values using `htonl()` and `htons()` functions.

Big-endian convention in the TCP/IP environment is also called "network byte order", while little-endian—"host byte order".

`BSWAP` instruction is also can be used for the conversion.

[1] Intel Architecture 64 (Itanium): 66

Part III

Finding important/interesting stuff in the code

Minimalism it is not a significant feature of modern software.

But not because programmers are writing a lot, but in a reason that all libraries are commonly linked statically to executable files. If all external libraries were shifted into external DLL files, the world would be different. (Another reason for C++ —STL and other template libraries.)

Thus, it is very important to determine origin of a function, if it is from standard library or well-known library (like Boost[2], libpng[3]), and which one —is related to what we are trying to find in the code.

It is just absurdly to rewrite all code to C/C++ to find what we looking for.

One of the primary reverse engineer's task is to find quickly in the code what is needed.

IDA disassembler allow us search among text strings, byte sequences, constants. It is even possible to export the code into .lst or .asm text file and then use `grep`, `awk`, etc.

When you try to understand what a code is doing, this easily could be some open-source library like libpng. So when you see some constants or text strings looks familiar, it is always worth to *google* it. And if you find the opensource project where it is used, then it will be enough just to compare the functions. It may solve some part of problem.

For example, if program use a XML files, the first step may be determining, which XML-library is used for processing, since standard (or well-known) library is usually used instead of self-made one.

For example, once upon a time I tried to understand how SAP 6.0 network packets compression/decompression is working. It is a huge software, but a detailed .PDB with debugging information is present, and that is cozily. I finally came to idea that one of the functions doing decompressing of network packet called CsDecomprLZC(). Immediately I tried to google its name and I quickly found the function named as the same is used in MaxDB (it is open-source SAP project)[4].

`http://www.google.com/search?q=CsDecomprLZC`

Astoundingly, MaxDB and SAP 6.0 software shared likewise code for network packets compression/decompression.

[2] `http://www.boost.org/`

[3] `http://www.libpng.org/pub/png/libpng.html`

[4] More about it in relevant section (57.1)

Chapter 34

Identification of executable files

34.1 Microsoft Visual C++

MSVC versions and DLLs which may be imported:

Marketing version	Internal version	CL.EXE version	DLLs may be imported	Re
6	6.0	12.00	msvcrt.dll, msvcp60.dll	Ju
.NET (2002)	7.0	13.00	msvcr70.dll, msvcp70.dll	Fe
.NET 2003	7.1	13.10	msvcr71.dll, msvcp71.dll	Ap
2005	8.0	14.00	msvcr80.dll, msvcp80.dll	No
2008	9.0	15.00	msvcr90.dll, msvcp90.dll	No
2010	10.0	16.00	msvcr100.dll, msvcp100.dll	Ap
2012	11.0	17.00	msvcr110.dll, msvcp110.dll	Se
2013	12.0	18.00	msvcr120.dll, msvcp120.dll	Oc

msvcp*.dll contain C++-related functions, so, if it is imported, this is probably C++ program.

34.1.1 Name mangling

Names are usually started with ? symbol.

Read more about MSVC name mangling here: 29.1.1.

34.2 GCC

Aside from *NIX targets, GCC is also present in win32 environment: in form of Cygwin and MinGW.

34.2.1 Name mangling

Names are usually started with `_Z` symbols.

Read more about GCC name mangling here: 29.1.1.

34.2.2 Cygwin

cygwin1.dll is often imported.

34.2.3 MinGW

msvcrt.dll may be imported.

34.3 Intel FORTRAN

libifcoremd.dll, libifportmd.dll and libiomp5md.dll (OpenMP support) may be imported.

libifcoremd.dll has a lot of functions prefixed with `for_`, meaning FORTRAN.

34.4 Watcom, OpenWatcom

34.4.1 Name mangling

Names are usually started with `W` symbol.

For example, that is how method named "method" of the class "class" not having arguments and returning *void* is encoded to:

```
W?method$_class$n__v
```

34.5 Borland

Here is an example of Borland Delphi and C++Builder name mangling:

```
@TApplication@IdleAction$qv
@TApplication@ProcessMDIAccels$qp6tagMSG
@TModule@$bctr$qpcpvt1
@TModule@$bdtr$qv
```

```
@TModule@ValidWindow$qp14TWindowsObject
@TrueColorTo8BitN$qpviiiiiit1iiiiii
@TrueColorTo16BitN$qpviiiiiit1iiiiii
@DIB24BitTo8BitBitmap$qpviiiiiit1iiiii
@TrueBitmap@$bctr$qpcl
@TrueBitmap@$bctr$qpvl
@TrueBitmap@$bctr$qiilll
```

Names are always started with `@` symbol, then class name came, method name, and encoded method argument types.

These names can be in .exe imports, .dll exports, debug data, etc.

Borland Visual Component Libraries (VCL) are stored in .bpl files instead of .dll ones, for example, vcl50.dll, rtl60.dll.

Other DLL might be imported: BORLNDMM.DLL.

34.5.1 Delphi

Almost all Delphi executables has "Boolean" text string at the very beginning of code segment, along with other type names.

This is a very typical beginning of `.text` segment of a Delphi program, this block came right after win32 PE file header:

```
00000400  04 10 40 00 03 07 42 6f  6f 6c 65 61 6e 01 00 00  |..@.|
    ↳ ...Boolean...|
00000410  00 00 01 00 00 00 00 10  40 00 05 46 61 6c 73 65  |
    ↳ |........@..False|
00000420  04 54 72 75 65 8d 40 00  2c 10 40 00 09 08 57 69  |.|
    ↳ True.@.,.@...Wi|
00000430  64 65 43 68 61 72 03 00  00 00 00 ff ff 00 00 90  |
    ↳ deChar..........|
00000440  44 10 40 00 02 04 43 68  61 72 01 00 00 00 00 ff  |D.@|
    ↳ ...Char......|
00000450  00 00 00 90 58 10 40 00  01 08 53 6d 61 6c 6c 69  |
    ↳ |....X.@...Smalli|
00000460  6e 74 02 00 80 ff ff ff  7f 00 00 90 70 10 40 00  |nt|
    ↳ ..........p.@.|
00000470  01 07 49 6e 74 65 67 65  72 04 00 00 00 80 ff ff  |..|
    ↳ Integer.......|
00000480  ff 7f 8b c0 88 10 40 00  01 04 42 79 74 65 01 00  |
    ↳ |......@...Byte..|
00000490  00 00 00 ff 00 00 00 90  9c 10 40 00 01 04 57 6f  |
    ↳ |..........@...Wo|
000004a0  72 64 03 00 00 00 00 ff  ff 00 00 90 b0 10 40 00  |rd|
    ↳ ............@.|
```

```
000004b0  01 08 43 61 72 64 69 6e  61 6c 05 00 00 00 00 ff  |..↙
    ↳ Cardinal......|
000004c0  ff ff ff 90 c8 10 40 00  10 05 49 6e 74 36 34 00  ↙
    ↳ |......@...Int64.|
000004d0  00 00 00 00 00 00 80 ff  ff ff ff ff ff ff 7f 90  ↙
    ↳ |................|
000004e0  e4 10 40 00 04 08 45 78  74 65 6e 64 65 64 02 90  |..@↙
    ↳ ...Extended..|
000004f0  f4 10 40 00 04 06 44 6f  75 62 6c 65 01 8d 40 00  |..@↙
    ↳ ...Double..@.|
00000500  04 11 40 00 04 08 43 75  72 72 65 6e 63 79 04 90  |..@↙
    ↳ ...Currency..|
00000510  14 11 40 00 0a 06 73 74  72 69 6e 67 20 11 40 00  |..@↙
    ↳ ...string .@.|
00000520  0b 0a 57 69 64 65 53 74  72 69 6e 67 30 11 40 00  |..↙
    ↳ WideString0.@.|
00000530  0c 07 56 61 72 69 61 6e  74 8d 40 00 40 11 40 00  |..↙
    ↳ Variant.@.@.@.|
00000540  0c 0a 4f 6c 65 56 61 72  69 61 6e 74 98 11 40 00  |..↙
    ↳ OleVariant..@.|
00000550  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ↙
    ↳ |................|
00000560  00 00 00 00 00 00 00 00  00 00 00 00 98 11 40 00  ↙
    ↳ |..............@.|
00000570  04 00 00 00 00 00 00 00  18 4d 40 00 24 4d 40 00  ↙
    ↳ |.........M@.$M@.|
00000580  28 4d 40 00 2c 4d 40 00  20 4d 40 00 68 4a 40 00  |(M@↙
    ↳ .,M@. M@.hJ@.|
00000590  84 4a 40 00 c0 4a 40 00  07 54 4f 62 6a 65 63 74  |.J@↙
    ↳ ..J@..TObject|
000005a0  a4 11 40 00 07 07 54 4f  62 6a 65 63 74 98 11 40  |..@↙
    ↳ ...TObject..@|
000005b0  00 00 00 00 00 00 00 06  53 79 73 74 65 6d 00 00  ↙
    ↳ |........System..|
000005c0  c4 11 40 00 0f 0a 49 49  6e 74 65 72 66 61 63 65  |..@↙
    ↳ ...IInterface|
000005d0  00 00 00 00 01 00 00 00  00 00 00 00 00 c0 00 00  ↙
    ↳ |................|
000005e0  00 00 00 00 46 06 53 79  73 74 65 6d 03 00 ff ff  ↙
    ↳ |....F.System....|
000005f0  f4 11 40 00 0f 09 49 44  69 73 70 61 74 63 68 c0  |..@↙
    ↳ ...IDispatch.|
00000600  11 40 00 01 00 04 02 00  00 00 00 00 c0 00 00 00  |.@↙
```

```
    ↪ ..............|
00000610  00 00 00 46 06 53 79 73  74 65 6d 04 00 ff ff 90  |...↙
    ↪ F.System.....|
00000620  cc 83 44 24 04 f8 e9 51  6c 00 00 83 44 24 04 f8  |..↙
    ↪ D$...Ql...D$..|
00000630  e9 6f 6c 00 00 83 44 24  04 f8 e9 79 6c 00 00 cc  |.ol↙
    ↪ ...D$...yl...|
00000640  cc 21 12 40 00 2b 12 40  00 35 12 40 00 01 00 00  |.!.↙
    ↪ @.+.@.5.@....|
00000650  00 00 00 00 00 00 00 00  00 c0 00 00 00 00 00 00  ↙
    ↪ |................|
00000660  46 41 12 40 00 08 00 00  00 00 00 00 00 8d 40 00  |FA.↙
    ↪ @..........@.|
00000670  bc 12 40 00 4d 12 40 00  00 00 00 00 00 00 00 00  |..@↙
    ↪ .M.@.........|
00000680  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ↙
    ↪ |................|
00000690  bc 12 40 00 0c 00 00 00  4c 11 40 00 18 4d 40 00  |..@↙
    ↪ .....L.@..M@.|
000006a0  50 7e 40 00 5c 7e 40 00  2c 4d 40 00 20 4d 40 00  |P~@↙
    ↪ .\~@.,M@. M@.|
000006b0  6c 7e 40 00 84 4a 40 00  c0 4a 40 00 11 54 49 6e  |l~@↙
    ↪ ..J@..J@..TIn|
000006c0  74 65 72 66 61 63 65 64  4f 62 6a 65 63 74 8b c0  |↙
    ↪ terfacedObject..|
000006d0  d4 12 40 00 07 11 54 49  6e 74 65 72 66 61 63 65  |..@↙
    ↪ ...TInterface|
000006e0  64 4f 62 6a 65 63 74 bc  12 40 00 a0 11 40 00 00  |↙
    ↪ dObject..@...@..|
000006f0  00 06 53 79 73 74 65 6d  00 00 8b c0 00 13 40 00  |..↙
    ↪ System......@.|
00000700  11 0b 54 42 6f 75 6e 64  41 72 72 61 79 04 00 00  |..↙
    ↪ TBoundArray...|
00000710  00 00 00 00 00 03 00 00  00 6c 10 40 00 06 53 79  ↙
    ↪ |.........l.@..Sy|
00000720  73 74 65 6d 28 13 40 00  04 09 54 44 61 74 65 54  |↙
    ↪ stem(.@...TDateT|
00000730  69 6d 65 01 ff 25 48 e0  c4 00 8b c0 ff 25 44 e0  |ime↙
    ↪ ..%H......%D.|
```

34.6 Other known DLLs

- vcomp*.dll—Microsoft implementation of OpenMP.

Chapter 35

Communication with the outer world (win32)

Files and registry access: for the very basic analysis, Process Monitor[1] utility from SysInternals may help.

For the basic analysis of network accesses, Wireshark[2] may help.

But then you will need to look inside anyway.

First what to look on is which functions from OS API[3] and standard libraries are used.

If the program is divided into main executable file and a group of DLL-files, sometimes, these function's names may be helpful.

If we are interesting, what exactly may lead to the `MessageBox()` call with specific text, first what we can try to do: find this text in data segment, find references to it and find the points from which a control may be passed to the `MessageBox()` call we're interesting in.

If we are talking about a video game and we're interesting, which events are more or less random in it, we may try to find `rand()` function or its replacement (like Mersenne twister algorithm) and find a places from which this function called and most important: how the results are used.

But if it is not a game, but `rand()` is used, it is also interesting, why. There are cases of unexpected `rand()` usage in data compression algorithm (for encryption imitation): http://blog.yurichev.com/node/44.

[1] http://technet.microsoft.com/en-us/sysinternals/bb896645.aspx

[2] http://www.wireshark.org/

[3] Application programming interface

35.1 Often used functions in Windows API

These functions may be among imported. It is worth to note that not every function might be used by the code written by author. A lot of functions might be called from library functions and CRT code.

- Registry access (advapi32.dll): RegEnumKeyEx[4] [5], RegEnumValue[6] [5], RegGetValue[7] [5], RegOpenKeyEx[8] [5], RegQueryValueEx[9] [5].
- Access to text .ini-files (kernel32.dll): GetPrivateProfileString [10] [5].
- Dialog boxes (user32.dll): MessageBox [11] [5], MessageBoxEx [12] [5], SetDlgItemText [13] [5], GetDlgItemText [14] [5].
- Resources access(48.2.8): (user32.dll): LoadMenu [15] [5].
- TCP/IP-network (ws2_32.dll): WSARecv [16], WSASend [17].
- File access (kernel32.dll): CreateFile [18] [5], ReadFile [19], ReadFileEx [20], WriteFile [21], WriteFileEx [22].
- High-level access to the Internet (wininet.dll): WinHttpOpen [23].
- Check digital signature of a executable file (wintrust.dll): WinVerifyTrust [24].
- Standard MSVC library (in case of dynamic linking) (msvcr*.dll): assert, itoa, ltoa, open, printf, read, strcmp, atol, atoi, fopen, fread, fwrite, memcmp, rand, strlen, strstr, strchr.

[4] http://msdn.microsoft.com/en-us/library/windows/desktop/ms724862(v=vs.85).aspx
[5] May have -A suffix for ASCII-version and -W for Unicode-version
[6] http://msdn.microsoft.com/en-us/library/windows/desktop/ms724865(v=vs.85).aspx
[7] http://msdn.microsoft.com/en-us/library/windows/desktop/ms724868(v=vs.85).aspx
[8] http://msdn.microsoft.com/en-us/library/windows/desktop/ms724897(v=vs.85).aspx
[9] http://msdn.microsoft.com/en-us/library/windows/desktop/ms724911(v=vs.85).aspx
[10] http://msdn.microsoft.com/en-us/library/windows/desktop/ms724353(v=vs.85).aspx
[11] http://msdn.microsoft.com/en-us/library/ms645505(VS.85).aspx
[12] http://msdn.microsoft.com/en-us/library/ms645507(v=vs.85).aspx
[13] http://msdn.microsoft.com/en-us/library/ms645521(v=vs.85).aspx
[14] http://msdn.microsoft.com/en-us/library/ms645489(v=vs.85).aspx
[15] http://msdn.microsoft.com/en-us/library/ms647990(v=vs.85).aspx
[16] http://msdn.microsoft.com/en-us/library/windows/desktop/ms741688(v=vs.85).aspx
[17] http://msdn.microsoft.com/en-us/library/windows/desktop/ms742203(v=vs.85).aspx
[18] http://msdn.microsoft.com/en-us/library/windows/desktop/aa363858(v=vs.85).aspx
[19] http://msdn.microsoft.com/en-us/library/windows/desktop/aa365467(v=vs.85).aspx
[20] http://msdn.microsoft.com/en-us/library/windows/desktop/aa365468(v=vs.85).aspx
[21] http://msdn.microsoft.com/en-us/library/windows/desktop/aa365747(v=vs.85).aspx
[22] http://msdn.microsoft.com/en-us/library/windows/desktop/aa365748(v=vs.85).aspx
[23] http://msdn.microsoft.com/en-us/library/windows/desktop/aa384098(v=vs.85).aspx
[24] http://msdn.microsoft.com/library/windows/desktop/aa388208.aspx

35.2 tracer: Intercepting all functions in specific module

There is INT3-breakpoints in tracer, triggering only once, however, they can be set to all functions in specific DLL.

```
--one-time-INT3-bp:somedll.dll!.*
```

Or, let's set INT3-breakpoints to all functions with `xml` prefix in name:

```
--one-time-INT3-bp:somedll.dll!xml.*
```

On the other side of coin, such breakpoints are triggered only once.

Tracer will show calling of a function, if it happens, but only once. Another drawback —it is impossible to see function's arguments.

Nevertheless, this feature is very useful when you know the program uses a DLL, but do not know which functions in it. And there are a lot of functions.

For example, let's see, what uptime cygwin-utility uses:

```
tracer -l:uptime.exe --one-time-INT3-bp:cygwin1.dll!.*
```

Thus we may see all cygwin1.dll library functions which were called at least once, and where from:

```
One-time INT3 breakpoint: cygwin1.dll!__main (called from uptime↙
    ↳ .exe!OEP+0x6d (0x40106d))
One-time INT3 breakpoint: cygwin1.dll!_geteuid32 (called from ↙
    ↳ uptime.exe!OEP+0xba3 (0x401ba3))
One-time INT3 breakpoint: cygwin1.dll!_getuid32 (called from ↙
    ↳ uptime.exe!OEP+0xbaa (0x401baa))
One-time INT3 breakpoint: cygwin1.dll!_getegid32 (called from ↙
    ↳ uptime.exe!OEP+0xcb7 (0x401cb7))
One-time INT3 breakpoint: cygwin1.dll!_getgid32 (called from ↙
    ↳ uptime.exe!OEP+0xcbe (0x401cbe))
One-time INT3 breakpoint: cygwin1.dll!sysconf (called from ↙
    ↳ uptime.exe!OEP+0x735 (0x401735))
One-time INT3 breakpoint: cygwin1.dll!setlocale (called from ↙
    ↳ uptime.exe!OEP+0x7b2 (0x4017b2))
One-time INT3 breakpoint: cygwin1.dll!_open64 (called from ↙
    ↳ uptime.exe!OEP+0x994 (0x401994))
One-time INT3 breakpoint: cygwin1.dll!_lseek64 (called from ↙
    ↳ uptime.exe!OEP+0x7ea (0x4017ea))
One-time INT3 breakpoint: cygwin1.dll!read (called from uptime.↙
    ↳ exe!OEP+0x809 (0x401809))
```

```
One-time INT3 breakpoint: cygwin1.dll!sscanf (called from uptime↙
    ↳ .exe!OEP+0x839 (0x401839))
One-time INT3 breakpoint: cygwin1.dll!uname (called from uptime.↙
    ↳ exe!OEP+0x139 (0x401139))
One-time INT3 breakpoint: cygwin1.dll!time (called from uptime.↙
    ↳ exe!OEP+0x22e (0x40122e))
One-time INT3 breakpoint: cygwin1.dll!localtime (called from ↙
    ↳ uptime.exe!OEP+0x236 (0x401236))
One-time INT3 breakpoint: cygwin1.dll!sprintf (called from ↙
    ↳ uptime.exe!OEP+0x25a (0x40125a))
One-time INT3 breakpoint: cygwin1.dll!setutent (called from ↙
    ↳ uptime.exe!OEP+0x3b1 (0x4013b1))
One-time INT3 breakpoint: cygwin1.dll!getutent (called from ↙
    ↳ uptime.exe!OEP+0x3c5 (0x4013c5))
One-time INT3 breakpoint: cygwin1.dll!endutent (called from ↙
    ↳ uptime.exe!OEP+0x3e6 (0x4013e6))
One-time INT3 breakpoint: cygwin1.dll!puts (called from uptime.↙
    ↳ exe!OEP+0x4c3 (0x4014c3))
```

Chapter 36

Strings

36.1 Text strings

Usual C-strings are zero-terminated (ASCIIZ-strings).

The reason why C string format is as it is (zero-terminating) is apparently hisorical. In [27] we can read:

> A minor difference was that the unit of I/O was the word, not the byte, because the PDP-7 was a word-addressed machine. In practice this meant merely that all programs dealing with character streams ignored null characters, because null was used to pad a file to an even number of characters.

In Hiew or FAR Manager these strings looks like as it is:

```
int main()
{
        printf ("Hello, world!\n");
};
```

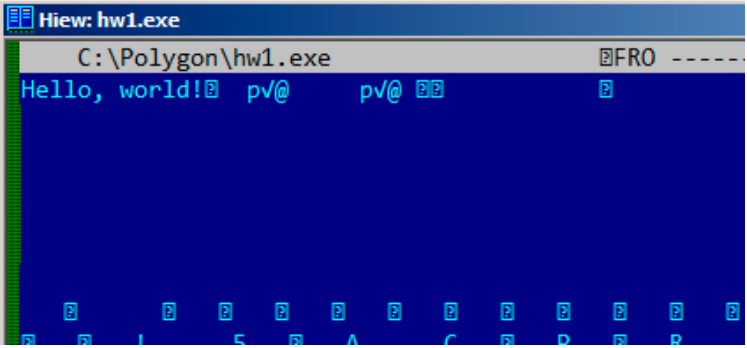


Figure 36.1: Hiew

The string is preceeded by 8-bit or 32-bit string length value.
For example:

Listing 36.1: Delphi

```
CODE:00518AC8                   dd 19h
CODE:00518ACC aLoading___Plea db 'Loading... , please wait.',0

...

CODE:00518AFC                   dd 10h
CODE:00518B00 aPreparingRun__ db 'Preparing run...',0
```

36.1.1 Unicode

Often, what is called by Unicode is a methods of strings encoding when each character occupies 2 bytes or 16 bits. This is common terminological mistake. Unicode is a standard assigning a number to each character of many writing systems of the world, but not describing encoding method.

Most popular encoding methods are: UTF-8 (often used in Internet and *NIX systems) and UTF-16LE (used in Windows).

UTF-8

UTF-8 is one of the most successful methods of character encoding. All Latin symbols are encoded just like in an ASCII-encoding, and symbols beyond ASCII-table are encoded by several bytes. 0 is encoded as it was before, so all standard C string functions works with UTF-8-strings just like any other string.

Let's see how symbols in various languages are encoded in UTF-8 and how it looks like in FAR in 437 codepage [1]:

```
How much? 100€?

(English) I can eat glass and it doesn't hurt me.
(Greek) Μπορώ να φάω σπασμένα γυαλιά χωρίς να πάθω τίποτα.
(Hungarian) Meg tudom enni az üveget, nem lesz tőle bajom.
(Icelandic) Ég get etið gler án þess að meiða mig.
(Polish) Mogę jeść szkło i mi nie szkodzi.
(Russian) Я могу есть стекло, оно мне не вредит.
(Arabic): أنا قادر على أكل الزجاج و هذا لا يؤلمني.
(Hebrew): אני יכול לאכול זכוכית וזה לא מזיק לי.
(Chinese) 我能吞下玻璃而不伤身体。
(Japanese) 私はガラスを食べられます。それは私を傷つけません。
(Hindi) मैं काँच खा सकता हूँ और मुझे उससे कोई चोट नहीं पहुंचती.
```

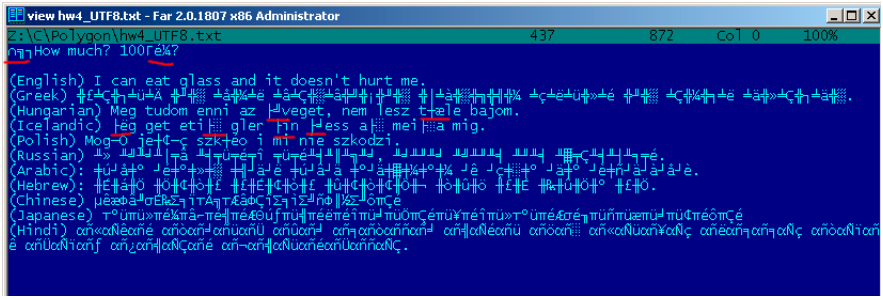


Figure 36.2: FAR: UTF-8

As it seems, English language string looks like as it is in ASCII-encoding. Hungarian language uses Latin symbols plus symbols with diacritic marks. These symbols are encoded by several bytes, I underscored them by red. The same story with Icelandic and Polish languages. I also used "Euro" currency symbol at the begin, which is encoded by 3 bytes. All the rest writing systems here have no connection with Latin. At least about Russian, Arabic, Hebrew and Hindi we could see recurring bytes, and that is not surprise: all symbols from the writing system is usually located in the same Unicode table, so their code begins with the same numbers.

At the very beginning, before "How much?" string we see 3 bytes, which is BOM[2] in fact. BOM defines encoding system to be used now.

[1] I've got example and translations from there: http://www.columbia.edu/~fdc/utf8/

[2] Byte order mark

UTF-16LE

Many win32 functions in Windows has a suffix `-A` and `-W`. The first functions works with usual strings, the next with UTF-16LE-strings (*wide*). As in the second case, each symbol is usually stored in 16-bit value of *short* type.

Latin symbols in UTF-16 strings looks in Hiew or FAR as interleaved with zero byte:

```
int wmain()
{
        wprintf (L"Hello, world!\n");
};
```

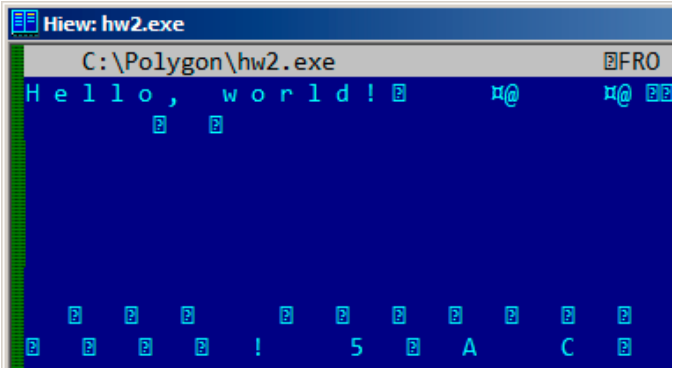


Figure 36.3: Hiew

We may often see this in Windows NT system files:

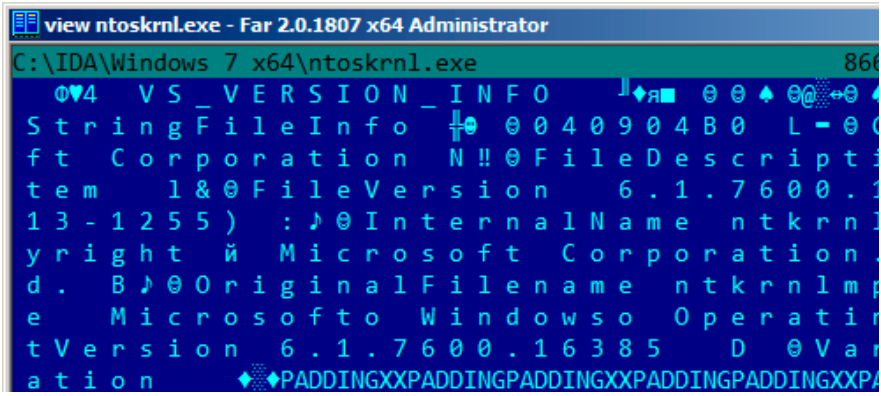


Figure 36.4: Hiew

String with characters occupying exactly 2 bytes are called by “Unicode” in IDA:

```
.data:0040E000 aHelloWorld:
.data:0040E000                 unicode 0, <Hello, world!>
.data:0040E000                 dw 0Ah, 0
```

Here is how Russian language string encoded in UTF-16LE may looks like:

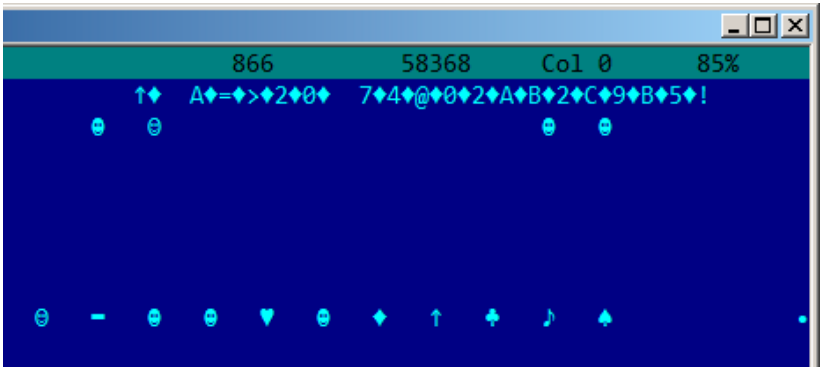


Figure 36.5: Hiew: UTF-16LE

What we can easily spot—is that symbols are interleaved by diamond character (which has code of 4). Indeed, Cyrillic symbols are located in the fourth Unicode plane [3]. Hence, all Cyrillic symbols in UTF-16LE are located in 0x400-0x4FF range.

Let’s back to the example with the string written in multiple languages. Here we can see it in UTF-16LE encoding.

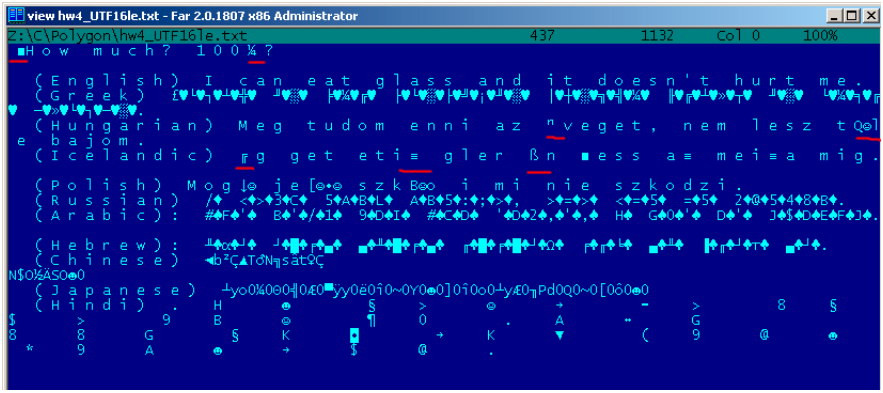


Figure 36.6: FAR: UTF-16LE

[3] https://en.wikipedia.org/wiki/Cyrillic_(Unicode_block)

Here we can also see BOM in the very beginning. All Latin characters are interleaved with zero byte. I also underscored by red some characters with diacritic marks (Hungarian and Icelandic languages).

36.2 Error/debug messages

Debugging messages are often very helpful if present. In some sense, debugging messages are reporting about what's going on in program right now. Often these are `printf()`-like functions, which writes to log-files, and sometimes, not writing anything but calls are still present since this build is not a debug build but *release* one. If local or global variables are dumped in debugging messages, it might be helpful as well since it is possible to get variable names at least. For example, one of such functions in Oracle RDBMS is `ksdwrt()`.

Meaningful text strings are often helpful. IDA disassembler may show from which function and from which point this specific string is used. Funny cases sometimes happen[4].

Error messages may help us as well. In Oracle RDBMS, errors are reporting using group of functions. More about it[5].

It is possible to find very quickly, which functions reporting about errors and in which conditions. By the way, it is often a reason why copy-protection systems has inarticulate cryptic error messages or just error numbers. No one happy when software cracker quickly understand why copy-protection is triggered just by error message.

One example of encrypted error messages is here: 55.2.

[4] http://blog.yurichev.com/node/32
[5] http://blog.yurichev.com/node/43

Chapter 37

Calls to assert()

Sometimes `assert()` macro presence is useful too: commonly this macro leaves source file name, line number and condition in code.

Most useful information is contained in assert-condition, we can deduce variable names, or structure field names from it. Another useful piece of information is file names —we can try to deduce what type of code is here. Also by file names it is possible to recognize a well-known open-source libraries.

Listing 37.1: Example of informative assert() calls

```
.text:107D4B29 mov  dx, [ecx+42h]
.text:107D4B2D cmp  edx, 1
.text:107D4B30 jz   short loc_107D4B4A
.text:107D4B32 push 1ECh
.text:107D4B37 push offset aWrite_c ; "write.c"
.text:107D4B3C push offset aTdTd_planarcon ; "td->↙
    ↳ td_planarconfig == PLANARCONFIG_CON"...
.text:107D4B41 call ds:_assert

...

.text:107D52CA mov  edx, [ebp-4]
.text:107D52CD and  edx, 3
.text:107D52D0 test edx, edx
.text:107D52D2 jz   short loc_107D52E9
.text:107D52D4 push 58h
.text:107D52D6 push offset aDumpmode_c ; "dumpmode.c"
.text:107D52DB push offset aN30     ; "(n & 3) == 0"
.text:107D52E0 call ds:_assert

...
```

```
.text:107D6759 mov  cx, [eax+6]
.text:107D675D cmp  ecx, 0Ch
.text:107D6760 jle  short loc_107D677A
.text:107D6762 push 2D8h
.text:107D6767 push offset aLzw_c   ; "lzw.c"
.text:107D676C push offset aSpLzw_nbitsBit ; "sp->lzw_nbits <= ↙
    ↳ BITS_MAX"
.text:107D6771 call ds:_assert
```

It is advisable to “google” both conditions and file names, that may lead us to open-source library. For example, if to “google” “sp->lzw_nbits <= BITS_MAX”, this predictably give us some open-source code, something related to LZW-compression.

Chapter 38

Constants

Humans, including programmers, often use round numbers like 10, 100, 1000, as well as in the code.

Practicing reverse engineer, usually know them well in hexadecimal representation: 10=0xA, 100=0x64, 1000=0x3E8, 10000=0x2710.

Some algorithms, especially cryptographical, use distinct constants, which is easy to find in code using IDA.

For example, MD5[1] algorithm initializes its own internal variables like:

```
var int h0 := 0x67452301
var int h1 := 0xEFCDAB89
var int h2 := 0x98BADCFE
var int h3 := 0x10325476
```

If you find these four constants usage in the code in a row —it is very high probability this function is related to MD5.

Another example is CRC16/CRC32 algorithms, often, calculation algorithms use precomputed tables like:

Listing 38.1: linux/lib/crc16.c

```
/** CRC table for the CRC-16. The poly is 0x8005 (x^16 + x^15 + ↙
    ↳ x^2 + 1) */
u16 const crc16_table[256] = {
        0x0000, 0xC0C1, 0xC181, 0x0140, 0xC301, 0x03C0, 0x0280, ↙
    ↳ 0xC241,
        0xC601, 0x06C0, 0x0780, 0xC741, 0x0500, 0xC5C1, 0xC481, ↙
    ↳ 0x0440,
```

[1] http://en.wikipedia.org/wiki/MD5

```
        0xCC01, 0x0CC0, 0x0D80, 0xCD41, 0x0F00, 0xCFC1, 0xCE81, ↙
    ↳ 0x0E40,
        ...
```

See also precomputed table for CRC32: 17.4.

38.1 Magic numbers

A lot of file formats defining a standard file header where *magic number*[2] is used.

For example, all Win32 and MS-DOS executables are started with two characters "MZ"[3].

At the MIDI-file beginning "MThd" signature must be present. If we have a program which uses MIDI-files for something, very likely, it must check MIDI-files for validity by checking at least first 4 bytes.

This could be done like:

(*buf* pointing to the beginning of loaded file into memory)

```
cmp [buf], 0x6468544D ; "MThd"
jnz _error_not_a_MIDI_file
```

...or by calling function for comparing memory blocks `memcmp()` or any other equivalent code up to a `CMPSB` (B.6.3) instruction.

When you find such point you already may say where MIDI-file loading is starting, also, we could see a location of MIDI-file contents buffer and what is used from the buffer, and how.

38.1.1 DHCP

This applies to network protocols as well. For example, DHCP protocol network packets contains so-called *magic cookie*: `0x63538263`. Any code generating DHCP protocol packets somewhere and somehow must embed this constant into packet. If we find it in the code we may find where it happen and not only this. *Something* received DHCP packet must check *magic cookie*, comparing it with the constant.

For example, let's take dhcpcore.dll file from Windows 7 x64 and search for the constant. And we found it, two times: it seems, the constant is used in two functions eloquently named as `DhcpExtractOptionsForValidation()` and `DhcpExtractFullOptio`

Listing 38.2: dhcpcore.dll (Windows 7 x64)

```
.rdata:000007FF6483CBE8 dword_7FF6483CBE8 dd 63538263h          ↙
    ↳ ; DATA XREF: DhcpExtractOptionsForValidation+79
.rdata:000007FF6483CBEC dword_7FF6483CBEC dd 63538263h          ↙
    ↳ ; DATA XREF: DhcpExtractFullOptions+97
```

[2] http://en.wikipedia.org/wiki/Magic_number_(programming)

[3] http://en.wikipedia.org/wiki/DOS_MZ_executable

And the places where these constants accessed:

Listing 38.3: dhcpcore.dll (Windows 7 x64)

```
.text:000007FF6480875F  mov     eax, [rsi]
.text:000007FF64808761  cmp     eax, cs:dword_7FF6483CBE8
.text:000007FF64808767  jnz     loc_7FF64817179
```

And:

Listing 38.4: dhcpcore.dll (Windows 7 x64)

```
.text:000007FF648082C7  mov     eax, [r12]
.text:000007FF648082CB  cmp     eax, cs:dword_7FF6483CBEC
.text:000007FF648082D1  jnz     loc_7FF648173AF
```

38.2 Constant searching

It is easy in IDA: Alt-B or Alt-I. And for searching for constant in big pile of files, or for searching it in non-executable files, I wrote small utility *binary grep*[4].

[4] https://github.com/yurichev/bgrep

Chapter 39

Finding the right instructions

If the program is utilizing FPU instructions and there are very few of them in a code, one can try to check each one manually by debugger.

For example, we may be interesting, how Microsoft Excel calculating formulae entered by user. For example, division operation.

If to load excel.exe (from Office 2010) version 14.0.4756.1000 into IDA, then make a full listing and to find each `FDIV` instructions (except ones which use constants as a second operand —obviously, it is not suits us):

```
cat EXCEL.lst | grep fdiv | grep -v dbl_ > EXCEL.fdiv
```

...then we realizing they are just 144.

We can enter string like `=(1/3)` in Excel and check each instruction.

Checking each instruction in debugger or tracer (one may check 4 instruction at a time), it seems, we are lucky here and sought-for instruction is just 14th:

```
.text:3011E919 DC 33                                    fdiv    ↙
    ↳ qword ptr [ebx]
```

```
PID=13944|TID=28744|(0) 0x2f64e919 (Excel.exe!BASE+0x11e919)
EAX=0x02088006 EBX=0x02088018 ECX=0x00000001 EDX=0x00000001
ESI=0x02088000 EDI=0x00544804 EBP=0x0274FA3C ESP=0x0274F9F8
EIP=0x2F64E919
FLAGS=PF IF
FPU ControlWord=IC RC=NEAR PC=64bits PM UM OM ZM DM IM
FPU StatusWord=
```

```
FPU ST(0): 1.000000
```

`ST(0)` holding first argument (1) and second one is in `[EBX]`.

Next instruction after `FDIV` writes result into memory:

```
.text:3011E91B DD 1E                                    fstp    ↙
    ↳ qword ptr [esi]
```

If to set breakpoint on it, we may see result:

```
PID=32852|TID=36488|(0) 0x2f40e91b (Excel.exe!BASE+0x11e91b)
EAX=0x00598006 EBX=0x00598018 ECX=0x00000001 EDX=0x00000001
ESI=0x00598000 EDI=0x00294804 EBP=0x026CF93C ESP=0x026CF8F8
EIP=0x2F40E91B
FLAGS=PF IF
FPU ControlWord=IC RC=NEAR PC=64bits PM UM OM ZM DM IM
FPU StatusWord=C1 P
FPU ST(0): 0.333333
```

Also as a practical joke, we can modify it on-fly:

```
tracer -l:excel.exe bpx=excel.exe!BASE+0x11E91B,set(st0,666)
```

```
PID=36540|TID=24056|(0) 0x2f40e91b (Excel.exe!BASE+0x11e91b)
EAX=0x00680006 EBX=0x00680018 ECX=0x00000001 EDX=0x00000001
ESI=0x00680000 EDI=0x00395404 EBP=0x0290FD9C ESP=0x0290FD58
EIP=0x2F40E91B
FLAGS=PF IF
FPU ControlWord=IC RC=NEAR PC=64bits PM UM OM ZM DM IM
FPU StatusWord=C1 P
FPU ST(0): 0.333333
Set ST0 register to 666.000000
```

Excel showing 666 in the cell what finally convincing us we find the right point.

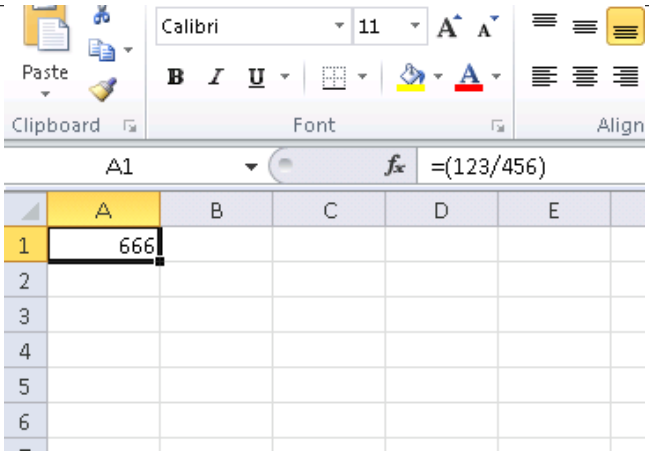


Figure 39.1: Practical joke worked

If to try the same Excel version, but x64, we will find only 12 `FDIV` instructions there, and the one we looking for —third.

```
tracer.exe -l:excel.exe bpx=excel.exe!BASE+0x1B7FCC,set(st0,666)
```

It seems, a lot of division operations of *float* and *double* types, compiler replaced by SSE-instructions like `DIVSD` (`DIVSD` present here 268 in total).

Chapter 40

Suspicious code patterns

40.1 XOR instructions

instructions like `XOR op, op` (for example, `XOR EAX, EAX`) are usually used for setting register value to zero, but if operands are different, *exclusive or* operation is executed. This operation is rare in common programming, but used often in cryptography, including amateur. Especially suspicious case if the second operand is big number. This may points to encrypting/decrypting, checksum computing, etc.

One exception to this observation worth to note is "canary" (16.3) generation and checking is often done using `XOR` instruction.

This AWK script can be used for processing IDA listing (.lst) files:

```
gawk -e '$2=="xor" { tmp=substr($3, 0, length($3)-1); if (tmp!=↙
    ↳ $4) if($4!="esp") if ($4!="ebp") { print $1, $2, tmp, ",",↙
    ↳  $4 } }' filename.lst
```

It is also worth to note that such script may also capture incorrectly disassembled code (28).

40.2 Hand-written assembly code

Modern compilers do not emit `LOOP` and `RCL` instructions. On the other hand, these instructions are well-known to coders who like to code in straight assembly language. If you spot these, it can be said, with a high probability, this fragment of code is hand-written. Such instructions are marked as (M) in the instructions list in appendix: B.6.

Also function prologue/epilogue is not commonly present in hand-written assembly

copy.

Commonly there is no fixed system in passing arguments into functions in the hand-written code.

Example from Windows 2003 kernel (ntoskrnl.exe file):

```
MultiplyTest proc near               ; CODE XREF: Get386Stepping
             xor     cx, cx
loc_620555:                          ; CODE XREF: MultiplyTest+E
             push    cx
             call    Multiply
             pop     cx
             jb      short locret_620563
             loop    loc_620555
             clc
locret_620563:                       ; CODE XREF: MultiplyTest+C
             retn
MultiplyTest endp

Multiply     proc near               ; CODE XREF: MultiplyTest+5
             mov     ecx, 81h
             mov     eax, 417A000h
             mul     ecx
             cmp     edx, 2
             stc
             jnz     short locret_62057F
             cmp     eax, 0FE7A000h
             stc
             jnz     short locret_62057F
             clc
locret_62057F:                       ; CODE XREF: Multiply+10
                                     ; Multiply+18
             retn
Multiply     endp
```

Indeed, if we look into WRK[1] v1.2 source code, this code can be found easily in the file *WRK-v1.2\base\ntos\ke\i386\cpu.asm*.

[1]Windows Research Kernel

Chapter 41

Using magic numbers while tracing

Often, main goal is to get to know, how a value was read from file, or received via network, being used. Often, manual tracing of a value is very labouring task. One of the simplest techniques (although not 100% reliable) is to use your own *magic number*.

This resembling X-ray computed tomography is some sense: radiocontrast agent is often injected into patient's blood, which is used for improving visibility of internal structures in X-rays. For example, it is well known how blood of healthy man/woman percolates in kidneys and if agent is in blood, it will be easily seen on tomography, how good and normal blood was percolating, are there any stones or tumors.

We can take a 32-bit number like `0x0badf00d`, or someone's birth date like `0x11101979` and to write this, 4 byte holding number, to some point in file used by the program we investigate.

Then, while tracing this program, with tracer in the *code coverage* mode, and then, with the help of *grep* or just by searching in the text file (of tracing results), we can easily see, where the value was used and how.

Example of *grepable* tracer results in the *cc* mode:

```
0x150bf66 (_kziaia+0x14), e=       1 [MOV EBX, [EBP+8]] [EBP↙
    ↳ +8]=0xf59c934
0x150bf69 (_kziaia+0x17), e=       1 [MOV EDX, [69AEB08h]] [69↙
    ↳ AEB08h]=0
0x150bf6f (_kziaia+0x1d), e=       1 [FS: MOV EAX, [2Ch]]
0x150bf75 (_kziaia+0x23), e=       1 [MOV ECX, [EAX+EDX*4]] [EAX↙
    ↳ +EDX*4]=0xf1ac360
0x150bf78 (_kziaia+0x26), e=       1 [MOV [EBP-4], ECX] ECX=0↙
    ↳ xf1ac360
```

This can be used for network packets as well. It is important to be unique for *magic number* and not to be present in the program's code.

Aside of tracer, DosBox (MS-DOS emulator) in heavydebug mode, is able to write information about all register's states for each executed instruction of program to plain text file[1], so this technique may be useful for DOS programs as well.

[1] See also my blog post about this DosBox feature: http://blog.yurichev.com/node/55

Chapter 42

Other things

RTTI (29.1.5)-data may be also useful for C++ classes identification.

Chapter 43

Old-school techniques, nevertheless, interesting to know

43.1 Memory "snapshots" comparing

The technique of straightforward two memory snapshots comparing in order to see changes, was often used to hack 8-bit computer games and hacking "high score" files.

For example, if you got a loaded game on 8-bit computer (it is not much memory on these, but game is usually consumes even less memory) and you know that you have now, let's say, 100 bullets, you can do a "snapshot" of all memory and back it up to some place. Then shoot somewhere, bullet count now 99, do second "snapshot" and then compare both: somewhere must be a byte which was 100 in the beginning and now it is 99. Considering a fact these 8-bit games were often written in assembly language and such variables were global, it can be said for sure, which address in memory holding bullets count. If to search all references to the address in disassembled game code, it is not very hard to find a piece of code decrementing bullets count, write NOP instruction there, or couple of NOP-s, we'll have a game with e.g 100 bullets forever. Games on these 8-bit computers was commonly loaded on the same address, also, there were no much different versions of each game (commonly just one version was popular for a long span of time), enthusiastic gamers knew, which byte must be written (using BASIC instruction POKE) to which address in order to hack it. This led to "cheat" lists containing of POKE instructions published in magazines related to 8-bit games. See also: http://en.wikipedia.org/wiki/PEEK_and_POKE.

Likewise, it is easy to modify "high score" files, this may work not only with 8-bit games. Let's notice your score count and back the file up somewhere. When "high score" count will be different, just compare two files, it can be even done with DOS-

utility FC[1] ("high score" files are often in binary form). There will be a point where couple of bytes will be different and it will be easy to see which ones are holding score number. However, game developers are aware of such tricks and may protect against it.

43.1.1 Windows registry

It is also possible to compare Windows registry before and after a program installation. It is very popular method of finding, which registry elements a program will use.

[1] MS-DOS utility for binary files comparing

Part IV

OS-specific

Chapter 44

Arguments passing methods (calling conventions)

44.1 cdecl

This is the most popular method for arguments passing to functions in C/C++ languages.

Caller pushing arguments to stack in reverse order: last argument, then penultimate element and finally —first argument. Caller also must return back value of the stack pointer (ESP) to its initial state after callee function exit.

Listing 44.1: cdecl

```
push arg3
push arg2
push arg1
call function
add esp, 12 ; returns ESP
```

44.2 stdcall

Almost the same thing as *cdecl*, with the exception the callee set ESP to initial state executing RET x instruction instead of RET, where `x = arguments number * sizeof(int)`[1]. Caller will not adjust stack pointer by `add esp, x` instruction.

Listing 44.2: stdcall

```
push arg3
```

[1] Size of *int* type variable is 4 in x86 systems and 8 in x64 systems

```
push arg2
push arg1
call function

function:
... do something ...
ret 12
```

The method is ubiquitous in win32 standard libraries, but not in win64 (see below about win64).

For example, we may take the function from 7.1 and change it slightly by adding `__stdcall` modifier:

```
int __stdcall f2 (int a, int b, int c)
{
        return a*b+c;
};
```

It will be compiled in almost the same way as 7.2, but you will see `RET 12` instead of `RET`. SP is not aligned in caller.

As a consequence, number of function arguments can be easily deduced from `RETN n` instruction: just divide n by 4.

Listing 44.3: MSVC 2010

```
_a$ = 8                                                 ; size =↙
    ↘  4
_b$ = 12                                                ; size =↙
    ↘  4
_c$ = 16                                                ; size =↙
    ↘  4
_f2@12  PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _a$[ebp]
        imul    eax, DWORD PTR _b$[ebp]
        add     eax, DWORD PTR _c$[ebp]
        pop     ebp
        ret     12                                      ; ↙
    ↘ 0000000cH
_f2@12  ENDP

; ...
        push    3
        push    2
```

```
        push    1
        call    _f2@12
        push    eax
        push    OFFSET $SG81369
        call    _printf
        add     esp, 8
```

44.2.1 Variable arguments number functions

`printf()`-like functions are, probably, the only case of variable arguments functions in C/C++, but it is easy to illustrate an important difference between *cdecl* and *stdcall* with the help of it. Let's start with the idea the compiler knows argument count of each `printf()` function calling. However, called `printf()`, which is already compiled and located in MSVCRT.DLL (if to talk about Windows), has not any information about how much arguments were passed, however it can determine it from format string. Thus, if `printf()` would be *stdcall*-function and restored stack pointer to its initial state by counting number of arguments in format string, this could be dangerous situation, when one programmer's typo may provoke sudden program crash. Thus it is not suitable for such functions to use *stdcall*, *cdecl* is better.

44.3 fastcall

That's general naming for a method of passing some of arguments via registers and all other —via stack. It worked faster than *cdecl*/*stdcall* on older CPUs (because of smaller stack pressure). It will not help to gain performance on modern much complex CPUs, however.

it is not a standardized way, so, various compilers may do it differently. Well known caveat: if you have two DLLs, one uses another, and they are built by different compilers with different *fastcall* calling conventions.

Both MSVC and GCC passing first and second argument via `ECX` and `EDX` and other arguments via stack.

Stack pointer must be restored to initial state by callee (like in *stdcall*).

Listing 44.4: fastcall

```
push arg3
mov edx, arg2
mov ecx, arg1
call function

function:
.. do something ..
ret 4
```

For example, we may take the function from 7.1 and change it slightly by adding `__fastcall` modifier:

```
int __fastcall f3 (int a, int b, int c)
{
        return a*b+c;
};
```

Here is how it will be compiled:

Listing 44.5: MSVC 2010 /Ox /Ob0

```
_c$ = 8                                                 ; size =↙
    ↘  4
@f3@12  PROC
; _a$ = ecx
; _b$ = edx
        mov     eax, ecx
        imul    eax, edx
        add     eax, DWORD PTR _c$[esp-4]
        ret     4
@f3@12  ENDP

; ...

        mov     edx, 2
        push    3
        lea     ecx, DWORD PTR [edx-1]
        call    @f3@12
        push    eax
        push    OFFSET $SG81390
        call    _printf
        add     esp, 8
```

We see that a callee returns SP by `RETN` instruction with operand. Which means, number of arguments can be deduced easily here as well.

44.3.1 GCC regparm

It is *fastcall* evolution[2] is some sense. With the `-mregparm` option it is possible to set, how many arguments will be passed via registers. 3 at maximum. Thus, `EAX`, `EDX` and `ECX` registers are to be used.

Of course, if number of arguments is less then 3, not all 3 registers are to be used.

Caller restores stack pointer to its initial state.

For the example, see (17.1.1).

[2] http://www.ohse.de/uwe/articles/gcc-attributes.html#func-regparm

44.3.2 Watcom/OpenWatcom

It is called "register calling convention" here. First 4 arguments are passed via EAX, EDX, EBX and ECX registers. All the rest—via stack. Functions have underscore added to the function name in order to distinguish them from those having other calling convention.

44.4 thiscall

In C++, it is a *this* pointer to object passing into function-method.

In MSVC, *this* is usually passed in the ECX register.

In GCC, *this* pointer is passed as a first function-method argument. Thus it will be seen: internally, all function-methods has extra argument.

For the example, see (29.1.1).

44.5 x86-64

44.5.1 Windows x64

The method of arguments passing in Win64 is somewhat resembling to fastcall. First 4 arguments are passed via RCX, RDX, R8, R9, other —via stack. Caller also must prepare a space for 32 bytes or 4 64-bit values, so then callee can save there first 4 arguments. Short functions may use argument values just from registers, but larger may save its values for further use.

Caller also must return stack pointer into initial state.

This calling convention is also used in Windows x86-64 system DLLs (instead of *stdcall* in win32).

Example:

```
#include <stdio.h>

void f1(int a, int b, int c, int d, int e, int f, int g)
{
        printf ("%d %d %d %d %d %d %d\n", a, b, c, d, e, f, g);
};

int main()
{
        f1(1,2,3,4,5,6,7);
};
```

Listing 44.6: MSVC 2012 /0b

```
$SG2937 DB      '%d %d %d %d %d %d %d', 0aH, 00H

main    PROC
        sub     rsp, 72                                 ; ↙
    ↳ 00000048H

        mov     DWORD PTR [rsp+48], 7
        mov     DWORD PTR [rsp+40], 6
        mov     DWORD PTR [rsp+32], 5
        mov     r9d, 4
        mov     r8d, 3
        mov     edx, 2
        mov     ecx, 1
        call    f1

        xor     eax, eax
        add     rsp, 72                                 ; ↙
    ↳ 00000048H
        ret     0
main    ENDP

a$ = 80
b$ = 88
c$ = 96
d$ = 104
e$ = 112
f$ = 120
g$ = 128
f1      PROC
$LN3:
        mov     DWORD PTR [rsp+32], r9d
        mov     DWORD PTR [rsp+24], r8d
        mov     DWORD PTR [rsp+16], edx
        mov     DWORD PTR [rsp+8], ecx
        sub     rsp, 72                                 ; ↙
    ↳ 00000048H

        mov     eax, DWORD PTR g$[rsp]
        mov     DWORD PTR [rsp+56], eax
        mov     eax, DWORD PTR f$[rsp]
        mov     DWORD PTR [rsp+48], eax
        mov     eax, DWORD PTR e$[rsp]
        mov     DWORD PTR [rsp+40], eax
```

```
        mov     eax, DWORD PTR d$[rsp]
        mov     DWORD PTR [rsp+32], eax
        mov     r9d, DWORD PTR c$[rsp]
        mov     r8d, DWORD PTR b$[rsp]
        mov     edx, DWORD PTR a$[rsp]
        lea     rcx, OFFSET FLAT:$SG2937
        call    printf

        add     rsp, 72                                 ; ↙
    ↳ 00000048H
        ret     0
f1      ENDP
```

Here we clearly see how 7 arguments are passed: 4 via registers and the rest 3 via stack. The code of f1() function's prologue saves the arguments in "scratch space"—a space in the stack intended exactly for the purpose. It is done because compiler may not be sure if it will be enough to use other registers without these 4, which will otherwise be occupied by arguments until function execution end. The "scratch space" allocation in the stack is on the caller's shoulders.

Listing 44.7: MSVC 2012 /Ox /0b

```
$SG2777 DB      '%d %d %d %d %d %d %d', 0aH, 00H

a$ = 80
b$ = 88
c$ = 96
d$ = 104
e$ = 112
f$ = 120
g$ = 128
f1      PROC
$LN3:
        sub     rsp, 72                                 ; ↙
    ↳ 00000048H

        mov     eax, DWORD PTR g$[rsp]
        mov     DWORD PTR [rsp+56], eax
        mov     eax, DWORD PTR f$[rsp]
        mov     DWORD PTR [rsp+48], eax
        mov     eax, DWORD PTR e$[rsp]
        mov     DWORD PTR [rsp+40], eax
        mov     DWORD PTR [rsp+32], r9d
        mov     r9d, r8d
        mov     r8d, edx
```

```
        mov     edx, ecx
        lea     rcx, OFFSET FLAT:$SG2777
        call    printf

        add     rsp, 72                                 ; ↙
    ↳ 00000048H
        ret     0
f1      ENDP

main    PROC
        sub     rsp, 72                                 ; ↙
    ↳ 00000048H

        mov     edx, 2
        mov     DWORD PTR [rsp+48], 7
        mov     DWORD PTR [rsp+40], 6
        lea     r9d, QWORD PTR [rdx+2]
        lea     r8d, QWORD PTR [rdx+1]
        lea     ecx, QWORD PTR [rdx-1]
        mov     DWORD PTR [rsp+32], 5
        call    f1

        xor     eax, eax
        add     rsp, 72                                 ; ↙
    ↳ 00000048H
        ret     0
main    ENDP
```

If to compile the example with optimization switch, it is almost the same, but "scratch space" is not used, because no need to.

Also take a look on how MSVC 2012 optimizes primitive value loads into registers by using `LEA` (B.6.2). I'm not sure if it worth so, but maybe.

this passing

this pointer is passed in `RCX`, first method argument in `RDX`, etc. See also for an example: 29.1.1.

44.5.2 Linux x64

The way arguments passed in Linux for x86-64 is almost the same as in Windows, but 6 registers are used instead of 4 (`RDI`, `RSI`, `RDX`, `RCX`, `R8`, `R9`) and there are no "scratch space", but callee may save register values in the stack, if it needs to.

Listing 44.8: GCC 4.7.3 -O3

```
.LC0:
        .string "%d %d %d %d %d %d %d\n"
f1:
        sub     rsp, 40
        mov     eax, DWORD PTR [rsp+48]
        mov     DWORD PTR [rsp+8], r9d
        mov     r9d, ecx
        mov     DWORD PTR [rsp], r8d
        mov     ecx, esi
        mov     r8d, edx
        mov     esi, OFFSET FLAT:.LC0
        mov     edx, edi
        mov     edi, 1
        mov     DWORD PTR [rsp+16], eax
        xor     eax, eax
        call    __printf_chk
        add     rsp, 40
        ret
main:
        sub     rsp, 24
        mov     r9d, 6
        mov     r8d, 5
        mov     DWORD PTR [rsp], 7
        mov     ecx, 4
        mov     edx, 3
        mov     esi, 2
        mov     edi, 1
        call    f1
        add     rsp, 24
        ret
```

N.B.: here values are written into 32-bit parts of registers (e.g., EAX) but not to the whole 64-bit register (RAX). This is because each write to low 32-bit part of register automatically clears high 32 bits. Supposedly, it was done for x86-64 code porting simplification.

44.6 Returning values of *float* and *double* type

In all conventions except of Win64, values of type *float* or *double* are returning via the FPU register `ST(0)`.

In Win64, values of *float* and *double* types are returned in the `XMM0` register instead of the `ST(0)`.

44.7 Modifying arguments

Sometimes, C/C++ programmers (not limited to these PL, though), may ask, what will happen if to modify arguments? The answer is simple: arguments are stored in the stack, that is where modification will occurr. Calling functions are not use them after callee exit (I have not seen any opposite case in my practice).

```
#include <stdio.h>

void f(int a, int b)
{
        a=a+b;
        printf ("%d\n", a);
};
```

Listing 44.9: MSVC 2012

```
_a$ = 8                                                 ; size =↙
    ↳ 4
_b$ = 12                                                ; size =↙
    ↳ 4
_f      PROC
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR _a$[ebp]
        add     eax, DWORD PTR _b$[ebp]
        mov     DWORD PTR _a$[ebp], eax
        mov     ecx, DWORD PTR _a$[ebp]
        push    ecx
        push    OFFSET $SG2938 ; '%d', 0aH
        call    _printf
        add     esp, 8
        pop     ebp
        ret     0
_f      ENDP
```

So yes, one may modify arguments easily. Of course, if it is not *references* in C++ (29.3), and if you not modify data a pointer pointing to (then the effect will be propagated outside of current function).

Chapter 45

Thread Local Storage

It is a data area, specific to each thread. Every thread can store there what it needs. One famous example is C standard global variable *errno*. Multiple threads may simultaneously call a functions which returns error code in the *errno*, so global variable will not work correctly here, for multi-thread programs, *errno* must be stored in the TLS.

In the C++11 standard, a new *thread_local* modifier was added, showing that each thread will have its own version of the variable, it can be initialized, and it is located in the TLS [1]:

Listing 45.1: C++11

```
#include <iostream>
#include <thread>

thread_local int tmp=3;

int main()
{
        std::cout << tmp << std::endl;
};
```

[2]

If to say about PE-files, in the resulting executable file, the *tmp* variable will be stored in the section devoted to TLS.

[1] C11 also has thread support, optional though

[2] Compiled in MinGW GCC 4.8.1, but not in MSVC 2012

Chapter 46

System calls (syscall-s)

As we know, all running processes inside OS are divided into two categories: those having all access to the hardware ("kernel space") and those have not ("user space").

There are OS kernel and usually drivers in the first category.

All applications are usually in the second category.

This separation is crucial for OS safety: it is very important not to give to any process possibility to screw up something in other processes or even in OS kernel. On the other hand, failing driver or error inside OS kernel usually lead to kernel panic or BSOD[1].

x86-processor protection allows to separate everything into 4 levels of protection (rings), but both in Linux and in Windows only two are used: ring0 ("kernel space") and ring3 ("user space").

System calls (syscall-s) is a point where these two areas are connected. It can be said, this is the most principal API providing to application software.

As in Windows NT, syscalls table reside in SSDT[2].

Usage of syscalls is very popular among shellcode and computer viruses authors, because it is hard to determine the addresses of needed functions in the system libraries, while it is easier to use syscalls, however, much more code should be written due to lower level of abstraction of the API. It is also worth noting that the numbers of syscalls e.g. in Windows, may be different from version to version.

46.1 Linux

In Linux, syscall is usually called via `int 0x80`. Call number is passed in the `EAX` register, and any other parameters —in the other registers.

[1] Black Screen of Death

[2] System Service Dispatch Table

Listing 46.1: Simple example of two syscalls usage

```
section .text
global  _start

_start:
        mov     edx,len ; buf len
        mov     ecx,msg ; buf
        mov     ebx,1   ; file descriptor. stdout is 1
        mov     eax,4   ; syscall number. sys_write is 4
        int     0x80

        mov     eax,1   ; syscall number. sys_exit is 4
        int     0x80

section .data

msg     db  'Hello, world!',0xa
len     equ $ - msg
```

Compilation:

```
nasm -f elf32 1.s
ld 1.o
```

The full list of syscalls in Linux: http://syscalls.kernelgrok.com/.
For system calls intercepting and tracing in Linux, strace(51) can be used.

46.2 Windows

They are called by `int 0x2e` or using special x86 instruction SYSENTER.
The full list of syscalls in Windows: http://j00ru.vexillium.org/ntapi/.
Further reading:
"Windows Syscall Shellcode" by Piotr Bania:
http://www.symantec.com/connect/articles/windows-syscall-shellcode.

Chapter 47

Linux

47.1 Position-independent code

While analyzing Linux shared (.so) libraries, one may frequently spot such code pattern:

Listing 47.1: libc-2.17.so x86

```
.text:0012D5E3 __x86_get_pc_thunk_bx proc near         ; CODE ↙
    ↳ XREF: sub_17350+3
.text:0012D5E3                                         ; ↙
    ↳ sub_173CC+4 ...
.text:0012D5E3                 mov     ebx, [esp+0]
.text:0012D5E6                 retn
.text:0012D5E6 __x86_get_pc_thunk_bx endp

...

.text:000576C0 sub_576C0       proc near               ; CODE ↙
    ↳ XREF: tmpfile+73

...

.text:000576C0                 push    ebp
.text:000576C1                 mov     ecx, large gs:0
.text:000576C8                 push    edi
.text:000576C9                 push    esi
.text:000576CA                 push    ebx
.text:000576CB                 call    __x86_get_pc_thunk_bx
.text:000576D0                 add     ebx, 157930h
.text:000576D6                 sub     esp, 9Ch
```

```
...

.text:000579F0                  lea     eax, (a__gen_tempname - 1↙
    ↳ AF000h)[ebx] ; "__gen_tempname"
.text:000579F6                  mov     [esp+0ACh+var_A0], eax
.text:000579FA                  lea     eax, (a__SysdepsPosix - 1↙
    ↳ AF000h)[ebx] ; "../sysdeps/posix/tempname.c"
.text:00057A00                  mov     [esp+0ACh+var_A8], eax
.text:00057A04                  lea     eax, (aInvalidKindIn_ - 1↙
    ↳ AF000h)[ebx] ; "! \"invalid KIND in __gen_tempname\""
.text:00057A0A                  mov     [esp+0ACh+var_A4], 14Ah
.text:00057A12                  mov     [esp+0ACh+var_AC], eax
.text:00057A15                  call    __assert_fail
```

All pointers to strings are corrected by a constant and by value in the EBX, which calculated at the beginning of each function. This is so-called PIC, it is intended to execute placed at any random point of memory, that is why it cannot contain any absolute memory addresses.

PIC was crucial in early computer systems and crucial now in embedded systems without virtual memory support (where processes are all placed in single continous memory block). It is also still used in *NIX systems for shared libraries since shared libraries are shared across many processes while loaded in memory only once. But all these processes may map the same shared library on different addresses, so that is why shared library should be working correctly without fixing on any absolute address.

Let's do a simple experiment:

```
#include <stdio.h>

int global_variable=123;

int f1(int var)
{
    int rt=global_variable+var;
    printf ("returning %d\n", rt);
    return rt;
};
```

Let's compile it in GCC 4.7.3 and see resulting .so file in IDA:

```
gcc -fPIC -shared -O3 -o 1.so 1.c
```

Listing 47.2: GCC 4.7.3

```
.text:00000440                  public __x86_get_pc_thunk_bx
.text:00000440 __x86_get_pc_thunk_bx proc near          ; CODE ↙
    ↳ XREF: _init_proc+4
.text:00000440                                          ; ↙
    ↳ deregister_tm_clones+4 ...
.text:00000440                  mov     ebx, [esp+0]
.text:00000443                  retn
.text:00000443 __x86_get_pc_thunk_bx endp

.text:00000570                  public f1
.text:00000570 f1               proc near
.text:00000570
.text:00000570 var_1C           = dword ptr -1Ch
.text:00000570 var_18           = dword ptr -18h
.text:00000570 var_14           = dword ptr -14h
.text:00000570 var_8            = dword ptr -8
.text:00000570 var_4            = dword ptr -4
.text:00000570 arg_0            = dword ptr  4
.text:00000570
.text:00000570                  sub     esp, 1Ch
.text:00000573                  mov     [esp+1Ch+var_8], ebx
.text:00000577                  call    __x86_get_pc_thunk_bx
.text:0000057C                  add     ebx, 1A84h
.text:00000582                  mov     [esp+1Ch+var_4], esi
.text:00000586                  mov     eax, ds:(↙
    ↳ global_variable_ptr - 2000h)[ebx]
.text:0000058C                  mov     esi, [eax]
.text:0000058E                  lea     eax, (aReturningD - 2000h↙
    ↳ )[ebx] ; "returning %d\n"
.text:00000594                  add     esi, [esp+1Ch+arg_0]
.text:00000598                  mov     [esp+1Ch+var_18], eax
.text:0000059C                  mov     [esp+1Ch+var_1C], 1
.text:000005A3                  mov     [esp+1Ch+var_14], esi
.text:000005A7                  call    ___printf_chk
.text:000005AC                  mov     eax, esi
.text:000005AE                  mov     ebx, [esp+1Ch+var_8]
.text:000005B2                  mov     esi, [esp+1Ch+var_4]
.text:000005B6                  add     esp, 1Ch
.text:000005B9                  retn
.text:000005B9 f1               endp
```

That's it: pointers to *«returning %d\n»* string and *global_variable* are to be corrected at each function execution. The `__x86_get_pc_thunk_bx()` function return address of the point after call to itself (`0x57C` here) in the `EBX`. That's the simple way

to get value of program counter (EIP) at some point. The 0x1A84 constant is related to the difference between this function begin and so-called *Global Offset Table Procedure Linkage Table* (GOT PLT), the section right after *Global Offset Table* (GOT), where pointer to *global_variable* is. IDA shows these offset processed, so to understand them easily, but in fact the code is:

```
.text:00000577                call    __x86_get_pc_thunk_bx
.text:0000057C                add     ebx, 1A84h
.text:00000582                mov     [esp+1Ch+var_4], esi
.text:00000586                mov     eax, [ebx-0Ch]
.text:0000058C                mov     esi, [eax]
.text:0000058E                lea     eax, [ebx-1A30h]
```

So, EBX pointing to the GOT PLT section and to calculate pointer to *global_variable* which stored in the GOT, 0xC must be subtracted. To calculate pointer to the *«returning %d\n»* string, 0x1A30 must be subtracted.

By the way, that is the reason why AMD64 instruction set supports RIP[1]-relative addressing, just to simplify PIC-code.

Let's compile the same C code in the same GCC version, but for x64.

IDA would simplify output code but suppressing RIP-relative addressing details, so I will run *objdump* instead to see the details:

```
0000000000000720 <f1>:
 720:   48 8b 05 b9 08 20 00    mov    rax,QWORD PTR [rip+0↙
    ↳ x2008b9]        # 200fe0 <_DYNAMIC+0x1d0>
 727:   53                      push   rbx
 728:   89 fb                   mov    ebx,edi
 72a:   48 8d 35 20 00 00 00    lea    rsi,[rip+0x20]        # ↙
    ↳ 751 <_fini+0x9>
 731:   bf 01 00 00 00          mov    edi,0x1
 736:   03 18                   add    ebx,DWORD PTR [rax]
 738:   31 c0                   xor    eax,eax
 73a:   89 da                   mov    edx,ebx
 73c:   e8 df fe ff ff          call   620 <__printf_chk@plt>
 741:   89 d8                   mov    eax,ebx
 743:   5b                      pop    rbx
 744:   c3                      ret
```

0x2008b9 is the difference between address of instruction at 0x720 and *global_variable* and 0x20 is the difference between the address of the instruction at 0x72A and the *«returning %d\n»* string.

As you might see, the need to recalculate addresses frequently makes execution slower (it is better in x64, though). So it is probably better to link statically if you aware of performance ([11]).

[1] program counter in AMD64

47.1.1 Windows

The PIC mechanism is not used in Windows DLLs. If Windows loader needs to load DLL on another base address, it "patches" DLL in memory (at the *FIXUP* places) in order to correct all addresses. This means, several Windows processes cannot share once loaded DLL on different addresses in different process' memory blocks —since each loaded into memory DLL instance *fixed* to be work only at these addresses..

47.2 *LD_PRELOAD* hack in Linux

This allows us to load our own dynamic libraries before others, even before system ones, like libc.so.6.

What, in turn, allows to "substitute" our written functions before original ones in system libraries. For example, it is easy to intercept all calls to the time(), read(), write(), etc.

Let's see, if we are able to fool *uptime* utility. As we know, it tells how long the computer is working. With the help of strace(51), it is possible to see that this information the utility takes from the `/proc/uptime` file:

```
$ strace uptime
...
open("/proc/uptime", O_RDONLY)          = 3
lseek(3, 0, SEEK_SET)                   = 0
read(3, "416166.86 414629.38\n", 2047)  = 20
...
```

It is not a real file on disk, it is a virtual one, its contents is generated on fly in Linux kernel. There are just two numbers:

```
$ cat /proc/uptime
416690.91 415152.03
```

What we can learn from wikipedia:

> The first number is the total number of seconds the system has been up. The second number is how much of that time the machine has spent idle, in seconds.

[2]

[2] https://en.wikipedia.org/wiki/Uptime

Let's try to write our own dynamic library with the open(), read(), close() functions working as we need.

At first, our open() will compare name of file to be opened with what we need and if it is so, it will write down the descriptor of the file opened. At second, read(), if it will be called for this file descriptor, will substitute output, and in other cases, will call original read() from libc.so.6. And also close(), will note, if the file we are currently follow is to be closed.

We will use the dlopen() and dlsym() functions to determine original addresses of functions in libc.so.6.

We need them because we must pass control to "real" functions.

On the other hand, if we could intercept e.g. strcmp(), and follow each string comparisons in program, then strcmp() could be implemented easily on one's own, while not using original function [3].

```
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <stdbool.h>
#include <unistd.h>
#include <dlfcn.h>
#include <string.h>

void *libc_handle = NULL;
int (*open_ptr)(const char *, int) = NULL;
int (*close_ptr)(int) = NULL;
ssize_t (*read_ptr)(int, void*, size_t) = NULL;

bool inited = false;

_Noreturn void die (const char * fmt, ...)
{
        va_list va;
        va_start (va, fmt);

        vprintf (fmt, va);
        exit(0);
};

static void find_original_functions ()
{
        if (inited)
                return;
```

[3] For example, here is how simple strcmp() interception is works in article [4] from Yong Huang

```
        libc_handle = dlopen ("libc.so.6", RTLD_LAZY);
        if (libc_handle==NULL)
                die ("can't open libc.so.6\n");

        open_ptr = dlsym (libc_handle, "open");
        if (open_ptr==NULL)
                die ("can't find open()\n");

        close_ptr = dlsym (libc_handle, "close");
        if (close_ptr==NULL)
                die ("can't find close()\n");

        read_ptr = dlsym (libc_handle, "read");
        if (read_ptr==NULL)
                die ("can't find read()\n");

        inited = true;
}

static int opened_fd=0;

int open(const char *pathname, int flags)
{
        find_original_functions();

        int fd=(*open_ptr)(pathname, flags);
        if (strcmp(pathname, "/proc/uptime")==0)
                opened_fd=fd; // that's our file! record its ↙
    ↳ file descriptor
        else
                opened_fd=0;
        return fd;
};

int close(int fd)
{
        find_original_functions();

        if (fd==opened_fd)
                opened_fd=0; // the file is not opened anymore
        return (*close_ptr)(fd);
};
```

```
ssize_t read(int fd, void *buf, size_t count)
{
        find_original_functions();

        if (opened_fd!=0 && fd==opened_fd)
        {
                // that's our file!
                return snprintf (buf, count, "%d %d", 0x7fffffff↙
    ↳ , 0x7fffffff)+1;
        };
        // not our file, go to real read() function
        return (*read_ptr)(fd, buf, count);
};
```

Let's compile it as common dynamic library:

```
gcc -fpic -shared -Wall -o fool_uptime.so fool_uptime.c -ldl
```

Let's run *uptime* while loading our library before others:

```
LD_PRELOAD=‘pwd‘/fool_uptime.so uptime
```

And we see:

```
 01:23:02 up 24855 days,  3:14,  3 users,  load average: 0.00, ↙
    ↳ 0.01, 0.05
```

If the *LD_PRELOAD* environment variable will always points to filename and path of our library, it will be loaded for all starting programs.

More examples:

- Very simple interception of the strcmp() (Yong Huang) http://yurichev.com/mirrors/LD_PRELOAD/Yong%20Huang%20LD_PRELOAD.txt
- Kevin Pulo — Fun with LD_PRELOAD. A lot of examples and ideas. http://yurichev.com/mirrors/LD_PRELOAD/lca2009.pdf
- File functions interception for compression/decompression files on fly (zlibc). ftp://metalab.unc.edu/pub/Linux/libs/compression

Chapter 48

Windows NT

48.1 CRT (win32)

Does program execution starts right at the `main()` function? No, it is not. If to open any executable file in IDA or HIEW, we will see OEP[1] pointing to another code. This is a code doing some maintenance and preparations before passing control flow to our code. It is called startup-code or CRT-code (C RunTime).

`main()` fucntion takes an array of arguments passed in the command line, and also environment variables. But in fact, a generic string is passed to the program, CRT-code will find spaces in it and cut by parts. CRT-code is also prepares environment variables array `envp`. As of GUI[2] win32 applications, `WinMain` is used instead of `main()`, having their own arguments:

```
int CALLBACK WinMain(
  _In_  HINSTANCE hInstance,
  _In_  HINSTANCE hPrevInstance,
  _In_  LPSTR lpCmdLine,
  _In_  int nCmdShow
);
```

CRT-code prepares them as well.

Also, the number returned by `main()` function is an exit code. It may be passed in CRT to the `ExitProcess()` function, taking exit code as argument.

Usually, each compiler has its own CRT-code.

Here is a typical CRT-code for MSVC 2008.

[1] Original Entry Point
[2] Graphical user interface

```
___tmainCRTStartup proc near

var_24 = dword ptr -24h
var_20 = dword ptr -20h
var_1C = dword ptr -1Ch
ms_exc = CPPEH_RECORD ptr -18h

        push    14h
        push    offset stru_4092D0
        call    __SEH_prolog4
        mov     eax, 5A4Dh
        cmp     ds:400000h, ax
        jnz     short loc_401096
        mov     eax, ds:40003Ch
        cmp     dword ptr [eax+400000h], 4550h
        jnz     short loc_401096
        mov     ecx, 10Bh
        cmp     [eax+400018h], cx
        jnz     short loc_401096
        cmp     dword ptr [eax+400074h], 0Eh
        jbe     short loc_401096
        xor     ecx, ecx
        cmp     [eax+4000E8h], ecx
        setnz   cl
        mov     [ebp+var_1C], ecx
        jmp     short loc_40109A


loc_401096: ; CODE XREF: ___tmainCRTStartup+18
            ; ___tmainCRTStartup+29 ...
        and     [ebp+var_1C], 0

loc_40109A: ; CODE XREF: ___tmainCRTStartup+50
        push    1
        call    __heap_init
        pop     ecx
        test    eax, eax
        jnz     short loc_4010AE
        push    1Ch
        call    _fast_error_exit
        pop     ecx

loc_4010AE: ; CODE XREF: ___tmainCRTStartup+60
```

```
        call    __mtinit
        test    eax, eax
        jnz     short loc_4010BF
        push    10h
        call    _fast_error_exit
        pop     ecx

loc_4010BF: ; CODE XREF: ___tmainCRTStartup+71
        call    sub_401F2B
        and     [ebp+ms_exc.disabled], 0
        call    __ioinit
        test    eax, eax
        jge     short loc_4010D9
        push    1Bh
        call    __amsg_exit
        pop     ecx

loc_4010D9: ; CODE XREF: ___tmainCRTStartup+8B
        call    ds:GetCommandLineA
        mov     dword_40B7F8, eax
        call    ___crtGetEnvironmentStringsA
        mov     dword_40AC60, eax
        call    __setargv
        test    eax, eax
        jge     short loc_4010FF
        push    8
        call    __amsg_exit
        pop     ecx

loc_4010FF: ; CODE XREF: ___tmainCRTStartup+B1
        call    __setenvp
        test    eax, eax
        jge     short loc_401110
        push    9
        call    __amsg_exit
        pop     ecx

loc_401110: ; CODE XREF: ___tmainCRTStartup+C2
        push    1
        call    __cinit
        pop     ecx
        test    eax, eax
        jz      short loc_401123
```

```
        push    eax
        call    __amsg_exit
        pop     ecx

loc_401123: ; CODE XREF: ___tmainCRTStartup+D6
        mov     eax, envp
        mov     dword_40AC80, eax
        push    eax             ; envp
        push    argv            ; argv
        push    argc            ; argc
        call    _main
        add     esp, 0Ch
        mov     [ebp+var_20], eax
        cmp     [ebp+var_1C], 0
        jnz     short $LN28
        push    eax             ; uExitCode
        call    $LN32

$LN28:      ; CODE XREF: ___tmainCRTStartup+105
        call    __cexit
        jmp     short loc_401186


$LN27:      ; DATA XREF: .rdata:stru_4092D0
        mov     eax, [ebp+ms_exc.exc_ptr] ; Exception filter 0 ↙
    ↳ for function 401044
        mov     ecx, [eax]
        mov     ecx, [ecx]
        mov     [ebp+var_24], ecx
        push    eax
        push    ecx
        call    __XcptFilter
        pop     ecx
        pop     ecx

$LN24:
        retn


$LN14:      ; DATA XREF: .rdata:stru_4092D0
        mov     esp, [ebp+ms_exc.old_esp] ; Exception handler 0 ↙
    ↳ for function 401044
        mov     eax, [ebp+var_24]
```

```
        mov     [ebp+var_20], eax
        cmp     [ebp+var_1C], 0
        jnz     short $LN29
        push    eax             ; int
        call    __exit


$LN29:      ; CODE XREF: ___tmainCRTStartup+135
        call    __c_exit

loc_401186: ; CODE XREF: ___tmainCRTStartup+112
        mov     [ebp+ms_exc.disabled], 0FFFFFFFEh
        mov     eax, [ebp+var_20]
        call    __SEH_epilog4
        retn
```

Here we may see calls to `GetCommandLineA()`, then to `setargv()` and `setenvp()`, which are, apparently, fills global variables `argc`, `argv`, `envp`.

