

---

# My services

The book you currently see is [free](#) and is [available in open source form](#). But sometimes I need to do something for money, so sorry in advance for placing my advertisement right here.

## Reverse engineering

I can't accept full-time job offers, I mostly work remotely on small tasks, like these:

### Decrypting a database, managing unknown type of files

Due to NDA agreement, I can't reveal many details about the last case, but the case in [8.7 on page 861](#) section is heavily based on a real case.

### Rewriting some kind of old EXE or DLL file back to C/C++

### Dongles

Occasionally I do [software copy-protection dongle](#) replacements or dongle emulators. In general, it is somewhat unlawful to break software protection, so I can do this only if these conditions are met:

- software company who developed the software product does not exist anymore to my best knowledge;
- the software product is older than 10 years;
- you have a dongle to read information from it. In other words, I can only help to those who still uses some very old software, completely satisfied with it, but afraid of dongle electrical breakage and there are no company who can still sell the dongle replacement.

These includes ancient MS-DOS and UNIX software. Software for exotic computer architectures (like MIPS, DEC Alpha, PowerPC) accepted as well.

Examples of my work you may find here:

- My book devoted to reverse engineering has a part about copy-protection dongles: [8.5](#).
- [Finding unknown algorithm using only input/output pairs and Z3 SMT solver article](#)
- [About MicroPhar \(93c46-based dongle\) emulation in DosBox](#).
- [Source code of DOS MicroPhar emulator using EMM386 I/O interception API](#)

## Contact me

E-Mail: [dennis\(a\)yurichev.com](mailto:dennis(a)yurichev.com).

Still want to hire reverse engineer/security researcher on full-time basis?

You may try [Reddit RE hiring thread](#). There is also Russian-speaking forum with a [section devoted to RE jobs](#).

# Reverse Engineering for Beginners



Dennis Yurichev

---

# Reverse Engineering for Beginners

Dennis Yurichev

`<dennis(a)yurichev.com>`



©2013-2016, Dennis Yurichev.

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International (CC BY-SA 4.0) license. To view a copy of this license, visit <https://creativecommons.org/licenses/by-sa/4.0/>.

Text version (February 11, 2017).

The latest version (and Russian edition) of this text is accessible at [beginners.re](http://beginners.re).

The cover was made by Andy Nечаевский: [facebook](#).

---

# Call for translators!

You may want to help me with translation this work into languages other than English and Russian. Just send me any piece of translated text (no matter how short) and I'll put it into my LaTeX source code.

[Read here.](#)

Speed isn't important, because this is open-source project, after all. Your name will be mentioned as project contributor. Korean, Chinese and Persian languages are reserved by publishers. English and Russian versions I do by myself, but my English is still that horrible, so I'm very grateful for any notes about grammar, etc. Even my Russian is also flawed, so I'm grateful for notes about Russian text as well!

So do not hesitate to contact me: [dennis\(a\)yurichev.com](mailto:dennis(a)yurichev.com).

# Abridged contents

<b>1 Code patterns</b>	<b>1</b>
<b>2 Important fundamentals</b>	<b>448</b>
<b>3 Slightly more advanced examples</b>	<b>467</b>
<b>4 Java</b>	<b>665</b>
<b>5 Finding important/interesting stuff in the code</b>	<b>704</b>
<b>6 OS-specific</b>	<b>732</b>
<b>7 Tools</b>	<b>787</b>
<b>8 Case studies</b>	<b>791</b>
<b>9 Examples of reversing proprietary file formats</b>	<b>925</b>
<b>10 Other things</b>	<b>985</b>
<b>11 Books/blogs worth reading</b>	<b>993</b>
<b>12 Communities</b>	<b>996</b>
<b>Afterword</b>	<b>998</b>
<b>Appendix</b>	<b>1000</b>
<b>Acronyms used</b>	<b>1029</b>
<b>Glossary</b>	<b>1034</b>
<b>Index</b>	<b>1036</b>

# Contents

<b>1</b>	<b>Code patterns</b>	<b>1</b>
1.1	The method . . . . .	1
1.2	Some basics . . . . .	2
1.2.1	A short introduction to the CPU	2
1.2.2	Numeral systems . . . . .	3
1.3	Empty function . . . . .	5
1.3.1	x86 . . . . .	5
1.3.2	ARM . . . . .	6
1.3.3	MIPS . . . . .	6
1.3.4	Empty functions in practice . . . . .	6
1.4	Returning value . . . . .	7
1.4.1	x86 . . . . .	7
1.4.2	ARM . . . . .	7
1.4.3	MIPS . . . . .	8
1.4.4	In practice . . . . .	8
1.5	Hello, world! . . . . .	8
1.5.1	x86 . . . . .	9
1.5.2	x86-64 . . . . .	14
1.5.3	GCC—one more thing . . . . .	18
1.5.4	ARM . . . . .	19
1.5.5	MIPS . . . . .	25
1.5.6	Conclusion . . . . .	30
1.5.7	Exercises . . . . .	30
1.6	Function prologue and epilogue . . . . .	30
1.6.1	Recursion . . . . .	30
1.7	Stack . . . . .	30
1.7.1	Why does the stack grow backwards? . . . . .	31
1.7.2	What is the stack used for? . . . . .	31
1.7.3	A typical stack layout . . . . .	38
1.7.4	Noise in stack . . . . .	38
1.7.5	Exercises . . . . .	42
1.8	printf() with several arguments . . . . .	42
1.8.1	x86 . . . . .	42
1.8.2	ARM . . . . .	53
1.8.3	MIPS . . . . .	59
1.8.4	Conclusion . . . . .	65
1.8.5	By the way . . . . .	66
1.9	scanf() . . . . .	66
1.9.1	Simple example . . . . .	66
1.9.2	Popular mistake . . . . .	75
1.9.3	Global variables . . . . .	76
1.9.4	scanf() . . . . .	85
1.9.5	Exercise . . . . .	97
1.10	Accessing passed arguments . . . . .	98
1.10.1	x86 . . . . .	98
1.10.2	x64 . . . . .	100
1.10.3	ARM . . . . .	103
1.10.4	MIPS . . . . .	106
1.11	More about results returning . . . . .	107
1.11.1	Attempt to use the result of a function returning void . . . . .	107
1.11.2	What if we do not use the function result? . . . . .	109
1.11.3	Returning a structure . . . . .	109

## CONTENTS

1.12 Pointers . . . . .	110
1.12.1 Swap input values . . . . .	110
1.12.2 Returning values . . . . .	111
1.13 GOTO operator . . . . .	121
1.13.1 Dead code . . . . .	124
1.13.2 Exercise . . . . .	125
1.14 Conditional jumps . . . . .	125
1.14.1 Simple example . . . . .	125
1.14.2 Calculating absolute value . . . . .	142
1.14.3 Ternary conditional operator . . . . .	144
1.14.4 Getting minimal and maximal values . . . . .	147
1.14.5 Conclusion . . . . .	152
1.14.6 Exercise . . . . .	153
1.15 switch() / case / default . . . . .	154
1.15.1 Small number of cases . . . . .	154
1.15.2 A lot of cases . . . . .	167
1.15.3 When there are several case statements in one block . . . . .	179
1.15.4 Fall-through . . . . .	183
1.15.5 Exercises . . . . .	184
1.16 Loops . . . . .	185
1.16.1 Simple example . . . . .	185
1.16.2 Memory blocks copying routine . . . . .	196
1.16.3 Conclusion . . . . .	199
1.16.4 Exercises . . . . .	200
1.17 More about strings . . . . .	201
1.17.1 strlen() . . . . .	201
1.17.2 Boundaries of strings . . . . .	212
1.18 Replacing arithmetic instructions to other ones . . . . .	212
1.18.1 Multiplication . . . . .	212
1.18.2 Division . . . . .	217
1.18.3 Exercise . . . . .	218
1.19 Floating-point unit . . . . .	218
1.19.1 IEEE 754 . . . . .	218
1.19.2 x86 . . . . .	218
1.19.3 ARM, MIPS, x86/x64 SIMD . . . . .	219
1.19.4 C/C++ . . . . .	219
1.19.5 Simple example . . . . .	219
1.19.6 Passing floating point numbers via arguments . . . . .	230
1.19.7 Comparison example . . . . .	233
1.19.8 Some constants . . . . .	267
1.19.9 Copying . . . . .	267
1.19.10 Stack, calculators and reverse Polish notation . . . . .	267
1.19.11 x64 . . . . .	267
1.19.12 Exercises . . . . .	267
1.20 Arrays . . . . .	268
1.20.1 Simple example . . . . .	268
1.20.2 Buffer overflow . . . . .	275
1.20.3 Buffer overflow protection methods . . . . .	283
1.20.4 One more word about arrays . . . . .	286
1.20.5 Array of pointers to strings . . . . .	287
1.20.6 Multidimensional arrays . . . . .	293
1.20.7 Pack of strings as a two-dimensional array . . . . .	300
1.20.8 Conclusion . . . . .	304
1.21 By the way . . . . .	304
1.21.1 Exercises . . . . .	304
1.22 Manipulating specific bit(s) . . . . .	304
1.22.1 Specific bit checking . . . . .	304
1.22.2 Setting and clearing specific bits . . . . .	308
1.22.3 Shifts . . . . .	317
1.22.4 Setting and clearing specific bits: FPU <sup>1</sup> example . . . . .	317
1.22.5 Counting bits set to 1 . . . . .	322
1.22.6 Conclusion . . . . .	337
1.22.7 Exercises . . . . .	339

<sup>1</sup>Floating-point unit

## CONTENTS

1.23	Linear congruential generator . . . . .	339
1.23.1	x86 . . . . .	340
1.23.2	x64 . . . . .	341
1.23.3	32-bit ARM . . . . .	341
1.23.4	MIPS . . . . .	342
1.23.5	Thread-safe version of the example . . . . .	344
1.24	Structures . . . . .	345
1.24.1	MSVC: SYSTEMTIME example . . . . .	345
1.24.2	Let's allocate space for a structure using malloc() . . . . .	349
1.24.3	UNIX: struct tm . . . . .	351
1.24.4	Fields packing in structure . . . . .	360
1.24.5	Nested structures . . . . .	367
1.24.6	Bit fields in a structure . . . . .	370
1.24.7	Exercises . . . . .	377
1.25	Unions . . . . .	377
1.25.1	Pseudo-random number generator example . . . . .	377
1.25.2	Calculating machine epsilon . . . . .	381
1.26	FSCALE replacement . . . . .	383
1.26.1	Fast square root calculation . . . . .	385
1.27	Pointers to functions . . . . .	385
1.27.1	MSVC . . . . .	386
1.27.2	GCC . . . . .	393
1.27.3	Danger of pointers to functions . . . . .	397
1.28	64-bit values in 32-bit environment . . . . .	397
1.28.1	Returning of 64-bit value . . . . .	397
1.28.2	Arguments passing, addition, subtraction . . . . .	398
1.28.3	Multiplication, division . . . . .	401
1.28.4	Shifting right . . . . .	405
1.28.5	Converting 32-bit value into 64-bit one . . . . .	406
1.29	SIMD . . . . .	407
1.29.1	Vectorization . . . . .	408
1.29.2	SIMD <code>strlen()</code> implementation . . . . .	417
1.30	64 bits . . . . .	421
1.30.1	x86-64 . . . . .	421
1.30.2	ARM . . . . .	427
1.30.3	Float point numbers . . . . .	428
1.30.4	64-bit architecture criticism . . . . .	428
1.31	Working with floating point numbers using SIMD . . . . .	428
1.31.1	Simple example . . . . .	428
1.31.2	Passing floating point number via arguments . . . . .	436
1.31.3	Comparison example . . . . .	437
1.31.4	Calculating machine epsilon: x64 and SIMD . . . . .	439
1.31.5	Pseudo-random number generator example revisited . . . . .	440
1.31.6	Summary . . . . .	440
1.32	ARM-specific details . . . . .	441
1.32.1	Number sign (#) before number . . . . .	441
1.32.2	Addressing modes . . . . .	441
1.32.3	Loading a constant into a register . . . . .	442
1.32.4	Relocs in ARM64 . . . . .	444
1.33	MIPS-specific details . . . . .	445
1.33.1	Loading a 32-bit constant into register . . . . .	445
1.33.2	Further reading about MIPS . . . . .	447
<b>2</b>	<b>Important fundamentals</b> . . . . .	<b>448</b>
2.1	Integral datatypes . . . . .	449
2.1.1	Bit . . . . .	449
2.1.2	Nibble AKA nybble . . . . .	449
2.1.3	Byte . . . . .	450
2.1.4	Wide char . . . . .	450
2.1.5	Signed integer vs unsigned . . . . .	451
2.1.6	Word . . . . .	451
2.1.7	Address register . . . . .	452
2.1.8	Numbers . . . . .	452
2.2	Signed number representations . . . . .	454

## CONTENTS

2.2.1 Using IMUL over MUL . . . . .	456
2.2.2 Couple of additions about two's complement form . . . . .	456
2.3 AND . . . . .	457
2.3.1 Checking if a value is on $2^n$ boundary . . . . .	457
2.3.2 KOI-8R Cyrillic encoding . . . . .	458
2.4 AND and OR as subtraction and addition . . . . .	459
2.4.1 ZX Spectrum ROM text strings . . . . .	459
2.5 XOR (exclusive OR) . . . . .	461
2.5.1 Everyday speech . . . . .	461
2.5.2 Encryption . . . . .	461
2.5.3 RAID <sup>2</sup> 4 . . . . .	461
2.5.4 XOR swap algorithm . . . . .	461
2.5.5 XOR linked list . . . . .	462
2.5.6 AND/OR/XOR as MOV . . . . .	462
2.6 Population count . . . . .	463
2.7 Endianness . . . . .	463
2.7.1 Big-endian . . . . .	463
2.7.2 Little-endian . . . . .	463
2.7.3 Example . . . . .	464
2.7.4 Bi-endian . . . . .	464
2.7.5 Converting data . . . . .	464
2.8 Memory . . . . .	464
2.9 CPU . . . . .	465
2.9.1 Branch predictors . . . . .	465
2.9.2 Data dependencies . . . . .	465
2.10 Hash functions . . . . .	465
2.10.1 How do one-way functions work? . . . . .	466
<b>3 Slightly more advanced examples</b> . . . . .	<b>467</b>
3.1 Double negation . . . . .	467
3.2 strstr() example . . . . .	468
3.3 Temperature converting . . . . .	468
3.3.1 Integer values . . . . .	468
3.3.2 Floating-point values . . . . .	470
3.4 Fibonacci numbers . . . . .	472
3.4.1 Example #1 . . . . .	473
3.4.2 Example #2 . . . . .	476
3.4.3 Summary . . . . .	480
3.5 CRC32 calculation example . . . . .	481
3.6 Network address calculation example . . . . .	484
3.6.1 calc_network_address() . . . . .	485
3.6.2 form_IP() . . . . .	485
3.6.3 print_as_IP() . . . . .	487
3.6.4 form_netmask() and set_bit() . . . . .	488
3.6.5 Summary . . . . .	489
3.7 Loops: several iterators . . . . .	489
3.7.1 Three iterators . . . . .	489
3.7.2 Two iterators . . . . .	490
3.7.3 Intel C++ 2011 case . . . . .	492
3.8 Duff's device . . . . .	493
3.8.1 Should one use unrolled loops? . . . . .	496
3.9 Division using multiplication . . . . .	496
3.9.1 x86 . . . . .	496
3.9.2 How it works . . . . .	497
3.9.3 ARM . . . . .	498
3.9.4 MIPS . . . . .	499
3.9.5 Exercise . . . . .	499
3.10 String to number conversion (atoi()) . . . . .	499
3.10.1 Simple example . . . . .	500
3.10.2 A slightly advanced example . . . . .	503
3.10.3 Exercise . . . . .	506
3.11 Inline functions . . . . .	506
3.11.1 Strings and memory functions . . . . .	507

<sup>2</sup>Redundant Array of Independent Disks

## CONTENTS

3.12C99 restrict . . . . .	515
3.13Branchless <i>abs()</i> function . . . . .	517
3.13.1Optimizing GCC 4.9.1 x64 . . . . .	517
3.13.2Optimizing GCC 4.9 ARM64 . . . . .	518
3.14Variadic functions . . . . .	518
3.14.1Computing arithmetic mean . . . . .	518
3.14.2 <i>vprintf()</i> function case . . . . .	522
3.14.3Pin case . . . . .	523
3.14.4Format string exploit . . . . .	524
3.15Strings trimming . . . . .	525
3.15.1x64: Optimizing MSVC 2013 . . . . .	526
3.15.2x64: Non-optimizing GCC 4.9.1 . . . . .	527
3.15.3x64: Optimizing GCC 4.9.1 . . . . .	528
3.15.4ARM64: Non-optimizing GCC (Linaro) 4.9 . . . . .	529
3.15.5ARM64: Optimizing GCC (Linaro) 4.9 . . . . .	530
3.15.6ARM: Optimizing Keil 6/2013 (ARM mode) . . . . .	531
3.15.7ARM: Optimizing Keil 6/2013 (Thumb mode) . . . . .	531
3.15.8MIPS . . . . .	532
3.16 <i>toupper()</i> function . . . . .	533
3.16.1x64 . . . . .	534
3.16.2ARM . . . . .	535
3.16.3Summary . . . . .	537
3.17Incorrectly disassembled code . . . . .	537
3.17.1Disassembling from an incorrect start (x86) . . . . .	537
3.17.2How does random noise looks disassembled? . . . . .	538
3.18Obfuscation . . . . .	541
3.18.1Text strings . . . . .	542
3.18.2Executable code . . . . .	542
3.18.3Virtual machine / pseudo-code . . . . .	544
3.18.4Other things to mention . . . . .	544
3.18.5Exercise . . . . .	544
3.19C++ . . . . .	544
3.19.1Classes . . . . .	544
3.19.2 <code>ostream</code> . . . . .	561
3.19.3References . . . . .	562
3.19.4STL . . . . .	563
3.19.5Memory . . . . .	596
3.20Negative array indices . . . . .	596
3.20.1Addressing string from the end . . . . .	597
3.20.2Addressing some kind of block from the end . . . . .	597
3.20.3Arrays started at 1 . . . . .	597
3.21Packing 12-bit values into array . . . . .	599
3.21.1Introduction . . . . .	600
3.21.2Data structure . . . . .	600
3.21.3The algorithm . . . . .	600
3.21.4The C/C++ code . . . . .	600
3.21.5How it works . . . . .	603
3.21.6Optimizing GCC 4.8.2 for x86-64 . . . . .	604
3.21.7Optimizing Keil 5.05 (Thumb mode) . . . . .	606
3.21.8Optimizing Keil 5.05 (ARM mode) . . . . .	608
3.21.9(32-bit ARM) Comparison of code density in Thumb and ARM modes . . . . .	610
3.21.10Optimizing GCC 4.9.3 for ARM64 . . . . .	610
3.21.11Optimizing GCC 4.4.5 for MIPS . . . . .	612
3.21.12Difference from the real FAT12 . . . . .	614
3.21.13Exercise . . . . .	614
3.21.14Summary . . . . .	615
3.21.15Conclusion . . . . .	615
3.22Windows 16-bit . . . . .	615
3.22.1Example #1 . . . . .	615
3.22.2Example #2 . . . . .	616
3.22.3Example #3 . . . . .	616
3.22.4Example #4 . . . . .	617
3.22.5Example #5 . . . . .	620
3.22.6Example #6 . . . . .	623