Finally, `main()` is called with these arguments.

There are also calls to the functions having self-describing names like `heap_init()`, `ioinit()`.

Heap is indeed initialized in CRT: if you will try to use `malloc()`, the program exiting abnormally with the error:

```
runtime error R6030
- CRT not initialized
```

Global objects initializations in C++ is also occurred in the CRT before `main()`: 29.4.1.

A variable `main()` returns is passed to `cexit()`, or to `$LN32`, which in turn calling `doexit()`.

Is it possible to get rid of CRT? Yes, if you know what you do.

MSVC linker has `/ENTRY` option for setting entry point.

```
#include <windows.h>

int main()
{
        MessageBox (NULL, "hello, world", "caption", MB_OK);
};
```

Let's compile it in MSVC 2008.

```
cl no_crt.c user32.lib /link /entry:main
```

We will get a runnable .exe with size 2560 bytes, there are PE-header inside, instructions calling `MessageBox`, two strings in the data segment, `MessageBox` function imported from `user32.dll` and nothing else.

This works, but you will not be able to write `WinMain` with its 4 arguments instead of `main()`. To be correct, you will be able to write so, but arguments will not be prepared at the moment of execution.

By the way, it is possible to make .exe even shorter by doing PE[3]-section aligning less than default 4096 bytes.

```
cl no_crt.c user32.lib /link /entry:main /align:16
```

Linker will say:

```
LINK : warning LNK4108: /ALIGN specified without /DRIVER; image ↙
    ↳ may not run
```

We getting .exe of 720 bytes size. It running in Windows 7 x86, but not in x64 (the error message will be showed when trying to execute). By applying even more efforts, it is possible to make executable even shorter, but as you can see, compatibility problems may arise quickly.

48.2 Win32 PE

PE is a executable file format used in Windows.

The difference between .exe, .dll and .sys is that .exe and .sys usually does not have exports, only imports.

A DLL[4], just as any other PE-file, has entry point (OEP) (the function DllMain() is located at it) but usually this function does nothing.

.sys is usually device driver.

As of drivers, Windows require the checksum is present in PE-file and must be correct [5].

Starting at Windows Vista, driver PE-files must be also signed by digital signature. It will fail to load without signature.

Any PE-file begins with tiny DOS-program, printing a message like "This program cannot be run in DOS mode." — if to run this program in DOS or Windows 3.1, this message will be printed.

48.2.1 Terminology

- Module — is a separate file, .exe or.dll.

[3] Portable Executable: 48.2

[4] Dynamic-link library

[5] For example, Hiew(53) can calculate it

- Process — a program loaded into memory and running. Commonly consisting of one .exe-file and bunch of .dll-files.
- Process memory — the memory a process works with. Each process has its own. There can usually be loaded modules, memory of the stack, heap(s), etc.
- VA[6] — is address which will be used in program.
- Base address—is the address within a process memory at which a module will be loaded.
- RVA[7]—is a VA-address minus base address. Many addresses in PE-file tables uses exactly RVA-addresses.
- IAT[8]—an array of addresses of imported symbols [9]. Sometimes, a `IMAGE_DIRECTORY_EN` data directory is points to the IAT. It is worth to note that IDA (as of 6.1) may allocate a pseudo-section named `.idata` for IAT, even if IAT is a part of another section!
- INT[10]—an array of names of symbols to be imported[11].

48.2.2 Base address

The fact is that several module authors may prepare DLL-files for others and there is no possibility to reach agreement, which addresses will be assigned to whose modules.

So that is why if two necessary for process loading DLLs has the same base addresses, one of which will be loaded at this base address, and another —at the other spare space in process memory, and each virtual addresses in the second DLL will be corrected.

Often, linker in MSVC generates an .exe-files with the base address `0x400000`, and with the code section started at `0x401000`. This mean RVA of code section begin is `0x1000`. DLLs are often generated by this linked with the base address `0x10000000` [12].

There is also another reason to load modules at various base addresses, rather at random ones.

It is ASLR[13] [14].

[6] Virtual Address
[7] Relative Virtual Address
[8] Import Address Table
[9] [24]
[10] Import Name Table
[11] [24]
[12] This can be changed by /BASE linker option
[13] Address Space Layout Randomization
[14] https://en.wikipedia.org/wiki/Address_space_layout_randomization

The fact is that a shellcode trying to be executed on a compromised system must call a system functions.

In older OS (in Windows NT line: before Windows Vista), system DLL (like kernel32.dll, user32.dll) were always loaded at the known addresses, and also if to recall that its versions were rarely changed, an addresses of functions were fixed and shellcode can call it directly.

In order to avoid this, ASLR method loads your program and all modules it needs at random base addresses, each time different.

ASLR support is denoted in PE-file by setting the flag `IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE` [30].

48.2.3 Subsystem

There is also *subsystem* field, usually it is native (.sys-driver), console (console application) or GUI (non-console).

48.2.4 OS version

A PE-file also has minimal Windows version needed in order to load it. The table of version numbers stored in PE-file and corresponding Windows codenames is here[15].

For example, MSVC 2005 compiles .exe-files running on Windows NT4 (version 4.00), but MSVC 2008 is not (files generated has version 5.00, at least Windows 2000 is needed to run them).

MSVC 2012 by default generates .exe-files of version 6.00, targeting at least Windows Vista, however, by by changing compiler's options [16], it is possible to force it to compile for Windows XP.

48.2.5 Sections

Division by sections, as it seems, are present in all executable file formats.

It is done in order to separate code from data, and data —from constant data.

- There will be flag *IMAGE_SCN_CNT_CODE* or *IMAGE_SCN_MEM_EXECUTE* on code section—this is executable code.
- On data section—*IMAGE_SCN_CNT_INITIALIZED_DATA*, *IMAGE_SCN_MEM_READ* and *IMAGE_SCN_MEM_WRITE* flags.
- On an empty section with uninitialized data—*IMAGE_SCN_CNT_UNINITIALIZED_DATA*, *IMAGE_SCN_MEM_READ* and *IMAGE_SCN_MEM_WRITE*.

[15] https://en.wikipedia.org/wiki/Windows_NT#Releases

[16] http://blogs.msdn.com/b/vcblog/archive/2012/10/08/10357555.aspx

- On a constant data section, in other words, protected from writing, there are may be flags *IMAGE_SCN_CNT_INITIALIZED_DATA* and *IMAGE_SCN_MEM_READ* without *IMAGE_SCN_ME* A process will crash if it would try to write to this section.

Each section in PE-file may have a name, however, it is not very important. Often (but not always) code section have the name `.text`, data section — `.data`, constant data section — `.rdata` *(readable data)*. Other popular section names are:

- `.idata`—imports section. IDA may create pseudo-section named like this: 48.2.1.
- `.edata`—exports section
- `.pdata`—section containing all information about exceptions in Windows NT for MIPS, IA64 and x64: 48.3.3
- `.reloc`—relocs section
- `.bss`—uninitialized data (BSS)
- `.tls`—thread local storage (TLS)
- `.rsrc`—resources
- `.CRT`—may present in binary files compiled by very old MSVC versions

PE-file packers/encryptors are often garble section names or replacing names to their own.

MSVC allows to declare data in arbitrarily named section [17].

Some compilers and linkers can add a section with debugging symbols and other debugging information (e.g. MinGW). However it is not so in modern versions of MSVC (a separate PDB-files are used there for this purpose).

That is how section described in the file:

```
typedef struct _IMAGE_SECTION_HEADER {
  BYTE  Name[IMAGE_SIZEOF_SHORT_NAME];
  union {
    DWORD PhysicalAddress;
    DWORD VirtualSize;
  } Misc;
  DWORD VirtualAddress;
  DWORD SizeOfRawData;
  DWORD PointerToRawData;
  DWORD PointerToRelocations;
  DWORD PointerToLinenumbers;
  WORD  NumberOfRelocations;
```

[17] http://msdn.microsoft.com/en-us/library/windows/desktop/cc307397.aspx

```
  WORD  NumberOfLinenumbers;
  DWORD Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

[18]

A word about terminology: *PointerToRawData* it called "Offset" and *VirtualAddress* is called "RVA" in Hiew.

48.2.6 Relocations (relocs)

AKA FIXUP-s (at least in Hiew).

This is also present in almost all executable file formats [19].

Obviously, modules can be loaded on various base addresses, but how to deal with e.g. global variables? They must be accessed by an address. One solution is position-independent code(47.1). But it is not always suitable.

That is why relocations table is present. The addresses of points needs to be corrected in case of loading on another base address are just enumerated in the table.

For example, there is a global variable at the address `0x410000` and this is how it is accessed:

```
A1 00 00 41 00          mov         eax,[000410000]
```

Base address of module is `0x400000`, RVA of global variable is `0x10000`.

If the module is loading on the base address `0x500000`, the factual address of the global variable must be `0x510000`.

As we can see, address of variable is encoded in the instruction `MOV`, after the byte `0xA1`.

That is why address of 4 bytes, after `0xA1`, is written into relocs table.

, OS-loader enumerates all addresses in table, finds each 32-bit word the address points on, subtracts real, original base address of it (we getting RVA here), and adds new base address to it.

If module is loading on original base address, nothing happens.

All global variables may be treated like that.

Relocs may have various types, however, in Windows, for x86 processors, the type is usually
IMAGE_REL_BASED_HIGHLOW.

By the way, relocs are darkened in Hiew, for example fig. 6.12.

48.2.7 Exports and imports

As all we know, any executable program must use OS services and other DLL-libraries somehow.

[18] http://msdn.microsoft.com/en-us/library/windows/desktop/ms680341(v=vs.85).aspx
[19] Even .exe-files in MS-DOS

It can be said, functions from one module (usually DLL) must be connected somehow to a points of their calls in other module (.exe-file or another DLL).

Each DLL has "exports" for this, this is table of functions plus its addresses in a module.

Each .exe-file or DLL has "imports", this is a table of functions it needs for execution including list of DLL filenames.

After loading main .exe-file, OS-loader, processes imports table: it loads additional DLL-files, finds function names among DLL exports and writes their addresses down in an IAT of main .exe-module.

As we can notice, during loading, loader must compare a lot of function names, but strings comparison is not a very fast procedure, so, there is a support of "ordinals" or "hints", that is a function numbers stored in the table instead of their names.

That is how they can be located faster in loading DLL. Ordinals are always present in "export" table.

For example, program using MFC[20] library usually loads mfc*.dll by ordinals, and in such programs there are no MFC function names in INT.

While loading such program in IDA, it will asks for a path to mfs*.dll files, in order to determine function names. If not to tell IDA path to this DLL, they will look like *mfc80_123* instead of function names.

Imports section

Often a separate section is allocated for imports table and everything related to it (with name like `.idata`), however, it is not a strict rule.

Imports is also confusing subject because of terminological mess. Let's try to collect all information in one place.

[20] Microsoft Foundation Classes

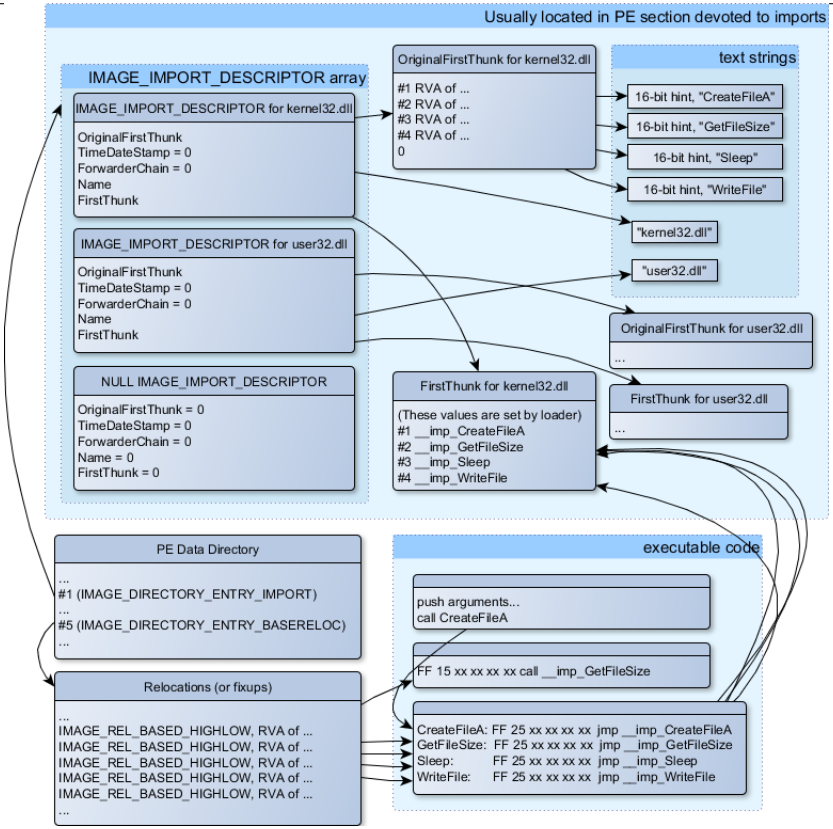


Figure 48.1: The scheme, uniting all PE-file structures related to imports

Main structure is the array of *IMAGE_IMPORT_DESCRIPTOR*. Each element for each DLL being imported.

Each element holds RVA-address of text string (DLL name) (*Name*).

OriginalFirstThink is a RVA-address of INT table. This is array of RVA-addresses, each of which points to the text string with function name. Each string is prefixed by 16-bit integer ("hint")—"ordinal" of function.

While loading, if it is possible to find function by ordinal, then strings comparison will not occur. Array is terminated by zero. There is also a pointer to the IAT table with a name *FirstThunk*, it is just RVA-address of the place where loader will write addresses of functions resolved.

The points where loader writes addresses, IDA marks like: *__imp_CreateFileA*, etc.

There are at least two ways to use addresses written by loader.

- The code will have instructions like *call __imp_CreateFileA*, and since the field with the address of function imported is a global variable in some sense, the

address of *call* instruction (plus 1 or 2) will be added to relocs table, for the case if module will be loaded on different base address.

But, obviously, this may enlarge relocs table significantly. Because there are might be a lot of calls to imported functions in the module. Furthermore, large relocs table slowing down the process of module loading.

- For each imported function, there is only one jump allocated, using `JMP` instruction plus reloc to this instruction. Such points are also called "thunks". All calls to the imported functions are just `CALL` instructions to the corresponding "thunk". In this case, additional relocs are not necessary because these CALL-s has relative addresses, they are not to be corrected.

Both of these methods can be combined. Apparently, linker creates individual "thunk" if there are too many calls to the functions, but by default it is not to be created.

By the way, an array of function addresses to which FirstThunk is pointing is not necessary to be located in IAT section. For example, I once wrote the PE_add_import[21] utility for adding import to an existing .exe-file. Some time earlier, in the previous versions of the utility, at the place of the function you want to substitute by call to another DLL, the following code my utility writed:

```
MOV EAX, [yourdll.dll!function]
JMP EAX
```

FirstThunk points to the first instruction. In other words, while loading yourdll.dll, loader writes address of the *function* function right in the code.

It also worth noting a code section is usually write-protected, so my utility adds *IMAGE_SCN_MEM_WRITE* flag for code section. Otherwise, the program will crash while loading with the error code 5 (access denied).

One might ask: what if I supply a program with the DLL files set which are not supposed to change, is it possible to speed up loading process?

Yes, it is possible to write addresses of the functions to be imported into FirstThunk arrays in advance. The *Timestamp* field is present in the *IMAGE_IMPORT_DESCRIPTOR* structure. If a value is present there, then loader compare this value with date-time of the DLL file. If the values are equal to each other, then the loader is not do anything, and loading process will be faster. This is what called "old-style binding" [22]. There is the BIND.EXE utility in Windows SDK for this. For speeding up of loading of your program, Matt Pietrek in [24], offers to do binding shortly after your program installation on the computer of the end user.

[21] http://yurichev.com/PE_add_imports.html

[22] http://blogs.msdn.com/b/oldnewthing/archive/2010/03/18/9980802.aspx. There is also "new-style binding", I will write about it in future

PE-files packers/encryptors may also compress/encrypt imports table. In this case, Windows loader, of course, will not load all necessary DLLs. Therefore, packer/encryptor do this on its own, with the help of *LoadLibrary()* and *GetProcAddress()* functions.

In the standard DLLs from Windows installation, often, IAT is located right in the beginning of PE-file. Supposedly, it is done for optimization. While loading, .exe file is not loaded into memory as a whole (recall huge install programs which are started suspiciously fast), it is "mapped", and loaded into memory by parts as they are accessed. Probably, Microsoft developers decided it will be faster.

48.2.8 Resources

Resources in a PE-file is just a set of icons, pictures, text strings, dialog descriptions. Perhaps, they were separated from the main code, so all these things could be multilingual, and it would be simpler to pick text or picture for the language that is currently set in OS.

As a side effect, they can be edited easily and saved back to the executable file, even, if one does not have special knowledge, e.g. using ResHack editor(48.2.11).

48.2.9 .NET

.NET programs are compiled not into machine code but into special bytecode. Strictly speaking, there is bytecode instead of usual x86-code in the .exe-file, however, entry point (OEP) is pointing to the tiny fragment of x86-code:

```
jmp          mscoree.dll!_CorExeMain
```

.NET-loader is located in mscoree.dll, it will process the PE-file. It was so in pre-Windows XP OS. Starting from XP, OS-loader able to detect the .NET-file and run it without execution of that JMP instruction [23].

48.2.10 TLS

This section holds initialized data for TLS(45) (if needed). When new thread starting, its TLS-data is initialized by the data from this section.

Aside from that, PE-file specification also provides initialization of TLS-section, so-called, TLS callbacks. If they are present, they will be called before control passing to the main entry point (OEP). This is used widely in the PE-file packers/encryptors.

[23] http://msdn.microsoft.com/en-us/library/xh0859k0(v=vs.110).aspx

48.2.11 Tools

- objdump (from cygwin) for dumping all PE-file structures.
- Hiew(53) as editor.
- pefile — Python-library for PE-file processing [24].
- ResHack AKA Resource Hacker — resources editor [25].
- PE_add_import[26] — simple tool for adding symbol(s) to PE executable import table.
- PE_patcher[27] — simple tool for patching PE executables.
- PE_search_str_refs[28] — simple tool for searching for a function in PE executables which use some text string.

48.2.12 Further reading

- Daniel Pistelli — The .NET File Format [29]

48.3 Windows SEH

48.3.1 Let's forget about MSVC

In Windows, SEH is intended for exceptions handling, nevertheless, it is language-agnostic, it is not connected to the C++ or OOP in any way. Here we will take a look on SEH in isolated (from C++ and MSVC extensions) form.

Each running process has a chain of SEH-handlers, TIB has address of the last handler. When exception occurred (division by zero, incorrect address access, user exception triggered by calling to `RaiseException()` function), OS will find the last handler in TIB, and will call it with passing all information about CPU state (register values, etc) at the moment of exception. Exception handler will consider exception, was it made for it? If so, it will handle exception. If no, it will signal to OS that it cannot handle it and OS will call next handler in chain, until a handler which is able to handle the exception will be found.

At the very end of the chain, there a standard handler, showing well-known dialog box, informing a process crash, some technical information about CPU state at the crash, and offering to collect all information and send it to developers in Microsoft.

[24] https://code.google.com/p/pefile/
[25] http://www.angusj.com/resourcehacker/
[26] http://yurichev.com/PE_add_imports.html
[27] http://yurichev.com/PE_patcher.html
[28] http://yurichev.com/PE_search_str_refs.html
[29] http://www.codeproject.com/Articles/12585/The-NET-File-Format

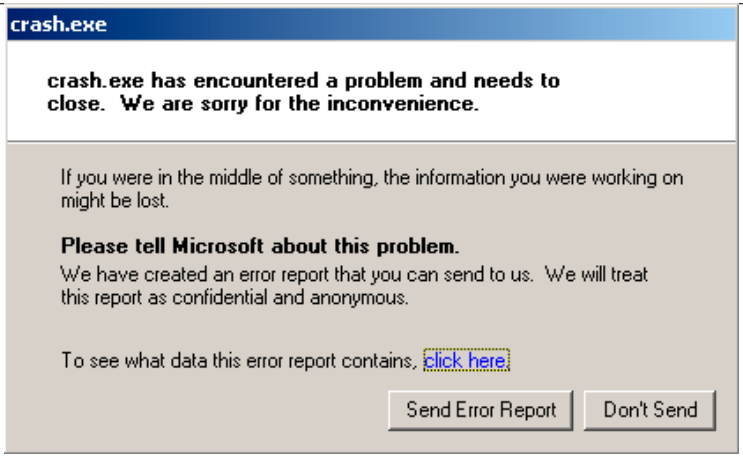


Figure 48.2: Windows XP

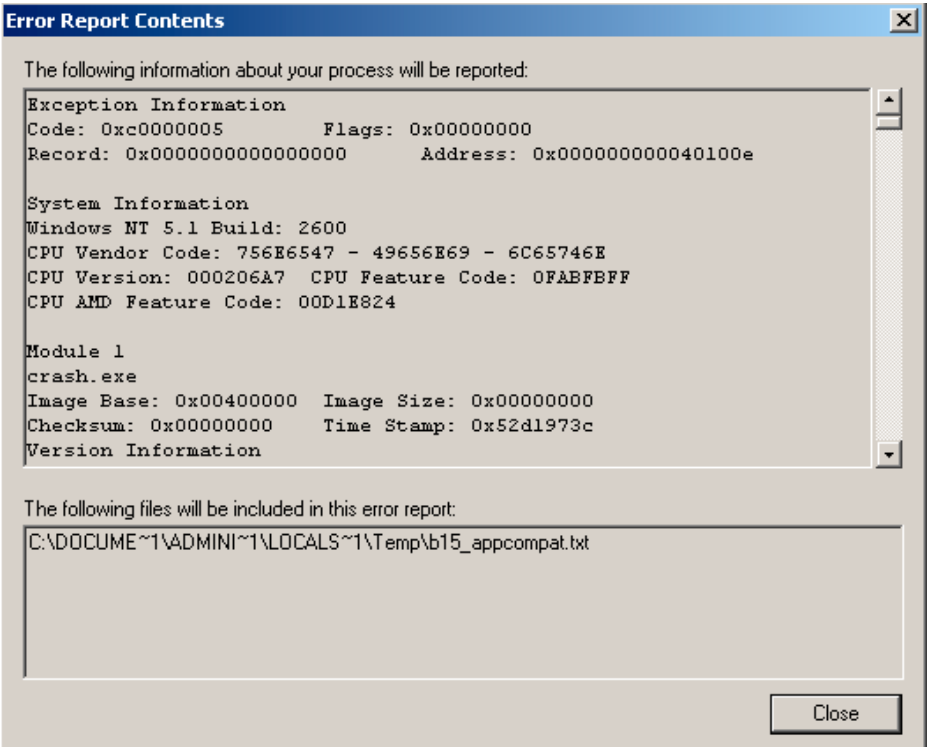


Figure 48.3: Windows XP

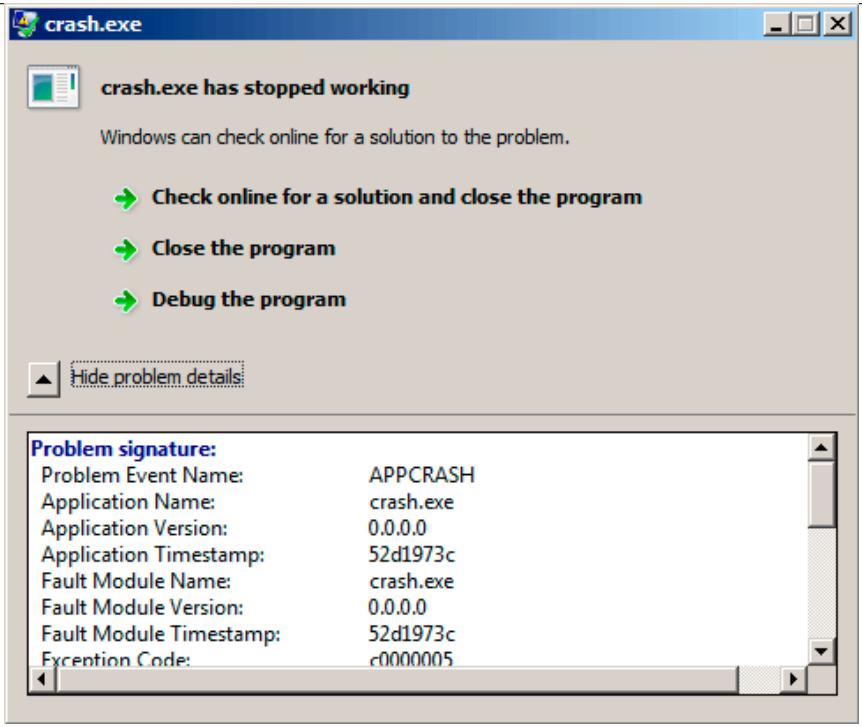


Figure 48.4: Windows 7

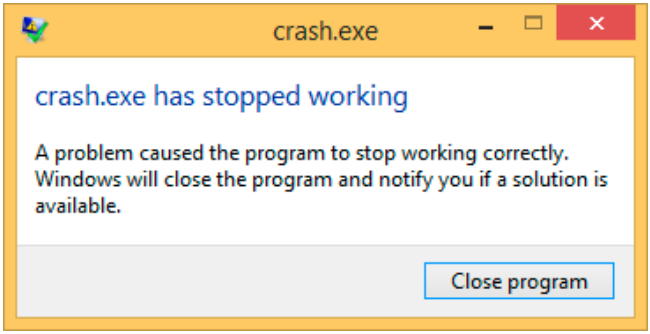


Figure 48.5: Windows 8.1

This handler was also called Dr. Watson earlier [30].

By the way, some developers made their own handler, sending information about

[30] https://en.wikipedia.org/wiki/Dr._Watson_(debugger)

program crash to themselves. It is registered with the help of SetUnhandledExceptionFilter and will be called if OS do not have any other way to handle exception. Other example is Oracle RDBMS it saves huge dumps containing all possible information about CPU and memory state.

Let's write our own primitive exception handler [31]:

```
#include <windows.h>
#include <stdio.h>

DWORD new_value=1234;

EXCEPTION_DISPOSITION __cdecl except_handler(
                struct _EXCEPTION_RECORD *ExceptionRecord,
                void * EstablisherFrame,
                struct _CONTEXT *ContextRecord,
                void * DispatcherContext )
{
        unsigned i;

        printf ("%s\n", __FUNCTION__);
        printf ("ExceptionRecord->ExceptionCode=0x%p\n", ↙
    ↳ ExceptionRecord->ExceptionCode);
        printf ("ExceptionRecord->ExceptionFlags=0x%p\n", ↙
    ↳ ExceptionRecord->ExceptionFlags);
        printf ("ExceptionRecord->ExceptionAddress=0x%p\n", ↙
    ↳ ExceptionRecord->ExceptionAddress);

        if (ExceptionRecord->ExceptionCode==0xE1223344)
        {
                printf ("That's for us\n");
                // yes, we "handled" the exception
                return ExceptionContinueExecution;
        }
        else if (ExceptionRecord->ExceptionCode==↙
    ↳ EXCEPTION_ACCESS_VIOLATION)
        {
                printf ("ContextRecord->Eax=0x%08X\n", ↙
    ↳ ContextRecord->Eax);
                // will it be possible to 'fix' it?
                printf ("Trying to fix wrong pointer address\n")↙
```

[31] The example is based on the example from [23]
It is compiled with the SAFESEH option: `cl seh1.cpp /link /safeseh:no`
More about SAFESEH here:
http://msdn.microsoft.com/en-us/library/9a89h429.aspx

```
    ↪ ;
                ContextRecord->Eax=(DWORD)&new_value;
                // yes, we "handled" the exception
                return ExceptionContinueExecution;
        }
        else
        {
                printf ("We do not handle this\n");
                // someone else's problem
                return ExceptionContinueSearch;
        };
}

int main()
{
        DWORD handler = (DWORD)except_handler; // take a pointer↙
    ↪  to our handler

        // install exception handler
        __asm
        {                                   // make ↙
    ↪ EXCEPTION_REGISTRATION record:
                push    handler             // address of handler ↙
    ↪ function
                push    FS:[0]              // address of previous ↙
    ↪ handler
                mov     FS:[0],ESP          // add new ↙
    ↪ EXECEPTION_REGISTRATION
        }

        RaiseException (0xE1223344, 0, 0, NULL);

        // now do something very bad
        int* ptr=NULL;
        int val=0;
        val=*ptr;
        printf ("val=%d\n", val);

        // deinstall exception handler
        __asm
        {                                   // remove our ↙
    ↪ EXECEPTION_REGISTRATION record
                mov     eax,[ESP]           // get pointer to ↙
```

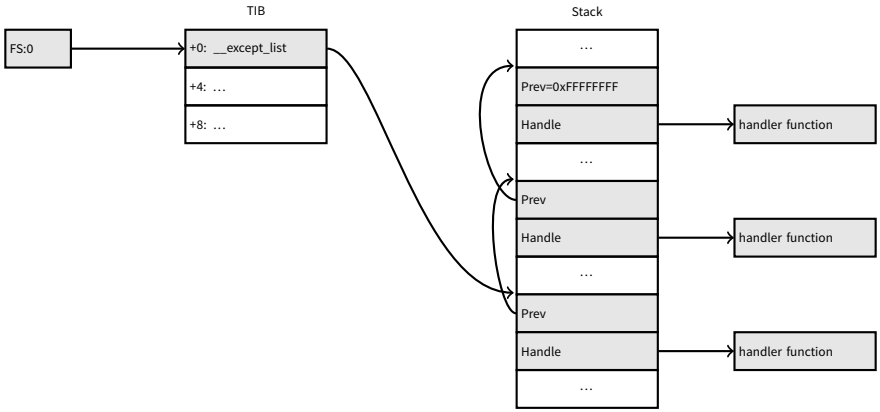
```
    ↳ previous record
                mov     FS:[0], EAX     // install previous ↙
    ↳ record
                add     esp, 8          // clean our ↙
    ↳ EXECEPTION_REGISTRATION off stack
        }

        return 0;
}
```

FS: segment register is pointing to the TIB in win32. The very first element in TIB is a pointer to the last handler in chain. We saving it in the stack and store an address of our handler there. The structure is named `_EXCEPTION_REGISTRATION`, it is a simplest singly-linked list and its elements are stored right in the stack.

Listing 48.1: MSVC/VC/crt/src/exsup.inc

```
\_EXCEPTION\_REGISTRATION struc
    prev    dd      ?
    handler dd      ?
\_EXCEPTION\_REGISTRATION ends
```

So each "handler" field points to handler and an each "prev" field points to previous record in the stack. The last record has `0xFFFFFFFF` (-1) in "prev" field.



When our handler is installed, let's call `RaiseException()` [32]. This is user exception. Handler will check the code. If the code is `0xE1223344`, it will return `ExceptionContinue` which means that handler fixes CPU state (it is usually EIP/ESP) and the OS can resume thread execution. If to alter the code slightly so the handler will return `ExceptionContin` then OS will call other handlers, and very unlikely the one who can handle it will be

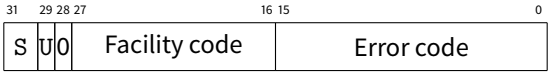
[32] http://msdn.microsoft.com/en-us/library/windows/desktop/ms680552(v=vs.85).aspx

founded, since no one have information about it (rather about its code). You will see the standard Windows dialog about process crash.

What is the difference between system exceptions and user? Here is a system ones:

as defined in WinBase.h	as defined in ntstatus.h
EXCEPTION_ACCESS_VIOLATION	STATUS_ACCESS_VIOLATION
EXCEPTION_DATATYPE_MISALIGNMENT	STATUS_DATATYPE_MISALIGNMENT
EXCEPTION_BREAKPOINT	STATUS_BREAKPOINT
EXCEPTION_SINGLE_STEP	STATUS_SINGLE_STEP
EXCEPTION_ARRAY_BOUNDS_EXCEEDED	STATUS_ARRAY_BOUNDS_EXCEEDED
EXCEPTION_FLT_DENORMAL_OPERAND	STATUS_FLOAT_DENORMAL_OPERAND
EXCEPTION_FLT_DIVIDE_BY_ZERO	STATUS_FLOAT_DIVIDE_BY_ZERO
EXCEPTION_FLT_INEXACT_RESULT	STATUS_FLOAT_INEXACT_RESULT
EXCEPTION_FLT_INVALID_OPERATION	STATUS_FLOAT_INVALID_OPERATION
EXCEPTION_FLT_OVERFLOW	STATUS_FLOAT_OVERFLOW
EXCEPTION_FLT_STACK_CHECK	STATUS_FLOAT_STACK_CHECK
EXCEPTION_FLT_UNDERFLOW	STATUS_FLOAT_UNDERFLOW
EXCEPTION_INT_DIVIDE_BY_ZERO	STATUS_INTEGER_DIVIDE_BY_ZERO
EXCEPTION_INT_OVERFLOW	STATUS_INTEGER_OVERFLOW
EXCEPTION_PRIV_INSTRUCTION	STATUS_PRIVILEGED_INSTRUCTION
EXCEPTION_IN_PAGE_ERROR	STATUS_IN_PAGE_ERROR
EXCEPTION_ILLEGAL_INSTRUCTION	STATUS_ILLEGAL_INSTRUCTION
EXCEPTION_NONCONTINUABLE_EXCEPTION	STATUS_NONCONTINUABLE_EXCEPTION
EXCEPTION_STACK_OVERFLOW	STATUS_STACK_OVERFLOW
EXCEPTION_INVALID_DISPOSITION	STATUS_INVALID_DISPOSITION
EXCEPTION_GUARD_PAGE	STATUS_GUARD_PAGE_VIOLATION
EXCEPTION_INVALID_HANDLE	STATUS_INVALID_HANDLE
EXCEPTION_POSSIBLE_DEADLOCK	STATUS_POSSIBLE_DEADLOCK
CONTROL_C_EXIT	STATUS_CONTROL_C_EXIT

That is how code is defined:



S is a basic status code: 11—error; 10—warning; 01—informational; 00—success. U—whether the code is user code.

That is why I chose 0xE1223344— 0xE (1110b) mean this is 1) user exception; 2) error. But to be honest, this example works finely without these high bits.

Then we try to read a value from memory at the 0th address. Of course, there are nothing at this address in win32, so exception is raised. However, the very first handler will be called — yours, it will be notified first, checking the code on equality to the `EXCEPTION_ACCESS_VIOLATION` constant.

The code reading from memory at 0th address is looks like:

Listing 48.2: MSVC 2010

```
...
xor     eax, eax
mov     eax, DWORD PTR [eax] ; exception will occur here
push    eax
push    OFFSET msg
call    _printf
add     esp, 8
...
```

Will it be possible to fix error "on fly" and to continue program execution? Yes, our exception handler can fix EAX value and now let OS will execute this instruction once again. So that is what we do. `printf()` will print 1234, because, after execution of our handler, EAX will not be 0, it will contain address of global variable `new_value`. Execution will be resumed.

That is what is going on: memory manager in CPU signaling about error, the CPU suspends the thread, it finds exception handler in the Windows kernel, latter, in turn, is starting to call all handlers in SEH chain, one by one.

I use MSVC 2010 now, but of course, there are no any guarantee that EAX will be used for pointer.

This address replacement trick is looks showingly, and I offer it here for SEH internals illustration. Nevertheless, I cannot recall where it is used for "on-fly" error fixing in practice.

Why SEH-related records are stored right in the stack instead of some other place? Supposedly because then OS will not need to care about freeing this information, these records will be disposed when function finishing its execution. But I'm not 100%-sure and can be wrong. This is somewhat like alloca(): (4.2.4).

48.3.2 Now let's get back to MSVC

Supposedly, Microsoft programmers need exceptions in C, but not in C++, so they added a non-standard C extension to MSVC[33]. It is not related to C++ PL exceptions.

```
__try
{
    ...
}
__except(filter code)
{
    handler code
}
```

[33] http://msdn.microsoft.com/en-us/library/swezty51.aspx

"Finally" block may be instead of handler code:

```
__try
{
    ...
}
__finally
{
    ...
}
```

The filter code is an expression, telling whether this handler code is coresponding to the exception raised. If your code is too big and cannot be fitted into one expression, a separate filter function can be defined.

There are a lot of such constructs in the Windows kernel. Here is couple of examples from there (WRK):

Listing 48.3: WRK-v1.2/base/ntos/ob/obwait.c

```
try {

    KeReleaseMutant( (PKMUTANT)SignalObject,
                     MUTANT_INCREMENT,
                     FALSE,
                     TRUE );

} except((GetExceptionCode () == STATUS_ABANDONED ||
          GetExceptionCode () == STATUS_MUTANT_NOT_OWNED)?
             EXCEPTION_EXECUTE_HANDLER :
             EXCEPTION_CONTINUE_SEARCH) {
    Status = GetExceptionCode();

    goto WaitExit;
}
```

Listing 48.4: WRK-v1.2/base/ntos/cache/cachesub.c

```
try {

    RtlCopyBytes( (PVOID)((PCHAR)CacheBuffer + PageOffset),
                  UserBuffer,
                  MorePages ?
                    (PAGE_SIZE - PageOffset) :
                    (ReceivedLength - PageOffset) );
```

```
} except( CcCopyReadExceptionFilter( GetExceptionInformation(),
                                                                    &↙
    ↳ Status ) ) {
```

Here is also filter code example:

Listing 48.5: WRK-v1.2/base/ntos/cache/copysup.c

```
LONG
CcCopyReadExceptionFilter(
    IN PEXCEPTION_POINTERS ExceptionPointer,
    IN PNTSTATUS ExceptionCode
    )

/*++

Routine Description:

    This routine serves as a exception filter and has the ↙
    ↳ special job of
    extracting the "real" I/O error when Mm raises ↙
    ↳ STATUS_IN_PAGE_ERROR
    beneath us.

Arguments:

    ExceptionPointer - A pointer to the exception record that ↙
    ↳ contains
                       the real Io Status.

    ExceptionCode - A pointer to an NTSTATUS that is to receive ↙
    ↳ the real
                    status.

Return Value:

    EXCEPTION_EXECUTE_HANDLER

--*/

{
    *ExceptionCode = ExceptionPointer->ExceptionRecord->↙
    ↳ ExceptionCode;

    if ( (*ExceptionCode == STATUS_IN_PAGE_ERROR) &&
```

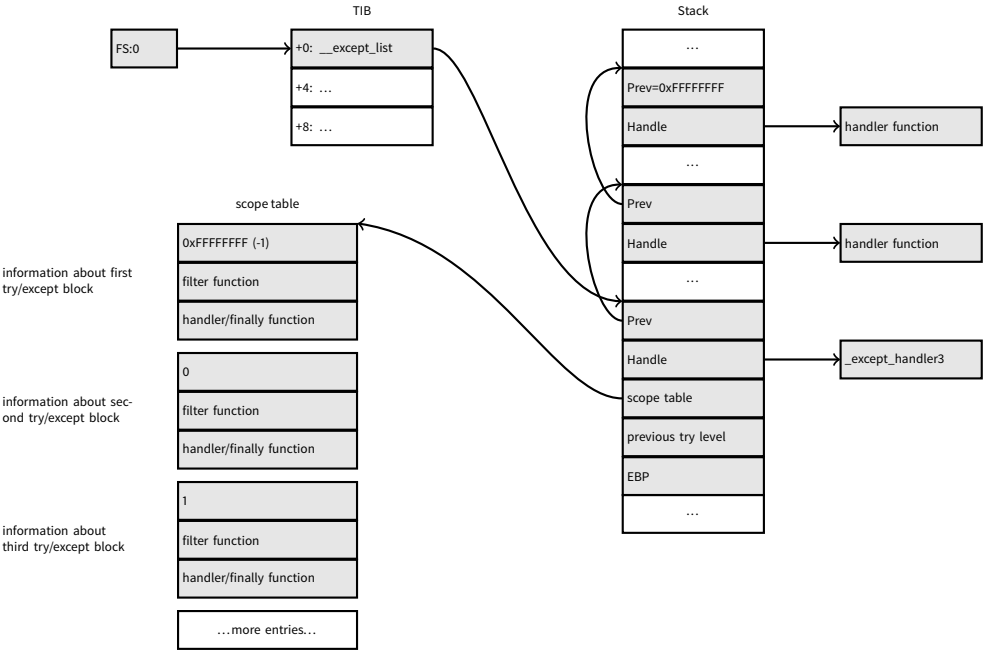
```
        (ExceptionPointer->ExceptionRecord->NumberParameters >=↙
    ↳ 3) ) {

        *ExceptionCode = (NTSTATUS) ExceptionPointer->↙
    ↳ ExceptionRecord->ExceptionInformation[2];
    }

    ASSERT( !NT_SUCCESS(*ExceptionCode) );

    return EXCEPTION_EXECUTE_HANDLER;
}
```

Internally, SEH is an extension of OS-supported exceptions. But the handler function is `_except_handler3` (for SEH3) or `_except_handler4` (for SEH4). The code of this handler is MSVC-related, it is located in its libraries, or in msvcr*.dll. It is very important to know that SEH is MSVC thing. Other compilers may offer something completely different.

SEH3

SEH3 has `_except_handler3` as handler functions, and extends `_EXCEPTION_REGISTRATION` table, adding a pointer to the *scope table* and *previous try level* variable. SEH4 extends *scope table* by 4 values for buffer overflow protection.

Scope table is a table consisting of pointers to the filter and handler codes, for each level of *try/except* nestedness.



Again, it is very important to understand that OS take care only of *prev/handle* fields, and nothing more. It is job of `_except_handler3` function to read other fields, read *scope table*, and decide, which handler to execute and when.

The source code of `_except_handler3` function is closed. However, Sanos OS, which have win32 compatibility layer, has the same functions redeveloped, which are somewhat equivalent to those in Windows [34]. Another reimplementations are present in Wine[35] and ReactOS[36].

If the *filter* pointer is zero, *handler* pointer is the pointer to a *finally* code.

During execution, *previous try level* value in the stack is changing, so the `_except_handler3` will know about current state of nestedness, in order to know which *scope table* entry to use.

SEH3: one try/except block example

```
#include <stdio.h>
```

[34] https://code.google.com/p/sanos/source/browse/src/win32/msvcrt/except.c
[35] https://github.com/mirrors/wine/blob/master/dlls/msvcrt/except_i386.c
[36] http://doxygen.reactos.org/d4/df2/lib_2sdk_2crt_2except_2except_8c_source.html

```
#include <windows.h>
#include <excpt.h>

int main()
{
    int* p = NULL;
    __try
    {
        printf("hello #1!\n");
        *p = 13;    // causes an access violation exception;
        printf("hello #2!\n");
    }
    __except(GetExceptionCode()==EXCEPTION_ACCESS_VIOLATION ?
            EXCEPTION_EXECUTE_HANDLER : ↙
    ↳ EXCEPTION_CONTINUE_SEARCH)
    {
        printf("access violation, can't recover\n");
    }
}
```

Listing 48.6: MSVC 2003

```
$SG74605 DB    'hello #1!', 0aH, 00H
$SG74606 DB    'hello #2!', 0aH, 00H
$SG74608 DB    'access violation, can''t recover', 0aH, 00H
_DATA    ENDS

; scope table

CONST     SEGMENT
$T74622   DD    0ffffffffH    ; previous try level
          DD    FLAT:$L74617  ; filter
          DD    FLAT:$L74618  ; handler

CONST    ENDS
_TEXT    SEGMENT
$T74621 = -32 ; size = 4
_p$ = -28     ; size = 4
__$SEHRec$ = -24 ; size = 24
_main    PROC NEAR
    push   ebp
    mov    ebp, esp
    push   -1                              ; previous try level
    push   OFFSET FLAT:$T74622             ; scope table
```

```
    push   OFFSET FLAT:__except_handler3   ; handler
    mov    eax, DWORD PTR fs:__except_list
    push   eax                             ; prev
    mov    DWORD PTR fs:__except_list, esp
    add    esp, -16
    push   ebx   ; saved 3 registers
    push   esi   ; saved 3 registers
    push   edi   ; saved 3 registers
    mov    DWORD PTR __$SEHRec$[ebp], esp
    mov    DWORD PTR _p$[ebp], 0
    mov    DWORD PTR __$SEHRec$[ebp+20], 0  ; previous try level
    push   OFFSET FLAT:$SG74605 ; 'hello #1!'
    call   _printf
    add    esp, 4
    mov    eax, DWORD PTR _p$[ebp]
    mov    DWORD PTR [eax], 13
    push   OFFSET FLAT:$SG74606 ; 'hello #2!'
    call   _printf
    add    esp, 4
    mov    DWORD PTR __$SEHRec$[ebp+20], -1  ; previous try ↙
    ↳ level
    jmp    SHORT $L74616

    ; filter code

$L74617:
$L74627:
    mov    ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov    edx, DWORD PTR [ecx]
    mov    eax, DWORD PTR [edx]
    mov    DWORD PTR $T74621[ebp], eax
    mov    eax, DWORD PTR $T74621[ebp]
    sub    eax, -1073741819; c0000005H
    neg    eax
    sbb    eax, eax
    inc    eax
$L74619:
$L74626:
    ret    0

    ; handler code

$L74618:
```

```
    mov    esp, DWORD PTR __$SEHRec$[ebp]
    push   OFFSET FLAT:$SG74608 ; 'access violation, can''t ↙
    ↳ recover'
    call   _printf
    add    esp, 4
    mov    DWORD PTR __$SEHRec$[ebp+20], -1 ; setting previous ↙
    ↳ try level back to -1
$L74616:
    xor    eax, eax
    mov    ecx, DWORD PTR __$SEHRec$[ebp+8]
    mov    DWORD PTR fs:__except_list, ecx
    pop    edi
    pop    esi
    pop    ebx
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP
_TEXT    ENDS
END
```

Here we see how SEH frame is being constructed in the stack. *Scope table* is located in the `CONST` segment— indeed, these fields will not be changed. An interesting thing is how *previous try level* variable is changed. Initial value is `0xFFFFFFFF` (-1). The moment when body of `try` statement is opened is marked as an instruction writing 0 to the variable. The moment when body of `try` statement is closed, -1 is returned back to it. We also see addresses of filter and handler code. Thus we can easily see the structure of *try/except* constructs in the function.

Since the SEH setup code in the function prologue may be shared between many of functions, sometimes compiler inserts a call to `SEH_prolog()` function in the prologue, which do that. SEH cleanup code may be in the `SEH_epilog()` function.

Let's try to run this example in tracer:

```
tracer.exe -l:2.exe --dump-seh
```

Listing 48.7: tracer.exe output

```
EXCEPTION_ACCESS_VIOLATION at 2.exe!main+0x44 (0x401054) ↙
    ↳ ExceptionInformation[0]=1
EAX=0x00000000 EBX=0x7efde000 ECX=0x0040cbc8 EDX=0x0008e3c8
ESI=0x00001db1 EDI=0x00000000 EBP=0x0018feac ESP=0x0018fe80
EIP=0x00401054
FLAGS=AF IF RF
```

```
* SEH frame at 0x18fe9c prev=0x18ff78 handler=0x401204 (2.exe!↙
    ↳ _except_handler3)
SEH3 frame. previous trylevel=0
scopetable entry[0]. previous try level=-1, filter=0x401070 (2.↙
    ↳ exe!main+0x60) handler=0x401088 (2.exe!main+0x78)
* SEH frame at 0x18ff78 prev=0x18ffc4 handler=0x401204 (2.exe!↙
    ↳ _except_handler3)
SEH3 frame. previous trylevel=0
scopetable entry[0]. previous try level=-1, filter=0x401531 (2.↙
    ↳ exe!mainCRTStartup+0x18d) handler=0x401545 (2.exe!↙
    ↳ mainCRTStartup+0x1a1)
* SEH frame at 0x18ffc4 prev=0x18ffe4 handler=0x771f71f5 (ntdll.↙
    ↳ dll!__except_handler4)
SEH4 frame. previous trylevel=0
SEH4 header:    GSCookieOffset=0xfffffffe GSCookieXOROffset=0x0
                EHCookieOffset=0xffffffcc EHCookieXOROffset=0x0
scopetable entry[0]. previous try level=-2, filter=0x771f74d0 (↙
    ↳ ntdll.dll!___safe_se_handler_table+0x20) handler=0↙
    ↳ x771f90eb (ntdll.dll!_TppTerminateProcess@4+0x43)
* SEH frame at 0x18ffe4 prev=0xffffffff handler=0x77247428 (↙
    ↳ ntdll.dll!_FinalExceptionHandler@16)
```

We that SEH chain consisting of 4 handlers.

First two are located in out example. Two? But we made only one? Yes, another one is setting up in CRT function `_mainCRTStartup()`, and as it seems, it handles at least FPU exceptions. Its source code can found in MSVS installation: `crt/src/winxfltr.c`.

Third is SEH4 frame in ntdll.dll, and the fourth handler is not MSVC-related located in ntdll.dll, and it has self-describing function name.

As you can see, there are 3 types of handlers in one chain: one is not related to MSVC at all (the last one) and two MSVC-related: SEH3 and SEH4.

SEH3: two try/except blocks example

```
#include <stdio.h>
#include <windows.h>
#include <excpt.h>

int filter_user_exceptions (unsigned int code, struct ↙
    ↳ _EXCEPTION_POINTERS *ep)
{
```

```
    printf("in filter. code=0x%08X\n", code);
    if (code == 0x112233)
    {
        printf("yes, that is our exception\n");
        return EXCEPTION_EXECUTE_HANDLER;
    }
    else
    {
        printf("not our exception\n");
        return EXCEPTION_CONTINUE_SEARCH;
    };
}
int main()
{
    int* p = NULL;
    __try
    {
        __try
        {
            printf ("hello!\n");
            RaiseException (0x112233, 0, 0, NULL);
            printf ("0x112233 raised. now let's crash\n");
            *p = 13;    // causes an access violation exception;
        }
        __except(GetExceptionCode()==EXCEPTION_ACCESS_VIOLATION ↙
    ↳ ?
                EXCEPTION_EXECUTE_HANDLER : ↙
    ↳ EXCEPTION_CONTINUE_SEARCH)
        {
            printf("access violation, can't recover\n");
        }
    }
    __except(filter_user_exceptions(GetExceptionCode(), ↙
    ↳ GetExceptionInformation()))
    {
        // the filter_user_exceptions() function answering to ↙
    ↳ the question
        // "is this exception belongs to this block?"
        // if yes, do the follow:
        printf("user exception caught\n");
    }
}
```

Now there are two `try` blocks. So the *scope table* now have two entries, each entry for each block. *Previous try level* is changing as execution flow entering or exiting `try` block.

Listing 48.8: MSVC 2003

```
$SG74606 DB    'in filter. code=0x%08X', 0aH, 00H
$SG74608 DB    'yes, that is our exception', 0aH, 00H
$SG74610 DB    'not our exception', 0aH, 00H
$SG74617 DB    'hello!', 0aH, 00H
$SG74619 DB    '0x112233 raised. now let''s crash', 0aH, 00H
$SG74621 DB    'access violation, can''t recover', 0aH, 00H
$SG74623 DB    'user exception caught', 0aH, 00H

_code$ = 8   ; size = 4
_ep$ = 12    ; size = 4
_filter_user_exceptions PROC NEAR
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _code$[ebp]
    push   eax
    push   OFFSET FLAT:$SG74606 ; 'in filter. code=0x%08X'
    call   _printf
    add    esp, 8
    cmp    DWORD PTR _code$[ebp], 1122867; 00112233H
    jne    SHORT $L74607
    push   OFFSET FLAT:$SG74608 ; 'yes, that is our exception'
    call   _printf
    add    esp, 4
    mov    eax, 1
    jmp    SHORT $L74605
$L74607:
    push   OFFSET FLAT:$SG74610 ; 'not our exception'
    call   _printf
    add    esp, 4
    xor    eax, eax
$L74605:
    pop    ebp
    ret    0
_filter_user_exceptions ENDP

; scope table

CONST    SEGMENT
```

```
$T74644   DD    0ffffffffH   ; previous try level for outer ↙
    ↳ block
          DD    FLAT:$L74634 ; outer block filter
          DD    FLAT:$L74635 ; outer block handler
          DD    00H          ; previous try level for inner ↙
    ↳ block
          DD    FLAT:$L74638 ; inner block filter
          DD    FLAT:$L74639 ; inner block handler
CONST    ENDS

$T74643 = -36     ; size = 4
$T74642 = -32     ; size = 4
_p$ = -28         ; size = 4
__$SEHRec$ = -24  ; size = 24
_main    PROC NEAR
    push   ebp
    mov    ebp, esp
    push   -1 ; previous try level
    push   OFFSET FLAT:$T74644
    push   OFFSET FLAT:__except_handler3
    mov    eax, DWORD PTR fs:__except_list
    push   eax
    mov    DWORD PTR fs:__except_list, esp
    add    esp, -20
    push   ebx
    push   esi
    push   edi
    mov    DWORD PTR __$SEHRec$[ebp], esp
    mov    DWORD PTR _p$[ebp], 0
    mov    DWORD PTR __$SEHRec$[ebp+20], 0 ; outer try block ↙
    ↳ entered. set previous try level to 0
    mov    DWORD PTR __$SEHRec$[ebp+20], 1 ; inner try block ↙
    ↳ entered. set previous try level to 1
    push   OFFSET FLAT:$SG74617 ; 'hello!'
    call   _printf
    add    esp, 4
    push   0
    push   0
    push   0
    push   1122867    ; 00112233H
    call   DWORD PTR __imp__RaiseException@16
    push   OFFSET FLAT:$SG74619 ; '0x112233 raised. now let''s ↙
    ↳ crash'
```

```
    call   _printf
    add    esp, 4
    mov    eax, DWORD PTR _p$[ebp]
    mov    DWORD PTR [eax], 13
    mov    DWORD PTR __$SEHRec$[ebp+20], 0 ; inner try block ↙
    ↳ exited. set previous try level back to 0
    jmp    SHORT $L74615

    ; inner block filter

$L74638:
$L74650:
    mov    ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov    edx, DWORD PTR [ecx]
    mov    eax, DWORD PTR [edx]
    mov    DWORD PTR $T74643[ebp], eax
    mov    eax, DWORD PTR $T74643[ebp]
    sub    eax, -1073741819; c0000005H
    neg    eax
    sbb    eax, eax
    inc    eax
$L74640:
$L74648:
    ret    0

    ; inner block handler

$L74639:
    mov    esp, DWORD PTR __$SEHRec$[ebp]
    push   OFFSET FLAT:$SG74621 ; 'access violation, can''t ↙
    ↳ recover'
    call   _printf
    add    esp, 4
    mov    DWORD PTR __$SEHRec$[ebp+20], 0 ; inner try block ↙
    ↳ exited. set previous try level back to 0

$L74615:
    mov    DWORD PTR __$SEHRec$[ebp+20], -1 ; outer try block ↙
    ↳ exited, set previous try level back to -1
    jmp    SHORT $L74633

    ; outer block filter
```

```
$L74634:
$L74651:
    mov    ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov    edx, DWORD PTR [ecx]
    mov    eax, DWORD PTR [edx]
    mov    DWORD PTR $T74642[ebp], eax
    mov    ecx, DWORD PTR __$SEHRec$[ebp+4]
    push   ecx
    mov    edx, DWORD PTR $T74642[ebp]
    push   edx
    call   _filter_user_exceptions
    add    esp, 8
$L74636:
$L74649:
    ret    0

    ; outer block handler

$L74635:
    mov    esp, DWORD PTR __$SEHRec$[ebp]
    push   OFFSET FLAT:$SG74623 ; 'user exception caught'
    call   _printf
    add    esp, 4
    mov    DWORD PTR __$SEHRec$[ebp+20], -1 ; both try blocks ↙
    ↳ exited. set previous try level back to -1
$L74633:
    xor    eax, eax
    mov    ecx, DWORD PTR __$SEHRec$[ebp+8]
    mov    DWORD PTR fs:__except_list, ecx
    pop    edi
    pop    esi
    pop    ebx
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP
```

If to set a breakpoint on `printf()` function which is called from the handler, we may also see how yet another SEH handler is added. Perhaps, yet another machinery inside of SEH handling process. Here we also see our *scope table* consisting of 2 entries.

```
tracer.exe -l:3.exe bpx=3.exe!printf --dump-seh
```

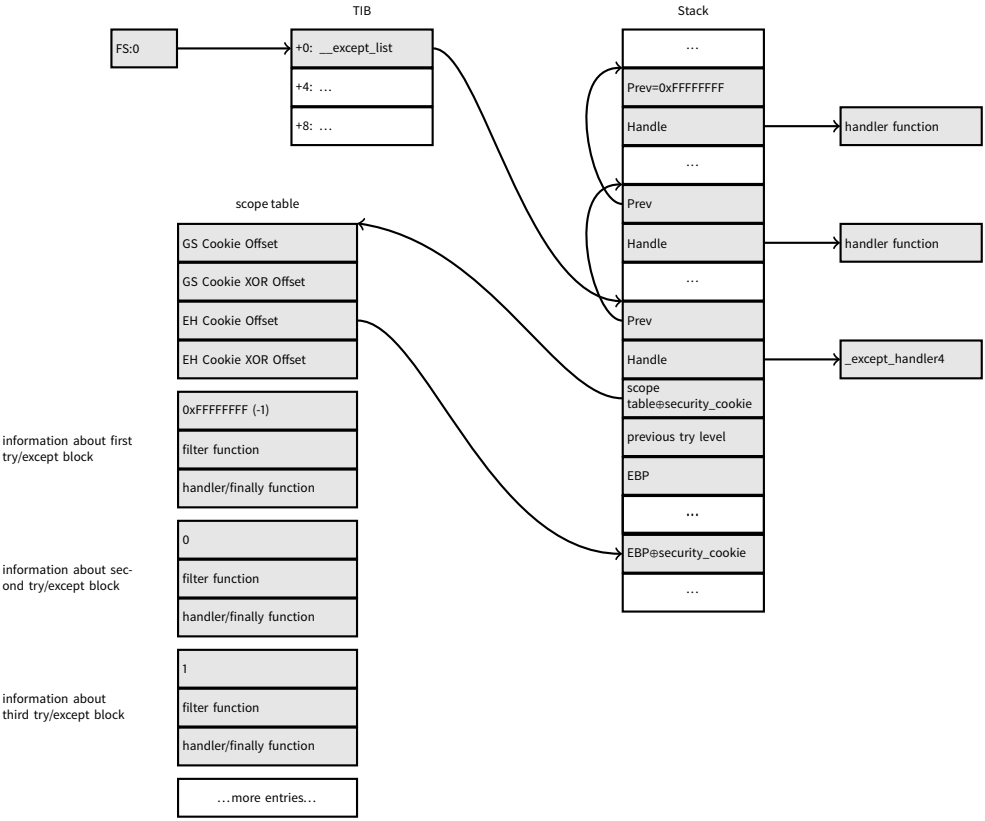
Listing 48.9: tracer.exe output

```
(0) 3.exe!printf
EAX=0x0000001b EBX=0x00000000 ECX=0x0040cc58 EDX=0x0008e3c8
ESI=0x00000000 EDI=0x00000000 EBP=0x0018f840 ESP=0x0018f838
EIP=0x004011b6
FLAGS=PF ZF IF
* SEH frame at 0x18f88c prev=0x18fe9c handler=0x771db4ad (ntdll.↙
    ↳ dll!ExecuteHandler2@20+0x3a)
* SEH frame at 0x18fe9c prev=0x18ff78 handler=0x4012e0 (3.exe!↙
    ↳ _except_handler3)
SEH3 frame. previous trylevel=1
scopetable entry[0]. previous try level=-1, filter=0x401120 (3.↙
    ↳ exe!main+0xb0) handler=0x40113b (3.exe!main+0xcb)
scopetable entry[1]. previous try level=0, filter=0x4010e8 (3.↙
    ↳ exe!main+0x78) handler=0x401100 (3.exe!main+0x90)
* SEH frame at 0x18ff78 prev=0x18ffc4 handler=0x4012e0 (3.exe!↙
    ↳ _except_handler3)
SEH3 frame. previous trylevel=0
scopetable entry[0]. previous try level=-1, filter=0x40160d (3.↙
    ↳ exe!mainCRTStartup+0x18d) handler=0x401621 (3.exe!↙
    ↳ mainCRTStartup+0x1a1)
* SEH frame at 0x18ffc4 prev=0x18ffe4 handler=0x771f71f5 (ntdll.↙
    ↳ dll!__except_handler4)
SEH4 frame. previous trylevel=0
SEH4 header:    GSCookieOffset=0xfffffffe GSCookieXOROffset=0x0
                EHCookieOffset=0xffffffcc EHCookieXOROffset=0x0
scopetable entry[0]. previous try level=-2, filter=0x771f74d0 (↙
    ↳ ntdll.dll!___safe_se_handler_table+0x20) handler=0↙
    ↳ x771f90eb (ntdll.dll!_TppTerminateProcess@4+0x43)
* SEH frame at 0x18ffe4 prev=0xffffffff handler=0x77247428 (↙
    ↳ ntdll.dll!_FinalExceptionHandler@16)
```

SEH4

During buffer overflow (16.2) attack, address of the *scope table* can be rewritten, so starting at MSVC 2005, SEH3 was upgraded to SEH4 in order to have buffer overflow protection. The pointer to *scope table* is now xored with security cookie. *Scope table* extended to have a header, consisting of two pointers to *security cookies*. Each element have an offset inside of stack of another value: this is address of stack frame (EBP) xored with `security_cookie` as well, placed in the stack. This value will be read during exception handling and checked, if it is correct. *Security cookie* in the stack is random each time, so remote attacker, hopefully, will not be able to predict it.

Initial *previous try level* is -2 in SEH4 instead of -1.



Here is both examples compiled in MSVC 2012 with SEH4:

Listing 48.10: MSVC 2012: one try block example

```
$SG85485 DB    'hello #1!', 0aH, 00H
$SG85486 DB    'hello #2!', 0aH, 00H
$SG85488 DB    'access violation, can''t recover', 0aH, 00H

; scope table

xdata$x         SEGMENT
__sehtable$_main DD 0fffffffeH   ; GS Cookie Offset
    DD          00H              ; GS Cookie XOR Offset
    DD          0fffffffccH      ; EH Cookie Offset
    DD          00H              ; EH Cookie XOR Offset
```

```
    DD          0ffffffeH      ; previous try level
    DD          FLAT:$LN12@main ; filter
    DD          FLAT:$LN8@main  ; handler
xdata$x         ENDS

$T2 = -36       ; size = 4
_p$ = -32       ; size = 4
tv68 = -28      ; size = 4
__$SEHRec$ = -24 ; size = 24
_main    PROC
    push   ebp
    mov    ebp, esp
    push   -2
    push   OFFSET __sehtable$_main
    push   OFFSET __except_handler4
    mov    eax, DWORD PTR fs:0
    push   eax
    add    esp, -20
    push   ebx
    push   esi
    push   edi
    mov    eax, DWORD PTR ___security_cookie
    xor    DWORD PTR __$SEHRec$[ebp+16], eax ; xored pointer to ↙
    ↳ scope table
    xor    eax, ebp
    push   eax                               ; ebp ^ ↙
    ↳ security_cookie
    lea    eax, DWORD PTR __$SEHRec$[ebp+8] ; pointer to ↙
    ↳ VC_EXCEPTION_REGISTRATION_RECORD
    mov    DWORD PTR fs:0, eax
    mov    DWORD PTR __$SEHRec$[ebp], esp
    mov    DWORD PTR _p$[ebp], 0
    mov    DWORD PTR __$SEHRec$[ebp+20], 0 ; previous try level
    push   OFFSET $SG85485 ; 'hello #1!'
    call   _printf
    add    esp, 4
    mov    eax, DWORD PTR _p$[ebp]
    mov    DWORD PTR [eax], 13
    push   OFFSET $SG85486 ; 'hello #2!'
    call   _printf
    add    esp, 4
    mov    DWORD PTR __$SEHRec$[ebp+20], -2 ; previous try level
    jmp    SHORT $LN6@main
```

```
    ; filter

$LN7@main:
$LN12@main:
    mov    ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov    edx, DWORD PTR [ecx]
    mov    eax, DWORD PTR [edx]
    mov    DWORD PTR $T2[ebp], eax
    cmp    DWORD PTR $T2[ebp], -1073741819 ; c0000005H
    jne    SHORT $LN4@main
    mov    DWORD PTR tv68[ebp], 1
    jmp    SHORT $LN5@main
$LN4@main:
    mov    DWORD PTR tv68[ebp], 0
$LN5@main:
    mov    eax, DWORD PTR tv68[ebp]
$LN9@main:
$LN11@main:
    ret    0

    ; handler

$LN8@main:
    mov    esp, DWORD PTR __$SEHRec$[ebp]
    push   OFFSET $SG85488 ; 'access violation, can''t recover'
    call   _printf
    add    esp, 4
    mov    DWORD PTR __$SEHRec$[ebp+20], -2 ; previous try level
$LN6@main:
    xor    eax, eax
    mov    ecx, DWORD PTR __$SEHRec$[ebp+8]
    mov    DWORD PTR fs:0, ecx
    pop    ecx
    pop    edi
    pop    esi
    pop    ebx
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP
```

Listing 48.11: MSVC 2012: two try blocks example

```
$SG85486 DB    'in filter. code=0x%08X', 0aH, 00H
$SG85488 DB    'yes, that is our exception', 0aH, 00H
$SG85490 DB    'not our exception', 0aH, 00H
$SG85497 DB    'hello!', 0aH, 00H
$SG85499 DB    '0x112233 raised. now let''s crash', 0aH, 00H
$SG85501 DB    'access violation, can''t recover', 0aH, 00H
$SG85503 DB    'user exception caught', 0aH, 00H

xdata$x    SEGMENT
__sehtable$_main DD 0ffffffeH         ; GS Cookie Offset
                 DD    00H            ; GS Cookie XOR Offset
                 DD    0fffffc8H      ; EH Cookie Offset
                 DD    00H            ; EH Cookie Offset
                 DD    0fffffffeH     ; previous try level for ↙
    ↳ outer block
                 DD    FLAT:$LN19@main ; outer block filter
                 DD    FLAT:$LN9@main  ; outer block handler
                 DD    00H            ; previous try level for ↙
    ↳ inner block
                 DD    FLAT:$LN18@main ; inner block filter
                 DD    FLAT:$LN13@main ; inner block handler
xdata$x    ENDS

$T2 = -40        ; size = 4
$T3 = -36        ; size = 4
_p$ = -32        ; size = 4
tv72 = -28       ; size = 4
__$SEHRec$ = -24 ; size = 24
_main    PROC
    push   ebp
    mov    ebp, esp
    push   -2  ; initial previous try level
    push   OFFSET __sehtable$_main
    push   OFFSET __except_handler4
    mov    eax, DWORD PTR fs:0
    push   eax ; prev
    add    esp, -24
    push   ebx
    push   esi
    push   edi
    mov    eax, DWORD PTR ___security_cookie
    xor    DWORD PTR __$SEHRec$[ebp+16], eax       ; xored ↙
    ↳ pointer to scope table
```

```
    xor     eax, ebp                                        ; ebp ^ ↙
    ↳ security_cookie
    push    eax
    lea     eax, DWORD PTR __$SEHRec$[ebp+8]        ; pointer to ↙
    ↳ VC_EXCEPTION_REGISTRATION_RECORD
    mov     DWORD PTR fs:0, eax
    mov     DWORD PTR __$SEHRec$[ebp], esp
    mov     DWORD PTR _p$[ebp], 0
    mov     DWORD PTR __$SEHRec$[ebp+20], 0 ; entering outer try ↙
    ↳ block, setting previous try level=0
    mov     DWORD PTR __$SEHRec$[ebp+20], 1 ; entering inner try ↙
    ↳ block, setting previous try level=1
    push    OFFSET $SG85497 ; 'hello!'
    call    _printf
    add     esp, 4
    push    0
    push    0
    push    0
    push    1122867 ; 00112233H
    call    DWORD PTR __imp__RaiseException@16
    push    OFFSET $SG85499 ; '0x112233 raised. now let''s crash'
    call    _printf
    add     esp, 4
    mov     eax, DWORD PTR _p$[ebp]
    mov     DWORD PTR [eax], 13
    mov     DWORD PTR __$SEHRec$[ebp+20], 0 ; exiting inner try ↙
    ↳ block, set previous try level back to 0
    jmp     SHORT $LN2@main

    ; inner block filter

$LN12@main:
$LN18@main:
    mov     ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov     edx, DWORD PTR [ecx]
    mov     eax, DWORD PTR [edx]
    mov     DWORD PTR $T3[ebp], eax
    cmp     DWORD PTR $T3[ebp], -1073741819 ; c0000005H
    jne     SHORT $LN5@main
    mov     DWORD PTR tv72[ebp], 1
    jmp     SHORT $LN6@main
$LN5@main:
    mov     DWORD PTR tv72[ebp], 0
```

```
$LN6@main:
    mov    eax, DWORD PTR tv72[ebp]
$LN14@main:
$LN16@main:
    ret    0

    ; inner block handler

$LN13@main:
    mov    esp, DWORD PTR __$SEHRec$[ebp]
    push   OFFSET $SG85501 ; 'access violation, can''t recover'
    call   _printf
    add    esp, 4
    mov    DWORD PTR __$SEHRec$[ebp+20], 0 ; exiting inner try ↙
    ↳ block, setting previous try level back to 0
$LN2@main:
    mov    DWORD PTR __$SEHRec$[ebp+20], -2 ; exiting both ↙
    ↳ blocks, setting previous try level back to -2
    jmp    SHORT $LN7@main

    ; outer block filter

$LN8@main:
$LN19@main:
    mov    ecx, DWORD PTR __$SEHRec$[ebp+4]
    mov    edx, DWORD PTR [ecx]
    mov    eax, DWORD PTR [edx]
    mov    DWORD PTR $T2[ebp], eax
    mov    ecx, DWORD PTR __$SEHRec$[ebp+4]
    push   ecx
    mov    edx, DWORD PTR $T2[ebp]
    push   edx
    call   _filter_user_exceptions
    add    esp, 8
$LN10@main:
$LN17@main:
    ret    0

    ; outer block handler

$LN9@main:
    mov    esp, DWORD PTR __$SEHRec$[ebp]
    push   OFFSET $SG85503 ; 'user exception caught'
```

```
    call   _printf
    add    esp, 4
    mov    DWORD PTR __$SEHRec$[ebp+20], -2 ; exiting both ↙
    ↳ blocks, setting previous try level back to -2
$LN7@main:
    xor    eax, eax
    mov    ecx, DWORD PTR __$SEHRec$[ebp+8]
    mov    DWORD PTR fs:0, ecx
    pop    ecx
    pop    edi
    pop    esi
    pop    ebx
    mov    esp, ebp
    pop    ebp
    ret    0
_main    ENDP

_code$ = 8  ; size = 4
_ep$ = 12   ; size = 4
_filter_user_exceptions PROC
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _code$[ebp]
    push   eax
    push   OFFSET $SG85486 ; 'in filter. code=0x%08X'
    call   _printf
    add    esp, 8
    cmp    DWORD PTR _code$[ebp], 1122867 ; 00112233H
    jne    SHORT $LN2@filter_use
    push   OFFSET $SG85488 ; 'yes, that is our exception'
    call   _printf
    add    esp, 4
    mov    eax, 1
    jmp    SHORT $LN3@filter_use
    jmp    SHORT $LN3@filter_use
$LN2@filter_use:
    push   OFFSET $SG85490 ; 'not our exception'
    call   _printf
    add    esp, 4
    xor    eax, eax
$LN3@filter_use:
    pop    ebp
    ret    0
```

```
_filter_user_exceptions ENDP
```

Here is a meaning of *cookies*: `Cookie Offset` is a difference between address of saved EBP value in stack and the $EBP \oplus security_cookie$ value in the stack. `Cookie XOR Offset` is additional difference between $EBP \oplus security_cookie$ value and what is stored in the stack. If this equation is not true, a process will be stopped due to stack corruption:

$$security_cookie \oplus (CookieXOROffset + addressofsavedEBP) == stack[addressofsavedEBP + CookieOffset]$$

If `Cookie Offset` is -2, it is not present.

Cookies checking is also implemented in my tracer, see https://github.com/dennis714/tracer/blob/master/SEH.c for details.

It is still possible to fall back to SEH3 in the compilers after (and including) MSVC 2005 by setting `/GS-` option, however, CRT code will use SEH4 anyway.

48.3.3 Windows x64

As you might think, it is not very fast thing to set up SEH frame at each function prologue. Another performance problem is to change *previous try level* value many times while function execution. So things are changed completely in x64: now all pointers to `try` blocks, filter and handler functions are stored in another PE-segment `.pdata`, that is where OS exception handler takes all the information.

These are two examples from the previous section compiled for x64:

Listing 48.12: MSVC 2012

```
$SG86276 DB     'hello #1!', 0aH, 00H
$SG86277 DB     'hello #2!', 0aH, 00H
$SG86279 DB     'access violation, can''t recover', 0aH, 00H

pdata   SEGMENT
$pdata$main DD  imagerel $LN9
        DD      imagerel $LN9+61
        DD      imagerel $unwind$main
pdata   ENDS
pdata   SEGMENT
$pdata$main$filt$0 DD imagerel main$filt$0
        DD      imagerel main$filt$0+32
        DD      imagerel $unwind$main$filt$0
pdata   ENDS
xdata   SEGMENT
$unwind$main DD 020609H
```

```
        DD      030023206H
        DD      imagerel __C_specific_handler
        DD      01H
        DD      imagerel $LN9+8
        DD      imagerel $LN9+40
        DD      imagerel main$filt$0
        DD      imagerel $LN9+40
$unwind$main$filt$0 DD 020601H
        DD      050023206H
xdata   ENDS

_TEXT   SEGMENT
main    PROC
$LN9:
        push    rbx
        sub     rsp, 32
        xor     ebx, ebx
        lea     rcx, OFFSET FLAT:$SG86276 ; 'hello #1!'
        call    printf
        mov     DWORD PTR [rbx], 13
        lea     rcx, OFFSET FLAT:$SG86277 ; 'hello #2!'
        call    printf
        jmp     SHORT $LN8@main
$LN6@main:
        lea     rcx, OFFSET FLAT:$SG86279 ; 'access violation, ↙
    ↳ can''t recover'
        call    printf
        npad    1
$LN8@main:
        xor     eax, eax
        add     rsp, 32
        pop     rbx
        ret     0
main    ENDP
_TEXT   ENDS

text$x  SEGMENT
main$filt$0 PROC
        push    rbp
        sub     rsp, 32
        mov     rbp, rdx
$LN5@main$filt$:
        mov     rax, QWORD PTR [rcx]
```

```
        xor     ecx, ecx
        cmp     DWORD PTR [rax], -1073741819; c0000005H
        sete    cl
        mov     eax, ecx
$LN7@main$filt$:
        add     rsp, 32
        pop     rbp
        ret     0
        int     3
main$filt$0 ENDP
text$x  ENDS
```

Listing 48.13: MSVC 2012

```
$SG86277 DB     'in filter. code=0x%08X', 0aH, 00H
$SG86279 DB     'yes, that is our exception', 0aH, 00H
$SG86281 DB     'not our exception', 0aH, 00H
$SG86288 DB     'hello!', 0aH, 00H
$SG86290 DB     '0x112233 raised. now let''s crash', 0aH, 00H
$SG86292 DB     'access violation, can''t recover', 0aH, 00H
$SG86294 DB     'user exception caught', 0aH, 00H

pdata   SEGMENT
$pdata$filter_user_exceptions DD imagerel $LN6
        DD      imagerel $LN6+73
        DD      imagerel $unwind$filter_user_exceptions
$pdata$main DD  imagerel $LN14
        DD      imagerel $LN14+95
        DD      imagerel $unwind$main
pdata   ENDS
pdata   SEGMENT
$pdata$main$filt$0 DD imagerel main$filt$0
        DD      imagerel main$filt$0+32
        DD      imagerel $unwind$main$filt$0
$pdata$main$filt$1 DD imagerel main$filt$1
        DD      imagerel main$filt$1+30
        DD      imagerel $unwind$main$filt$1
pdata   ENDS

xdata   SEGMENT
$unwind$filter_user_exceptions DD 020601H
        DD      030023206H
$unwind$main DD 020609H
        DD      030023206H
```

```
        DD      imagerel __C_specific_handler
        DD      02H
        DD      imagerel $LN14+8
        DD      imagerel $LN14+59
        DD      imagerel main$filt$0
        DD      imagerel $LN14+59
        DD      imagerel $LN14+8
        DD      imagerel $LN14+74
        DD      imagerel main$filt$1
        DD      imagerel $LN14+74
$unwind$main$filt$0 DD 020601H
        DD      050023206H
$unwind$main$filt$1 DD 020601H
        DD      050023206H
xdata   ENDS

_TEXT   SEGMENT
main    PROC
$LN14:
        push    rbx
        sub     rsp, 32
        xor     ebx, ebx
        lea     rcx, OFFSET FLAT:$SG86288 ; 'hello!'
        call    printf
        xor     r9d, r9d
        xor     r8d, r8d
        xor     edx, edx
        mov     ecx, 1122867 ; 00112233H
        call    QWORD PTR __imp_RaiseException
        lea     rcx, OFFSET FLAT:$SG86290 ; '0x112233 raised. ↙
    ↳ now let''s crash'
        call    printf
        mov     DWORD PTR [rbx], 13
        jmp     SHORT $LN13@main
$LN11@main:
        lea     rcx, OFFSET FLAT:$SG86292 ; 'access violation, ↙
    ↳ can''t recover'
        call    printf
        npad    1
$LN13@main:
        jmp     SHORT $LN9@main
$LN7@main:
        lea     rcx, OFFSET FLAT:$SG86294 ; 'user exception ↙
```

```
    ↳ caught'
        call    printf
        npad    1
$LN9@main:
        xor     eax, eax
        add     rsp, 32
        pop     rbx
        ret     0
main    ENDP

text$x  SEGMENT
main$filt$0 PROC
        push    rbp
        sub     rsp, 32
        mov     rbp, rdx
$LN10@main$filt$:
        mov     rax, QWORD PTR [rcx]
        xor     ecx, ecx
        cmp     DWORD PTR [rax], -1073741819; c0000005H
        sete    cl
        mov     eax, ecx
$LN12@main$filt$:
        add     rsp, 32
        pop     rbp
        ret     0
        int     3
main$filt$0 ENDP

main$filt$1 PROC
        push    rbp
        sub     rsp, 32
        mov     rbp, rdx
$LN6@main$filt$:
        mov     rax, QWORD PTR [rcx]
        mov     rdx, rcx
        mov     ecx, DWORD PTR [rax]
        call    filter_user_exceptions
        npad    1
$LN8@main$filt$:
        add     rsp, 32
        pop     rbp
        ret     0
        int     3
```

```
main$filt$1 ENDP
text$x  ENDS

_TEXT   SEGMENT
code$ = 48
ep$ = 56
filter_user_exceptions PROC
$LN6:
        push    rbx
        sub     rsp, 32
        mov     ebx, ecx
        mov     edx, ecx
        lea     rcx, OFFSET FLAT:$SG86277 ; 'in filter. code=0x↙
    ↳ %08X'
        call    printf
        cmp     ebx, 1122867; 00112233H
        jne     SHORT $LN2@filter_use
        lea     rcx, OFFSET FLAT:$SG86279 ; 'yes, that is our ↙
    ↳ exception'
        call    printf
        mov     eax, 1
        add     rsp, 32
        pop     rbx
        ret     0
$LN2@filter_use:
        lea     rcx, OFFSET FLAT:$SG86281 ; 'not our exception'
        call    printf
        xor     eax, eax
        add     rsp, 32
        pop     rbx
        ret     0
filter_user_exceptions ENDP
_TEXT   ENDS
```

Read [32] for more detailed information about this.

Aside from exception information, `.pdata` is a section containing addresses of almost all function starts and ends, hence it may be useful for a tools targetting automated analysis.

48.3.4 Read more about SEH

[23], [32].

48.4 Windows NT: Critical section

Critical sections in any OS are very important in multithreaded environment, mostly used for issuing a guarantee that only one thread will access some data, while blocking other threads and interrupts.

That is how CRITICAL_SECTION structure is declared in Windows NT line OS:

Listing 48.14: (Windows Research Kernel v1.2) public/sdk/inc/nturtl.h

```
typedef struct _RTL_CRITICAL_SECTION {
    PRTL_CRITICAL_SECTION_DEBUG DebugInfo;

    //
    //  The following three fields control entering and exiting ↙
    ↘ the critical
    //  section for the resource
    //

    LONG LockCount;
    LONG RecursionCount;
    HANDLE OwningThread;        // from the thread's ClientId->↙
    ↘ UniqueThread
    HANDLE LockSemaphore;
    ULONG_PTR SpinCount;        // force size on 64-bit systems ↙
    ↘ when packed
} RTL_CRITICAL_SECTION, *PRTL_CRITICAL_SECTION;
```

That's is how EnterCriticalSection() function works:

Listing 48.15: Windows 2008/ntdll.dll/x86 (begin)

```
_RtlEnterCriticalSection@4

var_C           = dword ptr -0Ch
var_8           = dword ptr -8
var_4           = dword ptr -4
arg_0           = dword ptr  8

                mov     edi, edi
                push    ebp
                mov     ebp, esp
                sub     esp, 0Ch
                push    esi
                push    edi
                mov     edi, [ebp+arg_0]
```

```
                lea     esi, [edi+4] ; LockCount
                mov     eax, esi
                lock btr dword ptr [eax], 0
                jnb     wait ; jump if CF=0

loc_7DE922DD:
                mov     eax, large fs:18h
                mov     ecx, [eax+24h]
                mov     [edi+0Ch], ecx
                mov     dword ptr [edi+8], 1
                pop     edi
                xor     eax, eax
                pop     esi
                mov     esp, ebp
                pop     ebp
                retn    4

... skipped
```

The most important instruction in this code fragment is BTR (prefixed with LOCK): the zeroth bit is stored in CF flag and cleared in memory. This is atomic operation, blocking all other CPUs to access this piece of memory (take a notice of LOCK prefix before BTR instruction). If the bit at LockCount was 1, fine, reset it and return from the function: we are in critical section. If not —critical section is already occupied by other thread, then wait.
Wait is done there using WaitForSingleObject().

And here is how LeaveCriticalSection() function works:

Listing 48.16: Windows 2008/ntdll.dll/x86 (begin)

```
_RtlLeaveCriticalSection@4 proc near

arg_0           = dword ptr  8

                mov     edi, edi
                push    ebp
                mov     ebp, esp
                push    esi
                mov     esi, [ebp+arg_0]
                add     dword ptr [esi+8], 0FFFFFFFFh ; ↙
    ↳ RecursionCount
                jnz     short loc_7DE922B2
                push    ebx
                push    edi
```

```
                lea     edi, [esi+4]    ; LockCount
                mov     dword ptr [esi+0Ch], 0
                mov     ebx, 1
                mov     eax, edi
                lock xadd [eax], ebx
                inc     ebx
                cmp     ebx, 0FFFFFFFFh
                jnz     loc_7DEA8EB7

loc_7DE922B0:
                pop     edi
                pop     ebx

loc_7DE922B2:
                xor     eax, eax
                pop     esi
                pop     ebp
                retn    4

... skipped
```

XADD is "exchange and add". In this case, it summing LockCount value and 1 and stores result in EBX register, and at the same time 1 goes to LockCount. This operation is atomic since it is prefixed by LOCK as well, meaning that all other CPUs or CPU cores in system are blocked from accessing this point of memory.

LOCK prefix is very important: two threads, each of which working on separate CPUs or CPU cores may try to enter critical section and to modify the value in memory simultaneously, this will result in unpredictable behaviour.

Part V

Tools

Chapter 49

Disassembler

49.1 IDA

Older freeware version is available for downloading [1].

Short hot-keys cheatsheet: F.1

[1] http://www.hex-rays.com/idapro/idadownfreeware.htm

Chapter 50

Debugger

50.1 tracer

I use *tracer*[1] instead of debugger.

I stopped to use debugger eventually, since all I need from it is to spot a function's arguments while execution, or registers' state at some point. To load debugger each time is too much, so I wrote a small utility *tracer*. It has console-interface, working from command-line, enable us to intercept function execution, set breakpoints at arbitrary places, spot registers' state, modify it, etc.

However, as for learning purposes, it is highly advisable to trace code in debugger manually, watch how register's state changing (e.g. classic SoftICE, OllyDbg, WinDbg highlighting changed registers), flags, data, change them manually, watch reaction, etc.

50.2 OllyDbg

Very popular user-mode win32 debugger:
http://www.ollydbg.de/.

Short hot-keys cheatsheet:

50.3 GDB

Not very popular debugger among reverse engineers, but very comfortable nevertheless. Some commands: F.5.

[1] http://yurichev.com/tracer-en.html

Chapter 51

System calls tracing

51.0.1 strace / dtruss

Will show which system calls (syscalls(46)) are called by process right now. For example:

```
# strace df -h

...

access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such ↙
    ↳ file or directory)
open("/lib/i386-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF↙
    ↳ \1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\220\232\1\0004\0\0\0"...,↙
    ↳  512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1770984, ...}) = 0
mmap2(NULL, 1780508, PROT_READ|PROT_EXEC, MAP_PRIVATE|↙
    ↳ MAP_DENYWRITE, 3, 0) = 0xb75b3000
```

Mac OS X has dtruss for the same aim.

The Cygwin also has strace, but if I understood correctly, it works only for .exe-files compiled for cygwin environment itself.

Chapter 52

Decompilers

There are only one known, publically available, high-quality decompiler to C code: Hex-Rays:
https://www.hex-rays.com/products/decompiler/

Chapter 53

Other tools

- Microsoft Visual Studio Express[1]: Stripped-down free Visual Studio version, convenient for simple experiments. Some useful options: F.3.
- Hiew[2] for small modifications of code in binary files.
- binary grep: the small utility for constants searching (or just any byte sequence) in a big pile of files, including non-executable: https://github.com/yurichev/bgrep.

[1] http://www.microsoft.com/express/Downloads/
[2] http://www.hiew.ru/

Part VI

More examples

Chapter 54

Hand decompiling + using Z3 SMT solver for defeating amateur cryptography

Amateur cryptography is usually (unintentionally) very weak and can be breaked easily—for cryptographers, of course.

But let's pretend we are not among these crypto-professionals.

I once found this one-way hash function, converting 64-bit value to another one and we need to try to reverse its flow back.

> But what is hash-function? Simplest example is CRC32, an algorithm providing "stronger" checksum for integrity checking purposes. it is impossible to restore original text from the hash value, it just has much less information: there can be long text, but CRC32 result is always limited to 32 bits. But CRC32 is not cryptographically secure: it is known how to alter a text in that way so the resulting CRC32 hash value will be one we need. Cryptographical hash functions are protected from this. They are widely used to hash user passwords in order to store them in the database, like MD5, SHA1, etc. Indeed: an internet forum database may not contain user passwords (stolen database will compromise all user's passwords) but only hashes (a cracker will not be able to reveal passwords). Besides, an internet forum engine is not aware of your password, it should only check if its hash is the same as in the database, then it will give you access in this case. One of the simplest passwords cracking methods is just to brute-force all passwords in order to wait when resulting value will be the same as we need. Other methods are much more complex.

54.1 Hand decompiling

Here its assembly language listing in IDA:

```
sub_401510      proc near
                ; ECX = input
                mov     rdx, 5D7E0D1F2E0F1F84h
                mov     rax, rcx        ; input
                imul    rax, rdx
                mov     rdx, 388D76AEE8CB1500h
                mov     ecx, eax
                and     ecx, 0Fh
                ror     rax, cl
                xor     rax, rdx
                mov     rdx, 0D2E9EE7E83C4285Bh
                mov     ecx, eax
                and     ecx, 0Fh
                rol     rax, cl
                lea     r8, [rax+rdx]
                mov     rdx, 8888888888888889h
                mov     rax, r8
                mul     rdx
                shr     rdx, 5
                mov     rax, rdx
                lea     rcx, [r8+rdx*4]
                shl     rax, 6
                sub     rcx, rax
                mov     rax, r8
                rol     rax, cl
                ; EAX = output
                retn
sub_401510      endp
```

The example was compiled by GCC, so the first argument is passed in `ECX`.

If Hex-Rays is not in list of our posessions, or we distrust to it, we may try to reverse this code manually. One method is to represent CPU registers as local C variables and replace each instruction by one-line equivalent expression, like:

```
uint64_t f(uint64_t input)
{
        uint64_t rax, rbx, rcx, rdx, r8;

        ecx=input;

        rdx=0x5D7E0D1F2E0F1F84;
```

```
        rax=rcx;
        rax*=rdx;
        rdx=0x388D76AEE8CB1500;
        rax=_lrotr(rax, rax&0xF); // rotate right
        rax^=rdx;
        rdx=0xD2E9EE7E83C4285B;
        rax=_lrotl(rax, rax&0xF); // rotate left
        r8=rax+rdx;
        rdx=0x8888888888888889;
        rax=r8;
        rax*=rdx;
        rdx=rdx>>5;
        rax=rdx;
        rcx=r8+rdx*4;
        rax=rax<<6;
        rcx=rcx-rax;
        rax=r8
        rax=_lrotl (rax, rcx&0xFF); // rotate left
        return rax;
};
```

If to be careful enough, this code can be compiled and will even work in the same way as original one.

Then, we will rewrite it gradually, keeping in mind all registers usage. Attention and focusing is very important here—any tiny typo may ruin all your work!

Here is a first step:

```
uint64_t f(uint64_t input)
{
        uint64_t rax, rbx, rcx, rdx, r8;

        ecx=input;

        rdx=0x5D7E0D1F2E0F1F84;
        rax=rcx;
        rax*=rdx;
        rdx=0x388D76AEE8CB1500;
        rax=_lrotr(rax, rax&0xF); // rotate right
        rax^=rdx;
        rdx=0xD2E9EE7E83C4285B;
        rax=_lrotl(rax, rax&0xF); // rotate left
        r8=rax+rdx;

        rdx=0x8888888888888889;
```

```
        rax=r8;
        rax*=rdx;
        // RDX here is a high part of multiplication result
        rdx=rdx>>5;
        // RDX here is division result!
        rax=rdx;

        rcx=r8+rdx*4;
        rax=rax<<6;
        rcx=rcx-rax;
        rax=r8
        rax=_lrotl (rax, rcx&0xFF); // rotate left
        return rax;
};
```

Next step:

```
uint64_t f(uint64_t input)
{
        uint64_t rax, rbx, rcx, rdx, r8;

        ecx=input;

        rdx=0x5D7E0D1F2E0F1F84;
        rax=rcx;
        rax*=rdx;
        rdx=0x388D76AEE8CB1500;
        rax=_lrotr(rax, rax&0xF); // rotate right
        rax^=rdx;
        rdx=0xD2E9EE7E83C4285B;
        rax=_lrotl(rax, rax&0xF); // rotate left
        r8=rax+rdx;

        rdx=0x8888888888888889;
        rax=r8;
        rax*=rdx;
        // RDX here is a high part of multiplication result
        rdx=rdx>>5;
        // RDX here is division result!
        rax=rdx;

        rcx=(r8+rdx*4)-(rax<<6);
        rax=r8
        rax=_lrotl (rax, rcx&0xFF); // rotate left
```

```
        return rax;
};
```

We may spot division using multiplication (14). Indeed, let's calculate divider in Wolfram Mathematica:

Listing 54.1: Wolfram Mathematica

```
In[1]:=N[2^(64 + 5)/16^^8888888888888889]
Out[1]:=60.
```

We get this:

```
uint64_t f(uint64_t input)
{
        uint64_t rax, rbx, rcx, rdx, r8;

        ecx=input;

        rdx=0x5D7E0D1F2E0F1F84;
        rax=rcx;
        rax*=rdx;
        rdx=0x388D76AEE8CB1500;
        rax=_lrotr(rax, rax&0xF); // rotate right
        rax^=rdx;
        rdx=0xD2E9EE7E83C4285B;
        rax=_lrotl(rax, rax&0xF); // rotate left
        r8=rax+rdx;

        rax=rdx=r8/60;

        rcx=(r8+rax*4)-(rax*64);
        rax=r8
        rax=_lrotl (rax, rcx&0xFF); // rotate left
        return rax;
};
```

Another step:

```
uint64_t f(uint64_t input)
{
        uint64_t rax, rbx, rcx, rdx, r8;

        rax=input;
        rax*=0x5D7E0D1F2E0F1F84;
        rax=_lrotr(rax, rax&0xF); // rotate right
```

```
        rax^=0x388D76AEE8CB1500;
        rax=_lrotl(rax, rax&0xF); // rotate left
        r8=rax+0xD2E9EE7E83C4285B;

        rcx=r8-(r8/60)*60;
        rax=r8
        rax=_lrotl (rax, rcx&0xFF); // rotate left
        return rax;
};
```

By simple reducing, we finally see that it's not quotient calculated, but division remainder:

```
uint64_t f(uint64_t input)
{
        uint64_t rax, rbx, rcx, rdx, r8;

        rax=input;
        rax*=0x5D7E0D1F2E0F1F84;
        rax=_lrotr(rax, rax&0xF); // rotate right
        rax^=0x388D76AEE8CB1500;
        rax=_lrotl(rax, rax&0xF); // rotate left
        r8=rax+0xD2E9EE7E83C4285B;

        return _lrotl (r8, r8 % 60); // rotate left
};
```

We end up on something fancy formatted source-code:

```
#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>
#include <string.h>
#include <intrin.h>

#define C1 0x5D7E0D1F2E0F1F84
#define C2 0x388D76AEE8CB1500
#define C3 0xD2E9EE7E83C4285B

uint64_t hash(uint64_t v)
{
        v*=C1;
        v=_lrotr(v, v&0xF); // rotate right
        v^=C2;
        v=_lrotl(v, v&0xF); // rotate left
```

```
        v+=C3;
        v=_lrotl(v, v % 60); // rotate left
        return v;
};

int main()
{
        printf ("%llu\n", hash(...));
};
```

Since we are not cryptoanalysts we can't find an easy way to generate input value for some specific output value. Rotate instruction coefficients are look frightening—it's a warranty that the function is not bijective, it has collisions, or, speaking more simply, many inputs may be possible for one output.

Brute-force is not solution because values are 64-bit ones, that's beyond reality.

54.2 Now let's use Z3 SMT solver

Still, without any special cryptographical knowledge, we may try to break this algorithm using excellent SMT solver from Microsoft Research named Z3[1]. It is in fact theorem prover, but we will use it as SMT solver. In terms of simplicity, we may think about it as a system capable of solving huge equation systems.

Here is a Python source code:

```
1  from z3 import *
2
3  C1=0x5D7E0D1F2E0F1F84
4  C2=0x388D76AEE8CB1500
5  C3=0xD2E9EE7E83C4285B
6
7  inp, i1, i2, i3, i4, i5, i6, outp = BitVecs('inp i1 i2 i3 i4 i5 ↙
       ↳ i6 outp', 64)
8
9  s = Solver()
10 s.add(i1==inp*C1)
11 s.add(i2==RotateRight (i1, i1 & 0xF))
12 s.add(i3==i2 ^ C2)
13 s.add(i4==RotateLeft(i3, i3 & 0xF))
14 s.add(i5==i4 + C3)
15 s.add(outp==RotateLeft (i5, URem(i5, 60)))
16
17 s.add(outp==10816636949158156260)
```

[1] http://z3.codeplex.com/

```
18
19 print s.check()
20 m=s.model()
21 print m
22 print (" inp=0x%X" % m[inp].as_long())
23 print ("outp=0x%X" % m[outp].as_long())
```

This will be our first solver.

We see variable difinitions on line 7. These are just 64-bit variables. `i1..i6` are intermediate variables, representing values in registers between instruction executions.

Then we add so called constraints on lines 10..15. The very last constraint at 17 is most important: we will try to find input value for which our algorithm will produce 10816636949158156260.

Essentially, SMT-solver searches for (any) values that satisfy all constraints.

RotateRight, RotateLeft, URem—are functions from Z3 Python API, they are not related to Python PL.

Then we run it:

```
...>python.exe 1.py
sat
[i1 = 3959740824832824396,
 i3 = 8957124831728646493,
 i5 = 10816636949158156260,
 inp = 1364123924608584563,
 outp = 10816636949158156260,
 i4 = 14065440378185297801,
 i2 = 4954926323707358301]
 inp=0x12EE577B63E80B73
outp=0x961C69FF0AEFD7E4
```

"sat" mean "satisfiable", i.e., solver was able to found at least one solution. The solution is printed inside square brackets. Two last lines are input/output pair in hexadecimal form. Yes, indeed, if we run our function with `0x12EE577B63E80B73` on input, the algorithm will produce the value we were looking for.

But, as we are noticed before, the function we work with is not bijective, so there are may be other correct input values. Z3 SMT solver is not capable of producing more than one result, but let's hack our example slightly, by adding line 19, meaining, look for any other results than this:

```
1 from z3 import *
2
3 C1=0x5D7E0D1F2E0F1F84
4 C2=0x388D76AEE8CB1500
5 C3=0xD2E9EE7E83C4285B
```

```
6
7  inp, i1, i2, i3, i4, i5, i6, outp = BitVecs('inp i1 i2 i3 i4 i5 ↙
       ↳ i6 outp', 64)
8
9  s = Solver()
10 s.add(i1==inp*C1)
11 s.add(i2==RotateRight (i1, i1 & 0xF))
12 s.add(i3==i2 ^ C2)
13 s.add(i4==RotateLeft(i3, i3 & 0xF))
14 s.add(i5==i4 + C3)
15 s.add(outp==RotateLeft (i5, URem(i5, 60)))
16
17 s.add(outp==10816636949158156260)
18
19 s.add(inp!=0x12EE577B63E80B73)
20
21 print s.check()
22 m=s.model()
23 print m
24 print (" inp=0x%X" % m[inp].as_long())
25 print ("outp=0x%X" % m[outp].as_long())
```

Indeed, it found other correct result:

```
...>python.exe 2.py
sat
[i1 = 3959740824832824396,
 i3 = 8957124831728646493,
 i5 = 10816636949158156260,
 inp = 10587495961463360371,
 outp = 10816636949158156260,
 i4 = 14065440378185297801,
 i2 = 4954926323707358301]
 inp=0x92EE577B63E80B73
outp=0x961C69FF0AEFD7E4
```

This can be automated. Each found result may be added as constraint and the next result will be searched for. Here is slightly sophisticated example:

```
1  from z3 import *
2
3  C1=0x5D7E0D1F2E0F1F84
4  C2=0x388D76AEE8CB1500
5  C3=0xD2E9EE7E83C4285B
6
```

```
inp, i1, i2, i3, i4, i5, i6, outp = BitVecs('inp i1 i2 i3 i4 i5 i6 outp', 64)

s = Solver()
s.add(i1==inp*C1)
s.add(i2==RotateRight (i1, i1 & 0xF))
s.add(i3==i2 ^ C2)
s.add(i4==RotateLeft(i3, i3 & 0xF))
s.add(i5==i4 + C3)
s.add(outp==RotateLeft (i5, URem(i5, 60)))

s.add(outp==10816636949158156260)

# copypasted from http://stackoverflow.com/questions/11867611/z3py-checking-all-solutions-for-equation
result=[]
while True:
    if s.check() == sat:
        m = s.model()
        print m[inp]
        result.append(m)
        # Create a new constraint the blocks the current model
        block = []
        for d in m:
            # d is a declaration
            if d.arity() > 0:
                raise Z3Exception("uninterpreted functions are not suppported")
            # create a constant from declaration
            c=d()
            if is_array(c) or c.sort().kind() == Z3_UNINTERPRETED_SORT:
                raise Z3Exception("arrays and uninterpreted sorts are not supported")
            block.append(c != m[d])
        s.add(Or(block))
    else:
            print "results total=",len(result)
            break
```

We got:

```
1364123924608584563
1234567890
```

```
9223372038089343698
4611686019661955794
13835058056516731602
3096040143925676201
12319412180780452009
7707726162353064105
16931098199207839913
1906652839273745429
11130024876128521237
15741710894555909141
6518338857701133333
5975809943035972467
15199181979890748275
10587495961463360371
results total= 16
```

So there are 16 correct input values are possible for `0x92EE577B63E80B73` as a result.

The second is 1234567890—it is indeed a value I used originally while preparing this example.

Let's also try to research our algorithm more. By some sadistic purposes, let's find, are there any possible input/output pair in which lower 32-bit parts are equal to each other?

Let's remove *outp* constraint and add another, at line 17:

```
1  from z3 import *
2
3  C1=0x5D7E0D1F2E0F1F84
4  C2=0x388D76AEE8CB1500
5  C3=0xD2E9EE7E83C4285B
6
7  inp, i1, i2, i3, i4, i5, i6, outp = BitVecs('inp i1 i2 i3 i4 i5 ↙
       ↳ i6 outp', 64)
8
9  s = Solver()
10 s.add(i1==inp*C1)
11 s.add(i2==RotateRight (i1, i1 & 0xF))
12 s.add(i3==i2 ^ C2)
13 s.add(i4==RotateLeft(i3, i3 & 0xF))
14 s.add(i5==i4 + C3)
15 s.add(outp==RotateLeft (i5, URem(i5, 60)))
16
17 s.add(outp & 0xFFFFFFFF == inp & 0xFFFFFFFF)
18
```

```
print s.check()
m=s.model()
print m
print (" inp=0x%X" % m[inp].as_long())
print ("outp=0x%X" % m[outp].as_long())
```

It is indeed so:

```
sat
[i1 = 14869545517796235860,
 i3 = 8388171335828825253,
 i5 = 6918262285561543945,
 inp = 1370377541658871093,
 outp = 14543180351754208565,
 i4 = 10167065714588685486,
 i2 = 5541032613289652645]
 inp=0x13048F1D12C00535
outp=0xC9D3C17A12C00535
```

Let's be more sadistic and add another constaint: last 16-bit should be 0x1234:

```
from z3 import *

C1=0x5D7E0D1F2E0F1F84
C2=0x388D76AEE8CB1500
C3=0xD2E9EE7E83C4285B

inp, i1, i2, i3, i4, i5, i6, outp = BitVecs('inp i1 i2 i3 i4 i5 i6 outp', 64)

s = Solver()
s.add(i1==inp*C1)
s.add(i2==RotateRight (i1, i1 & 0xF))
s.add(i3==i2 ^ C2)
s.add(i4==RotateLeft(i3, i3 & 0xF))
s.add(i5==i4 + C3)
s.add(outp==RotateLeft (i5, URem(i5, 60)))

s.add(outp & 0xFFFFFFFF == inp & 0xFFFFFFFF)
s.add(outp & 0xFFFF == 0x1234)

print s.check()
m=s.model()
print m
print (" inp=0x%X" % m[inp].as_long())
```

```
24 print ("outp=0x%X" % m[outp].as_long())
```

Oh yes, this possible as well:

```
sat
[i1 = 2834222860503985872,
 i3 = 2294680776671411152,
 i5 = 17492621421353821227,
 inp = 461881484695179828,
 outp = 419247225543463476,
 i4 = 2294680776671411152,
 i2 = 2834222860503985872]
 inp=0x668EEC35F961234
outp=0x5D177215F961234
```

Z3 works very fast and it means that algorithm is weak, it is not cryptographical at all (like the most of amateur cryptography).

Will it be possible to tackle real cryptography by these methods? Real algorithms like AES, RSA, etc, can also be represented as huge system of equations, but these are that huge that are impossible to work with on computers, now or in near future. Of course, cryptographers are aware of this.

Another article I wrote about Z3 is [36].

Chapter 55

Dongles

Occasionally I do software copy-protection dongle replacements, or “dongle emulators” and here are couple examples of my work [1].

About one of not described cases you may also read here: [36].

55.1 Example #1: MacOS Classic and PowerPC

I’ve got a program for MacOS Classic [2], for PowerPC. The company who developed the software product was disappeared long time ago, so the (legal) customer was afraid of physical dongle damage.

While running without dongle connected, a message box with a text "Invalid Security Device" appeared. Luckily, this text string can be found easily in the executable binary file.

I was not very familiar both with Mac OS Classic and PowerPC, but I tried anyway.

IDA opens the executable file smoothly, reported its type as "PEF (Mac OS or Be OS executable)" (indeed, it is a standard Mac OS Classic file format).

By searching for the text string with error message, I’ve got into this code fragment:

```
...

seg000:000C87FC 38 60 00 01                 li      %r3, 1
seg000:000C8800 48 03 93 41                 bl      check1
seg000:000C8804 60 00 00 00                 nop
seg000:000C8808 54 60 06 3F                 clrlwi. %r0, %r3, 24
seg000:000C880C 40 82 00 40                 bne     OK
```

[1] Read more about it: http://yurichev.com/dongles.html

[2] pre-UNIX MacOS

```
seg000:000C8810 80 62 9F D8                 lwz     %r3, ↙
    ↳ TC_aInvalidSecurityDevice

...
```

Yes, this is PowerPC code. The CPU is very typical 32-bit RISC of 1990s era. Each instruction occupies 4 bytes (just as in MIPS and ARM) and its names are somewhat resembling MIPS instruction names.

`check1()` is a function name I gave it to lately. `BL` is *Branch Link* instruction, e.g., intended for subroutines calling. The crucial point is BNE instruction jumping if dongle protection check is passed or not jumping if error is occurred: then the address of the text string being loaded into r3 register for the subsequent passage into message box routine.

From the [33] I've got to know the r3 register is used for values returning (and r4, in case of 64-bit values).

Another yet unknown instruction is `CLRLWI`. From [13] I've got to know that this instruction do both clearing and loading. In our case, it clears 24 high bits from the value in r3 and put it to r0, so it is analogical to `MOVZX` in x86 (13.1.2), but it also sets the flags, so the BNE can check them after.

Let's take a look into `check1()` function:

```
seg000:00101B40              check1: # CODE XREF: seg000:00063↙
    ↳ E7Cp
seg000:00101B40                      # sub_64070+160p ...
seg000:00101B40
seg000:00101B40              .set arg_8,  8
seg000:00101B40
seg000:00101B40 7C 08 02 A6          mflr    %r0
seg000:00101B44 90 01 00 08          stw     %r0, arg_8(%sp)
seg000:00101B48 94 21 FF C0          stwu    %sp, -0x40(%sp)
seg000:00101B4C 48 01 6B 39          bl      check2
seg000:00101B50 60 00 00 00          nop
seg000:00101B54 80 01 00 48          lwz     %r0, 0x40+arg_8(%sp)
seg000:00101B58 38 21 00 40          addi    %sp, %sp, 0x40
seg000:00101B5C 7C 08 03 A6          mtlr    %r0
seg000:00101B60 4E 80 00 20          blr
seg000:00101B60              # End of function check1
```

As I can see in IDA, that function is called from many places in program, but only r3 register value is checked right after each call. All this function does is calling other function, so it is thunk function: there is function prologue and epilogue, but r3 register is not touched, so `check1()` returns what `check2()` returns.

BLR[3] is seems return from function, but since IDA does functions layout, we proba-

[3] (PowerPC) Branch to Link Register

bly do not need to be interesting in this. It seems, since it is a typical RISC, subroutines are called using link register, just like in ARM.

`check2()` function is more complex:

```
seg000:00118684                check2: # CODE XREF: check1+Cp
seg000:00118684
seg000:00118684                .set var_18, -0x18
seg000:00118684                .set var_C, -0xC
seg000:00118684                .set var_8, -8
seg000:00118684                .set var_4, -4
seg000:00118684                .set arg_8,  8
seg000:00118684
seg000:00118684 93 E1 FF FC    stw     %r31, var_4(%sp)
seg000:00118688 7C 08 02 A6    mflr    %r0
seg000:0011868C 83 E2 95 A8    lwz     %r31, off_1485E8 # ↙
    ↳ dword_24B704
seg000:00118690                  .using dword_24B704, %r31
seg000:00118690 93 C1 FF F8    stw     %r30, var_8(%sp)
seg000:00118694 93 A1 FF F4    stw     %r29, var_C(%sp)
seg000:00118698 7C 7D 1B 78    mr      %r29, %r3
seg000:0011869C 90 01 00 08    stw     %r0, arg_8(%sp)
seg000:001186A0 54 60 06 3E    clrlwi  %r0, %r3, 24
seg000:001186A4 28 00 00 01    cmplwi  %r0, 1
seg000:001186A8 94 21 FF B0    stwu    %sp, -0x50(%sp)
seg000:001186AC 40 82 00 0C    bne     loc_1186B8
seg000:001186B0 38 60 00 01    li      %r3, 1
seg000:001186B4 48 00 00 6C    b       exit
seg000:001186B8
seg000:001186B8                loc_1186B8: # CODE XREF: check2+28j
seg000:001186B8 48 00 03 D5    bl      sub_118A8C
seg000:001186BC 60 00 00 00    nop
seg000:001186C0 3B C0 00 00    li      %r30, 0
seg000:001186C4
seg000:001186C4                skip:       # CODE XREF: check2+94j
seg000:001186C4 57 C0 06 3F    clrlwi. %r0, %r30, 24
seg000:001186C8 41 82 00 18    beq     loc_1186E0
seg000:001186CC 38 61 00 38    addi    %r3, %sp, 0x50+var_18
seg000:001186D0 80 9F 00 00    lwz     %r4, dword_24B704
seg000:001186D4 48 00 C0 55    bl      .RBEFINDNEXT
seg000:001186D8 60 00 00 00    nop
seg000:001186DC 48 00 00 1C    b       loc_1186F8
seg000:001186E0
seg000:001186E0                loc_1186E0: # CODE XREF: check2+44j
seg000:001186E0 80 BF 00 00    lwz     %r5, dword_24B704
```

```
seg000:001186E4 38 81 00 38   addi    %r4, %sp, 0x50+var_18
seg000:001186E8 38 60 08 C2   li      %r3, 0x1234
seg000:001186EC 48 00 BF 99   bl      .RBEFINDFIRST
seg000:001186F0 60 00 00 00   nop
seg000:001186F4 3B C0 00 01   li      %r30, 1
seg000:001186F8
seg000:001186F8             loc_1186F8: # CODE XREF: check2+58j
seg000:001186F8 54 60 04 3F   clrlwi. %r0, %r3, 16
seg000:001186FC 41 82 00 0C   beq     must_jump
seg000:00118700 38 60 00 00   li      %r3, 0          # error
seg000:00118704 48 00 00 1C   b       exit
seg000:00118708
seg000:00118708             must_jump:  # CODE XREF: check2+78j
seg000:00118708 7F A3 EB 78   mr      %r3, %r29
seg000:0011870C 48 00 00 31   bl      check3
seg000:00118710 60 00 00 00   nop
seg000:00118714 54 60 06 3F   clrlwi. %r0, %r3, 24
seg000:00118718 41 82 FF AC   beq     skip
seg000:0011871C 38 60 00 01   li      %r3, 1
seg000:00118720
seg000:00118720             exit:       # CODE XREF: check2+30j
seg000:00118720                         # check2+80j
seg000:00118720 80 01 00 58   lwz     %r0, 0x50+arg_8(%sp)
seg000:00118724 38 21 00 50   addi    %sp, %sp, 0x50
seg000:00118728 83 E1 FF FC   lwz     %r31, var_4(%sp)
seg000:0011872C 7C 08 03 A6   mtlr    %r0
seg000:00118730 83 C1 FF F8   lwz     %r30, var_8(%sp)
seg000:00118734 83 A1 FF F4   lwz     %r29, var_C(%sp)
seg000:00118738 4E 80 00 20   blr
seg000:00118738             # End of function check2
```

I'm lucky again: some function names are leaved in the executable (debug symbols section? I'm not sure, since I'm not very familiar with the file format, maybe it is some kind of PE exports? (48.2.7)), like `.RBEFINDNEXT()` and `.RBEFINDFIRST()`. Eventually these functions are calling other functions with names like `.GetNextDeviceViaUSB` `.USBSendPKT()`, so these are clearly dealing with USB device.

There are even a function named `.GetNextEve3Device()`—sounds familiar, there was Sentinel Eve3 dongle for ADB port (present on Macs) in 1990s.

Let's first take a look on how r3 register is set before return simultaneously ignoring all we see. We know that "good" r3 value should be non-zero, zero r3 will lead execution flow to the message box with an error message.

There are two instructions `li %r3, 1` present in the function and one `li %r3, 0` (*Load Immediate*, i.e., loading value into register). The very first instruction at `0x001186B0`—frankly speaking, I don't know what it mean, I need some more time to learn PowerPC

assembly language.

What we see next is, however, easier to understand: `.RBEFINDFIRST()` is called: in case of its failure, 0 is written into r3 and we jump to *exit*, otherwise another function is called (`check3()`) —if it is failing too, the `.RBEFINDNEXT()` is called, probably, in order to look for another USB device.

N.B.: `clrlwi. %r0, %r3, 16` it is analogical to what we already saw, but it clears 16 bits, i.e., `.RBEFINDFIRST()` probably returns 16-bit value.

`B` meaning *branch* is unconditional jump.

BEQ is inverse instruction of BNE.

Let's see `check3()`:

```
seg000:0011873C                check3: # CODE XREF: check2+88p
seg000:0011873C
seg000:0011873C                .set var_18, -0x18
seg000:0011873C                .set var_C, -0xC
seg000:0011873C                .set var_8, -8
seg000:0011873C                .set var_4, -4
seg000:0011873C                .set arg_8,  8
seg000:0011873C
seg000:0011873C 93 E1 FF FC    stw     %r31, var_4(%sp)
seg000:00118740 7C 08 02 A6    mflr    %r0
seg000:00118744 38 A0 00 00    li      %r5, 0
seg000:00118748 93 C1 FF F8    stw     %r30, var_8(%sp)
seg000:0011874C 83 C2 95 A8    lwz     %r30, off_1485E8 # ↙
    ↳ dword_24B704
seg000:00118750                .using dword_24B704, %r30
seg000:00118750 93 A1 FF F4    stw     %r29, var_C(%sp)
seg000:00118754 3B A3 00 00    addi    %r29, %r3, 0
seg000:00118758 38 60 00 00    li      %r3, 0
seg000:0011875C 90 01 00 08    stw     %r0, arg_8(%sp)
seg000:00118760 94 21 FF B0    stwu    %sp, -0x50(%sp)
seg000:00118764 80 DE 00 00    lwz     %r6, dword_24B704
seg000:00118768 38 81 00 38    addi    %r4, %sp, 0x50+var_18
seg000:0011876C 48 00 C0 5D    bl      .RBEREAD
seg000:00118770 60 00 00 00    nop
seg000:00118774 54 60 04 3F    clrlwi. %r0, %r3, 16
seg000:00118778 41 82 00 0C    beq     loc_118784
seg000:0011877C 38 60 00 00    li      %r3, 0
seg000:00118780 48 00 02 F0    b       exit
seg000:00118784
seg000:00118784                loc_118784: # CODE XREF: check3+3Cj
seg000:00118784 A0 01 00 38    lhz     %r0, 0x50+var_18(%sp)
seg000:00118788 28 00 04 B2    cmplwi  %r0, 0x1100
seg000:0011878C 41 82 00 0C    beq     loc_118798
```

```
seg000:00118790 38 60 00 00   li      %r3, 0
seg000:00118794 48 00 02 DC   b       exit
seg000:00118798
seg000:00118798             loc_118798: # CODE XREF: check3+50j
seg000:00118798 80 DE 00 00   lwz     %r6, dword_24B704
seg000:0011879C 38 81 00 38   addi    %r4, %sp, 0x50+var_18
seg000:001187A0 38 60 00 01   li      %r3, 1
seg000:001187A4 38 A0 00 00   li      %r5, 0
seg000:001187A8 48 00 C0 21   bl      .RBEREAD
seg000:001187AC 60 00 00 00   nop
seg000:001187B0 54 60 04 3F   clrlwi. %r0, %r3, 16
seg000:001187B4 41 82 00 0C   beq     loc_1187C0
seg000:001187B8 38 60 00 00   li      %r3, 0
seg000:001187BC 48 00 02 B4   b       exit
seg000:001187C0
seg000:001187C0             loc_1187C0: # CODE XREF: check3+78j
seg000:001187C0 A0 01 00 38   lhz     %r0, 0x50+var_18(%sp)
seg000:001187C4 28 00 06 4B   cmplwi  %r0, 0x09AB
seg000:001187C8 41 82 00 0C   beq     loc_1187D4
seg000:001187CC 38 60 00 00   li      %r3, 0
seg000:001187D0 48 00 02 A0   b       exit
seg000:001187D4
seg000:001187D4             loc_1187D4: # CODE XREF: check3+8Cj
seg000:001187D4 4B F9 F3 D9   bl      sub_B7BAC
seg000:001187D8 60 00 00 00   nop
seg000:001187DC 54 60 06 3E   clrlwi  %r0, %r3, 24
seg000:001187E0 2C 00 00 05   cmpwi   %r0, 5
seg000:001187E4 41 82 01 00   beq     loc_1188E4
seg000:001187E8 40 80 00 10   bge     loc_1187F8
seg000:001187EC 2C 00 00 04   cmpwi   %r0, 4
seg000:001187F0 40 80 00 58   bge     loc_118848
seg000:001187F4 48 00 01 8C   b       loc_118980
seg000:001187F8
seg000:001187F8             loc_1187F8: # CODE XREF: check3+ACj
seg000:001187F8 2C 00 00 0B   cmpwi   %r0, 0xB
seg000:001187FC 41 82 00 08   beq     loc_118804
seg000:00118800 48 00 01 80   b       loc_118980
seg000:00118804
seg000:00118804             loc_118804: # CODE XREF: check3+C0j
seg000:00118804 80 DE 00 00   lwz     %r6, dword_24B704
seg000:00118808 38 81 00 38   addi    %r4, %sp, 0x50+var_18
seg000:0011880C 38 60 00 08   li      %r3, 8
seg000:00118810 38 A0 00 00   li      %r5, 0
```

```
seg000:00118814 48 00 BF B5   bl      .RBEREAD
seg000:00118818 60 00 00 00   nop
seg000:0011881C 54 60 04 3F   clrlwi. %r0, %r3, 16
seg000:00118820 41 82 00 0C   beq     loc_11882C
seg000:00118824 38 60 00 00   li      %r3, 0
seg000:00118828 48 00 02 48   b       exit
seg000:0011882C
seg000:0011882C             loc_11882C: # CODE XREF: check3+E4j
seg000:0011882C A0 01 00 38   lhz     %r0, 0x50+var_18(%sp)
seg000:00118830 28 00 11 30   cmplwi  %r0, 0xFEA0
seg000:00118834 41 82 00 0C   beq     loc_118840
seg000:00118838 38 60 00 00   li      %r3, 0
seg000:0011883C 48 00 02 34   b       exit
seg000:00118840
seg000:00118840             loc_118840: # CODE XREF: check3+F8j
seg000:00118840 38 60 00 01   li      %r3, 1
seg000:00118844 48 00 02 2C   b       exit
seg000:00118848
seg000:00118848             loc_118848: # CODE XREF: check3+B4j
seg000:00118848 80 DE 00 00   lwz     %r6, dword_24B704
seg000:0011884C 38 81 00 38   addi    %r4, %sp, 0x50+var_18
seg000:00118850 38 60 00 0A   li      %r3, 0xA
seg000:00118854 38 A0 00 00   li      %r5, 0
seg000:00118858 48 00 BF 71   bl      .RBEREAD
seg000:0011885C 60 00 00 00   nop
seg000:00118860 54 60 04 3F   clrlwi. %r0, %r3, 16
seg000:00118864 41 82 00 0C   beq     loc_118870
seg000:00118868 38 60 00 00   li      %r3, 0
seg000:0011886C 48 00 02 04   b       exit
seg000:00118870
seg000:00118870             loc_118870: # CODE XREF: check3+128j
seg000:00118870 A0 01 00 38   lhz     %r0, 0x50+var_18(%sp)
seg000:00118874 28 00 03 F3   cmplwi  %r0, 0xA6E1
seg000:00118878 41 82 00 0C   beq     loc_118884
seg000:0011887C 38 60 00 00   li      %r3, 0
seg000:00118880 48 00 01 F0   b       exit
seg000:00118884
seg000:00118884             loc_118884: # CODE XREF: check3+13Cj
seg000:00118884 57 BF 06 3E   clrlwi  %r31, %r29, 24
seg000:00118888 28 1F 00 02   cmplwi  %r31, 2
seg000:0011888C 40 82 00 0C   bne     loc_118898
seg000:00118890 38 60 00 01   li      %r3, 1
seg000:00118894 48 00 01 DC   b       exit
```

```
seg000:00118898
seg000:00118898                 loc_118898: # CODE XREF: check3+150j
seg000:00118898 80 DE 00 00     lwz     %r6, dword_24B704
seg000:0011889C 38 81 00 38     addi    %r4, %sp, 0x50+var_18
seg000:001188A0 38 60 00 0B     li      %r3, 0xB
seg000:001188A4 38 A0 00 00     li      %r5, 0
seg000:001188A8 48 00 BF 21     bl      .RBEREAD
seg000:001188AC 60 00 00 00     nop
seg000:001188B0 54 60 04 3F     clrlwi. %r0, %r3, 16
seg000:001188B4 41 82 00 0C     beq     loc_1188C0
seg000:001188B8 38 60 00 00     li      %r3, 0
seg000:001188BC 48 00 01 B4     b       exit
seg000:001188C0
seg000:001188C0                 loc_1188C0: # CODE XREF: check3+178j
seg000:001188C0 A0 01 00 38     lhz     %r0, 0x50+var_18(%sp)
seg000:001188C4 28 00 23 1C     cmplwi  %r0, 0x1C20
seg000:001188C8 41 82 00 0C     beq     loc_1188D4
seg000:001188CC 38 60 00 00     li      %r3, 0
seg000:001188D0 48 00 01 A0     b       exit
seg000:001188D4
seg000:001188D4                 loc_1188D4: # CODE XREF: check3+18Cj
seg000:001188D4 28 1F 00 03     cmplwi  %r31, 3
seg000:001188D8 40 82 01 94     bne     error
seg000:001188DC 38 60 00 01     li      %r3, 1
seg000:001188E0 48 00 01 90     b       exit
seg000:001188E4
seg000:001188E4                 loc_1188E4: # CODE XREF: check3+A8j
seg000:001188E4 80 DE 00 00     lwz     %r6, dword_24B704
seg000:001188E8 38 81 00 38     addi    %r4, %sp, 0x50+var_18
seg000:001188EC 38 60 00 0C     li      %r3, 0xC
seg000:001188F0 38 A0 00 00     li      %r5, 0
seg000:001188F4 48 00 BE D5     bl      .RBEREAD
seg000:001188F8 60 00 00 00     nop
seg000:001188FC 54 60 04 3F     clrlwi. %r0, %r3, 16
seg000:00118900 41 82 00 0C     beq     loc_11890C
seg000:00118904 38 60 00 00     li      %r3, 0
seg000:00118908 48 00 01 68     b       exit
seg000:0011890C
seg000:0011890C                 loc_11890C: # CODE XREF: check3+1C4j
seg000:0011890C A0 01 00 38     lhz     %r0, 0x50+var_18(%sp)
seg000:00118910 28 00 1F 40     cmplwi  %r0, 0x40FF
seg000:00118914 41 82 00 0C     beq     loc_118920
seg000:00118918 38 60 00 00     li      %r3, 0
```

```
seg000:0011891C 48 00 01 54   b       exit
seg000:00118920
seg000:00118920             loc_118920: # CODE XREF: check3+1D8j
seg000:00118920 57 BF 06 3E   clrlwi  %r31, %r29, 24
seg000:00118924 28 1F 00 02   cmplwi  %r31, 2
seg000:00118928 40 82 00 0C   bne     loc_118934
seg000:0011892C 38 60 00 01   li      %r3, 1
seg000:00118930 48 00 01 40   b       exit
seg000:00118934
seg000:00118934             loc_118934: # CODE XREF: check3+1ECj
seg000:00118934 80 DE 00 00   lwz     %r6, dword_24B704
seg000:00118938 38 81 00 38   addi    %r4, %sp, 0x50+var_18
seg000:0011893C 38 60 00 0D   li      %r3, 0xD
seg000:00118940 38 A0 00 00   li      %r5, 0
seg000:00118944 48 00 BE 85   bl      .RBEREAD
seg000:00118948 60 00 00 00   nop
seg000:0011894C 54 60 04 3F   clrlwi. %r0, %r3, 16
seg000:00118950 41 82 00 0C   beq     loc_11895C
seg000:00118954 38 60 00 00   li      %r3, 0
seg000:00118958 48 00 01 18   b       exit
seg000:0011895C
seg000:0011895C             loc_11895C: # CODE XREF: check3+214j
seg000:0011895C A0 01 00 38   lhz     %r0, 0x50+var_18(%sp)
seg000:00118960 28 00 07 CF   cmplwi  %r0, 0xFC7
seg000:00118964 41 82 00 0C   beq     loc_118970
seg000:00118968 38 60 00 00   li      %r3, 0
seg000:0011896C 48 00 01 04   b       exit
seg000:00118970
seg000:00118970             loc_118970: # CODE XREF: check3+228j
seg000:00118970 28 1F 00 03   cmplwi  %r31, 3
seg000:00118974 40 82 00 F8   bne     error
seg000:00118978 38 60 00 01   li      %r3, 1
seg000:0011897C 48 00 00 F4   b       exit
seg000:00118980
seg000:00118980             loc_118980: # CODE XREF: check3+B8j
seg000:00118980                           # check3+C4j
seg000:00118980 80 DE 00 00   lwz     %r6, dword_24B704
seg000:00118984 38 81 00 38   addi    %r4, %sp, 0x50+var_18
seg000:00118988 3B E0 00 00   li      %r31, 0
seg000:0011898C 38 60 00 04   li      %r3, 4
seg000:00118990 38 A0 00 00   li      %r5, 0
seg000:00118994 48 00 BE 35   bl      .RBEREAD
seg000:00118998 60 00 00 00   nop
```

```
seg000:0011899C 54 60 04 3F   clrlwi. %r0, %r3, 16
seg000:001189A0 41 82 00 0C   beq     loc_1189AC
seg000:001189A4 38 60 00 00   li      %r3, 0
seg000:001189A8 48 00 00 C8   b       exit
seg000:001189AC
seg000:001189AC             loc_1189AC: # CODE XREF: check3+264j
seg000:001189AC A0 01 00 38   lhz     %r0, 0x50+var_18(%sp)
seg000:001189B0 28 00 1D 6A   cmplwi  %r0, 0xAED0
seg000:001189B4 40 82 00 0C   bne     loc_1189C0
seg000:001189B8 3B E0 00 01   li      %r31, 1
seg000:001189BC 48 00 00 14   b       loc_1189D0
seg000:001189C0
seg000:001189C0             loc_1189C0: # CODE XREF: check3+278j
seg000:001189C0 28 00 18 28   cmplwi  %r0, 0x2818
seg000:001189C4 41 82 00 0C   beq     loc_1189D0
seg000:001189C8 38 60 00 00   li      %r3, 0
seg000:001189CC 48 00 00 A4   b       exit
seg000:001189D0
seg000:001189D0             loc_1189D0: # CODE XREF: check3+280j
seg000:001189D0                         # check3+288j
seg000:001189D0 57 A0 06 3E   clrlwi  %r0, %r29, 24
seg000:001189D4 28 00 00 02   cmplwi  %r0, 2
seg000:001189D8 40 82 00 20   bne     loc_1189F8
seg000:001189DC 57 E0 06 3F   clrlwi. %r0, %r31, 24
seg000:001189E0 41 82 00 10   beq     good2
seg000:001189E4 48 00 4C 69   bl      sub_11D64C
seg000:001189E8 60 00 00 00   nop
seg000:001189EC 48 00 00 84   b       exit
seg000:001189F0
seg000:001189F0             good2:      # CODE XREF: check3+2A4j
seg000:001189F0 38 60 00 01   li      %r3, 1
seg000:001189F4 48 00 00 7C   b       exit
seg000:001189F8
seg000:001189F8             loc_1189F8: # CODE XREF: check3+29Cj
seg000:001189F8 80 DE 00 00   lwz     %r6, dword_24B704
seg000:001189FC 38 81 00 38   addi    %r4, %sp, 0x50+var_18
seg000:00118A00 38 60 00 05   li      %r3, 5
seg000:00118A04 38 A0 00 00   li      %r5, 0
seg000:00118A08 48 00 BD C1   bl      .RBEREAD
seg000:00118A0C 60 00 00 00   nop
seg000:00118A10 54 60 04 3F   clrlwi. %r0, %r3, 16
seg000:00118A14 41 82 00 0C   beq     loc_118A20
seg000:00118A18 38 60 00 00   li      %r3, 0
```

```
seg000:00118A1C 48 00 00 54   b       exit
seg000:00118A20
seg000:00118A20             loc_118A20: # CODE XREF: check3+2D8j
seg000:00118A20 A0 01 00 38   lhz     %r0, 0x50+var_18(%sp)
seg000:00118A24 28 00 11 D3   cmplwi  %r0, 0xD300
seg000:00118A28 40 82 00 0C   bne     loc_118A34
seg000:00118A2C 3B E0 00 01   li      %r31, 1
seg000:00118A30 48 00 00 14   b       good1
seg000:00118A34
seg000:00118A34             loc_118A34: # CODE XREF: check3+2ECj
seg000:00118A34 28 00 1A EB   cmplwi  %r0, 0xEBA1
seg000:00118A38 41 82 00 0C   beq     good1
seg000:00118A3C 38 60 00 00   li      %r3, 0
seg000:00118A40 48 00 00 30   b       exit
seg000:00118A44
seg000:00118A44             good1:      # CODE XREF: check3+2F4j
seg000:00118A44                         # check3+2FCj
seg000:00118A44 57 A0 06 3E   clrlwi  %r0, %r29, 24
seg000:00118A48 28 00 00 03   cmplwi  %r0, 3
seg000:00118A4C 40 82 00 20   bne     error
seg000:00118A50 57 E0 06 3F   clrlwi. %r0, %r31, 24
seg000:00118A54 41 82 00 10   beq     good
seg000:00118A58 48 00 4B F5   bl      sub_11D64C
seg000:00118A5C 60 00 00 00   nop
seg000:00118A60 48 00 00 10   b       exit
seg000:00118A64
seg000:00118A64             good:       # CODE XREF: check3+318j
seg000:00118A64 38 60 00 01   li      %r3, 1
seg000:00118A68 48 00 00 08   b       exit
seg000:00118A6C
seg000:00118A6C             error:      # CODE XREF: check3+19Cj
seg000:00118A6C                         # check3+238j ...
seg000:00118A6C 38 60 00 00   li      %r3, 0
seg000:00118A70
seg000:00118A70             exit:       # CODE XREF: check3+44j
seg000:00118A70                         # check3+58j ...
seg000:00118A70 80 01 00 58   lwz     %r0, 0x50+arg_8(%sp)
seg000:00118A74 38 21 00 50   addi    %sp, %sp, 0x50
seg000:00118A78 83 E1 FF FC   lwz     %r31, var_4(%sp)
seg000:00118A7C 7C 08 03 A6   mtlr    %r0
seg000:00118A80 83 C1 FF F8   lwz     %r30, var_8(%sp)
seg000:00118A84 83 A1 FF F4   lwz     %r29, var_C(%sp)
seg000:00118A88 4E 80 00 20   blr
```

```
seg000:00118A88                 # End of function check3
```

There are a lot of calls to `.RBEREAD()`. The function is probably return some values from the dongle, so they are compared here with hard-coded variables using `CMPLWI`.

We also see that r3 register is also filled before each call to `.RBEREAD()` by one of these values: 0, 1, 8, 0xA, 0xB, 0xC, 0xD, 4, 5. Probably memory address or something like that?

Yes, indeed, by googling these function names it is easy to find Sentinel Eve3 dongle manual!

I probably even do not need to learn other PowerPC instructions: all this function does is just calls `.RBEREAD()`, compare its results with constants and returns 1 if comparisons are fine or 0 otherwise.

OK, all we've got is that `check1()` should return always 1 or any other non-zero value. But since I'm not very confident in PowerPC instructions, I will be careful: I will patch jumps in `check2()` at `0x001186FC` and `0x00118718`.

At `0x001186FC` I wrote bytes 0x48 and 0 thus converting BEQ instruction into `B` (unconditional jump): I spot its opcode in the code without even referring to [13].

At `0x00118718` I wrote 0x60 and 3 zero bytes thus converting it to NOP instruction: I spot its opcode in the code too.

Summarizing, such small modifications can be done with IDA and minimal assembly language knowledge.

55.2 Example #2: SCO OpenServer

An ancient software for SCO OpenServer from 1997 developed by a company disappeared long time ago.

There is a special dongle driver to be installed in the system, containing text strings: "Copyright 1989, Rainbow Technologies, Inc., Irvine, CA" and "Sentinel Integrated Driver Ver. 3.0 ".

After driver installation in SCO OpenServer, these device files are appeared in /dev filesystem:

```
/dev/rbsl8
/dev/rbsl9
/dev/rbsl10
```

The program without dongle connected reports error, but the error string cannot be found in the executables.

Thanks to IDA, it does its job perfectly working out COFF executable used in SCO OpenServer.

I've tried to find "rbsl" and indeed, found it in this code fragment:

```
.text:00022AB8         public SSQC
```

```
.text:00022AB8 SSQC   proc near ; CODE XREF: SSQ+7p
.text:00022AB8
.text:00022AB8 var_44 = byte ptr -44h
.text:00022AB8 var_29 = byte ptr -29h
.text:00022AB8 arg_0  = dword ptr  8
.text:00022AB8
.text:00022AB8        push    ebp
.text:00022AB9        mov     ebp, esp
.text:00022ABB        sub     esp, 44h
.text:00022ABE        push    edi
.text:00022ABF        mov     edi, offset unk_4035D0
.text:00022AC4        push    esi
.text:00022AC5        mov     esi, [ebp+arg_0]
.text:00022AC8        push    ebx
.text:00022AC9        push    esi
.text:00022ACA        call    strlen
.text:00022ACF        add     esp, 4
.text:00022AD2        cmp     eax, 2
.text:00022AD7        jnz     loc_22BA4
.text:00022ADD        inc     esi
.text:00022ADE        mov     al, [esi-1]
.text:00022AE1        movsx   eax, al
.text:00022AE4        cmp     eax, '3'
.text:00022AE9        jz      loc_22B84
.text:00022AEF        cmp     eax, '4'
.text:00022AF4        jz      loc_22B94
.text:00022AFA        cmp     eax, '5'
.text:00022AFF        jnz     short loc_22B6B
.text:00022B01        movsx   ebx, byte ptr [esi]
.text:00022B04        sub     ebx, '0'
.text:00022B07        mov     eax, 7
.text:00022B0C        add     eax, ebx
.text:00022B0E        push    eax
.text:00022B0F        lea     eax, [ebp+var_44]
.text:00022B12        push    offset aDevSlD  ; "/dev/sl%d"
.text:00022B17        push    eax
.text:00022B18        call    nl_sprintf
.text:00022B1D        push    0               ; int
.text:00022B1F        push    offset aDevRbsl8 ; char *
.text:00022B24        call    _access
.text:00022B29        add     esp, 14h
.text:00022B2C        cmp     eax, 0FFFFFFFFh
.text:00022B31        jz      short loc_22B48
```

```
.text:00022B33        lea     eax, [ebx+7]
.text:00022B36        push    eax
.text:00022B37        lea     eax, [ebp+var_44]
.text:00022B3A        push    offset aDevRbslD ; "/dev/rbsl%d"
.text:00022B3F        push    eax
.text:00022B40        call    nl_sprintf
.text:00022B45        add     esp, 0Ch
.text:00022B48
.text:00022B48 loc_22B48: ; CODE XREF: SSQC+79j
.text:00022B48        mov     edx, [edi]
.text:00022B4A        test    edx, edx
.text:00022B4C        jle     short loc_22B57
.text:00022B4E        push    edx             ; int
.text:00022B4F        call    _close
.text:00022B54        add     esp, 4
.text:00022B57
.text:00022B57 loc_22B57: ; CODE XREF: SSQC+94j
.text:00022B57        push    2               ; int
.text:00022B59        lea     eax, [ebp+var_44]
.text:00022B5C        push    eax             ; char *
.text:00022B5D        call    _open
.text:00022B62        add     esp, 8
.text:00022B65        test    eax, eax
.text:00022B67        mov     [edi], eax
.text:00022B69        jge     short loc_22B78
.text:00022B6B
.text:00022B6B loc_22B6B: ; CODE XREF: SSQC+47j
.text:00022B6B        mov     eax, 0FFFFFFFFh
.text:00022B70        pop     ebx
.text:00022B71        pop     esi
.text:00022B72        pop     edi
.text:00022B73        mov     esp, ebp
.text:00022B75        pop     ebp
.text:00022B76        retn
.text:00022B78
.text:00022B78 loc_22B78: ; CODE XREF: SSQC+B1j
.text:00022B78        pop     ebx
.text:00022B79        pop     esi
.text:00022B7A        pop     edi
.text:00022B7B        xor     eax, eax
.text:00022B7D        mov     esp, ebp
.text:00022B7F        pop     ebp
.text:00022B80        retn
```

```
.text:00022B84
.text:00022B84 loc_22B84: ; CODE XREF: SSQC+31j
.text:00022B84        mov     al, [esi]
.text:00022B86        pop     ebx
.text:00022B87        pop     esi
.text:00022B88        pop     edi
.text:00022B89        mov     ds:byte_407224, al
.text:00022B8E        mov     esp, ebp
.text:00022B90        xor     eax, eax
.text:00022B92        pop     ebp
.text:00022B93        retn
.text:00022B94
.text:00022B94 loc_22B94: ; CODE XREF: SSQC+3Cj
.text:00022B94        mov     al, [esi]
.text:00022B96        pop     ebx
.text:00022B97        pop     esi
.text:00022B98        pop     edi
.text:00022B99        mov     ds:byte_407225, al
.text:00022B9E        mov     esp, ebp
.text:00022BA0        xor     eax, eax
.text:00022BA2        pop     ebp
.text:00022BA3        retn
.text:00022BA4
.text:00022BA4 loc_22BA4: ; CODE XREF: SSQC+1Fj
.text:00022BA4        movsx   eax, ds:byte_407225
.text:00022BAB        push    esi
.text:00022BAC        push    eax
.text:00022BAD        movsx   eax, ds:byte_407224
.text:00022BB4        push    eax
.text:00022BB5        lea     eax, [ebp+var_44]
.text:00022BB8        push    offset a46CCS   ; "46%c%c%s"
.text:00022BBD        push    eax
.text:00022BBE        call    nl_sprintf
.text:00022BC3        lea     eax, [ebp+var_44]
.text:00022BC6        push    eax
.text:00022BC7        call    strlen
.text:00022BCC        add     esp, 18h
.text:00022BCF        cmp     eax, 1Bh
.text:00022BD4        jle     short loc_22BDA
.text:00022BD6        mov     [ebp+var_29], 0
.text:00022BDA
.text:00022BDA loc_22BDA: ; CODE XREF: SSQC+11Cj
.text:00022BDA        lea     eax, [ebp+var_44]
```

```
.text:00022BDD         push    eax
.text:00022BDE         call    strlen
.text:00022BE3         push    eax             ; unsigned int
.text:00022BE4         lea     eax, [ebp+var_44]
.text:00022BE7         push    eax             ; void *
.text:00022BE8         mov     eax, [edi]
.text:00022BEA         push    eax             ; int
.text:00022BEB         call    _write
.text:00022BF0         add     esp, 10h
.text:00022BF3         pop     ebx
.text:00022BF4         pop     esi
.text:00022BF5         pop     edi
.text:00022BF6         mov     esp, ebp
.text:00022BF8         pop     ebp
.text:00022BF9         retn
.text:00022BFA         db 0Eh dup(90h)
.text:00022BFA SSQC    endp
```

Yes, indeed, the program should communicate with driver somehow and that is how it is.

The only place `SSQC()` function called is the thunk function:

```
.text:0000DBE8         public SSQ
.text:0000DBE8 SSQ     proc near ; CODE XREF: sys_info+A9p
.text:0000DBE8                   ; sys_info+CBp ...
.text:0000DBE8
.text:0000DBE8 arg_0   = dword ptr  8
.text:0000DBE8
.text:0000DBE8         push    ebp
.text:0000DBE9         mov     ebp, esp
.text:0000DBEB         mov     edx, [ebp+arg_0]
.text:0000DBEE         push    edx
.text:0000DBEF         call    SSQC
.text:0000DBF4         add     esp, 4
.text:0000DBF7         mov     esp, ebp
.text:0000DBF9         pop     ebp
.text:0000DBFA         retn
.text:0000DBFB SSQ     endp
```

SSQ() is called at least from 2 functions.

One of these is:

```
.data:0040169C _51_52_53       dd offset aPressAnyKeyT_0 ; DATA ↙
    ↳ XREF: init_sys+392r
```

```
.data:0040169C                                          ; ↙
    ↳ sys_info+A1r
.data:0040169C                                          ; "PRESS ↙
    ↳ ANY KEY TO CONTINUE: "
.data:004016A0                  dd offset a51           ; "51"
.data:004016A4                  dd offset a52           ; "52"
.data:004016A8                  dd offset a53           ; "53"

...

.data:004016B8 _3C_or_3E        dd offset a3c           ; DATA ↙
    ↳ XREF: sys_info:loc_D67Br
.data:004016B8                                          ; "3C"
.data:004016BC                  dd offset a3e           ; "3E"

; these names I gave to the labels:
.data:004016C0 answers1         dd 6B05h                ; DATA ↙
    ↳ XREF: sys_info+E7r
.data:004016C4                  dd 3D87h
.data:004016C8 answers2         dd 3Ch                  ; DATA ↙
    ↳ XREF: sys_info+F2r
.data:004016CC                  dd 832h
.data:004016D0 _C_and_B         db 0Ch                  ; DATA ↙
    ↳ XREF: sys_info+BAr
.data:004016D0                                          ; ↙
    ↳ sys_info:OKr
.data:004016D1 byte_4016D1      db 0Bh                  ; DATA ↙
    ↳ XREF: sys_info+FDr
.data:004016D2                  db    0

...

.text:0000D652                  xor     eax, eax
.text:0000D654                  mov     al, ds:ctl_port
.text:0000D659                  mov     ecx, _51_52_53[eax*4]
.text:0000D660                  push    ecx
.text:0000D661                  call    SSQ
.text:0000D666                  add     esp, 4
.text:0000D669                  cmp     eax, 0FFFFFFFFh
.text:0000D66E                  jz      short loc_D6D1
.text:0000D670                  xor     ebx, ebx
.text:0000D672                  mov     al, _C_and_B
.text:0000D677                  test    al, al
```

```
.text:0000D679                  jz      short loc_D6C0
.text:0000D67B
.text:0000D67B loc_D67B: ; CODE XREF: sys_info+106j
.text:0000D67B                  mov     eax, _3C_or_3E[ebx*4]
.text:0000D682                  push    eax
.text:0000D683                  call    SSQ
.text:0000D688                  push    offset a4g      ; "4G"
.text:0000D68D                  call    SSQ
.text:0000D692                  push    offset a0123456789 ; ↙
    ↳ "0123456789"
.text:0000D697                  call    SSQ
.text:0000D69C                  add     esp, 0Ch
.text:0000D69F                  mov     edx, answers1[ebx*4]
.text:0000D6A6                  cmp     eax, edx
.text:0000D6A8                  jz      short OK
.text:0000D6AA                  mov     ecx, answers2[ebx*4]
.text:0000D6B1                  cmp     eax, ecx
.text:0000D6B3                  jz      short OK
.text:0000D6B5                  mov     al, byte_4016D1[ebx]
.text:0000D6BB                  inc     ebx
.text:0000D6BC                  test    al, al
.text:0000D6BE                  jnz     short loc_D67B
.text:0000D6C0
.text:0000D6C0 loc_D6C0: ; CODE XREF: sys_info+C1j
.text:0000D6C0                  inc     ds:ctl_port
.text:0000D6C6                  xor     eax, eax
.text:0000D6C8                  mov     al, ds:ctl_port
.text:0000D6CD                  cmp     eax, edi
.text:0000D6CF                  jle     short loc_D652
.text:0000D6D1
.text:0000D6D1 loc_D6D1: ; CODE XREF: sys_info+98j
.text:0000D6D1                ; sys_info+B6j
.text:0000D6D1                  mov     edx, [ebp+var_8]
.text:0000D6D4                  inc     edx
.text:0000D6D5                  mov     [ebp+var_8], edx
.text:0000D6D8                  cmp     edx, 3
.text:0000D6DB                  jle     loc_D641
.text:0000D6E1
.text:0000D6E1 loc_D6E1: ; CODE XREF: sys_info+16j
.text:0000D6E1                ; sys_info+51j ...
.text:0000D6E1                  pop     ebx
.text:0000D6E2                  pop     edi
.text:0000D6E3                  mov     esp, ebp
```

```
.text:0000D6E5                  pop     ebp
.text:0000D6E6                  retn
.text:0000D6E8 OK:          ; CODE XREF: sys_info+F0j
.text:0000D6E8              ; sys_info+FBj
.text:0000D6E8                  mov     al, _C_and_B[ebx]
.text:0000D6EE                  pop     ebx
.text:0000D6EF                  pop     edi
.text:0000D6F0                  mov     ds:ctl_model, al
.text:0000D6F5                  mov     esp, ebp
.text:0000D6F7                  pop     ebp
.text:0000D6F8                  retn
.text:0000D6F8 sys_info         endp
```

"3C" and "3E" are sounds familiar: there was a Sentinel Pro dongle by Rainbow with no memory, providing only one crypto-hashing secret function.

A short description about what hash function is, read here: 54.

But let's back to the program. So the program can only check the presence or absence dongle connected. No other information can be written to such dongle with no memory. Two-character codes are commands (we can see how commands are handled in `SSQC()` function) and all other strings are hashed inside the dongle transforming into 16-bit number. The algorithm was secret, so it was not possible to write driver replacement or to remake dongle hardware emulating it perfectly. However, it was always possible to intercept all accesses to it and to find what constants the hash function results compared to. Needless to say it is possible to build a robust software copy protection scheme based on secret cryptographical hash-function: let it to encrypt/decrypt data files your software dealing with.

But let's back to the code.

Codes 51/52/53 are used for LPT printer port selection. 3x/4x is for "family" selection (that's how Sentinel Pro dongles are differentiated from each other: more than one dongle can be connected to LPT port).

The only non-2-character string passed to the hashing function is "0123456789". Then, the result is compared against the set of valid results. If it is correct, 0xC or 0xB is to be written into global variable `ctl_model`.

Another text string to be passed is "PRESS ANY KEY TO CONTINUE: ", but the result is not checked. I don't know why, probably by mistake. (What a strange feeling: to reveal bugs in such ancient software.)

Let's see where the value from the global variable `ctl_mode` is used.

One of such places is:

```
.text:0000D708 prep_sys proc near ; CODE XREF: init_sys+46Ap
.text:0000D708
.text:0000D708 var_14   = dword ptr -14h
.text:0000D708 var_10   = byte ptr -10h
.text:0000D708 var_8    = dword ptr -8
```

```
.text:0000D708 var_2    = word ptr -2
.text:0000D708
.text:0000D708          push    ebp
.text:0000D709          mov     eax, ds:net_env
.text:0000D70E          mov     ebp, esp
.text:0000D710          sub     esp, 1Ch
.text:0000D713          test    eax, eax
.text:0000D715          jnz     short loc_D734
.text:0000D717          mov     al, ds:ctl_model
.text:0000D71C          test    al, al
.text:0000D71E          jnz     short loc_D77E
.text:0000D720          mov     [ebp+var_8], offset ↙
    ↳ aIeCvulnvvOkgT_ ; "Ie-cvulnvV\\\bOKG]T_"
.text:0000D727          mov     edx, 7
.text:0000D72C          jmp     loc_D7E7

...

.text:0000D7E7 loc_D7E7: ; CODE XREF: prep_sys+24j
.text:0000D7E7           ; prep_sys+33j
.text:0000D7E7          push    edx
.text:0000D7E8          mov     edx, [ebp+var_8]
.text:0000D7EB          push    20h
.text:0000D7ED          push    edx
.text:0000D7EE          push    16h
.text:0000D7F0          call    err_warn
.text:0000D7F5          push    offset station_sem
.text:0000D7FA          call    ClosSem
.text:0000D7FF          call    startup_err
```

If it is 0, an encrypted error message is passed into decryption routine and printed. Error strings decryption routine is seems simple xoring:

```
.text:0000A43C err_warn         proc near               ; CODE ↙
    ↳ XREF: prep_sys+E8p
.text:0000A43C                                          ; ↙
    ↳ prep_sys2+2Fp ...
.text:0000A43C
.text:0000A43C var_55           = byte ptr -55h
.text:0000A43C var_54           = byte ptr -54h
.text:0000A43C arg_0            = dword ptr  8
.text:0000A43C arg_4            = dword ptr  0Ch
.text:0000A43C arg_8            = dword ptr  10h
.text:0000A43C arg_C            = dword ptr  14h
```

```
.text:0000A43C
.text:0000A43C                  push    ebp
.text:0000A43D                  mov     ebp, esp
.text:0000A43F                  sub     esp, 54h
.text:0000A442                  push    edi
.text:0000A443                  mov     ecx, [ebp+arg_8]
.text:0000A446                  xor     edi, edi
.text:0000A448                  test    ecx, ecx
.text:0000A44A                  push    esi
.text:0000A44B                  jle     short loc_A466
.text:0000A44D                  mov     esi, [ebp+arg_C] ; key
.text:0000A450                  mov     edx, [ebp+arg_4] ; string
.text:0000A453
.text:0000A453 loc_A453:                                ; CODE ↙
    ↳ XREF: err_warn+28j
.text:0000A453                  xor     eax, eax
.text:0000A455                  mov     al, [edx+edi]
.text:0000A458                  xor     eax, esi
.text:0000A45A                  add     esi, 3
.text:0000A45D                  inc     edi
.text:0000A45E                  cmp     edi, ecx
.text:0000A460                  mov     [ebp+edi+var_55], al
.text:0000A464                  jl      short loc_A453
.text:0000A466
.text:0000A466 loc_A466:                                ; CODE ↙
    ↳ XREF: err_warn+Fj
.text:0000A466                  mov     [ebp+edi+var_54], 0
.text:0000A46B                  mov     eax, [ebp+arg_0]
.text:0000A46E                  cmp     eax, 18h
.text:0000A473                  jnz     short loc_A49C
.text:0000A475                  lea     eax, [ebp+var_54]
.text:0000A478                  push    eax
.text:0000A479                  call    status_line
.text:0000A47E                  add     esp, 4
.text:0000A481
.text:0000A481 loc_A481:                                ; CODE ↙
    ↳ XREF: err_warn+72j
.text:0000A481                  push    50h
.text:0000A483                  push    0
.text:0000A485                  lea     eax, [ebp+var_54]
.text:0000A488                  push    eax
.text:0000A489                  call    memset
.text:0000A48E                  call    pcv_refresh
```

```
.text:0000A493                  add     esp, 0Ch
.text:0000A496                  pop     esi
.text:0000A497                  pop     edi
.text:0000A498                  mov     esp, ebp
.text:0000A49A                  pop     ebp
.text:0000A49B                  retn
.text:0000A49C
.text:0000A49C loc_A49C:                                ; CODE ↙
    ↳ XREF: err_warn+37j
.text:0000A49C                  push    0
.text:0000A49E                  lea     eax, [ebp+var_54]
.text:0000A4A1                  mov     edx, [ebp+arg_0]
.text:0000A4A4                  push    edx
.text:0000A4A5                  push    eax
.text:0000A4A6                  call    pcv_lputs
.text:0000A4AB                  add     esp, 0Ch
.text:0000A4AE                  jmp     short loc_A481
.text:0000A4AE err_warn         endp
```

That's why I was unable to find error messages in the executable files, because they are encrypted, this is popular practice.

Another call to `SSQ()` hashing function passes "offln" string to it and comparing result with `0xFE81` and `0x12A9`. If it not so, it deals with some `timer()` function (maybe waiting for poorly connected dongle to be reconnected and check again?) and then decrypt another error message to dump.

```
.text:0000DA55 loc_DA55:                                ; CODE ↙
    ↳ XREF: sync_sys+24Cj
.text:0000DA55                  push    offset aOffln   ; "offln"
.text:0000DA5A                  call    SSQ
.text:0000DA5F                  add     esp, 4
.text:0000DA62                  mov     dl, [ebx]
.text:0000DA64                  mov     esi, eax
.text:0000DA66                  cmp     dl, 0Bh
.text:0000DA69                  jnz     short loc_DA83
.text:0000DA6B                  cmp     esi, 0FE81h
.text:0000DA71                  jz      OK
.text:0000DA77                  cmp     esi, 0FFFFF8EFh
.text:0000DA7D                  jz      OK
.text:0000DA83
.text:0000DA83 loc_DA83:                                ; CODE ↙
    ↳ XREF: sync_sys+201j
.text:0000DA83                  mov     cl, [ebx]
.text:0000DA85                  cmp     cl, 0Ch
```

```
.text:0000DA88                  jnz     short loc_DA9F
.text:0000DA8A                  cmp     esi, 12A9h
.text:0000DA90                  jz      OK
.text:0000DA96                  cmp     esi, 0FFFFFFF5h
.text:0000DA99                  jz      OK
.text:0000DA9F
.text:0000DA9F loc_DA9F:                                ; CODE ↙
    ↳ XREF: sync_sys+220j
.text:0000DA9F                  mov     eax, [ebp+var_18]
.text:0000DAA2                  test    eax, eax
.text:0000DAA4                  jz      short loc_DAB0
.text:0000DAA6                  push    24h
.text:0000DAA8                  call    timer
.text:0000DAAD                  add     esp, 4
.text:0000DAB0
.text:0000DAB0 loc_DAB0:                                ; CODE ↙
    ↳ XREF: sync_sys+23Cj
.text:0000DAB0                  inc     edi
.text:0000DAB1                  cmp     edi, 3
.text:0000DAB4                  jle     short loc_DA55
.text:0000DAB6                  mov     eax, ds:net_env
.text:0000DABB                  test    eax, eax
.text:0000DABD                  jz      short error

...

.text:0000DAF7 error:                                   ; CODE ↙
    ↳ XREF: sync_sys+255j
.text:0000DAF7                                          ; ↙
    ↳ sync_sys+274j ...
.text:0000DAF7                  mov     [ebp+var_8], offset ↙
    ↳ encrypted_error_message2
.text:0000DAFE                  mov     [ebp+var_C], 17h ; ↙
    ↳ decrypting key
.text:0000DB05                  jmp     decrypt_end_print_message

...

; this name I gave to label:
.text:0000D9B6 decrypt_end_print_message:               ; CODE ↙
    ↳ XREF: sync_sys+29Dj
.text:0000D9B6                                          ; ↙
    ↳ sync_sys+2ABj
```

```
.text:0000D9B6                  mov     eax, [ebp+var_18]
.text:0000D9B9                  test    eax, eax
.text:0000D9BB                  jnz     short loc_D9FB
.text:0000D9BD                  mov     edx, [ebp+var_C] ; key
.text:0000D9C0                  mov     ecx, [ebp+var_8] ; string
.text:0000D9C3                  push    edx
.text:0000D9C4                  push    20h
.text:0000D9C6                  push    ecx
.text:0000D9C7                  push    18h
.text:0000D9C9                  call    err_warn
.text:0000D9CE                  push    0Fh
.text:0000D9D0                  push    190h
.text:0000D9D5                  call    sound
.text:0000D9DA                  mov     [ebp+var_18], 1
.text:0000D9E1                  add     esp, 18h
.text:0000D9E4                  call    pcv_kbhit
.text:0000D9E9                  test    eax, eax
.text:0000D9EB                  jz      short loc_D9FB

...

; this name I gave to label:
.data:00401736 encrypted_error_message2 db 74h, 72h, 78h, 43h, ↙
    ↳ 48h, 6, 5Ah, 49h, 4Ch, 2 dup(47h)
.data:00401736                  db 51h, 4Fh, 47h, 61h, 20h, 22h, ↙
    ↳ 3Ch, 24h, 33h, 36h, 76h
.data:00401736                  db 3Ah, 33h, 31h, 0Ch, 0, 0Bh, 1↙
    ↳ Fh, 7, 1Eh, 1Ah
```

Dongle bypassing is pretty straightforward: just patch all jumps after `CMP` the relevant instructions.

Another option is to write our own SCO OpenServer driver.

55.2.1 Decrypting error messages

By the way, we can also try to decrypt all error messages. The algorithm, locating in `err_warn()` function is very simple, indeed:

Listing 55.1: Decrypting function

```
.text:0000A44D                  mov     esi, [ebp+arg_C] ; key
.text:0000A450                  mov     edx, [ebp+arg_4] ; string
.text:0000A453 loc_A453:
.text:0000A453                  xor     eax, eax
```

```
.text:0000A455                  mov     al, [edx+edi] ; load ↙
    ↳ encrypted byte
.text:0000A458                  xor     eax, esi      ; decrypt ↙
    ↳ it
.text:0000A45A                  add     esi, 3        ; change ↙
    ↳ key for the next byte
.text:0000A45D                  inc     edi
.text:0000A45E                  cmp     edi, ecx
.text:0000A460                  mov     [ebp+edi+var_55], al
.text:0000A464                  jl      short loc_A453
```

As we can see, not just string supplied to the decrypting function, but also the key:

```
.text:0000DAF7 error:                                   ; CODE ↙
    ↳ XREF: sync_sys+255j
.text:0000DAF7                                          ; ↙
    ↳ sync_sys+274j ...
.text:0000DAF7                  mov     [ebp+var_8], offset ↙
    ↳ encrypted_error_message2
.text:0000DAFE                  mov     [ebp+var_C], 17h ; ↙
    ↳ decrypting key
.text:0000DB05                  jmp     decrypt_end_print_message

...