## CONTENTS

3.23 More about pointers . . . . .	626
3.23.1 Working with addresses instead of pointers . . . . .	626
3.23.2 Passing values as pointers; tagged unions . . . . .	629
3.23.3 Pointers abuse in Windows kernel . . . . .	629
3.23.4 Null pointers . . . . .	634
3.23.5 Array as function argument . . . . .	637
3.23.6 Pointer to function . . . . .	638
3.23.7 Pointer as object identifier . . . . .	639
3.24 Loop optimizations . . . . .	640
3.24.1 Weird loop optimization . . . . .	640
3.24.2 Another loop optimization . . . . .	641
3.25 More about structures . . . . .	643
3.25.1 Sometimes a C structure can be used instead of array . . . . .	643
3.25.2 Unsized array in C structure . . . . .	644
3.25.3 Version of C structure . . . . .	645
3.25.4 High-score file in “Block out” game and primitive serialization . . . . .	647
3.26 memmove() and memcpy() . . . . .	652
3.26.1 Anti-debugging trick . . . . .	652
3.27 setjmp/longjmp . . . . .	653
3.28 Other weird stack hacks . . . . .	655
3.28.1 Accessing arguments/local variables of caller . . . . .	655
3.28.2 Returning string . . . . .	657
3.29 OpenMP . . . . .	658
3.29.1 MSVC . . . . .	660
3.29.2 GCC . . . . .	662
3.30 Another heisenbug . . . . .	663
<b>4 Java</b> . . . . .	<b>665</b>
4.1 Java . . . . .	665
4.1.1 Introduction . . . . .	665
4.1.2 Returning a value . . . . .	665
4.1.3 Simple calculating functions . . . . .	670
4.1.4 JVM <sup>3</sup> memory model . . . . .	672
4.1.5 Simple function calling . . . . .	673
4.1.6 Calling beep() . . . . .	674
4.1.7 Linear congruential PRNG <sup>4</sup> . . . . .	675
4.1.8 Conditional jumps . . . . .	676
4.1.9 Passing arguments . . . . .	678
4.1.10 Bitfields . . . . .	679
4.1.11 Loops . . . . .	680
4.1.12 switch() . . . . .	682
4.1.13 Arrays . . . . .	683
4.1.14 Strings . . . . .	691
4.1.15 Exceptions . . . . .	693
4.1.16 Classes . . . . .	696
4.1.17 Simple patching . . . . .	698
4.1.18 Summary . . . . .	703
<b>5 Finding important/interesting stuff in the code</b> . . . . .	<b>704</b>
5.1 Identification of executable files . . . . .	704
5.1.1 Microsoft Visual C++ . . . . .	704
5.1.2 GCC . . . . .	705
5.1.3 Intel Fortran . . . . .	705
5.1.4 Watcom, OpenWatcom . . . . .	705
5.1.5 Borland . . . . .	706
5.1.6 Other known DLLs . . . . .	707
5.2 Communication with outer world (function level) . . . . .	707
5.3 Communication with the outer world (win32) . . . . .	707
5.3.1 Often used functions in the Windows API . . . . .	708
5.3.2 Extending trial period . . . . .	708
5.3.3 Removing nag dialog box . . . . .	708
5.3.4 tracer: Intercepting all functions in specific module . . . . .	708

<sup>3</sup>Java virtual machine

<sup>4</sup>Pseudorandom number generator

## CONTENTS

5.4 Strings . . . . .	709
5.4.1 Text strings . . . . .	709
5.4.2 Finding strings in binary . . . . .	714
5.4.3 Error/debug messages . . . . .	715
5.4.4 Suspicious magic strings . . . . .	715
5.5 Calls to assert() . . . . .	716
5.6 Constants . . . . .	716
5.6.1 Magic numbers . . . . .	717
5.6.2 Specific constants . . . . .	719
5.6.3 Searching for constants . . . . .	719
5.7 Finding the right instructions . . . . .	719
5.8 Suspicious code patterns . . . . .	720
5.8.1 XOR instructions . . . . .	720
5.8.2 Hand-written assembly code . . . . .	721
5.9 Using magic numbers while tracing . . . . .	722
5.10 Other things . . . . .	722
5.10.1 General idea . . . . .	722
5.10.2 Order of functions in binary code . . . . .	722
5.10.3 Tiny functions . . . . .	722
5.10.4 C++ . . . . .	722
5.10.5 Some binary file patterns . . . . .	723
5.10.6 Memory “snapshots” comparing . . . . .	730
<b>6 OS-specific</b> . . . . .	<b>732</b>
6.1 Arguments passing methods (calling conventions) . . . . .	732
6.1.1 cdecl . . . . .	732
6.1.2 stdcall . . . . .	732
6.1.3 fastcall . . . . .	733
6.1.4 thiscall . . . . .	735
6.1.5 x86-64 . . . . .	735
6.1.6 Return values of <i>float</i> and <i>double</i> type . . . . .	738
6.1.7 Modifying arguments . . . . .	738
6.1.8 Taking a pointer to function argument . . . . .	739
6.2 Thread Local Storage . . . . .	740
6.2.1 Linear congruential generator revisited . . . . .	741
6.3 System calls (syscall-s) . . . . .	745
6.3.1 Linux . . . . .	746
6.3.2 Windows . . . . .	746
6.4 Linux . . . . .	746
6.4.1 Position-independent code . . . . .	746
6.4.2 <i>LD_PRELOAD</i> hack in Linux . . . . .	749
6.5 Windows NT . . . . .	751
6.5.1 CRT (win32) . . . . .	751
6.5.2 Win32 PE . . . . .	755
6.5.3 Windows SEH . . . . .	762
6.5.4 Windows NT: Critical section . . . . .	785
<b>7 Tools</b> . . . . .	<b>787</b>
7.1 Binary analysis . . . . .	787
7.1.1 Disassemblers . . . . .	787
7.1.2 Decompilers . . . . .	788
7.1.3 Patch comparison/diffing . . . . .	788
7.2 Live analysis . . . . .	788
7.2.1 Debuggers . . . . .	788
7.2.2 Library calls tracing . . . . .	788
7.2.3 System calls tracing . . . . .	789
7.2.4 Network sniffing . . . . .	789
7.2.5 Sysinternals . . . . .	789
7.2.6 Valgrind . . . . .	789
7.2.7 Emulators . . . . .	789
7.3 Other tools . . . . .	790
7.4 Something missing here? . . . . .	790
<b>8 Case studies</b> . . . . .	<b>791</b>
8.1 Task manager practical joke (Windows Vista) . . . . .	792

## CONTENTS

8.1.1 Using LEA to load values . . . . .	795
8.2 Color Lines game practical joke . . . . .	797
8.3 Minesweeper (Windows XP) . . . . .	800
8.3.1 Exercises . . . . .	805
8.4 Hacking Windows clock . . . . .	806
8.5 Dongles . . . . .	812
8.5.1 Example #1: MacOS Classic and PowerPC . . . . .	812
8.5.2 Example #2: SCO OpenServer . . . . .	820
8.5.3 Example #3: MS-DOS . . . . .	829
8.6 "QR9": Rubik's cube inspired amateur crypto-algorithm . . . . .	834
8.7 Encrypted database case #1 . . . . .	861
8.7.1 Base64 and entropy . . . . .	862
8.7.2 Is it compressed? . . . . .	864
8.7.3 Is it encrypted? . . . . .	865
8.7.4 CryptoPP . . . . .	865
8.7.5 Cipher Feedback mode . . . . .	867
8.7.6 Initializing Vector . . . . .	869
8.7.7 Structure of the buffer . . . . .	870
8.7.8 Noise at the end . . . . .	872
8.7.9 Conclusion . . . . .	872
8.7.10 Post Scriptum: brute-forcing IV . . . . .	873
8.8 Overclocking Cointerra Bitcoin miner . . . . .	873
8.9 Breaking simple executable cryptor . . . . .	877
8.9.1 Other ideas to consider . . . . .	882
8.10 SAP . . . . .	882
8.10.1 About SAP client network traffic compression . . . . .	882
8.10.2 SAP 6.0 password checking functions . . . . .	893
8.11 Oracle RDBMS . . . . .	897
8.11.1 V\$VERSION table in the Oracle RDBMS . . . . .	897
8.11.2 X\$KSMLRU table in Oracle RDBMS . . . . .	905
8.11.3 V\$TIMER table in Oracle RDBMS . . . . .	906
8.12 Handwritten assembly code . . . . .	910
8.12.1 EICAR test file . . . . .	910
8.13 Demos . . . . .	911
8.13.1 10 PRINT CHR\$(205.5+RND(1)); : GOTO 10 . . . . .	911
8.13.2 Mandelbrot set . . . . .	914
8.14 Other examples . . . . .	924
<b>9 Examples of reversing proprietary file formats</b> . . . . .	<b>925</b>
9.1 Primitive XOR-encryption . . . . .	925
9.1.1 Norton Guide: simplest possible 1-byte XOR encryption . . . . .	926
9.1.2 Simplest possible 4-byte XOR encryption . . . . .	929
9.1.3 Simple encryption using XOR mask . . . . .	933
9.1.4 Simple encryption using XOR mask, case II . . . . .	939
9.2 Analyzing using information entropy . . . . .	945
9.2.1 Analyzing entropy in Mathematica . . . . .	945
9.2.2 Conclusion . . . . .	952
9.2.3 Tools . . . . .	952
9.2.4 A word about primitive encryption like XORing . . . . .	952
9.2.5 More about entropy of executable code . . . . .	952
9.2.6 Random number generators . . . . .	953
9.2.7 Entropy of various files . . . . .	953
9.2.8 Making lower level of entropy . . . . .	954
9.3 Millenium game save file . . . . .	955
9.4 fortune program indexing file . . . . .	962
9.4.1 Hacking . . . . .	966
9.4.2 The files . . . . .	967
9.5 Oracle RDBMS: .SYM-files . . . . .	967
9.6 Oracle RDBMS: .MSB-files . . . . .	977
9.6.1 Summary . . . . .	984
9.7 Exercise . . . . .	984
<b>10 Other things</b> . . . . .	<b>985</b>
10.1 Executable files patching . . . . .	985