; this name I gave to label:
.text:0000D9B6 decrypt_end_print_message:               ; CODE ↙
    ↳ XREF: sync_sys+29Dj
.text:0000D9B6                                          ; ↙
    ↳ sync_sys+2ABj
.text:0000D9B6                  mov     eax, [ebp+var_18]
.text:0000D9B9                  test    eax, eax
.text:0000D9BB                  jnz     short loc_D9FB
.text:0000D9BD                  mov     edx, [ebp+var_C] ; key
.text:0000D9C0                  mov     ecx, [ebp+var_8] ; string
.text:0000D9C3                  push    edx
.text:0000D9C4                  push    20h
.text:0000D9C6                  push    ecx
.text:0000D9C7                  push    18h
.text:0000D9C9                  call    err_warn
```

The algorithm is simple xoring: each byte is xored with a key, but key is increased by 3 after processing of each byte.

I wrote a simple Python script to check my insights:

Listing 55.2: Python 3.x

```
#!/usr/bin/python
import sys

msg=[0x74, 0x72, 0x78, 0x43, 0x48, 0x6, 0x5A, 0x49, 0x4C, 0x47, ↙
    ↳ 0x47,
0x51, 0x4F, 0x47, 0x61, 0x20, 0x22, 0x3C, 0x24, 0x33, 0x36, 0x76↙
    ↳ ,
0x3A, 0x33, 0x31, 0x0C, 0x0, 0x0B, 0x1F, 0x7, 0x1E, 0x1A]

key=0x17
tmp=key
for i in msg:
        sys.stdout.write ("%c" % (i^tmp))
        tmp=tmp+3
sys.stdout.flush()
```

And it prints: "check security device connection". So yes, this is decrypted message.

There are also other encrypted messages with corresponding keys. But needless to say that it is possible to decrypt them without keys. First, we may observe that key is byte in fact. It is because core decrypting instruction (`XOR`) works on byte level. Key is located in `ESI` register, but only byte part of `ESI` is used. Hence, key may be greater than 255, but its value will always be rounded.

As a consequence, we can just try brute-force, trying all possible keys in 0..255 range. We will also skip messages containing unprintable characters.

Listing 55.3: Python 3.x

```
#!/usr/bin/python
import sys, curses.ascii

msgs=[
[0x74, 0x72, 0x78, 0x43, 0x48, 0x6, 0x5A, 0x49, 0x4C, 0x47, 0x47↙
    ↳ ,
0x51, 0x4F, 0x47, 0x61, 0x20, 0x22, 0x3C, 0x24, 0x33, 0x36, 0x76↙
    ↳ ,
0x3A, 0x33, 0x31, 0x0C, 0x0, 0x0B, 0x1F, 0x7, 0x1E, 0x1A],

[0x49, 0x65, 0x2D, 0x63, 0x76, 0x75, 0x6C, 0x6E, 0x76, 0x56, 0↙
    ↳ x5C,
8, 0x4F, 0x4B, 0x47, 0x5D, 0x54, 0x5F, 0x1D, 0x26, 0x2C, 0x33,
0x27, 0x28, 0x6F, 0x72, 0x75, 0x78, 0x7B, 0x7E, 0x41, 0x44],
```

```
[0x45, 0x61, 0x31, 0x67, 0x72, 0x79, 0x68, 0x52, 0x4A, 0x52, 0↙
    ↳ x50,
0x0C, 0x4B, 0x57, 0x43, 0x51, 0x58, 0x5B, 0x61, 0x37, 0x33, 0x2B↙
    ↳ ,
0x39, 0x39, 0x3C, 0x38, 0x79, 0x3A, 0x30, 0x17, 0x0B, 0x0C],

[0x40, 0x64, 0x79, 0x75, 0x7F, 0x6F, 0x0, 0x4C, 0x40, 0x9, 0x4D,↙
    ↳  0x5A,
0x46, 0x5D, 0x57, 0x49, 0x57, 0x3B, 0x21, 0x23, 0x6A, 0x38, 0x23↙
    ↳ ,
0x36, 0x24, 0x2A, 0x7C, 0x3A, 0x1A, 0x6, 0x0D, 0x0E, 0x0A, 0x14,
0x10],

[0x72, 0x7C, 0x72, 0x79, 0x76, 0x0,
0x50, 0x43, 0x4A, 0x59, 0x5D, 0x5B, 0x41, 0x41, 0x1B, 0x5A,
0x24, 0x32, 0x2E, 0x29, 0x28, 0x70, 0x20, 0x22, 0x38, 0x28, 0x36↙
    ↳ ,
0x0D, 0x0B, 0x48, 0x4B, 0x4E]]


def is_string_printable(s):
    return all(list(map(lambda x: curses.ascii.isprint(x), s)))

cnt=1
for msg in msgs:
        print ("message #%d" % cnt)
        for key in range(0,256):
                result=[]
                tmp=key
                for i in msg:
                        result.append (i^tmp)
                        tmp=tmp+3
                if is_string_printable (result):
                        print ("key=", key, "value=", "".join(↙
    ↳ list(map(chr, result))))
        cnt=cnt+1
```

And we getting:

Listing 55.4: Results

```
message #1
key= 20 value= ‘eb^h%|‘‘hudw|_af{n~f%ljmSbnwlpk
key= 21 value= ajc]i"}cawtgv{^bgto}g"millcmvkqh
key= 22 value= bkd\j#rbbvsfuz!cduh|d#bhomdlujni
```

```
key= 23 value= check security device connection
key= 24 value= lifbl!pd|tqhsx#ejwjbb!‘nQofbshlo
message #2
key= 7 value= No security device found
key= 8 value= An#rbbvsVuz!cduhld#ghtme?!#!’!#!
message #3
key= 7 value= Bk<waoqNUpu$‘yreoa\wpmpusj,bkIjh
key= 8 value= Mj?vfnrOjqv%gxqd‘‘_vwlstlk/clHii
key= 9 value= Lm>ugasLkvw&fgpgag^uvcrwml.‘mwhj
key= 10 value= Ol!td‘tMhwx’efwfbf!tubuvnm!anvok
key= 11 value= No security device station found
key= 12 value= In#rjbvsnuz!{duhdd#r{‘whho#gPtme
message #4
key= 14 value= Number of authorized users exceeded
key= 15 value= Ovlmdq!hg#‘juknuhydk!vrbsp!Zy‘dbefe
message #5
key= 17 value= check security device station
key= 18 value= ‘ijbh!td‘tmhwx’efwfbf!tubuVnm!’!
```

There are some garbage, but we can quickly find English-language messages!

By the way, since algorithm is simple xoring encryption, the very same function can be used for encrypting messages. If we need, we can encrypt our own messages, and patch the program by inserting them.

55.3 Example #3: MS-DOS

Another very old software for MS-DOS from 1995 also developed by a company disappeared long time ago.

In the pre-DOS extenders era, all the software for MS-DOS were mostly rely on 16-bit 8086 or 80286 CPUs, so en masse code was 16-bit. 16-bit code is mostly same as you already saw in this book, but all registers are 16-bit and there are less number of instructions available.

MS-DOS environment has no any system drivers, any program may deal with bare hardware via ports, so here you may see `OUT`/`IN` instructions, which are mostly present in drivers in our times (it is impossible to access ports directly in user mode in all modern OS).

Given that, the MS-DOS program working with a dongle should access LPT printer port directly. So we can just search for such instructions. And yes, here it is:

```
seg030:0034             out_port proc far ; CODE XREF: sent_pro↙
    ↳ +22p
seg030:0034                               ; sent_pro+2Ap ...
seg030:0034
```

```
seg030:0034              arg_0    = byte ptr  6
seg030:0034
seg030:0034 55                    push    bp
seg030:0035 8B EC                 mov     bp, sp
seg030:0037 8B 16 7E E7           mov     dx, _out_port ; 0x378
seg030:003B 8A 46 06              mov     al, [bp+arg_0]
seg030:003E EE                    out     dx, al
seg030:003F 5D                    pop     bp
seg030:0040 CB                    retf
seg030:0040              out_port endp
```

(All label names in this example were given by me).
`out_port()` is referenced only in one function:

```
seg030:0041              sent_pro proc far ; CODE XREF: ↙
    ↳ check_dongle+34p
seg030:0041
seg030:0041              var_3    = byte ptr -3
seg030:0041              var_2    = word ptr -2
seg030:0041              arg_0    = dword ptr  6
seg030:0041
seg030:0041 C8 04 00 00           enter   4, 0
seg030:0045 56                    push    si
seg030:0046 57                    push    di
seg030:0047 8B 16 82 E7           mov     dx, _in_port_1 ; 0x37A
seg030:004B EC                    in      al, dx
seg030:004C 8A D8                 mov     bl, al
seg030:004E 80 E3 FE              and     bl, 0FEh
seg030:0051 80 CB 04              or      bl, 4
seg030:0054 8A C3                 mov     al, bl
seg030:0056 88 46 FD              mov     [bp+var_3], al
seg030:0059 80 E3 1F              and     bl, 1Fh
seg030:005C 8A C3                 mov     al, bl
seg030:005E EE                    out     dx, al
seg030:005F 68 FF 00              push    0FFh
seg030:0062 0E                    push    cs
seg030:0063 E8 CE FF              call    near ptr out_port
seg030:0066 59                    pop     cx
seg030:0067 68 D3 00              push    0D3h
seg030:006A 0E                    push    cs
seg030:006B E8 C6 FF              call    near ptr out_port
seg030:006E 59                    pop     cx
seg030:006F 33 F6                 xor     si, si
seg030:0071 EB 01                 jmp     short loc_359D4
```

```
seg030:0073
seg030:0073                loc_359D3: ; CODE XREF: sent_pro+37j
seg030:0073 46                        inc     si
seg030:0074
seg030:0074                loc_359D4: ; CODE XREF: sent_pro+30j
seg030:0074 81 FE 96 00               cmp     si, 96h
seg030:0078 7C F9                     jl      short loc_359D3
seg030:007A 68 C3 00                  push    0C3h
seg030:007D 0E                        push    cs
seg030:007E E8 B3 FF                  call    near ptr out_port
seg030:0081 59                        pop     cx
seg030:0082 68 C7 00                  push    0C7h
seg030:0085 0E                        push    cs
seg030:0086 E8 AB FF                  call    near ptr out_port
seg030:0089 59                        pop     cx
seg030:008A 68 D3 00                  push    0D3h
seg030:008D 0E                        push    cs
seg030:008E E8 A3 FF                  call    near ptr out_port
seg030:0091 59                        pop     cx
seg030:0092 68 C3 00                  push    0C3h
seg030:0095 0E                        push    cs
seg030:0096 E8 9B FF                  call    near ptr out_port
seg030:0099 59                        pop     cx
seg030:009A 68 C7 00                  push    0C7h
seg030:009D 0E                        push    cs
seg030:009E E8 93 FF                  call    near ptr out_port
seg030:00A1 59                        pop     cx
seg030:00A2 68 D3 00                  push    0D3h
seg030:00A5 0E                        push    cs
seg030:00A6 E8 8B FF                  call    near ptr out_port
seg030:00A9 59                        pop     cx
seg030:00AA BF FF FF                  mov     di, 0FFFFh
seg030:00AD EB 40                     jmp     short loc_35A4F
seg030:00AF
seg030:00AF                loc_35A0F: ; CODE XREF: sent_pro+BDj
seg030:00AF BE 04 00                  mov     si, 4
seg030:00B2
seg030:00B2                loc_35A12: ; CODE XREF: sent_pro+ACj
seg030:00B2 D1 E7                     shl     di, 1
seg030:00B4 8B 16 80 E7               mov     dx, _in_port_2 ; 0x379
seg030:00B8 EC                        in      al, dx
seg030:00B9 A8 80                     test    al, 80h
seg030:00BB 75 03                     jnz     short loc_35A20
```

```
seg030:00BD 83 CF 01            or      di, 1
seg030:00C0
seg030:00C0             loc_35A20: ; CODE XREF: sent_pro+7Aj
seg030:00C0 F7 46 FE 08+        test    [bp+var_2], 8
seg030:00C5 74 05               jz      short loc_35A2C
seg030:00C7 68 D7 00            push    0D7h ; '+'
seg030:00CA EB 0B               jmp     short loc_35A37
seg030:00CC
seg030:00CC             loc_35A2C: ; CODE XREF: sent_pro+84j
seg030:00CC 68 C3 00            push    0C3h
seg030:00CF 0E                  push    cs
seg030:00D0 E8 61 FF            call    near ptr out_port
seg030:00D3 59                  pop     cx
seg030:00D4 68 C7 00            push    0C7h
seg030:00D7
seg030:00D7             loc_35A37: ; CODE XREF: sent_pro+89j
seg030:00D7 0E                  push    cs
seg030:00D8 E8 59 FF            call    near ptr out_port
seg030:00DB 59                  pop     cx
seg030:00DC 68 D3 00            push    0D3h
seg030:00DF 0E                  push    cs
seg030:00E0 E8 51 FF            call    near ptr out_port
seg030:00E3 59                  pop     cx
seg030:00E4 8B 46 FE            mov     ax, [bp+var_2]
seg030:00E7 D1 E0               shl     ax, 1
seg030:00E9 89 46 FE            mov     [bp+var_2], ax
seg030:00EC 4E                  dec     si
seg030:00ED 75 C3               jnz     short loc_35A12
seg030:00EF
seg030:00EF             loc_35A4F: ; CODE XREF: sent_pro+6Cj
seg030:00EF C4 5E 06            les     bx, [bp+arg_0]
seg030:00F2 FF 46 06            inc     word ptr [bp+arg_0]
seg030:00F5 26 8A 07            mov     al, es:[bx]
seg030:00F8 98                  cbw
seg030:00F9 89 46 FE            mov     [bp+var_2], ax
seg030:00FC 0B C0               or      ax, ax
seg030:00FE 75 AF               jnz     short loc_35A0F
seg030:0100 68 FF 00            push    0FFh
seg030:0103 0E                  push    cs
seg030:0104 E8 2D FF            call    near ptr out_port
seg030:0107 59                  pop     cx
seg030:0108 8B 16 82 E7         mov     dx, _in_port_1 ; 0x37A
seg030:010C EC                  in      al, dx
```

```
seg030:010D 8A C8                mov     cl, al
seg030:010F 80 E1 5F             and     cl, 5Fh
seg030:0112 8A C1                mov     al, cl
seg030:0114 EE                   out     dx, al
seg030:0115 EC                   in      al, dx
seg030:0116 8A C8                mov     cl, al
seg030:0118 F6 C1 20             test    cl, 20h
seg030:011B 74 08                jz      short loc_35A85
seg030:011D 8A 5E FD             mov     bl, [bp+var_3]
seg030:0120 80 E3 DF             and     bl, 0DFh
seg030:0123 EB 03                jmp     short loc_35A88
seg030:0125
seg030:0125              loc_35A85: ; CODE XREF: sent_pro+DAj
seg030:0125 8A 5E FD             mov     bl, [bp+var_3]
seg030:0128
seg030:0128              loc_35A88: ; CODE XREF: sent_pro+E2j
seg030:0128 F6 C1 80             test    cl, 80h
seg030:012B 74 03                jz      short loc_35A90
seg030:012D 80 E3 7F             and     bl, 7Fh
seg030:0130
seg030:0130              loc_35A90: ; CODE XREF: sent_pro+EAj
seg030:0130 8B 16 82 E7          mov     dx, _in_port_1 ; 0x37A
seg030:0134 8A C3                mov     al, bl
seg030:0136 EE                   out     dx, al
seg030:0137 8B C7                mov     ax, di
seg030:0139 5F                   pop     di
seg030:013A 5E                   pop     si
seg030:013B C9                   leave
seg030:013C CB                   retf
seg030:013C              sent_pro endp
```

It is also Sentinel Pro "hashing" dongle as in the previous example. I figured out its type by noticing that a text strings are also passed here and 16 bit values are also returned and compared with others.

So that is how Sentinel Pro is accessed via ports. Output port address is usually 0x378, i.e., printer port, the data to the old printers in pre-USB era were passed to it. The port is one-directional, because when it was developed, no one can imagined someone will need to transfer information from the printer [4]. The only way to get information from the printer, is a status register on port 0x379, it contain such bits as "paper out", "ack", "busy" —thus printer may signal to the host computer that it is ready or not and if a paper present in it. So the dongle return information from one of these bits, by one bit at each iteration.

[4] If to consider Centronics only. Following IEEE 1284 standard allows to transfer information from the printer.

_in_port_2 has address of status word (0x379) and _in_port_1 has control register address (0x37A).

It seems, the dongle return information via "busy" flag at seg030:00B9: each bit is stored in the DI register, later returned at the function end.

What all these bytes sent to output port mean? I don't know. Probably commands to the dongle. But generally speaking, it is not necessary to know: it is easy to solve our task without that knowledge.

Here is a dongle checking routine:

```
00000000 struct_0        struc ; (sizeof=0x1B)
00000000 field_0         db 25 dup(?)            ; string(C)
00000019 _A              dw ?
0000001B struct_0        ends

dseg:3CBC 61 63 72 75+_Q   struct_0 <'hello', 01122h>
dseg:3CBC 6E 00 00 00+       ; DATA XREF: check_dongle+2Eo

... skipped ...

dseg:3E00 63 6F 66 66+     struct_0 <'coffee', 7EB7h>
dseg:3E1B 64 6F 67 00+     struct_0 <'dog', 0FFADh>
dseg:3E36 63 61 74 00+     struct_0 <'cat', 0FF5Fh>
dseg:3E51 70 61 70 65+     struct_0 <'paper', 0FFDFh>
dseg:3E6C 63 6F 6B 65+     struct_0 <'coke', 0F568h>
dseg:3E87 63 6C 6F 63+     struct_0 <'clock', 55EAh>
dseg:3EA2 64 69 72 00+     struct_0 <'dir', 0FFAEh>
dseg:3EBD 63 6F 70 79+     struct_0 <'copy', 0F557h>


seg030:0145              check_dongle proc far ; CODE XREF: ↙
    ↳ sub_3771D+3EP
seg030:0145
seg030:0145              var_6 = dword ptr -6
seg030:0145              var_2 = word ptr -2
seg030:0145
seg030:0145 C8 06 00 00        enter   6, 0
seg030:0149 56                 push    si
seg030:014A 66 6A 00           push    large 0         ; newtime
seg030:014D 6A 00              push    0               ; cmd
seg030:014F 9A C1 18 00+       call    _biostime
seg030:0154 52                 push    dx
seg030:0155 50                 push    ax
seg030:0156 66 58              pop     eax
seg030:0158 83 C4 06           add     sp, 6
```

```
seg030:015B 66 89 46 FA       mov     [bp+var_6], eax
seg030:015F 66 3B 06 D8+      cmp     eax, _expiration
seg030:0164 7E 44             jle     short loc_35B0A
seg030:0166 6A 14             push    14h
seg030:0168 90                nop
seg030:0169 0E                push    cs
seg030:016A E8 52 00          call    near ptr get_rand
seg030:016D 59                pop     cx
seg030:016E 8B F0             mov     si, ax
seg030:0170 6B C0 1B          imul    ax, 1Bh
seg030:0173 05 BC 3C          add     ax, offset _Q
seg030:0176 1E                push    ds
seg030:0177 50                push    ax
seg030:0178 0E                push    cs
seg030:0179 E8 C5 FE          call    near ptr sent_pro
seg030:017C 83 C4 04          add     sp, 4
seg030:017F 89 46 FE          mov     [bp+var_2], ax
seg030:0182 8B C6             mov     ax, si
seg030:0184 6B C0 12          imul    ax, 18
seg030:0187 66 0F BF C0       movsx   eax, ax
seg030:018B 66 8B 56 FA       mov     edx, [bp+var_6]
seg030:018F 66 03 D0          add     edx, eax
seg030:0192 66 89 16 D8+      mov     _expiration, edx
seg030:0197 8B DE             mov     bx, si
seg030:0199 6B DB 1B          imul    bx, 27
seg030:019C 8B 87 D5 3C       mov     ax, _Q._A[bx]
seg030:01A0 3B 46 FE          cmp     ax, [bp+var_2]
seg030:01A3 74 05             jz      short loc_35B0A
seg030:01A5 B8 01 00          mov     ax, 1
seg030:01A8 EB 02             jmp     short loc_35B0C
seg030:01AA
seg030:01AA             loc_35B0A: ; CODE XREF: check_dongle+1Fj
seg030:01AA                        ; check_dongle+5Ej
seg030:01AA 33 C0               xor     ax, ax
seg030:01AC
seg030:01AC             loc_35B0C: ; CODE XREF: check_dongle+63j
seg030:01AC 5E                  pop     si
seg030:01AD C9                  leave
seg030:01AE CB                  retf
seg030:01AE             check_dongle endp
```

Since the routine may be called too frequently, e.g., before each important software feature executing, and the dongle accessing process is generally slow (because

of slow printer port and also slow MCU[5] in the dongle), so they probably added a way to skip dongle checking too often, using checking current time in `biostime()` function.

`get_rand()` function uses standard C function:

```
seg030:01BF             get_rand proc far ; CODE XREF: ↙
    ↳ check_dongle+25p
seg030:01BF
seg030:01BF             arg_0    = word ptr  6
seg030:01BF
seg030:01BF 55                   push    bp
seg030:01C0 8B EC                mov     bp, sp
seg030:01C2 9A 3D 21 00+         call    _rand
seg030:01C7 66 0F BF C0          movsx   eax, ax
seg030:01CB 66 0F BF 56+         movsx   edx, [bp+arg_0]
seg030:01D0 66 0F AF C2          imul    eax, edx
seg030:01D4 66 BB 00 80+         mov     ebx, 8000h
seg030:01DA 66 99                cdq
seg030:01DC 66 F7 FB             idiv    ebx
seg030:01DF 5D                   pop     bp
seg030:01E0 CB                   retf
seg030:01E0             get_rand endp
```

So the text string is selected randomly, passed into dongle, and then the result of hashing is compared with correct value.

Text strings are seems to be chosen randomly as well.

And that is how the main dongle checking function is called:

```
seg033:087B 9A 45 01 96+   call    check_dongle
seg033:0880 0B C0          or      ax, ax
seg033:0882 74 62          jz      short OK
seg033:0884 83 3E 60 42+   cmp     word_620E0, 0
seg033:0889 75 5B          jnz     short OK
seg033:088B FF 06 60 42    inc     word_620E0
seg033:088F 1E             push    ds
seg033:0890 68 22 44       push    offset aTrupcRequiresA ; "↙
    ↳ This Software Requires a Software Lock\n"
seg033:0893 1E             push    ds
seg033:0894 68 60 E9       push    offset byte_6C7E0 ; dest
seg033:0897 9A 79 65 00+   call    _strcpy
seg033:089C 83 C4 08       add     sp, 8
seg033:089F 1E             push    ds
seg033:08A0 68 42 44       push    offset aPleaseContactA ; "↙
    ↳ Please Contact ..."
```

[5] Microcontroller unit

```
seg033:08A3 1E             push    ds
seg033:08A4 68 60 E9       push    offset byte_6C7E0 ; dest
seg033:08A7 9A CD 64 00+   call    _strcat
```

Dongle bypassing is easy, just force the `check_dongle()` function to always return 0.

For example, by inserting this code at its beginning:

```
mov ax,0
retf
```

Observant reader might recall that `strcpy()` C function usually requires two pointers in arguments, but we saw how 4 values are passed:

```
seg033:088F 1E                          push    ds
seg033:0890 68 22 44                    push    offset ↙
    ↳ aTrupcRequiresA ; "This Software Requires a Software Lock\↙
    ↳ n"
seg033:0893 1E                          push    ds
seg033:0894 68 60 E9                    push    offset ↙
    ↳ byte_6C7E0 ; dest
seg033:0897 9A 79 65 00+                call    _strcpy
seg033:089C 83 C4 08                    add     sp, 8
```

Read more about it here: 67.

So as you may see, `strcpy()` and any other function taking pointer(s) in arguments, works with 16-bit pairs.

Let's back to our example. `DS` is currently set to data segment located in the executable, that is where the text string is stored.

In the `sent_pro()` function, each byte of string is loaded at `seg030:00EF`: the `LES` instruction loads ES:BX pair simultaneously from the passed argument. The `MOV` at `seg030:00F5` loads the byte from the memory to which ES:BX pair points.

At `seg030:00F2` only 16-bit word is incremented, not segment value. This means, the string passed to the function cannot be located on two data segments boundaries.

Chapter 56

"QR9": Rubik's cube inspired amateur crypto-algorithm

Sometimes amateur cryptosystems appear to be pretty bizarre.

I was asked to reverse engineer an amateur cryptoalgorithm of some data crypting utility, source code of which was lost[1].

Here is also IDA exported listing from original crypting utility:

```
.text:00541000 set_bit         proc near               ; CODE ↙
    ↳ XREF: rotate1+42
.text:00541000                                         ; rotate2↙
    ↳ +42 ...
.text:00541000
.text:00541000 arg_0           = dword ptr  4
.text:00541000 arg_4           = dword ptr  8
.text:00541000 arg_8           = dword ptr  0Ch
.text:00541000 arg_C           = byte ptr  10h
.text:00541000
.text:00541000                 mov     al, [esp+arg_C]
.text:00541004                 mov     ecx, [esp+arg_8]
.text:00541008                 push    esi
.text:00541009                 mov     esi, [esp+4+arg_0]
.text:0054100D                 test    al, al
.text:0054100F                 mov     eax, [esp+4+arg_4]
.text:00541013                 mov     dl, 1
.text:00541015                 jz      short loc_54102B
.text:00541017                 shl     dl, cl
.text:00541019                 mov     cl, cube64[eax+esi*8]
```

[1] I also got permit from customer to publish the algorithm details

```
.text:00541020                  or      cl, dl
.text:00541022                  mov     cube64[eax+esi*8], cl
.text:00541029                  pop     esi
.text:0054102A                  retn
.text:0054102B
.text:0054102B loc_54102B:                              ; CODE ↙
    ↳ XREF: set_bit+15
.text:0054102B                  shl     dl, cl
.text:0054102D                  mov     cl, cube64[eax+esi*8]
.text:00541034                  not     dl
.text:00541036                  and     cl, dl
.text:00541038                  mov     cube64[eax+esi*8], cl
.text:0054103F                  pop     esi
.text:00541040                  retn
.text:00541040 set_bit          endp
.text:00541040
.text:00541041                  align 10h
.text:00541050
.text:00541050 ; =============== S U B R O U T I N E ↙
    ↳ =======================================
.text:00541050
.text:00541050
.text:00541050 get_bit          proc near               ; CODE ↙
    ↳ XREF: rotate1+16
.text:00541050                                          ; rotate2↙
    ↳ +16 ...
.text:00541050
.text:00541050 arg_0            = dword ptr  4
.text:00541050 arg_4            = dword ptr  8
.text:00541050 arg_8            = byte ptr  0Ch
.text:00541050
.text:00541050                  mov     eax, [esp+arg_4]
.text:00541054                  mov     ecx, [esp+arg_0]
.text:00541058                  mov     al, cube64[eax+ecx*8]
.text:0054105F                  mov     cl, [esp+arg_8]
.text:00541063                  shr     al, cl
.text:00541065                  and     al, 1
.text:00541067                  retn
.text:00541067 get_bit          endp
.text:00541067
.text:00541068                  align 10h
.text:00541070
.text:00541070 ; =============== S U B R O U T I N E ↙
```

```
    ↳ ====================================
.text:00541070
.text:00541070
.text:00541070 rotate1         proc near               ; CODE ↙
    ↳ XREF: rotate_all_with_password+8E
.text:00541070
.text:00541070 internal_array_64= byte ptr -40h
.text:00541070 arg_0           = dword ptr  4
.text:00541070
.text:00541070                 sub     esp, 40h
.text:00541073                 push    ebx
.text:00541074                 push    ebp
.text:00541075                 mov     ebp, [esp+48h+arg_0]
.text:00541079                 push    esi
.text:0054107A                 push    edi
.text:0054107B                 xor     edi, edi        ; EDI is ↙
    ↳ loop1 counter
.text:0054107D                 lea     ebx, [esp+50h+↙
    ↳ internal_array_64]
.text:00541081
.text:00541081 first_loop1_begin:                      ; CODE ↙
    ↳ XREF: rotate1+2E
.text:00541081                 xor     esi, esi        ; ESI is ↙
    ↳ loop2 counter
.text:00541083
.text:00541083 first_loop2_begin:                      ; CODE ↙
    ↳ XREF: rotate1+25
.text:00541083                 push    ebp             ; arg_0
.text:00541084                 push    esi
.text:00541085                 push    edi
.text:00541086                 call    get_bit
.text:0054108B                 add     esp, 0Ch
.text:0054108E                 mov     [ebx+esi], al   ; store ↙
    ↳ to internal array
.text:00541091                 inc     esi
.text:00541092                 cmp     esi, 8
.text:00541095                 jl      short first_loop2_begin
.text:00541097                 inc     edi
.text:00541098                 add     ebx, 8
.text:0054109B                 cmp     edi, 8
.text:0054109E                 jl      short first_loop1_begin
.text:005410A0                 lea     ebx, [esp+50h+↙
    ↳ internal_array_64]
```

```
.text:005410A4                  mov     edi, 7          ; EDI is ↙
    ↳ loop1 counter, initial state is 7
.text:005410A9
.text:005410A9 second_loop1_begin:                      ; CODE ↙
    ↳ XREF: rotate1+57
.text:005410A9                  xor     esi, esi        ; ESI is ↙
    ↳ loop2 counter
.text:005410AB
.text:005410AB second_loop2_begin:                      ; CODE ↙
    ↳ XREF: rotate1+4E
.text:005410AB                  mov     al, [ebx+esi]   ; value ↙
    ↳ from internal array
.text:005410AE                  push    eax
.text:005410AF                  push    ebp             ; arg_0
.text:005410B0                  push    edi
.text:005410B1                  push    esi
.text:005410B2                  call    set_bit
.text:005410B7                  add     esp, 10h
.text:005410BA                  inc     esi             ; ↙
    ↳ increment loop2 counter
.text:005410BB                  cmp     esi, 8
.text:005410BE                  jl      short second_loop2_begin
.text:005410C0                  dec     edi             ; ↙
    ↳ decrement loop2 counter
.text:005410C1                  add     ebx, 8
.text:005410C4                  cmp     edi, 0FFFFFFFFh
.text:005410C7                  jg      short second_loop1_begin
.text:005410C9                  pop     edi
.text:005410CA                  pop     esi
.text:005410CB                  pop     ebp
.text:005410CC                  pop     ebx
.text:005410CD                  add     esp, 40h
.text:005410D0                  retn
.text:005410D0 rotate1          endp
.text:005410D0
.text:005410D1                  align 10h
.text:005410E0
.text:005410E0 ; =============== S U B R O U T I N E ↙
    ↳ =======================================
.text:005410E0
.text:005410E0
.text:005410E0 rotate2          proc near               ; CODE ↙
    ↳ XREF: rotate_all_with_password+7A
```

```
.text:005410E0
.text:005410E0 internal_array_64= byte ptr -40h
.text:005410E0 arg_0           = dword ptr  4
.text:005410E0
.text:005410E0                 sub     esp, 40h
.text:005410E3                 push    ebx
.text:005410E4                 push    ebp
.text:005410E5                 mov     ebp, [esp+48h+arg_0]
.text:005410E9                 push    esi
.text:005410EA                 push    edi
.text:005410EB                 xor     edi, edi        ; loop1 counter
.text:005410ED                 lea     ebx, [esp+50h+internal_array_64]
.text:005410F1
.text:005410F1 loc_5410F1:                             ; CODE XREF: rotate2+2E
.text:005410F1                 xor     esi, esi        ; loop2 counter
.text:005410F3
.text:005410F3 loc_5410F3:                             ; CODE XREF: rotate2+25
.text:005410F3                 push    esi             ; loop2
.text:005410F4                 push    edi             ; loop1
.text:005410F5                 push    ebp             ; arg_0
.text:005410F6                 call    get_bit
.text:005410FB                 add     esp, 0Ch
.text:005410FE                 mov     [ebx+esi], al   ; store to internal array
.text:00541101                 inc     esi             ; increment loop1 counter
.text:00541102                 cmp     esi, 8
.text:00541105                 jl      short loc_5410F3
.text:00541107                 inc     edi             ; increment loop2 counter
.text:00541108                 add     ebx, 8
.text:0054110B                 cmp     edi, 8
.text:0054110E                 jl      short loc_5410F1
.text:00541110                 lea     ebx, [esp+50h+internal_array_64]
.text:00541114                 mov     edi, 7          ; loop1 counter is initial state 7
.text:00541119
```

```
.text:00541119 loc_541119:                             ; CODE ↙
    ↳ XREF: rotate2+57
.text:00541119                 xor     esi, esi        ; loop2 ↙
    ↳ counter
.text:0054111B
.text:0054111B loc_54111B:                             ; CODE ↙
    ↳ XREF: rotate2+4E
.text:0054111B                 mov     al, [ebx+esi]   ; get ↙
    ↳ byte from internal array
.text:0054111E                 push    eax
.text:0054111F                 push    edi             ; loop1 ↙
    ↳ counter
.text:00541120                 push    esi             ; loop2 ↙
    ↳ counter
.text:00541121                 push    ebp             ; arg_0
.text:00541122                 call    set_bit
.text:00541127                 add     esp, 10h
.text:0054112A                 inc     esi             ; ↙
    ↳ increment loop2 counter
.text:0054112B                 cmp     esi, 8
.text:0054112E                 jl      short loc_54111B
.text:00541130                 dec     edi             ; ↙
    ↳ decrement loop2 counter
.text:00541131                 add     ebx, 8
.text:00541134                 cmp     edi, 0FFFFFFFFh
.text:00541137                 jg      short loc_541119
.text:00541139                 pop     edi
.text:0054113A                 pop     esi
.text:0054113B                 pop     ebp
.text:0054113C                 pop     ebx
.text:0054113D                 add     esp, 40h
.text:00541140                 retn
.text:00541140 rotate2         endp
.text:00541140
.text:00541141                 align 10h
.text:00541150
.text:00541150 ; =============== S U B R O U T I N E ↙
    ↳ =======================================
.text:00541150
.text:00541150
.text:00541150 rotate3         proc near               ; CODE ↙
    ↳ XREF: rotate_all_with_password+66
.text:00541150
```

```
.text:00541150 var_40          = byte ptr -40h
.text:00541150 arg_0           = dword ptr  4
.text:00541150
.text:00541150                 sub     esp, 40h
.text:00541153                 push    ebx
.text:00541154                 push    ebp
.text:00541155                 mov     ebp, [esp+48h+arg_0]
.text:00541159                 push    esi
.text:0054115A                 push    edi
.text:0054115B                 xor     edi, edi
.text:0054115D                 lea     ebx, [esp+50h+var_40]
.text:00541161
.text:00541161 loc_541161:                             ; CODE ↙
    ↳ XREF: rotate3+2E
.text:00541161                 xor     esi, esi
.text:00541163
.text:00541163 loc_541163:                             ; CODE ↙
    ↳ XREF: rotate3+25
.text:00541163                 push    esi
.text:00541164                 push    ebp
.text:00541165                 push    edi
.text:00541166                 call    get_bit
.text:0054116B                 add     esp, 0Ch
.text:0054116E                 mov     [ebx+esi], al
.text:00541171                 inc     esi
.text:00541172                 cmp     esi, 8
.text:00541175                 jl      short loc_541163
.text:00541177                 inc     edi
.text:00541178                 add     ebx, 8
.text:0054117B                 cmp     edi, 8
.text:0054117E                 jl      short loc_541161
.text:00541180                 xor     ebx, ebx
.text:00541182                 lea     edi, [esp+50h+var_40]
.text:00541186
.text:00541186 loc_541186:                             ; CODE ↙
    ↳ XREF: rotate3+54
.text:00541186                 mov     esi, 7
.text:0054118B
.text:0054118B loc_54118B:                             ; CODE ↙
    ↳ XREF: rotate3+4E
.text:0054118B                 mov     al, [edi]
.text:0054118D                 push    eax
.text:0054118E                 push    ebx
```

```
.text:0054118F                  push    ebp
.text:00541190                  push    esi
.text:00541191                  call    set_bit
.text:00541196                  add     esp, 10h
.text:00541199                  inc     edi
.text:0054119A                  dec     esi
.text:0054119B                  cmp     esi, 0FFFFFFFFh
.text:0054119E                  jg      short loc_54118B
.text:005411A0                  inc     ebx
.text:005411A1                  cmp     ebx, 8
.text:005411A4                  jl      short loc_541186
.text:005411A6                  pop     edi
.text:005411A7                  pop     esi
.text:005411A8                  pop     ebp
.text:005411A9                  pop     ebx
.text:005411AA                  add     esp, 40h
.text:005411AD                  retn
.text:005411AD rotate3          endp
.text:005411AD
.text:005411AE                  align 10h
.text:005411B0
.text:005411B0 ; =============== S U B R O U T I N E ⇙
    ↳ =======================================
.text:005411B0
.text:005411B0
.text:005411B0 rotate_all_with_password proc near      ; CODE ⇙
    ↳ XREF: crypt+1F
.text:005411B0                                         ; decrypt⇙
    ↳ +36
.text:005411B0
.text:005411B0 arg_0            = dword ptr  4
.text:005411B0 arg_4            = dword ptr  8
.text:005411B0
.text:005411B0                  mov     eax, [esp+arg_0]
.text:005411B4                  push    ebp
.text:005411B5                  mov     ebp, eax
.text:005411B7                  cmp     byte ptr [eax], 0
.text:005411BA                  jz      exit
.text:005411C0                  push    ebx
.text:005411C1                  mov     ebx, [esp+8+arg_4]
.text:005411C5                  push    esi
.text:005411C6                  push    edi
.text:005411C7
```

```
.text:005411C7 loop_begin:                             ; CODE ↙
    ↳ XREF: rotate_all_with_password+9F
.text:005411C7                 movsx   eax, byte ptr [ebp+0]
.text:005411CB                 push    eax             ; C
.text:005411CC                 call    _tolower
.text:005411D1                 add     esp, 4
.text:005411D4                 cmp     al, 'a'
.text:005411D6                 jl      short ↙
    ↳ next_character_in_password
.text:005411D8                 cmp     al, 'z'
.text:005411DA                 jg      short ↙
    ↳ next_character_in_password
.text:005411DC                 movsx   ecx, al
.text:005411DF                 sub     ecx, 'a'
.text:005411E2                 cmp     ecx, 24
.text:005411E5                 jle     short skip_subtracting
.text:005411E7                 sub     ecx, 24
.text:005411EA
.text:005411EA skip_subtracting:                       ; CODE ↙
    ↳ XREF: rotate_all_with_password+35
.text:005411EA                 mov     eax, 55555556h
.text:005411EF                 imul    ecx
.text:005411F1                 mov     eax, edx
.text:005411F3                 shr     eax, 1Fh
.text:005411F6                 add     edx, eax
.text:005411F8                 mov     eax, ecx
.text:005411FA                 mov     esi, edx
.text:005411FC                 mov     ecx, 3
.text:00541201                 cdq
.text:00541202                 idiv    ecx
.text:00541204                 sub     edx, 0
.text:00541207                 jz      short call_rotate1
.text:00541209                 dec     edx
.text:0054120A                 jz      short call_rotate2
.text:0054120C                 dec     edx
.text:0054120D                 jnz     short ↙
    ↳ next_character_in_password
.text:0054120F                 test    ebx, ebx
.text:00541211                 jle     short ↙
    ↳ next_character_in_password
.text:00541213                 mov     edi, ebx
.text:00541215
.text:00541215 call_rotate3:                           ; CODE ↙
```

```
    ↳ XREF: rotate_all_with_password+6F
.text:00541215                 push    esi
.text:00541216                 call    rotate3
.text:0054121B                 add     esp, 4
.text:0054121E                 dec     edi
.text:0054121F                 jnz     short call_rotate3
.text:00541221                 jmp     short ↙
    ↳ next_character_in_password
.text:00541223
.text:00541223 call_rotate2:                           ; CODE ↙
    ↳ XREF: rotate_all_with_password+5A
.text:00541223                 test    ebx, ebx
.text:00541225                 jle     short ↙
    ↳ next_character_in_password
.text:00541227                 mov     edi, ebx
.text:00541229
.text:00541229 loc_541229:                             ; CODE ↙
    ↳ XREF: rotate_all_with_password+83
.text:00541229                 push    esi
.text:0054122A                 call    rotate2
.text:0054122F                 add     esp, 4
.text:00541232                 dec     edi
.text:00541233                 jnz     short loc_541229
.text:00541235                 jmp     short ↙
    ↳ next_character_in_password
.text:00541237
.text:00541237 call_rotate1:                           ; CODE ↙
    ↳ XREF: rotate_all_with_password+57
.text:00541237                 test    ebx, ebx
.text:00541239                 jle     short ↙
    ↳ next_character_in_password
.text:0054123B                 mov     edi, ebx
.text:0054123D
.text:0054123D loc_54123D:                             ; CODE ↙
    ↳ XREF: rotate_all_with_password+97
.text:0054123D                 push    esi
.text:0054123E                 call    rotate1
.text:00541243                 add     esp, 4
.text:00541246                 dec     edi
.text:00541247                 jnz     short loc_54123D
.text:00541249
.text:00541249 next_character_in_password:             ; CODE ↙
    ↳ XREF: rotate_all_with_password+26
```

```
.text:00541249                                          ; ↙
    ↳ rotate_all_with_password+2A ...
.text:00541249                  mov     al, [ebp+1]
.text:0054124C                  inc     ebp
.text:0054124D                  test    al, al
.text:0054124F                  jnz     loop_begin
.text:00541255                  pop     edi
.text:00541256                  pop     esi
.text:00541257                  pop     ebx
.text:00541258
.text:00541258 exit:                                    ; CODE ↙
    ↳ XREF: rotate_all_with_password+A
.text:00541258                  pop     ebp
.text:00541259                  retn
.text:00541259 rotate_all_with_password endp
.text:00541259
.text:0054125A                  align 10h
.text:00541260
.text:00541260 ; =============== S U B R O U T I N E ↙
    ↳ =======================================
.text:00541260
.text:00541260
.text:00541260 crypt            proc near               ; CODE ↙
    ↳ XREF: crypt_file+8A
.text:00541260
.text:00541260 arg_0            = dword ptr  4
.text:00541260 arg_4            = dword ptr  8
.text:00541260 arg_8            = dword ptr  0Ch
.text:00541260
.text:00541260                  push    ebx
.text:00541261                  mov     ebx, [esp+4+arg_0]
.text:00541265                  push    ebp
.text:00541266                  push    esi
.text:00541267                  push    edi
.text:00541268                  xor     ebp, ebp
.text:0054126A
.text:0054126A loc_54126A:                              ; CODE ↙
    ↳ XREF: crypt+41
.text:0054126A                  mov     eax, [esp+10h+arg_8]
.text:0054126E                  mov     ecx, 10h
.text:00541273                  mov     esi, ebx
.text:00541275                  mov     edi, offset cube64
.text:0054127A                  push    1
```

```
.text:0054127C                  push    eax
.text:0054127D                  rep movsd
.text:0054127F                  call    rotate_all_with_password
.text:00541284                  mov     eax, [esp+18h+arg_4]
.text:00541288                  mov     edi, ebx
.text:0054128A                  add     ebp, 40h
.text:0054128D                  add     esp, 8
.text:00541290                  mov     ecx, 10h
.text:00541295                  mov     esi, offset cube64
.text:0054129A                  add     ebx, 40h
.text:0054129D                  cmp     ebp, eax
.text:0054129F                  rep movsd
.text:005412A1                  jl      short loc_54126A
.text:005412A3                  pop     edi
.text:005412A4                  pop     esi
.text:005412A5                  pop     ebp
.text:005412A6                  pop     ebx
.text:005412A7                  retn
.text:005412A7 crypt            endp
.text:005412A7
.text:005412A8                  align 10h
.text:005412B0
.text:005412B0 ; =============== S U B R O U T I N E ↙
    ↳ =======================================
.text:005412B0
.text:005412B0
.text:005412B0 ; int __cdecl decrypt(int, int, void *Src)
.text:005412B0 decrypt          proc near               ; CODE ↙
    ↳ XREF: decrypt_file+99
.text:005412B0
.text:005412B0 arg_0            = dword ptr  4
.text:005412B0 arg_4            = dword ptr  8
.text:005412B0 Src              = dword ptr  0Ch
.text:005412B0
.text:005412B0                  mov     eax, [esp+Src]
.text:005412B4                  push    ebx
.text:005412B5                  push    ebp
.text:005412B6                  push    esi
.text:005412B7                  push    edi
.text:005412B8                  push    eax             ; Src
.text:005412B9                  call    __strdup
.text:005412BE                  push    eax             ; Str
.text:005412BF                  mov     [esp+18h+Src], eax
```

```
.text:005412C3                call    __strrev
.text:005412C8                mov     ebx, [esp+18h+arg_0]
.text:005412CC                add     esp, 8
.text:005412CF                xor     ebp, ebp
.text:005412D1
.text:005412D1 loc_5412D1:                            ; CODE ↙
    ↳ XREF: decrypt+58
.text:005412D1                mov     ecx, 10h
.text:005412D6                mov     esi, ebx
.text:005412D8                mov     edi, offset cube64
.text:005412DD                push    3
.text:005412DF                rep movsd
.text:005412E1                mov     ecx, [esp+14h+Src]
.text:005412E5                push    ecx
.text:005412E6                call    rotate_all_with_password
.text:005412EB                mov     eax, [esp+18h+arg_4]
.text:005412EF                mov     edi, ebx
.text:005412F1                add     ebp, 40h
.text:005412F4                add     esp, 8
.text:005412F7                mov     ecx, 10h
.text:005412FC                mov     esi, offset cube64
.text:00541301                add     ebx, 40h
.text:00541304                cmp     ebp, eax
.text:00541306                rep movsd
.text:00541308                jl      short loc_5412D1
.text:0054130A                mov     edx, [esp+10h+Src]
.text:0054130E                push    edx             ; Memory
.text:0054130F                call    _free
.text:00541314                add     esp, 4
.text:00541317                pop     edi
.text:00541318                pop     esi
.text:00541319                pop     ebp
.text:0054131A                pop     ebx
.text:0054131B                retn
.text:0054131B decrypt        endp
.text:0054131B
.text:0054131C                align 10h
.text:00541320
.text:00541320 ; =============== S U B R O U T I N E ↙
    ↳ =======================================
.text:00541320
.text:00541320
.text:00541320 ; int __cdecl crypt_file(int Str, char *Filename,↙
```

```
    ↳  int password)
.text:00541320 crypt_file      proc near               ; CODE ↙
    ↳ XREF: _main+42
.text:00541320
.text:00541320 Str             = dword ptr  4
.text:00541320 Filename        = dword ptr  8
.text:00541320 password        = dword ptr  0Ch
.text:00541320
.text:00541320                 mov     eax, [esp+Str]
.text:00541324                 push    ebp
.text:00541325                 push    offset Mode     ; "rb"
.text:0054132A                 push    eax             ; ↙
    ↳ Filename
.text:0054132B                 call    _fopen          ; open ↙
    ↳ file
.text:00541330                 mov     ebp, eax
.text:00541332                 add     esp, 8
.text:00541335                 test    ebp, ebp
.text:00541337                 jnz     short loc_541348
.text:00541339                 push    offset Format   ; "Cannot↙
    ↳  open input file!\n"
.text:0054133E                 call    _printf
.text:00541343                 add     esp, 4
.text:00541346                 pop     ebp
.text:00541347                 retn
.text:00541348
.text:00541348 loc_541348:                             ; CODE ↙
    ↳ XREF: crypt_file+17
.text:00541348                 push    ebx
.text:00541349                 push    esi
.text:0054134A                 push    edi
.text:0054134B                 push    2               ; Origin
.text:0054134D                 push    0               ; Offset
.text:0054134F                 push    ebp             ; File
.text:00541350                 call    _fseek
.text:00541355                 push    ebp             ; File
.text:00541356                 call    _ftell          ; get ↙
    ↳ file size
.text:0054135B                 push    0               ; Origin
.text:0054135D                 push    0               ; Offset
.text:0054135F                 push    ebp             ; File
.text:00541360                 mov     [esp+2Ch+Str], eax
.text:00541364                 call    _fseek          ; rewind ↙
```

```
    ↳ to start
.text:00541369                  mov     esi, [esp+2Ch+Str]
.text:0054136D                  and     esi, 0FFFFFFC0h ; reset ↙
    ↳ all lowest 6 bits
.text:00541370                  add     esi, 40h        ; align ↙
    ↳ size to 64-byte border
.text:00541373                  push    esi             ; Size
.text:00541374                  call    _malloc
.text:00541379                  mov     ecx, esi
.text:0054137B                  mov     ebx, eax        ; ↙
    ↳ allocated buffer pointer -> to EBX
.text:0054137D                  mov     edx, ecx
.text:0054137F                  xor     eax, eax
.text:00541381                  mov     edi, ebx
.text:00541383                  push    ebp             ; File
.text:00541384                  shr     ecx, 2
.text:00541387                  rep stosd
.text:00541389                  mov     ecx, edx
.text:0054138B                  push    1               ; Count
.text:0054138D                  and     ecx, 3
.text:00541390                  rep stosb               ; memset ↙
    ↳ (buffer, 0, aligned_size)
.text:00541392                  mov     eax, [esp+38h+Str]
.text:00541396                  push    eax             ; ↙
    ↳ ElementSize
.text:00541397                  push    ebx             ; DstBuf
.text:00541398                  call    _fread          ; read ↙
    ↳ file
.text:0054139D                  push    ebp             ; File
.text:0054139E                  call    _fclose
.text:005413A3                  mov     ecx, [esp+44h+password]
.text:005413A7                  push    ecx             ; ↙
    ↳ password
.text:005413A8                  push    esi             ; aligned↙
    ↳  size
.text:005413A9                  push    ebx             ; buffer
.text:005413AA                  call    crypt           ; do ↙
    ↳ crypt
.text:005413AF                  mov     edx, [esp+50h+Filename]
.text:005413B3                  add     esp, 40h
.text:005413B6                  push    offset aWb      ; "wb"
.text:005413BB                  push    edx             ; ↙
    ↳ Filename
```

```
.text:005413BC                  call    _fopen
.text:005413C1                  mov     edi, eax
.text:005413C3                  push    edi             ; File
.text:005413C4                  push    1               ; Count
.text:005413C6                  push    3               ; Size
.text:005413C8                  push    offset aQr9     ; "QR9"
.text:005413CD                  call    _fwrite         ; write ↙
    ↳ file signature
.text:005413D2                  push    edi             ; File
.text:005413D3                  push    1               ; Count
.text:005413D5                  lea     eax, [esp+30h+Str]
.text:005413D9                  push    4               ; Size
.text:005413DB                  push    eax             ; Str
.text:005413DC                  call    _fwrite         ; write ↙
    ↳ original file size
.text:005413E1                  push    edi             ; File
.text:005413E2                  push    1               ; Count
.text:005413E4                  push    esi             ; Size
.text:005413E5                  push    ebx             ; Str
.text:005413E6                  call    _fwrite         ; write ↙
    ↳ crypted file
.text:005413EB                  push    edi             ; File
.text:005413EC                  call    _fclose
.text:005413F1                  push    ebx             ; Memory
.text:005413F2                  call    _free
.text:005413F7                  add     esp, 40h
.text:005413FA                  pop     edi
.text:005413FB                  pop     esi
.text:005413FC                  pop     ebx
.text:005413FD                  pop     ebp
.text:005413FE                  retn
.text:005413FE crypt_file       endp
.text:005413FE
.text:005413FF                  align 10h
.text:00541400
.text:00541400 ; =============== S U B R O U T I N E ↙
    ↳ =======================================
.text:00541400
.text:00541400
.text:00541400 ; int __cdecl decrypt_file(char *Filename, int, ↙
    ↳ void *Src)
.text:00541400 decrypt_file     proc near               ; CODE ↙
    ↳ XREF: _main+6E
```

```
.text:00541400
.text:00541400 Filename        = dword ptr  4
.text:00541400 arg_4           = dword ptr  8
.text:00541400 Src             = dword ptr  0Ch
.text:00541400
.text:00541400                 mov     eax, [esp+Filename]
.text:00541404                 push    ebx
.text:00541405                 push    ebp
.text:00541406                 push    esi
.text:00541407                 push    edi
.text:00541408                 push    offset aRb      ; "rb"
.text:0054140D                 push    eax             ; ↙
    ↳ Filename
.text:0054140E                 call    _fopen
.text:00541413                 mov     esi, eax
.text:00541415                 add     esp, 8
.text:00541418                 test    esi, esi
.text:0054141A                 jnz     short loc_54142E
.text:0054141C                 push    offset aCannotOpenIn_0 ; ↙
    ↳ "Cannot open input file!\n"
.text:00541421                 call    _printf
.text:00541426                 add     esp, 4
.text:00541429                 pop     edi
.text:0054142A                 pop     esi
.text:0054142B                 pop     ebp
.text:0054142C                 pop     ebx
.text:0054142D                 retn
.text:0054142E
.text:0054142E loc_54142E:                             ; CODE ↙
    ↳ XREF: decrypt_file+1A
.text:0054142E                 push    2               ; Origin
.text:00541430                 push    0               ; Offset
.text:00541432                 push    esi             ; File
.text:00541433                 call    _fseek
.text:00541438                 push    esi             ; File
.text:00541439                 call    _ftell
.text:0054143E                 push    0               ; Origin
.text:00541440                 push    0               ; Offset
.text:00541442                 push    esi             ; File
.text:00541443                 mov     ebp, eax
.text:00541445                 call    _fseek
.text:0054144A                 push    ebp             ; Size
.text:0054144B                 call    _malloc
```

```
.text:00541450                 push    esi             ; File
.text:00541451                 mov     ebx, eax
.text:00541453                 push    1               ; Count
.text:00541455                 push    ebp             ; ↙
    ↳ ElementSize
.text:00541456                 push    ebx             ; DstBuf
.text:00541457                 call    _fread
.text:0054145C                 push    esi             ; File
.text:0054145D                 call    _fclose
.text:00541462                 add     esp, 34h
.text:00541465                 mov     ecx, 3
.text:0054146A                 mov     edi, offset aQr9_0 ; "QR9↙
    ↳ "
.text:0054146F                 mov     esi, ebx
.text:00541471                 xor     edx, edx
.text:00541473                 repe cmpsb
.text:00541475                 jz      short loc_541489
.text:00541477                 push    offset aFileIsNotCrypt ; ↙
    ↳ "File is not crypted!\n"
.text:0054147C                 call    _printf
.text:00541481                 add     esp, 4
.text:00541484                 pop     edi
.text:00541485                 pop     esi
.text:00541486                 pop     ebp
.text:00541487                 pop     ebx
.text:00541488                 retn
.text:00541489
.text:00541489 loc_541489:                             ; CODE ↙
    ↳ XREF: decrypt_file+75
.text:00541489                 mov     eax, [esp+10h+Src]
.text:0054148D                 mov     edi, [ebx+3]
.text:00541490                 add     ebp, 0FFFFFFF9h
.text:00541493                 lea     esi, [ebx+7]
.text:00541496                 push    eax             ; Src
.text:00541497                 push    ebp             ; int
.text:00541498                 push    esi             ; int
.text:00541499                 call    decrypt
.text:0054149E                 mov     ecx, [esp+1Ch+arg_4]
.text:005414A2                 push    offset aWb_0    ; "wb"
.text:005414A7                 push    ecx             ; ↙
    ↳ Filename
.text:005414A8                 call    _fopen
.text:005414AD                 mov     ebp, eax
```

```
.text:005414AF                  push    ebp             ; File
.text:005414B0                  push    1               ; Count
.text:005414B2                  push    edi             ; Size
.text:005414B3                  push    esi             ; Str
.text:005414B4                  call    _fwrite
.text:005414B9                  push    ebp             ; File
.text:005414BA                  call    _fclose
.text:005414BF                  push    ebx             ; Memory
.text:005414C0                  call    _free
.text:005414C5                  add     esp, 2Ch
.text:005414C8                  pop     edi
.text:005414C9                  pop     esi
.text:005414CA                  pop     ebp
.text:005414CB                  pop     ebx
.text:005414CC                  retn
.text:005414CC decrypt_file     endp
```

All function and label names are given by me while analysis.
I started from top. Here is a function taking two file names and password.

```
.text:00541320 ; int __cdecl crypt_file(int Str, char *Filename,↙
    ↳  int password)
.text:00541320 crypt_file       proc near
.text:00541320
.text:00541320 Str              = dword ptr  4
.text:00541320 Filename         = dword ptr  8
.text:00541320 password         = dword ptr  0Ch
.text:00541320
```

Open file and report error in case of error:

```
.text:00541320                  mov     eax, [esp+Str]
.text:00541324                  push    ebp
.text:00541325                  push    offset Mode     ; "rb"
.text:0054132A                  push    eax             ; ↙
    ↳ Filename
.text:0054132B                  call    _fopen          ; open ↙
    ↳ file
.text:00541330                  mov     ebp, eax
.text:00541332                  add     esp, 8
.text:00541335                  test    ebp, ebp
.text:00541337                  jnz     short loc_541348
.text:00541339                  push    offset Format   ; "Cannot↙
    ↳  open input file!\n"
.text:0054133E                  call    _printf
```

```
.text:00541343                  add     esp, 4
.text:00541346                  pop     ebp
.text:00541347                  retn
.text:00541348
.text:00541348 loc_541348:
```

Get file size via `fseek()`/`ftell()`:

```
.text:00541348 push    ebx
.text:00541349 push    esi
.text:0054134A push    edi
.text:0054134B push    2               ; Origin
.text:0054134D push    0               ; Offset
.text:0054134F push    ebp             ; File

; move current file position to the end
.text:00541350 call    _fseek
.text:00541355 push    ebp             ; File
.text:00541356 call    _ftell          ; get current file ↙
    ↳ position
.text:0054135B push    0               ; Origin
.text:0054135D push    0               ; Offset
.text:0054135F push    ebp             ; File
.text:00541360 mov     [esp+2Ch+Str], eax

; move current file position to the start
.text:00541364 call    _fseek
```

This fragment of code calculates file size aligned on a 64-byte boundary. This is because this cryptoalgorithm works with only 64-byte blocks. Its operation is pretty straightforward: divide file size by 64, forget about remainder and add 1, then multiple by 64. The following code removes remainder as if value was already divided by 64 and adds 64. It is almost the same.

```
.text:00541369 mov     esi, [esp+2Ch+Str]
.text:0054136D and     esi, 0FFFFFFC0h ; reset all lowest 6 bits
.text:00541370 add     esi, 40h        ; align size to 64-byte ↙
    ↳ border
```

Allocate buffer with aligned size:

```
.text:00541373                  push    esi             ; Size
.text:00541374                  call    _malloc
```

Call memset(), e.g., clears allocated buffer[2].

[2] malloc() + memset() could be replaced by calloc()

```
.text:00541379 mov     ecx, esi
.text:0054137B mov     ebx, eax        ; allocated buffer ↙
    ↳ pointer -> to EBX
.text:0054137D mov     edx, ecx
.text:0054137F xor     eax, eax
.text:00541381 mov     edi, ebx
.text:00541383 push    ebp             ; File
.text:00541384 shr     ecx, 2
.text:00541387 rep stosd
.text:00541389 mov     ecx, edx
.text:0054138B push    1               ; Count
.text:0054138D and     ecx, 3
.text:00541390 rep stosb               ; memset (buffer, 0, ↙
    ↳ aligned_size)
```

Read file via standard C function `fread()`.

```
.text:00541392                  mov     eax, [esp+38h+Str]
.text:00541396                  push    eax             ; ↙
    ↳ ElementSize
.text:00541397                  push    ebx             ; DstBuf
.text:00541398                  call    _fread          ; read ↙
    ↳ file
.text:0054139D                  push    ebp             ; File
.text:0054139E                  call    _fclose
```

Call `crypt()`. This function takes buffer, buffer size (aligned) and password string.

```
.text:005413A3                  mov     ecx, [esp+44h+password]
.text:005413A7                  push    ecx             ; ↙
    ↳ password
.text:005413A8                  push    esi             ; aligned↙
    ↳  size
.text:005413A9                  push    ebx             ; buffer
.text:005413AA                  call    crypt           ; do ↙
    ↳ crypt
```

Create output file. By the way, developer forgot to check if it is was created correctly! File opening result is being checked though.

```
.text:005413AF                  mov     edx, [esp+50h+Filename]
.text:005413B3                  add     esp, 40h
.text:005413B6                  push    offset aWb      ; "wb"
.text:005413BB                  push    edx             ; ↙
    ↳ Filename
```

```
.text:005413BC                  call    _fopen
.text:005413C1                  mov     edi, eax
```

Newly created file handle is in the EDI register now. Write signature "QR9".

```
.text:005413C3                  push    edi             ; File
.text:005413C4                  push    1               ; Count
.text:005413C6                  push    3               ; Size
.text:005413C8                  push    offset aQr9     ; "QR9"
.text:005413CD                  call    _fwrite         ; write ↙
    ↳ file signature
```

Write actual file size (not aligned):

```
.text:005413D2                  push    edi             ; File
.text:005413D3                  push    1               ; Count
.text:005413D5                  lea     eax, [esp+30h+Str]
.text:005413D9                  push    4               ; Size
.text:005413DB                  push    eax             ; Str
.text:005413DC                  call    _fwrite         ; write ↙
    ↳ original file size
```

Write crypted buffer:

```
.text:005413E1                  push    edi             ; File
.text:005413E2                  push    1               ; Count
.text:005413E4                  push    esi             ; Size
.text:005413E5                  push    ebx             ; Str
.text:005413E6                  call    _fwrite         ; write ↙
    ↳ encrypted file
```

Close file and free allocated buffer:

```
.text:005413EB                  push    edi             ; File
.text:005413EC                  call    _fclose
.text:005413F1                  push    ebx             ; Memory
.text:005413F2                  call    _free
.text:005413F7                  add     esp, 40h
.text:005413FA                  pop     edi
.text:005413FB                  pop     esi
.text:005413FC                  pop     ebx
.text:005413FD                  pop     ebp
.text:005413FE                  retn
.text:005413FE crypt_file       endp
```

Here is reconstructed C-code:

```
void crypt_file(char *fin, char* fout, char *pw)
{
        FILE *f;
        int flen, flen_aligned;
        BYTE *buf;

        f=fopen(fin, "rb");

        if (f==NULL)
        {
                printf ("Cannot open input file!\n");
                return;
        };

        fseek (f, 0, SEEK_END);
        flen=ftell (f);
        fseek (f, 0, SEEK_SET);

        flen_aligned=(flen&0xFFFFFFC0)+0x40;

        buf=(BYTE*)malloc (flen_aligned);
        memset (buf, 0, flen_aligned);

        fread (buf, flen, 1, f);

        fclose (f);

        crypt (buf, flen_aligned, pw);

        f=fopen(fout, "wb");

        fwrite ("QR9", 3, 1, f);
        fwrite (&flen, 4, 1, f);
        fwrite (buf, flen_aligned, 1, f);

        fclose (f);

        free (buf);
};
```

Decrypting procedure is almost the same:

```
.text:00541400 ; int __cdecl decrypt_file(char *Filename, int, ↙
    ↳ void *Src)
```

```
.text:00541400 decrypt_file    proc near
.text:00541400
.text:00541400 Filename        = dword ptr  4
.text:00541400 arg_4           = dword ptr  8
.text:00541400 Src             = dword ptr  0Ch
.text:00541400
.text:00541400                 mov     eax, [esp+Filename]
.text:00541404                 push    ebx
.text:00541405                 push    ebp
.text:00541406                 push    esi
.text:00541407                 push    edi
.text:00541408                 push    offset aRb      ; "rb"
.text:0054140D                 push    eax             ; ↙
    ↳ Filename
.text:0054140E                 call    _fopen
.text:00541413                 mov     esi, eax
.text:00541415                 add     esp, 8
.text:00541418                 test    esi, esi
.text:0054141A                 jnz     short loc_54142E
.text:0054141C                 push    offset aCannotOpenIn_0 ; ↙
    ↳ "Cannot open input file!\n"
.text:00541421                 call    _printf
.text:00541426                 add     esp, 4
.text:00541429                 pop     edi
.text:0054142A                 pop     esi
.text:0054142B                 pop     ebp
.text:0054142C                 pop     ebx
.text:0054142D                 retn
.text:0054142E
.text:0054142E loc_54142E:
.text:0054142E                 push    2               ; Origin
.text:00541430                 push    0               ; Offset
.text:00541432                 push    esi             ; File
.text:00541433                 call    _fseek
.text:00541438                 push    esi             ; File
.text:00541439                 call    _ftell
.text:0054143E                 push    0               ; Origin
.text:00541440                 push    0               ; Offset
.text:00541442                 push    esi             ; File
.text:00541443                 mov     ebp, eax
.text:00541445                 call    _fseek
.text:0054144A                 push    ebp             ; Size
.text:0054144B                 call    _malloc
```

```
.text:00541450                  push    esi             ; File
.text:00541451                  mov     ebx, eax
.text:00541453                  push    1               ; Count
.text:00541455                  push    ebp             ; ↙
    ↳ ElementSize
.text:00541456                  push    ebx             ; DstBuf
.text:00541457                  call    _fread
.text:0054145C                  push    esi             ; File
.text:0054145D                  call    _fclose
```

Check signature (first 3 bytes):

```
.text:00541462                  add     esp, 34h
.text:00541465                  mov     ecx, 3
.text:0054146A                  mov     edi, offset aQr9_0 ; "QR9↙
    ↳ "
.text:0054146F                  mov     esi, ebx
.text:00541471                  xor     edx, edx
.text:00541473                  repe cmpsb
.text:00541475                  jz      short loc_541489
```

Report an error if signature is absent:

```
.text:00541477                  push    offset aFileIsNotCrypt ; ↙
    ↳ "File is not crypted!\n"
.text:0054147C                  call    _printf
.text:00541481                  add     esp, 4
.text:00541484                  pop     edi
.text:00541485                  pop     esi
.text:00541486                  pop     ebp
.text:00541487                  pop     ebx
.text:00541488                  retn
.text:00541489
.text:00541489 loc_541489:
```

Call `decrypt()`.

```
.text:00541489                  mov     eax, [esp+10h+Src]
.text:0054148D                  mov     edi, [ebx+3]
.text:00541490                  add     ebp, 0FFFFFFF9h
.text:00541493                  lea     esi, [ebx+7]
.text:00541496                  push    eax             ; Src
.text:00541497                  push    ebp             ; int
.text:00541498                  push    esi             ; int
.text:00541499                  call    decrypt
.text:0054149E                  mov     ecx, [esp+1Ch+arg_4]
```

```
.text:005414A2                  push    offset aWb_0    ; "wb"
.text:005414A7                  push    ecx             ; ↙
    ↳ Filename
.text:005414A8                  call    _fopen
.text:005414AD                  mov     ebp, eax
.text:005414AF                  push    ebp             ; File
.text:005414B0                  push    1               ; Count
.text:005414B2                  push    edi             ; Size
.text:005414B3                  push    esi             ; Str
.text:005414B4                  call    _fwrite
.text:005414B9                  push    ebp             ; File
.text:005414BA                  call    _fclose
.text:005414BF                  push    ebx             ; Memory
.text:005414C0                  call    _free
.text:005414C5                  add     esp, 2Ch
.text:005414C8                  pop     edi
.text:005414C9                  pop     esi
.text:005414CA                  pop     ebp
.text:005414CB                  pop     ebx
.text:005414CC                  retn
.text:005414CC decrypt_file     endp
```

Here is reconstructed C-code:

```
void decrypt_file(char *fin, char* fout, char *pw)
{
        FILE *f;
        int real_flen, flen;
        BYTE *buf;

        f=fopen(fin, "rb");

        if (f==NULL)
        {
                printf ("Cannot open input file!\n");
                return;
        };

        fseek (f, 0, SEEK_END);
        flen=ftell (f);
        fseek (f, 0, SEEK_SET);

        buf=(BYTE*)malloc (flen);
```

```
        fread (buf, flen, 1, f);

        fclose (f);

        if (memcmp (buf, "QR9", 3)!=0)
        {
                printf ("File is not crypted!\n");
                return;
        };

        memcpy (&real_flen, buf+3, 4);

        decrypt (buf+(3+4), flen-(3+4), pw);

        f=fopen(fout, "wb");

        fwrite (buf+(3+4), real_flen, 1, f);

        fclose (f);

        free (buf);
};
```

OK, now let's go deeper.

Function `crypt()`:

```
.text:00541260 crypt           proc near
.text:00541260
.text:00541260 arg_0           = dword ptr  4
.text:00541260 arg_4           = dword ptr  8
.text:00541260 arg_8           = dword ptr  0Ch
.text:00541260
.text:00541260                 push    ebx
.text:00541261                 mov     ebx, [esp+4+arg_0]
.text:00541265                 push    ebp
.text:00541266                 push    esi
.text:00541267                 push    edi
.text:00541268                 xor     ebp, ebp
.text:0054126A
.text:0054126A loc_54126A:
```

This fragment of code copies part of input buffer to internal array I named later "cube64". The size is in the `ECX` register. `MOVSD` means *move 32-bit dword*, so, 16 of 32-bit dwords are exactly 64 bytes.

```
.text:0054126A                  mov     eax, [esp+10h+arg_8]
.text:0054126E                  mov     ecx, 10h
.text:00541273                  mov     esi, ebx   ; EBX is ↙
    ↳ pointer within input buffer
.text:00541275                  mov     edi, offset cube64
.text:0054127A                  push    1
.text:0054127C                  push    eax
.text:0054127D                  rep movsd
```

Call `rotate_all_with_password()`:

```
.text:0054127F                  call    rotate_all_with_password
```

Copy crypted contents back from "cube64" to buffer:

```
.text:00541284                  mov     eax, [esp+18h+arg_4]
.text:00541288                  mov     edi, ebx
.text:0054128A                  add     ebp, 40h
.text:0054128D                  add     esp, 8
.text:00541290                  mov     ecx, 10h
.text:00541295                  mov     esi, offset cube64
.text:0054129A                  add     ebx, 40h  ; add 64 to ↙
    ↳ input buffer pointer
.text:0054129D                  cmp     ebp, eax  ; EBP contain ↙
    ↳ amount of crypted data.
.text:0054129F                  rep movsd
```

If `EBP` is not bigger that input argument size, then continue to next block.

```
.text:005412A1                  jl      short loc_54126A
.text:005412A3                  pop     edi
.text:005412A4                  pop     esi
.text:005412A5                  pop     ebp
.text:005412A6                  pop     ebx
.text:005412A7                  retn
.text:005412A7 crypt            endp
```

Reconstructed `crypt()` function:

```
void crypt (BYTE *buf, int sz, char *pw)
{
        int i=0;

        do
        {
                memcpy (cube, buf+i, 8*8);
```

```
                rotate_all (pw, 1);
                memcpy (buf+i, cube, 8*8);
                i+=64;
        }
        while (i<sz);
};
```

OK, now let's go deeper into function `rotate_all_with_password()`. It takes two arguments: password string and number. In `crypt()`, number 1 is used, and in the `decrypt()` function (where `rotate_all_with_password()` function is called too), number is 3.

```
.text:005411B0 rotate_all_with_password proc near
.text:005411B0
.text:005411B0 arg_0           = dword ptr  4
.text:005411B0 arg_4           = dword ptr  8
.text:005411B0
.text:005411B0                 mov     eax, [esp+arg_0]
.text:005411B4                 push    ebp
.text:005411B5                 mov     ebp, eax
```

Check for character in password. If it is zero, exit:

```
.text:005411B7                 cmp     byte ptr [eax], 0
.text:005411BA                 jz      exit
.text:005411C0                 push    ebx
.text:005411C1                 mov     ebx, [esp+8+arg_4]
.text:005411C5                 push    esi
.text:005411C6                 push    edi
.text:005411C7
.text:005411C7 loop_begin:
```

Call `tolower()`, standard C function.

```
.text:005411C7                 movsx   eax, byte ptr [ebp+0]
.text:005411CB                 push    eax             ; C
.text:005411CC                 call    _tolower
.text:005411D1                 add     esp, 4
```

Hmm, if password contains non-alphabetical latin character, it is skipped! Indeed, if we run crypting utility and try non-alphabetical latin characters in password, they seem to be ignored.

```
.text:005411D4                 cmp     al, 'a'
.text:005411D6                 jl      short ↙
    ↳ next_character_in_password
```

```
.text:005411D8                  cmp     al, 'z'
.text:005411DA                  jg      short ↙
    ↳ next_character_in_password
.text:005411DC                  movsx   ecx, al
```

Subtract "a" value (97) from character.

```
.text:005411DF                  sub     ecx, 'a'  ; 97
```

After subtracting, we'll get 0 for "a" here, 1 for "b", etc. And 25 for "z".

```
.text:005411E2                  cmp     ecx, 24
.text:005411E5                  jle     short skip_subtracting
.text:005411E7                  sub     ecx, 24
```

It seems, "y" and "z" are exceptional characters too. After that fragment of code, "y" becomes 0 and "z" —1. This means, 26 Latin alphabet symbols will become values in range 0..23, (24 in total).

```
.text:005411EA
.text:005411EA skip_subtracting:                        ; CODE ↙
    ↳ XREF: rotate_all_with_password+35
```

This is actually division via multiplication. Read more about it in the "Division by 9" section (14).

The code actually divides password character value by 3.

```
.text:005411EA                  mov     eax, 55555556h
.text:005411EF                  imul    ecx
.text:005411F1                  mov     eax, edx
.text:005411F3                  shr     eax, 1Fh
.text:005411F6                  add     edx, eax
.text:005411F8                  mov     eax, ecx
.text:005411FA                  mov     esi, edx
.text:005411FC                  mov     ecx, 3
.text:00541201                  cdq
.text:00541202                  idiv    ecx
```

EDX is the remainder of division.

```
.text:00541204 sub     edx, 0
.text:00541207 jz      short call_rotate1 ; if remainder is zero↙
    ↳ , go to rotate1
.text:00541209 dec     edx
.text:0054120A jz      short call_rotate2 ; .. it it is 1, go to↙
    ↳  rotate2
.text:0054120C dec     edx
```

```
.text:0054120D jnz     short next_character_in_password
.text:0054120F test    ebx, ebx
.text:00541211 jle     short next_character_in_password
.text:00541213 mov     edi, ebx
```

If remainder is 2, call `rotate3()`. The `EDI` is a second argument of the `rotate_all_with_p` function. As I already wrote, 1 is for crypting operations and 3 is for decrypting. So, here is a loop. When crypting, rotate1/2/3 will be called the same number of times as given in the first argument.

```
.text:00541215 call_rotate3:
.text:00541215                 push    esi
.text:00541216                 call    rotate3
.text:0054121B                 add     esp, 4
.text:0054121E                 dec     edi
.text:0054121F                 jnz     short call_rotate3
.text:00541221                 jmp     short ↙
    ↳ next_character_in_password
.text:00541223
.text:00541223 call_rotate2:
.text:00541223                 test    ebx, ebx
.text:00541225                 jle     short ↙
    ↳ next_character_in_password
.text:00541227                 mov     edi, ebx
.text:00541229
.text:00541229 loc_541229:
.text:00541229                 push    esi
.text:0054122A                 call    rotate2
.text:0054122F                 add     esp, 4
.text:00541232                 dec     edi
.text:00541233                 jnz     short loc_541229
.text:00541235                 jmp     short ↙
    ↳ next_character_in_password
.text:00541237
.text:00541237 call_rotate1:
.text:00541237                 test    ebx, ebx
.text:00541239                 jle     short ↙
    ↳ next_character_in_password
.text:0054123B                 mov     edi, ebx
.text:0054123D
.text:0054123D loc_54123D:
.text:0054123D                 push    esi
.text:0054123E                 call    rotate1
.text:00541243                 add     esp, 4
```

```
.text:00541246                  dec     edi
.text:00541247                  jnz     short loc_54123D
.text:00541249
```

Fetch next character from password string.

```
.text:00541249 next_character_in_password:
.text:00541249                  mov     al, [ebp+1]
```

Increment character pointer within password string:

```
.text:0054124C                  inc     ebp
.text:0054124D                  test    al, al
.text:0054124F                  jnz     loop_begin
.text:00541255                  pop     edi
.text:00541256                  pop     esi
.text:00541257                  pop     ebx
.text:00541258
.text:00541258 exit:
.text:00541258                  pop     ebp
.text:00541259                  retn
.text:00541259 rotate_all_with_password endp
```

Here is reconstructed C code:

```
void rotate_all (char *pwd, int v)
{
        char *p=pwd;

        while (*p)
        {
                char c=*p;
                int q;

                c=tolower (c);

                if (c>='a' && c<='z')
                {
                        q=c-'a';
                        if (q>24)
                                q-=24;

                        int quotient=q/3;
                        int remainder=q % 3;

                        switch (remainder)
```

```
                        {
                        case 0: for (int i=0; i<v; i++) rotate1 ↙
    ↳ (quotient); break;
                        case 1: for (int i=0; i<v; i++) rotate2 ↙
    ↳ (quotient); break;
                        case 2: for (int i=0; i<v; i++) rotate3 ↙
    ↳ (quotient); break;
                        };
                };

                p++;
        };
};
```

Now let's go deeper and investigate rotate1/2/3 functions. Each function calls two another functions. I eventually gave them names `set_bit()` and `get_bit()`.

Let's start with `get_bit()`:

```
.text:00541050 get_bit         proc near
.text:00541050
.text:00541050 arg_0           = dword ptr  4
.text:00541050 arg_4           = dword ptr  8
.text:00541050 arg_8           = byte ptr  0Ch
.text:00541050
.text:00541050                 mov     eax, [esp+arg_4]
.text:00541054                 mov     ecx, [esp+arg_0]
.text:00541058                 mov     al, cube64[eax+ecx*8]
.text:0054105F                 mov     cl, [esp+arg_8]
.text:00541063                 shr     al, cl
.text:00541065                 and     al, 1
.text:00541067                 retn
.text:00541067 get_bit         endp
```

…in other words: calculate an index in the array cube64: *arg_4 + arg_0 * 8*. Then shift a byte from an array by arg_8 bits right. Isolate lowest bit and return it.

Let's see another function, `set_bit()`:

```
.text:00541000 set_bit         proc near
.text:00541000
.text:00541000 arg_0           = dword ptr  4
.text:00541000 arg_4           = dword ptr  8
.text:00541000 arg_8           = dword ptr  0Ch
.text:00541000 arg_C           = byte ptr  10h
.text:00541000
.text:00541000                 mov     al, [esp+arg_C]
```

```
.text:00541004                  mov     ecx, [esp+arg_8]
.text:00541008                  push    esi
.text:00541009                  mov     esi, [esp+4+arg_0]
.text:0054100D                  test    al, al
.text:0054100F                  mov     eax, [esp+4+arg_4]
.text:00541013                  mov     dl, 1
.text:00541015                  jz      short loc_54102B
```

Value in the DL is 1 here. Shift left it by arg_8. For example, if arg_8 is 4, value in the DL register became 0x10 or 1000 in binary form.

```
.text:00541017                  shl     dl, cl
.text:00541019                  mov     cl, cube64[eax+esi*8]
```

Get bit from array and explicitly set one.

```
.text:00541020                  or      cl, dl
```

Store it back:

```
.text:00541022                  mov     cube64[eax+esi*8], cl
.text:00541029                  pop     esi
.text:0054102A                  retn
.text:0054102B
.text:0054102B loc_54102B:
.text:0054102B                  shl     dl, cl
```

If arg_C is not zero…

```
.text:0054102D                  mov     cl, cube64[eax+esi*8]
```

…invert DL. For example, if DL state after shift was 0x10 or 1000 in binary form, there will be 0xEF after NOT instruction or 11101111 in binary form.

```
.text:00541034                  not     dl
```

This instruction clears bit, in other words, it saves all bits in CL which are also set in DL except those in DL which are cleared. This means that if DL is e.g. 11101111 in binary form, all bits will be saved except 5th (counting from lowest bit).

```
.text:00541036                  and     cl, dl
```

Store it back:

```
.text:00541038                  mov     cube64[eax+esi*8], cl
.text:0054103F                  pop     esi
.text:00541040                  retn
.text:00541040 set_bit          endp
```

It is almost the same as `get_bit()`, except, if arg_C is zero, the function clears specific bit in array, or sets it otherwise.

We also know the array size is 64. First two arguments both in the `set_bit()` and `get_bit()` functions could be seen as 2D coordinates. Then array will be 8*8 matrix.

Here is C representation of what we already know:

```
#define IS_SET(flag, bit)       ((flag) & (bit))
#define SET_BIT(var, bit)       ((var) |= (bit))
#define REMOVE_BIT(var, bit)    ((var) &= ~(bit))

char cube[8][8];

void set_bit (int x, int y, int shift, int bit)
{
        if (bit)
                SET_BIT (cube[x][y], 1<<shift);
        else
                REMOVE_BIT (cube[x][y], 1<<shift);
};

int get_bit (int x, int y, int shift)
{
        if ((cube[x][y]>>shift)&1==1)
                return 1;
        return 0;
};
```

Now let's get back to rotate1/2/3 functions.

```
.text:00541070 rotate1         proc near
.text:00541070
```

Internal array allocation in local stack, its size 64 bytes:

```
.text:00541070 internal_array_64= byte ptr -40h
.text:00541070 arg_0            = dword ptr  4
.text:00541070
.text:00541070                  sub     esp, 40h
.text:00541073                  push    ebx
.text:00541074                  push    ebp
.text:00541075                  mov     ebp, [esp+48h+arg_0]
.text:00541079                  push    esi
.text:0054107A                  push    edi
.text:0054107B                  xor     edi, edi        ; EDI is ↙
    ↳ loop1 counter
```

EBX is a pointer to internal array:

```
.text:0054107D                  lea     ebx, [esp+50h+↙
    ↳ internal_array_64]
.text:00541081
```

Two nested loops are here:

```
.text:00541081 first_loop1_begin:
.text:00541081    xor     esi, esi        ; ESI is loop 2 ↙
    ↳ counter
.text:00541083
.text:00541083 first_loop2_begin:
.text:00541083    push    ebp             ; arg_0
.text:00541084    push    esi             ; loop 1 counter
.text:00541085    push    edi             ; loop 2 counter
.text:00541086    call    get_bit
.text:0054108B    add     esp, 0Ch
.text:0054108E    mov     [ebx+esi], al   ; store to internal ↙
    ↳ array
.text:00541091    inc     esi             ; increment loop 1 ↙
    ↳ counter
.text:00541092    cmp     esi, 8
.text:00541095    jl      short first_loop2_begin
.text:00541097    inc     edi             ; increment loop 2 ↙
    ↳ counter
.text:00541098    add     ebx, 8          ; increment internal ↙
    ↳ array pointer by 8 at each loop 1 iteration
.text:0054109B    cmp     edi, 8
.text:0054109E    jl      short first_loop1_begin
```

…we see that both loop counters are in range 0..7. Also they are used as the first and the second arguments of the `get_bit()` function. Third argument of the `get_bit()` is the only argument of `rotate1()`. What `get_bit()` returns, is being placed into internal array.

Prepare pointer to internal array again:

```
.text:005410A0    lea     ebx, [esp+50h+internal_array_64]
.text:005410A4    mov     edi, 7          ; EDI is loop 1 ↙
    ↳ counter, initial state is 7
.text:005410A9
.text:005410A9 second_loop1_begin:
.text:005410A9    xor     esi, esi        ; ESI is loop 2 ↙
    ↳ counter
.text:005410AB
.text:005410AB second_loop2_begin:
```

```
.text:005410AB    mov     al, [ebx+esi]   ; value from internal ↙
    ↳ array
.text:005410AE    push    eax
.text:005410AF    push    ebp             ; arg_0
.text:005410B0    push    edi             ; loop 1 counter
.text:005410B1    push    esi             ; loop 2 counter
.text:005410B2    call    set_bit
.text:005410B7    add     esp, 10h
.text:005410BA    inc     esi             ; increment loop 2 ↙
    ↳ counter
.text:005410BB    cmp     esi, 8
.text:005410BE    jl      short second_loop2_begin
.text:005410C0    dec     edi             ; decrement loop 2 ↙
    ↳ counter
.text:005410C1    add     ebx, 8          ; increment pointer in↙
    ↳  internal array
.text:005410C4    cmp     edi, 0FFFFFFFFh
.text:005410C7    jg      short second_loop1_begin
.text:005410C9    pop     edi
.text:005410CA    pop     esi
.text:005410CB    pop     ebp
.text:005410CC    pop     ebx
.text:005410CD    add     esp, 40h
.text:005410D0    retn
.text:005410D0 rotate1          endp
```

…this code is placing contents from internal array to cube global array via `set_bit()` function, *but*, in different order! Now loop 1 counter is in range 7 to 0, decrementing at each iteration!

C code representation looks like:

```
void rotate1 (int v)
{
        bool tmp[8][8]; // internal array
        int i, j;

        for (i=0; i<8; i++)
                for (j=0; j<8; j++)
                        tmp[i][j]=get_bit (i, j, v);

        for (i=0; i<8; i++)
                for (j=0; j<8; j++)
                        set_bit (j, 7-i, v, tmp[x][y]);
};
```

Not very understandable, but if we will take a look at `rotate2()` function:

```
.text:005410E0 rotate2 proc near
.text:005410E0
.text:005410E0 internal_array_64 = byte ptr -40h
.text:005410E0 arg_0 = dword ptr  4
.text:005410E0
.text:005410E0     sub     esp, 40h
.text:005410E3     push    ebx
.text:005410E4     push    ebp
.text:005410E5     mov     ebp, [esp+48h+arg_0]
.text:005410E9     push    esi
.text:005410EA     push    edi
.text:005410EB     xor     edi, edi        ; loop 1 counter
.text:005410ED     lea     ebx, [esp+50h+internal_array_64]
.text:005410F1
.text:005410F1 loc_5410F1:
.text:005410F1     xor     esi, esi        ; loop 2 counter
.text:005410F3
.text:005410F3 loc_5410F3:
.text:005410F3     push    esi             ; loop 2 counter
.text:005410F4     push    edi             ; loop 1 counter
.text:005410F5     push    ebp             ; arg_0
.text:005410F6     call    get_bit
.text:005410FB     add     esp, 0Ch
.text:005410FE     mov     [ebx+esi], al   ; store to internal ↙
    ↳ array
.text:00541101     inc     esi             ; increment loop 1 ↙
    ↳ counter
.text:00541102     cmp     esi, 8
.text:00541105     jl      short loc_5410F3
.text:00541107     inc     edi             ; increment loop 2 ↙
    ↳ counter
.text:00541108     add     ebx, 8
.text:0054110B     cmp     edi, 8
.text:0054110E     jl      short loc_5410F1
.text:00541110     lea     ebx, [esp+50h+internal_array_64]
.text:00541114     mov     edi, 7          ; loop 1 counter is ↙
    ↳ initial state 7
.text:00541119
.text:00541119 loc_541119:
.text:00541119     xor     esi, esi        ; loop 2 counter
.text:0054111B
.text:0054111B loc_54111B:
```

```
.text:0054111B     mov     al, [ebx+esi]   ; get byte from ↙
    ↳ internal array
.text:0054111E     push    eax
.text:0054111F     push    edi             ; loop 1 counter
.text:00541120     push    esi             ; loop 2 counter
.text:00541121     push    ebp             ; arg_0
.text:00541122     call    set_bit
.text:00541127     add     esp, 10h
.text:0054112A     inc     esi             ; increment loop 2 ↙
    ↳ counter
.text:0054112B     cmp     esi, 8
.text:0054112E     jl      short loc_54111B
.text:00541130     dec     edi             ; decrement loop 2 ↙
    ↳ counter
.text:00541131     add     ebx, 8
.text:00541134     cmp     edi, 0FFFFFFFFh
.text:00541137     jg      short loc_541119
.text:00541139     pop     edi
.text:0054113A     pop     esi
.text:0054113B     pop     ebp
.text:0054113C     pop     ebx
.text:0054113D     add     esp, 40h
.text:00541140     retn
.text:00541140 rotate2 endp
```

It is *almost* the same, except of different order of arguments of the `get_bit()` and `set_bit()`. Let's rewrite it in C-like code:

```
void rotate2 (int v)
{
        bool tmp[8][8]; // internal array
        int i, j;

        for (i=0; i<8; i++)
                for (j=0; j<8; j++)
                        tmp[i][j]=get_bit (v, i, j);

        for (i=0; i<8; i++)
                for (j=0; j<8; j++)
                        set_bit (v, j, 7-i, tmp[i][j]);
};
```

Let's also rewrite `rotate3()` function:

```
void rotate3 (int v)
```

```
{
        bool tmp[8][8];
        int i, j;

        for (i=0; i<8; i++)
                for (j=0; j<8; j++)
                        tmp[i][j]=get_bit (i, v, j);

        for (i=0; i<8; i++)
                for (j=0; j<8; j++)
                        set_bit (7-j, v, i, tmp[i][j]);
};
```

Well, now things are simpler. If we consider cube64 as 3D cube 8*8*8, where each element is bit, `get_bit()` and `set_bit()` take just coordinates of bit on input.

rotate1/2/3 functions are in fact rotating all bits in specific plane. Three functions are each for each cube side and `v` argument is setting plane in range 0..7.

Maybe, algorithm's author was thinking of 8*8*8 Rubik's cube [3]?!

Yes, indeed.

Let's get closer into `decrypt()` function, I rewrote it here:

```
void decrypt (BYTE *buf, int sz, char *pw)
{
        char *p=strdup (pw);
        strrev (p);
        int i=0;

        do
        {
                memcpy (cube, buf+i, 8*8);
                rotate_all (p, 3);
                memcpy (buf+i, cube, 8*8);
                i+=64;
        }
        while (i<sz);

        free (p);
};
```

It is almost the same except of `crypt()`, *but* password string is reversed by strrev() [4] standard C function and `rotate_all()` is called with argument 3.

This means, in case of decryption, each corresponding rotate1/2/3 call will be performed thrice.

[3] http://en.wikipedia.org/wiki/Rubik's_Cube

[4] http://msdn.microsoft.com/en-us/library/9hby7w40(VS.80).aspx

This is almost as in Rubik'c cube! If you want to get back, do the same in reverse order and direction! If you need to undo effect of rotating one place in clockwise direction, rotate it thrice in counter-clockwise direction.

`rotate1()` is apparently for rotating "front" plane. `rotate2()` is apparently for rotating "top" plane. `rotate3()` is apparently for rotating "left" plane.

Let's get back to the core of `rotate_all()` function:

```
q=c-'a';
if (q>24)
        q-=24;

int quotient=q/3; // in range 0..7
int remainder=q % 3;

switch (remainder)
{
    case 0: for (int i=0; i<v; i++) rotate1 (quotient); break; ↙
    ↳ // front
    case 1: for (int i=0; i<v; i++) rotate2 (quotient); break; ↙
    ↳ // top
    case 2: for (int i=0; i<v; i++) rotate3 (quotient); break; ↙
    ↳ // left
};
```

Now it is much simpler to understand: each password character defines side (one of three) and plane (one of 8). 3*8 = 24, that is why two last characters of Latin alphabet are remapped to fit an alphabet of exactly 24 elements.

The algorithm is clearly weak: in case of short passwords, one can see, that in crypted file there are an original bytes of the original file in binary files editor.

Here is reconstructed whole source code:

```
#include <windows.h>

#include <stdio.h>
#include <assert.h>

#define IS_SET(flag, bit)       ((flag) & (bit))
#define SET_BIT(var, bit)       ((var) |= (bit))
#define REMOVE_BIT(var, bit)    ((var) &= ~(bit))

static BYTE cube[8][8];

void set_bit (int x, int y, int z, bool bit)
{
        if (bit)
```

```
                SET_BIT (cube[x][y], 1<<z);
        else
                REMOVE_BIT (cube[x][y], 1<<z);
};

bool get_bit (int x, int y, int z)
{
        if ((cube[x][y]>>z)&1==1)
                return true;
        return false;
};

void rotate_f (int row)
{
        bool tmp[8][8];
        int x, y;

        for (x=0; x<8; x++)
                for (y=0; y<8; y++)
                        tmp[x][y]=get_bit (x, y, row);

        for (x=0; x<8; x++)
                for (y=0; y<8; y++)
                        set_bit (y, 7-x, row, tmp[x][y]);
};

void rotate_t (int row)
{
        bool tmp[8][8];
        int y, z;

        for (y=0; y<8; y++)
                for (z=0; z<8; z++)
                        tmp[y][z]=get_bit (row, y, z);

        for (y=0; y<8; y++)
                for (z=0; z<8; z++)
                        set_bit (row, z, 7-y, tmp[y][z]);
};

void rotate_l (int row)
{
        bool tmp[8][8];
```

```
        int x, z;

        for (x=0; x<8; x++)
                for (z=0; z<8; z++)
                        tmp[x][z]=get_bit (x, row, z);

        for (x=0; x<8; x++)
                for (z=0; z<8; z++)
                        set_bit (7-z, row, x, tmp[x][z]);
};

void rotate_all (char *pwd, int v)
{
        char *p=pwd;

        while (*p)
        {
                char c=*p;
                int q;

                c=tolower (c);

                if (c>='a' && c<='z')
                {
                        q=c-'a';
                        if (q>24)
                                q-=24;

                        int quotient=q/3;
                        int remainder=q % 3;

                        switch (remainder)
                        {
                        case 0: for (int i=0; i<v; i++) rotate1 ↙
    ↳ (quotient); break;
                        case 1: for (int i=0; i<v; i++) rotate2 ↙
    ↳ (quotient); break;
                        case 2: for (int i=0; i<v; i++) rotate3 ↙
    ↳ (quotient); break;
                        };
                };

                p++;
```

```
        };
};

void crypt (BYTE *buf, int sz, char *pw)
{
        int i=0;

        do
        {
                memcpy (cube, buf+i, 8*8);
                rotate_all (pw, 1);
                memcpy (buf+i, cube, 8*8);
                i+=64;
        }
        while (i<sz);
};

void decrypt (BYTE *buf, int sz, char *pw)
{
        char *p=strdup (pw);
        strrev (p);
        int i=0;

        do
        {
                memcpy (cube, buf+i, 8*8);
                rotate_all (p, 3);
                memcpy (buf+i, cube, 8*8);
                i+=64;
        }
        while (i<sz);

        free (p);
};

void crypt_file(char *fin, char* fout, char *pw)
{
        FILE *f;
        int flen, flen_aligned;
        BYTE *buf;

        f=fopen(fin, "rb");
```

```
        if (f==NULL)
        {
                printf ("Cannot open input file!\n");
                return;
        };

        fseek (f, 0, SEEK_END);
        flen=ftell (f);
        fseek (f, 0, SEEK_SET);

        flen_aligned=(flen&0xFFFFFFC0)+0x40;

        buf=(BYTE*)malloc (flen_aligned);
        memset (buf, 0, flen_aligned);

        fread (buf, flen, 1, f);

        fclose (f);

        crypt (buf, flen_aligned, pw);

        f=fopen(fout, "wb");

        fwrite ("QR9", 3, 1, f);
        fwrite (&flen, 4, 1, f);
        fwrite (buf, flen_aligned, 1, f);

        fclose (f);

        free (buf);

};

void decrypt_file(char *fin, char* fout, char *pw)
{
        FILE *f;
        int real_flen, flen;
        BYTE *buf;

        f=fopen(fin, "rb");

        if (f==NULL)
        {
```

```
                printf ("Cannot open input file!\n");
                return;
        };

        fseek (f, 0, SEEK_END);
        flen=ftell (f);
        fseek (f, 0, SEEK_SET);

        buf=(BYTE*)malloc (flen);

        fread (buf, flen, 1, f);

        fclose (f);

        if (memcmp (buf, "QR9", 3)!=0)
        {
                printf ("File is not crypted!\n");
                return;
        };

        memcpy (&real_flen, buf+3, 4);

        decrypt (buf+(3+4), flen-(3+4), pw);

        f=fopen(fout, "wb");

        fwrite (buf+(3+4), real_flen, 1, f);

        fclose (f);

        free (buf);
};

// run: input output 0/1 password
// 0 for encrypt, 1 for decrypt

int main(int argc, char *argv[])
{
        if (argc!=5)
        {
                printf ("Incorrect parameters!\n");
                return 1;
        };
```

```
        if (strcmp (argv[3], "0")==0)
                crypt_file (argv[1], argv[2], argv[4]);
        else
                if (strcmp (argv[3], "1")==0)
                        decrypt_file (argv[1], argv[2], argv[4])↙
    ↳ ;
                else
                        printf ("Wrong param %s\n", argv[3]);

        return 0;
};
```

Chapter 57

SAP

57.1 About SAP client network traffic compression

(Tracing connection between TDW_NOCOMPRESS SAPGUI[1] environment variable to the pesky nagging pop-up window and actual data compression routine.)

It is known that network traffic between SAPGUI and SAP is not crypted by default, it is rather compressed (read here[2] and here[3]).

It is also known that by setting environment variable *TDW_NOCOMPRESS* to 1, it is possible to turn network packets compression off.

But you will see a nagging pop-up window cannot be closed:

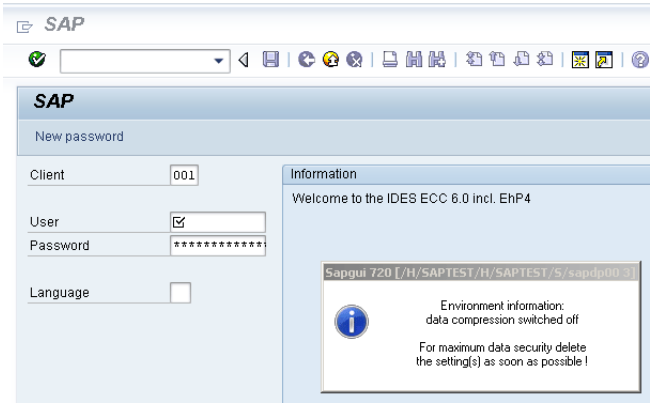


Figure 57.1: Screenshot

[1] SAP GUI client

[2] http://blog.yurichev.com/node/44

[3] http://blog.yurichev.com/node/47

Let's see, if we can remove the window somehow.

But before this, let's see what we already know. First: we know the environment variable *TDW_NOCOMPRESS* is checked somewhere inside of SAPGUI client. Second: string like "data compression switched off" must be present somewhere too. With the help of FAR file manager I found that both of these strings are stored in the SAPguilib.dll file.

So let's open SAPguilib.dll in IDA and search for *"TDW_NOCOMPRESS"* string. Yes, it is present and there is only one reference to it.

We see the following fragment of code (all file offsets are valid for SAPGUI 720 win32, SAPguilib.dll file version 7200,1,0,9009):

```
.text:6440D51B                  lea     eax, [ebp+2108h+var_211C]
.text:6440D51E                  push    eax             ; int
.text:6440D51F                  push    offset aTdw_nocompress ; ↙
    ↳ "TDW_NOCOMPRESS"
.text:6440D524                  mov     byte ptr [edi+15h], 0
.text:6440D528                  call    chk_env
.text:6440D52D                  pop     ecx
.text:6440D52E                  pop     ecx
.text:6440D52F                  push    offset byte_64443AF8
.text:6440D534                  lea     ecx, [ebp+2108h+var_211C]

; demangled name: int ATL::CStringT::Compare(char const *)const
.text:6440D537                  call    ds:mfc90_1603
.text:6440D53D                  test    eax, eax
.text:6440D53F                  jz      short loc_6440D55A
.text:6440D541                  lea     ecx, [ebp+2108h+var_211C]

; demangled name: const char* ATL::CSimpleStringT::operator ↙
    ↳ PCXSTR
.text:6440D544                  call    ds:mfc90_910
.text:6440D54A                  push    eax             ; Str
.text:6440D54B                  call    ds:atoi
.text:6440D551                  test    eax, eax
.text:6440D553                  setnz   al
.text:6440D556                  pop     ecx
.text:6440D557                  mov     [edi+15h], al
```

String returned by `chk_env()` via second argument is then handled by MFC string functions and then `atoi()`[4] is called. After that, numerical value is stored to `edi+15h`.

Also take a look onto `chk_env()` function (I gave a name to it):

```
.text:64413F20 ; int __cdecl chk_env(char *VarName, int)
```

[4] standard C library function, coverting number in string into number

```
.text:64413F20 chk_env         proc near
.text:64413F20
.text:64413F20 DstSize         = dword ptr -0Ch
.text:64413F20 var_8           = dword ptr -8
.text:64413F20 DstBuf          = dword ptr -4
.text:64413F20 VarName         = dword ptr  8
.text:64413F20 arg_4           = dword ptr  0Ch
.text:64413F20
.text:64413F20                 push    ebp
.text:64413F21                 mov     ebp, esp
.text:64413F23                 sub     esp, 0Ch
.text:64413F26                 mov     [ebp+DstSize], 0
.text:64413F2D                 mov     [ebp+DstBuf], 0
.text:64413F34                 push    offset unk_6444C88C
.text:64413F39                 mov     ecx, [ebp+arg_4]

; (demangled name) ATL::CStringT::operator=(char const *)
.text:64413F3C                 call    ds:mfc90_820
.text:64413F42                 mov     eax, [ebp+VarName]
.text:64413F45                 push    eax             ; VarName
.text:64413F46                 mov     ecx, [ebp+DstSize]
.text:64413F49                 push    ecx             ; DstSize
.text:64413F4A                 mov     edx, [ebp+DstBuf]
.text:64413F4D                 push    edx             ; DstBuf
.text:64413F4E                 lea     eax, [ebp+DstSize]
.text:64413F51                 push    eax             ; ↙
    ↳ ReturnSize
.text:64413F52                 call    ds:getenv_s
.text:64413F58                 add     esp, 10h
.text:64413F5B                 mov     [ebp+var_8], eax
.text:64413F5E                 cmp     [ebp+var_8], 0
.text:64413F62                 jz      short loc_64413F68
.text:64413F64                 xor     eax, eax
.text:64413F66                 jmp     short loc_64413FBC
.text:64413F68
.text:64413F68 loc_64413F68:
.text:64413F68                 cmp     [ebp+DstSize], 0
.text:64413F6C                 jnz     short loc_64413F72
.text:64413F6E                 xor     eax, eax
.text:64413F70                 jmp     short loc_64413FBC
.text:64413F72
.text:64413F72 loc_64413F72:
.text:64413F72                 mov     ecx, [ebp+DstSize]
```

```
.text:64413F75                  push    ecx
.text:64413F76                  mov     ecx, [ebp+arg_4]

; demangled name: ATL::CSimpleStringT<char, 1>::Preallocate(int)
.text:64413F79                  call    ds:mfc90_2691
.text:64413F7F                  mov     [ebp+DstBuf], eax
.text:64413F82                  mov     edx, [ebp+VarName]
.text:64413F85                  push    edx             ; VarName
.text:64413F86                  mov     eax, [ebp+DstSize]
.text:64413F89                  push    eax             ; DstSize
.text:64413F8A                  mov     ecx, [ebp+DstBuf]
.text:64413F8D                  push    ecx             ; DstBuf
.text:64413F8E                  lea     edx, [ebp+DstSize]
.text:64413F91                  push    edx             ; ↙
    ↳ ReturnSize
.text:64413F92                  call    ds:getenv_s
.text:64413F98                  add     esp, 10h
.text:64413F9B                  mov     [ebp+var_8], eax
.text:64413F9E                  push    0FFFFFFFFh
.text:64413FA0                  mov     ecx, [ebp+arg_4]

; demangled name: ATL::CSimpleStringT::ReleaseBuffer(int)
.text:64413FA3                  call    ds:mfc90_5835
.text:64413FA9                  cmp     [ebp+var_8], 0
.text:64413FAD                  jz      short loc_64413FB3
.text:64413FAF                  xor     eax, eax
.text:64413FB1                  jmp     short loc_64413FBC
.text:64413FB3
.text:64413FB3 loc_64413FB3:
.text:64413FB3                  mov     ecx, [ebp+arg_4]

; demangled name: const char* ATL::CSimpleStringT::operator ↙
    ↳ PCXSTR
.text:64413FB6                  call    ds:mfc90_910
.text:64413FBC
.text:64413FBC loc_64413FBC:
.text:64413FBC
.text:64413FBC                  mov     esp, ebp
.text:64413FBE                  pop     ebp
.text:64413FBF                  retn
.text:64413FBF chk_env          endp
```

Yes. `getenv_s()`[5] function is Microsoft security-enhanced version of `getenv()`[6].

There is also a MFC string manipulations.

Lots of other environment variables are checked as well. Here is a list of all variables being checked and what SAPGUI could write to trace log when logging is turned on:

DPTRACE	"GUI-OPTION: Trace set to %d"
TDW_HEXDUMP	"GUI-OPTION: Hexdump enabled"
TDW_WORKDIR	"GUI-OPTION: working directory '%s'"
TDW_SPLASHSRCEENOFF	"GUI-OPTION: Splash Screen Off" / "GUI-OPTION: Splash Sc
TDW_REPLYTIMEOUT	"GUI-OPTION: reply timeout %d milliseconds"
TDW_PLAYBACKTIMEOUT	"GUI-OPTION: PlaybackTimeout set to %d milliseconds"
TDW_NOCOMPRESS	"GUI-OPTION: no compression read"
TDW_EXPERT	"GUI-OPTION: expert mode"
TDW_PLAYBACKPROGRESS	"GUI-OPTION: PlaybackProgress"
TDW_PLAYBACKNETTRAFFIC	"GUI-OPTION: PlaybackNetTraffic"
TDW_PLAYLOG	"GUI-OPTION: /PlayLog is YES, file %s"
TDW_PLAYTIME	"GUI-OPTION: /PlayTime set to %d milliseconds"
TDW_LOGFILE	"GUI-OPTION: TDW_LOGFILE '%s'"
TDW_WAN	"GUI-OPTION: WAN - low speed connection enabled"
TDW_FULLMENU	"GUI-OPTION: FullMenu enabled"
SAP_CP / SAP_CODEPAGE	"GUI-OPTION: SAP_CODEPAGE '%d'"
UPDOWNLOAD_CP	"GUI-OPTION: UPDOWNLOAD_CP '%d'"
SNC_PARTNERNAME	"GUI-OPTION: SNC name '%s'"
SNC_QOP	"GUI-OPTION: SNC_QOP '%s'"
SNC_LIB	"GUI-OPTION: SNC is set to: %s"
SAPGUI_INPLACE	"GUI-OPTION: environment variable SAPGUI_INPLACE is on

Settings for each variable are written to the array via pointer in the `EDI` register. `EDI` is being set before the function call:

```
.text:6440EE00                  lea     edi, [ebp+2884h+var_2884]
    ↳  ; options here like +0x15...
.text:6440EE03                  lea     ecx, [esi+24h]
.text:6440EE06                  call    load_command_line
.text:6440EE0B                  mov     edi, eax
.text:6440EE0D                  xor     ebx, ebx
.text:6440EE0F                  cmp     edi, ebx
.text:6440EE11                  jz      short loc_6440EE42
.text:6440EE13                  push    edi
```

[5] http://msdn.microsoft.com/en-us/library/tb2sfw2z(VS.80).aspx

[6] Standard C library returning environment variable

```
.text:6440EE14                  push    offset aSapguiStoppedA ; ↙
    ↳ "Sapgui stopped after commandline interp"...
.text:6440EE19                  push    dword_644F93E8
.text:6440EE1F                  call    FEWTraceError
```

Now, can we find *"data record mode switched on"* string? Yes, and here is the only reference in function `CDwsGui::PrepareInfoWindow()`. How do I know class/method names? There is a lot of special debugging calls writing to log-files like:

```
.text:64405160                  push    dword ptr [esi+2854h]
.text:64405166                  push    offset aCdwsguiPrepare ; ↙
    ↳ "\nCDwsGui::PrepareInfoWindow: sapgui env"...
.text:6440516B                  push    dword ptr [esi+2848h]
.text:64405171                  call    dbg
.text:64405176                  add     esp, 0Ch
```

...or:

```
.text:6440237A                  push    eax
.text:6440237B                  push    offset aCclientStart_6 ; ↙
    ↳ "CClient::Start: set shortcut user to '\%"...
.text:64402380                  push    dword ptr [edi+4]
.text:64402383                  call    dbg
.text:64402388                  add     esp, 0Ch
```

It is **very** useful.

So let's see contents of the pesky nagging pop-up window function:

```
.text:64404F4F CDwsGui__PrepareInfoWindow proc near
.text:64404F4F
.text:64404F4F pvParam          = byte ptr -3Ch
.text:64404F4F var_38           = dword ptr -38h
.text:64404F4F var_34           = dword ptr -34h
.text:64404F4F rc               = tagRECT ptr -2Ch
.text:64404F4F cy               = dword ptr -1Ch
.text:64404F4F h                = dword ptr -18h
.text:64404F4F var_14           = dword ptr -14h
.text:64404F4F var_10           = dword ptr -10h
.text:64404F4F var_4            = dword ptr -4
.text:64404F4F
.text:64404F4F                  push    30h
.text:64404F51                  mov     eax, offset loc_64438E00
.text:64404F56                  call    __EH_prolog3
.text:64404F5B                  mov     esi, ecx        ; ECX is ↙
    ↳ pointer to object
.text:64404F5D                  xor     ebx, ebx
```

```
.text:64404F5F                  lea     ecx, [ebp+var_14]
.text:64404F62                  mov     [ebp+var_10], ebx

; demangled name: ATL::CStringT(void)
.text:64404F65                  call    ds:mfc90_316
.text:64404F6B                  mov     [ebp+var_4], ebx
.text:64404F6E                  lea     edi, [esi+2854h]
.text:64404F74                  push    offset aEnvironmentInf ; ↙
    ↳ "Environment information:\n"
.text:64404F79                  mov     ecx, edi

; demangled name: ATL::CStringT::operator=(char const *)
.text:64404F7B                  call    ds:mfc90_820
.text:64404F81                  cmp     [esi+38h], ebx
.text:64404F84                  mov     ebx, ds:mfc90_2539
.text:64404F8A                  jbe     short loc_64404FA9
.text:64404F8C                  push    dword ptr [esi+34h]
.text:64404F8F                  lea     eax, [ebp+var_14]
.text:64404F92                  push    offset aWorkingDirecto ; ↙
    ↳ "working directory: '\%s'\n"
.text:64404F97                  push    eax

; demangled name: ATL::CStringT::Format(char const *,...)
.text:64404F98                  call    ebx ; mfc90_2539
.text:64404F9A                  add     esp, 0Ch
.text:64404F9D                  lea     eax, [ebp+var_14]
.text:64404FA0                  push    eax
.text:64404FA1                  mov     ecx, edi

; demangled name: ATL::CStringT::operator+=(class ATL::↙
    ↳ CSimpleStringT<char, 1> const &)
.text:64404FA3                  call    ds:mfc90_941
.text:64404FA9
.text:64404FA9 loc_64404FA9:
.text:64404FA9                  mov     eax, [esi+38h]
.text:64404FAC                  test    eax, eax
.text:64404FAE                  jbe     short loc_64404FD3
.text:64404FB0                  push    eax
.text:64404FB1                  lea     eax, [ebp+var_14]
.text:64404FB4                  push    offset aTraceLevelDAct ; ↙
    ↳ "trace level \%d activated\n"
.text:64404FB9                  push    eax
```

```
; demangled name: ATL::CStringT::Format(char const *,...)
.text:64404FBA                call    ebx ; mfc90_2539
.text:64404FBC                add     esp, 0Ch
.text:64404FBF                lea     eax, [ebp+var_14]
.text:64404FC2                push    eax
.text:64404FC3                mov     ecx, edi

; demangled name: ATL::CStringT::operator+=(class ATL::↙
    ↳ CSimpleStringT<char, 1> const &)
.text:64404FC5                call    ds:mfc90_941
.text:64404FCB                xor     ebx, ebx
.text:64404FCD                inc     ebx
.text:64404FCE                mov     [ebp+var_10], ebx
.text:64404FD1                jmp     short loc_64404FD6
.text:64404FD3
.text:64404FD3 loc_64404FD3:
.text:64404FD3                xor     ebx, ebx
.text:64404FD5                inc     ebx
.text:64404FD6
.text:64404FD6 loc_64404FD6:
.text:64404FD6                cmp     [esi+38h], ebx
.text:64404FD9                jbe     short loc_64404FF1
.text:64404FDB                cmp     dword ptr [esi+2978h], 0
.text:64404FE2                jz      short loc_64404FF1
.text:64404FE4                push    offset aHexdumpInTrace ; ↙
    ↳ "hexdump in trace activated\n"
.text:64404FE9                mov     ecx, edi

; demangled name: ATL::CStringT::operator+=(char const *)
.text:64404FEB                call    ds:mfc90_945
.text:64404FF1
.text:64404FF1 loc_64404FF1:
.text:64404FF1
.text:64404FF1                cmp     byte ptr [esi+78h], 0
.text:64404FF5                jz      short loc_64405007
.text:64404FF7                push    offset aLoggingActivat ; ↙
    ↳ "logging activated\n"
.text:64404FFC                mov     ecx, edi

; demangled name: ATL::CStringT::operator+=(char const *)
.text:64404FFE                call    ds:mfc90_945
.text:64405004                mov     [ebp+var_10], ebx
.text:64405007
```

```
.text:64405007 loc_64405007:
.text:64405007                 cmp     byte ptr [esi+3Dh], 0
.text:6440500B                 jz      short bypass
.text:6440500D                 push    offset aDataCompressio ; ↙
    ↳ "data compression switched off\n"
.text:64405012                 mov     ecx, edi

; demangled name: ATL::CStringT::operator+=(char const *)
.text:64405014                 call    ds:mfc90_945
.text:6440501A                 mov     [ebp+var_10], ebx
.text:6440501D
.text:6440501D bypass:
.text:6440501D                 mov     eax, [esi+20h]
.text:64405020                 test    eax, eax
.text:64405022                 jz      short loc_6440503A
.text:64405024                 cmp     dword ptr [eax+28h], 0
.text:64405028                 jz      short loc_6440503A
.text:6440502A                 push    offset aDataRecordMode ; ↙
    ↳ "data record mode switched on\n"
.text:6440502F                 mov     ecx, edi

; demangled name: ATL::CStringT::operator+=(char const *)
.text:64405031                 call    ds:mfc90_945
.text:64405037                 mov     [ebp+var_10], ebx
.text:6440503A
.text:6440503A loc_6440503A:
.text:6440503A
.text:6440503A                 mov     ecx, edi
.text:6440503C                 cmp     [ebp+var_10], ebx
.text:6440503F                 jnz     loc_64405142
.text:64405045                 push    offset aForMaximumData ; ↙
    ↳ "\nFor maximum data security delete\nthe s"...

; demangled name: ATL::CStringT::operator+=(char const *)
.text:6440504A                 call    ds:mfc90_945
.text:64405050                 xor     edi, edi
.text:64405052                 push    edi             ; fWinIni
.text:64405053                 lea     eax, [ebp+pvParam]
.text:64405056                 push    eax             ; pvParam
.text:64405057                 push    edi             ; uiParam
.text:64405058                 push    30h             ; ↙
    ↳ uiAction
.text:6440505A                 call    ds:SystemParametersInfoA
```

```
.text:64405060                  mov     eax, [ebp+var_34]
.text:64405063                  cmp     eax, 1600
.text:64405068                  jle     short loc_64405072
.text:6440506A                  cdq
.text:6440506B                  sub     eax, edx
.text:6440506D                  sar     eax, 1
.text:6440506F                  mov     [ebp+var_34], eax
.text:64405072
.text:64405072 loc_64405072:
.text:64405072                  push    edi             ; hWnd
.text:64405073                  mov     [ebp+cy], 0A0h
.text:6440507A                  call    ds:GetDC
.text:64405080                  mov     [ebp+var_10], eax
.text:64405083                  mov     ebx, 12Ch
.text:64405088                  cmp     eax, edi
.text:6440508A                  jz      loc_64405113
.text:64405090                  push    11h             ; i
.text:64405092                  call    ds:GetStockObject
.text:64405098                  mov     edi, ds:SelectObject
.text:6440509E                  push    eax             ; h
.text:6440509F                  push    [ebp+var_10]    ; hdc
.text:644050A2                  call    edi ; SelectObject
.text:644050A4                  and     [ebp+rc.left], 0
.text:644050A8                  and     [ebp+rc.top], 0
.text:644050AC                  mov     [ebp+h], eax
.text:644050AF                  push    401h            ; format
.text:644050B4                  lea     eax, [ebp+rc]
.text:644050B7                  push    eax             ; lprc
.text:644050B8                  lea     ecx, [esi+2854h]
.text:644050BE                  mov     [ebp+rc.right], ebx
.text:644050C1                  mov     [ebp+rc.bottom], 0B4h

; demangled name: ATL::CSimpleStringT::GetLength(void)
.text:644050C8                  call    ds:mfc90_3178
.text:644050CE                  push    eax             ; cchText
.text:644050CF                  lea     ecx, [esi+2854h]

; demangled name: const char* ATL::CSimpleStringT::operator ↙
    ↳ PCXSTR
.text:644050D5                  call    ds:mfc90_910
.text:644050DB                  push    eax             ; ↙
    ↳ lpchText
.text:644050DC                  push    [ebp+var_10]    ; hdc
```

```
.text:644050DF                  call    ds:DrawTextA
.text:644050E5                  push    4               ; nIndex
.text:644050E7                  call    ds:GetSystemMetrics
.text:644050ED                  mov     ecx, [ebp+rc.bottom]
.text:644050F0                  sub     ecx, [ebp+rc.top]
.text:644050F3                  cmp     [ebp+h], 0
.text:644050F7                  lea     eax, [eax+ecx+28h]
.text:644050FB                  mov     [ebp+cy], eax
.text:644050FE                  jz      short loc_64405108
.text:64405100                  push    [ebp+h]         ; h
.text:64405103                  push    [ebp+var_10]    ; hdc
.text:64405106                  call    edi ; SelectObject
.text:64405108
.text:64405108 loc_64405108:
.text:64405108                  push    [ebp+var_10]    ; hDC
.text:6440510B                  push    0               ; hWnd
.text:6440510D                  call    ds:ReleaseDC
.text:64405113
.text:64405113 loc_64405113:
.text:64405113                  mov     eax, [ebp+var_38]
.text:64405116                  push    80h             ; uFlags
.text:6440511B                  push    [ebp+cy]        ; cy
.text:6440511E                  inc     eax
.text:6440511F                  push    ebx             ; cx
.text:64405120                  push    eax             ; Y
.text:64405121                  mov     eax, [ebp+var_34]
.text:64405124                  add     eax, 0FFFFFED4h
.text:64405129                  cdq
.text:6440512A                  sub     eax, edx
.text:6440512C                  sar     eax, 1
.text:6440512E                  push    eax             ; X
.text:6440512F                  push    0               ; ↙
    ↳ hWndInsertAfter
.text:64405131                  push    dword ptr [esi+285Ch] ; ↙
    ↳ hWnd
.text:64405137                  call    ds:SetWindowPos
.text:6440513D                  xor     ebx, ebx
.text:6440513F                  inc     ebx
.text:64405140                  jmp     short loc_6440514D
.text:64405142
.text:64405142 loc_64405142:
.text:64405142                  push    offset byte_64443AF8
```

```
; demangled name: ATL::CStringT::operator=(char const *)
.text:64405147                call    ds:mfc90_820
.text:6440514D
.text:6440514D loc_6440514D:
.text:6440514D                cmp     dword_6450B970, ebx
.text:64405153                jl      short loc_64405188
.text:64405155                call    sub_6441C910
.text:6440515A                mov     dword_644F858C, ebx
.text:64405160                push    dword ptr [esi+2854h]
.text:64405166                push    offset aCdwsguiPrepare ; ↙
    ↳ "\nCDwsGui::PrepareInfoWindow: sapgui env"...
.text:6440516B                push    dword ptr [esi+2848h]
.text:64405171                call    dbg
.text:64405176                add     esp, 0Ch
.text:64405179                mov     dword_644F858C, 2
.text:64405183                call    sub_6441C920
.text:64405188
.text:64405188 loc_64405188:
.text:64405188                or      [ebp+var_4], 0FFFFFFFFh
.text:6440518C                lea     ecx, [ebp+var_14]

; demangled name: ATL::CStringT::~CStringT()
.text:6440518F                call    ds:mfc90_601
.text:64405195                call    __EH_epilog3
.text:6440519A                retn
.text:6440519A CDwsGui__PrepareInfoWindow endp
```

ECX at function start gets pointer to object (since it is thiscall (29.1.1)-type of function). In our case, the object obviously has class type *CDwsGui*. Depends of option turned on in the object, specific message part will be concatenated to resulting message.

If value at this+0x3D address is not zero, compression is off:

```
.text:64405007 loc_64405007:
.text:64405007                cmp     byte ptr [esi+3Dh], 0
.text:6440500B                jz      short bypass
.text:6440500D                push    offset aDataCompressio ; ↙
    ↳ "data compression switched off\n"
.text:64405012                mov     ecx, edi

; demangled name: ATL::CStringT::operator+=(char const *)
.text:64405014                call    ds:mfc90_945
.text:6440501A                mov     [ebp+var_10], ebx
.text:6440501D
```

```
.text:6440501D bypass:
```

It is interesting, that finally, *var_10* variable state defines whether the message is to be shown at all:

```
.text:6440503C                  cmp     [ebp+var_10], ebx
.text:6440503F                  jnz     exit ; bypass drawing

; add strings "For maximum data security delete" / "the setting(s) as soon as possible !":

.text:64405045                  push    offset aForMaximumData ; "\nFor maximum data security delete\nthe s"...
.text:6440504A                  call    ds:mfc90_945 ; ATL::CStringT::operator+=(char const *)
.text:64405050                  xor     edi, edi
.text:64405052                  push    edi             ; fWinIni
.text:64405053                  lea     eax, [ebp+pvParam]
.text:64405056                  push    eax             ; pvParam
.text:64405057                  push    edi             ; uiParam
.text:64405058                  push    30h             ; uiAction
.text:6440505A                  call    ds:SystemParametersInfoA
.text:64405060                  mov     eax, [ebp+var_34]
.text:64405063                  cmp     eax, 1600
.text:64405068                  jle     short loc_64405072
.text:6440506A                  cdq
.text:6440506B                  sub     eax, edx
.text:6440506D                  sar     eax, 1
.text:6440506F                  mov     [ebp+var_34], eax
.text:64405072
.text:64405072 loc_64405072:

start drawing:

.text:64405072                  push    edi             ; hWnd
.text:64405073                  mov     [ebp+cy], 0A0h
.text:6440507A                  call    ds:GetDC
```

Let's check our theory on practice.

JNZ at this line …

```
.text:6440503F                  jnz     exit ; bypass drawing
```

…replace it with just JMP, and get SAPGUI working without the pesky nagging pop-up window appearing!

Now let's dig deeper and find connection between 0x15 offset in the `load_command_line()` (I gave the name to the function) function and `this+0x3D` variable in the *CDwsGui::PrepareInfo*. Are we sure the value is the same?

I'm starting to search for all occurrences of `0x15` value in code. For a small programs like SAPGUI, it sometimes works. Here is the first occurrence I got:

```
.text:64404C19 sub_64404C19    proc near
.text:64404C19
.text:64404C19 arg_0           = dword ptr  4
.text:64404C19
.text:64404C19                 push    ebx
.text:64404C1A                 push    ebp
.text:64404C1B                 push    esi
.text:64404C1C                 push    edi
.text:64404C1D                 mov     edi, [esp+10h+arg_0]
.text:64404C21                 mov     eax, [edi]
.text:64404C23                 mov     esi, ecx ; ESI/ECX are ↙
    ↳ pointers to some unknown object.
.text:64404C25                 mov     [esi], eax
.text:64404C27                 mov     eax, [edi+4]
.text:64404C2A                 mov     [esi+4], eax
.text:64404C2D                 mov     eax, [edi+8]
.text:64404C30                 mov     [esi+8], eax
.text:64404C33                 lea     eax, [edi+0Ch]
.text:64404C36                 push    eax
.text:64404C37                 lea     ecx, [esi+0Ch]

; demangled name:  ATL::CStringT::operator=(class ATL::CStringT ↙
    ↳ ... &)
.text:64404C3A                 call    ds:mfc90_817
.text:64404C40                 mov     eax, [edi+10h]
.text:64404C43                 mov     [esi+10h], eax
.text:64404C46                 mov     al, [edi+14h]
.text:64404C49                 mov     [esi+14h], al
.text:64404C4C                 mov     al, [edi+15h] ; copy byte↙
    ↳  from 0x15 offset
.text:64404C4F                 mov     [esi+15h], al ; to 0x15 ↙
    ↳ offset in CDwsGui object
```

The function was called from the function named *CDwsGui::CopyOptions*! And thanks again for debugging information.

But the real answer in the function *CDwsGui::Init()*:

```
.text:6440B0BF loc_6440B0BF:
.text:6440B0BF                 mov     eax, [ebp+arg_0]
```

```
.text:6440B0C2                  push    [ebp+arg_4]
.text:6440B0C5                  mov     [esi+2844h], eax
.text:6440B0CB                  lea     eax, [esi+28h] ; ESI is ↙
    ↳ pointer to CDwsGui object
.text:6440B0CE                  push    eax
.text:6440B0CF                  call    CDwsGui__CopyOptions
```

Finally, we understand: array filled in the `load_command_line()` function is actually placed in the *CDwsGui* class but on `this`+0x28 address. 0x15 + 0x28 is exactly 0x3D. OK, we found the point where the value is copied to.

Let's also find other places where 0x3D offset is used. Here is one of them in the *CDwsGui::SapguiRun* function (again, thanks to debugging calls):

```
.text:64409D58                  cmp     [esi+3Dh], bl   ; ESI is ↙
    ↳ pointer to CDwsGui object
.text:64409D5B                  lea     ecx, [esi+2B8h]
.text:64409D61                  setz    al
.text:64409D64                  push    eax             ; arg_10 ↙
    ↳ of CConnectionContext::CreateNetwork
.text:64409D65                  push    dword ptr [esi+64h]

; demangled name: const char* ATL::CSimpleStringT::operator ↙
    ↳ PCXSTR
.text:64409D68                  call    ds:mfc90_910
.text:64409D68                                          ; no ↙
    ↳ arguments
.text:64409D6E                  push    eax
.text:64409D6F                  lea     ecx, [esi+2BCh]

; demangled name: const char* ATL::CSimpleStringT::operator ↙
    ↳ PCXSTR
.text:64409D75                  call    ds:mfc90_910
.text:64409D75                                          ; no ↙
    ↳ arguments
.text:64409D7B                  push    eax
.text:64409D7C                  push    esi
.text:64409D7D                  lea     ecx, [esi+8]
.text:64409D80                  call    ↙
    ↳ CConnectionContext__CreateNetwork
```

Let's check our findings. Replace the `setz al` here to the `xor eax, eax / nop` instructions, clear TDW_NOCOMPRESS environment variable and run SAPGUI. Wow! There is no more pesky nagging window (just as expected, because variable is not set) but in Wireshark we can see the network packets are not compressed anymore! Obviously, this is the point where compression flag is to be set in the *CConnectionContext*

object.

So, compression flag is passed in the 5th argument of function *CConnectionContext::CreateNetwork*. Inside the function, another one is called:

```
...
.text:64403476                  push    [ebp+compression]
.text:64403479                  push    [ebp+arg_C]
.text:6440347C                  push    [ebp+arg_8]
.text:6440347F                  push    [ebp+arg_4]
.text:64403482                  push    [ebp+arg_0]
.text:64403485                  call    CNetwork__CNetwork
```

Compression flag is passing here in the 5th argument to the *CNetwork::CNetwork* constructor.

And here is how *CNetwork* constructor sets a flag in the *CNetwork* object according to the 5th argument **and** an another variable which probably could affect network packets compression too.

```
.text:64411DF1                  cmp     [ebp+compression], esi
.text:64411DF7                  jz      short set_EAX_to_0
.text:64411DF9                  mov     al, [ebx+78h]   ; another↙
    ↳  value may affect compression?
.text:64411DFC                  cmp     al, '3'
.text:64411DFE                  jz      short set_EAX_to_1
.text:64411E00                  cmp     al, '4'
.text:64411E02                  jnz     short set_EAX_to_0
.text:64411E04
.text:64411E04 set_EAX_to_1:
.text:64411E04                  xor     eax, eax
.text:64411E06                  inc     eax             ; EAX -> ↙
    ↳ 1
.text:64411E07                  jmp     short loc_64411E0B
.text:64411E09
.text:64411E09 set_EAX_to_0:
.text:64411E09
.text:64411E09                  xor     eax, eax        ; EAX -> ↙
    ↳ 0
.text:64411E0B
.text:64411E0B loc_64411E0B:
.text:64411E0B                  mov     [ebx+3A4h], eax ; EBX is ↙
    ↳ pointer to CNetwork object
```

At this point we know the compression flag is stored in the *CNetwork* class at *this+0x3A4* address.

Now let's dig across SAPguilib.dll for `0x3A4` value. And here is the second occurrence in the *CDwsGui::OnClientMessageWrite* (endless thanks for debugging information):

```
.text:64406F76 loc_64406F76:
.text:64406F76                 mov     ecx, [ebp+7728h+var_7794]
.text:64406F79                 cmp     dword ptr [ecx+3A4h], 1
.text:64406F80                 jnz     compression_flag_is_zero
.text:64406F86                 mov     byte ptr [ebx+7], 1
.text:64406F8A                 mov     eax, [esi+18h]
.text:64406F8D                 mov     ecx, eax
.text:64406F8F                 test    eax, eax
.text:64406F91                 ja      short loc_64406FFF
.text:64406F93                 mov     ecx, [esi+14h]
.text:64406F96                 mov     eax, [esi+20h]
.text:64406F99
.text:64406F99 loc_64406F99:
.text:64406F99                 push    dword ptr [edi+2868h] ; ↙
    ↳ int
.text:64406F9F                 lea     edx, [ebp+7728h+var_77A4]
.text:64406FA2                 push    edx             ; int
.text:64406FA3                 push    30000           ; int
.text:64406FA8                 lea     edx, [ebp+7728h+Dst]
.text:64406FAB                 push    edx             ; Dst
.text:64406FAC                 push    ecx             ; int
.text:64406FAD                 push    eax             ; Src
.text:64406FAE                 push    dword ptr [edi+28C0h] ; ↙
    ↳ int
.text:64406FB4                 call    sub_644055C5       ; ↙
    ↳ actual compression routine
.text:64406FB9                 add     esp, 1Ch
.text:64406FBC                 cmp     eax, 0FFFFFFF6h
.text:64406FBF                 jz      short loc_64407004
.text:64406FC1                 cmp     eax, 1
.text:64406FC4                 jz      loc_6440708C
.text:64406FCA                 cmp     eax, 2
.text:64406FCD                 jz      short loc_64407004
.text:64406FCF                 push    eax
.text:64406FD0                 push    offset aCompressionErr ; ↙
    ↳ "compression error [rc = \%d]- program wi"...
.text:64406FD5                 push    offset aGui_err_compre ; ↙
    ↳ "GUI_ERR_COMPRESS"
.text:64406FDA                 push    dword ptr [edi+28D0h]
.text:64406FE0                 call    SapPcTxtRead
```

Let's take a look into *sub_644055C5*. In it we can only see call to memcpy() and an other function named (by IDA) *sub_64417440*.

And, let's take a look inside *sub_64417440*. What we see is:

```
.text:6441747C                  push    offset aErrorCsrcompre ; ↙
    ↳ "\nERROR: CsRCompress: invalid handle"
.text:64417481                  call    eax ; dword_644F94C8
.text:64417483                  add     esp, 4
```

Voilà! We've found the function which actually compresses data. As I revealed in past [7], this function is used in SAP and also open-source MaxDB project. So it is available in sources.

Doing last check here:

```
.text:64406F79                  cmp     dword ptr [ecx+3A4h], 1
.text:64406F80                  jnz     compression_flag_is_zero
```

Replace JNZ here for unconditional JMP. Remove environment variable TDW_NOCOMPRES
Voilà! In Wireshark we see the client messages are not compressed. Server responses, however, are compressed.

So we found exact connection between environment variable and the point where data compression routine may be called or may be bypassed.

57.2 SAP 6.0 password checking functions

While returning again to my SAP 6.0 IDES installed in VMware box, I figured out I forgot the password for SAP* account, then it back to my memory, but now I got error message «*Password logon no longer possible - too many failed attempts*», since I've spent all these attempts in trying to recall it.

First extremely good news is the full *disp+work.pdb* file is supplied with SAP, it contain almost everything: function names, structures, types, local variable and argument names, etc. What a lavish gift!

I got TYPEINFODUMP[8] utility for converting PDB files into something readable and grepable.

Here is an example of function information + its arguments + its local variables:

```
FUNCTION ThVmcSysEvent
  Address:         10143190  Size:      675 bytes  Index:     ↙
    ↳ 60483  TypeIndex:    60484
  Type: int NEAR_C ThVmcSysEvent (unsigned int, unsigned char, ↙
    ↳ unsigned short*)
Flags: 0
```

[7] http://conus.info/utils/SAP_pkt_decompr.txt
[8] http://www.debuginfo.com/tools/typeinfodump.html

```
PARAMETER events
  Address: Reg335+288  Size:        4 bytes  Index:    60488  ↙
    ↳ TypeIndex:    60489
  Type: unsigned int
Flags: d0
PARAMETER opcode
  Address: Reg335+296  Size:        1 bytes  Index:    60490  ↙
    ↳ TypeIndex:    60491
  Type: unsigned char
Flags: d0
PARAMETER serverName
  Address: Reg335+304  Size:        8 bytes  Index:    60492  ↙
    ↳ TypeIndex:    60493
  Type: unsigned short*
Flags: d0
STATIC_LOCAL_VAR func
  Address:         12274af0  Size:        8 bytes  Index:    ↙
    ↳ 60495  TypeIndex:    60496
  Type: wchar_t*
Flags: 80
LOCAL_VAR admhead
  Address: Reg335+304  Size:        8 bytes  Index:    60498  ↙
    ↳ TypeIndex:    60499
  Type: unsigned char*
Flags: 90
LOCAL_VAR record
  Address: Reg335+64  Size:      204 bytes  Index:    60501  ↙
    ↳ TypeIndex:    60502
  Type: AD_RECORD
Flags: 90
LOCAL_VAR adlen
  Address: Reg335+296  Size:        4 bytes  Index:    60508  ↙
    ↳ TypeIndex:    60509
  Type: int
Flags: 90
```

And here is an example of some structure:

```
STRUCT DBSL_STMTID
Size: 120  Variables: 4  Functions: 0  Base classes: 0
MEMBER moduletype
  Type:  DBSL_MODULETYPE
  Offset:        0  Index:        3  TypeIndex:    38653
MEMBER module
```

```
  Type:  wchar_t module[40]
  Offset:        4  Index:        3  TypeIndex:      831
MEMBER stmtnum
  Type:  long
  Offset:       84  Index:        3  TypeIndex:      440
MEMBER timestamp
  Type:  wchar_t timestamp[15]
  Offset:       88  Index:        3  TypeIndex:     6612
```

Wow!

Another good news is: *debugging* calls (there are plenty of them) are very useful. Here you can also notice *ct_level* global variable[9], reflecting current trace level. There is a lot of such debugging inclusions in the *disp+work.exe* file:

```
cmp     cs:ct_level, 1
jl      short loc_1400375DA
call    DpLock
lea     rcx, aDpxxtool4_c ; "dpxxtool4.c"
mov     edx, 4Eh        ; line
call    CTrcSaveLocation
mov     r8, cs:func_48
mov     rcx, cs:hdl     ; hdl
lea     rdx, aSDpreadmemvalu ; "%s: DpReadMemValue (%d)"
mov     r9d, ebx
call    DpTrcErr
call    DpUnlock
```

If current trace level is bigger or equal to threshold defined in the code here, debugging message will be written to log files like *dev_w0*, *dev_disp*, and other *dev** files.

Let's do grepping on file I got with the help of TYPEINFODUMP utility:

```
cat "disp+work.pdb.d" | grep FUNCTION | grep -i password
```

I got:

```
FUNCTION rcui::AgiPassword::DiagISelection
FUNCTION ssf_password_encrypt
FUNCTION ssf_password_decrypt
FUNCTION password_logon_disabled
FUNCTION dySignSkipUserPassword
FUNCTION migrate_password_history
FUNCTION password_is_initial
```

[9] More about trace level: http://help.sap.com/saphelp_nwpi71/helpdata/en/46/962416a5a613e8e10000000a155369/content.htm

```
FUNCTION rcui::AgiPassword::IsVisible
FUNCTION password_distance_ok
FUNCTION get_password_downwards_compatibility
FUNCTION dySignUnSkipUserPassword
FUNCTION rcui::AgiPassword::GetTypeName
FUNCTION `rcui::AgiPassword::AgiPassword'::`1'::dtor$2
FUNCTION `rcui::AgiPassword::AgiPassword'::`1'::dtor$0
FUNCTION `rcui::AgiPassword::AgiPassword'::`1'::dtor$1
FUNCTION usm_set_password
FUNCTION rcui::AgiPassword::TraceTo
FUNCTION days_since_last_password_change
FUNCTION rsecgrp_generate_random_password
FUNCTION rcui::AgiPassword::`scalar deleting destructor'
FUNCTION password_attempt_limit_exceeded
FUNCTION handle_incorrect_password
FUNCTION `rcui::AgiPassword::`scalar deleting destructor↙
    ↳ ''::`1'::dtor$1
FUNCTION calculate_new_password_hash
FUNCTION shift_password_to_history
FUNCTION rcui::AgiPassword::GetType
FUNCTION found_password_in_history
FUNCTION `rcui::AgiPassword::`scalar deleting destructor↙
    ↳ ''::`1'::dtor$0
FUNCTION rcui::AgiObj::IsaPassword
FUNCTION password_idle_check
FUNCTION SlicHwPasswordForDay
FUNCTION rcui::AgiPassword::IsaPassword
FUNCTION rcui::AgiPassword::AgiPassword
FUNCTION delete_user_password
FUNCTION usm_set_user_password
FUNCTION Password_API
FUNCTION get_password_change_for_SSO
FUNCTION password_in_USR40
FUNCTION rsec_agrp_abap_generate_random_password
```

Let's also try to search for debug messages which contain words *«password»* and *«locked»*. One of them is the string *«user was locked by subsequently failed password logon attempts»* referenced in
function *password_attempt_limit_exceeded()*.

Other string this function I found may write to log file are: *«password logon attempt will be rejected immediately (preventing dictionary attacks)»*, *«failed-logon lock: expired (but not removed due to 'read-only' operation)»*, *«failed-logon lock: expired => removed»*.

After playing for a little with this function, I quickly noticed the problem is exactly

in it. It is called from *chckpass()* function —one of the password checking functions.

First, I would like to be sure I'm at the correct point:

Run my tracer:

```
tracer64.exe -a:disp+work.exe bpf=disp+work.exe!chckpass,args:3,↙
    ↳ unicode
```

```
PID=2236|TID=2248|(0) disp+work.exe!chckpass (0x202c770, L"↙
    ↳ Brewered1                                  ", 0x41) (called ↙
    ↳ from 0x1402f1060 (disp+work.exe!usrexist+0x3c0))
PID=2236|TID=2248|(0) disp+work.exe!chckpass -> 0x35
```

Call path is: *syssigni() -> DylSigni() -> dychkusr() -> usrexist() -> chckpass()*.

Number 0x35 is an error returning in *chckpass()* at that point:

```
.text:00000001402ED567 loc_1402ED567:                          ;↙
    ↳  CODE XREF: chckpass+B4
.text:00000001402ED567                  mov     rcx, rbx       ;↙
    ↳  usr02
.text:00000001402ED56A                  call    ↙
    ↳ password_idle_check
.text:00000001402ED56F                  cmp     eax, 33h
.text:00000001402ED572                  jz      loc_1402EDB4E
.text:00000001402ED578                  cmp     eax, 36h
.text:00000001402ED57B                  jz      loc_1402EDB3D
.text:00000001402ED581                  xor     edx, edx       ;↙
    ↳  usr02_readonly
.text:00000001402ED583                  mov     rcx, rbx       ;↙
    ↳  usr02
.text:00000001402ED586                  call    ↙
    ↳ password_attempt_limit_exceeded
.text:00000001402ED58B                  test    al, al
.text:00000001402ED58D                  jz      short ↙
    ↳ loc_1402ED5A0
.text:00000001402ED58F                  mov     eax, 35h
.text:00000001402ED594                  add     rsp, 60h
.text:00000001402ED598                  pop     r14
.text:00000001402ED59A                  pop     r12
.text:00000001402ED59C                  pop     rdi
.text:00000001402ED59D                  pop     rsi
.text:00000001402ED59E                  pop     rbx
.text:00000001402ED59F                  retn
```

Fine, let's check:

```
tracer64.exe -a:disp+work.exe bpf=disp+work.exe!↙
    ↳ password_attempt_limit_exceeded,args:4,unicode,rt:0
```

```
PID=2744|TID=360|(0) disp+work.exe!↙
    ↳ password_attempt_limit_exceeded (0x202c770, 0, 0x257758, ↙
    ↳ 0) (called from 0x1402ed58b (disp+work.exe!chckpass+0xeb))
PID=2744|TID=360|(0) disp+work.exe!↙
    ↳ password_attempt_limit_exceeded -> 1
PID=2744|TID=360|We modify return value (EAX/RAX) of this ↙
    ↳ function to 0
PID=2744|TID=360|(0) disp+work.exe!↙
    ↳ password_attempt_limit_exceeded (0x202c770, 0, 0, 0) (↙
    ↳ called from 0x1402e9794 (disp+work.exe!chngpass+0xe4))
PID=2744|TID=360|(0) disp+work.exe!↙
    ↳ password_attempt_limit_exceeded -> 1
PID=2744|TID=360|We modify return value (EAX/RAX) of this ↙
    ↳ function to 0
```

Excellent! I can successfully login now.

By the way, if I try to pretend I forgot the password, fixing *chckpass()* function return value at 0 is enough to bypass check:

```
tracer64.exe -a:disp+work.exe bpf=disp+work.exe!chckpass,args:3,↙
    ↳ unicode,rt:0
```

```
PID=2744|TID=360|(0) disp+work.exe!chckpass (0x202c770, L"bogus ↙
    ↳                                ", 0x41) (called from 0↙
    ↳ x1402f1060 (disp+work.exe!usrexist+0x3c0))
PID=2744|TID=360|(0) disp+work.exe!chckpass -> 0x35
PID=2744|TID=360|We modify return value (EAX/RAX) of this ↙
    ↳ function to 0
```

What also can be said while analyzing *password_attempt_limit_exceeded()* function is that at the very beginning of it, this call might be seen:

```
lea     rcx, aLoginFailed_us ; "login/failed_user_auto_unlock"
call    sapgparam
test    rax, rax
jz      short loc_1402E19DE
movzx   eax, word ptr [rax]
cmp     ax, 'N'
jz      short loc_1402E19D4
cmp     ax, 'n'
jz      short loc_1402E19D4
```

```
cmp     ax, '0'
jnz     short loc_1402E19DE
```

Obviously, function *sapgparam()* used to query value of some configuration parameter. This function can be called from 1768 different places. It seems, with the help of this information, we can easily find places in code, control flow of which can be affected by specific configuration parameters.

It is really sweet. Function names are very clear, much clearer than in the Oracle RDBMS. It seems, *disp+work* process written in C++. It was apparently rewritten some time ago?

Chapter 58

Oracle RDBMS

58.1 V$VERSION table in the Oracle RDBMS

Oracle RDBMS 11.2 is a huge program, main module `oracle.exe` contain approx. 124,000 functions. For comparison, Windows 7 x86 kernel (ntoskrnl.exe) —approx. 11,000 functions and Linux 3.9.8 kernel (with default drivers compiled) —31,000 functions.

Let's start with an easy question. Where Oracle RDBMS get all this information, when we execute such simple statement in SQL*Plus:

```
SQL> select * from V$VERSION;
```

And we've got:

```
BANNER
--------------------------------------------------------------------------------
    ↳

Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - ↙
    ↳ Production
PL/SQL Release 11.2.0.1.0 - Production
CORE    11.2.0.1.0      Production
TNS for 32-bit Windows: Version 11.2.0.1.0 - Production
NLSRTL Version 11.2.0.1.0 - Production
```

Let's start. Where in the Oracle RDBMS we may find a string `V$VERSION`?

As of win32-version, `oracle.exe` file contain the string, which can be investigated easily. But we can also use object (.o) files from Linux version of Oracle RDBMS since, unlike win32 version `oracle.exe`, function names (and global variables as well) are preserved there.

So, kqf.o file contain V$VERSION string. The object file is in the main Oracle-library libserver11.a.

A reference to this text string we may find in the kqfviw table stored in the same file, kqf.o:

Listing 58.1: kqf.o

```
.rodata:0800C4A0 kqfviw          dd 0Bh                  ; DATA ↙
    ↘ XREF: kqfchk:loc_8003A6D
.rodata:0800C4A0                                         ; ↙
    ↘ kqfgbn+34
.rodata:0800C4A4                 dd offset _2__STRING_10102_0 ; ↙
    ↘ "GV$WAITSTAT"
.rodata:0800C4A8                 dd 4
.rodata:0800C4AC                 dd offset _2__STRING_10103_0 ; ↙
    ↘ "NULL"
.rodata:0800C4B0                 dd 3
.rodata:0800C4B4                 dd 0
.rodata:0800C4B8                 dd 195h
.rodata:0800C4BC                 dd 4
.rodata:0800C4C0                 dd 0
.rodata:0800C4C4                 dd 0FFFFFC1CBh
.rodata:0800C4C8                 dd 3
.rodata:0800C4CC                 dd 0
.rodata:0800C4D0                 dd 0Ah
.rodata:0800C4D4                 dd offset _2__STRING_10104_0 ; ↙
    ↘ "V$WAITSTAT"
.rodata:0800C4D8                 dd 4
.rodata:0800C4DC                 dd offset _2__STRING_10103_0 ; ↙
    ↘ "NULL"
.rodata:0800C4E0                 dd 3
.rodata:0800C4E4                 dd 0
.rodata:0800C4E8                 dd 4Eh
.rodata:0800C4EC                 dd 3
.rodata:0800C4F0                 dd 0
.rodata:0800C4F4                 dd 0FFFFC003h
.rodata:0800C4F8                 dd 4
.rodata:0800C4FC                 dd 0
.rodata:0800C500                 dd 5
.rodata:0800C504                 dd offset _2__STRING_10105_0 ; ↙
    ↘ "GV$BH"
.rodata:0800C508                 dd 4
.rodata:0800C50C                 dd offset _2__STRING_10103_0 ; ↙
    ↘ "NULL"
.rodata:0800C510                 dd 3
```

```
.rodata:0800C514                dd 0
.rodata:0800C518                dd 269h
.rodata:0800C51C                dd 15h
.rodata:0800C520                dd 0
.rodata:0800C524                dd 0FFFFC1EDh
.rodata:0800C528                dd 8
.rodata:0800C52C                dd 0
.rodata:0800C530                dd 4
.rodata:0800C534                dd offset _2__STRING_10106_0 ; ↙
    ↳ "V$BH"
.rodata:0800C538                dd 4
.rodata:0800C53C                dd offset _2__STRING_10103_0 ; ↙
    ↳ "NULL"
.rodata:0800C540                dd 3
.rodata:0800C544                dd 0
.rodata:0800C548                dd 0F5h
.rodata:0800C54C                dd 14h
.rodata:0800C550                dd 0
.rodata:0800C554                dd 0FFFFC1EEh
.rodata:0800C558                dd 5
.rodata:0800C55C                dd 0
```

By the way, often, while analysing Oracle RDBMS internals, you may ask yourself, why functions and global variable names are so weird. Supposedly, since Oracle RDBMS is very old product and was developed in C in 1980-s. And that was a time when C standard guaranteed function names/variables support only up to 6 characters inclusive: «6 significant initial characters in an external identifier»[1]

Probably, the table `kqfviw` contain most (maybe even all) views prefixed with V$, these are *fixed views*, present all the time. Superficially, by noticing cyclic recurrence of data, we can easily see that each `kqfviw` table element has 12 32-bit fields. It is very simple to create a 12-elements structure in IDA and apply it to all table elements. As of Oracle RDBMS version 11.2, there are 1023 table elements, i.e., there are described 1023 of all possible *fixed views*. We will return to this number later.

As we can see, there is not much information in these numbers in fields. The very first number is always equals to name of view (without terminating zero. This is correct for each element. But this information is not very useful.

We also know that information about all fixed views can be retrieved from *fixed view* named `V$FIXED_VIEW_DEFINITION` (by the way, the information for this view is also taken from `kqfviw` and `kqfvip` tables.) By the way, there are 1023 elements too.

```
SQL> select * from V$FIXED_VIEW_DEFINITION where view_name='↙
    ↳ V$VERSION';
```

[1] Draft ANSI C Standard (ANSI X3J11/88-090) (May 13, 1988)

```
VIEW_NAME
------------------------------
VIEW_DEFINITION
--------------------------------------------------------------------------------
    ↳

V$VERSION
select  BANNER from GV$VERSION where inst_id = USERENV('Instance↙
    ↳ ')
```

So, V$VERSION is some kind of *thunk view* for another view, named GV$VERSION, which is, in turn:

```
SQL> select * from V$FIXED_VIEW_DEFINITION where view_name='↙
    ↳ GV$VERSION';

VIEW_NAME
------------------------------
VIEW_DEFINITION
--------------------------------------------------------------------------------
    ↳

GV$VERSION
select inst_id, banner from x$version
```

Tables prefixed as X$ in the Oracle RDBMS– is service tables too, undocumented, cannot be changed by user and refreshed dynamically.

Let's also try to search the text `select BANNER from GV$VERSION where inst_id = USERENV('Instance')` in the `kqf.o` file and we find it in the `kqfvip` table:

.

Listing 58.2: kqf.o

```
rodata:080185A0 kqfvip          dd offset _2__STRING_11126_0 ; ↙
    ↳ DATA XREF: kqfgvcn+18
.rodata:080185A0                                             ; ↙
    ↳ kqfgvt+F
.rodata:080185A0                                             ; "↙
    ↳ select inst_id,decode(indx,1,'data bloc"...
.rodata:080185A4                 dd offset kqfv459_c_0
.rodata:080185A8                 dd 0
.rodata:080185AC                 dd 0

...
```

```
.rodata:08019570                dd offset _2__STRING_11378_0 ; ↙
    ↳ "select  BANNER from GV$VERSION where in"...
.rodata:08019574                dd offset kqfv133_c_0
.rodata:08019578                dd 0
.rodata:0801957C                dd 0
.rodata:08019580                dd offset _2__STRING_11379_0 ; ↙
    ↳ "select inst_id,decode(bitand(cfflg,1),0"...
.rodata:08019584                dd offset kqfv403_c_0
.rodata:08019588                dd 0
.rodata:0801958C                dd 0
.rodata:08019590                dd offset _2__STRING_11380_0 ; ↙
    ↳ "select  STATUS , NAME, IS_RECOVERY_DEST"...
.rodata:08019594                dd offset kqfv199_c_0
```

The table appear to have 4 fields in each element. By the way, there are 1023 elements too. The second field pointing to another table, containing table fields for this *fixed view*. As of V$VERSION, this table contain only two elements, first is 6 and second is BANNER string (the number (6) is this string length) and after, *terminating* element contain 0 and *null* C-string:

Listing 58.3: kqf.o

```
.rodata:080BBAC4 kqfv133_c_0    dd 6                    ; DATA ↙
    ↳ XREF: .rodata:08019574
.rodata:080BBAC8                dd offset _2__STRING_5017_0 ; "↙
    ↳ BANNER"
.rodata:080BBACC                dd 0
.rodata:080BBAD0                dd offset _2__STRING_0_0
```

By joining data from both kqfviw and kqfvip tables, we may get SQL-statements which are executed when user wants to query information from specific *fixed view*.

So I wrote an oracle tables[2] program, so to gather all this information from Oracle RDBMS for Linux object files. For V$VERSION, we may find this:

Listing 58.4: Result of oracle tables

```
kqfviw_element.viewname: [V$VERSION] ?: 0x3 0x43 0x1 0xffffc085 ↙
    ↳ 0x4
kqfvip_element.statement: [select  BANNER from GV$VERSION where ↙
    ↳ inst_id = USERENV('Instance')]
kqfvip_element.params:
[BANNER]
```

and:

[2] http://yurichev.com/oracle_tables.html

Listing 58.5: Result of oracle tables

```
kqfviw_element.viewname: [GV$VERSION] ?: 0x3 0x26 0x2 0xffffc192↙
    ↳  0x1
kqfvip_element.statement: [select inst_id, banner from x$version↙
    ↳ ]
kqfvip_element.params:
[INST_ID] [BANNER]
```

GV$VERSION *fixed view* is distinct from V$VERSION in only that way that it contains one more field with *instance* identifier. Anyway, we stuck at the table X$VERSION. Just like any other X$-tables, it is undocumented, however, we can query it:

```
SQL> select * from x$version;

ADDR            INDX    INST_ID
-------- ---------- ----------
BANNER
--------------------------------------------------------------------------------
    ↳

0DBAF574          0          1
Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - ↙
    ↳ Production


...
```

This table has additional fields like ADDR and INDX.

While scrolling kqf.o in IDA we may spot another table containing pointer to the X$VERSION string, this is kqftab:

Listing 58.6: kqf.o

```
.rodata:0803CAC0                dd 9                    ; ↙
    ↳ element number 0x1f6
.rodata:0803CAC4                dd offset _2__STRING_13113_0 ; ↙
    ↳ "X$VERSION"
.rodata:0803CAC8                dd 4
.rodata:0803CACC                dd offset _2__STRING_13114_0 ; ↙
    ↳ "kqvt"
.rodata:0803CAD0                dd 4
.rodata:0803CAD4                dd 4
.rodata:0803CAD8                dd 0
.rodata:0803CADC                dd 4
.rodata:0803CAE0                dd 0Ch
.rodata:0803CAE4                dd 0FFFFC075h
```

```
.rodata:0803CAE8                dd 3
.rodata:0803CAEC                dd 0
.rodata:0803CAF0                dd 7
.rodata:0803CAF4                dd offset _2__STRING_13115_0 ; ↙
    ↳ "X$KQFSZ"
.rodata:0803CAF8                dd 5
.rodata:0803CAFC                dd offset _2__STRING_13116_0 ; ↙
    ↳ "kqfsz"
.rodata:0803CB00                dd 1
.rodata:0803CB04                dd 38h
.rodata:0803CB08                dd 0
.rodata:0803CB0C                dd 7
.rodata:0803CB10                dd 0
.rodata:0803CB14                dd 0FFFFC09Dh
.rodata:0803CB18                dd 2
.rodata:0803CB1C                dd 0
```

There are a lot of references to X$-table names, apparently, to all Oracle RDBMS 11.2 X$-tables. But again, we have not enough information. I have no idea, what `kqvt` string means. `kq` prefix may means *kernel* and *query*. `v`, apparently, means *version* and `t` —*type*? Frankly speaking, I do not know.

The table named similarly can be found in `kqf.o`:

Listing 58.7: kqf.o

```
.rodata:0808C360 kqvt_c_0       kqftap_param <4, offset ↙
    ↳ _2__STRING_19_0, 917h, 0, 0, 0, 4, 0, 0>
.rodata:0808C360                                        ; DATA ↙
    ↳ XREF: .rodata:08042680
.rodata:0808C360                                        ; "ADDR↙
    ↳ "
.rodata:0808C384                kqftap_param <4, offset ↙
    ↳ _2__STRING_20_0, 0B02h, 0, 0, 0, 4, 0, 0> ; "INDX"
.rodata:0808C3A8                kqftap_param <7, offset ↙
    ↳ _2__STRING_21_0, 0B02h, 0, 0, 0, 4, 0, 0> ; "INST_ID"
.rodata:0808C3CC                kqftap_param <6, offset ↙
    ↳ _2__STRING_5017_0, 601h, 0, 0, 0, 50h, 0, 0> ; "BANNER"
.rodata:0808C3F0                kqftap_param <0, offset ↙
    ↳ _2__STRING_0_0, 0, 0, 0, 0, 0, 0, 0>
```

It contain information about all fields in the `X$VERSION` table. The only reference to this table present in the `kqftap` table:

Listing 58.8: kqf.o

```
.rodata:08042680                    kqftap_element <0, offset ↙
    ↳ kqvt_c_0, offset kqvrow, 0> ; element 0x1f6
```

It is interesting that this element here is 0x1f6th (502nd), just as a pointer to the X$VERSION string in the kqftab table. Probably, kqftap and kqftab tables are complement each other, just like kqfvip and kqfviw. We also see a pointer to the kqvrow() function. Finally, we got something useful!

So I added these tables to my oracle tables[3] utility too. For X$VERSION I've got:

Listing 58.9: Result of oracle tables

```
kqftab_element.name: [X$VERSION] ?: [kqvt] 0x4 0x4 0x4 0xc 0↙
    ↳ xffffc075 0x3
kqftap_param.name=[ADDR] ?: 0x917 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[INDX] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[INST_ID] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[BANNER] ?: 0x601 0x0 0x0 0x0 0x50 0x0 0x0
kqftap_element.fn1=kqvrow
kqftap_element.fn2=NULL
```

With the help of tracer, it is easy to check that this function called 6 times in row (from the qerfxFetch() function) while querying X$VERSION table.

Let's run tracer in the cc mode (it will comment each executed instruction):

```
tracer -a:oracle.exe bpf=oracle.exe!_kqvrow,trace:cc
```

```
_kqvrow_  proc near

var_7C    = byte ptr -7Ch
var_18    = dword ptr -18h
var_14    = dword ptr -14h
Dest      = dword ptr -10h
var_C     = dword ptr -0Ch
var_8     = dword ptr -8
var_4     = dword ptr -4
arg_8     = dword ptr  10h
arg_C     = dword ptr  14h
arg_14    = dword ptr  1Ch
arg_18    = dword ptr  20h

; FUNCTION CHUNK AT .text1:056C11A0 SIZE 00000049 BYTES

          push    ebp
          mov     ebp, esp
```

[3] http://yurichev.com/oracle_tables.html

```
          sub     esp, 7Ch
          mov     eax, [ebp+arg_14] ; [EBP+1Ch]=1
          mov     ecx, TlsIndex   ; [69AEB08h]=0
          mov     edx, large fs:2Ch
          mov     edx, [edx+ecx*4] ; [EDX+ECX*4]=0xc98c938
          cmp     eax, 2          ; EAX=1
          mov     eax, [ebp+arg_8] ; [EBP+10h]=0xcdfe554
          jz      loc_2CE1288
          mov     ecx, [eax]      ; [EAX]=0..5
          mov     [ebp+var_4], edi ; EDI=0xc98c938

loc_2CE10F6: ; CODE XREF: _kqvrow_+10A
             ; _kqvrow_+1A9
          cmp     ecx, 5          ; ECX=0..5
          ja      loc_56C11C7
          mov     edi, [ebp+arg_18] ; [EBP+20h]=0
          mov     [ebp+var_14], edx ; EDX=0xc98c938
          mov     [ebp+var_8], ebx ; EBX=0
          mov     ebx, eax        ; EAX=0xcdfe554
          mov     [ebp+var_C], esi ; ESI=0xcdfe248

loc_2CE110D: ; CODE XREF: _kqvrow_+29E00E6
          mov     edx, ds:off_628B09C[ecx*4] ; [ECX*4+628B09Ch↙
    ↳ ]=0x2ce1116, 0x2ce11ac, 0x2ce11db, 0x2ce11f6, 0x2ce1236, 0↙
    ↳ x2ce127a
          jmp     edx             ; EDX=0x2ce1116, 0x2ce11ac, 0↙
    ↳ x2ce11db, 0x2ce11f6, 0x2ce1236, 0x2ce127a

loc_2CE1116: ; DATA XREF: .rdata:off_628B09C
          push    offset aXKqvvsnBuffer ; "x$kqvvsn buffer"
          mov     ecx, [ebp+arg_C] ; [EBP+14h]=0x8a172b4
          xor     edx, edx
          mov     esi, [ebp+var_14] ; [EBP-14h]=0xc98c938
          push    edx             ; EDX=0
          push    edx             ; EDX=0
          push    50h
          push    ecx             ; ECX=0x8a172b4
          push    dword ptr [esi+10494h] ; [ESI+10494h]=0↙
    ↳ xc98cd58
          call    _kghalf         ; tracing nested maximum level↙
    ↳  (1) reached, skipping this CALL
          mov     esi, ds:__imp__vsnnum ; [59771A8h]=0x61bc49e0
          mov     [ebp+Dest], eax ; EAX=0xce2ffb0
```

```
        mov     [ebx+8], eax    ; EAX=0xce2ffb0
        mov     [ebx+4], eax    ; EAX=0xce2ffb0
        mov     edi, [esi]      ; [ESI]=0xb200100
        mov     esi, ds:__imp__vsnstr ; [597D6D4h]=0x65852148,↙
    ↳  "- Production"
        push    esi             ; ESI=0x65852148, "- ↙
    ↳ Production"
        mov     ebx, edi        ; EDI=0xb200100
        shr     ebx, 18h        ; EBX=0xb200100
        mov     ecx, edi        ; EDI=0xb200100
        shr     ecx, 14h        ; ECX=0xb200100
        and     ecx, 0Fh        ; ECX=0xb2
        mov     edx, edi        ; EDI=0xb200100
        shr     edx, 0Ch        ; EDX=0xb200100
        movzx   edx, dl         ; DL=0
        mov     eax, edi        ; EDI=0xb200100
        shr     eax, 8          ; EAX=0xb200100
        and     eax, 0Fh        ; EAX=0xb2001
        and     edi, 0FFh       ; EDI=0xb200100
        push    edi             ; EDI=0
        mov     edi, [ebp+arg_18] ; [EBP+20h]=0
        push    eax             ; EAX=1
        mov     eax, ds:__imp__vsnban ; [597D6D8h]=0x65852100,↙
    ↳  "Oracle Database 11g Enterprise Edition Release %d.%d.%d↙
    ↳ .%d.%d %s"
        push    edx             ; EDX=0
        push    ecx             ; ECX=2
        push    ebx             ; EBX=0xb
        mov     ebx, [ebp+arg_8] ; [EBP+10h]=0xcdfe554
        push    eax             ; EAX=0x65852100, "Oracle ↙
    ↳ Database 11g Enterprise Edition Release %d.%d.%d.%d.%d %s"
        mov     eax, [ebp+Dest] ; [EBP-10h]=0xce2ffb0
        push    eax             ; EAX=0xce2ffb0
        call    ds:__imp__sprintf ; op1=MSVCR80.dll!sprintf ↙
    ↳ tracing nested maximum level (1) reached, skipping this ↙
    ↳ CALL
        add     esp, 38h
        mov     dword ptr [ebx], 1

loc_2CE1192: ; CODE XREF: _kqvrow_+FB
             ; _kqvrow_+128 ...
        test    edi, edi        ; EDI=0
        jnz     __VInfreq__kqvrow
```

```
          mov     esi, [ebp+var_C] ; [EBP-0Ch]=0xcdfe248
          mov     edi, [ebp+var_4] ; [EBP-4]=0xc98c938
          mov     eax, ebx         ; EBX=0xcdfe554
          mov     ebx, [ebp+var_8] ; [EBP-8]=0
          lea     eax, [eax+4]     ; [EAX+4]=0xce2ffb0, "NLSRTL ↙
    ↘ Version 11.2.0.1.0 - Production", "Oracle Database 11g ↙
    ↘ Enterprise Edition Release 11.2.0.1.0 - Production", "PL/↙
    ↘ SQL Release 11.2.0.1.0 - Production", "TNS for 32-bit ↙
    ↘ Windows: Version 11.2.0.1.0 - Production"

loc_2CE11A8: ; CODE XREF: _kqvrow_+29E00F6
          mov     esp, ebp
          pop     ebp
          retn                     ; EAX=0xcdfe558

loc_2CE11AC: ; DATA XREF: .rdata:0628B0A0
          mov     edx, [ebx+8]     ; [EBX+8]=0xce2ffb0, "Oracle ↙
    ↘ Database 11g Enterprise Edition Release 11.2.0.1.0 - ↙
    ↘ Production"
          mov     dword ptr [ebx], 2
          mov     [ebx+4], edx     ; EDX=0xce2ffb0, "Oracle ↙
    ↘ Database 11g Enterprise Edition Release 11.2.0.1.0 - ↙
    ↘ Production"
          push    edx              ; EDX=0xce2ffb0, "Oracle ↙
    ↘ Database 11g Enterprise Edition Release 11.2.0.1.0 - ↙
    ↘ Production"
          call    _kkxvsn          ; tracing nested maximum level↙
    ↘  (1) reached, skipping this CALL
          pop     ecx
          mov     edx, [ebx+4]     ; [EBX+4]=0xce2ffb0, "PL/SQL ↙
    ↘ Release 11.2.0.1.0 - Production"
          movzx   ecx, byte ptr [edx] ; [EDX]=0x50
          test    ecx, ecx         ; ECX=0x50
          jnz     short loc_2CE1192
          mov     edx, [ebp+var_14]
          mov     esi, [ebp+var_C]
          mov     eax, ebx
          mov     ebx, [ebp+var_8]
          mov     ecx, [eax]
          jmp     loc_2CE10F6

loc_2CE11DB: ; DATA XREF: .rdata:0628B0A4
          push    0
```

```
          push    50h
          mov     edx, [ebx+8]    ; [EBX+8]=0xce2ffb0, "PL/SQL ↙
    ↳ Release 11.2.0.1.0 - Production"
          mov     [ebx+4], edx    ; EDX=0xce2ffb0, "PL/SQL ↙
    ↳ Release 11.2.0.1.0 - Production"
          push    edx             ; EDX=0xce2ffb0, "PL/SQL ↙
    ↳ Release 11.2.0.1.0 - Production"
          call    _lmxver         ; tracing nested maximum level↙
    ↳  (1) reached, skipping this CALL
          add     esp, 0Ch
          mov     dword ptr [ebx], 3
          jmp     short loc_2CE1192

loc_2CE11F6: ; DATA XREF: .rdata:0628B0A8
          mov     edx, [ebx+8]    ; [EBX+8]=0xce2ffb0
          mov     [ebp+var_18], 50h
          mov     [ebx+4], edx    ; EDX=0xce2ffb0
          push    0
          call    _npinli         ; tracing nested maximum level↙
    ↳  (1) reached, skipping this CALL
          pop     ecx
          test    eax, eax        ; EAX=0
          jnz     loc_56C11DA
          mov     ecx, [ebp+var_14] ; [EBP-14h]=0xc98c938
          lea     edx, [ebp+var_18] ; [EBP-18h]=0x50
          push    edx             ; EDX=0xd76c93c
          push    dword ptr [ebx+8] ; [EBX+8]=0xce2ffb0
          push    dword ptr [ecx+13278h] ; [ECX+13278h]=0↙
    ↳ xacce190
          call    _nrtnsvrs       ; tracing nested maximum level↙
    ↳  (1) reached, skipping this CALL
          add     esp, 0Ch

loc_2CE122B: ; CODE XREF: _kqvrow_+29E0118
          mov     dword ptr [ebx], 4
          jmp     loc_2CE1192

loc_2CE1236: ; DATA XREF: .rdata:0628B0AC
          lea     edx, [ebp+var_7C] ; [EBP-7Ch]=1
          push    edx             ; EDX=0xd76c8d8
          push    0
          mov     esi, [ebx+8]    ; [EBX+8]=0xce2ffb0, "TNS for ↙
    ↳ 32-bit Windows: Version 11.2.0.1.0 - Production"
```

```
          mov     [ebx+4], esi    ; ESI=0xce2ffb0, "TNS for 32-↙
    ↳ bit Windows: Version 11.2.0.1.0 - Production"
          mov     ecx, 50h
          mov     [ebp+var_18], ecx ; ECX=0x50
          push    ecx             ; ECX=0x50
          push    esi             ; ESI=0xce2ffb0, "TNS for 32-↙
    ↳ bit Windows: Version 11.2.0.1.0 - Production"
          call    _lxvers         ; tracing nested maximum level↙
    ↳  (1) reached, skipping this CALL
          add     esp, 10h
          mov     edx, [ebp+var_18] ; [EBP-18h]=0x50
          mov     dword ptr [ebx], 5
          test    edx, edx        ; EDX=0x50
          jnz     loc_2CE1192
          mov     edx, [ebp+var_14]
          mov     esi, [ebp+var_C]
          mov     eax, ebx
          mov     ebx, [ebp+var_8]
          mov     ecx, 5
          jmp     loc_2CE10F6

loc_2CE127A: ; DATA XREF: .rdata:0628B0B0
          mov     edx, [ebp+var_14] ; [EBP-14h]=0xc98c938
          mov     esi, [ebp+var_C] ; [EBP-0Ch]=0xcdfe248
          mov     edi, [ebp+var_4] ; [EBP-4]=0xc98c938
          mov     eax, ebx        ; EBX=0xcdfe554
          mov     ebx, [ebp+var_8] ; [EBP-8]=0

loc_2CE1288: ; CODE XREF: _kqvrow_+1F
          mov     eax, [eax+8]    ; [EAX+8]=0xce2ffb0, "NLSRTL ↙
    ↳ Version 11.2.0.1.0 - Production"
          test    eax, eax        ; EAX=0xce2ffb0, "NLSRTL ↙
    ↳ Version 11.2.0.1.0 - Production"
          jz      short loc_2CE12A7
          push    offset aXKqvvsnBuffer ; "x$kqvvsn buffer"
          push    eax             ; EAX=0xce2ffb0, "NLSRTL ↙
    ↳ Version 11.2.0.1.0 - Production"
          mov     eax, [ebp+arg_C] ; [EBP+14h]=0x8a172b4
          push    eax             ; EAX=0x8a172b4
          push    dword ptr [edx+10494h] ; [EDX+10494h]=0↙
    ↳ xc98cd58
          call    _kghfrf         ; tracing nested maximum level↙
    ↳  (1) reached, skipping this CALL
```

```
          add     esp, 10h

loc_2CE12A7: ; CODE XREF: _kqvrow_+1C1
          xor     eax, eax
          mov     esp, ebp
          pop     ebp
          retn                    ; EAX=0
_kqvrow_  endp
```

Now it is easy to see that row number is passed from outside of function. The function returns the string constructing it as follows:

String 1	Using `vsnstr`, `vsnnum`, `vsnban` global variables. Calling `sprintf()`.
String 2	Calling `kkxvsn()`.
String 3	Calling `lmxver()`.
String 4	Calling `npinli()`, `nrtnsvrs()`.
String 5	Calling `lxvers()`.

That's how corresponding functions are called for determining each module's version.

58.2 X$KSMLRU table in Oracle RDBMS

There is a mention of a special table in the *Diagnosing and Resolving Error ORA-04031 on the Shared Pool or Other Memory Pools [Video] [ID 146599.1]* note:

> There is a fixed table called X$KSMLRU that tracks allocations in the shared pool that cause other objects in the shared pool to be aged out. This fixed table can be used to identify what is causing the large allocation.
>
> If many objects are being periodically flushed from the shared pool then this will cause response time problems and will likely cause library cache latch contention problems when the objects are reloaded into the shared pool.
>
> One unusual thing about the X$KSMLRU fixed table is that the contents of the fixed table are erased whenever someone selects from the fixed table. This is done since the fixed table stores only the largest allocations that have occurred. The values are reset after being selected so that subsequent large allocations can be noted even if they were not quite as large as others that occurred previously. Because of this resetting, the output of selecting from this table should

be carefully kept since it cannot be retrieved back after the query is issued.

However, as it can be easily checked, this table's contents is cleared each time table querying. Are we able to find why? Let's back to tables we already know: `kqftab` and `kqftap` which were generated with oracle tables[4] help, containing all information about X$-tables, now we can see here, the `ksmlrs()` function is called to prepare this table's elements:

Listing 58.10: Result of oracle tables

```
kqftab_element.name: [X$KSMLRU] ?: [ksmlr] 0x4 0x64 0x11 0xc 0↙
    ↳ xffffc0bb 0x5
kqftap_param.name=[ADDR] ?: 0x917 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[INDX] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[INST_ID] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[KSMLRIDX] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[KSMLRDUR] ?: 0xb02 0x0 0x0 0x0 0x4 0x4 0x0
kqftap_param.name=[KSMLRSHRPOOL] ?: 0xb02 0x0 0x0 0x0 0x4 0x8 0↙
    ↳ x0
kqftap_param.name=[KSMLRCOM] ?: 0x501 0x0 0x0 0x0 0x14 0xc 0x0
kqftap_param.name=[KSMLRSIZ] ?: 0x2 0x0 0x0 0x0 0x4 0x20 0x0
kqftap_param.name=[KSMLRNUM] ?: 0x2 0x0 0x0 0x0 0x4 0x24 0x0
kqftap_param.name=[KSMLRHON] ?: 0x501 0x0 0x0 0x0 0x20 0x28 0x0
kqftap_param.name=[KSMLROHV] ?: 0xb02 0x0 0x0 0x0 0x4 0x48 0x0
kqftap_param.name=[KSMLRSES] ?: 0x17 0x0 0x0 0x0 0x4 0x4c 0x0
kqftap_param.name=[KSMLRADU] ?: 0x2 0x0 0x0 0x0 0x4 0x50 0x0
kqftap_param.name=[KSMLRNID] ?: 0x2 0x0 0x0 0x0 0x4 0x54 0x0
kqftap_param.name=[KSMLRNSD] ?: 0x2 0x0 0x0 0x0 0x4 0x58 0x0
kqftap_param.name=[KSMLRNCD] ?: 0x2 0x0 0x0 0x0 0x4 0x5c 0x0
kqftap_param.name=[KSMLRNED] ?: 0x2 0x0 0x0 0x0 0x4 0x60 0x0
kqftap_element.fn1=ksmlrs
kqftap_element.fn2=NULL
```

Indeed, with the tracer help it is easy to see this function is called each time we query the `X$KSMLRU` table.

Here we see a references to the `ksmsplu_sp()` and `ksmsplu_jp()` functions, each of them call the `ksmsplu()` finally. At the end of the `ksmsplu()` function we see a call to the `memset()`:

Listing 58.11: ksm.o

[4] http://yurichev.com/oracle_tables.html

```
...