## CONTENTS

10.1.1Text strings . . . . .	985
10.1.2x86 code . . . . .	985
10.2Function arguments statistics . . . . .	986
10.3Compiler intrinsic . . . . .	986
10.4Compiler's anomalies . . . . .	987
10.4.1Oracle RDBMS 11.2 and Intel C++ 10.1 . . . . .	987
10.4.2MSVC 6.0 . . . . .	987
10.4.3Summary . . . . .	988
10.5Itanium . . . . .	988
10.68086 memory model . . . . .	990
10.7Basic blocks reordering . . . . .	991
10.7.1Profile-guided optimization . . . . .	991
<b>11 Books/blogs worth reading</b> . . . . .	<b>993</b>
11.1Books and other materials . . . . .	993
11.1.1Reverse Engineering . . . . .	993
11.1.2Windows . . . . .	993
11.1.3C/C++ . . . . .	993
11.1.4x86 / x86-64 . . . . .	994
11.1.5ARM . . . . .	994
11.1.6Java . . . . .	994
11.1.7UNIX . . . . .	994
11.1.8Programming in general . . . . .	994
11.1.9Cryptography . . . . .	995
<b>12 Communities</b> . . . . .	<b>996</b>
<b>Afterword</b> . . . . .	<b>998</b>
12.1Questions? . . . . .	998
<b>Appendix</b> . . . . .	<b>1000</b>
.1 x86 . . . . .	1000
.1.1 Terminology . . . . .	1000
.1.2 General purpose registers . . . . .	1000
.1.3 FPU registers . . . . .	1004
.1.4 SIMD registers . . . . .	1006
.1.5 Debugging registers . . . . .	1006
.1.6 Instructions . . . . .	1007
.1.7 npad . . . . .	1019
.2 ARM . . . . .	1020
.2.1 Terminology . . . . .	1020
.2.2 Versions . . . . .	1021
.2.3 32-bit ARM (AArch32) . . . . .	1021
.2.4 64-bit ARM (AArch64) . . . . .	1022
.2.5 Instructions . . . . .	1022
.3 MIPS . . . . .	1023
.3.1 Registers . . . . .	1023
.3.2 Instructions . . . . .	1024
.4 Some GCC library functions . . . . .	1024
.5 Some MSVC library functions . . . . .	1024
.6 Cheatsheets . . . . .	1025
.6.1 IDA . . . . .	1025
.6.2 OllyDbg . . . . .	1025
.6.3 MSVC . . . . .	1025
.6.4 GCC . . . . .	1026
.6.5 GDB . . . . .	1026
<b>Acronyms used</b> . . . . .	<b>1029</b>
<b>Glossary</b> . . . . .	<b>1034</b>
<b>Index</b> . . . . .	<b>1036</b>

There are several popular meanings of the term “reverse engineering”: 1) The reverse engineering of software: researching compiled programs; 2) The scanning of 3D structures and the subsequent digital manipulation required in order to duplicate them; 3) Recreating [DBMS<sup>5</sup>](#) structure. This book is about the first meaning.

## Topics discussed in-depth

x86/x64, ARM/ARM64, MIPS, Java/JVM.

## Topics touched upon

Oracle RDBMS ( [8.11 on page 897](#)), Itanium ( [10.5 on page 988](#)), copy-protection dongles ( [8.5 on page 812](#)), LD\_PRELOAD ( [6.4.2 on page 749](#)), stack overflow, [ELF<sup>6</sup>](#), win32 PE file format ( [6.5.2 on page 755](#)), x86-64 ( [1.30.1 on page 421](#)), critical sections ( [6.5.4 on page 785](#)), syscalls ( [6.3 on page 745](#)), [TLS<sup>7</sup>](#), position-independent code (PIC<sup>8</sup>) ( [6.4.1 on page 746](#)), profile-guided optimization ( [10.7.1 on page 991](#)), C++ STL ( [3.19.4 on page 563](#)), OpenMP ( [3.29 on page 658](#)), SEH ( [6.5.3 on page 762](#)).

## Prerequisites

Basic C [PL<sup>9</sup>](#) knowledge. Recommended reading: [11.1.3 on page 993](#).

## Exercises and tasks

...are all moved to the separate website: <http://challenges.re>.

## About the author



Dennis Yurichev is an experienced reverse engineer and programmer. He can be contacted by email: [dennis\(a\)yurichev.com](mailto:dennis(a)yurichev.com).

## Praise for *Reverse Engineering for Beginners*

- “Now that Dennis Yurichev has made this book free (libre), it is a contribution to the world of free knowledge and free education.” Richard M. Stallman, GNU founder, software freedom activist.
- “It’s very well done .. and for free .. amazing.”<sup>10</sup> Daniel Bilar, Siege Technologies, LLC.

<sup>5</sup>Database management systems

<sup>6</sup>Executable file format widely used in \*NIX systems including Linux

<sup>7</sup>Thread Local Storage

<sup>8</sup>Position Independent Code: [6.4.1 on page 746](#)

<sup>9</sup>Programming language

<sup>10</sup>[twitter.com/daniel\\_bilar/status/436578617221742593](https://twitter.com/daniel_bilar/status/436578617221742593)

## CONTENTS

---

- "... excellent and free"<sup>11</sup> 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, co-author of *Modern Operating Systems (4th Edition)*.
- "... 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.
- "... reasonable intro to some of the techniques."<sup>12</sup> Mike Stay, teacher at the Federal Law Enforcement Training Center, Georgia, US.
- "I love this book! I have several students reading it at the moment, plan to use it in graduate course."<sup>13</sup> Sergey Bratus , Research Assistant Professor at the Computer Science Department at Dartmouth College
- "Dennis @Yurichev has published an impressive (and free!) book on reverse engineering"<sup>14</sup> Tanel Poder, Oracle RDBMS performance tuning expert .
- "This book is some kind of Wikipedia to beginners..." Archer, Chinese Translator, IT Security Researcher.
- "First class reference for people wanting to learn reverse engineering. And it's free for all." Mikko Hypponen, F-Secure.

## Thanks

For patiently answering all my questions: Andrey "hermit" Baranovich, Slava "Avid" Kazakov.

For sending me notes about mistakes and inaccuracies: Stanislav "Beaver" Bobrytskyy, Alexander Lysenko, Alexander "Solar Designer" Peslyak, Federico Ramondino, Mark Wilson, Shell Rocket, Zhu Ruijin, Changmin Heo, Vitor Vidal, Stijn Crevits, Jean-Gregoire Foulon<sup>15</sup>, Ben L., Etienne Khan, Norbert Szetei<sup>16</sup>, Marc Remy..

For helping me in other ways: Andrew Zubinski, Arnaud Patard (rtp on #debian-arm IRC), noshadow on #gcc IRC, Aliaksandr Autayeu, Mohsen Mostafa Jokar.

For translating the book into Simplified Chinese: Antiy Labs ([antiy.cn](#)), Archer.

For translating the book into Korean: Byungho Min.

For translating the book into Dutch: Cedric Sambre (AKA Midas).

For translating the book into Spanish: Diego Boy, Luis Alberto Espinosa Calvo, Fernando Guida.

For translating the book into Portuguese: Thales Stevan de A. Gois.

For translating the book into Italian: Federico Ramondino<sup>17</sup>, Paolo Stivanin<sup>18</sup>, twyK.

For translating the book into French: Florent Besnard<sup>19</sup>, Marc Remy<sup>20</sup>, Baudouin Landais, Téo Dacquet<sup>21</sup>.

For translating the book into German: Dennis Siekmeier<sup>22</sup>, Julius Angres<sup>23</sup>, Dirk Loser.

For proofreading: Alexander "Lstar" Chernenkiy, Vladimir Botov, Andrei Brazhuk, Mark "Logxen" Cooper, Yuan Jochen Kang, Mal Malakov, Lewis Porter, Jarle Thorsen, Hong Xie.

Vasil Kolev<sup>24</sup> did a great amount of work in proofreading and correcting many mistakes.

---

<sup>11</sup>[twitter.com/petefinnigan/status/400551705797869568](https://twitter.com/petefinnigan/status/400551705797869568)

<sup>12</sup>[reddit](https://www.reddit.com/r/reverseengineering/)

<sup>13</sup>[twitter.com/sergeybratus/status/505590326560833536](https://twitter.com/sergeybratus/status/505590326560833536)

<sup>14</sup>[twitter.com/TanelPoder/status/524668104065159169](https://twitter.com/TanelPoder/status/524668104065159169)

<sup>15</sup><https://github.com/pixjuan>

<sup>16</sup><https://github.com/73696e65>

<sup>17</sup><https://github.com/pinkrab>

<sup>18</sup><https://github.com/paolostivanin>

<sup>19</sup><https://github.com/besnardf>

<sup>20</sup><https://github.com/mremy>

<sup>21</sup><https://github.com/T30rix>

<sup>22</sup><https://github.com/DSiekmeier>

<sup>23</sup><https://github.com/JAngres>

<sup>24</sup><https://vasil.ludost.net/>

## CONTENTS

---

For illustrations and cover art: Andy Nечаевский.

Thanks also to all the folks on [github.com](https://github.com) who have contributed notes and corrections<sup>25</sup>.

Many  $\text{\LaTeX}$  packages were used: I would like to thank the authors as well.

## Donors

Those who supported me during the time when I wrote significant part of the book:

2 \* Oleg Vygovsky (50+100 UAH), Daniel Bilar (\$50), James Truscott (\$4.5), Luis Rocha (\$63), Joris van de Vis (\$127), Richard S Shultz (\$20), Jang Minchang (\$20), Shade Atlas (5 AUD), Yao Xiao (\$10), Paweł Szczur (40 CHF), Justin Simms (\$20), Shawn the R0ck (\$27), Ki Chan Ahn (\$50), Triop AB (100 SEK), Ange Albertini (€10+50), Sergey Lukianov (300 RUR), Ludvig Gislason (200 SEK), Gérard Labadie (€40), Sergey Volchkov (10 AUD), Vankayala Vigneswararao (\$50), Philippe Teuwen (\$4), Martin Haeberli (\$10), Victor Cazacov (€5), Tobias Sturzenegger (10 CHF), Sonny Thai (\$15), Bayna AlZaabi (\$75), Redfive B.V. (€25), Joonas Oskari Heikkilä (€5), Marshall Bishop (\$50), Nicolas Werner (€12), Jeremy Brown (\$100), Alexandre Borges (\$25), Vladimir Dikovski (€50), Jiarui Hong (100.00 SEK), Jim Di (500 RUR), Tan Vincent (\$30), Sri Harsha Kandarakota (10 AUD), Pillay Harish (10 SGD), Timur Valiev (230 RUR), Carlos Garcia Prado (€10), Salikov Alexander (500 RUR), Oliver Whitehouse (30 GBP), Katy Moe (\$14), Maxim Dyakonov (\$3), Sebastian Aguilera (€20), Hans-Martin Münch (€15), Jarle Thorsen (100 NOK), Vitaly Osipov (\$100), Yuri Romanov (1000 RUR), Aliaksandr Autayeu (€10), Tudor Azoitei (\$40), Z0vsky (€10), Yu Dai (\$10).