.text:00434C50 loc_434C50:                             ; DATA ↙
    ↳ XREF: .rdata:off_5E50EA8
.text:00434C50                 mov     edx, [ebp-4]
.text:00434C53                 mov     [eax], esi
.text:00434C55                 mov     esi, [edi]
.text:00434C57                 mov     [eax+4], esi
.text:00434C5A                 mov     [edi], eax
.text:00434C5C                 add     edx, 1
.text:00434C5F                 mov     [ebp-4], edx
.text:00434C62                 jnz     loc_434B7D
.text:00434C68                 mov     ecx, [ebp+14h]
.text:00434C6B                 mov     ebx, [ebp-10h]
.text:00434C6E                 mov     esi, [ebp-0Ch]
.text:00434C71                 mov     edi, [ebp-8]
.text:00434C74                 lea     eax, [ecx+8Ch]
.text:00434C7A                 push    370h            ; Size
.text:00434C7F                 push    0               ; Val
.text:00434C81                 push    eax             ; Dst
.text:00434C82                 call    __intel_fast_memset
.text:00434C87                 add     esp, 0Ch
.text:00434C8A                 mov     esp, ebp
.text:00434C8C                 pop     ebp
.text:00434C8D                 retn
.text:00434C8D _ksmsplu        endp
```

Constructions like `memset (block, 0, size)` are often used just to zero memory block. What if we would take a risk, block `memset()` call and see what will happen?

Let's run tracer with the following options: set breakpoint at `0x434C7A` (the point where `memset()` arguments are to be passed), thus, that tracer set program counter `EIP` at this point to the point where passed to the `memset()` arguments are to be cleared (at `0x434C8A`) It can be said, we just simulate an unconditional jump from the address `0x434C7A` to `0x434C8A`.

```
tracer -a:oracle.exe bpx=oracle.exe!0x00434C7A,set(eip,0↙
    ↳ x00434C8A)
```

(Important: all these addresses are valid only for win32-version of Oracle RDBMS 11.2)

Indeed, now we can query `X$KSMLRU` table as many times as we want and it is not clearing anymore!

~~Do not try this at home ("MythBusters")~~ Do not try this on your production servers.

It is probably not a very useful or desired system behaviour, but as an experiment of locating piece of code we need, that is perfectly suit our needs!

58.3 V$TIMER table in Oracle RDBMS

V$TIMER is another *fixed view*, reflecting a rapidly changing value:

> V$TIMER displays the elapsed time in hundredths of a second. Time is measured since the beginning of the epoch, which is operating system specific, and wraps around to 0 again whenever the value overflows four bytes (roughly 497 days).

(From Oracle RDBMS documentation [5])

It is interesting the periods are different for Oracle for win32 and for Linux. Will we able to find a function generating this value?

As we can see, this information is finally taken from X$KSUTM table.

```
SQL> select * from V$FIXED_VIEW_DEFINITION where view_name='↙
    ↳ V$TIMER';

VIEW_NAME
------------------------------
VIEW_DEFINITION
--------------------------------------------------------------------------------
    ↳

V$TIMER
select  HSECS from GV$TIMER where inst_id = USERENV('Instance')

SQL> select * from V$FIXED_VIEW_DEFINITION where view_name='↙
    ↳ GV$TIMER';

VIEW_NAME
------------------------------
VIEW_DEFINITION
--------------------------------------------------------------------------------
    ↳

GV$TIMER
```

[5] http://docs.oracle.com/cd/B28359_01/server.111/b28320/dynviews_3104.htm

```
select inst_id,ksutmtim from x$ksutm
```

Now we stuck in a small problem, there are no references to value generating function(s) in the tables kqftab/kqftap:

Listing 58.12: Result of oracle tables

```
kqftab_element.name: [X$KSUTM] ?: [ksutm] 0x1 0x4 0x4 0x0 0↙
    ↳ xffffc09b 0x3
kqftap_param.name=[ADDR] ?: 0x10917 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[INDX] ?: 0x20b02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[INST_ID] ?: 0xb02 0x0 0x0 0x0 0x4 0x0 0x0
kqftap_param.name=[KSUTMTIM] ?: 0x1302 0x0 0x0 0x0 0x4 0x0 0x1e
kqftap_element.fn1=NULL
kqftap_element.fn2=NULL
```

Let's try to find a string KSUTMTIM, and we find it in this function:

```
kqfd_DRN_ksutm_c proc near              ; DATA XREF: .rodata↙
    ↳ :0805B4E8

arg_0           = dword ptr  8
arg_8           = dword ptr  10h
arg_C           = dword ptr  14h

                push    ebp
                mov     ebp, esp
                push    [ebp+arg_C]
                push    offset ksugtm
                push    offset _2__STRING_1263_0 ; "KSUTMTIM"
                push    [ebp+arg_8]
                push    [ebp+arg_0]
                call    kqfd_cfui_drain
                add     esp, 14h
                mov     esp, ebp
                pop     ebp
                retn
kqfd_DRN_ksutm_c endp
```

The function kqfd_DRN_ksutm_c() is mentioned in kqfd_tab_registry_0 table:

```
dd offset _2__STRING_62_0 ; "X$KSUTM"
dd offset kqfd_OPN_ksutm_c
dd offset kqfd_tabl_fetch
dd 0
```

```
dd 0
dd offset kqfd_DRN_ksutm_c
```

There are is a function `ksugtm()` referenced here. Let's see what's in it (Linux x86):

Listing 58.13: ksu.o

```
ksugtm          proc near

var_1C          = byte ptr -1Ch
arg_4           = dword ptr  0Ch

                push    ebp
                mov     ebp, esp
                sub     esp, 1Ch
                lea     eax, [ebp+var_1C]
                push    eax
                call    slgcs
                pop     ecx
                mov     edx, [ebp+arg_4]
                mov     [edx], eax
                mov     eax, 4
                mov     esp, ebp
                pop     ebp
                retn
ksugtm          endp
```

Almost the same code in win32-version.

Is this the function we are looking for? Let's see:

```
tracer -a:oracle.exe bpf=oracle.exe!_ksugtm,args:2,dump_args:0x4
```

Let's try again:

```
SQL> select * from V$TIMER;

     HSECS
----------
  27294929

SQL> select * from V$TIMER;

     HSECS
----------
  27295006
```

```
SQL> select * from V$TIMER;

     HSECS
----------
  27295167
```

Listing 58.14: tracer output

```
TID=2428|(0) oracle.exe!_ksugtm (0x0, 0xd76c5f0) (called from ↙
    ↳ oracle.exe!__VInfreq__qerfxFetch+0xfad (0x56bb6d5))
Argument 2/2
0D76C5F0: 38 C9                                           "8.    ↙
    ↳             "
TID=2428|(0) oracle.exe!_ksugtm () -> 0x4 (0x4)
Argument 2/2 difference
00000000: D1 7C A0 01                                     ".|..  ↙
    ↳             "
TID=2428|(0) oracle.exe!_ksugtm (0x0, 0xd76c5f0) (called from ↙
    ↳ oracle.exe!__VInfreq__qerfxFetch+0xfad (0x56bb6d5))
Argument 2/2
0D76C5F0: 38 C9                                           "8.    ↙
    ↳             "
TID=2428|(0) oracle.exe!_ksugtm () -> 0x4 (0x4)
Argument 2/2 difference
00000000: 1E 7D A0 01                                     ".}..  ↙
    ↳             "
TID=2428|(0) oracle.exe!_ksugtm (0x0, 0xd76c5f0) (called from ↙
    ↳ oracle.exe!__VInfreq__qerfxFetch+0xfad (0x56bb6d5))
Argument 2/2
0D76C5F0: 38 C9                                           "8.    ↙
    ↳             "
TID=2428|(0) oracle.exe!_ksugtm () -> 0x4 (0x4)
Argument 2/2 difference
00000000: BF 7D A0 01                                     ".}..  ↙
    ↳             "
```

Indeed —the value is the same we see in SQL*Plus and it is returning via second argument.

Let's see what is in `slgcs()` (Linux x86):

```
slgcs           proc near

var_4           = dword ptr -4
arg_0           = dword ptr  8
```

```
                push    ebp
                mov     ebp, esp
                push    esi
                mov     [ebp+var_4], ebx
                mov     eax, [ebp+arg_0]
                call    $+5
                pop     ebx
                nop                     ; PIC mode
                mov     ebx, offset _GLOBAL_OFFSET_TABLE_
                mov     dword ptr [eax], 0
                call    sltrgatime64    ; PIC mode
                push    0
                push    0Ah
                push    edx
                push    eax
                call    __udivdi3       ; PIC mode
                mov     ebx, [ebp+var_4]
                add     esp, 10h
                mov     esp, ebp
                pop     ebp
                retn
slgcs           endp
```

(it is just a call to `sltrgatime64()` and division of its result by 10 (14))
And win32-version:

```
_slgcs          proc near               ; CODE XREF: ↙
    ↳ _dbgefgHtElResetCount+15
                                        ; _dbgerRunActions+1528
                db      66h
                nop
                push    ebp
                mov     ebp, esp
                mov     eax, [ebp+8]
                mov     dword ptr [eax], 0
                call    ds:__imp__GetTickCount@0 ; GetTickCount↙
    ↳ ()
                mov     edx, eax
                mov     eax, 0CCCCCCCDh
                mul     edx
                shr     edx, 3
                mov     eax, edx
                mov     esp, ebp
                pop     ebp
```

```
                retn
_slgcs          endp
```

It is just result of `GetTickCount()` [6] divided by 10 (14).

Voilà! That's why win32-version and Linux x86 version show different results, just because they are generated by different OS functions.

Drain apparently means *connecting* specific table column to specific function.

I added the table `kqfd_tab_registry_0` to oracle tables[7], now we can see, how table column's variables are *connected* to specific functions:

```
[X$KSUTM] [kqfd_OPN_ksutm_c] [kqfd_tabl_fetch] [NULL] [NULL] [↙
    ↳ kqfd_DRN_ksutm_c]
[X$KSUSGIF] [kqfd_OPN_ksusg_c] [kqfd_tabl_fetch] [NULL] [NULL] [↙
    ↳ kqfd_DRN_ksusg_c]
```

OPN, apparently, *open*, and *DRN*, apparently, meaning *drain*.

[6] http://msdn.microsoft.com/en-us/library/windows/desktop/ms724408(v=vs.85).aspx
[7] http://yurichev.com/oracle_tables.html

Chapter 59

Handwritten assembly code

59.1 EICAR test file

This .COM-file is intended for antivirus testing, it is possible to run in in MS-DOS and it will print string: "EICAR-STANDARD-ANTIVIRUS-TEST-FILE!" [1].

Its important property is that it's entirely consisting of printable ASCII-symbols, which, in turn, makes possible to create it in any text editor:

```
X5O!P%@AP[4\PZX54(P^)7CC)7}$EICAR-STANDARD-ANTIVIRUS-TEST-FILE!↙
    ↳ $H+H*
```

Let's decompile it:

```
; initial conditions: SP=0FFFEh, SS:[SP]=0
0100 58            pop     ax
; AX=0, SP=0
0101 35 4F 21      xor     ax, 214Fh
; AX = 214Fh and SP = 0
0104 50            push    ax
; AX = 214Fh, SP = FFFEh and SS:[FFFE] = 214Fh
0105 25 40 41      and     ax, 4140h
; AX = 140h, SP = FFFEh and SS:[FFFE] = 214Fh
0108 50            push    ax
; AX = 140h, SP = FFFCh, SS:[FFFC] = 140h and SS:[FFFE] = 214Fh
0109 5B            pop     bx
; AX = 140h, BX = 140h, SP = FFFEh and SS:[FFFE] = 214Fh
010A 34 5C         xor     al, 5Ch
; AX = 11Ch, BX = 140h, SP = FFFEh and SS:[FFFE] = 214Fh
010C 50            push    ax
```

[1] {https://en.wikipedia.org/wiki/EICAR_test_file}

```
010D 5A            pop     dx
; AX = 11Ch, BX = 140h, DX = 11Ch, SP = FFFEh and SS:[FFFE] = ↙
    ↳ 214Fh
010E 58            pop     ax
; AX = 214Fh, BX = 140h, DX = 11Ch and SP = 0
010F 35 34 28      xor     ax, 2834h
; AX = 97Bh, BX = 140h, DX = 11Ch and SP = 0
0112 50            push    ax
0113 5E            pop     si
; AX = 97Bh, BX = 140h, DX = 11Ch, SI = 97Bh and SP = 0
0114 29 37         sub     [bx], si
0116 43            inc     bx
0117 43            inc     bx
0118 29 37         sub     [bx], si
011A 7D 24         jge     short near ptr word_10140
011C 45 49 43 ...  db 'EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$'
0140 48 2B   word_10140  dw 2B48h  ; CD 21 (INT 21) will be here
0142 48 2A               dw 2A48h  ; CD 20 (INT 20) will be here
0144 0D                  db  0Dh
0145 0A                  db  0Ah
```

I added comments about registers and stack after each instruction.

Essentially, all these instructions are here only to execute this code:

```
B4 09     MOV AH, 9
BA 1C 01  MOV DX, 11Ch
CD 21     INT 21h
CD 20     INT 20h
```

`INT 21h` with 9th function (passed in `AH`) just prints a string, address of which is passed in `DS:DX`. By the way, the string should be terminated with '$' sign. Apparently, it's inherited from CP/M and this function was leaved in DOS for compatibility. `INT 20h` exits to DOS.

But as we can see, these instruction's opcodes are not strictly printable. So the main part of EICAR-file is:

- preparing register (AH and DX) values we need;
- preparing INT 21 and INT 20 opcodes in memory;
- executing INT 21 and INT 20.

By the way, this technique is widely used in shellcode constructing, when one need to pass x86-code in the string form.

Here is also a list of all x86 instructions which has printable opcodes: B.6.6.

Chapter 60

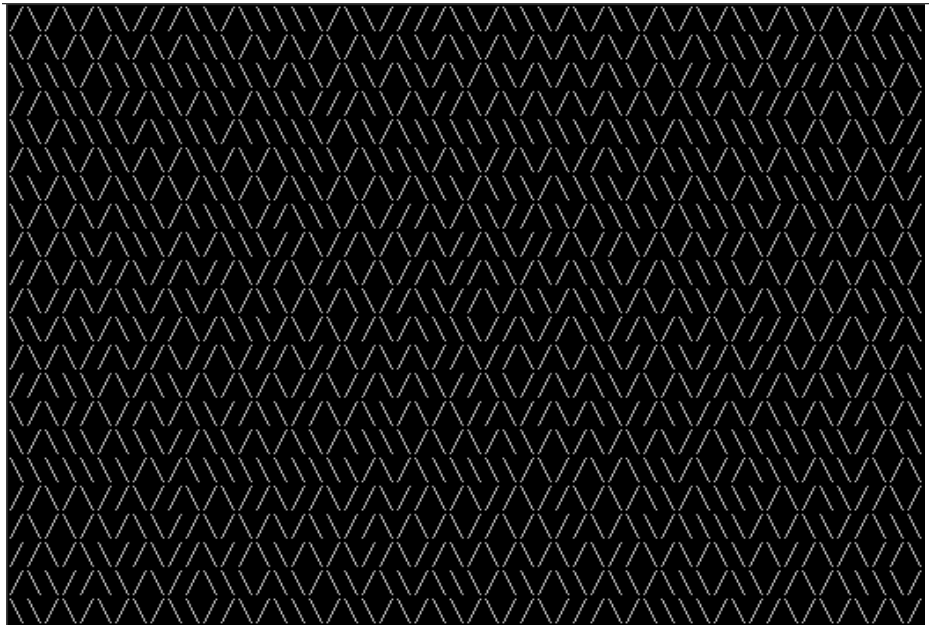
Demos

Demos (or demomaking) was an excellent exercise in mathematics, computer graphics programming and very tight x86 hand coding.

60.1 10 PRINT CHR$(205.5+RND(1)); : GOTO 10

All examples here are MS-DOS .COM files.

In [9] we can read about one of the most simplest possible random maze generators. It just prints slash or backslash character randomly and endlessly, resulting something like:



There are some known implementations for 16-bit x86.

60.1.1 Trixter's 42 byte version

The listing taken from his website[1], but comments are mine.

```
00000000: B001          mov         al,1        ; set 40x25 ↙
    ↳ videomode
00000002: CD10          int         010
00000004: 30FF          xor         bh,bh       ; set videopage ↙
    ↳ for int 10h call
00000006: B9D007        mov         cx,007D0    ; 2000 characters↙
    ↳  to output
00000009: 31C0          xor         ax,ax
0000000B: 9C            pushf                   ; push flags
; get random value from timer chip
0000000C: FA            cli                     ; disable ↙
    ↳ interrupts
0000000D: E643          out         043,al      ; write 0 to port↙
    ↳  43h
; read 16-bit value from port 40h
0000000F: E440          in          al,040
```

[1] http://trixter.oldskool.org/2012/12/17/maze-generation-in-thirteen-bytes/

```
00000011: 88C4          mov         ah,al
00000013: E440          in          al,040
00000015: 9D            popf                    ; enable ↙
    ↳ interrupts by restoring IF flag
00000016: 86C4          xchg        ah,al
; here we have 16-bit pseudorandom value
00000018: D1E8          shr         ax,1
0000001A: D1E8          shr         ax,1
; CF currently have second bit from the value
0000001C: B05C          mov         al,05C ;'\'
; if CF=1, skip the next instruction
0000001E: 7202          jc          000000022
; if CF=0, reload AL register with another character
00000020: B02F          mov         al,02F ;'/'
; output character
00000022: B40E          mov         ah,00E
00000024: CD10          int         010
00000026: E2E1          loop        000000009 ; loop 2000 times
00000028: CD20          int         020       ; exit to DOS
```

Pseudo-random value here is in fact the time passed from the system boot, taken from 8253 time chip, the value increases by one 18.2 times per second.

By writing zero to port 43h, we mean the command is "select counter 0", "counter latch", "binary counter" (not BCD[2] value).

Interrupts enabled back with POPF instruction, which restores IF flag as well.

It is not possible to use IN instruction with other registers instead of AL, hence that shuffling.

60.1.2 My attempt to reduce Trixter's version: 27 bytes

We can say that since we use timer not to get precise time value, but pseudo-random one, so we may not spent time (and code) to disable interrupts. Another thing we might say that we need only bit from a low 8-bit part, so let's read only it.

I reduced the code slightly and I've got 27 bytes:

```
00000000: B9D007    mov     cx,007D0 ; limit output to 2000 ↙
    ↳ characters
00000003: 31C0      xor     ax,ax    ; command to timer chip
00000005: E643      out     043,al
00000007: E440      in      al,040   ; read 8-bit of timer
00000009: D1E8      shr     ax,1     ; get second bit to CF flag
0000000B: D1E8      shr     ax,1
0000000D: B05C      mov     al,05C   ; prepare '\'
```

[2] Binary-coded decimal

```
0000000F: 7202      jc      000000013
00000011: B02F      mov     al,02F    ; prepare '/'
; output character to screen
00000013: B40E      mov     ah,00E
00000015: CD10      int     010
00000017: E2EA      loop    000000003
; exit to DOS
00000019: CD20      int     020
```

60.1.3 Take a random memory garbage as a source of randomness

Since it is MS-DOS, there are no memory protection at all, we can read from whatever address. Even more than that: simple `LODSB` instruction will read byte from `DS:SI` address, but it's not a problem if register values are not setted up, let it read 1) random bytes; 2) from random memory place!

So it is suggested in Trixter webpage[3] to use `LODSB` without any setup.

It is also suggested that `SCASB` instruction can be used instead, because it sets flag according to the byte it read.

Another idea to minimize code is to use `INT 29h` DOS syscall, which just prints character stored in `AL` register.

That is what Peter Ferrie and Andrey "herm1t" Baranovich did (11 and 10 bytes) [4]:

Listing 60.1: Andrey "herm1t" Baranovich: 11 bytes

```
00000000: B05C       mov         al,05C    ;'\'
; read AL byte from random place of memory
00000002: AE         scasb
; PF = parity(AL - random_memory_byte) = parity(5Ch - ↙
    ↳ random_memory_byte)
00000003: 7A02       jp          000000007
00000005: B02F       mov         al,02F    ;'/'
00000007: CD29       int         029       ; output AL to screen
00000009: EBF5       jmp         000000000 ; loop endlessly
```

`SCASB` also use value in `AL` register, it subtract random memory byte value from `5Ch` value in `AL`. `JP` is rare instruction, here it used for checking parity flag (PF), which is generated by the formulae in the listing. As a consequence, the output character is determined not by some bit in random memory byte, but by sum of bits, this (hopefully) makes result more distributed.

It is possible to make this even shorter by using undocumented x86 instruction `SALC` (AKA `SETALC`) ("Set AL CF"). It was introduced in NEC V20 CPU and sets `AL` to `0xFF` if `CF` is 1 or to 0 if otherwise. So this code will not run on 8086/8088.

[3] http://trixter.oldskool.org/2012/12/17/maze-generation-in-thirteen-bytes/

[4] http://pferrie.host22.com/misc/10print.htm

Listing 60.2: Peter Ferrie: 10 bytes

```
; AL is random at this point
00000000: AE          scasb
; CF is set accoring subtracting random memory byte from AL.
; so it is somewhat random at this point
00000001: D6          setalc
; AL is set to 0xFF if CF=1 or to 0 if otherwise
00000002: 242D        and         al,02D ;'-'
; AL here is 0x2D or 0
00000004: 042F        add         al,02F ;'/'
; AL here is 0x5C or 0x2F
00000006: CD29        int         029       ; output AL to screen
00000008: EBF6        jmps        000000000 ; loop endlessly
```

So it is possible to get rid of conditional jumps at all. The ASCII[5] code of backslash ("\") is 0x5C and 0x2F for slash ("/"). So we need to convert one (pseudo-random) bit in CF flag to 0x5C or 0x2F value.

This is done easily: by AND-ing all bits in AL (where all 8 bits are set or cleared) with 0x2D we have just 0 or 0x2D. By adding 0x2F to this value, we get 0x5C or 0x2F. Then just ouptut it to screen.

60.1.4 Conclusion

It is also worth adding that result may be different in DOSBox, Windows NT and even MS-DOS, due to different conditions: timer chip may be emulated differently, initial register contents may be different as well.

[5] American Standard Code for Information Interchange

Part VII

Other things

Chapter 61

npad

It is an assembly language macro for label aligning by a specific border.

That's often need for the busy labels to where control flow is often passed, e.g., loop body begin. So the CPU will effectively load data or code from the memory, through memory bus, cache lines, etc.

Taken from `listing.inc` (MSVC):

By the way, it is curious example of different `NOP` variations. All these instructions has no effects whatsoever, but has different size.

```
;; LISTING.INC
;;
;; This file contains assembler macros and is included by the ↙
    ↳ files created
;; with the -FA compiler switch to be assembled by MASM (↙
    ↳ Microsoft Macro
;; Assembler).
;;
;; Copyright (c) 1993-2003, Microsoft Corporation. All rights ↙
    ↳ reserved.

;; non destructive nops
npad macro size
if size eq 1
  nop
else
 if size eq 2
   mov edi, edi
 else
  if size eq 3
    ; lea ecx, [ecx+00]
```

```
  DB 8DH, 49H, 00H
 else
  if size eq 4
    ; lea esp, [esp+00]
    DB 8DH, 64H, 24H, 00H
  else
   if size eq 5
     add eax, DWORD PTR 0
   else
    if size eq 6
      ; lea ebx, [ebx+00000000]
      DB 8DH, 9BH, 00H, 00H, 00H, 00H
    else
     if size eq 7
       ; lea esp, [esp+00000000]
       DB 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H
     else
      if size eq 8
       ; jmp .+8; .npad 6
       DB 0EBH, 06H, 8DH, 9BH, 00H, 00H, 00H, 00H
      else
       if size eq 9
        ; jmp .+9; .npad 7
        DB 0EBH, 07H, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H
       else
        if size eq 10
         ; jmp .+A; .npad 7; .npad 1
         DB 0EBH, 08H, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 90H
        else
         if size eq 11
          ; jmp .+B; .npad 7; .npad 2
          DB 0EBH, 09H, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 8BH↙
↳ , 0FFH
        else
         if size eq 12
          ; jmp .+C; .npad 7; .npad 3
          DB 0EBH, 0AH, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 8↙
↳ DH, 49H, 00H
         else
          if size eq 13
           ; jmp .+D; .npad 7; .npad 4
           DB 0EBH, 0BH, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 8↙
↳ DH, 64H, 24H, 00H
```

```
            else
             if size eq 14
              ; jmp .+E; .npad 7; .npad 5
              DB 0EBH, 0CH, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, ↙
    ↳ 05H, 00H, 00H, 00H, 00H
             else
              if size eq 15
               ; jmp .+F; .npad 7; .npad 6
               DB 0EBH, 0DH, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H,↙
    ↳  8DH, 9BH, 00H, 00H, 00H, 00H
              else
               %out error: unsupported npad size
               .err
              endif
             endif
            endif
           endif
          endif
         endif
        endif
       endif
      endif
     endif
    endif
   endif
  endif
 endif
endif
endm
```

Chapter 62

Executable files patching

62.1 Text strings

C strings are most easily patched (unless they are encrypted) in any hex editor. This technique available even for those who are not aware of machine code and executabe file formats. New string should not be bigger than old, because it's a risk to overwrite some other value or code there. Using this method, a lot of software was *localized* in MS-DOS era, at least in ex-USSR countries in 80's and 90's. It was a reason why so weird abbreviations was present in *localized* software: it was no room for longer strings.

As of Delphi strings, a string size should also be corrected, if needed.

62.2 x86 code

Frequent patching tasks are:

- One of the most frequently job is to disable some instruction. It is often done by filling it by byte `0x90` (NOP).
- Conditional jumps, which have opcode like `74 xx` (`JZ`), may also be filled by two NOPs. It is also possible to disable conditional jump by writing 0 at the second byte (*jump offset*).
- Another frequent job is to make conditional jump to trigger always: this can be done by writing `0xEB` instead of opcode, it mean `JMP`.
- A function execution can be disabled by writing `RETN` (0xC3) at its beginning. This is true for all functions excluding `stdcall` (44.2). While patching `stdcall` functions, one should determine number of arguments (for example, by finding `RETN` in this function), and use `RETN` with 16-bit argument (0xC2).

- Sometimes, a disabled functions should return 0 or 1. This can be done by `MOV EAX, 0` or `MOV EAX, 1`, but it's slightly verbose. Better way is `XOR EAX, EAX` (2 bytes) or `XOR EAX, EAX / INC EAX` (3 bytes).

A software may be protected against modifications. This protection is often done by reading executable code and doing some checksumming. Therefore, the code should be read before protection will be triggered. This can be determined by setting breakpoint on reading memory.

tracer has BPM option for this.

PE executable file relocs (48.2.6) should not be touched while patching, because Windows loader will overwrite a new code. (They are grayed in Hiew, for example: fig. 6.12). As a last resort, it is possible to write jumps circumventing relocs, or one will need to edit relocs table.

Chapter 63

Compiler intrinsic

A function specific to a compiler which is not usual library function. Compiler generate a specific machine code instead of call to it. It is often a pseudofunction for specific CPU instruction.

For example, there are no cyclic shift operations in C/C++ languages, but present in most CPUs. For programmer's convenience, at least MSVC has pseudofunctions *_rotl()* and *_rotr()*[1] which are translated by compiler directly to the ROL/ROR x86 instructions.

Another example are functions enabling to generate SSE-instructions right in the code.

Full list of MSVC intrinsics: http://msdn.microsoft.com/en-us/library/26td21ds.aspx.

[1] http://msdn.microsoft.com/en-us/library/5cc576c4.aspx

Chapter 64

Compiler's anomalies

Intel C++ 10.1, which was used for Oracle RDBMS 11.2 Linux86 compilation, may emit two JZ in row, and there are no references to the second JZ. Second JZ is thus senseless.

Listing 64.1: kdli.o from libserver11.a

```
.text:08114CF1                    loc_8114CF1: ; CODE XREF: ↙
    ↳ __PGOSF539_kdlimemSer+89A
.text:08114CF1                                 ; ↙
    ↳ __PGOSF539_kdlimemSer+3994
.text:08114CF1 8B 45 08              mov     eax, [ebp+arg_0]
.text:08114CF4 0F B6 50 14           movzx   edx, byte ptr [eax↙
    ↳ +14h]
.text:08114CF8 F6 C2 01              test    dl, 1
.text:08114CFB 0F 85 17 08 00 00     jnz     loc_8115518
.text:08114D01 85 C9                 test    ecx, ecx
.text:08114D03 0F 84 8A 00 00 00     jz      loc_8114D93
.text:08114D09 0F 84 09 08 00 00     jz      loc_8115518
.text:08114D0F 8B 53 08              mov     edx, [ebx+8]
.text:08114D12 89 55 FC              mov     [ebp+var_4], edx
.text:08114D15 31 C0                 xor     eax, eax
.text:08114D17 89 45 F4              mov     [ebp+var_C], eax
.text:08114D1A 50                    push    eax
.text:08114D1B 52                    push    edx
.text:08114D1C E8 03 54 00 00        call    len2nbytes
.text:08114D21 83 C4 08              add     esp, 8
```

Listing 64.2: from the same code

```
.text:0811A2A5                  loc_811A2A5: ; CODE XREF: ↙
    ↳ kdliSerLengths+11C
.text:0811A2A5                                ; kdliSerLengths+1↙
    ↳ C1
.text:0811A2A5 8B 7D 08              mov     edi, [ebp+arg_0]
.text:0811A2A8 8B 7F 10              mov     edi, [edi+10h]
.text:0811A2AB 0F B6 57 14           movzx   edx, byte ptr [edi↙
    ↳ +14h]
.text:0811A2AF F6 C2 01              test    dl, 1
.text:0811A2B2 75 3E                 jnz     short loc_811A2F2
.text:0811A2B4 83 E0 01              and     eax, 1
.text:0811A2B7 74 1F                 jz      short loc_811A2D8
.text:0811A2B9 74 37                 jz      short loc_811A2F2
.text:0811A2BB 6A 00                 push    0
.text:0811A2BD FF 71 08              push    dword ptr [ecx+8]
.text:0811A2C0 E8 5F FE FF FF        call    len2nbytes
```

It is probably code generator bug was not found by tests, because, resulting code is working correctly anyway.

Another compiler anomaly I described here (17.2.4).

I demonstrate such cases here, so to understand that such compilers errors are possible and sometimes one should not to rack one's brain and think why compiler generated such strange code.

Chapter 65

OpenMP

OpenMP is one of the simplest ways to parallelize simple algorithm.

As an example, let's try to build a program to compute cryptographic *nonce*. In my simplistic example, *nonce* is a number added to the plain unencrypted text in order to produce hash with some specific feature. For example, at some step, Bitcoin protocol require to find a such *nonce* so resulting hash will contain specific number of running zeroes. This is also called "proof of work" [1] (i.e., system prove it did some intensive calculations and spent some time for it).

My example is not related to Bitcoin, it will try to add a numbers to the "hello, world!_" string in order to find such number when "hello, world!_<number>" will contain at least 3 zero bytes after hashing this string by SHA512 algorithm.

Let's limit our brute-force to the interval in 0..INT32_MAX-1 (i.e., 0x7FFFFFFE or 2147483646).

The algorithm is pretty straightforward:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include "sha512.h"

int found=0;
int32_t checked=0;

int32_t* __min;
int32_t* __max;

time_t start;
```

[1] https://en.wikipedia.org/wiki/Proof-of-work_system

```
#ifdef __GNUC__
#define min(X,Y) ((X) < (Y) ? (X) : (Y))
#define max(X,Y) ((X) > (Y) ? (X) : (Y))
#endif

void check_nonce (int32_t nonce)
{
        uint8_t buf[32];
        struct sha512_ctx ctx;
        uint8_t res[64];

        // update statistics
        int t=omp_get_thread_num();

        if (__min[t]==-1)
                __min[t]=nonce;
        if (__max[t]==-1)
                __max[t]=nonce;

        __min[t]=min(__min[t], nonce);
        __max[t]=max(__max[t], nonce);

        // idle if valid nonce found
        if (found)
                return;

        memset (buf, 0, sizeof(buf));
        sprintf (buf, "hello, world!_%d", nonce);

        sha512_init_ctx (&ctx);
        sha512_process_bytes (buf, strlen(buf), &ctx);
        sha512_finish_ctx (&ctx, &res);
        if (res[0]==0 && res[1]==0 && res[2]==0)
        {
                printf ("found (thread %d): [%s]. seconds spent↙
    ↳ =%d\n", t, buf, time(NULL)-start);
                found=1;
        };
        #pragma omp atomic
        checked++;

        #pragma omp critical
        if ((checked % 100000)==0)
```

```
                printf ("checked=%d\n", checked);
};

int main()
{
        int32_t i;
        int threads=omp_get_max_threads();
        printf ("threads=%d\n", threads);

        __min=(int32_t*)malloc(threads*sizeof(int32_t));
        __max=(int32_t*)malloc(threads*sizeof(int32_t));
        for (i=0; i<threads; i++)
                __min[i]=__max[i]=-1;

        start=time(NULL);

        #pragma omp parallel for
        for (i=0; i<INT32_MAX; i++)
                check_nonce (i);

        for (i=0; i<threads; i++)
                printf ("__min[%d]=0x%08x __max[%d]=0x%08x\n", i↙
    ↳ , __min[i], i, __max[i]);

        free(__min); free(__max);
};
```

`check_nonce()` function is just add a number to the string, hashes it by SHA512 and checks for 3 zero bytes in the result.

Very important part of the code is:

```
        #pragma omp parallel for
        for (i=0; i<INT32_MAX; i++)
                check_nonce (i);
```

Yes, that simple, without `#pragma` we just call `check_nonce()` for each number from 0 to `INT32_MAX` (`0x7fffffff` or 2147483647). With `#pragma`, a compiler adds a special code which will slice the loop interval to smaller intervals, to run them by all CPU cores available [2].

The example may be compiled [3] in MSVC 2012:

[2] N.B.: I intentionally demonstrate here simplest possible example, but in practice, usage of OpenMP may be harder and more complex

[3] sha512.(c|h) and u64.h files can be taken from the OpenSSL library: http://www.openssl.org/source/

```
cl openmp_example.c sha512.obj /openmp /O1 /Zi /Faopenmp_example↙
    ↳ .asm
```

Or in GCC:

```
gcc -fopenmp 2.c sha512.c -S -masm=intel
```

65.1 MSVC

Now that's how MSVC 2012 generates main loop:

Listing 65.1: MSVC 2012

```
        push    OFFSET _main$omp$1
        push    0
        push    1
        call    __vcomp_fork
        add     esp, 16                                 ; ↙
    ↳ 00000010H
```

All functions prefixed by `vcomp` are OpenMP-related and stored in the vcomp*.dll file. So here is a group of threads are started.

Let's take a look on `_main$omp$1`:

Listing 65.2: MSVC 2012

```
$T1 = -8                                                ; size =↙
    ↳  4
$T2 = -4                                                ; size =↙
    ↳  4
_main$omp$1 PROC                                        ; COMDAT
        push    ebp
        mov     ebp, esp
        push    ecx
        push    ecx
        push    esi
        lea     eax, DWORD PTR $T2[ebp]
        push    eax
        lea     eax, DWORD PTR $T1[ebp]
        push    eax
        push    1
        push    1
        push    2147483646                              ; 7↙
    ↳ fffffffeH
```

```
        push    0
        call    __vcomp_for_static_simple_init
        mov     esi, DWORD PTR $T1[ebp]
        add     esp, 24                                 ; ↙
    ↳ 00000018H
        jmp     SHORT $LN6@main$omp$1
$LL2@main$omp$1:
        push    esi
        call    _check_nonce
        pop     ecx
        inc     esi
$LN6@main$omp$1:
        cmp     esi, DWORD PTR $T2[ebp]
        jle     SHORT $LL2@main$omp$1
        call    __vcomp_for_static_end
        pop     esi
        leave
        ret     0
_main$omp$1 ENDP
```

This function will be started n times in parallel, where n is number of CPU cores. `vcomp_for_static_simple_init()` is calculating interval for the for() construct for the current thread, depending on the current thread number. Loop begin and end values are stored in `$T1` and `$T2` local variables. You may also notice `7ffffffeh` (or 2147483646) as an argument to the `vcomp_for_static_simple_init()` function—this is a number of iterations of the whole loop to by divided evenly.

Then we see a new loop with a call to `check_nonce()` function which do all work.

I also added some code in the beginning of `check_nonce()` function to gather statistics, with which arguments the function was called.

This is what we see while run it:

```
threads=4
...
checked=2800000
checked=3000000
checked=3200000
checked=3300000
found (thread 3): [hello, world!_1611446522]. seconds spent=3
__min[0]=0x00000000 __max[0]=0x1fffffff
__min[1]=0x20000000 __max[1]=0x3fffffff
__min[2]=0x40000000 __max[2]=0x5fffffff
__min[3]=0x60000000 __max[3]=0x7ffffffe
```

Yes, result is correct, first 3 bytes are zeroes:

```
C:\...\sha512sum test
000000↙
    ↳ f4a8fac5a4ed38794da4c1e39f54279ad5d9bb3c5465cdf57adaf60403df6e3fe60
    ↳
0fee50eb38dd4c0279cb114672e2 *test
```

Running time is $\approx 2..3$ seconds on my 4-core Intel Xeon E3-1220 3.10 GHz. In the task manager I see 5 threads: 1 main thread + 4 more started. I did not any further optimizations to keep my example as small and clear as possible. But probably it can be done much faster. My CPU has 4 cores, that is why OpenMP started exactly 4 threads.

By looking at the statistics table we can clearly see how loop was finely sliced by 4 even parts. Oh well, almost even, if not to consider the last bit.

There are also pragmas for atomic operations.

Let's see how this code is compiled:

```
        #pragma omp atomic
        checked++;

        #pragma omp critical
        if ((checked % 100000)==0)
                printf ("checked=%d\n", checked);
```

Listing 65.3: MSVC 2012

```
        push    edi
        push    OFFSET _checked
        call    __vcomp_atomic_add_i4
; Line 55
        push    OFFSET _$vcomp$critsect$
        call    __vcomp_enter_critsect
        add     esp, 12                                 ; ↙
    ↳ 0000000cH
; Line 56
        mov     ecx, DWORD PTR _checked
        mov     eax, ecx
        cdq
        mov     esi, 100000                             ; 000186↙
    ↳ a0H
        idiv    esi
        test    edx, edx
        jne     SHORT $LN1@check_nonc
; Line 57
        push    ecx
        push    OFFSET ??_C@_0M@NPNHLIOO@checked?$DN?$CFd?6?$AA@
```

```
        call    _printf
        pop     ecx
        pop     ecx
$LN1@check_nonc:
        push    DWORD PTR _$vcomp$critsect$
        call    __vcomp_leave_critsect
        pop     ecx
```

As it turns out, `vcomp_atomic_add_i4()` function in the vcomp*.dll is just a a tiny function having `LOCK XADD` instruction[4].

`vcomp_enter_critsect()` eventually calling win32 API function `EnterCriticalSectio`[5].

65.2 GCC

GCC 4.8.1 produces the program which shows exactly the same statistics table, so, GCC implementation divides the loop by parts in the same fashion.

Listing 65.4: GCC 4.8.1

```
        mov     edi, OFFSET FLAT:main._omp_fn.0
        call    GOMP_parallel_start
        mov     edi, 0
        call    main._omp_fn.0
        call    GOMP_parallel_end
```

Unlike MSVC implementation, what GCC code is doing is starting 3 threads, but also runs fourth in the current thread. So there will be 4 threads instead of 5 as in MSVC.

Here is a `main._omp_fn.0` function:

Listing 65.5: GCC 4.8.1

```
main._omp_fn.0:
        push    rbp
        mov     rbp, rsp
        push    rbx
        sub     rsp, 40
        mov     QWORD PTR [rbp-40], rdi
        call    omp_get_num_threads
        mov     ebx, eax
        call    omp_get_thread_num
        mov     esi, eax
```

[4] Read more about LOCK prefix: B.6.1

[5] Read more about critical sections here: 48.4

```
        mov     eax, 2147483647 ; 0x7FFFFFFF
        cdq
        idiv    ebx
        mov     ecx, eax
        mov     eax, 2147483647 ; 0x7FFFFFFF
        cdq
        idiv    ebx
        mov     eax, edx
        cmp     esi, eax
        jl      .L15
.L18:
        imul    esi, ecx
        mov     edx, esi
        add     eax, edx
        lea     ebx, [rax+rcx]
        cmp     eax, ebx
        jge     .L14
        mov     DWORD PTR [rbp-20], eax
.L17:
        mov     eax, DWORD PTR [rbp-20]
        mov     edi, eax
        call    check_nonce
        add     DWORD PTR [rbp-20], 1
        cmp     DWORD PTR [rbp-20], ebx
        jl      .L17
        jmp     .L14
.L15:
        mov     eax, 0
        add     ecx, 1
        jmp     .L18
.L14:
        add     rsp, 40
        pop     rbx
        pop     rbp
        ret
```

Here we see that division clearly: by calling to `omp_get_num_threads()` and `omp_get_thr` we got number of threads running, and also current thread number, and then determine loop interval. Then run `check_nonce()`.

GCC also inserted `LOCK ADD` instruction right in the code, where MSVC generated call to separate DLL function:

Listing 65.6: GCC 4.8.1

```
        lock add        DWORD PTR checked[rip], 1
```

```
        call    GOMP_critical_start
        mov     ecx, DWORD PTR checked[rip]
        mov     edx, 351843721
        mov     eax, ecx
        imul    edx
        sar     edx, 13
        mov     eax, ecx
        sar     eax, 31
        sub     edx, eax
        mov     eax, edx
        imul    eax, eax, 100000
        sub     ecx, eax
        mov     eax, ecx
        test    eax, eax
        jne     .L7
        mov     eax, DWORD PTR checked[rip]
        mov     esi, eax
        mov     edi, OFFSET FLAT:.LC2 ; "checked=%d\n"
        mov     eax, 0
        call    printf
.L7:
        call    GOMP_critical_end
```

Functions prefixed with GOMP are from GNU OpenMP library. Unlike vcomp*.dll, its sources are freely available: https://github.com/mirrors/gcc/tree/master/libgomp.

Chapter 66

Itanium

Although almost failed, another very interesting architecture is Intel Itanium (IA64). While OOE[1] CPUs decides how to rearrange instructions and execute them in parallel, EPIC[2] was an attempt to shift these decisions to the compiler: to let it group instructions at the compile stage.

This result in notoriously complex compilers.

Here is one sample of IA64 code: simple cryptoalgorithm from Linux kernel:

Listing 66.1: Linux kernel 3.2.0.4

```
#define TEA_ROUNDS              32
#define TEA_DELTA               0x9e3779b9

static void tea_encrypt(struct crypto_tfm *tfm, u8 *dst, const ↙
    ↳ u8 *src)
{
        u32 y, z, n, sum = 0;
        u32 k0, k1, k2, k3;
        struct tea_ctx *ctx = crypto_tfm_ctx(tfm);
        const __le32 *in = (const __le32 *)src;
        __le32 *out = (__le32 *)dst;

        y = le32_to_cpu(in[0]);
        z = le32_to_cpu(in[1]);

        k0 = ctx->KEY[0];
        k1 = ctx->KEY[1];
        k2 = ctx->KEY[2];
        k3 = ctx->KEY[3];
```

[1] Out-of-order execution

[2] Explicitly parallel instruction computing

```
        n = TEA_ROUNDS;

        while (n-- > 0) {
                sum += TEA_DELTA;
                y += ((z << 4) + k0) ^ (z + sum) ^ ((z >> 5) + ↙
    ↳ k1);
                z += ((y << 4) + k2) ^ (y + sum) ^ ((y >> 5) + ↙
    ↳ k3);
        }

        out[0] = cpu_to_le32(y);
        out[1] = cpu_to_le32(z);
}
```

Here is how it was compiled:

Listing 66.2: Linux Kernel 3.2.0.4 for Itanium 2 (McKinley)

```
0090|                                     tea_encrypt:
0090|08 80 80 41 00 21                        adds r16 = 96, r32    ↙
    ↳             // ptr to ctx->KEY[2]
0096|80 C0 82 00 42 00                        adds r8 = 88, r32     ↙
    ↳             // ptr to ctx->KEY[0]
009C|00 00 04 00                              nop.i 0
00A0|09 18 70 41 00 21                        adds r3 = 92, r32     ↙
    ↳             // ptr to ctx->KEY[1]
00A6|F0 20 88 20 28 00                        ld4 r15 = [r34], 4    ↙
    ↳             // load z
00AC|44 06 01 84                              adds r32 = 100, r32;;↙
    ↳             // ptr to ctx->KEY[3]
00B0|08 98 00 20 10 10                        ld4 r19 = [r16]       ↙
    ↳             // r19=k2
00B6|00 01 00 00 42 40                        mov r16 = r0          ↙
    ↳             // r0 always contain zero
00BC|00 08 CA 00                              mov.i r2 = ar.lc      ↙
    ↳             // save lc register
00C0|05 70 00 44 10 10 9E FF FF FF 7F 20   ld4 r14 = [r34]       ↙
    ↳             // load y
00CC|92 F3 CE 6B                              movl r17 = 0↙
    ↳ xFFFFFFFF9E3779B9;; // TEA_DELTA
00D0|08 00 00 00 01 00                        nop.m 0
00D6|50 01 20 20 20 00                        ld4 r21 = [r8]        ↙
    ↳             // r21=k0
00DC|F0 09 2A 00                              mov.i ar.lc = 31      ↙
```

```
    ↳            // TEA_ROUNDS is 32
00E0|0A A0 00 06 10 10                    ld4 r20 = [r3];;     ↙
    ↳            // r20=k1
00E6|20 01 80 20 20 00                    ld4 r18 = [r32]      ↙
    ↳            // r18=k3
00EC|00 00 04 00                          nop.i 0
00F0|
00F0|                                 loc_F0:
00F0|09 80 40 22 00 20                    add r16 = r16, r17   ↙
    ↳            // r16=sum, r17=TEA_DELTA
00F6|D0 71 54 26 40 80                    shladd r29 = r14, 4, ↙
    ↳ r21        // r14=y, r21=k0
00FC|A3 70 68 52                          extr.u r28 = r14, 5, ↙
    ↳ 27;;
0100|03 F0 40 1C 00 20                    add r30 = r16, r14
0106|B0 E1 50 00 40 40                    add r27 = r28, r20;; ↙
    ↳            // r20=k1
010C|D3 F1 3C 80                          xor r26 = r29, r30;;
0110|0B C8 6C 34 0F 20                    xor r25 = r27, r26;;
0116|F0 78 64 00 40 00                    add r15 = r15, r25   ↙
    ↳            // r15=z
011C|00 00 04 00                          nop.i 0;;
0120|00 00 00 00 01 00                    nop.m 0
0126|80 51 3C 34 29 60                    extr.u r24 = r15, 5, ↙
    ↳ 27
012C|F1 98 4C 80                          shladd r11 = r15, 4, ↙
    ↳ r19        // r19=k2
0130|0B B8 3C 20 00 20                    add r23 = r15, r16;;
0136|A0 C0 48 00 40 00                    add r10 = r24, r18   ↙
    ↳            // r18=k3
013C|00 00 04 00                          nop.i 0;;
0140|0B 48 28 16 0F 20                    xor r9 = r10, r11;;
0146|60 B9 24 1E 40 00                    xor r22 = r23, r9
014C|00 00 04 00                          nop.i 0;;
0150|11 00 00 00 01 00                    nop.m 0
0156|E0 70 58 00 40 A0                    add r14 = r14, r22
015C|A0 FF FF 48                          br.cloop.sptk.few ↙
    ↳ loc_F0;;
0160|09 20 3C 42 90 15                    st4 [r33] = r15, 4   ↙
    ↳            // store z
0166|00 00 00 02 00 00                    nop.m 0
016C|20 08 AA 00                          mov.i ar.lc = r2;;   ↙
    ↳            // restore lc register
```

```
0170|11 00 38 42 90 11                    st4 [r33] = r14      ↙
    ↳           // store y
0176|00 00 00 02 00 80                    nop.i 0
017C|08 00 84 00                          br.ret.sptk.many b0;;
```

First of all, all IA64 instructions are grouped into 3-instruction bundles. Each bundle has size of 16 bytes and consists of template code + 3 instructions. IDA shows bundles into 6+6+4 bytes —you may easily spot the pattern.

All 3 instructions from each bundle usually executes simultaneously, unless one of instructions have "stop bit".

Supposedly, Intel and HP engineers gathered statistics of most occurred instruction patterns and decided to bring bundle types (AKA "templates"): a bundle code defines instruction types in the bundle. There are 12 of them. For example, zeroth bundle type is `MII`, meaning: first instruction is Memory (load or store), second and third are I (integer instructions). Another example is bundle type 0x1d: `MFB`: first instruction is Memory (load or store), second is Float (FPU instruction), third is Branch (branch instruction).

If compiler cannot pick suitable instruction to relevant bundle slot, it may insert NOP: you may see here `nop.i` instructions (NOP at the place where integer instructrion might be) or `nop.m` (a memory instruction might be at this slot). NOPs are inserted automatically when one use assembly language manually.

And that is not all. Bundles are also grouped. Each bundle may have "stop bit", so all the consecutive bundles with terminating bundle which have "stop bit" may be executed simultaneously. In practice, Itanium 2 may execute 2 bundles at once, resulting execution of 6 instructions at once.

So all instructions inside bundle and bundle group cannot interfere with each other (i.e., should not have data hazards). If they do, results will be undefined.

Each stop bit is marked in assembly language as `;;` (two semicolons) after instruction. So, instructions at [180-19c] may be executed simultaneously: they do not interfere. Next group is [1a0-1bc].

We also see a stop bit at 22c. The next instruction at 230 have stop bit too. This mean, this instruction is to be executed as isolated from all others (as in CISC). Indeed: the next instructrion at 236 use result from it (value in register r10), so they cannot be executed at the same time. Apparently, compiler was not able to find a better way to parallelize instructions, which is, in other words, to load CPU as much as possible, hence too much stop bits and NOPs. Manual assembly programming is tedious job as well: programmer should group instructions manually.

Programmer is still able to add stop-bits to each instructions, but this will degrade all performance Itanium was made for.

Interesting examples of manual IA64 assembly code can be found in Linux kernel sources:

http://lxr.free-electrons.com/source/arch/ia64/lib/.

Another introductory Itanium assembly paper: [5].

Another very interesting Itanium feature is *speculative execution* and NaT ("not a thing") bit, somewhat resembling NaN numbers: http://blogs.msdn.com/b/oldnewthing/archive/2004/01/19/60162.aspx.

Chapter 67

8086 memory model

Dealing with 16-bit programs for MS-DOS or Win16 (55.3 or 31.5), we can see that pointer consisting of two 16-bit values. What it means? Oh yes, that is another MS-DOS and 8086 weird artefact.

8086/8088 was a 16-bit CPU, but was able to address 20-bit address RAM (thus resulting 1MB external memory). External memory address space was divided between RAM (640KB max), ROM[1], windows for video memory, EMS cards, etc.

Let's also recall that 8086/8088 was in fact inheritor of 8-bit 8080 CPU. The 8080 has 16-bit memory spaces, i.e., it was able to address only 64KB. And probably of old software porting reason[2], 8086 can support 64KB windows, many of them placed simultaneously within 1MB address space. This is some kind of toy-level virtualization. All 8086 registers are 16-bit, so to address more, a special segment registers (CS, DS, ES, SS) were introduced. Each 20-bit pointer is calculated using values from a segment register and an address register pair (e.g. DS:BX) as follows:

$$real_address = (segment_register \ll 4) + address_register$$

For example, graphics (EGA[3], VGA[4]) video RAM window on old IBM PC-compatibles has size of 64KB. For accessing it, a 0xA000 value should be stored in one of segment registers, e.g. into DS. Then DS:0 will address the very first byte of video RAM and DS:0xFFFF is the very last byte of RAM. The real address on 20-bit address bus, however, will range from 0xA0000 to 0xAFFFF.

The program may contain hardcoded addresses like 0x1234, but OS may need to load program on arbitrary addresses, so it recalculates segment register values in such a way, so the program will not care about where in the RAM it is placed.

So, any pointer it old MS-DOS environment was in fact consisted of segment address and the address inside segment, i.e., two 16-bit values. 20-bit was enough for

[1] Read-only memory
[2] I'm not 100% sure here
[3] Enhanced Graphics Adapter
[4] Video Graphics Array

that, though, but one will need to recalculate the addresses very often: passing more information on stack is seems better space/convenience balance.

By the way, because of all this, it was not possible to allocate the memory block larger than 64KB.

Segment registers were reused at 80286 as selectors, serving different function.

When 80386 CPU and computers with bigger RAM were introduced, MS-DOS was still popular, so the DOS extenders are emerged: these were in fact a step toward "serious" OS, switching CPU into protected mode and providing much better memory APIs for the programs which still needs to be runned from MS-DOS. Widely popular examples include DOS/4GW (DOOM video game was compiled for it), Phar Lap, PMODE.

By the way, the same was of addressing memory was in 16-bit line of Windows 3.x, before Win32.

Chapter 68

Basic blocks reordering

68.1 Profile-guided optimization

This optimization method may move some basic blocks to another section of the executable binary file.

Obviously, there are parts in function which are executed most often (e.g., loop bodies) and less often (e.g., error reporting code, exception handlers).

The compiler adding instrumentation code into the executable, then developer run it with a lot of tests for statistics collecting. Then the compiler, with the help of statistics gathered, prepares final executable file with all infrequently executed code moved into another section.

As a result, all frequently executed function code is compacted, and that is very important for execution speed and cache memory.

Example from Oracle RDBMS code, which was compiled by Intel C++:

Listing 68.1: orageneric11.dll (win32)

```
                public _skgfsync
_skgfsync       proc near

; address 0x6030D86A

                db      66h
                nop
                push    ebp
                mov     ebp, esp
                mov     edx, [ebp+0Ch]
                test    edx, edx
                jz      short loc_6030D884
                mov     eax, [edx+30h]
```

```
                test    eax, 400h
                jnz     __VInfreq__skgfsync  ; write to log
continue:
                mov     eax, [ebp+8]
                mov     edx, [ebp+10h]
                mov     dword ptr [eax], 0
                lea     eax, [edx+0Fh]
                and     eax, 0FFFFFFFCh
                mov     ecx, [eax]
                cmp     ecx, 45726963h
                jnz     error                ; exit with error
                mov     esp, ebp
                pop     ebp
                retn
_skgfsync       endp

...

; address 0x60B953F0

__VInfreq__skgfsync:
                mov     eax, [edx]
                test    eax, eax
                jz      continue
                mov     ecx, [ebp+10h]
                push    ecx
                mov     ecx, [ebp+8]
                push    edx
                push    ecx
                push    offset ... ; "skgfsync(se=0x%x, ctx=0x%x↙
    ↪ , iov=0x%x)\n"
                push    dword ptr [edx+4]
                call    dword ptr [eax] ; write to log
                add     esp, 14h
                jmp     continue

error:
                mov     edx, [ebp+8]
                mov     dword ptr [edx], 69AAh ; 27050 "function↙
    ↪  called with invalid FIB/IOV structure"
                mov     eax, [eax]
                mov     [edx+4], eax
                mov     dword ptr [edx+8], 0FA4h ; 4004
```

```
                mov     esp, ebp
                pop     ebp
                retn
; END OF FUNCTION CHUNK FOR _skgfsync
```

The distance of addresses of these two code fragments is almost 9 MB.

All infrequently executed code was placed at the end of the code section of DLL file, among all function parts. This part of function was marked by Intel C++ compiler with `VInfreq` prefix. Here we see that a part of function which writes to log-file (presumably in case of error or warning or something like that) which was probably not executed very often when Oracle developers gathered statistics (if was executed at all). The writing to log basic block is eventually return control flow into the "hot" part of the function.

Another "infrequent" part is a basic block returning error code 27050.

In Linux ELF files, all infrequently executed code is moved by Intel C++ into separate `text.unlikely` section, leaving all "hot" code in the `text.hot` section.

From a reverse engineer's perspective, this information may help to split the function to its core and error handling parts.

Part VIII

Books/blogs worth reading

Chapter 69

Books

69.1 Windows

[30].

69.2 C/C++

[16].

69.3 x86 / x86-64

[14], [1]

69.4 ARM

ARM manuals: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.architecture.reference/index.html

Chapter 70

Blogs

70.1 Windows

- Microsoft: Raymond Chen
- http://www.nynaeve.net/

Chapter 71

Other

There are two excellent RE[1]-related subreddits on reddit.com: ReverseEngineering and REMath (for the topics on the intersection of RE and mathematics).

There are also RE part of Stack Exchange website: http://reverseengineering.stackexchange.com/.

[1]Reverse Engineering

Part IX

Exercises

There are two questions almost for every exercise, if otherwise is not specified:

1) What this function does? Answer in one-sentence form.

2) Rewrite this function into C/C++.

It is allowed to use Google to search for any leads. However, if you like to make your task harder, you may try to solve it without Google.

Hints and solutions are in the appendix of this book.

Chapter 72

Level 1

Level 1 exercises are ones you may try to solve in mind.

72.1 Exercise 1.1

72.1.1 MSVC 2012 x64 + /Ox

```
a$ = 8
b$ = 16
f       PROC
        cmp     ecx, edx
        cmovg   edx, ecx
        mov     eax, edx
        ret     0
f       ENDP
```

72.1.2 Keil (ARM)

```
        CMP     r0,r1
        MOVLE   r0,r1
        BX      lr
```

72.1.3 Keil (thumb)

```
        CMP     r0,r1
        BGT     |L0.6|
```

```
        MOVS    r0,r1
|L0.6|
        BX      lr
```

72.2 Exercise 1.2

Why LOOP instruction is not used by compilers anymore?

72.3 Exercise 1.3

Take an loop example from "Loops" section (12), compile it in your favorite OS and compiler and modify (patch) executable file, so the loop range will be [6..20].

72.4 Exercise 1.4

This program requires password. Find it.

As an additional exercise, try to change the password by patching executable file. It may also has a different length.

Try also to crash the program using only string input.

- win32: http://yurichev.com/RE-exercises/1/4/password1.exe
- Linux x86: http://yurichev.com/RE-exercises/1/4/password1_Linux_x86.tar
- Mac OS X: http://yurichev.com/RE-exercises/1/4/password1_MacOSX64.tar

Chapter 73

Level 2

For solving exercises of level 2, you probably will need text editor or paper with pencil.

73.1 Exercise 2.1

This is standard C library function. Source code taken from OpenWatcom.

73.1.1 MSVC 2010

```
_TEXT    SEGMENT
_input$ = 8                             ; size = 1
_f PROC
    push   ebp
    mov    ebp, esp
    movsx  eax, BYTE PTR _input$[ebp]
    cmp    eax, 97          ; 00000061H
    jl     SHORT $LN1@f
    movsx  ecx, BYTE PTR _input$[ebp]
    cmp    ecx, 122         ; 0000007aH
    jg     SHORT $LN1@f
    movsx  edx, BYTE PTR _input$[ebp]
    sub    edx, 32          ; 00000020H
    mov    BYTE PTR _input$[ebp], dl
$LN1@f:
    mov    al, BYTE PTR _input$[ebp]
    pop    ebp
    ret    0
_f ENDP
```

```
_TEXT    ENDS
```

73.1.2 GCC 4.4.1 + -O3

```
_f              proc near

input           = dword ptr  8

                push    ebp
                mov     ebp, esp
                movzx   eax, byte ptr [ebp+input]
                lea     edx, [eax-61h]
                cmp     dl, 19h
                ja      short loc_80483F2
                sub     eax, 20h

loc_80483F2:
                pop     ebp
                retn
_f              endp
```

73.1.3 Keil (ARM) + -O3

```
        SUB     r1,r0,#0x61
        CMP     r1,#0x19
        SUBLS   r0,r0,#0x20
        ANDLS   r0,r0,#0xff
        BX      lr
```

73.1.4 Keil (thumb) + -O3

```
        MOVS    r1,r0
        SUBS    r1,r1,#0x61
        CMP     r1,#0x19
        BHI     |L0.14|
        SUBS    r0,r0,#0x20
        LSLS    r0,r0,#24
        LSRS    r0,r0,#24
|L0.14|
        BX      lr
```

73.2 Exercise 2.2

. This is also standard C library function. Source code is taken from OpenWatcom and modified slightly.

This function also use these standard C functions: isspace() and isdigit().

73.2.1 MSVC 2010 + /Ox

```
EXTRN    _isdigit:PROC
EXTRN    _isspace:PROC
EXTRN    ___ptr_check:PROC
; Function compile flags: /Ogtpy
_TEXT    SEGMENT
_p$ = 8                                 ; size = 4
_f    PROC
    push   ebx
    push   esi
    mov    esi, DWORD PTR _p$[esp+4]
    push   edi
    push   0
    push   esi
    call   ___ptr_check
    mov    eax, DWORD PTR [esi]
    push   eax
    call   _isspace
    add    esp, 12                      ; 0000000cH
    test   eax, eax
    je     SHORT $LN6@f
    npad   2
$LL7@f:
    mov    ecx, DWORD PTR [esi+4]
    add    esi, 4
    push   ecx
    call   _isspace
    add    esp, 4
    test   eax, eax
    jne    SHORT $LL7@f
$LN6@f:
    mov    bl, BYTE PTR [esi]
    cmp    bl, 43                       ; 0000002bH
    je     SHORT $LN4@f
    cmp    bl, 45                       ; 0000002dH
    jne    SHORT $LN5@f
```

```
$LN4@f:
    add     esi, 4
$LN5@f:
    mov     edx, DWORD PTR [esi]
    push    edx
    xor     edi, edi
    call    _isdigit
    add     esp, 4
    test    eax, eax
    je      SHORT $LN2@f
$LL3@f:
    mov     ecx, DWORD PTR [esi]
    mov     edx, DWORD PTR [esi+4]
    add     esi, 4
    lea     eax, DWORD PTR [edi+edi*4]
    push    edx
    lea     edi, DWORD PTR [ecx+eax*2-48]
    call    _isdigit
    add     esp, 4
    test    eax, eax
    jne     SHORT $LL3@f
$LN2@f:
    cmp     bl, 45                      ; 0000002dH
    jne     SHORT $LN14@f
    neg     edi
$LN14@f:
    mov     eax, edi
    pop     edi
    pop     esi
    pop     ebx
    ret     0
_f    ENDP
_TEXT    ENDS
```

73.2.2 GCC 4.4.1

This exercise is slightly harder since GCC compiled isspace() and isdigit() functions as inline-functions and inserted their bodies right into the code.

```
_f              proc near

var_10          = dword ptr -10h
var_9           = byte ptr -9
```

```
input           = dword ptr  8

                push    ebp
                mov     ebp, esp
                sub     esp, 18h
                jmp     short loc_8048410
loc_804840C:
                add     [ebp+input], 4

loc_8048410:
                call    ___ctype_b_loc
                mov     edx, [eax]
                mov     eax, [ebp+input]
                mov     eax, [eax]
                add     eax, eax
                lea     eax, [edx+eax]
                movzx   eax, word ptr [eax]
                movzx   eax, ax
                and     eax, 2000h
                test    eax, eax
                jnz     short loc_804840C
                mov     eax, [ebp+input]
                mov     eax, [eax]
                mov     [ebp+var_9], al
                cmp     [ebp+var_9], '+'
                jz      short loc_8048444
                cmp     [ebp+var_9], '-'
                jnz     short loc_8048448

loc_8048444:
                add     [ebp+input], 4

loc_8048448:
                mov     [ebp+var_10], 0
                jmp     short loc_8048471

loc_8048451:
                mov     edx, [ebp+var_10]
                mov     eax, edx
                shl     eax, 2
                add     eax, edx
                add     eax, eax
                mov     edx, eax
```

```
                mov     eax, [ebp+input]
                mov     eax, [eax]
                lea     eax, [edx+eax]
                sub     eax, 30h
                mov     [ebp+var_10], eax
                add     [ebp+input], 4

loc_8048471:
                call    ___ctype_b_loc
                mov     edx, [eax]
                mov     eax, [ebp+input]
                mov     eax, [eax]
                add     eax, eax
                lea     eax, [edx+eax]
                movzx   eax, word ptr [eax]
                movzx   eax, ax
                and     eax, 800h
                test    eax, eax
                jnz     short loc_8048451
                cmp     [ebp+var_9], 2Dh
                jnz     short loc_804849A
                neg     [ebp+var_10]

loc_804849A:
                mov     eax, [ebp+var_10]
                leave
                retn
_f              endp
```

73.2.3 Keil (ARM) + -O3

```
        PUSH    {r4,lr}
        MOV     r4,r0
        BL      __rt_ctype_table
        LDR     r2,[r0,#0]
|L0.16|
        LDR     r0,[r4,#0]
        LDRB    r0,[r2,r0]
        TST     r0,#1
        ADDNE   r4,r4,#4
        BNE     |L0.16|
        LDRB    r1,[r4,#0]
```

```
        MOV     r0,#0
        CMP     r1,#0x2b
        CMPNE   r1,#0x2d
        ADDEQ   r4,r4,#4
        B       |L0.76|
|L0.60|
        ADD     r0,r0,r0,LSL #2
        ADD     r0,r3,r0,LSL #1
        SUB     r0,r0,#0x30
        ADD     r4,r4,#4
|L0.76|
        LDR     r3,[r4,#0]
        LDRB    r12,[r2,r3]
        CMP     r12,#0x20
        BEQ     |L0.60|
        CMP     r1,#0x2d
        RSBEQ   r0,r0,#0
        POP     {r4,pc}
```

73.2.4 Keil (thumb) + -O3

```
        PUSH    {r4-r6,lr}
        MOVS    r4,r0
        BL      __rt_ctype_table
        LDR     r2,[r0,#0]
        B       |L0.14|
|L0.12|
        ADDS    r4,r4,#4
|L0.14|
        LDR     r0,[r4,#0]
        LDRB    r0,[r2,r0]
        LSLS    r0,r0,#31
        BNE     |L0.12|
        LDRB    r1,[r4,#0]
        CMP     r1,#0x2b
        BEQ     |L0.32|
        CMP     r1,#0x2d
        BNE     |L0.34|
|L0.32|
        ADDS    r4,r4,#4
|L0.34|
        MOVS    r0,#0
```

```
        B       |L0.48|
|L0.38|
        MOVS    r5,#0xa
        MULS    r0,r5,r0
        ADDS    r4,r4,#4
        SUBS    r0,r0,#0x30
        ADDS    r0,r3,r0
|L0.48|
        LDR     r3,[r4,#0]
        LDRB    r5,[r2,r3]
        CMP     r5,#0x20
        BEQ     |L0.38|
        CMP     r1,#0x2d
        BNE     |L0.62|
        RSBS    r0,r0,#0
|L0.62|
        POP     {r4-r6,pc}
```

73.3 Exercise 2.3

This is standard C function too, actually, two functions working in pair. Source code taken from MSVC 2010 and modified slightly.

The matter of modification is that this function can work properly in multi-threaded environment, and I removed its support for simplification (or for confusion).

73.3.1 MSVC 2010 + /Ox

```
_BSS    SEGMENT
_v    DD    01H DUP (?)
_BSS    ENDS

_TEXT    SEGMENT
_s$ = 8                                 ; size = 4
f1    PROC
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _s$[ebp]
    mov    DWORD PTR _v, eax
    pop    ebp
    ret    0
f1    ENDP
```

```
_TEXT    ENDS
PUBLIC    f2

_TEXT    SEGMENT
f2    PROC
    push   ebp
    mov    ebp, esp
    mov    eax, DWORD PTR _v
    imul   eax, 214013      ; 000343fdH
    add    eax, 2531011     ; 00269ec3H
    mov    DWORD PTR _v, eax
    mov    eax, DWORD PTR _v
    shr    eax, 16          ; 00000010H
    and    eax, 32767       ; 00007fffH
    pop    ebp
    ret    0
f2    ENDP
_TEXT    ENDS
END
```

73.3.2 GCC 4.4.1

```
                public f1
f1              proc near

arg_0           = dword ptr  8

                push    ebp
                mov     ebp, esp
                mov     eax, [ebp+arg_0]
                mov     ds:v, eax
                pop     ebp
                retn
f1              endp

                public f2
f2              proc near
                push    ebp
                mov     ebp, esp
                mov     eax, ds:v
                imul    eax, 343FDh
                add     eax, 269EC3h
```

```
                mov     ds:v, eax
                mov     eax, ds:v
                shr     eax, 10h
                and     eax, 7FFFh
                pop     ebp
                retn
f2              endp

bss             segment dword public 'BSS' use32
                assume cs:_bss
                dd ?
bss             ends
```

73.3.3 Keil (ARM) + -O3

```
f1 PROC
        LDR      r1,|L0.52|
        STR      r0,[r1,#0]  ; v
        BX       lr
        ENDP

f2 PROC
        LDR      r0,|L0.52|
        LDR      r2,|L0.56|
        LDR      r1,[r0,#0]  ; v
        MUL      r1,r2,r1
        LDR      r2,|L0.60|
        ADD      r1,r1,r2
        STR      r1,[r0,#0]  ; v
        MVN      r0,#0x8000
        AND      r0,r0,r1,LSR #16
        BX       lr
        ENDP

|L0.52|
        DCD      ||.data||
|L0.56|
        DCD      0x000343fd
|L0.60|
        DCD      0x00269ec3
```

73.3.4 Keil (thumb) + -O3

```
f1 PROC
        LDR      r1,|L0.28|
        STR      r0,[r1,#0]  ; v
        BX       lr
        ENDP

f2 PROC
        LDR      r0,|L0.28|
        LDR      r2,|L0.32|
        LDR      r1,[r0,#0]  ; v
        MULS     r1,r2,r1
        LDR      r2,|L0.36|
        ADDS     r1,r1,r2
        STR      r1,[r0,#0]  ; v
        LSLS     r0,r1,#1
        LSRS     r0,r0,#17
        BX       lr
        ENDP

|L0.28|
        DCD      ||.data||
|L0.32|
        DCD      0x000343fd
|L0.36|
        DCD      0x00269ec3
```

73.4 Exercise 2.4

This is standard C library function. Source code taken from MSVC 2010.

73.4.1 MSVC 2010 + /Ox

```
PUBLIC    _f
_TEXT    SEGMENT
_arg1$ = 8            ; size = 4
_arg2$ = 12           ; size = 4
_f    PROC
    push   esi
    mov    esi, DWORD PTR _arg1$[esp]
```

```
    push   edi
    mov    edi, DWORD PTR _arg2$[esp+4]
    cmp    BYTE PTR [edi], 0
    mov    eax, esi
    je     SHORT $LN7@f
    mov    dl, BYTE PTR [esi]
    push   ebx
    test   dl, dl
    je     SHORT $LN4@f
    sub    esi, edi
    npad   6
$LL5@f:
    mov    ecx, edi
    test   dl, dl
    je     SHORT $LN2@f
$LL3@f:
    mov    dl, BYTE PTR [ecx]
    test   dl, dl
    je     SHORT $LN14@f
    movsx  ebx, BYTE PTR [esi+ecx]
    movsx  edx, dl
    sub    ebx, edx
    jne    SHORT $LN2@f
    inc    ecx
    cmp    BYTE PTR [esi+ecx], bl
    jne    SHORT $LL3@f
$LN2@f:
    cmp    BYTE PTR [ecx], 0
    je     SHORT $LN14@f
    mov    dl, BYTE PTR [eax+1]
    inc    eax
    inc    esi
    test   dl, dl
    jne    SHORT $LL5@f
    xor    eax, eax
    pop    ebx
    pop    edi
    pop    esi
    ret    0
_f    ENDP
_TEXT    ENDS
END
```

73.4.2 GCC 4.4.1

```
                public f
f               proc near

var_C           = dword ptr -0Ch
var_8           = dword ptr -8
var_4           = dword ptr -4
arg_0           = dword ptr  8
arg_4           = dword ptr  0Ch

                push    ebp
                mov     ebp, esp
                sub     esp, 10h
                mov     eax, [ebp+arg_0]
                mov     [ebp+var_4], eax
                mov     eax, [ebp+arg_4]
                movzx   eax, byte ptr [eax]
                test    al, al
                jnz     short loc_8048443
                mov     eax, [ebp+arg_0]
                jmp     short locret_8048453

loc_80483F4:
                mov     eax, [ebp+var_4]
                mov     [ebp+var_8], eax
                mov     eax, [ebp+arg_4]
                mov     [ebp+var_C], eax
                jmp     short loc_804840A

loc_8048402:
                add     [ebp+var_8], 1
                add     [ebp+var_C], 1

loc_804840A:
                mov     eax, [ebp+var_8]
                movzx   eax, byte ptr [eax]
                test    al, al
                jz      short loc_804842E
                mov     eax, [ebp+var_C]
                movzx   eax, byte ptr [eax]
                test    al, al
                jz      short loc_804842E
```

```
                mov     eax, [ebp+var_8]
                movzx   edx, byte ptr [eax]
                mov     eax, [ebp+var_C]
                movzx   eax, byte ptr [eax]
                cmp     dl, al
                jz      short loc_8048402

loc_804842E:
                mov     eax, [ebp+var_C]
                movzx   eax, byte ptr [eax]
                test    al, al
                jnz     short loc_804843D
                mov     eax, [ebp+var_4]
                jmp     short locret_8048453

loc_804843D:
                add     [ebp+var_4], 1
                jmp     short loc_8048444


loc_8048443:
                nop

loc_8048444:
                mov     eax, [ebp+var_4]
                movzx   eax, byte ptr [eax]
                test    al, al
                jnz     short loc_80483F4
                mov     eax, 0

locret_8048453:
                leave
                retn
f               endp
```

73.4.3 Keil (ARM) + -O3

```
        PUSH    {r4,lr}
        LDRB    r2,[r1,#0]
        CMP     r2,#0
        POPEQ   {r4,pc}
        B       |L0.80|
```

```
|L0.20|
        LDRB     r12,[r3,#0]
        CMP      r12,#0
        BEQ      |L0.64|
        LDRB     r4,[r2,#0]
        CMP      r4,#0
        POPEQ    {r4,pc}
        CMP      r12,r4
        ADDEQ    r3,r3,#1
        ADDEQ    r2,r2,#1
        BEQ      |L0.20|
        B        |L0.76|
|L0.64|
        LDRB     r2,[r2,#0]
        CMP      r2,#0
        POPEQ    {r4,pc}
|L0.76|
        ADD      r0,r0,#1
|L0.80|
        LDRB     r2,[r0,#0]
        CMP      r2,#0
        MOVNE    r3,r0
        MOVNE    r2,r1
        MOVEQ    r0,#0
        BNE      |L0.20|
        POP      {r4,pc}
```

73.4.4 Keil (thumb) + -O3

```
        PUSH     {r4,r5,lr}
        LDRB     r2,[r1,#0]
        CMP      r2,#0
        BEQ      |L0.54|
        B        |L0.46|
|L0.10|
        MOVS     r3,r0
        MOVS     r2,r1
        B        |L0.20|
|L0.16|
        ADDS     r3,r3,#1
        ADDS     r2,r2,#1
|L0.20|
```

```
        LDRB     r4,[r3,#0]
        CMP      r4,#0
        BEQ      |L0.38|
        LDRB     r5,[r2,#0]
        CMP      r5,#0
        BEQ      |L0.54|
        CMP      r4,r5
        BEQ      |L0.16|
        B        |L0.44|
|L0.38|
        LDRB     r2,[r2,#0]
        CMP      r2,#0
        BEQ      |L0.54|
|L0.44|
        ADDS     r0,r0,#1
|L0.46|
        LDRB     r2,[r0,#0]
        CMP      r2,#0
        BNE      |L0.10|
        MOVS     r0,#0
|L0.54|
        POP      {r4,r5,pc}
```

73.5 Exercise 2.5

This exercise is rather on knowledge than on reading code.
. The function is taken from OpenWatcom.

73.5.1 MSVC 2010 + /Ox

```
_DATA    SEGMENT
COMM     __v:DWORD
_DATA    ENDS
PUBLIC   __real@3e45798ee2308c3a
PUBLIC   __real@4147ffff80000000
PUBLIC   __real@4150017ec0000000
PUBLIC   _f
EXTRN    __fltused:DWORD
CONST    SEGMENT
__real@3e45798ee2308c3a DQ 03e45798ee2308c3ar    ; 1e-008
__real@4147ffff80000000 DQ 04147ffff80000000r    ; 3.14573e+006
```

```
__real@4150017ec0000000 DQ 04150017ec0000000r    ; 4.19584e+006
CONST    ENDS
_TEXT    SEGMENT
_v1$ = -16                ; size = 8
_v2$ = -8                 ; size = 8
_f    PROC
    sub    esp, 16        ; 00000010H
    fld    QWORD PTR __real@4150017ec0000000
    fstp   QWORD PTR _v1$[esp+16]
    fld    QWORD PTR __real@4147ffff80000000
    fstp   QWORD PTR _v2$[esp+16]
    fld    QWORD PTR _v1$[esp+16]
    fld    QWORD PTR _v1$[esp+16]
    fdiv   QWORD PTR _v2$[esp+16]
    fmul   QWORD PTR _v2$[esp+16]
    fsubp  ST(1), ST(0)
    fcomp  QWORD PTR __real@3e45798ee2308c3a
    fnstsw ax
    test   ah, 65         ; 00000041H
    jne    SHORT $LN1@f
    or     DWORD PTR __v, 1
$LN1@f:
    add    esp, 16        ; 00000010H
    ret    0
_f    ENDP
_TEXT    ENDS
```

73.6 Exercise 2.6

73.6.1 MSVC 2010 + /Ox

```
PUBLIC    _f
; Function compile flags: /Ogtpy
_TEXT    SEGMENT
_k0$ = -12            ; size = 4
_k3$ = -8             ; size = 4
_k2$ = -4             ; size = 4
_v$ = 8               ; size = 4
_k1$ = 12             ; size = 4
_k$ = 12              ; size = 4
_f    PROC
    sub    esp, 12    ; 0000000cH
```

```
    mov     ecx, DWORD PTR _v$[esp+8]
    mov     eax, DWORD PTR [ecx]
    mov     ecx, DWORD PTR [ecx+4]
    push    ebx
    push    esi
    mov     esi, DWORD PTR _k$[esp+16]
    push    edi
    mov     edi, DWORD PTR [esi]
    mov     DWORD PTR _k0$[esp+24], edi
    mov     edi, DWORD PTR [esi+4]
    mov     DWORD PTR _k1$[esp+20], edi
    mov     edi, DWORD PTR [esi+8]
    mov     esi, DWORD PTR [esi+12]
    xor     edx, edx
    mov     DWORD PTR _k2$[esp+24], edi
    mov     DWORD PTR _k3$[esp+24], esi
    lea     edi, DWORD PTR [edx+32]
$LL8@f:
    mov     esi, ecx
    shr     esi, 5
    add     esi, DWORD PTR _k1$[esp+20]
    mov     ebx, ecx
    shl     ebx, 4
    add     ebx, DWORD PTR _k0$[esp+24]
    sub     edx, 1640531527   ; 61c88647H
    xor     esi, ebx
    lea     ebx, DWORD PTR [edx+ecx]
    xor     esi, ebx
    add     eax, esi
    mov     esi, eax
    shr     esi, 5
    add     esi, DWORD PTR _k3$[esp+24]
    mov     ebx, eax
    shl     ebx, 4
    add     ebx, DWORD PTR _k2$[esp+24]
    xor     esi, ebx
    lea     ebx, DWORD PTR [edx+eax]
    xor     esi, ebx
    add     ecx, esi
    dec     edi
    jne     SHORT $LL8@f
    mov     edx, DWORD PTR _v$[esp+20]
    pop     edi
```

```
    pop     esi
    mov     DWORD PTR [edx], eax
    mov     DWORD PTR [edx+4], ecx
    pop     ebx
    add     esp, 12                     ; 0000000cH
    ret     0
_f    ENDP
```

73.6.2 Keil (ARM) + -O3

```
        PUSH    {r4-r10,lr}
        ADD     r5,r1,#8
        LDM     r5,{r5,r7}
        LDR     r2,[r0,#4]
        LDR     r3,[r0,#0]
        LDR     r4,|L0.116|
        LDR     r6,[r1,#4]
        LDR     r8,[r1,#0]
        MOV     r12,#0
        MOV     r1,r12
|L0.40|
        ADD     r12,r12,r4
        ADD     r9,r8,r2,LSL #4
        ADD     r10,r2,r12
        EOR     r9,r9,r10
        ADD     r10,r6,r2,LSR #5
        EOR     r9,r9,r10
        ADD     r3,r3,r9
        ADD     r9,r5,r3,LSL #4
        ADD     r10,r3,r12
        EOR     r9,r9,r10
        ADD     r10,r7,r3,LSR #5
        EOR     r9,r9,r10
        ADD     r1,r1,#1
        CMP     r1,#0x20
        ADD     r2,r2,r9
        STRCS   r2,[r0,#4]
        STRCS   r3,[r0,#0]
        BCC     |L0.40|
        POP     {r4-r10,pc}

|L0.116|
```

```
        DCD     0x9e3779b9
```

73.6.3 Keil (thumb) + -O3

```
        PUSH    {r1-r7,lr}
        LDR     r5,|L0.84|
        LDR     r3,[r0,#0]
        LDR     r2,[r0,#4]
        STR     r5,[sp,#8]
        MOVS    r6,r1
        LDM     r6,{r6,r7}
        LDR     r5,[r1,#8]
        STR     r6,[sp,#4]
        LDR     r6,[r1,#0xc]
        MOVS    r4,#0
        MOVS    r1,r4
        MOV     lr,r5
        MOV     r12,r6
        STR     r7,[sp,#0]
|L0.30|
        LDR     r5,[sp,#8]
        LSLS    r6,r2,#4
        ADDS    r4,r4,r5
        LDR     r5,[sp,#4]
        LSRS    r7,r2,#5
        ADDS    r5,r6,r5
        ADDS    r6,r2,r4
        EORS    r5,r5,r6
        LDR     r6,[sp,#0]
        ADDS    r1,r1,#1
        ADDS    r6,r7,r6
        EORS    r5,r5,r6
        ADDS    r3,r5,r3
        LSLS    r5,r3,#4
        ADDS    r6,r3,r4
        ADD     r5,r5,lr
        EORS    r5,r5,r6
        LSRS    r6,r3,#5
        ADD     r6,r6,r12
        EORS    r5,r5,r6
        ADDS    r2,r5,r2
        CMP     r1,#0x20
```

```
        BCC     |L0.30|
        STR     r3,[r0,#0]
        STR     r2,[r0,#4]
        POP     {r1-r7,pc}

|L0.84|
        DCD     0x9e3779b9
```

73.7 Exercise 2.7

This function is taken from Linux 2.6 kernel.

73.7.1 MSVC 2010 + /Ox

```
_table    db 000h, 080h, 040h, 0c0h, 020h, 0a0h, 060h, 0e0h
    db 010h, 090h, 050h, 0d0h, 030h, 0b0h, 070h, 0f0h
    db 008h, 088h, 048h, 0c8h, 028h, 0a8h, 068h, 0e8h
    db 018h, 098h, 058h, 0d8h, 038h, 0b8h, 078h, 0f8h
    db 004h, 084h, 044h, 0c4h, 024h, 0a4h, 064h, 0e4h
    db 014h, 094h, 054h, 0d4h, 034h, 0b4h, 074h, 0f4h
    db 00ch, 08ch, 04ch, 0cch, 02ch, 0ach, 06ch, 0ech
    db 01ch, 09ch, 05ch, 0dch, 03ch, 0bch, 07ch, 0fch
    db 002h, 082h, 042h, 0c2h, 022h, 0a2h, 062h, 0e2h
    db 012h, 092h, 052h, 0d2h, 032h, 0b2h, 072h, 0f2h
    db 00ah, 08ah, 04ah, 0cah, 02ah, 0aah, 06ah, 0eah
    db 01ah, 09ah, 05ah, 0dah, 03ah, 0bah, 07ah, 0fah
    db 006h, 086h, 046h, 0c6h, 026h, 0a6h, 066h, 0e6h
    db 016h, 096h, 056h, 0d6h, 036h, 0b6h, 076h, 0f6h
    db 00eh, 08eh, 04eh, 0ceh, 02eh, 0aeh, 06eh, 0eeh
    db 01eh, 09eh, 05eh, 0deh, 03eh, 0beh, 07eh, 0feh
    db 001h, 081h, 041h, 0c1h, 021h, 0a1h, 061h, 0e1h
    db 011h, 091h, 051h, 0d1h, 031h, 0b1h, 071h, 0f1h
    db 009h, 089h, 049h, 0c9h, 029h, 0a9h, 069h, 0e9h
    db 019h, 099h, 059h, 0d9h, 039h, 0b9h, 079h, 0f9h
    db 005h, 085h, 045h, 0c5h, 025h, 0a5h, 065h, 0e5h
    db 015h, 095h, 055h, 0d5h, 035h, 0b5h, 075h, 0f5h
    db 00dh, 08dh, 04dh, 0cdh, 02dh, 0adh, 06dh, 0edh
    db 01dh, 09dh, 05dh, 0ddh, 03dh, 0bdh, 07dh, 0fdh
    db 003h, 083h, 043h, 0c3h, 023h, 0a3h, 063h, 0e3h
    db 013h, 093h, 053h, 0d3h, 033h, 0b3h, 073h, 0f3h
    db 00bh, 08bh, 04bh, 0cbh, 02bh, 0abh, 06bh, 0ebh
```

```
    db 01bh, 09bh, 05bh, 0dbh, 03bh, 0bbh, 07bh, 0fbh
    db 007h, 087h, 047h, 0c7h, 027h, 0a7h, 067h, 0e7h
    db 017h, 097h, 057h, 0d7h, 037h, 0b7h, 077h, 0f7h
    db 00fh, 08fh, 04fh, 0cfh, 02fh, 0afh, 06fh, 0efh
    db 01fh, 09fh, 05fh, 0dfh, 03fh, 0bfh, 07fh, 0ffh

f               proc near

arg_0           = dword ptr  4

                mov     edx, [esp+arg_0]
                movzx   eax, dl
                movzx   eax, _table[eax]
                mov     ecx, edx
                shr     edx, 8
                movzx   edx, dl
                movzx   edx, _table[edx]
                shl     ax, 8
                movzx   eax, ax
                or      eax, edx
                shr     ecx, 10h
                movzx   edx, cl
                movzx   edx, _table[edx]
                shr     ecx, 8
                movzx   ecx, cl
                movzx   ecx, _table[ecx]
                shl     dx, 8
                movzx   edx, dx
                shl     eax, 10h
                or      edx, ecx
                or      eax, edx
                retn
f               endp
```

73.7.2 Keil (ARM) + -O3

```
f2 PROC
        LDR      r1,|L0.76|
        LDRB     r2,[r1,r0,LSR #8]
        AND      r0,r0,#0xff
        LDRB     r0,[r1,r0]
        ORR      r0,r2,r0,LSL #8
```

```
        BX      lr
        ENDP

f3 PROC
        MOV     r3,r0
        LSR     r0,r0,#16
        PUSH    {lr}
        BL      f2
        MOV     r12,r0
        LSL     r0,r3,#16
        LSR     r0,r0,#16
        BL      f2
        ORR     r0,r12,r0,LSL #16
        POP     {pc}
        ENDP

|L0.76|
        DCB     0x00,0x80,0x40,0xc0
        DCB     0x20,0xa0,0x60,0xe0
        DCB     0x10,0x90,0x50,0xd0
        DCB     0x30,0xb0,0x70,0xf0
        DCB     0x08,0x88,0x48,0xc8
        DCB     0x28,0xa8,0x68,0xe8
        DCB     0x18,0x98,0x58,0xd8
        DCB     0x38,0xb8,0x78,0xf8
        DCB     0x04,0x84,0x44,0xc4
        DCB     0x24,0xa4,0x64,0xe4
        DCB     0x14,0x94,0x54,0xd4
        DCB     0x34,0xb4,0x74,0xf4
        DCB     0x0c,0x8c,0x4c,0xcc
        DCB     0x2c,0xac,0x6c,0xec
        DCB     0x1c,0x9c,0x5c,0xdc
        DCB     0x3c,0xbc,0x7c,0xfc
        DCB     0x02,0x82,0x42,0xc2
        DCB     0x22,0xa2,0x62,0xe2
        DCB     0x12,0x92,0x52,0xd2
        DCB     0x32,0xb2,0x72,0xf2
        DCB     0x0a,0x8a,0x4a,0xca
        DCB     0x2a,0xaa,0x6a,0xea
        DCB     0x1a,0x9a,0x5a,0xda
        DCB     0x3a,0xba,0x7a,0xfa
        DCB     0x06,0x86,0x46,0xc6
        DCB     0x26,0xa6,0x66,0xe6
```

```
        DCB     0x16,0x96,0x56,0xd6
        DCB     0x36,0xb6,0x76,0xf6
        DCB     0x0e,0x8e,0x4e,0xce
        DCB     0x2e,0xae,0x6e,0xee
        DCB     0x1e,0x9e,0x5e,0xde
        DCB     0x3e,0xbe,0x7e,0xfe
        DCB     0x01,0x81,0x41,0xc1
        DCB     0x21,0xa1,0x61,0xe1
        DCB     0x11,0x91,0x51,0xd1
        DCB     0x31,0xb1,0x71,0xf1
        DCB     0x09,0x89,0x49,0xc9
        DCB     0x29,0xa9,0x69,0xe9
        DCB     0x19,0x99,0x59,0xd9
        DCB     0x39,0xb9,0x79,0xf9
        DCB     0x05,0x85,0x45,0xc5
        DCB     0x25,0xa5,0x65,0xe5
        DCB     0x15,0x95,0x55,0xd5
        DCB     0x35,0xb5,0x75,0xf5
        DCB     0x0d,0x8d,0x4d,0xcd
        DCB     0x2d,0xad,0x6d,0xed
        DCB     0x1d,0x9d,0x5d,0xdd
        DCB     0x3d,0xbd,0x7d,0xfd
        DCB     0x03,0x83,0x43,0xc3
        DCB     0x23,0xa3,0x63,0xe3
        DCB     0x13,0x93,0x53,0xd3
        DCB     0x33,0xb3,0x73,0xf3
        DCB     0x0b,0x8b,0x4b,0xcb
        DCB     0x2b,0xab,0x6b,0xeb
        DCB     0x1b,0x9b,0x5b,0xdb
        DCB     0x3b,0xbb,0x7b,0xfb
        DCB     0x07,0x87,0x47,0xc7
        DCB     0x27,0xa7,0x67,0xe7
        DCB     0x17,0x97,0x57,0xd7
        DCB     0x37,0xb7,0x77,0xf7
        DCB     0x0f,0x8f,0x4f,0xcf
        DCB     0x2f,0xaf,0x6f,0xef
        DCB     0x1f,0x9f,0x5f,0xdf
        DCB     0x3f,0xbf,0x7f,0xff
```

73.7.3 Keil (thumb) + -O3

```
f2 PROC
```

```
        LDR     r1,|L0.48|
        LSLS    r2,r0,#24
        LSRS    r2,r2,#24
        LDRB    r2,[r1,r2]
        LSLS    r2,r2,#8
        LSRS    r0,r0,#8
        LDRB    r0,[r1,r0]
        ORRS    r0,r0,r2
        BX      lr
        ENDP

f3 PROC
        MOVS    r3,r0
        LSLS    r0,r0,#16
        PUSH    {r4,lr}
        LSRS    r0,r0,#16
        BL      f2
        LSLS    r4,r0,#16
        LSRS    r0,r3,#16
        BL      f2
        ORRS    r0,r0,r4
        POP     {r4,pc}
        ENDP

|L0.48|
        DCB     0x00,0x80,0x40,0xc0
        DCB     0x20,0xa0,0x60,0xe0
        DCB     0x10,0x90,0x50,0xd0
        DCB     0x30,0xb0,0x70,0xf0
        DCB     0x08,0x88,0x48,0xc8
        DCB     0x28,0xa8,0x68,0xe8
        DCB     0x18,0x98,0x58,0xd8
        DCB     0x38,0xb8,0x78,0xf8
        DCB     0x04,0x84,0x44,0xc4
        DCB     0x24,0xa4,0x64,0xe4
        DCB     0x14,0x94,0x54,0xd4
        DCB     0x34,0xb4,0x74,0xf4
        DCB     0x0c,0x8c,0x4c,0xcc
        DCB     0x2c,0xac,0x6c,0xec
        DCB     0x1c,0x9c,0x5c,0xdc
        DCB     0x3c,0xbc,0x7c,0xfc
        DCB     0x02,0x82,0x42,0xc2
        DCB     0x22,0xa2,0x62,0xe2
```

```
        DCB     0x12,0x92,0x52,0xd2
        DCB     0x32,0xb2,0x72,0xf2
        DCB     0x0a,0x8a,0x4a,0xca
        DCB     0x2a,0xaa,0x6a,0xea
        DCB     0x1a,0x9a,0x5a,0xda
        DCB     0x3a,0xba,0x7a,0xfa
        DCB     0x06,0x86,0x46,0xc6
        DCB     0x26,0xa6,0x66,0xe6
        DCB     0x16,0x96,0x56,0xd6
        DCB     0x36,0xb6,0x76,0xf6
        DCB     0x0e,0x8e,0x4e,0xce
        DCB     0x2e,0xae,0x6e,0xee
        DCB     0x1e,0x9e,0x5e,0xde
        DCB     0x3e,0xbe,0x7e,0xfe
        DCB     0x01,0x81,0x41,0xc1
        DCB     0x21,0xa1,0x61,0xe1
        DCB     0x11,0x91,0x51,0xd1
        DCB     0x31,0xb1,0x71,0xf1
        DCB     0x09,0x89,0x49,0xc9
        DCB     0x29,0xa9,0x69,0xe9
        DCB     0x19,0x99,0x59,0xd9
        DCB     0x39,0xb9,0x79,0xf9
        DCB     0x05,0x85,0x45,0xc5
        DCB     0x25,0xa5,0x65,0xe5
        DCB     0x15,0x95,0x55,0xd5
        DCB     0x35,0xb5,0x75,0xf5
        DCB     0x0d,0x8d,0x4d,0xcd
        DCB     0x2d,0xad,0x6d,0xed
        DCB     0x1d,0x9d,0x5d,0xdd
        DCB     0x3d,0xbd,0x7d,0xfd
        DCB     0x03,0x83,0x43,0xc3
        DCB     0x23,0xa3,0x63,0xe3
        DCB     0x13,0x93,0x53,0xd3
        DCB     0x33,0xb3,0x73,0xf3
        DCB     0x0b,0x8b,0x4b,0xcb
        DCB     0x2b,0xab,0x6b,0xeb
        DCB     0x1b,0x9b,0x5b,0xdb
        DCB     0x3b,0xbb,0x7b,0xfb
        DCB     0x07,0x87,0x47,0xc7
        DCB     0x27,0xa7,0x67,0xe7
        DCB     0x17,0x97,0x57,0xd7
        DCB     0x37,0xb7,0x77,0xf7
        DCB     0x0f,0x8f,0x4f,0xcf
```

```
        DCB     0x2f,0xaf,0x6f,0xef
        DCB     0x1f,0x9f,0x5f,0xdf
        DCB     0x3f,0xbf,0x7f,0xff
```

73.8 Exercise 2.8

73.8.1 MSVC 2010 + /O1

(/O1: minimize space).

```
_a$ = 8         ; size = 4
_b$ = 12        ; size = 4
_c$ = 16        ; size = 4
?s@@YAXPAN00@Z PROC     ; s, COMDAT
    mov    eax, DWORD PTR _b$[esp-4]
    mov    ecx, DWORD PTR _a$[esp-4]
    mov    edx, DWORD PTR _c$[esp-4]
    push   esi
    push   edi
    sub    ecx, eax
    sub    edx, eax
    mov    edi, 200     ; 000000c8H
$LL6@s:
    push   100          ; 00000064H
    pop    esi
$LL3@s:
    fld    QWORD PTR [ecx+eax]
    fadd   QWORD PTR [eax]
    fstp   QWORD PTR [edx+eax]
    add    eax, 8
    dec    esi
    jne    SHORT $LL3@s
    dec    edi
    jne    SHORT $LL6@s
    pop    edi
    pop    esi
    ret    0
?s@@YAXPAN00@Z ENDP     ; s
```

73.8.2 Keil (ARM) + -O3

```
        PUSH    {r4-r12,lr}
        MOV     r9,r2
        MOV     r10,r1
        MOV     r11,r0
        MOV     r5,#0
|L0.20|
        ADD     r0,r5,r5,LSL #3
        ADD     r0,r0,r5,LSL #4
        MOV     r4,#0
        ADD     r8,r10,r0,LSL #5
        ADD     r7,r11,r0,LSL #5
        ADD     r6,r9,r0,LSL #5
|L0.44|
        ADD     r0,r8,r4,LSL #3
        LDM     r0,{r2,r3}
        ADD     r1,r7,r4,LSL #3
        LDM     r1,{r0,r1}
        BL      __aeabi_dadd
        ADD     r2,r6,r4,LSL #3
        ADD     r4,r4,#1
        STM     r2,{r0,r1}
        CMP     r4,#0x64
        BLT     |L0.44|
        ADD     r5,r5,#1
        CMP     r5,#0xc8
        BLT     |L0.20|
        POP     {r4-r12,pc}
```

73.8.3 Keil (thumb) + -O3

```
        PUSH    {r0-r2,r4-r7,lr}
        MOVS    r4,#0
        SUB     sp,sp,#8
|L0.6|
        MOVS    r1,#0x19
        MOVS    r0,r4
        LSLS    r1,r1,#5
        MULS    r0,r1,r0
        LDR     r2,[sp,#8]
        LDR     r1,[sp,#0xc]
        ADDS    r2,r0,r2
        STR     r2,[sp,#0]
```

```
        LDR      r2,[sp,#0x10]
        MOVS     r5,#0
        ADDS     r7,r0,r2
        ADDS     r0,r0,r1
        STR      r0,[sp,#4]
|L0.32|
        LSLS     r6,r5,#3
        ADDS     r0,r0,r6
        LDM      r0!,{r2,r3}
        LDR      r0,[sp,#0]
        ADDS     r1,r0,r6
        LDM      r1,{r0,r1}
        BL       __aeabi_dadd
        ADDS     r2,r7,r6
        ADDS     r5,r5,#1
        STM      r2!,{r0,r1}
        CMP      r5,#0x64
        BGE      |L0.62|
        LDR      r0,[sp,#4]
        B        |L0.32|
|L0.62|
        ADDS     r4,r4,#1
        CMP      r4,#0xc8
        BLT      |L0.6|
        ADD      sp,sp,#0x14
        POP      {r4-r7,pc}
```

73.9 Exercise 2.9

73.9.1 MSVC 2010 + /O1

(/O1: minimize space).

```
tv315 = -8              ; size = 4
tv291 = -4              ; size = 4
_a$ = 8                 ; size = 4
_b$ = 12                ; size = 4
_c$ = 16                ; size = 4
?m@@YAXPAN00@Z PROC     ; m, COMDAT
    push   ebp
    mov    ebp, esp
    push   ecx
    push   ecx
```

```
    mov     edx, DWORD PTR _a$[ebp]
    push    ebx
    mov     ebx, DWORD PTR _c$[ebp]
    push    esi
    mov     esi, DWORD PTR _b$[ebp]
    sub     edx, esi
    push    edi
    sub     esi, ebx
    mov     DWORD PTR tv315[ebp], 100  ; 00000064H
$LL9@m:
    mov     eax, ebx
    mov     DWORD PTR tv291[ebp], 300  ; 0000012cH
$LL6@m:
    fldz
    lea     ecx, DWORD PTR [esi+eax]
    fstp    QWORD PTR [eax]
    mov     edi, 200                   ; 000000c8H
$LL3@m:
    dec     edi
    fld     QWORD PTR [ecx+edx]
    fmul    QWORD PTR [ecx]
    fadd    QWORD PTR [eax]
    fstp    QWORD PTR [eax]
    jne     HORT $LL3@m
    add     eax, 8
    dec     DWORD PTR tv291[ebp]
    jne     SHORT $LL6@m
    add     ebx, 800                   ; 00000320H
    dec     DWORD PTR tv315[ebp]
    jne     SHORT $LL9@m
    pop     edi
    pop     esi
    pop     ebx
    leave
    ret     0
?m@@YAXPAN00@Z ENDP                    ; m
```

73.9.2 Keil (ARM) + -O3

```
        PUSH    {r0-r2,r4-r11,lr}
        SUB     sp,sp,#8
        MOV     r5,#0
```

```
|L0.12|
        LDR      r1,[sp,#0xc]
        ADD      r0,r5,r5,LSL #3
        ADD      r0,r0,r5,LSL #4
        ADD      r1,r1,r0,LSL #5
        STR      r1,[sp,#0]
        LDR      r1,[sp,#8]
        MOV      r4,#0
        ADD      r11,r1,r0,LSL #5
        LDR      r1,[sp,#0x10]
        ADD      r10,r1,r0,LSL #5
|L0.52|
        MOV      r0,#0
        MOV      r1,r0
        ADD      r7,r10,r4,LSL #3
        STM      r7,{r0,r1}
        MOV      r6,r0
        LDR      r0,[sp,#0]
        ADD      r8,r11,r4,LSL #3
        ADD      r9,r0,r4,LSL #3
|L0.84|
        LDM      r9,{r2,r3}
        LDM      r8,{r0,r1}
        BL       __aeabi_dmul
        LDM      r7,{r2,r3}
        BL       __aeabi_dadd
        ADD      r6,r6,#1
        STM      r7,{r0,r1}
        CMP      r6,#0xc8
        BLT      |L0.84|
        ADD      r4,r4,#1
        CMP      r4,#0x12c
        BLT      |L0.52|
        ADD      r5,r5,#1
        CMP      r5,#0x64
        BLT      |L0.12|
        ADD      sp,sp,#0x14
        POP      {r4-r11,pc}
```

73.9.3 Keil (thumb) + -O3

```
        PUSH     {r0-r2,r4-r7,lr}
```

```
        MOVS    r0,#0
        SUB     sp,sp,#0x10
        STR     r0,[sp,#0]
|L0.8|
        MOVS    r1,#0x19
        LSLS    r1,r1,#5
        MULS    r0,r1,r0
        LDR     r2,[sp,#0x10]
        LDR     r1,[sp,#0x14]
        ADDS    r2,r0,r2
        STR     r2,[sp,#4]
        LDR     r2,[sp,#0x18]
        MOVS    r5,#0
        ADDS    r7,r0,r2
        ADDS    r0,r0,r1
        STR     r0,[sp,#8]
|L0.32|
        LSLS    r4,r5,#3
        MOVS    r0,#0
        ADDS    r2,r7,r4
        STR     r0,[r2,#0]
        MOVS    r6,r0
        STR     r0,[r2,#4]
|L0.44|
        LDR     r0,[sp,#8]
        ADDS    r0,r0,r4
        LDM     r0!,{r2,r3}
        LDR     r0,[sp,#4]
        ADDS    r1,r0,r4
        LDM     r1,{r0,r1}
        BL      __aeabi_dmul
        ADDS    r3,r7,r4
        LDM     r3,{r2,r3}
        BL      __aeabi_dadd
        ADDS    r2,r7,r4
        ADDS    r6,r6,#1
        STM     r2!,{r0,r1}
        CMP     r6,#0xc8
        BLT     |L0.44|
        MOVS    r0,#0xff
        ADDS    r5,r5,#1
        ADDS    r0,r0,#0x2d
        CMP     r5,r0
```

```
        BLT     |L0.32|
        LDR     r0,[sp,#0]
        ADDS    r0,r0,#1
        CMP     r0,#0x64
        STR     r0,[sp,#0]
        BLT     |L0.8|
        ADD     sp,sp,#0x1c
        POP     {r4-r7,pc}
```

73.10 Exercise 2.10

If to compile this piece of code and run, a number will be printed. Where it came from? Where it came from if to compile it in MSVC with optimization (`/Ox`)?

```
#include <stdio.h>

int main()
{
        printf ("%d\n");

        return 0;
};
```

73.11 Exercise 2.11

As a practical joke, "fool" your Windows Task Manager to show much more CPUs/CPU cores than your machine actually has:

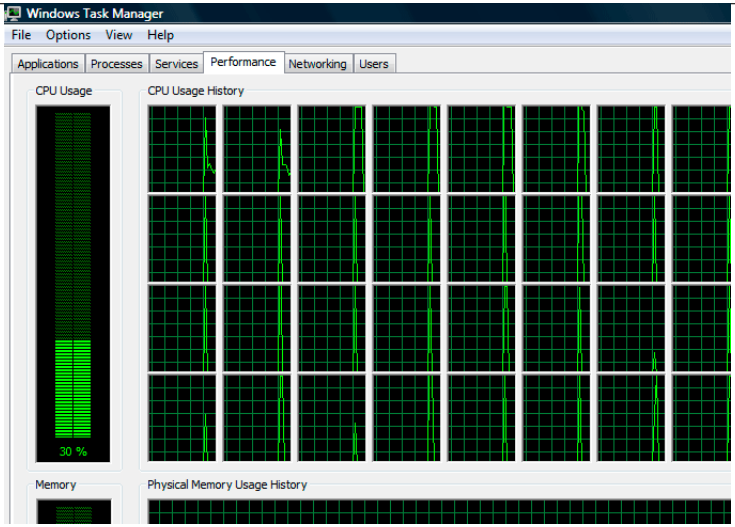


Figure 73.1: Fooled Windows Task Manager

73.12 Exercise 2.12

This is a well-known algorithm. How it's called?

73.12.1 MSVC 2012 x64 + /Ox

```
s$ = 8
f       PROC
        cmp     BYTE PTR [rcx], 0
        mov     r9, rcx
        je      SHORT $LN13@f
        npad    8
$LL5@f:
        movzx   edx, BYTE PTR [rcx]
        lea     eax, DWORD PTR [rdx-97]
        cmp     al, 25
        ja      SHORT $LN3@f
        movsx   r8d, dl
        mov     eax, 1321528399                         ; 4↙
    ↳ ec4ec4fH
        sub     r8d, 84                                 ; ↙
    ↳ 00000054H
        imul    r8d
```

```
        sar     edx, 3
        mov     eax, edx
        shr     eax, 31
        add     edx, eax
        imul    edx, 26
        sub     r8d, edx
        add     r8b, 97                                 ; ↙
    ↳ 00000061H
        jmp     SHORT $LN14@f
$LN3@f:
        lea     eax, DWORD PTR [rdx-65]
        cmp     al, 25
        ja      SHORT $LN1@f
        movsx   r8d, dl
        mov     eax, 1321528399                         ; 4↙
    ↳ ec4ec4fH
        sub     r8d, 52                                 ; ↙
    ↳ 00000034H
        imul    r8d
        sar     edx, 3
        mov     eax, edx
        shr     eax, 31
        add     edx, eax
        imul    edx, 26
        sub     r8d, edx
        add     r8b, 65                                 ; ↙
    ↳ 00000041H
$LN14@f:
        mov     BYTE PTR [rcx], r8b
$LN1@f:
        inc     rcx
        cmp     BYTE PTR [rcx], 0
        jne     SHORT $LL5@f
$LN13@f:
        mov     rax, r9
        ret     0
f       ENDP
```

73.12.2 Keil (ARM)

```
f PROC
        PUSH    {r4-r6,lr}
```

```
        MOV     r4,r0
        MOV     r5,r0
        B       |L0.84|
|L0.16|
        SUB     r1,r0,#0x61
        CMP     r1,#0x19
        BHI     |L0.48|
        SUB     r0,r0,#0x54
        MOV     r1,#0x1a
        BL      __aeabi_idivmod
        ADD     r0,r1,#0x61
        B       |L0.76|
|L0.48|
        SUB     r1,r0,#0x41
        CMP     r1,#0x19
        BHI     |L0.80|
        SUB     r0,r0,#0x34
        MOV     r1,#0x1a
        BL      __aeabi_idivmod
        ADD     r0,r1,#0x41
|L0.76|
        STRB    r0,[r4,#0]
|L0.80|
        ADD     r4,r4,#1
|L0.84|
        LDRB    r0,[r4,#0]
        CMP     r0,#0
        MOVEQ   r0,r5
        BNE     |L0.16|
        POP     {r4-r6,pc}
        ENDP
```

73.12.3 Keil (thumb)

```
f PROC
        PUSH    {r4-r6,lr}
        MOVS    r4,r0
        MOVS    r5,r0
        B       |L0.50|
|L0.8|
        MOVS    r1,r0
        SUBS    r1,r1,#0x61
```

```
        CMP     r1,#0x19
        BHI     |L0.28|
        SUBS    r0,r0,#0x54
        MOVS    r1,#0x1a
        BL      __aeabi_idivmod
        ADDS    r1,r1,#0x61
        B       |L0.46|
|L0.28|
        MOVS    r1,r0
        SUBS    r1,r1,#0x41
        CMP     r1,#0x19
        BHI     |L0.48|
        SUBS    r0,r0,#0x34
        MOVS    r1,#0x1a
        BL      __aeabi_idivmod
        ADDS    r1,r1,#0x41
|L0.46|
        STRB    r1,[r4,#0]
|L0.48|
        ADDS    r4,r4,#1
|L0.50|
        LDRB    r0,[r4,#0]
        CMP     r0,#0
        BNE     |L0.8|
        MOVS    r0,r5
        POP     {r4-r6,pc}
        ENDP
```

73.13 Exercise 2.13

This is a well-known cryptoalgorithm of the past. How it's called?

73.13.1 MSVC 2012 + /Ox

```
_in$ = 8                                                ; size =↙
    ↳ 2
_f      PROC
        movzx   ecx, WORD PTR _in$[esp-4]
        lea     eax, DWORD PTR [ecx*4]
        xor     eax, ecx
        add     eax, eax
```

```
        xor     eax, ecx
        shl     eax, 2
        xor     eax, ecx
        and     eax, 32                                 ; ↙
    ↳ 00000020H
        shl     eax, 10                                 ; ↙
    ↳ 0000000aH
        shr     ecx, 1
        or      eax, ecx
        ret     0
_f      ENDP
```

73.13.2 Keil (ARM)

```
f PROC
        EOR     r1,r0,r0,LSR #2
        EOR     r1,r1,r0,LSR #3
        EOR     r1,r1,r0,LSR #5
        AND     r1,r1,#1
        LSR     r0,r0,#1
        ORR     r0,r0,r1,LSL #15
        BX      lr
        ENDP
```

73.13.3 Keil (thumb)

```
f PROC
        LSRS    r1,r0,#2
        EORS    r1,r1,r0
        LSRS    r2,r0,#3
        EORS    r1,r1,r2
        LSRS    r2,r0,#5
        EORS    r1,r1,r2
        LSLS    r1,r1,#31
        LSRS    r0,r0,#1
        LSRS    r1,r1,#16
        ORRS    r0,r0,r1
        BX      lr
        ENDP
```

73.14 Exercise 2.14

Another well-known algorithm. The function takes two variables and returning one.

73.14.1 MSVC 2012

```
_rt$1 = -4                                              ; size =↙
    ↳  4
_rt$2 = 8                                               ; size =↙
    ↳  4
_x$ = 8                                                 ; size =↙
    ↳  4
_y$ = 12                                                ; size =↙
    ↳  4
?f@@YAIII@Z PROC                                        ; f
        push    ecx
        push    esi
        mov     esi, DWORD PTR _x$[esp+4]
        test    esi, esi
        jne     SHORT $LN7@f
        mov     eax, DWORD PTR _y$[esp+4]
        pop     esi
        pop     ecx
        ret     0
$LN7@f:
        mov     edx, DWORD PTR _y$[esp+4]
        mov     eax, esi
        test    edx, edx
        je      SHORT $LN8@f
        or      eax, edx
        push    edi
        bsf     edi, eax
        bsf     eax, esi
        mov     ecx, eax
        mov     DWORD PTR _rt$1[esp+12], eax
        bsf     eax, edx
        shr     esi, cl
        mov     ecx, eax
        shr     edx, cl
        mov     DWORD PTR _rt$2[esp+8], eax
        cmp     esi, edx
        je      SHORT $LN22@f
$LN23@f:
```

```
        jbe     SHORT $LN2@f
        xor     esi, edx
        xor     edx, esi
        xor     esi, edx
$LN2@f:
        cmp     esi, 1
        je      SHORT $LN22@f
        sub     edx, esi
        bsf     eax, edx
        mov     ecx, eax
        shr     edx, cl
        mov     DWORD PTR _rt$2[esp+8], eax
        cmp     esi, edx
        jne     SHORT $LN23@f
$LN22@f:
        mov     ecx, edi
        shl     esi, cl
        pop     edi
        mov     eax, esi
$LN8@f:
        pop     esi
        pop     ecx
        ret     0
?f@@YAIII@Z ENDP
```

73.14.2 Keil (ARM mode)

```
||f1|| PROC
        CMP      r0,#0
        RSB      r1,r0,#0
        AND      r0,r0,r1
        CLZ      r0,r0
        RSBNE    r0,r0,#0x1f
        BX       lr
        ENDP

f PROC
        MOVS     r2,r0
        MOV      r3,r1
        MOVEQ    r0,r1
        CMPNE    r3,#0
        PUSH     {lr}
```

```
        POPEQ   {pc}
        ORR     r0,r2,r3
        BL      ||f1||
        MOV     r12,r0
        MOV     r0,r2
        BL      ||f1||
        LSR     r2,r2,r0
|L0.196|
        MOV     r0,r3
        BL      ||f1||
        LSR     r0,r3,r0
        CMP     r2,r0
        EORHI   r1,r2,r0
        EORHI   r0,r0,r1
        EORHI   r2,r1,r0
        BEQ     |L0.240|
        CMP     r2,#1
        SUBNE   r3,r0,r2
        BNE     |L0.196|
|L0.240|
        LSL     r0,r2,r12
        POP     {pc}
        ENDP
```

73.14.3 GCC 4.6.3 for Raspberry Pi (ARM mode)

```
f:
        subs    r3, r0, #0
        beq     .L162
        cmp     r1, #0
        moveq   r1, r3
        beq     .L162
        orr     r2, r1, r3
        rsb     ip, r2, #0
        and     ip, ip, r2
        cmp     r2, #0
        rsb     r2, r3, #0
        and     r2, r2, r3
        clz     r2, r2
        rsb     r2, r2, #31
        clz     ip, ip
        rsbne   ip, ip, #31
```

```
        mov     r3, r3, lsr r2
        b       .L169
.L171:
        eorhi   r1, r1, r2
        eorhi   r3, r1, r2
        cmp     r3, #1
        rsb     r1, r3, r1
        beq     .L167
.L169:
        rsb     r0, r1, #0
        and     r0, r0, r1
        cmp     r1, #0
        clz     r0, r0
        mov     r2, r0
        rsbne   r2, r0, #31
        mov     r1, r1, lsr r2
        cmp     r3, r1
        eor     r2, r1, r3
        bne     .L171
.L167:
        mov     r1, r3, asl ip
.L162:
        mov     r0, r1
        bx      lr
```

73.15 Exercise 2.15

Well-known algorithm again. What it does?

Take also notice that the code for x86 uses FPU, but SIMD-instructions are used instead in x64 code. That's OK: 24.

73.15.1 MSVC 2012 x64 /Ox

```
__real@412e848000000000 DQ 0412e848000000000r   ; 1e+006
__real@4010000000000000 DQ 0401000000000000r   ; 4
__real@4008000000000000 DQ 0400800000000000r   ; 3
__real@3f800000 DD 03f800000r                   ; 1

tmp$1 = 8
tmp$2 = 8
f       PROC
```

```
        movsdx  xmm3, QWORD PTR __real@4008000000000000
        movss   xmm4, DWORD PTR __real@3f800000
        mov     edx, DWORD PTR ?RNG_state@?1??get_rand@@9@9
        xor     ecx, ecx
        mov     r8d, 200000                             ; 00030↙
    ↳ d40H
        npad    2
$LL4@f:
        imul    edx, 1664525                            ; ↙
    ↳ 0019660dH
        add     edx, 1013904223                         ; 3↙
    ↳ c6ef35fH
        mov     eax, edx
        and     eax, 8388607                            ; 007↙
    ↳ fffffH
        imul    edx, 1664525                            ; ↙
    ↳ 0019660dH
        bts     eax, 30
        add     edx, 1013904223                         ; 3↙
    ↳ c6ef35fH
        mov     DWORD PTR tmp$2[rsp], eax
        mov     eax, edx
        and     eax, 8388607                            ; 007↙
    ↳ fffffH
        bts     eax, 30
        movss   xmm0, DWORD PTR tmp$2[rsp]
        mov     DWORD PTR tmp$1[rsp], eax
        cvtps2pd xmm0, xmm0
        subsd   xmm0, xmm3
        cvtpd2ps xmm2, xmm0
        movss   xmm0, DWORD PTR tmp$1[rsp]
        cvtps2pd xmm0, xmm0
        mulss   xmm2, xmm2
        subsd   xmm0, xmm3
        cvtpd2ps xmm1, xmm0
        mulss   xmm1, xmm1
        addss   xmm1, xmm2
        comiss  xmm4, xmm1
        jbe     SHORT $LN3@f
        inc     ecx
$LN3@f:
        imul    edx, 1664525                            ; ↙
    ↳ 0019660dH
```

```
        add     edx, 1013904223                         ; 3↙
    ↳ c6ef35fH
        mov     eax, edx
        and     eax, 8388607                            ; 007↙
    ↳ fffffH
        imul    edx, 1664525                            ; ↙
    ↳ 0019660dH
        bts     eax, 30
        add     edx, 1013904223                         ; 3↙
    ↳ c6ef35fH
        mov     DWORD PTR tmp$2[rsp], eax
        mov     eax, edx
        and     eax, 8388607                            ; 007↙
    ↳ fffffH
        bts     eax, 30
        movss   xmm0, DWORD PTR tmp$2[rsp]
        mov     DWORD PTR tmp$1[rsp], eax
        cvtps2pd xmm0, xmm0
        subsd   xmm0, xmm3
        cvtpd2ps xmm2, xmm0
        movss   xmm0, DWORD PTR tmp$1[rsp]
        cvtps2pd xmm0, xmm0
        mulss   xmm2, xmm2
        subsd   xmm0, xmm3
        cvtpd2ps xmm1, xmm0
        mulss   xmm1, xmm1
        addss   xmm1, xmm2
        comiss  xmm4, xmm1
        jbe     SHORT $LN15@f
        inc     ecx
$LN15@f:
        imul    edx, 1664525                            ; ↙
    ↳ 0019660dH
        add     edx, 1013904223                         ; 3↙
    ↳ c6ef35fH
        mov     eax, edx
        and     eax, 8388607                            ; 007↙
    ↳ fffffH
        imul    edx, 1664525                            ; ↙
    ↳ 0019660dH
        bts     eax, 30
        add     edx, 1013904223                         ; 3↙
    ↳ c6ef35fH
```

```
        mov     DWORD PTR tmp$2[rsp], eax
        mov     eax, edx
        and     eax, 8388607                            ; 007↙
    ↳ fffffH
        bts     eax, 30
        movss   xmm0, DWORD PTR tmp$2[rsp]
        mov     DWORD PTR tmp$1[rsp], eax
        cvtps2pd xmm0, xmm0
        subsd   xmm0, xmm3
        cvtpd2ps xmm2, xmm0
        movss   xmm0, DWORD PTR tmp$1[rsp]
        cvtps2pd xmm0, xmm0
        mulss   xmm2, xmm2
        subsd   xmm0, xmm3
        cvtpd2ps xmm1, xmm0
        mulss   xmm1, xmm1
        addss   xmm1, xmm2
        comiss  xmm4, xmm1
        jbe     SHORT $LN16@f
        inc     ecx
$LN16@f:
        imul    edx, 1664525                            ; ↙
    ↳ 0019660dH
        add     edx, 1013904223                         ; 3↙
    ↳ c6ef35fH
        mov     eax, edx
        and     eax, 8388607                            ; 007↙
    ↳ fffffH
        imul    edx, 1664525                            ; ↙
    ↳ 0019660dH
        bts     eax, 30
        add     edx, 1013904223                         ; 3↙
    ↳ c6ef35fH
        mov     DWORD PTR tmp$2[rsp], eax
        mov     eax, edx
        and     eax, 8388607                            ; 007↙
    ↳ fffffH
        bts     eax, 30
        movss   xmm0, DWORD PTR tmp$2[rsp]
        mov     DWORD PTR tmp$1[rsp], eax
        cvtps2pd xmm0, xmm0
        subsd   xmm0, xmm3
        cvtpd2ps xmm2, xmm0
```

```
        movss   xmm0, DWORD PTR tmp$1[rsp]
        cvtps2pd xmm0, xmm0
        mulss   xmm2, xmm2
        subsd   xmm0, xmm3
        cvtpd2ps xmm1, xmm0
        mulss   xmm1, xmm1
        addss   xmm1, xmm2
        comiss  xmm4, xmm1
        jbe     SHORT $LN17@f
        inc     ecx
$LN17@f:
        imul    edx, 1664525                            ; ↙
    ↳ 0019660dH
        add     edx, 1013904223                         ; 3↙
    ↳ c6ef35fH
        mov     eax, edx
        and     eax, 8388607                            ; 007↙
    ↳ fffffH
        imul    edx, 1664525                            ; ↙
    ↳ 0019660dH
        bts     eax, 30
        add     edx, 1013904223                         ; 3↙
    ↳ c6ef35fH
        mov     DWORD PTR tmp$2[rsp], eax
        mov     eax, edx
        and     eax, 8388607                            ; 007↙
    ↳ fffffH
        bts     eax, 30
        movss   xmm0, DWORD PTR tmp$2[rsp]
        mov     DWORD PTR tmp$1[rsp], eax
        cvtps2pd xmm0, xmm0
        subsd   xmm0, xmm3
        cvtpd2ps xmm2, xmm0
        movss   xmm0, DWORD PTR tmp$1[rsp]
        cvtps2pd xmm0, xmm0
        mulss   xmm2, xmm2
        subsd   xmm0, xmm3
        cvtpd2ps xmm1, xmm0
        mulss   xmm1, xmm1
        addss   xmm1, xmm2
        comiss  xmm4, xmm1
        jbe     SHORT $LN18@f
        inc     ecx
```

```
$LN18@f:
        dec     r8
        jne     $LL4@f
        movd    xmm0, ecx
        mov     DWORD PTR ?RNG_state@?1??get_rand@@9@9, edx
        cvtdq2ps xmm0, xmm0
        cvtps2pd xmm1, xmm0
        mulsd   xmm1, QWORD PTR __real@4010000000000000
        divsd   xmm1, QWORD PTR __real@412e848000000000
        cvtpd2ps xmm0, xmm1
        ret     0
f       ENDP
```

73.15.2 GCC 4.4.6 -O3 x64

```
f1:
        mov     eax, DWORD PTR v1.2084[rip]
        imul    eax, eax, 1664525
        add     eax, 1013904223
        mov     DWORD PTR v1.2084[rip], eax
        and     eax, 8388607
        or      eax, 1073741824
        mov     DWORD PTR [rsp-4], eax
        movss   xmm0, DWORD PTR [rsp-4]
        subss   xmm0, DWORD PTR .LC0[rip]
        ret
f:
        push    rbp
        xor     ebp, ebp
        push    rbx
        xor     ebx, ebx
        sub     rsp, 16
.L6:
        xor     eax, eax
        call    f1
        xor     eax, eax
        movss   DWORD PTR [rsp], xmm0
        call    f1
        movss   xmm1, DWORD PTR [rsp]
        mulss   xmm0, xmm0
        mulss   xmm1, xmm1
        lea     eax, [rbx+1]
```

```
        addss   xmm1, xmm0
        movss   xmm0, DWORD PTR .LC1[rip]
        ucomiss xmm0, xmm1
        cmova   ebx, eax
        add     ebp, 1
        cmp     ebp, 1000000
        jne     .L6
        cvtsi2ss        xmm0, ebx
        unpcklps        xmm0, xmm0
        cvtps2pd        xmm0, xmm0
        mulsd   xmm0, QWORD PTR .LC2[rip]
        divsd   xmm0, QWORD PTR .LC3[rip]
        add     rsp, 16
        pop     rbx
        pop     rbp
        unpcklpd        xmm0, xmm0
        cvtpd2ps        xmm0, xmm0
        ret
v1.2084:
        .long   305419896
.LC0:
        .long   1077936128
.LC1:
        .long   1065353216
.LC2:
        .long   0
        .long   1074790400
.LC3:
        .long   0
        .long   1093567616
```

73.15.3 GCC 4.8.1 -O3 x86

```
f1:
        sub     esp, 4
        imul    eax, DWORD PTR v1.2023, 1664525
        add     eax, 1013904223
        mov     DWORD PTR v1.2023, eax
        and     eax, 8388607
        or      eax, 1073741824
        mov     DWORD PTR [esp], eax
        fld     DWORD PTR [esp]
```

```
        fsub    DWORD PTR .LC0
        add     esp, 4
        ret
f:
        push    esi
        mov     esi, 1000000
        push    ebx
        xor     ebx, ebx
        sub     esp, 16
.L7:
        call    f1
        fstp    DWORD PTR [esp]
        call    f1
        lea     eax, [ebx+1]
        fld     DWORD PTR [esp]
        fmul    st, st(0)
        fxch    st(1)
        fmul    st, st(0)
        faddp   st(1), st
        fld1
        fucomip st, st(1)
        fstp    st(0)
        cmova   ebx, eax
        sub     esi, 1
        jne     .L7
        mov     DWORD PTR [esp+4], ebx
        fild    DWORD PTR [esp+4]
        fmul    DWORD PTR .LC3
        fdiv    DWORD PTR .LC4
        fstp    DWORD PTR [esp+8]
        fld     DWORD PTR [esp+8]
        add     esp, 16
        pop     ebx
        pop     esi
        ret

v1.2023:
        .long   305419896
.LC0:
        .long   1077936128
.LC3:
        .long   1082130432
.LC4:
```

```
        .long   1232348160
```

73.15.4 Keil (ARM mode): Cortex-R4F CPU as target

```
f1      PROC
        LDR      r1,|L0.184|
        LDR      r0,[r1,#0]  ; v1
        LDR      r2,|L0.188|
        VMOV.F32 s1,#3.00000000
        MUL      r0,r0,r2
        LDR      r2,|L0.192|
        ADD      r0,r0,r2
        STR      r0,[r1,#0]  ; v1
        BFC      r0,#23,#9
        ORR      r0,r0,#0x40000000
        VMOV     s0,r0
        VSUB.F32 s0,s0,s1
        BX       lr
        ENDP

f       PROC
        PUSH     {r4,r5,lr}
        MOV      r4,#0
        LDR      r5,|L0.196|
        MOV      r3,r4
|L0.68|
        BL       f1
        VMOV.F32 s2,s0
        BL       f1
        VMOV.F32 s1,s2
        ADD      r3,r3,#1
        VMUL.F32 s1,s1,s1
        VMLA.F32 s1,s0,s0
        VMOV     r0,s1
        CMP      r0,#0x3f800000
        ADDLT    r4,r4,#1
        CMP      r3,r5
        BLT      |L0.68|
        VMOV     s0,r4
        VMOV.F64 d1,#4.00000000
        VCVT.F32.S32 s0,s0
        VCVT.F64.F32 d0,s0
```

```
        VMUL.F64 d0,d0,d1
        VLDR     d1,|L0.200|
        VDIV.F64 d2,d0,d1
        VCVT.F32.F64 s0,d2
        POP      {r4,r5,pc}
        ENDP

|L0.184|
        DCD      ||.data||
|L0.188|
        DCD      0x0019660d
|L0.192|
        DCD      0x3c6ef35f
|L0.196|
        DCD      0x000f4240
|L0.200|
        DCFD     0x412e848000000000 ; 1000000

        DCD      0x00000000
        AREA ||.data||, DATA, ALIGN=2
v1
        DCD      0x12345678
```

73.16 Exercise 2.16

Well-known function. What it computes? Why stack overflows if 4 and 2 are supplied at input? Are there any error?

73.16.1 MSVC 2012 x64 /Ox

```
m$ = 48
n$ = 56
f       PROC
$LN14:
        push    rbx
        sub     rsp, 32
        mov     eax, edx
        mov     ebx, ecx
        test    ecx, ecx
        je      SHORT $LN11@f
$LL5@f:
```

```
        test    eax, eax
        jne     SHORT $LN1@f
        mov     eax, 1
        jmp     SHORT $LN12@f
$LN1@f:
        lea     edx, DWORD PTR [rax-1]
        mov     ecx, ebx
        call    f
$LN12@f:
        dec     ebx
        test    ebx, ebx
        jne     SHORT $LL5@f
$LN11@f:
        inc     eax
        add     rsp, 32
        pop     rbx
        ret     0
f       ENDP
```

73.16.2 Keil (ARM) -O3

```
f PROC
        PUSH    {r4,lr}
        MOVS    r4,r0
        ADDEQ   r0,r1,#1
        POPEQ   {r4,pc}
        CMP     r1,#0
        MOVEQ   r1,#1
        SUBEQ   r0,r0,#1
        BEQ     |L0.48|
        SUB     r1,r1,#1
        BL      f
        MOV     r1,r0
        SUB     r0,r4,#1
|L0.48|
        POP     {r4,lr}
        B       f
        ENDP
```

73.16.3 Keil (thumb) -O3

```
f PROC
        PUSH     {r4,lr}
        MOVS     r4,r0
        BEQ      |L0.26|
        CMP      r1,#0
        BEQ      |L0.30|
        SUBS     r1,r1,#1
        BL       f
        MOVS     r1,r0
|L0.18|
        SUBS     r0,r4,#1
        BL       f
        POP      {r4,pc}
|L0.26|
        ADDS     r0,r1,#1
        POP      {r4,pc}
|L0.30|
        MOVS     r1,#1
        B        |L0.18|
        ENDP
```

73.17 Exercise 2.17

This program prints some information to *stdout*, each time different. What is it?

Compiled binaries:

- Linux x64: http://yurichev.com/RE-exercises/2/17/17_Linux_x64.tar
- Mac OS X: http://yurichev.com/RE-exercises/2/17/17_MacOSX_x64.tar
- Win32: http://yurichev.com/RE-exercises/2/17/17_win32.exe
- Win64: http://yurichev.com/RE-exercises/2/17/17_win64.exe

As of Windows versions, you may need to install MSVC 2012 redist.

73.18 Exercise 2.18

This program requires password. Find it.

By the way, multiple passwords may work. Try to find more.

As an additional exercise, try to change the password by patching executable file.

- Win32: http://yurichev.com/RE-exercises/2/18/password2.exe

- Linux x86: http://yurichev.com/RE-exercises/2/18/password2_Linux_x86.tar
- Mac OS X: http://yurichev.com/RE-exercises/2/18/password2_MacOSX64.tar

73.19 Exercise 2.19

The same as in exercise 2.18.

- Win32: http://yurichev.com/RE-exercises/2/19/password3.exe
- Linux x86: http://yurichev.com/RE-exercises/2/19/password3_Linux_x86.tar
- Mac OS X: http://yurichev.com/RE-exercises/2/19/password3_MacOSX64.tar

Chapter 74

Level 3

For solving level 3 tasks, you'll probably need considerable ammount of time, maybe up to one day.

74.1 Exercise 3.1

Well-known algorithm, also included in standard C library. Source code was taken from glibc 2.11.1. Compiled in GCC 4.4.1 with `-Os` option (code size optimization). Listing was done by IDA 4.9 disassembler from ELF-file generated by GCC and linker.

For those who wants use IDA while learning, here you may find .elf and .idb files, .idb can be opened with freeware IDA 4.9:

http://yurichev.com/RE-exercises/3/1/

```
f               proc near

var_150         = dword ptr -150h
var_14C         = dword ptr -14Ch
var_13C         = dword ptr -13Ch
var_138         = dword ptr -138h
var_134         = dword ptr -134h
var_130         = dword ptr -130h
var_128         = dword ptr -128h
var_124         = dword ptr -124h
var_120         = dword ptr -120h
var_11C         = dword ptr -11Ch
var_118         = dword ptr -118h
var_114         = dword ptr -114h
var_110         = dword ptr -110h
var_C           = dword ptr -0Ch
```

```
arg_0           = dword ptr  8
arg_4           = dword ptr  0Ch
arg_8           = dword ptr  10h
arg_C           = dword ptr  14h
arg_10          = dword ptr  18h

                push    ebp
                mov     ebp, esp
                push    edi
                push    esi
                push    ebx
                sub     esp, 14Ch
                mov     ebx, [ebp+arg_8]
                cmp     [ebp+arg_4], 0
                jz      loc_804877D
                cmp     [ebp+arg_4], 4
                lea     eax, ds:0[ebx*4]
                mov     [ebp+var_130], eax
                jbe     loc_804864C
                mov     eax, [ebp+arg_4]
                mov     ecx, ebx
                mov     esi, [ebp+arg_0]
                lea     edx, [ebp+var_110]
                neg     ecx
                mov     [ebp+var_118], 0
                mov     [ebp+var_114], 0
                dec     eax
                imul    eax, ebx
                add     eax, [ebp+arg_0]
                mov     [ebp+var_11C], edx
                mov     [ebp+var_134], ecx
                mov     [ebp+var_124], eax
                lea     eax, [ebp+var_118]
                mov     [ebp+var_14C], eax
                mov     [ebp+var_120], ebx

loc_8048433:                            ; CODE XREF: f+28C
                mov     eax, [ebp+var_124]
                xor     edx, edx
                push    edi
                push    [ebp+arg_10]
                sub     eax, esi
                div     [ebp+var_120]
```

```
                push    esi
                shr     eax, 1
                imul    eax, [ebp+var_120]
                lea     edx, [esi+eax]
                push    edx
                mov     [ebp+var_138], edx
                call    [ebp+arg_C]
                add     esp, 10h
                mov     edx, [ebp+var_138]
                test    eax, eax
                jns     short loc_8048482
                xor     eax, eax

loc_804846D:                            ; CODE XREF: f+CC
                mov     cl, [edx+eax]
                mov     bl, [esi+eax]
                mov     [edx+eax], bl
                mov     [esi+eax], cl
                inc     eax
                cmp     [ebp+var_120], eax
                jnz     short loc_804846D

loc_8048482:                            ; CODE XREF: f+B5
                push    ebx
                push    [ebp+arg_10]
                mov     [ebp+var_138], edx
                push    edx
                push    [ebp+var_124]
                call    [ebp+arg_C]
                mov     edx, [ebp+var_138]
                add     esp, 10h
                test    eax, eax
                jns     short loc_80484F6
                mov     ecx, [ebp+var_124]
                xor     eax, eax

loc_80484AB:                            ; CODE XREF: f+10D
                movzx   edi, byte ptr [edx+eax]
                mov     bl, [ecx+eax]
                mov     [edx+eax], bl
                mov     ebx, edi
                mov     [ecx+eax], bl
                inc     eax
```

```
                cmp     [ebp+var_120], eax
                jnz     short loc_80484AB
                push    ecx
                push    [ebp+arg_10]
                mov     [ebp+var_138], edx
                push    esi
                push    edx
                call    [ebp+arg_C]
                add     esp, 10h
                mov     edx, [ebp+var_138]
                test    eax, eax
                jns     short loc_80484F6
                xor     eax, eax

loc_80484E1:                            ; CODE XREF: f+140
                mov     cl, [edx+eax]
                mov     bl, [esi+eax]
                mov     [edx+eax], bl
                mov     [esi+eax], cl
                inc     eax
                cmp     [ebp+var_120], eax
                jnz     short loc_80484E1

loc_80484F6:                            ; CODE XREF: f+ED
                                        ; f+129
                mov     eax, [ebp+var_120]
                mov     edi, [ebp+var_124]
                add     edi, [ebp+var_134]
                lea     ebx, [esi+eax]
                jmp     short loc_8048513

loc_804850D:                            ; CODE XREF: f+17B
                add     ebx, [ebp+var_120]

loc_8048513:                            ; CODE XREF: f+157
                                        ; f+1F9
                push    eax
                push    [ebp+arg_10]
                mov     [ebp+var_138], edx
                push    edx
                push    ebx
                call    [ebp+arg_C]
                add     esp, 10h
```

```
                mov     edx, [ebp+var_138]
                test    eax, eax
                jns     short loc_8048537
                jmp     short loc_804850D

loc_8048531:                            ; CODE XREF: f+19D
                add     edi, [ebp+var_134]

loc_8048537:                            ; CODE XREF: f+179
                push    ecx
                push    [ebp+arg_10]
                mov     [ebp+var_138], edx
                push    edi
                push    edx
                call    [ebp+arg_C]
                add     esp, 10h
                mov     edx, [ebp+var_138]
                test    eax, eax
                js      short loc_8048531
                cmp     ebx, edi
                jnb     short loc_8048596
                xor     eax, eax
                mov     [ebp+var_128], edx

loc_804855F:                            ; CODE XREF: f+1BE
                mov     cl, [ebx+eax]
                mov     dl, [edi+eax]
                mov     [ebx+eax], dl
                mov     [edi+eax], cl
                inc     eax
                cmp     [ebp+var_120], eax
                jnz     short loc_804855F
                mov     edx, [ebp+var_128]
                cmp     edx, ebx
                jnz     short loc_8048582
                mov     edx, edi
                jmp     short loc_8048588

loc_8048582:                            ; CODE XREF: f+1C8
                cmp     edx, edi
                jnz     short loc_8048588
                mov     edx, ebx
```

```
loc_8048588:                            ; CODE XREF: f+1CC
                                        ; f+1D0
                add     ebx, [ebp+var_120]
                add     edi, [ebp+var_134]
                jmp     short loc_80485AB

loc_8048596:                            ; CODE XREF: f+1A1
                jnz     short loc_80485AB
                mov     ecx, [ebp+var_134]
                mov     eax, [ebp+var_120]
                lea     edi, [ebx+ecx]
                add     ebx, eax
                jmp     short loc_80485B3

loc_80485AB:                            ; CODE XREF: f+1E0
                                        ; f:loc_8048596
                cmp     ebx, edi
                jbe     loc_8048513

loc_80485B3:                            ; CODE XREF: f+1F5
                mov     eax, edi
                sub     eax, esi
                cmp     eax, [ebp+var_130]
                ja      short loc_80485EB
                mov     eax, [ebp+var_124]
                mov     esi, ebx
                sub     eax, ebx
                cmp     eax, [ebp+var_130]
                ja      short loc_8048634
                sub     [ebp+var_11C], 8
                mov     edx, [ebp+var_11C]
                mov     ecx, [edx+4]
                mov     esi, [edx]
                mov     [ebp+var_124], ecx
                jmp     short loc_8048634

loc_80485EB:                            ; CODE XREF: f+209
                mov     edx, [ebp+var_124]
                sub     edx, ebx
                cmp     edx, [ebp+var_130]
                jbe     short loc_804862E
                cmp     eax, edx
                mov     edx, [ebp+var_11C]
```

```
                lea     eax, [edx+8]
                jle     short loc_8048617
                mov     [edx], esi
                mov     esi, ebx
                mov     [edx+4], edi
                mov     [ebp+var_11C], eax
                jmp     short loc_8048634

loc_8048617:                            ; CODE XREF: f+252
                mov     ecx, [ebp+var_11C]
                mov     [ebp+var_11C], eax
                mov     [ecx], ebx
                mov     ebx, [ebp+var_124]
                mov     [ecx+4], ebx

loc_804862E:                            ; CODE XREF: f+245
                mov     [ebp+var_124], edi

loc_8048634:                            ; CODE XREF: f+21B
                                        ; f+235 ...
                mov     eax, [ebp+var_14C]
                cmp     [ebp+var_11C], eax
                ja      loc_8048433
                mov     ebx, [ebp+var_120]

loc_804864C:                            ; CODE XREF: f+2A
                mov     eax, [ebp+arg_4]
                mov     ecx, [ebp+arg_0]
                add     ecx, [ebp+var_130]
                dec     eax
                imul    eax, ebx
                add     eax, [ebp+arg_0]
                cmp     ecx, eax
                mov     [ebp+var_120], eax
                jbe     short loc_804866B
                mov     ecx, eax

loc_804866B:                            ; CODE XREF: f+2B3
                mov     esi, [ebp+arg_0]
                mov     edi, [ebp+arg_0]
                add     esi, ebx
                mov     edx, esi
                jmp     short loc_80486A3
```

```
loc_8048677:                            ; CODE XREF: f+2F1
                push    eax
                push    [ebp+arg_10]
                mov     [ebp+var_138], edx
                mov     [ebp+var_13C], ecx
                push    edi
                push    edx
                call    [ebp+arg_C]
                add     esp, 10h
                mov     edx, [ebp+var_138]
                mov     ecx, [ebp+var_13C]
                test    eax, eax
                jns     short loc_80486A1
                mov     edi, edx

loc_80486A1:                            ; CODE XREF: f+2E9
                add     edx, ebx

loc_80486A3:                            ; CODE XREF: f+2C1
                cmp     edx, ecx
                jbe     short loc_8048677
                cmp     edi, [ebp+arg_0]
                jz      loc_8048762
                xor     eax, eax

loc_80486B2:                            ; CODE XREF: f+313
                mov     ecx, [ebp+arg_0]
                mov     dl, [edi+eax]
                mov     cl, [ecx+eax]
                mov     [edi+eax], cl
                mov     ecx, [ebp+arg_0]
                mov     [ecx+eax], dl
                inc     eax
                cmp     ebx, eax
                jnz     short loc_80486B2
                jmp     loc_8048762

loc_80486CE:                            ; CODE XREF: f+3C3
                lea     edx, [esi+edi]
                jmp     short loc_80486D5

loc_80486D3:                            ; CODE XREF: f+33B
```

```
                add     edx, edi

loc_80486D5:                            ; CODE XREF: f+31D
                push    eax
                push    [ebp+arg_10]
                mov     [ebp+var_138], edx
                push    edx
                push    esi
                call    [ebp+arg_C]
                add     esp, 10h
                mov     edx, [ebp+var_138]
                test    eax, eax
                js      short loc_80486D3
                add     edx, ebx
                cmp     edx, esi
                mov     [ebp+var_124], edx
                jz      short loc_804876F
                mov     edx, [ebp+var_134]
                lea     eax, [esi+ebx]
                add     edx, eax
                mov     [ebp+var_11C], edx
                jmp     short loc_804875B

loc_8048710:                            ; CODE XREF: f+3AA
                mov     cl, [eax]
                mov     edx, [ebp+var_11C]
                mov     [ebp+var_150], eax
                mov     byte ptr [ebp+var_130], cl
                mov     ecx, eax
                jmp     short loc_8048733

loc_8048728:                            ; CODE XREF: f+391
                mov     al, [edx+ebx]
                mov     [ecx], al
                mov     ecx, [ebp+var_128]

loc_8048733:                            ; CODE XREF: f+372
                mov     [ebp+var_128], edx
                add     edx, edi
                mov     eax, edx
                sub     eax, edi
                cmp     [ebp+var_124], eax
                jbe     short loc_8048728
```

```
                mov     dl, byte ptr [ebp+var_130]
                mov     eax, [ebp+var_150]
                mov     [ecx], dl
                dec     [ebp+var_11C]

loc_804875B:                            ; CODE XREF: f+35A
                dec     eax
                cmp     eax, esi
                jnb     short loc_8048710
                jmp     short loc_804876F

loc_8048762:                            ; CODE XREF: f+2F6
                                        ; f+315
                mov     edi, ebx
                neg     edi
                lea     ecx, [edi-1]
                mov     [ebp+var_134], ecx

loc_804876F:                            ; CODE XREF: f+347
                                        ; f+3AC
                add     esi, ebx
                cmp     esi, [ebp+var_120]
                jbe     loc_80486CE

loc_804877D:                            ; CODE XREF: f+13
                lea     esp, [ebp-0Ch]
                pop     ebx
                pop     esi
                pop     edi
                pop     ebp
                retn
f               endp
```

74.2 Exercise 3.2

There is a small executable file with a well-known cryptosystem inside. Try to identify it.

- Windows x86: http://yurichev.com/RE-exercises/3/2/unknown_cryptosystem.exe
- Linux x86: http://yurichev.com/RE-exercises/3/2/unknown_encryption_linux86.tar

- Mac OS X (x64): http://yurichev.com/RE-exercises/3/2/unknown_encryption_MacOSX.tar

74.3 Exercise 3.3

There is a small executable file, some utility. It opens another file, reads it, calculate something and prints a float number. Try to understand what it do.

- Windows x86: http://yurichev.com/RE-exercises/3/3/unknown_utility_2_3.exe
- Linux x86: http://yurichev.com/RE-exercises/3/3/unknown_utility_2_3_Linux86.tar
- Mac OS X (x64): http://yurichev.com/RE-exercises/3/3/unknown_utility_2_3_MacOSX.tar

74.4 Exercise 3.4

There is an utility which encrypts/decrypts files, by password. There is an encrypted text file, password is unknown. Encrypted file is a text in English language. The utility uses relatively strong cryptosystem, nevertheless, it was implemented with a serious blunder. Since the mistake present, it is possible to decrypt the file with a little effort..

Try to find the mistake and decrypt the file.

- Windows x86: http://yurichev.com/RE-exercises/3/4/amateur_cryptor.exe
- Text file: http://yurichev.com/RE-exercises/3/4/text_encrypted

74.5 Exercise 3.5

This is software copy protection imitation, which uses key file. The key file contain user (or customer) name and serial number.

There are two tasks:

- (Easy) with the help of tracer or any other debugger, force the program to accept changed key file.
- (Medium) your goal is to modify user name to another, however, it is not allowed to patch the program.
- Windows x86: http://yurichev.com/RE-exercises/3/5/super_mega_protection.exe

- Linux x86: http://yurichev.com/RE-exercises/3/5/super_mega_protection.tar
- Mac OS X (x64) http://yurichev.com/RE-exercises/3/5/super_mega_protectio MacOSX.tar
- Key file: http://yurichev.com/RE-exercises/3/5/sample.key

74.6 Exercise 3.6

Here is a very primitive toy web-server, supporting only static files, without CGI[1], etc. At least 4 vulnerabilities are leaved here intentionally. Try to find them all and exploit them in order for breaking into a remote host.

- Windows x86: http://yurichev.com/RE-exercises/3/6/webserv_win32.rar
- Linux x86: http://yurichev.com/RE-exercises/3/6/webserv_Linux_x86.tar
- Mac OS X (x64): http://yurichev.com/RE-exercises/3/6/webserv_MacOSX_x64.tar

74.7 Exercise 3.7

With the help of tracer or any other win32 debugger, reveal hidden mines in the MineSweeper standard Widnows game during play.

Hint: [34] have some insights about MineSweeper's internals.

74.8 Exercise 3.8

It's a well known data compression algorithm. However, due to mistake (or typo), it decompress incorrectly. Here we can see this bug in these examples.
This is a text used as a source: http://yurichev.com/RE-exercises/3/8/test.txt
This is a text compressed correctly: http://yurichev.com/RE-exercises/3/8/test.compressed
This is incorrectly uncompressed text: http://yurichev.com/RE-exercises/3/8/test.uncompressed_incorrectly.

Try to find and fix bug. With some effort, it can be done even by patching.

[1] Common Gateway Interface

- Windows x86: http://yurichev.com/RE-exercises/3/8/compressor_win32.exe
- Linux x86: http://yurichev.com/RE-exercises/3/8/compressor_linux86.tar
- Mac OS X (x64): http://yurichev.com/RE-exercises/3/8/compressor_MacOSX64.tar

Chapter 75

crackme / keygenme

Couple of my keygenmes:
http://crackmes.de/users/yonkie/

Afterword

Chapter 76

Questions?

Do not hesitate to mail any questions to the author: <dennis@yurichev.com>

Please, also do not hesitate to send me any corrections (including grammar ones (you see how horrible my English is?)), etc.

Appendix

Appendix A

Common terminology

word usually is a variable fitting into GPR of CPU. In the computers older than personal, memory size was often measured in words rather then bytes.

Appendix B

x86

B.1 Terminology

Common for 16-bit (8086/80286), 32-bit (80386, etc), 64-bit.

byte 8-bit. DB assembly directive is used for defining variables and array of bytes. Bytes are passed in 8-bit part of registers: `AL/BL/CL/DL/AH/BH/CH/DH/SIL/DIL/R*L`.

word 16-bit. DW assembly directive —"—. Words are passed in 16-bit part of registers: `AX/BX/CX/DX/SI/DI/R*W`.

double word ("dword") 32-bit. DD assembly directive —"—. Double words are passed in registers (x86) or in 32-bit part of registers (x64). In 16-bit code, double words are passed in 16-bit register pairs.

quad word ("qword") 64-bit. DQ assembly directive —"—. In 32-bit code, quad words are passed in 32-bit register pairs.

tbyte (10 bytes) 80-bit or 10 bytes (used for IEEE 754 FPU registers).

paragraph (16 bytes)—term was popular in MS-DOS environment.

Data types of the same width (BYTE, WORD, DWORD) are also the same in Windows API.

B.2 General purpose registers

It is possible to access many registers by byte or 16-bit word parts. It is all inheritance from older Intel CPUs (up to 8-bit 8080) still supported for backward compatibility. For example, this feature is usually not present in RISC CPUs.

Registers prefixed with R- appeared in x86-84, and those prefixed with E- —in 80386. Thus, R-registers are 64-bit, and E-registers —32-bit.

8 more GPR's were added in x86-86: R8-R15.

N.B.: In the Intel manuals byte parts of these registers are prefixed by *L*, e.g.: *R8L*, but IDA names these registers by adding *B* suffix, e.g.: *R8B*.

B.2.1 RAX/EAX/AX/AL

7th (byte number)	6th	5th	4th	3rd	2nd	1st	0th
RAX^{x64}							
				EAX			
						AX	
						AH	AL

AKA accumulator. The result of function if usually returned via this register.

B.2.2 RBX/EBX/BX/BL

7th (byte number)	6th	5th	4th	3rd	2nd	1st	0th
RBX^{x64}							
				EBX			
						BX	
						BH	BL

B.2.3 RCX/ECX/CX/CL

7th (byte number)	6th	5th	4th	3rd	2nd	1st	0th
RCX^{x64}							
				ECX			
						CX	
						CH	CL

AKA counter: in this role it is used in REP prefixed instructions and also in shift instructions (SHL/SHR/RxL/RxR).

B.2.4 RDX/EDX/DX/DL

7th (byte number)	6th	5th	4th	3rd	2nd	1st	0th
RDX^{x64}							
				EDX			
						DX	
						DH	DL

B.2.5 RSI/ESI/SI/SIL

7th (byte number)	6th	5th	4th	3rd	2nd	1st	0th
RSI[x64]							
				ESI			
						SI	
							SIL[x64]

AKA "source". Used as source in the instructions REP MOVSx, REP CMPSx.

B.2.6 RDI/EDI/DI/DIL

7th (byte number)	6th	5th	4th	3rd	2nd	1st	0th
RDI[x64]							
				EDI			
						DI	
							DIL[x64]

AKA "destination". Used as a pointer to destination place in the instructions REP MOVSx, REP STOSx.

B.2.7 R8/R8D/R8W/R8L

7th (byte number)	6th	5th	4th	3rd	2nd	1st	0th
R8							
				R8D			
						R8W	
						R8L	

B.2.8 R9/R9D/R9W/R9L

7th (byte number)	6th	5th	4th	3rd	2nd	1st	0th
R9							
				R9D			
						R9W	
						R9L	

B.2.9 R10/R10D/R10W/R10L

7th (byte number)	6th	5th	4th	3rd	2nd	1st	0th
R10							
				R10D			
						R10W	
						R10L	

B.2.10 R11/R11D/R11W/R11L

7th (byte number)	6th	5th	4th	3rd	2nd	1st	0th
R11							
				R11D			
						R11W	
						R11L	

B.2.11 R12/R12D/R12W/R12L

7th (byte number)	6th	5th	4th	3rd	2nd	1st	0th
R12							
				R12D			
						R12W	
						R12L	

B.2.12 R13/R13D/R13W/R13L

7th (byte number)	6th	5th	4th	3rd	2nd	1st	0th
R13							
				R13D			
						R13W	
						R13L	

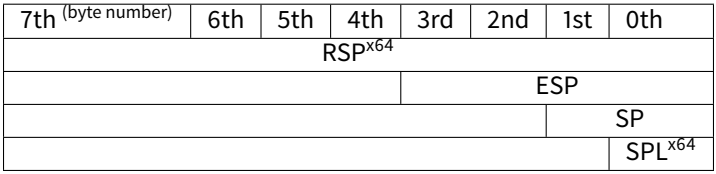
B.2.13 R14/R14D/R14W/R14L

7th (byte number)	6th	5th	4th	3rd	2nd	1st	0th
R14							
				R14D			
						R14W	
						R14L	

B.2.14 R15/R15D/R15W/R15L

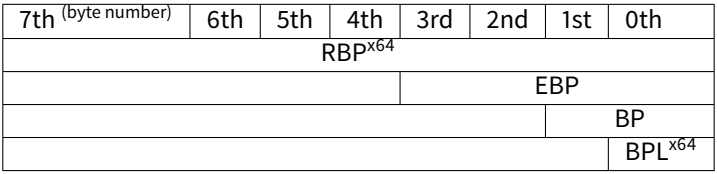
7th (byte number)	6th	5th	4th	3rd	2nd	1st	0th
R15							
				R15D			
						R15W	
						R15L	

B.2.15 RSP/ESP/SP/SPL



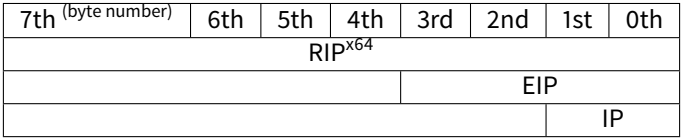
AKA stack pointer. Usually points to the current stack except those cases when it is not yet initialized.