Thanks a lot to every donor!

## mini-FAQ

Q: What are prerequisites for reading this book?

A: Basic understanding of C/C++ is desirable.

Q: Can I buy Russian/English hardcopy/paper book?

A: Unfortunately no, no publisher got interested in publishing Russian or English version so far. Meanwhile, you can ask your favorite copy shop to print/bind it.

Q: Is there epub/mobi version?

A: The book is highly dependent on TeX/LaTeX-specific hacks, so converting to HTML (epub/mobi is a set of HTMLs) will not be easy.

Q: Why should one learn assembly language these days?

A: Unless you are an [OS<sup>26</sup>](#) developer, you probably don't need to code in assembly—latest compilers (2010s) are much better at performing optimizations than humans<sup>27</sup>.

Also, latest [CPU<sup>28</sup>](#)s are very complex devices and assembly knowledge doesn't really help one to understand their internals.

That being said, there are at least two areas where a good understanding of assembly can be helpful: First and foremost, security/malware research. It is also a good way to gain a better understanding of your compiled code whilst debugging. This book is therefore intended for those who want to understand assembly language rather than to code in it, which is why there are many examples of compiler output contained within.

Q: I clicked on a hyperlink inside a PDF-document, how do I go back?

A: In Adobe Acrobat Reader click Alt+LeftArrow. In Evince click “<” button.

Q: May I print this book / use it for teaching?

A: Of course! That's why the book is licensed under the Creative Commons license (CC BY-SA 4.0).

Q: Why is this book free? You've done great job. This is suspicious, as many other free things.

A: In my own experience, authors of technical literature do this mostly for self-advertisement purposes. It's not possible to get any decent money from such work.

---

<sup>25</sup><https://github.com/dennis714/RE-for-beginners/graphs/contributors>

<sup>26</sup>Operating System

<sup>27</sup>A very good text about this topic: [Agner Fog, *The microarchitecture of Intel, AMD and VIA CPUs*, (2016)]

<sup>28</sup>Central processing unit

## CONTENTS

---

Q: How does one get a job in reverse engineering?

A: There are hiring threads that appear from time to time on reddit, devoted to RE<sup>29</sup> (2016). Try looking there.

A somewhat related hiring thread can be found in the “netsec” subreddit: 2016.

Q: I have a question...

A: Send it to me by email (dennis(a)yurichev.com).

## About the Korean translation

In January 2015, the Acorn publishing company ([www.acornpub.co.kr](http://www.acornpub.co.kr)) in South Korea did a huge amount of work in translating and publishing my book (as it was in August 2014) into Korean.

It's now available at [their website](#).

The translator is Byungho Min ([twitter/tais9](#)). The cover art was done by my artistic friend, Andy Nechaevsky: [facebook/andydinka](#). They also hold the copyright to the Korean translation.

So, if you want to have a *real* book on your shelf in Korean and want to support my work, it is now available for purchase.

---

<sup>29</sup>[reddit.com/r/ReverseEngineering/](http://reddit.com/r/ReverseEngineering/)

# Chapter 1

## Code patterns

Everything is comprehended through comparison

---

Author unknown

### 1.1 The method

When the author of this book first started learning C and, later, C++, he used to write small pieces of code, compile them, and then look at the assembly language output. This made it very easy for him to understand what was going on in the code that he had written.<sup>1</sup>. He did it so many times that the relationship between the C/C++ code and what the compiler produced was imprinted deeply in his mind. It's easy to imagine instantly a rough outline of C code's appearance and function. Perhaps this technique could be helpful for others.

Sometimes ancient compilers are used here, in order to get the shortest (or simplest) possible code snippet.

### Exercises

When the author of this book studied assembly language, he also often compiled small C-functions and then rewrote them gradually to assembly, trying to make their code as short as possible. This probably is not worth doing in real-world scenarios today, because it's hard to compete with latest compilers in terms of efficiency. It is, however, a very good way to gain a better understanding of assembly. Feel free, therefore, to take any assembly code from this book and try to make it shorter. However, don't forget to test what you have written.

### Optimization levels and debug information

Source code can be compiled by different compilers with various optimization levels. A typical compiler has about three such levels, where level zero means disable optimization. Optimization can also be targeted towards code size or code speed. A non-optimizing compiler is faster and produces more understandable (albeit verbose) code, whereas an optimizing compiler is slower and tries to produce code that runs faster (but is not necessarily more compact). In addition to optimization levels, a compiler can include in the resulting file some debug information, thus producing code for easy debugging. One of the important features of the 'debug' code is that it might contain links between each line of the source code and the respective machine code addresses. Optimizing compilers, on the other hand, tend to produce output where entire lines of source code can be optimized away and thus not even be present in the resulting machine code. Reverse engineers can encounter either version, simply because some developers turn on the compiler's optimization flags and others do not. Because of this, we'll try to work on examples of both debug and release versions of the code featured in this book, where possible.

---

<sup>1</sup>In fact, he still does it when he can't understand what a particular bit of code does.

## 1.2 Some basics

### 1.2.1 A short introduction to the CPU

The **CPU** is the device that executes the machine code a program consists of.

#### A short glossary:

**Instruction** : A primitive **CPU** command. The simplest examples include: moving data between registers, working with memory, primitive arithmetic operations. As a rule, each **CPU** has its own instruction set architecture (**ISA**<sup>2</sup>).

**Machine code** : Code that the **CPU** directly processes. Each instruction is usually encoded by several bytes.

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

**CPU register** : Each **CPU** has a fixed set of general purpose registers (**GPR**<sup>3</sup>). ≈ 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 if you were working with a high-level **PL** and could only use eight 32-bit (or 64-bit) variables. Yet a lot can be done using just these!

One might wonder why there needs to be a difference between machine code and a **PL**. The answer lies in the fact that humans and **CPU**s are not alike—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** that can execute high-level **PL** code, but it would be many times more complex than the **CPU**s we know of today. In a similar fashion, it is very inconvenient for humans to write in assembly language, due to it being so low-level and difficult to write in without making a huge number of annoying mistakes. The program that converts the high-level **PL** code into assembly is called a *compiler*. <sup>4</sup>.

#### A couple of words about different **ISAs**

The x86 **ISA** has always been one with variable-length instructions, so when the 64-bit era came, the x64 extensions did not impact the **ISA** very significantly. In fact, the x86 **ISA** still contains a lot of instructions that first appeared in 16-bit 8086 CPU, yet are still found in the CPUs of today. ARM is a **RISC**<sup>5</sup> **CPU** designed with constant-length instructions in mind, which had some advantages in the past. In the very beginning, all ARM instructions were encoded in 4 bytes<sup>6</sup>. This is now referred to as "ARM mode". Then they thought it wasn't as frugal as they first imagined. In fact, most used **CPU** instructions<sup>7</sup> in real world applications can be encoded using less information. They therefore added another **ISA**, called Thumb, where each instruction was encoded in just 2 bytes. This is now referred as "Thumb mode". However, not *all* ARM instructions can be encoded in just 2 bytes, so the Thumb instruction set is somewhat limited. It is worth noting that code compiled for ARM mode and Thumb mode may of course coexist within one single program. The ARM creators thought Thumb could be extended, giving rise to Thumb-2, which appeared in ARMv7. Thumb-2 still uses 2-byte instructions, but has some new instructions which have the size of 4 bytes. There is a common misconception that Thumb-2 is a mix of ARM and Thumb. This is incorrect. Rather, Thumb-2 was extended to fully support all processor features so it could compete with ARM mode—a goal that was clearly achieved, as the majority of applications for iPod/iPhone/iPad are compiled for the Thumb-2 instruction set (admittedly, largely due to the fact that Xcode does this by default). Later the 64-bit ARM came out. This **ISA** has 4-byte instructions, and lacked the need of any additional Thumb mode. However, the 64-bit requirements affected the **ISA**, resulting in us now having three ARM instruction sets: ARM mode, Thumb mode (including Thumb-2) and ARM64. These **ISAs** intersect partially, but it can be said that they are different **ISAs**, rather than variations of the same one. Therefore, we would try to add fragments of code in all three ARM **ISAs** in this book. There are, by the way, many other **RISC ISAs** with fixed length 32-bit instructions, such as MIPS, PowerPC and Alpha AXP.

<sup>2</sup>Instruction Set Architecture

<sup>3</sup>General Purpose Registers

<sup>4</sup>Old-school Russian literature also use term "translator".

<sup>5</sup>Reduced instruction set computing

<sup>6</sup>By the way, fixed-length instructions are handy because one can calculate the next (or previous) instruction address without effort. This feature will be discussed in the switch() operator ( [1.15.2 on page 174](#) ) section.

<sup>7</sup>These are MOV/PUSH/CALL/Jcc

## 1.2.2 Numeral systems

Humans accustomed to decimal numeral system probably because almost all ones has 10 fingers. Nevertheless, number 10 has no significant meaning in science and mathematics. Natural numeral system in digital electronics is binary: 0 is for absence of current in wire and 1 for presence. 10 in binary is 2 in decimal; 100 in binary is 4 in decimal and so on.

If the numeral system has 10 digits, it has *radix* (or *base*) of 10. Binary numeral system has *radix* of 2.

Important things to recall: 1) *number* is a number, while *digit* is a term of writing system and is usually one character; 2) number is not changed when converted to another radix: writing notation is.

How to convert a number from one radix to another?

Positional notation is used almost everywhere, this means, a digit (number placed in single character) has some weight depending on where it is placed. If 2 is placed at the rightmost place, it's 2. If it is placed at the place one digit before rightmost, it's 20.