B.2.16 RBP/EBP/BP/BPL



AKA frame pointer. Usually used for local variables and arguments of function accessing. More about it: (6.2.1).

B.2.17 RIP/EIP/IP



AKA "instruction pointer" [1]. Usually always points to the current instruction. Cannot be modified, however, it is possible to do (which is equivalent to):

```
mov eax...
jmp eax
```

Or:

```
push val
ret
```

[1] Sometimes also called "program counter"

B.2.18 CS/DS/ES/SS/FS/GS

16-bit registers containing code selector (CS), data selector (DS), stack selector (SS).

FS in win32 points to TLS, GS took this role in Linux. It is done for faster access to the TLS and other structures like TIB.
In the past, these registers were used as segment registers (67).

B.2.19 Flags register

AKA EFLAGS.

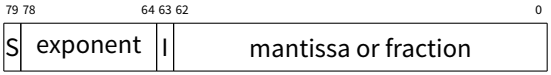
Bit (mask)	Abbreviation (meaning)	Description
0 (1)	CF (Carry)	The CLC/STC/CMC instructions are for setting/resetting/toggling this
2 (4)	PF (Parity)	(15.3.1).
4 (0x10)	AF (Adjust)	
6 (0x40)	ZF (Zero)	Setting to 0 if the last operation's result was 0.
7 (0x80)	SF (Sign)	
8 (0x100)	TF (Trap)	Used for debugging. If turned on, an exception will be generated after each instruction ex
9 (0x200)	IF (Interrupt enable)	Are interrupts enabled. The CLI/STI instructions are used for the flag setting/resetting
10 (0x400)	DF (Direction)	A directions is set for the REP MOVSx, REP CMPSx, REP LODS The CLD/STD instructions are used for the flag setting/resetting
11 (0x800)	OF (Overflow)	
12, 13 (0x3000)	IOPL (I/O privilege level)[80286]	
14 (0x4000)	NT (Nested task)[80286]	
16 (0x10000)	RF (Resume)[80386]	Used for debugging. CPU will ignore hardware breakpo if the flag is set.
17 (0x20000)	VM (Virtual 8086 mode)[80386]	
18 (0x40000)	AC (Alignment check)[80486]	
19 (0x80000)	VIF (Virtual interrupt)[Pentium]	
20 (0x100000)	VIP (Virtual interrupt pending)[Pentium]	
21 (0x200000)	ID (Identification)[Pentium]	

All the rest flags are reserved.

B.3 FPU-registers

8 80-bit registers working as a stack: ST(0)-ST(7). N.B.: IDA calls ST(0) as just ST. Numbers are stored in the IEEE 754 format.

long double value format:



(S—sign, I—integer part)

B.3.1 Control Word

Register controlling behaviour of the FPU.

Bit	Abbreviation (meaning)	Description
0	IM (Invalid operation Mask)	
1	DM (Denormalized operand Mask)	
2	ZM (Zero divide Mask)	
3	OM (Overflow Mask)	
4	UM (Underflow Mask)	
5	PM (Precision Mask)	
7	IEM (Interrupt Enable Mask)	Exceptions enabling, 1 by default (disabled)
8, 9	PC (Precision Control)	00 — 24 bits (REAL4) 10 — 53 bits (REAL8) 11 — 64 bits (REAL10)
10, 11	RC (Rounding Control)	00 — (by default) round to nearest 01 — round toward $-\infty$ 10 — round toward $+\infty$ 11 — round toward 0
12	IC (Infinity Control)	0 — (by default) treat $+\infty$ and $-\infty$ as unsigned 1 — respect both $+\infty$ and $-\infty$

The PM, UM, OM, ZM, DM, IM flags are defining if to generate exception in case of corresponding errors.

B.3.2 Status Word

Read-only register.

Bit	Abbreviation (meaning)	Description
15	B (Busy)	Is FPU do something (1) or results are ready (0)
14	C3	
13, 12, 11	TOP	points to the currently zeroth register
10	C2	
9	C1	
8	C0	
7	IR (Interrupt Request)	
6	SF (Stack Fault)	
5	P (Precision)	
4	U (Underflow)	
3	O (Overflow)	
2	Z (Zero)	
1	D (Denormalized)	
0	I (Invalid operation)	

The SF, P, U, O, Z, D, I bits are signaling about exceptions.

About the C3, C2, C1, C0 read more: (15.3.1).

N.B.: When ST(x) is used, FPU adds x to TOP (by modulo 8) and that is how it gets internal register's number.

B.3.3 Tag Word

The register has current information about number's registers usage.

Bit	Abbreviation (meaning)
15, 14	Tag(7)
13, 12	Tag(6)
11, 10	Tag(5)
9, 8	Tag(4)
7, 6	Tag(3)
5, 4	Tag(2)
3, 2	Tag(1)
1, 0	Tag(0)

For each tag:

- 00 — The register contains a non-zero value
- 01 — The register contains 0
- 10 — The register contains a special value (NAN[2], ∞, or denormal)
- 11 — The register is empty

[2] Not a Number

B.4 SIMD-registers

B.4.1 MMX-registers

8 64-bit registers: MM0..MM7.

B.4.2 SSE and AVX-registers

SSE: 8 128-bit registers: XMM0..XMM7. In the x86-64 8 more registers were added: XMM8..XMM15.

AVX is the extension of all these registers to 256 bits.

B.5 Debugging registers

Used for hardware breakpoints control.

- DR0 — address of breakpoint #1
- DR1 — address of breakpoint #2
- DR2 — address of breakpoint #3
- DR3 — address of breakpoint #4
- DR6 — a cause of break is reflected here
- DR7 — breakpoint types are set here

B.5.1 DR6

Bit (mask)	Description
0 (1)	B0 — breakpoint #1 was triggered
1 (2)	B1 — breakpoint #2 was triggered
2 (4)	B2 — breakpoint #3 was triggered
3 (8)	B3 — breakpoint #4 was triggered
13 (0x2000)	BD — modification attempt of one of DRx registers. may be raised if GD is enabled
14 (0x4000)	BS — single step breakpoint (TF flag was set in EFLAGS). Highest priority. Other bits may also be set.
15 (0x8000)	BT (task switch flag)

N.B. Single step breakpoint is a breakpoint occurring after each instruction. It can be enabled by setting TF in EFLAGS (B.2.19).

B.5.2 DR7

Breakpoint types are set here.

Bit (mask)	Description
0 (1)	L0 — enable breakpoint #1 for the current task
1 (2)	G0 — enable breakpoint #1 for all tasks
2 (4)	L1 — enable breakpoint #2 for the current task
3 (8)	G1 — enable breakpoint #2 for all tasks
4 (0x10)	L2 — enable breakpoint #3 for the current task
5 (0x20)	G2 — enable breakpoint #3 for all tasks
6 (0x40)	L3 — enable breakpoint #4 for the current task
7 (0x80)	G3 — enable breakpoint #4 for all tasks
8 (0x100)	LE — not supported since P6
9 (0x200)	GE — not supported since P6
13 (0x2000)	GD — exception will be raised if any MOV instruction tries to modify one of DRx registers
16,17 (0x30000)	breakpoint #1: R/W — type
18,19 (0xC0000)	breakpoint #1: LEN — length
20,21 (0x300000)	breakpoint #2: R/W — type
22,23 (0xC00000)	breakpoint #2: LEN — length
24,25 (0x3000000)	breakpoint #3: R/W — type
26,27 (0xC000000)	breakpoint #3: LEN — length
28,29 (0x30000000)	breakpoint #4: R/W — type
30,31 (0xC0000000)	breakpoint #4: LEN — length

Breakpoint type is to be set as follows (R/W):

- 00 — instruction execution
- 01 — data writes
- 10 — I/O reads or writes (not available in user-mode)
- 11 — on data reads or writes

N.B.: breakpoint type for data reads is absent, indeed.

Breakpoint length is to be set as follows (LEN):

- 00 — one-byte
- 01 — two-byte
- 10 — undefined for 32-bit mode, eight-byte in 64-bit mode
- 11 — four-byte

B.6 Instructions

Instructions marked as (M) are not usually generated by compiler: if you see it, it is probably hand-written piece of assembly code, or this is compiler intrinsic (63).

Only most frequently used instructions are listed here. Read [14] or [1] for a full documentation.

B.6.1 Prefixes

LOCK force CPU to make exclusive access to the RAM in multiprocessor environment. For the sake of simplification, it can be said that when instruction with this prefix is executed, all other CPUs in multiprocessor system is stopped. Most often it is used for critical sections, semaphores, mutexes. Commonly used with ADD, AND, BTR, BTS, CMPXCHG, OR, XADD, XOR. Read more about critical sections (48.4).

REP used with MOVSx and STOSx: execute the instruction in loop, counter is located in the CX/ECX/RCX register. For detailed description, read more about MOVSx (B.6.2) and STOSx (B.6.2) instructions.

Instructions prefixed by REP are sensitive to DF flag, which is used to set direction.

REPE/REPNE (AKA REPZ/REPNZ) used with CMPSx and SCASx: execute the last instruction in loop, count is set in the CX/ECX/RCX register. It will terminate prematurely if ZF is 0 (REPE) or if ZF is 1 (REPNE).

For detailed description, read more about CMPSx (B.6.3) and SCASx (B.6.2) instructions.

Instructions prefixed by REPE/REPNE are sensitive to DF flag, which is used to set direction.

B.6.2 Most frequently used instructions

These can be memorized in the first place.

ADC (*add with carry*) add values, increment result if CF flag is set. often used for addition of large values, for example, to add two 64-bit values in 32-bit environment using two ADD and ADC instructions, for example:

```
; work with 64-bit values: add val1 to val2.
; .lo mean lowest 32 bits, .hi means highest.
ADD val1.lo, val2.lo
ADC val1.hi, val2.hi ; use CF set or cleared at the previous
    ↪ instruction
```

One more example: 21.

ADD add two values

AND logical "and"

CALL call another function: `PUSH address_after_CALL_instruction; JMP label`

CMP compare values and set flags, the same as `SUB` but no results writing

DEC decrement. CF flag is not touched.

IMUL signed multiply

INC increment. CF flag is not touched.

JCXZ, JECXZ, JRCXZ (M) jump if CX/ECX/RCX=0

JMP jump to another address. Opcode has jump offset.

Jcc (where cc—condition code)

A lot of instructions has synonyms (denoted with AKA), this was done for convenience. Synonymous instructions are translating into the same opcode. Opcode has jump offset.

JAE AKA JNC: jump if above or equal (unsigned): CF=0

JA AKA JNBE: jump if greater (unsigned): CF=0 and ZF=0

JBE jump if lesser or equal (unsigned): CF=1 or ZF=1

JB AKA JC: jump if below (unsigned): CF=1

JC AKA JB: jump if CF=1

JE AKA JZ: jump if equal or zero: ZF=1

JGE jump if greater or equal (signed): SF=OF

JG jump if greater (signed): ZF=0 and SF=OF

JLE jump if lesser or equal (signed): ZF=1 or SF≠OF

JL jump if lesser (signed): SF≠OF

JNAE AKA JC: jump if not above or equal (unsigned) CF=1

JNA jump if not above (unsigned) CF=1 and ZF=1

JNBE jump if not below or equal (unsigned): CF=0 and ZF=0

JNB AKA JNC: jump if not below (unsigned): CF=0

JNC AKA JAE: jump CF=0 synonymous to JNB.

JNE AKA JNZ: jump if not equal or not zero: ZF=0

JNGE jump if not greater or equal (signed): SF≠OF

JNG jump if not greater (signed): ZF=1 or SF≠OF

JNLE jump if not lesser (signed): ZF=0 and SF=OF

JNL jump if not lesser (signed): SF=OF

JNO jump if not overflow: OF=0

JNS jump if SF flag is cleared

JNZ AKA JNE: jump if not equal or not zero: ZF=0

JO jump if overflow: OF=1

JPO jump if PF flag is cleared

JP AKA JPE: jump if PF flag is set

JS jump if SF flag is set

JZ AKA JE: jump if equal or zero: ZF=1

LAHF copy some flag bits to AH

LEAVE equivalent of the `MOV ESP, EBP` and `POP EBP` instruction pair—in other words, this instruction sets the stack pointer (`ESP`) back and restores the `EBP` register to its initial state.

LEA (*Load Effective Address*) form address

This instruction was intended not for values summing and multiplication but for address forming, e.g., for forming address of array element by adding array address, element index, with multiplication of element size[3].

So, the difference between `MOV` and `LEA` is that `MOV` forms memory address and loads value from memory or stores it there, but `LEA` just forms an address.

But nevertheless, it is can be used for any other calculations.

`LEA` is convenient because the computations performing by it is not alter CPU flags.

```
int f(int a, int b)
{
        return a*8+b;
};
```

[3] See also: http://en.wikipedia.org/wiki/Addressing_mode

Listing B.1: MSVC 2010 /Ox

```
_a$ = 8                                                    ; ↙
    ↳ size = 4
_b$ = 12                                                   ; ↙
    ↳ size = 4
_f      PROC
        mov     eax, DWORD PTR _b$[esp-4]
        mov     ecx, DWORD PTR _a$[esp-4]
        lea     eax, DWORD PTR [eax+ecx*8]
        ret     0
_f      ENDP
```

Intel C++ uses LEA even more:

```
int f1(int a)
{
        return a*13;
};
```

Listing B.2: Intel C++ 2011

```
_f1     PROC NEAR
        mov       ecx, DWORD PTR [4+esp]      ; ecx = a
        lea       edx, DWORD PTR [ecx+ecx*8]  ; edx = a*9
        lea       eax, DWORD PTR [edx+ecx*4]  ; eax = a*9 + ↙
    ↳ a*4 = a*13
        ret
```

These two instructions instead of one IMUL will perform faster.

MOVSB/MOVSW/MOVSD/MOVSQ copy byte/ 16-bit word/ 32-bit word/ 64-bit word address of which is in the SI/ESI/RSI into the place address of which is in the DI/EDI/RDI.

Together with REP prefix, it will repeated in loop, count is stored in the CX/ECX/RCX register: it works like memcpy() in C. If block size is known to compiler on compile stage, memcpy() is often inlined into short code fragment using REP MOVSx, sometimes even as several instructions.

memcpy(EDI, ESI, 15) equivalent is:

```
; copy 15 bytes from ESI to EDI
CLD         ; set direction to "forward"
MOV ECX, 3
REP MOVSD   ; copy 12 bytes
MOVSW       ; copy 2 more bytes
MOVSB       ; copy remaining byte
```

(Supposedly, it will work faster then copying 15 bytes using just one REP MOVSB).

MOVSX load with sign extension see also: (13.1.1)

MOVZX load and clear all the rest bits see also: (13.1.2)

MOV load value. this instruction was named awry resulting confusion (data are not moved), in other architectures the same instructions is usually named "LOAD" or something like that.

One important thing: if to set low 16-bit part of 32-bit register in 32-bit mode, high 16 bits will remain as they were. But if to modify low 32-bit of register in 64-bit mode, high 32 bits of registers will be cleared.

Supposedly, it was done for x86-64 code porting simplification.

MUL unsigned multiply

NEG negation: $op = -op$

NOP NOP. Opcode is 0x90, so it is in fact mean `XCHG EAX,EAX` idle instruction. This means, x86 do not have dedicated NOP instruction (as in many RISC). More examples of such operations: (61).

NOP may be generated by compiler for aligning labels on 16-byte boundary. Another very popular usage of NOP is to replace manually (patch) some instruction like conditional jump to NOP in order to disable its execution.

NOT op1: $op1 = \neg op1$. logical inversion

OR logical "or"

POP get value from the stack: `value=SS:[ESP]; ESP=ESP+4 (or 8)`

PUSH push value to stack: `ESP=ESP-4 (or 8); SS:[ESP]=value`

RET : return from subroutine: `POP tmp; JMP tmp`.

In fact, RET is a assembly language macro, in Windows and *NIX environment is translating into RETN ("return near") or, in MS-DOS times, where memory was addressed differently (67), into RETF ("return far").

`RET` may have operand. Its algorithm then will be: `POP tmp; ADD ESP op1; JMP tmp`. `RET` with operand usually end functions with *stdcall* calling convention, see also: 44.2.

SAHF copy bits from AH to flags, see also: 15.3.3

SBB (*subtraction with borrow*) subtract values, decrement result if CF flag is set. often used for subtraction of large values, for example, to subtract two 64-bit values in 32-bit environment using two SUB and SBB instructions, for example:

```
; work with 64-bit values: subtract val2 from val1.
; .lo mean lowest 32 bits, .hi means highest.
SUB val1.lo, val2.lo
SBB val1.hi, val2.hi ; use CF set or cleared at the previous↙
    ↳ instruction
```

One more example: 21.

SCASB/SCASW/SCASD/SCASQ (M) compare byte/ 16-bit word/ 32-bit word/ 64-bit word stored in the AX/EAX/RAX with a variable address of which is in the DI/EDI/RDI. Set flags as CMP does.

This instruction is often used with REPNE prefix: continue to scan a buffer until a special value stored in AX/EAX/RAX is found. Hence "NE" in REPNE: continue to scan if compared values are not equal and stop when equal.

It is often used as strlen() C standard function, to determine ASCIIZ string length:

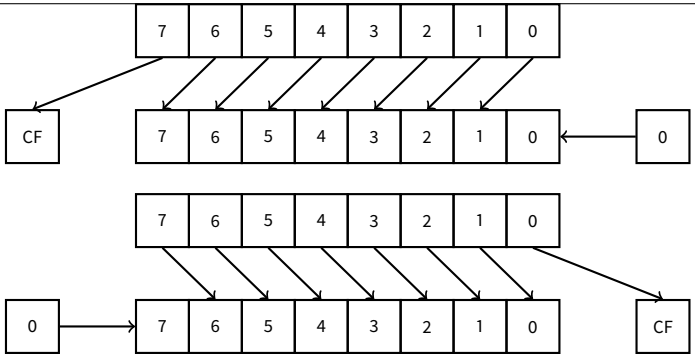
Example:

```
lea     edi, string
mov     ecx, 0FFFFFFFFh ; scan 2^32-1 bytes, i.e., almost "↙
    ↳ infinitely"
xor     eax, eax        ; 0 is the terminator
repne scasb
add     edi, 0FFFFFFFFh ; correct it

; now EDI points to the last character of the ASCIIZ string.

; let's determine string length
; current ECX = -1-strlen

not     ecx
dec     ecx

; now ECX contain string length
```

If to use different AX/EAX/RAX value, the function will act as memchr() standard C function, i.e., it will find specific byte.

SHL shift value left

SHR shift value right:



This instruction is frequently used for multiplication and division by 2^n. Another very frequent application is bit fields processing: 17.

SHRD op1, op2, op3: shift value in op2 right by op3 bits, taking bits from op1.

Example: 21.

STOSB/STOSW/STOSD/STOSQ store byte/ 16-bit word/ 32-bit word/ 64-bit word from AX/EAX/RAX into the place address of which is in the DI/EDI/RDI.

Together with REP prefix, it will repeated in loop, count is stored in the CX/ECX/RCX register: it works like memset() in C. If block size is known to compiler on compile stage, memset() is often inlined into short code fragment using REP MOVSx, sometimes even as several instructions.

memset(EDI, 0xAA, 15) equivalent is:

```
; store 15 0xAA bytes to EDI
CLD                 ; set direction to "forward"
MOV EAX, 0AAAAAAAAh
MOV ECX, 3
REP STOSD           ; write 12 bytes
STOSW               ; write 2 more bytes
STOSB               ; write remaining byte
```

(Supposedly, it will work faster then storing 15 bytes using just one REP STOSB).

SUB subtract values. frequently occurred pattern `SUB reg,reg` meaning write 0 to reg.

TEST same as AND but without results saving, see also: 17

XCHG exchange values in operands

XOR op1, op2: XOR[4] values. $op1 = op1 \oplus op2$. frequently occurred pattern `XOR reg,reg` meaning write 0 to reg.

[4] eXclusive OR

B.6.3 Less frequently used instructions

BSF *bit scan forward*, see also: 22.2

BSR *bit scan reverse*

BSWAP *(byte swap)*, change value endianness.

BTC bit test and complement

BTR bit test and reset

BTS bit test and set

BT bit test

CBW/CWD/CWDE/CDQ/CDQE Sign-extend value:

CBW : convert byte in AL to word in AX

CWD : convert word in AX to doubleword in DX:AX

CWDE : convert word in AX to doubleword in EAX

CDQ : convert doubleword in EAX to quadword in EDX:EAX

CDQE (x64): convert doubleword in EAX to quadword in RAX

These instructions consider value's sign, extending it to high part of newly constructed value. See also: 21.4.

CLD clear DF flag.

CLI (M) clear IF flag

CMC (M) toggle CF flag

CMOVcc conditional MOV: load if condition is true The condition codes are the same as in Jcc instructions (B.6.2).

CMPSB/CMPSW/CMPSD/CMPSQ (M) compare byte/ 16-bit word/ 32-bit word/ 64-bit word from the place address of which is in the SI/ESI/RSI with a variable address of which is in the DI/EDI/RDI. Set flags as `CMP` does.

Together with REP prefix, it will repeated in loop, count is stored in the CX/ECX/RCX register, the process will be running util ZF flag is zero (e.g., until compared values are equal to each other, hence "E" in REPE).

It works like memcmp() in C.

Example from Windows NT kernel (WRK v1.2):

Listing B.3: base\ntos\rtl\i386\movemem.asm

```
; ULONG
; RtlCompareMemory (
;    IN PVOID Source1,
;    IN PVOID Source2,
;    IN ULONG Length
;    )
;
; Routine Description:
;
;    This function compares two blocks of memory and returns the number
;    of bytes that compared equal.
;
; Arguments:
;
;    Source1 (esp+4) - Supplies a pointer to the first block of memory to
;       compare.
;
;    Source2 (esp+8) - Supplies a pointer to the second block of memory to
;       compare.
;
;    Length (esp+12) - Supplies the Length, in bytes, of the memory to be
;       compared.
;
; Return Value:
;
;    The number of bytes that compared equal is returned as the function
;    value. If all bytes compared equal, then the length of the original
;    block of memory is returned.
;
;--

RcmSource1      equ     [esp+12]
RcmSource2      equ     [esp+16]
RcmLength       equ     [esp+20]

CODE_ALIGNMENT
```

```
cPublicProc _RtlCompareMemory,3
cPublicFpo 3,0

        push    esi                     ; save registers
        push    edi                     ;
        cld                             ; clear direction
        mov     esi,RcmSource1          ; (esi) -> first ↙
    ↳ block to compare
        mov     edi,RcmSource2          ; (edi) -> second ↙
    ↳ block to compare

;
;   Compare dwords, if any.
;

rcm10:  mov     ecx,RcmLength           ; (ecx) = length in ↙
    ↳ bytes
        shr     ecx,2                   ; (ecx) = length in ↙
    ↳ dwords
        jz      rcm20                   ; no dwords, try ↙
    ↳ bytes
        repe    cmpsd                   ; compare dwords
        jnz     rcm40                   ; mismatch, go find ↙
    ↳ byte

;
;   Compare residual bytes, if any.
;

rcm20:  mov     ecx,RcmLength           ; (ecx) = length in ↙
    ↳ bytes
        and     ecx,3                   ; (ecx) = length mod↙
    ↳  4
        jz      rcm30                   ; 0 odd bytes, go do↙
    ↳  dwords
        repe    cmpsb                   ; compare odd bytes
        jnz     rcm50                   ; mismatch, go ↙
    ↳ report how far we got

;
;   All bytes in the block match.
;
```

```
rcm30:  mov     eax,RcmLength           ; set number of ↙
    ↳ matching bytes
        pop     edi                     ; restore registers
        pop     esi                     ;
        stdRET  _RtlCompareMemory

;
;   When we come to rcm40, esi (and edi) points to the dword↙
    ↳  after the
;   one which caused the mismatch.  Back up 1 dword and find↙
    ↳  the byte.
;   Since we know the dword didn't match, we can assume one ↙
    ↳ byte won't.
;

rcm40:  sub     esi,4                   ; back up
        sub     edi,4                   ; back up
        mov     ecx,5                   ; ensure that ecx ↙
    ↳ doesn't count out
        repe    cmpsb                   ; find mismatch byte

;
;   When we come to rcm50, esi points to the byte after the ↙
    ↳ one that
;   did not match, which is TWO after the last byte that did↙
    ↳  match.
;

rcm50:  dec     esi                     ; back up
        sub     esi,RcmSource1          ; compute bytes that↙
    ↳  matched
        mov     eax,esi                 ;
        pop     edi                     ; restore registers
        pop     esi                     ;
        stdRET  _RtlCompareMemory

stdENDP _RtlCompareMemory
```

N.B.: this function uses 32-bit words comparison (CMPSD) if block size is multiple of 4, or per-byte comparison (CMPSB) otherwise.

CPUID get information about CPU features. see also: (18.6.1).

DIV unsigned division

IDIV signed division

INT (M): `INT x` is analogous to `PUSHF; CALL dword ptr [x*4]` in 16-bit environment. It was widely used in MS-DOS, functioning as syscalls. Registers AX/BX/CX/DX/SI/D were filled by arguments and jump to the address in the Interrupt Vector Table (located at the address space beginning) will be occurred. It was popular because INT has short opcode (2 bytes) and the program which needs some MS-DOS services is not bothering by determining service's entry point address. Interrupt handler return control flow to called using IRET instruction.

Most busy MS-DOS interrupt number was 0x21, serving a huge amount of its API. See also: [4] for the most comprehensive interrupt lists and other MS-DOS information.

In post-MS-DOS era, this instruction was still used as syscall both in Linux and Windows (46), but later replaced by SYSENTER or SYSCALL instruction.

INT 3 (M): this instruction is somewhat standing aside of `INT`, it has its own 1-byte opcode (`0xCC`), and actively used while debugging. Often, debuggers just write `0xCC` byte at the address of breakpoint to be set, and when exception is raised, original byte will be restored and original instruction at this address will be re-executed.
As of Windows NT, an `EXCEPTION_BREAKPOINT` exception will be raised when CPU executes this instruction. This debugging event may be intercepted and handled by a host debugger, if loaded. If it is not loaded, Windows will offer to run one of the registered in the system debuggers. If MSVS[5] is installed, its debugger may be loaded and connected to the process. In order to protect from reverse engineering, a lot of anti-debugging methods are checking integrity of the code loaded.

MSVC has compiler intrinsic for the instruction: `__debugbreak()`[6].

There are also a win32 function in kernel32.dll named `DebugBreak()`[7], which also executes `INT 3`.

IN (M) input data from port. The instruction is usually can be seen in OS drivers or in old MS-DOS code, for example (55.3).

IRET : was used in MS-DOS environment for returning from interrupt handler after it was called by INT instruction. Equivalent to `POP tmp; POPF; JMP tmp`.

LOOP (M) decrement CX/ECX/RCX, jump if it is still not zero.

OUT (M) output data to port. The instruction is usually can be seen in OS drivers or in old MS-DOS code, for example (55.3).

[5] Microsoft Visual Studio
[6] http://msdn.microsoft.com/en-us/library/f408b4et.aspx
[7] http://msdn.microsoft.com/en-us/library/windows/desktop/ms679297(v=vs.85).aspx

POPA (M) restores values of (R|E)DI, (R|E)SI, (R|E)BP, (R|E)BX, (R|E)DX, (R|E)CX, (R|E)AX registers from stack.

POPCNT population count. counts number of 1 bits in value. AKA "hamming weight". AKA "NSA instruction" because of rumors:

> This branch of cryptography is fast-paced and very politically charged. Most designs are secret; a majority of military encryptions systems in use today are based on LFSRs. In fact, most Cray computers (Cray 1, Cray X-MP, Cray Y-MP) have a rather curious instruction generally known as "population count." It counts the 1 bits in a register and can be used both to efficiently calculate the Hamming distance between two binary words and to implement a vectorized version of a LFSR. I've heard this called the canonical NSA instruction, demanded by almost all computer contracts.

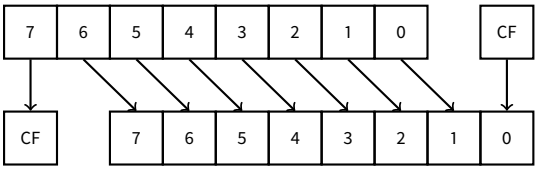
[31]

POPF restore flags from stack (AKA EFLAGS register)

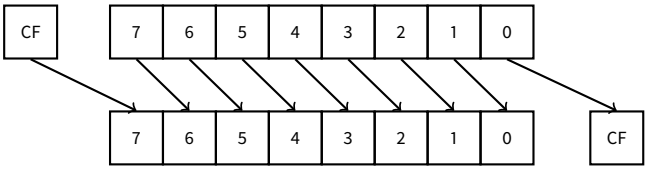
PUSHA (M) pushes values of (R|E)AX, (R|E)CX, (R|E)DX, (R|E)BX, (R|E)BP, (R|E)SI, (R|E)DI registers to the stack.

PUSHF push flags (AKA EFLAGS register)

RCL (M) rotate left via CF flag:

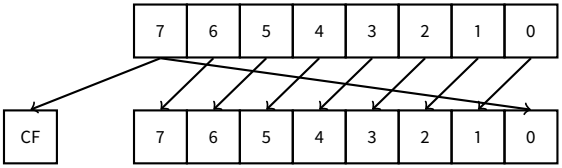


RCR (M) rotate right via CF flag:

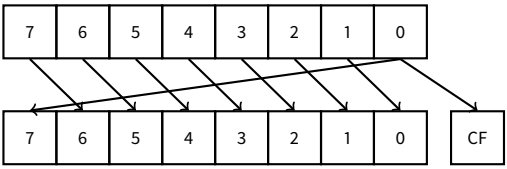


ROL/ROR (M) cyclic shift

ROL: rotate left:



ROR: rotate right:

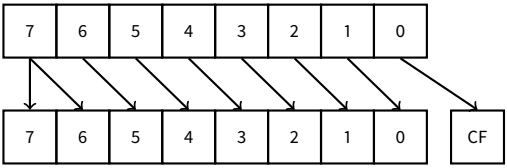


Despite the fact that almost all CPUs has these instructions, there are no corresponding operations in the C/C++, so the compilers of these PLs are usually not generating these instructions.

For programmer's convenience, at least MSVC has pseudofunctions (compiler intrinsics) *_rotl()* and *_rotr()*[8], which are translated by compiler directly to these instructions.

SAL Arithmetic shift left, synonymous to `SHL`

SAR Arithmetic shift right



Hence, sign bit is always stayed at the place of MSB[9].

SETcc op: load 1 to op (byte only) if condition is true or zero otherwise. The condition codes are the same as in Jcc instructions (B.6.2).

STC (M) set CF flag

[8] http://msdn.microsoft.com/en-us/library/5cc576c4.aspx

[9] Most significant bit/byte

STD (M) set DF flag

STI (M) set IF flag

SYSCALL (AMD) call syscall (46)

SYSENTER (Intel) call syscall (46)

UD2 (M) undefined instruction, raises exception. used for testing.

B.6.4 FPU instructions

-R in mnemonic usually means that operands are reversed, -P means that one element is popped from the stack after instruction execution, -PP means that two elements are popped.

-P instructions are often useful when we do not need a value in the FPU stack to be present anymore.

FABS replace value in ST(0) by absolute value in ST(0)

FADD op: ST(0)=op+ST(0)

FADD ST(0), ST(i): ST(0)=ST(0)+ST(i)

FADDP ST(1)=ST(0)+ST(1); pop one element from the stack, i.e., summed values in the stack are replaced by sum

FCHS : ST(0)=-ST(0)

FCOM compare ST(0) with ST(1)

FCOM op: compare ST(0) with op

FCOMP compare ST(0) with ST(1); pop one element from the stack

FCOMPP compare ST(0) with ST(1); pop two elements from the stack

FDIVR op: ST(0)=op/ST(0)

FDIVR ST(i), ST(j): ST(i)=ST(j)/ST(i)

FDIVRP op: ST(0)=op/ST(0); pop one element from the stack

FDIVRP ST(i), ST(j): ST(i)=ST(j)/ST(i); pop one element from the stack

FDIV op: ST(0)=ST(0)/op

FDIV ST(i), ST(j): ST(i)=ST(i)/ST(j)

FDIVP ST(1)=ST(0)/ST(1); pop one element from the stack, i.e., dividend and divisor values in the stack are replaced by quotient

FILD op: convert integer and push it to the stack.

FIST op: convert ST(0) to integer op

FISTP op: convert ST(0) to integer op; pop one element from the stack

FLD1 push 1 to stack

FLDCW op: load FPU control word (B.3) from 16-bit op.

FLDZ push zero to stack

FLD op: push op to the stack.

FMUL op: ST(0)=ST(0)*op

FMUL ST(i), ST(j): ST(i)=ST(i)*ST(j)

FMULP op: ST(0)=ST(0)*op; pop one element from the stack

FMULP ST(i), ST(j): ST(i)=ST(i)*ST(j); pop one element from the stack

FSINCOS : tmp=ST(0); ST(1)=sin(tmp); ST(0)=cos(tmp)

FSQRT : $ST(0) = \sqrt{ST(0)}$

FSTCW op: store FPU control word (B.3) into 16-bit op after checking for pending exceptions.

FNSTCW op: store FPU control word (B.3) into 16-bit op.

FSTSW op: store FPU status word (B.3.2) into 16-bit op after checking for pending exceptions.

FNSTSW op: store FPU status word (B.3.2) into 16-bit op.

FST op: copy ST(0) to op

FSTP op: copy ST(0) to op; pop one element from the stack

FSUBR op: ST(0)=op-ST(0)

FSUBR ST(0), ST(i): ST(0)=ST(i)-ST(0)

FSUBRP ST(1)=ST(0)-ST(1); pop one element from the stack, i.e., summed values in the stack are replaced by difference

FSUB op: ST(0)=ST(0)-op

FSUB ST(0), ST(i): ST(0)=ST(0)-ST(i)

FSUBP ST(1)=ST(1)-ST(0); pop one element from the stack, i.e., summed values in the stack are replaced by difference

FUCOM ST(i): compare ST(0) and ST(i)

FUCOM : compare ST(0) and ST(1)

FUCOMP : compare ST(0) and ST(1); pop one element from stack.

FUCOMPP : compare ST(0) and ST(1); pop two elements from stack.

The instructions performs just like FCOM, but exception is raised only if one of operands is SNaN, while QNaN numbers are processed smoothly.

FXCH ST(i) exchange values in ST(0) and ST(i)

FXCH exchange values in ST(0) and ST(1)

B.6.5 SIMD instructions

B.6.6 Instructions having printable ASCII opcode

(In 32-bit mode).

It can be suitable for shellcode constructing. See also: 59.1.

ASCII character	hexadecimal code	x86 instruction
0	30	XOR
1	31	XOR
2	32	XOR
3	33	XOR
4	34	XOR
5	35	XOR
7	37	AAA
8	38	CMP
9	39	CMP
:	3a	CMP
;	3b	CMP
<	3c	CMP
=	3d	CMP
?	3f	AAS
@	40	INC
A	41	INC
B	42	INC
C	43	INC
D	44	INC
E	45	INC

F	46	INC
G	47	INC
H	48	DEC
I	49	DEC
J	4a	DEC
K	4b	DEC
L	4c	DEC
M	4d	DEC
N	4e	DEC
O	4f	DEC
P	50	PUSH
Q	51	PUSH
R	52	PUSH
S	53	PUSH
T	54	PUSH
U	55	PUSH
V	56	PUSH
W	57	PUSH
X	58	POP
Y	59	POP
Z	5a	POP
[	5b	POP
\	5c	POP
]	5d	POP
^	5e	POP
_	5f	POP
‘	60	PUSHA
a	61	POPA
f	66	(in 32-bit mode) switch to 16-bit operand size
g	67	in 32-bit mode) switch to 16-bit address size
h	68	PUSH
i	69	IMUL
j	6a	PUSH
k	6b	IMUL
p	70	JO
q	71	JNO
r	72	JB
s	73	JAE
t	74	JE
u	75	JNE
v	76	JBE

w	77	JA
x	78	JS
y	79	JNS
z	7a	JP

Summarizing: AAA, AAS, CMP, DEC, IMUL, INC, JA, JAE, JB, JBE, JE, JNE, JNO, JNS, JO, JP, JS, POP, POPA, PUSH, PUSHA, XOR.

Appendix C

ARM

C.1 Terminology

ARM was initially developed as 32-bit CPU, so that's why *word* here, unlike x86, is 32-bit.

byte 8-bit. DB assembly directive is used for defining variables and array of bytes.

halfword 16-bit. DCW assembly directive —"—.

word 32-bit. DCD assembly directive —"—.

C.2 General purpose registers

- R0 — function result is usually returned using R0
- R1
- R2
- R3
- R4
- R5
- R6
- R7
- R8
- R9

- R10
- R11
- R12
- R13 — AKA SP (stack pointer)
- R14 — AKA LR (link register)
- R15 — AKA PC (program counter)

R0-R3 are also called “scratch registers”: function arguments are usually passed in them, and values in them are not necessary to restore upon function exit.

C.3 Current Program Status Register (CPSR)

Bit	Description
0..4	M — processor mode
5	T — Thumb state
6	F — FIQ disable
7	I — IRQ disable
8	A — imprecise data abort disable
9	E — data endianness
10..15, 25, 26	IT — if-then state
16..19	GE — greater-than-or-equal-to
20..23	DNM — do not modify
24	J — Java state
27	Q — sticky overflow
28	V — overflow
29	C — carry/borrow/extend
30	Z — zero bit
31	N — negative/less than

C.4 VFP (floating point) and NEON registers

<table>
<tr><td>0..31 bits</td><td>32..64</td><td>65..96</td><td>97..127</td></tr>
<tr><td colspan="4">Q0 128 bits</td></tr>
<tr><td colspan="2">D0 64 bits</td><td colspan="2">D1</td></tr>
<tr><td>S0 32 bits</td><td>S1</td><td>S2</td><td>S3</td></tr>
</table>

S-registers are 32-bit ones, used for single precision numbers storage.
D-registers are 64-bit ones, used for double precision numbers storage.

D- and S-registers share the same physical space in CPU—it is possible to access D-register via S-registers (it is senseless though).

Likewise, NEON Q-registers are 128-bit ones and share the same physical space in CPU with other floating point registers.

In VFP 32 S-registers are present: S0..S31.

In VFPv2 there are 16 D-registers added, which are, in fact, occupy the same space as S0..S31.

In VFPv3 (NEON or “Advanced SIMD”) there are 16 more D-registers added, resulting D0..D31, but D16..D31 registers are not sharing a space with other S-registers.

In NEON or “Advanced SIMD” there are also 16 128-bit Q-registers added, which share the same space as D0..D31.

Appendix D

Some GCC library functions

name	meaning
`__divdi3`	signed division
`__moddi3`	getting remainder (modulo) of signed division
`__udivdi3`	unsigned division
`__umoddi3`	getting remainder (modulo) of unsigned division

Appendix E

Some MSVC library functions

ll in function name mean “long long”, e.g., 64-bit data type.

name	meaning
__alldiv	signed division
__allmul	multiplication
__allrem	remainder of signed division
__allshl	shift left
__allshr	signed shift right
__aulldiv	unsigned division
__aullrem	remainder of unsigned division
__aullshr	unsigned shift right

Multiplication and shift left procedures are the same for both signed and unsigned numbers, hence only one function for each operation here.

The source code of these function can be founded in the installed MSVS, in VC/crt/src/intel

Appendix F

Cheatsheets

F.1 IDA

Short hot-keys cheatsheet:

key	meaning
Space	switch listing and graph view
C	convert to code
D	convert to data
A	convert to string
*	convert to array
U	undefine
O	make offset of operand
H	make decimal number
R	make char
B	make binary number
Q	make hexadecimal number
N	rename identificator
?	calculator
G	jump to address
:	add comment
Ctrl-X	show refernces to the current function, label, variable (incl. in local stack)
X	show references to the function, label, variable, etc
Alt-I	search for constant
Ctrl-I	search for the next occurrence of constant
Alt-B	search for byte sequence
Ctrl-B	search for the next occurrence of byte sequence
Alt-T	search for text (including instructions, etc)
Ctrl-T	search for the next occurrence of text
Alt-P	edit current function
Enter	jump to function, variable, etc
Esc	get back
Num -	fold function or selected area
Num +	unhide function or area

Function/area folding may be useful for hiding function parts when you realize what they do. this is used in my script[1] for hiding some often used patterns of inline code.

F.2 OllyDbg

Short hot-keys cheatsheet:

[1] https://github.com/yurichev/IDA_scripts

hot-key	meaning
F7	trace into
F8	step over
F9	run
Ctrl-F2	restart

F.3 MSVC

Some useful options I used through this book.

option	meaning
/O1	minimize space
/Ob0	no inline expansion
/Ox	maximum optimizations
/GS-	disable security checks (buffer overflows)
/Fa(file)	generate assembly listing
/Zi	enable debugging information
/Zp(n)	pack structs on n-byte boundary
/MD	produced executable will use `MSVCR*.DLL`

F.4 GCC

Some useful options I used through this book.

option	meaning
-Os	code size optimization
-O3	maximum optimization
-regparm=	how many arguments will be passed in registers
-o file	set name of output file
-g	produce debugging information in resulting executable
-S	generate assembly listing file
-masm=intel	produce listing in Intel syntax

F.5 GDB

Some of commands I used in this book:

option	meaning
break filename.c:number	set a breakpoint on line number in source code
break function	set a breakpoint on function
break *address	set a breakpoint on address
b	—”—
p variable	print value of variable
run	run
r	—”—
cont	continue execution
c	—”—
bt	print stack
set disassembly-flavor intel	set Intel syntax
disas	disassemble current function
disas function	disassemble function
disas function,+50	disassemble portion
disas $eip,+0x10	—”—
info registers	print all registers
info locals	dump local variables (if known)
x/w ...	dump memory as 32-bit word
x/w $rdi	dump memory as 32-bit word at address stored in `RDI`
x/10w ...	dump 10 memory words
x/s ...	dump memory as string
x/i ...	dump memory as code
x/10c ...	dump 10 characters
x/b ...	dump bytes
x/h ...	dump 16-bit halfwords
x/g ...	dump giant (64-bit) words
finish	execute till the end of function
next	next instruction (don’t dive into functions)
step	next instruction (dive into functions)
frame n	switch stack frame
info break	list of breakpoints
del n	delete breakpoint

Appendix G

Exercise solutions

G.1 Level 1

G.1.1 Exercise 1.1

That was a function returning maximal value from two.

G.1.2 Exercise 1.4

Source code: http://yurichev.com/RE-exercise-solutions/1/4/password1.c

G.2 Level 2

G.2.1 Exercise 2.1

Solution: `toupper()`.

C source code:

```
char toupper ( char c )
{
    if( c >= 'a' && c <= 'z' ) {
        c = c - 'a' + 'A';
    }
    return( c );
}
```

G.2.2 Exercise 2.2

Solution: `atoi()`

C source code:

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

int atoi ( const *p )  /* convert ASCII string to integer */
{
    int i;
    char s;

    while( isspace ( *p ) )
        ++p;
    s = *p;
    if( s == '+' || s == '-' )
        ++p;
    i = 0;
    while( isdigit(*p) ) {
        i = i * 10 + *p - '0';
        ++p;
    }
    if( s == '-' )
        i = - i;
    return( i );
}
```

G.2.3 Exercise 2.3

Solution: `srand()` / `rand()`.

C source code:

```
static unsigned int v;

void srand (unsigned int s)
{
        v = s;
}

int rand ()
{
        return( ((v = v * 214013L
```

```
            + 2531011L) >> 16) & 0x7fff );
}
```

G.2.4 Exercise 2.4

Solution: `strstr()`.

C source code:

```
char * strstr (
        const char * str1,
        const char * str2
        )
{
        char *cp = (char *) str1;
        char *s1, *s2;

        if ( !*str2 )
            return((char *)str1);

        while (*cp)
        {
                s1 = cp;
                s2 = (char *) str2;

                while ( *s1 && *s2 && !(*s1-*s2) )
                        s1++, s2++;

                if (!*s2)
                        return(cp);

                cp++;
        }

        return(NULL);

}
```

G.2.5 Exercise 2.5

Hint #1: Keep in mind that `__v`—global variable.

Hint #2: The function is called in CRT startup code, before `main()` execution.

Solution: early Pentium CPU FDIV bug checking[1].

C source code:

```
unsigned _v; // _v

enum e {
    PROB_P5_DIV = 0x0001
};

void f( void ) // __verify_pentium_fdiv_bug
{
    /*
        Verify we have got the Pentium FDIV problem.
        The volatiles are to scare the optimizer away.
    */
    volatile double     v1     = 4195835;
    volatile double     v2   = 3145727;

    if( (v1 - (v1/v2)*v2) > 1.0e-8 ) {
        _v |= PROB_P5_DIV;
    }
}
```

G.2.6 Exercise 2.6

Hint: it might be helpful to google a constant used here.

Solution: TEA[2] encryption algorithm.

C source code (taken from http://en.wikipedia.org/wiki/Tiny_Encryption_Algorithm):

```
void f (unsigned int* v, unsigned int* k) {
    unsigned int v0=v[0], v1=v[1], sum=0, i;           /* set up⇙
    ↳  */
    unsigned int delta=0x9e3779b9;                     /* a key ⇙
    ↳ schedule constant */
    unsigned int k0=k[0], k1=k[1], k2=k[2], k3=k[3];   /* cache ⇙
    ↳ key */
    for (i=0; i < 32; i++) {                       /* basic ⇙
    ↳ cycle start */
        sum += delta;
        v0 += ((v1<<4) + k0) ^ (v1 + sum) ^ ((v1>>5) + k1);
```

[1] http://en.wikipedia.org/wiki/Pentium_FDIV_bug

[2] Tiny Encryption Algorithm

```
        v1 += ((v0<<4) + k2) ^ (v0 + sum) ^ ((v0>>5) + k3);
    }                                                   /* end cycle ↙
    ↳ */
    v[0]=v0; v[1]=v1;
}
```

G.2.7 Exercise 2.7

Hint: the table contain pre-calculated values. It is possible to implement the function without it, but it will work slower, though.

Solution: this function reverse all bits in input 32-bit integer. It is `lib/bitrev.c` from Linux kernel.

C source code:

```
const unsigned char byte_rev_table[256] = {
        0x00, 0x80, 0x40, 0xc0, 0x20, 0xa0, 0x60, 0xe0,
        0x10, 0x90, 0x50, 0xd0, 0x30, 0xb0, 0x70, 0xf0,
        0x08, 0x88, 0x48, 0xc8, 0x28, 0xa8, 0x68, 0xe8,
        0x18, 0x98, 0x58, 0xd8, 0x38, 0xb8, 0x78, 0xf8,
        0x04, 0x84, 0x44, 0xc4, 0x24, 0xa4, 0x64, 0xe4,
        0x14, 0x94, 0x54, 0xd4, 0x34, 0xb4, 0x74, 0xf4,
        0x0c, 0x8c, 0x4c, 0xcc, 0x2c, 0xac, 0x6c, 0xec,
        0x1c, 0x9c, 0x5c, 0xdc, 0x3c, 0xbc, 0x7c, 0xfc,
        0x02, 0x82, 0x42, 0xc2, 0x22, 0xa2, 0x62, 0xe2,
        0x12, 0x92, 0x52, 0xd2, 0x32, 0xb2, 0x72, 0xf2,
        0x0a, 0x8a, 0x4a, 0xca, 0x2a, 0xaa, 0x6a, 0xea,
        0x1a, 0x9a, 0x5a, 0xda, 0x3a, 0xba, 0x7a, 0xfa,
        0x06, 0x86, 0x46, 0xc6, 0x26, 0xa6, 0x66, 0xe6,
        0x16, 0x96, 0x56, 0xd6, 0x36, 0xb6, 0x76, 0xf6,
        0x0e, 0x8e, 0x4e, 0xce, 0x2e, 0xae, 0x6e, 0xee,
        0x1e, 0x9e, 0x5e, 0xde, 0x3e, 0xbe, 0x7e, 0xfe,
        0x01, 0x81, 0x41, 0xc1, 0x21, 0xa1, 0x61, 0xe1,
        0x11, 0x91, 0x51, 0xd1, 0x31, 0xb1, 0x71, 0xf1,
        0x09, 0x89, 0x49, 0xc9, 0x29, 0xa9, 0x69, 0xe9,
        0x19, 0x99, 0x59, 0xd9, 0x39, 0xb9, 0x79, 0xf9,
        0x05, 0x85, 0x45, 0xc5, 0x25, 0xa5, 0x65, 0xe5,
        0x15, 0x95, 0x55, 0xd5, 0x35, 0xb5, 0x75, 0xf5,
        0x0d, 0x8d, 0x4d, 0xcd, 0x2d, 0xad, 0x6d, 0xed,
        0x1d, 0x9d, 0x5d, 0xdd, 0x3d, 0xbd, 0x7d, 0xfd,
        0x03, 0x83, 0x43, 0xc3, 0x23, 0xa3, 0x63, 0xe3,
        0x13, 0x93, 0x53, 0xd3, 0x33, 0xb3, 0x73, 0xf3,
        0x0b, 0x8b, 0x4b, 0xcb, 0x2b, 0xab, 0x6b, 0xeb,
        0x1b, 0x9b, 0x5b, 0xdb, 0x3b, 0xbb, 0x7b, 0xfb,
```

```
        0x07, 0x87, 0x47, 0xc7, 0x27, 0xa7, 0x67, 0xe7,
        0x17, 0x97, 0x57, 0xd7, 0x37, 0xb7, 0x77, 0xf7,
        0x0f, 0x8f, 0x4f, 0xcf, 0x2f, 0xaf, 0x6f, 0xef,
        0x1f, 0x9f, 0x5f, 0xdf, 0x3f, 0xbf, 0x7f, 0xff,
};

unsigned char bitrev8(unsigned char byte)
{
        return byte_rev_table[byte];
}

unsigned short bitrev16(unsigned short x)
{
        return (bitrev8(x & 0xff) << 8) | bitrev8(x >> 8);
}

/**
 * bitrev32 - reverse the order of bits in a unsigned int value
 * @x: value to be bit-reversed
 */

unsigned int bitrev32(unsigned int x)
{
        return (bitrev16(x & 0xffff) << 16) | bitrev16(x >> 16);
}
```

G.2.8 Exercise 2.8

Solution: two 100*200 matrices of *double* type addition.

C/C++ source code:

```
#define M    100
#define N    200

void s(double *a, double *b, double *c)
{
  for(int i=0;i<N;i++)
    for(int j=0;j<M;j++)
      *(c+i*M+j)=*(a+i*M+j) + *(b+i*M+j);
};
```

G.2.9 Exercise 2.9

Solution: two matrices (one is 100*200, second is 100*300) of *double* type multiplication, result: 100*300 matrix.

C/C++ source code:

```
#define M     100
#define N     200
#define P     300

void m(double *a, double *b, double *c)
{
  for(int i=0;i<M;i++)
    for(int j=0;j<P;j++)
    {
      *(c+i*M+j)=0;
      for (int k=0;k<N;k++) *(c+i*M+j)+=*(a+i*M+j) * *(b+i*M+j);
    }
};
```

G.2.10 Exercise 2.11

Hint: Task Manager get CPU/CPU cores count using function call `NtQuerySystemInformation(SystemBasicInformation, ..., ..., ...)`, it is possible to find that call and to substitute resulting number.

And of course, the Task Manager will show incorrect results in CPU usage history.

G.2.11 Exercise 2.12

This is a primitive cryptographic algorithm named ROT13, once popular in UseNet and mailing lists [3].

Source code: http://yurichev.com/RE-exercise-solutions/2/12/ROT13.c

G.2.12 Exercise 2.13

The cryptoalgorithm is linear feedback shift register [4].

Source code: http://yurichev.com/RE-exercise-solutions/2/13/LFSR.c

[3] https://en.wikipedia.org/wiki/ROT13

[4] https://en.wikipedia.org/wiki/Linear_feedback_shift_register

G.2.13 Exercise 2.14

This is algorithm of finding greater common divisor (GCD).

Source code: http://yurichev.com/RE-exercise-solutions/2/14/GCD.c

G.2.14 Exercise 2.15

Pi value calculation using Monte-Carlo method.

Source code: http://yurichev.com/RE-exercise-solutions/2/15/monte.c

G.2.15 Exercise 2.16

It is Ackermann function [5].

```
int ack (int m, int n)
{
        if (m==0)
                return n+1;
        if (n==0)
                return ack (m-1, 1);
        return ack(m-1, ack (m, n-1));
};
```

G.2.16 Exercise 2.17

This is 1D cellular automation working by *Rule 110*: https://en.wikipedia.org/wiki/Rule_110.

Source code: http://yurichev.com/RE-exercise-solutions/2/17/CA.c

G.2.17 Exercise 2.18

Source code: http://yurichev.com/RE-exercise-solutions/2/18/

G.2.18 Exercise 2.19

Source code: http://yurichev.com/RE-exercise-solutions/2/19/

[5] https://en.wikipedia.org/wiki/Ackermann_function

G.3 Level 3

G.3.1 Exercise 3.1

Hint #1: The code has one characteristic thing, if considering it, it may help narrowing search of right function among glibc functions.

Solution: characteristic —is callback-function calling (20), pointer to which is passed in 4th argument. It is `quicksort()`.

C source code: http://yurichev.com/RE-exercise-solutions/3/1/2_1.c

G.3.2 Exercise 3.2

Hint: easiest way is to find by values in the tables.

Commented C source code:
http://yurichev.com/RE-exercise-solutions/3/2/gost.c

G.3.3 Exercise 3.3

Commented C source code:
http://yurichev.com/RE-exercise-solutions/3/3/entropy.c

G.3.4 Exercise 3.4

Commented C source code, and also decrypted file: http://yurichev.com/RE-exercise-s
3/4/

G.3.5 Exercise 3.5

Hint: as we can see, the string with user name occupies not the whole file.

Bytes after terminated zero till offset `0x7F` are ignored by program.

Commented C source code:
http://yurichev.com/RE-exercise-solutions/3/5/crc16_keyfile_check.c

G.3.6 Exercise 3.6

Commented C source code:
http://yurichev.com/RE-exercise-solutions/3/6/

As another exercise, now you may try to fix all vulnerabilities you found in this web-server.

G.3.7 Exercise 3.8

Commented C source code:
http://yurichev.com/RE-exercise-solutions/3/8/

Acronyms used

OS Operating System xxi

FAQ Frequently Asked Questions xxi

OOP Object-Oriented Programming 354

PL Programming language 3

PRNG Pseudorandom number generator 249

RA Return Address 17

PE Portable Executable: 48.2 523

SP stack pointer. SP/ESP/RSP in x86/x64. SP in ARM. 13

DLL Dynamic-link library 523

PC Program Counter. IP/EIP/RIP in x86/64. PC in ARM. 13

LR Link Register 13

IDA Interactive Disassembler 6

IAT Import Address Table 524

INT Import Name Table 524

RVA Relative Virtual Address 524

VA Virtual Address 524

OEP Original Entry Point 518

MSVC Microsoft Visual C++

MSVS Microsoft Visual Studio 855

ASLR Address Space Layout Randomization 524

MFC Microsoft Foundation Classes 528

TLS Thread Local Storage xxi

AKA Also Known As

CRT C runtime library: sec:CRT 6

CPU Central processing unit xxi

FPU Floating-point unit 155

CISC Complex instruction set computing 14

RISC Reduced instruction set computing 14

GUI Graphical user interface 518

RTTI Run-time type information 377

BSS Block Started by Symbol 93

SIMD Single instruction, multiple data 277

BSOD Black Screen of Death 508

DBMS Database management systems xxi

ISA Instruction Set Architecture xxii

CGI Common Gateway Interface 827

HPC High-Performance Computing 321

SOC System on Chip 12

SEH Structured Exception Handling: 48.3 29

ELF Executable file format widely used in *NIX system including Linux xxi

TIB Thread Information Block 190

TEA Tiny Encryption Algorithm 875

PIC Position Independent Code: 47.1 xxi

NAN Not a Number 841

NOP No OPeration 73

BEQ (PowerPC, ARM) Branch if Equal 76

BNE (PowerPC, ARM) Branch if Not Equal 145

BLR (PowerPC) Branch to Link Register 591

XOR eXclusive OR 850

MCU Microcontroller unit 624

RAM Random-access memory 65

ROM Read-only memory 749

EGA Enhanced Graphics Adapter 749

VGA Video Graphics Array 749

API Application programming interface 471

ASCII American Standard Code for Information Interchange 725

ASCIIZ ASCII Zero (null-terminated ASCII string) 72

IA64 Intel Architecture 64 (Itanium): 66 462

EPIC Explicitly parallel instruction computing 744

OOE Out-of-order execution 744

MSDN Microsoft Developer Network xxiii

MSB Most significant bit/byte 857

STL (C++) Standard Template Library: 29.4 387

PODT (C++) Plain Old Data Type 405

HDD Hard disk drive 425

VM Virtual Memory

WRK Windows Research Kernel 490

GPR General Purpose Registers 3

SSDT System Service Dispatch Table 508

RE Reverse Engineering 757

SSE Streaming SIMD Extensions 304

BCD Binary-coded decimal 723

BOM Byte order mark 477

GDB GNU debugger 36

Bibliography

[1] AMD. AMD64 Architecture Programmer's Manual. 2013. Also available as http://developer.amd.com/resources/documentation-articles/developer-guides-manuals/.

[2] Apple. iOS ABI Function Call Guide. 2010. Also available as http://developer.apple.com/library/ios/documentation/Xcode/Conceptual/iPhoneOSABIReference/iPhoneOSABIReference.pdf.

[3] blexim. Basic integer overflows. Phrack, 2002. Also available as http://yurichev.com/mirrors/phrack/p60-0x0a.txt.

[4] Ralf Brown. The x86 interrupt list. Also available as http://www.cs.cmu.edu/~ralf/files.html.

[5] Mike Burrell. Writing effcient itanium 2 assembly code. Also available as http://yurichev.com/mirrors/RE/itanium.pdf.

[6] Marshall Cline. C++ faq. Also available as http://www.parashift.com/c++-faq-lite/index.html.

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms, Third Edition. The MIT Press, 3rd edition, 2009.

[8] Stephen Dolan. mov is turing-complete. 2013. Also available as http://www.cl.cam.ac.uk/~sd601/papers/mov.pdf.

[9] Nick Montfort et al. 10 PRINT CHR$(205.5+RND(1)); : GOTO 10. The MIT Press, 2012. Also available as http://trope-tank.mit.edu/10_PRINT_121114.pdf.

[10] Agner Fog. The microarchitecture of Intel, AMD and VIA CPUs / An optimization guide for assembly programmers and compiler makers. 2013. http://agner.org/optimize/microarchitecture.pdf.

[11] Agner Fog. Optimizing software in C++: An optimization guide for Windows, Linux and Mac platforms. 2013. http://agner.org/optimize/optimizing_cpp.pdf.

[12] Agner Fog. Calling conventions. 2014. http://www.agner.org/optimize/calling_conventions.pdf.

[13] IBM. PowerPC(tm) Microprocessor Family: The Programming Environments for 32-Bit Microprocessors. 2000. Also available as http://yurichev.com/mirrors/PowerPC/6xx_pem.pdf.

[14] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes:1, 2A, 2B, 2C, 3A, 3B, and 3C. 2013. Also available as http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf.

[15] ISO. ISO/IEC 9899:TC3 (C C99 standard). 2007. Also available as http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf.

[16] ISO. ISO/IEC 14882:2011 (C++ 11 standard). 2013. Also available as http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3690.pdf.

[17] Brian W. Kernighan. The C Programming Language. Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[18] Donald E. Knuth. The Art of Computer Programming Volumes 1-3 Boxed Set. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1998.

[19] Eugene Loh. The ideal hpc programming language. Queue, 8(6):30:30–30:38, June 2010.

[20] Advanced RISC Machines Ltd. The ARM Cookbook. 1994. Also available as http://yurichev.com/ref/ARM%20Cookbook%20(1994).

[21] Michael Matz / Jan Hubicka / Andreas Jaeger / Mark Mitchell. System v application binary interface. amd64 architecture processor supplement, 2013. Also available as http://x86-64.org/documentation/abi.pdf.

[22] Aleph One. Smashing the stack for fun and profit. Phrack, 1996. Also available as http://yurichev.com/mirrors/phrack/p49-0x0e.txt.

[23] Matt Pietrek. A crash course on the depths of win32™ structured exception handling. MSDN magazine.

[24] Matt Pietrek. An in-depth look into the win32 portable executable file format. MSDN magazine, 2002.

[25] Eric S. Raymond. The Art of UNIX Programming. Pearson Education, 2003. Also available as http://catb.org/esr/writings/taoup/html/.

[26] D. M. Ritchie and K. Thompson. The unix time sharing system. 1974. Also available as http://dl.acm.org/citation.cfm?id=361061.

[27] Dennis M. Ritchie. The evolution of the unix time-sharing system. 1979.

[28] Dennis M. Ritchie. Where did ++ come from? (net.lang.c). http://yurichev.com/mirrors/C/c_dmr_postincrement.txt, 1986. [Online; accessed 2013].

[29] Dennis M. Ritchie. The development of the c language. SIGPLAN Not., 28(3):201–208, March 1993. Also available as http://yurichev.com/mirrors/C/dmr-The%20Development%20of%20the%20C%20Language-1993.pdf.

[30] Mark E. Russinovich and David A. Solomon with Alex Ionescu. Windows® Internals: Including Windows Server 2008 and Windows Vista, Fifth Edition. 2009.

[31] Bruce Schneier. Applied Cryptography: Protocols, Algorithms, and Source Code in C. 1994.

[32] Igor Skochinsky. Compiler internals: Exceptions and rtti, 2012. Also available as http://yurichev.com/mirrors/RE/Recon-2012-Skochinsky-Compiler-Internals.pdf.

[33] SunSoft Steve Zucker and IBM Kari Karhi. SYSTEM V APPLICATION BINARY INTERFACE: PowerPC Processor Supplement. 1995. Also available as http://yurichev.com/mirrors/PowerPC/elfspec_ppc.pdf.

[34] trew. Introduction to reverse engineering win32 applications. uninformed. Also available as http://yurichev.com/mirrors/RE/uninformed_v1a7.pdf.

[35] Henry S. Warren. Hacker's Delight. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[36] Dennis Yurichev. Finding unknown algorithm using only input/output pairs and z3 smt solver. 2012. Also available as http://yurichev.com/writings/z3_rockey.pdf.

[37] Dennis Yurichev. C/C++ programming language notes. 2013. Also available as http://yurichev.com/writings/C-notes-en.pdf.

Glossary

decrement Decrease by 1. 13, 125, 139, 145, 494, 662, 845, 848, 855

increment Increase by 1. 14, 125, 130, 139, 143, 145, 625, 657, 844, 845

product Multiplication result. 278

stack pointer A register pointing to the place in the stack.. 6, 7, 13, 14, 21, 25, 31, 45, 47, 59, 80, 356, 442, 497, 499–501, 838, 846, 864, 882

tail call It is when compiler (or interpreter) transforms recursion (with which it is possible: *tail recursion*) into iteration for efficiency: `http://en.wikipedia.org/wiki/Tail_call`. 20

quotient Division result. 212, 582

anti-pattern Generally considered as bad practice. 24, 59

atomic operation "$\alpha\tau o\mu o\varsigma$" mean "indivisible" in Greek, so atomic operation is what guaranteed not to be broke up during operation by other threads. 568, 740

basic block a group of instructions not having jump/branch instructions, and also not having jumps inside block from the outside. In IDA it looks just like as a list of instructions without breaking empty lines . 751, 753

callee A function being called by another. 19, 24, 25, 67, 77, 80, 84, 114, 292, 357, 442, 497, 499–501, 504, 506

caller A function calling another. 6, 67, 77, 78, 82, 89, 114, 292, 302, 310, 357, 497, 498, 500, 501

compiler intrinsic A function specific to a compiler which is not usual library function. Compiler generate a specific machine code instead of call to it. It is often a pseudofunction for specific CPU instruction. Read more: (63). 855

CP/M Control Program for Microcomputers: a very basic disk OS used before MS-DOS. 720

debuggee A program being debugged. 184

dongle Dongle is a small piece of hardware connected to LPT printer port (in past) or to USB. Its function was akin to security token, it has some memory and, sometimes, secret (crypto-)hashing algorithm.. 590

endianness Byte order: 33. 17, 61, 851

GiB Gibibyte: 2^{30} or 1024 mebibytes or 1073741824 bytes. 11

heap usually, a big chunk of memory provided by OS so that applications can divide it by themselves as they wish. malloc()/free() works with heap.. 22, 25, 223, 381, 385, 405, 408, 522, 524

jump offset a part of JMP or Jcc instruction opcode, it just to be added to the address of the next instruction, and thus is how new PC is calculated. May be negative as well.. 73, 104, 845

kernel mode A restrictions-free CPU mode in which it executes OS kernel and drivers. cf. user mode.. 892

keygenme A program which imitates fictional software protection, for which one needs to make a keys/licenses generator. 829

leaf function A function which is not calling any other function. 24

link register (RISC) A register where return address is usually stored. This makes calling leaf functions without stack usage, i.e., faster.. 24, 592, 864

loop unwinding It is when a compiler instead of generation loop code of n iteration, generates just n copies of the loop body, in order to get rid of loop maintenance instructions. 129

name mangling used at least in C++, where compiler need to encode name of class, method and argument types in the one string, which will become internal name of the function. read more here: 29.1.1. 354, 465, 466

NaN not a number: special cases of floating point numbers, usually signaling about errors . 169, 748

NEON AKA “Advanced SIMD”—SIMD from ARM. 865

NOP “no operation”, idle instruction. 494

POKE BASIC language instruction writing byte on specific address. 494

register allocator Compiler's function assigning local variables to CPU registers. 138, 202, 293

reverse engineering act of understanding, how the thing works, sometimes, in order to clone it. xxi, xxii, 855

security cookie A random value, different at each execution. Read more about it: 16.3. 553

stack frame Part of stack containing information specific to the current functions: local variables, function arguments, RA, etc. 53, 78, 79, 553

thunk function Tiny function with a single role: call another function.. 18, 261, 591, 605

tracer My own simple debugging tool. Read more about it: 50.1. 131, 132, 184, 473, 486, 491, 492, 546, 561, 694, 704, 711, 712, 716, 731, 826, 827

user mode A restricted CPU mode in which it executes all applied software code. cf. kernel mode.. 617, 891

Windows NT Windows NT, 2000, XP, Vista, 7, 8. 289, 440, 478, 508, 525, 567, 725, 855

xoring often used in English language, meaning applying XOR operation. 553, 609, 614

Index

.NET, 531
AT&T syntax, 9, 28
Buffer Overflow, 182, 553
C language elements
    Pointers, 51, 58, 91, 253, 292
    Post-decrement, 144
    Post-increment, 144
    Pre-decrement, 144
    Pre-increment, 144
    C99, 89
        bool, 199
        restrict, 318
        variable length arrays, 193
    const, 5, 65
    for, 125, 217
    if, 98, 113
    restrict, 318
    return, 6, 67, 89
    switch, 111, 113, 114
    while, 136
C standard library
    alloca(), 26, 193
    assert(), 481
    atexit(), 389
    atoi(), 873
    calloc(), 645
    close(), 514
    localtime(), 455
    longjmp(), 114
    malloc(), 224
    memchr(), 849
    memcmp(), 484, 851
    memcpy(), 9, 52, 847
    memset(), 712, 850
    open(), 514
    qsort(), 253, 880
    rand(), 471, 624, 873
    read(), 514
    scanf(), 51
    srand(), 873
    strcmp(), 515
    strcpy(), 9, 625
    strlen(), 136, 287, 849
    strstr(), 874
    time(), 455
    tolower(), 654
    toupper(), 872
Compiler's anomalies, 209, 733
C++, 696
    C++11, 405, 507
    ostream, 377
    References, 379
    STL
        std::forward_list, 405
        std::list, 390
        std::map, 417
        std::set, 417
        std::string, 380
        std::vector, 405
grep usage, 132, 173, 464, 486, 491, 692
Intel syntax, 9, 12
Mac OS X, 573
position-independent code, 13, 510
RAM, 65
ROM, 65
Recursion, 19, 23

Tail recursion, 19
Stack, 21, 77, 113
Syntactic Sugar, 113, 231
iPod/iPhone/iPad, 11
OllyDbg, 31, 54, 61, 92, 101, 130, 140, 257, 572, 869
Oracle RDBMS, 6, 278, 480, 535, 697, 710, 713, 733, 751

8080, 144
8086, 617
8253, 723
80286, 617, 750
80386, 750

Angry Birds, 172, 173
ARM, 144, 345, 351, 591
ARM mode, 12
Instructions
ADD, 16, 107, 115, 133, 150, 213
ADDAL, 107
ADDCC, 121
ADDS, 86, 115
ADR, 13, 107
ADREQ, 108, 115
ADRGT, 107
ADRHI, 108
ADRNE, 115
ASRS, 150, 208
B, 45, 107, 109
BCS, 109, 175
BEQ, 76, 115
BGE, 109
BIC, 208
BL, 13, 15, 18, 108
BLE, 109
BLEQ, 108
BLGT, 108
BLHI, 108
BLS, 109
BLT, 133
BLX, 17
BNE, 109
BX, 86, 124
CLZ, 800, 801
CMP, 76, 107, 115, 121, 133, 213
IDIV, 147
IT, 172, 192
LDMCSFD, 108
LDMEA, 21
LDMED, 21
LDMFA, 21
LDMFD, 14, 21, 108
LDMGEFD, 108
LDR, 47, 59, 180
LDR.W, 198
LDRB, 237
LDRB.W, 144
LDRSB, 143
LSL, 213
LSL.W, 213
LSLS, 181
MLA, 85
MOV, 14, 149, 213
MOVT, 15, 149
MOVT.W, 17
MOVW, 17
MULS, 86
MVNS, 145
ORR, 208
POP, 13, 14, 21, 24
PUSH, 21, 24
RSB, 197, 213
SMMUL, 150
STMEA, 21
STMED, 21
STMFA, 21, 49
STMFD, 13, 21
STMIA, 47
STMIB, 49
STR, 47, 180
SUB, 47, 197, 213
SUBEQ, 146
SXTB, 238
TEST, 138
TST, 205, 213
VADD, 160

VDIV, 160
VLDR, 159
VMOV, 159, 171
VMOVGT, 171
VMRS, 171
VMUL, 160
Registers
APSR, 171
FPSCR, 171
Link Register, 13, 14, 24, 45, 124, 864
R0, 87, 863
scratch registers, 144, 864
Z, 76, 864
thumb mode, 12, 109, 124
thumb-2 mode, 12, 124, 172, 173
armel, 161
armhf, 161
Condition codes, 107
Data processing instructions, 150
DCB, 14
hard float, 161
if-then block, 172
Leaf function, 24
Optional operators
ASR, 150, 213
LSL, 180, 197, 213
LSR, 150, 213
ROR, 213
RRX, 213
soft float, 161
ASLR, 524
AWK, 489

bash, 88
BASIC
POKE, 494
binary grep, 485, 575
BIND.EXE, 530
Bitcoin, 735
Borland C++Builder, 466
Borland Delphi, 466, 476, 730
BSoD, 508
BSS, 526

C11, 507
Callbacks, 253
Canary, 187
cdecl, 31, 497
COFF, 601
column-major order, 194
Compiler intrinsic, 27, 732
CRC32, 214, 577
CRT, 518, 547
Cygwin, 466
cygwin, 473, 532, 573

DES, 277, 293
dlopen(), 515
dlsym(), 515
DOSBox, 725
DosBox, 492
double, 156, 505
dtruss, 573

EICAR, 719
ELF, 62
Error messages, 480

fastcall, 10, 50, 201, 499
float, 156, 505
FORTRAN, 194, 318, 466
Function epilogue, 19, 45, 47, 108, 237, 489
Function prologue, 7, 19, 24, 47, 187, 489
Fused multiply–add, 85

GCC, 466, 866, 870
GDB, 36, 41, 186, 262, 263, 572, 870
Glibc, 262

Hex-Rays, 578
Hiew, 72, 103, 475, 527, 532, 731

IAT, 524
IDA, 67, 478, 868
var_?, 47, 58
IEEE 754, 155, 249, 304, 834
Inline code, 135, 208, 322, 363
INT, 524
int 0x2e, 509

int 0x80, 508
Intel C++, 6, 278, 733, 751, 847
Itanium, 744

jumptable, 118, 124

Keil, 11
kernel panic, 508
kernel space, 508

LD_PRELOAD, 514
Linux, 697
    libc.so.6, 201, 261
LLVM, 11
long double, 156
Loop unwinding, 128

Mac OS Classic, 590
MD5, 483, 577
MFC, 528
MIDI, 484
MinGW, 466
MIPS, 348, 351, 526, 591
MS-DOS, 190, 484, 492, 494, 523, 617, 719, 721, 730, 834, 855
    DOS extenders, 750
MSVC, 867, 870

Name mangling, 354
NEC V20, 724

objdump, 513, 532
OEP, 523, 531
opaque predicate, 437
OpenMP, 470, 735
OpenWatcom, 466, 501, 762, 764, 777
Ordinal, 528

Page (memory), 289
Pascal, 476
PDB, 464, 526, 690
PDP-11, 144
PowerPC, 590

Raspberry Pi, 11, 161
ReactOS, 543
Register allocation, 293
Relocation, 17
row-major order, 194
RTTI, 377
RVA, 524

SAP, 464, 690
SCO OpenServer, 601
Scratch space, 503
Security cookie, 187, 553
SHA1, 577
SHA512, 735
Shadow space, 82, 84, 305
Shellcode, 720, 860
shellcode, 436, 508, 524
Signed numbers, 100, 459
SIMD, 304
SSE, 304
SSE2, 304
stdcall, 497, 730
strace, 514, 573
syscall, 508
syscalls, 201, 573

TCP/IP, 462
thiscall, 354, 356, 501
ThumbTwoMode, 17
TLS, 190, 507, 526, 531, 839
    Callbacks, 531
tracer, 131, 184, 258, 260, 473, 486, 491, 546, 561, 572, 694, 704, 711, 712, 715, 731, 826, 827

Unicode, 476
Unrolled loop, 135, 192
uptime, 514
user space, 508
UTF-16LE, 476, 478
UTF-8, 476

VA, 524

Watcom, 466
Win32, 478, 750

RaiseException(), 532
SetUnhandledExceptionFilter(), 535
Windows
GetProcAddress, 531
KERNEL32.DLL, 200
LoadLibrary, 531
MSVCR80.DLL, 256
ntoskrnl.exe, 697
Structured Exception Handling, 29, 532
TIB, 190, 532, 839
Windows 2000, 525
Windows NT4, 525
Windows Vista, 523
Windows XP, 525, 531
Windows 3.x, 440, 750
Windows API, 834
Wine, 543
Wolfram Mathematica, 153, 154, 581

x86
Instructions
AAA, 862
AAS, 862
ADC, 269, 447, 844
ADD, 6, 31, 78, 447, 845
ADDSD, 305
ADDSS, 309
AND, 7, 200, 206, 210, 243, 845
BSF, 291, 802, 851
BSR, 851
BSWAP, 462, 851
BT, 851
BTC, 851
BTR, 568, 851
BTS, 851
CALL, 5, 22, 529, 845
CBW, 851
CDQ, 275, 851
CDQE, 851
CLD, 851
CLI, 851
CMC, 851
CMOVcc, 109, 760, 851
CMP, 67, 845, 862
CMPSB, 484, 851
CMPSD, 851
CMPSQ, 851
CMPSW, 851
COMISD, 308
COMISS, 309
CPUID, 239, 854
CWD, 447, 851
CWDE, 851
DEC, 139, 845, 862
DIV, 854
DIVSD, 305, 488
FABS, 858
FADD, 858
FADDP, 158, 159, 858
FCHS, 858
FCOM, 167, 169, 858
FCOMP, 165, 858
FCOMPP, 858
FDIV, 157, 487, 858, 874
FDIVP, 157, 858
FDIVR, 159, 858
FDIVRP, 858
FILD, 858
FIST, 859
FISTP, 859
FLD, 162, 165, 859
FLD1, 859
FLDCW, 859
FLDZ, 859
FMUL, 157, 859
FMULP, 859
FNSTCW, 859
FNSTSW, 165, 169, 859
FSINCOS, 859
FSQRT, 859
FST, 859
FSTCW, 859
FSTP, 162, 859
FSTSW, 859
FSUB, 859
FSUBP, 859
FSUBR, 859

FSUBRP, 859
FUCOM, 169, 860
FUCOMP, 860
FUCOMPP, 168, 860
FWAIT, 155
FXCH, 860
IDIV, 854
IMUL, 78, 845, 862
IN, 617, 723, 855
INC, 139, 845, 862
INT, 720, 855
IRET, 855
JA, 100, 459, 845, 862
JAE, 100, 845, 862
JB, 100, 459, 845, 862
JBE, 100, 845, 862
JC, 845
JCXZ, 845
JE, 113, 845, 862
JECXZ, 845
JG, 100, 459, 845
JGE, 100, 845
JL, 100, 459, 845
JLE, 99, 845
JMP, 22, 45, 530, 845
JNA, 845
JNAE, 845
JNB, 845
JNBE, 169, 845
JNC, 845
JNE, 67, 99, 845, 862
JNG, 845
JNGE, 845
JNL, 845
JNLE, 845
JNO, 845, 862
JNS, 845, 862
JNZ, 845
JO, 845, 862
JP, 166, 724, 845, 862
JPO, 845
JRCXZ, 845
JS, 845, 862
JZ, 76, 113, 733, 845
LAHF, 846
LEA, 53, 81, 219, 228, 846
LEAVE, 7, 846
LES, 625
LOCK, 567
LODSB, 724
LOOP, 125, 489, 761, 855
MAXSD, 308
MOV, 6, 9, 527, 848
MOVDQA, 283
MOVDQU, 283
MOVSB, 847
MOVSD, 306, 652, 847
MOVSDX, 306
MOVSQ, 847
MOVSS, 309
MOVSW, 847
MOVSX, 137, 143, 237, 238, 848
MOVZX, 138, 225, 591, 848
MUL, 848
MULSD, 305
NEG, 848
NOP, 219, 727, 848
NOT, 142, 145, 659, 848
OR, 206, 848
OUT, 617, 855
PADDD, 283
PCMPEQB, 290
PLMULHW, 278
PLMULLD, 278
PMOVMSKB, 290
POP, 6, 21, 23, 848, 862
POPA, 855, 862
POPCNT, 856
POPF, 723, 856
PUSH, 5, 7, 21, 22, 53, 848, 862
PUSHA, 856, 862
PUSHF, 856
PXOR, 290
RCL, 489, 856
RCR, 856
RET, 6, 23, 187, 356, 848

ROL, 732, 856
ROR, 732, 856
SAHF, 169, 848
SAL, 857
SALC, 724
SAR, 857
SBB, 270, 848
SCASB, 724, 849
SCASD, 849
SCASQ, 849
SCASW, 849
SETALC, 724
SETcc, 169, 857
SETNBE, 169
SETNZ, 139
SHL, 178, 209, 849
SHR, 212, 243, 849
SHRD, 274, 850
STC, 857
STD, 857
STI, 858
STOSB, 850
STOSD, 850
STOSQ, 850
STOSW, 850
SUB, 6, 7, 67, 113, 850
SYSCALL, 855, 858
SYSENTER, 509, 855, 858
TEST, 137, 200, 205, 850
UD2, 858
XADD, 569
XCHG, 850
XOR, 6, 67, 142, 489, 609, 850, 862
Registers
Flags, 67
EAX, 67, 87
EBP, 53, 78
ECX, 354
ESP, 31, 53
JMP, 119
RIP, 513
ZF, 67, 200
8086, 144, 207
80386, 207
80486, 155
AVX, 277
FPU, 155, 840
MMX, 277
SSE, 277
SSE2, 277
x86-64, 9, 10, 39, 51, 56, 74, 80, 292, 304, 334, 501, 513, 834, 842
Xcode, 11

Z3, 583