What does 1234 stand for?

$$10^3 \cdot 1 + 10^2 \cdot 2 + 10^1 \cdot 3 + 1 \cdot 4 = 1234 \text{ or } 1000 \cdot 1 + 100 \cdot 2 + 10 \cdot 3 + 4 = 1234$$

Same story for binary numbers, but base is 2 instead of 10. What does 0b101011 stand for?

$$2^5 \cdot 1 + 2^4 \cdot 0 + 2^3 \cdot 1 + 2^2 \cdot 0 + 2^1 \cdot 1 + 2^0 \cdot 1 = 43 \text{ or } 32 \cdot 1 + 16 \cdot 0 + 8 \cdot 1 + 4 \cdot 0 + 2 \cdot 1 + 1 = 43$$

Positional notation can be opposed to non-positional notation such as Roman numeric system<sup>8</sup>. Perhaps, humankind switched to positional notation because it's easier to do basic operations (addition, multiplication, etc.) on paper by hand.

Indeed, binary numbers can be added, subtracted and so on in the very same as taught in schools, but only 2 digits are available.

Binary numbers are bulky when represented in source code and dumps, so that is where hexadecimal numeral system can be used. Hexadecimal radix uses 0..9 numbers and also 6 Latin characters: A..F. Each hexadecimal digit takes 4 bits or 4 binary digits, so it's very easy to convert from binary number to hexadecimal and back, even manually, in one's mind.

hexadecimal	binary	decimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

How to understand, which radix is currently used?

Decimal numbers are usually written as is, i.e., 1234. But some assemblers allows to make emphasis on decimal radix and this number can be written with "d" suffix: 1234d.

Binary numbers sometimes prepended with "0b" prefix: 0b100110111 (GCC<sup>9</sup> has non-standard language extension for this<sup>10</sup>). There is also another way: "b" suffix, for example: 100110111b. I'll try to stick to "0b" prefix throughout the book for binary numbers.

<sup>8</sup>About numeric system evolution, see [Donald E. Knuth, *The Art of Computer Programming*, Volume 2, 3rd ed., (1997), 195–213.]

<sup>9</sup>GNU Compiler Collection

<sup>10</sup><https://gcc.gnu.org/onlinedocs/gcc/Binary-constants.html>

## 1.2. SOME BASICS

Hexadecimal numbers are prepended with "0x" prefix in C/C++ and other PLs: 0x1234ABCD. Or they are has "h" suffix: 1234ABCDh - this is common way of representing them in assemblers and debuggers. If the number is started with A..F digit, 0 is to be added before: 0ABCDEFh. I'll try to stick to "0x" prefix throughout the book for hexadecimal numbers.

Should one learn to convert numbers in mind? A table of 1-digit hexadecimal numbers can easily be memorized. As of larger numbers, probably, it's not worth to torment yourself.

Perhaps, the most visible to all people hexadecimal numbers are in [URL<sup>11</sup>s](#). This is the way how non-Latin characters are encoded. For example: <https://en.wiktionary.org/wiki/na%C3%AFvet%C3%A9> is the URL of Wiktionary article about "naïveté" word.

## Octal radix

Another numeral system heavily used in past of computer programming is octal: there are 8 digits (0..7) and each is mapped to 3 bits, so it's easy to convert numbers back and forth. It has been superseded by hexadecimal system almost everywhere, but surprisingly, there is \*NIX utility used by many people often which takes octal number as argument: `chmod`.

As many \*NIX users know, `chmod` argument can be a number of 3 digits. The first one is rights for owner of file, second is rights for group (to which file belongs), third is for everyone else. And each digit can be represented in binary form:

decimal	binary	meaning
7	111	<b>rwx</b>
6	110	<b>rw-</b>
5	101	<b>r-x</b>
4	100	<b>r--</b>
3	011	<b>-wx</b>
2	010	<b>-w-</b>
1	001	<b>--x</b>
0	000	<b>---</b>

So each bit is mapped to a flag: read/write/execute.

Now the reason why I'm talking about `chmod` here is that the whole number in argument can be represented as octal number. Let's take for example, 644. When you run `chmod 644 file`, you set read/write permissions for owner, read permissions for group and again, read permissions for everyone else. Let's convert 644 octal number to binary, this will be `110100100`, or (in groups of 3 bits) `110 100 100`.

Now we see that each triplet describe permissions for owner/group/others: first is `rw-`, second is `r--` and third is `r--`.

Octal numeral system was also popular on old computers like PDP-8, because word there could be 12, 24 or 36 bits, and these numbers are divisible by 3, so octal system was natural on that environment. Nowadays, all popular computers employs word/address size of 16, 32 or 64 bits, and these numbers are divisible by 4, so hexadecimal system is more natural here.

Octal numeral system is supported by all standard C/C++ compilers. This is source of confusion sometimes, because octal numbers are encoded with zero prepended, for example, 0377 is 255. And sometimes, you may make a typo and write "09" instead of 9, and the compiler wouldn't allow you. GCC may report something like that:

```
error: invalid digit "9" in octal constant.
```

## Divisibility

When you see a decimal number like 120, you can quickly deduce that it's divisible by 10, because the last digit is zero. In the same way, 123400 is divisible by 100, because two last digits are zeros.

Likewise, hexadecimal number 0x1230 is divisible by 0x10 (or 16), 0x123000 is divisible by 0x1000 (or 4096), etc.

<sup>11</sup>Uniform Resource Locator

### 1.3. EMPTY FUNCTION

---

Binary number 0b1000101000 is divisible by 0b1000 (8), etc.

This property can be used often to realize quickly if a size of some block in memory is padded to some boundary. For example, sections in PE<sup>12</sup> files are almost always started at addresses ending with 3 hexadecimal zeros: 0x41000, 0x10001000, etc. The reason behind this is in the fact that almost all PE sections are padded to boundary of 0x1000 (4096) bytes.

### Multi-precision arithmetic and radix

Multi-precision arithmetic uses huge numbers, and each one may be stored in several bytes. For example, RSA keys, both public and private, are spanning up to 4096 bits and maybe even more.

In [Donald E. Knuth, *The Art of Computer Programming*, Volume 2, 3rd ed., (1997), 265] we can find the following idea: when you store multi-precision number in several bytes, the whole number can be represented as having a radix of  $2^8 = 256$ , and each digit goes to corresponding byte. Likewise, if you store multi-precision number in several 32-bit integer values, each digit goes to each 32-bit slot, and you may think about this number as stored in radix of  $2^{32}$ .

### Pronouncement

Numbers in non-decimal base are usually pronounced by one digit: “one-zero-zero-one-one-...”. Words like “ten”, “thousand”, etc, are usually not pronounced, because it will be confused with decimal base then.

### Floating point numbers

To distinguish floating point numbers from integer ones, they are usually written with “.0” at the end, like 0.0, 123.0, etc.

## 1.3 Empty function

The simplest possible function is arguably one that does nothing:

Listing 1.1: C/C++ Code

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

Lets compile it!

### 1.3.1 x86

Here's what both the optimizing GCC and MSVC compilers produce on the x86 platform:

Listing 1.2: Optimizing GCC/MSVC (assembly output)

```
f:
    ret
```

There is just one instruction: `RET`, which returns execution to the [caller](#).

---

<sup>12</sup>Portable Executable

### 1.3.2 ARM

Listing 1.3: Optimizing Keil 6/2013 (ARM mode) assembly output

```
f      PROC
      BX      lr
      ENDP
```

The return address is not saved on the local stack in the ARM [ISA](#), but rather in the link register, so the `BX LR` instruction causes execution to jump to that address—effectively returning execution to the [caller](#).

### 1.3.3 MIPS

There are two naming conventions used in the world of MIPS when naming registers: by number (from \$0 to \$31) or by pseudo name (\$V0, \$A0, etc.).

The GCC assembly output below lists registers by number:

Listing 1.4: Optimizing GCC 4.4.5 (assembly output)

```
j      $31
nop
```

...while [IDA](#)<sup>13</sup> does it—by their pseudo names:

Listing 1.5: Optimizing GCC 4.4.5 (IDA)

```
j      $ra
nop
```

The first instruction is the jump instruction (J or JR) which returns the execution flow to the [caller](#), jumping to the address in the \$31 (or \$RA) register.

This is the register analogous to [LR](#)<sup>14</sup> in ARM.

The second instruction is [NOP](#)<sup>15</sup>, which does nothing. We can ignore it so far.

#### A note about MIPS instruction/register names

Register and instruction names in the world of MIPS are traditionally written in lowercase. However, for the sake of consistency, we'll stick to using uppercase letters, as it is the convention followed by all other [ISAs](#) featured this book.

### 1.3.4 Empty functions in practice

Despite the fact empty functions are useless, they are quite frequent in low-level code.

First of all, debugging functions are quite popular, like this one:

Listing 1.6: C/C++ Code

```
void dbg_print (const char *fmt, ...)
{
#ifndef _DEBUG
    // open log file
    // write to log file
    // close log file
#endif
};

void some_function()
{
    ...
}
```

<sup>13</sup> Interactive Disassembler and debugger developed by [Hex-Rays](#)

<sup>14</sup>Link Register

<sup>15</sup>No OPeration

## 1.4. RETURNING VALUE

```
    dbg_print ("we did something\n");
    ...
};
```

In non-debug build (e.g., “release”), `_DEBUG` is not defined, so `dbg_print()` function, despite still being called during execution, will be empty.

Another popular way of software protection is make several builds: one for legal customers, and a demo build. Demo build can lack some important functions, like this:

Listing 1.7: C/C++ Code

```
void save_file ()
{
#ifndef DEMO
    // actual saving code
#endif
};
```

`save_file()` function can be called when user click `File->Save` menu. Demo version may be delivered with this menu item disabled, but even if software cracker will enable it, empty function with no useful code will be called.

IDA marks such functions with names like `nullsub_00`, `nullsub_01`, etc.

## 1.4 Returning value

Another sample function is the one that simply returns a constant value:

Here it is:

Listing 1.8: C/C++ Code

```
int f()
{
    return 123;
};
```

Lets compile it.

### 1.4.1 x86

Here’s what both the optimizing GCC and MSVC compilers produce on the x86 platform:

Listing 1.9: Optimizing GCC/MSVC (assembly output)

```
f:
    mov     eax, 123
    ret
```

There are just two instructions: the first places the value 123 into the `EAX` register, which is used by convention for storing the return value and the second one is `RET`, which returns execution to the [caller](#).

The caller will take the result from the `EAX` register.

### 1.4.2 ARM

There are a few differences on the ARM platform:

## 1.5. HELLO, WORLD!

Listing 1.10: Optimizing Keil 6/2013 (ARM mode) ASM Output

```
f      PROC
      MOV      r0,#0x7b ; 123
      BX       lr
      ENDP
```

ARM uses the register `R0` for returning the results of functions, so 123 is copied into `R0`.

It is worth noting that `MOV` is a misleading name for the instruction in both x86 and ARM [ISAs](#).

The data is not in fact *moved*, but *copied*.

### 1.4.3 MIPS

The GCC assembly output below lists registers by number:

Listing 1.11: Optimizing GCC 4.4.5 (assembly output)

```
j      $31
li     $2,123          # 0x7b
```

...while [IDA](#) does it—by their pseudo names:

Listing 1.12: Optimizing GCC 4.4.5 (IDA)

```
jr    $ra
li    $v0, 0x7B
```

The `$2` (or `$V0`) register is used to store the function's return value. `LI` stands for “Load Immediate” and is the MIPS equivalent to `MOV`.

The other instruction is the jump instruction (`J` or `JR`) which returns the execution flow to the [caller](#).

You might be wondering why positions of the load instruction (`LI`) and the jump instruction (`J` or `JR`) are swapped. This is due to a [RISC](#) feature called “branch delay slot”.

The reason this happens is a quirk in the architecture of some RISC [ISAs](#) and isn't important for our purposes—we just must keep in mind that in MIPS, the instruction following a jump or branch instruction is executed *before* the jump/branch instruction itself.

As a consequence, branch instructions always swap places with the instruction which must be executed beforehand.

### 1.4.4 In practice

Functions which merely returns 1 (*true*) or 0 (*false*) are very frequent in practice.

## 1.5 Hello, world!

Let's use the famous example from the book [Brian W. Kernighan, Dennis M. Ritchie, *The C Programming Language*, (1988)]:

```
#include <stdio.h>

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

## 1.5.1 x86

### MSVC

Let's compile it in MSVC 2010:

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

(`/Fa` option instructs the compiler to generate assembly listing file)

Listing 1.13: MSVC 2010

```

CONST SEGMENT
$SG3830 DB      'hello, world', 0AH, 00H
CONST ENDS
PUBLIC _main
EXTERN _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 in [1.5.1 on page 11](#).

The compiler generated the file, `1.obj`, which is to be linked into `1.exe`. In our case, the file contains two segments: `CONST` (for data constants) and `_TEXT` (for code).

The string `hello, world` in C/C++ has type `const char[]` [Bjarne Stroustrup, *The C++ Programming Language, 4th Edition*, (2013)p176, 7.3.2], but 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.

That is why the example may be rewritten as follows:

```

#include <stdio.h>

const char $SG3830[]="hello, world\n";

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

```

Let's go 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/C++ strings: [5.4.1 on page 709](#).

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) <sup>16</sup>.

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 the control to the `main()` function, the string address (or a pointer to it) is still on the stack. Since we do not need it anymore, the `stack pointer` (the `ESP` register) needs to be corrected.

<sup>16</sup>You can read more about it in the section about function prologues and epilogues ( [1.6 on page 30](#)).

## 1.5. HELLO, WORLD!

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

Why 4? Since this is a 32-bit program, we need exactly 4 bytes for address passing through the stack. If it was x64 code we would need 8 bytes. ADD ESP, 4 is effectively equivalent to POP register but without using any register<sup>17</sup>.

For the same purpose, some compilers (like the Intel C++ Compiler) may emit POP ECX instead of ADD (e.g., such a pattern can be observed in the Oracle RDBMS code as it is compiled with the Intel C++ compiler). This instruction has almost the same effect but the ECX register contents will be overwritten. The Intel C++ compiler supposedly uses POP ECX since this instruction's opcode is shorter than ADD ESP, x (1 byte for POP against 3 for ADD ).

Here is an example of using POP instead of ADD from Oracle RDBMS:

Listing 1.14: Oracle RDBMS 10.2 Linux (app.o file)

```
.text:0800029A          push    ebx
.text:0800029B          call    qksfroChild
.text:080002A0          pop     ecx
```

After calling printf(), the original C/C++ code contains the statement return 0 —return 0 as the result of the main() function.

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

XOR is in fact just “eXclusive OR”<sup>18</sup> but the compilers often use it instead of MOV EAX, 0—again because it is a slightly shorter opcode (2 bytes for XOR against 5 for MOV ).

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

The last instruction RET returns the control to the caller. Usually, this is C/C++ CRT<sup>19</sup> code, which, in turn, returns control to the OS.

## GCC

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 disassembler, let's see how the main() function was created. IDA, like MSVC, uses Intel-syntax<sup>20</sup>.

Listing 1.15: code in IDA

```
main          proc near
var_10        = dword ptr -10h

        push    ebp
        mov     ebp, esp
        and     esp, 0FFFFFFF0h
        sub     esp, 10h
        mov     eax, offset aHelloWorld ; "hello, world\n"
        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 loaded in the EAX register first and then it is saved onto the stack.

<sup>17</sup>CPU flags, however, are modified

<sup>18</sup>wikipedia

<sup>19</sup>C runtime library

<sup>20</sup>We could also have GCC produce assembly listings in Intel-syntax by applying the options -S -masm=intel .

## 1.5. HELLO, WORLD!

In addition, the function prologue has `AND ESP, 0FFFFFFF0h` —this instruction aligns the `ESP` register value on a 16-byte boundary. This results in all values in the stack being aligned the same way (The CPU performs better if the values it is dealing with are located in memory at addresses aligned on a 4-byte or 16-byte boundary)<sup>21</sup>.

`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 stored directly onto the stack 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 (by executing `MOV EBP, ESP / AND ESP, ...`).

## GCC: AT&T syntax

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

Listing 1.16: let's compile in GCC 4.7.3

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

We get this:

Listing 1.17: GCC 4.7.3

```
.file "1_1.c"
.section .rodata
.LC0:
.string "hello, world\n"
.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
```

<sup>21</sup>[Wikipedia: Data structure alignment](#)

## 1.5. HELLO, WORLD!

The listing contains many macros (beginning with dot). These are not interesting for us at the moment.

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 <sup>22</sup>:

Listing 1.18: GCC 4.7.3

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

- Source and destination operands are written in opposite order.

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

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

Here is an easy way to memorize the difference: when you deal with Intel-syntax, you can imagine that there is an equality sign (=) between operands and when you deal with AT&T-syntax imagine there is a right arrow (→) <sup>23</sup>.

- 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 suffix is added to instructions to define the operand size:
  - q — quad (64 bits)
  - 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: `0xFFFFFFFF0h` is presented as `$-16`. It is the same thing: `16` in the decimal system is `0x10` in hexadecimal. `-0x10` is equal to `0xFFFFFFFF0` (for a 32-bit data type).

One more thing: the return value is to be set to 0 by using the usual `MOV`, not `XOR`. `MOV` just loads a value to a register. Its name is a misnomer (data is not moved but rather copied). In other architectures, this instruction is named “LOAD” or “STORE” or something similar.

## String patching (Win32)

We can easily find “hello, world” string in executable file using Hiew:

<sup>22</sup>This GCC option can be used to eliminate “unnecessary” macros: `-fno-asynchronous-unwind-tables`

<sup>23</sup>By the way, in some C standard functions (e.g., `memcpy()`, `strcpy()`) the arguments are listed in the same way as in Intel-syntax: first the pointer to the destination memory block, and then the pointer to the source memory block.

## 1.5. HELLO, WORLD!

```
Hiew: hw_spanish.exe
C:\tmp\hw_spanish.exe 0FW0 ----- PE+.00000001`40003000 Hiew 8.02
.400025E0: 00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
.400025F0: 00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
.40003000: 68 65 6C 6C-6F 2C 20 77-6F 72 6C 64-0A 00 00 00 hello, world
.40003010: 01 00 00 00-FE FF FF FF-FF FF FF FF-00 00 00 00
.40003020: 32 A2 DF 2D-99 2B 00 00-CD 5D 20 D2-66 D4 FF FF 2d-Ö+ =] ¶fL
.40003030: 00 00 00 00-00 00 00 00-00 00 00 00 00-00 00 00 00
.40003040: 00 00 00 00-00 00 00 00-00 00 00 00 00-00 00 00 00
```

Figure 1.1: Hiew

And we can try to translate our message to Spanish language:

```
Hiew: hw_spanish.exe
C:\tmp\hw_spanish.exe 0FW0 EDITMODE PE+ 00000000`0000120D Hiew 8.02
000011E0: 00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
000011F0: 00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
00001200: 68 6F 6C 61-2C 20 6D 75-6E 64 6F 0A-00 00 00 00 hola, mundo
00001210: 01 00 00 00-FE FF FF FF-FF FF FF FF-00 00 00 00
00001220: 32 A2 DF 2D-99 2B 00 00-CD 5D 20 D2-66 D4 FF FF 2d-Ö+ =] ¶fL
00001230: 00 00 00 00-00 00 00 00-00 00 00 00 00-00 00 00 00
00001240: 00 00 00 00-00 00 00 00-00 00 00 00 00-00 00 00 00
```

Figure 1.2: Hiew

Spanish text is one byte shorter than English, so we also add 0x0A byte at the end (\n) and zero byte. It works.

What if we want to insert longer message? There are some zero bytes after original English text. Hard to say if they can be overwritten: they may be used somewhere in [CRT](#) code, or maybe not. Anyway, you can overwrite them only if you really know what you are doing.

## String patching (Linux x64)

Let's try to patch Linux x64 executable using rada.re:

Listing 1.19: rada.re session

```
dennis@bigbox ~/tmp % gcc hw.c
dennis@bigbox ~/tmp % radare2 a.out
-- SHALL WE PLAY A GAME?
[0x00400430]> / hello
Searching 5 bytes from 0x00400000 to 0x00601040: 68 65 6c 6c 6f
Searching 5 bytes in [0x400000-0x601040]
hits: 1
0x004005c4 hit0_0 .HHello, world;0.

[0x00400430]> s 0x004005c4

[0x004005c4]> px
- offset - 0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
0x004005c4 6865 6c6c 6f2c 2077 6f72 6c64 0000 0000 hello, world....
0x004005d4 011b 033b 3000 0000 0500 0000 1cfe ffff ...;0.....
0x004005e4 7c00 0000 5cfe ffff 4c00 0000 52ff ffff |...\\...L...R...
0x004005f4 a400 0000 6cff ffff c400 0000 dcff ffff ....l.....
0x00400604 0c01 0000 1400 0000 0000 0000 017a 5200 .....zR.
```

## 1.5. HELLO, WORLD!

```
0x00400614 0178 1001 1b0c 0708 9001 0710 1400 0000 .x.....  
0x00400624 1c00 0000 08fe ffff 2a00 0000 0000 0000 .....*....  
0x00400634 0000 0000 1400 0000 0000 0000 017a 5200 .....zR.  
0x00400644 0178 1001 1b0c 0708 9001 0000 2400 0000 .x.....$...  
0x00400654 1c00 0000 98fd ffff 3000 0000 000e 1046 .....0.....F  
0x00400664 0e18 4a0f 0b77 0880 003f 1a3b 2a33 2422 ..J..w...?.;*3$"  
0x00400674 0000 0000 1c00 0000 4400 0000 a6fe ffff .....D.....  
0x00400684 1500 0000 0041 0e10 8602 430d 0650 0c07 .....A....C..P..  
0x00400694 0800 0000 4400 0000 6400 0000 a0fe ffff ....D...d.....  
0x004006a4 6500 0000 0042 0e10 8f02 420e 188e 0345 e....B....B....E  
0x004006b4 0e20 8d04 420e 288c 0548 0e30 8606 480e . . .B.(..H.0..H.
```

```
[0x004005c4]> oo+  
File a.out reopened in read-write mode
```

```
[0x004005c4]> w hola, mundo\x00
```

```
[0x004005c4]> q
```

```
dennis@bigbox ~/tmp % ./a.out  
holo, mundo
```

What I do here: I search for “hello” string using `/` command, then I set *cursor* (or *seek* in rada.re terms) to that address. Then I want to be sure that this is really that place: `px` dumps bytes there. `oo+` switches rada.re to *read-write* mode. `w` writes ASCII string at the current *seek*. Note `\x00` at the end—this is zero byte. `q` quits.

## Software localization of MS-DOS era

The way I described was a common way to translate MS-DOS software to Russian language back to 1980’s and 1990’s. Russian words and sentences are usually slightly longer than its English counterparts, so that is why *localized* software has a lot of weird acronyms and hardly readable abbreviations.

Perhaps, this also happened to other languages during that era.

### 1.5.2 x86-64

#### MSVC: x86-64

Let’s also try 64-bit MSVC:

Listing 1.20: MSVC 2012 x64

```
$SG2989 DB      'hello, world', 0AH, 00H  
  
main    PROC  
        sub     rsp, 40  
        lea     rcx, OFFSET FLAT:$SG2989  
        call    printf  
        xor    eax, eax  
        add    rsp, 40  
        ret    0  
main    ENDP
```

In x86-64, all registers were extended to 64-bit and now their names have an `R-` prefix. In order to use the stack less often (in other words, to access external memory/cache less often), there exists a popular way to pass function arguments via registers (*fastcall*) [6.1.3 on page 733](#). I.e., a part of the function arguments is passed in registers, the rest—via the stack. In Win64, 4 function arguments are passed in the `RCX`, `RDX`, `R8`, `R9` registers. That is what we see here: a pointer to the string for `printf()` is now passed not in the stack, but in the `RCX` register. The pointers are 64-bit now, so they are passed in the 64-bit registers (which have the `R-` prefix). However, for backward compatibility, it is still possible to access the 32-bit parts, using the `E-` prefix. This is how the `RAX/EAX/AX/AL` register looks like in x86-64:

Byte number:							
7th	6th	5th	4th	3rd	2nd	1st	0th
RAX <sup>x64</sup>							
						EAX	
						AX	
						AH	AL

The `main()` function returns an *int*-typed value, which is, in C/C++, for better backward compatibility and portability, still 32-bit, so that is why the `EAX` register is cleared at the function end (i.e., the 32-bit part of the register) instead of `RAX`. There are also 40 bytes allocated in the local stack. This is called the “shadow space”, about which we are going to talk later: [1.10.2 on page 101](#).

## GCC: x86-64

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

Listing 1.21: GCC 4.4.6 x64

```
.string "hello, world\n"
main:
    sub    rsp, 8
    mov    edi, OFFSET FLAT:.LC0 ; "hello, world\n"
    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 is [Michael Matz, Jan Hubicka, Andreas Jaeger, Mark Mitchell, *System V Application Binary Interface. AMD64 Architecture Processor Supplement*, (2013)] <sup>24</sup>.

The first 6 arguments are passed in the `RCI`, `RSI`, `RDX`, `RCX`, `R8`, `R9` registers, and the rest—via the stack.

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

It is important to keep in mind that all `MOV` instructions in 64-bit mode that write something into the lower 32-bit register part also clear the higher 32-bits (as stated in Intel manuals: [11.1.4 on page 994](#)).

I.e., the `MOV EAX, 011223344h` writes a value into `RAX` correctly, since the higher bits will be cleared.

If we open the compiled object file (.o), we can also see all the instructions' opcodes <sup>25</sup>:

Listing 1.22: 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\n"
.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 that writes into `EDI` at `0x4004D4` occupies 5 bytes. The same instruction writing a 64-bit value into `RCI` occupies 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 that the `EAX` register has been cleared before the `printf()` function call. This is done because according to **ABI**<sup>26</sup> standard mentioned above, the number of used vector registers is passed in `EAX` in \*NIX systems on x86-64.

<sup>24</sup>Also available as <https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf>

<sup>25</sup>This must be enabled in **Options** → **Disassembly** → **Number of opcode bytes**

<sup>26</sup>**ABI**!

## 1.5. *HELLO, WORLD!*

## **Address patching (Win64)**

When our example compiled in MSVC2013 using `\MD` switch (meaning smaller executable due to `MSVCR*.DLL` file linkage), the `main()` function came first and can be easily found:

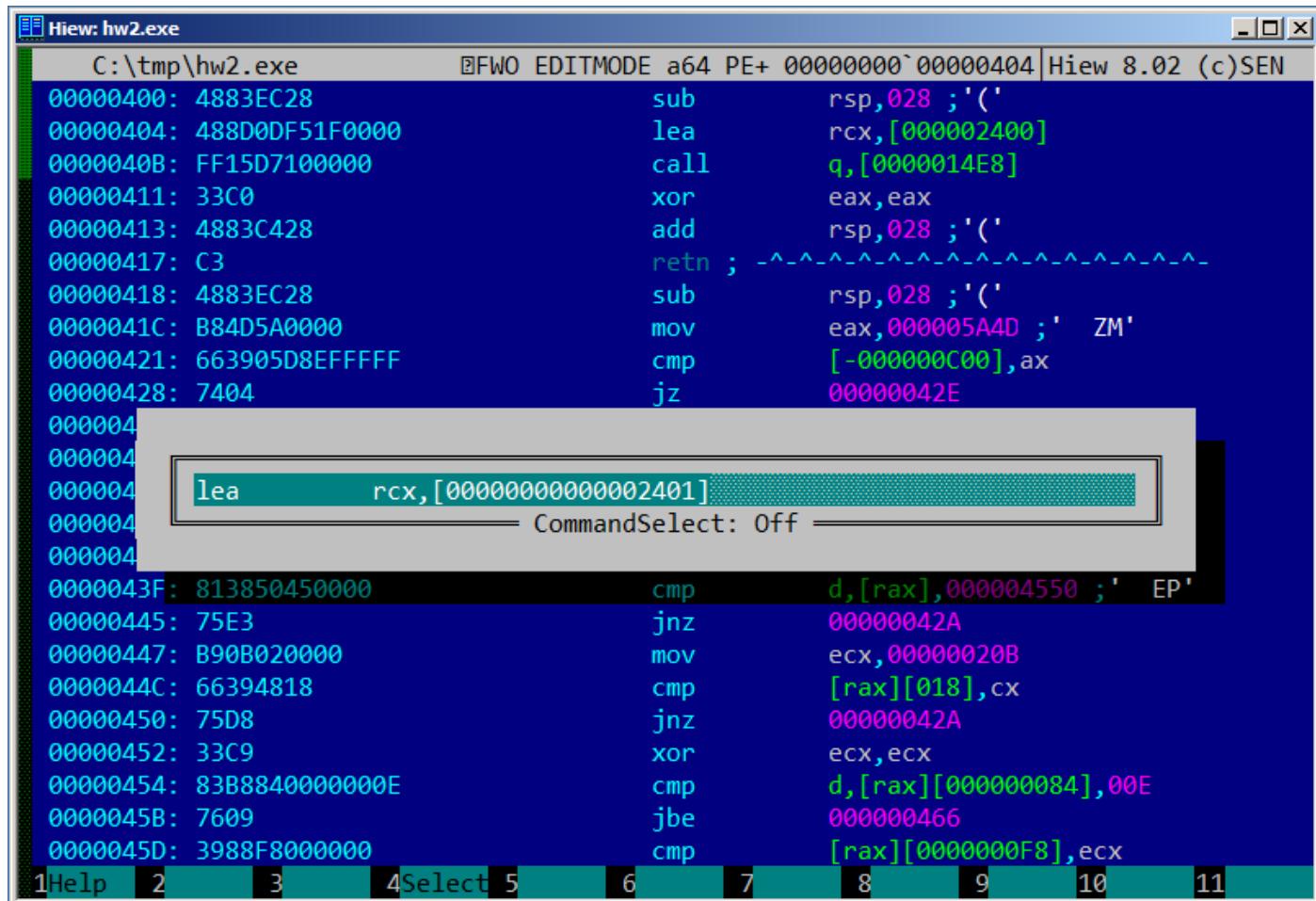
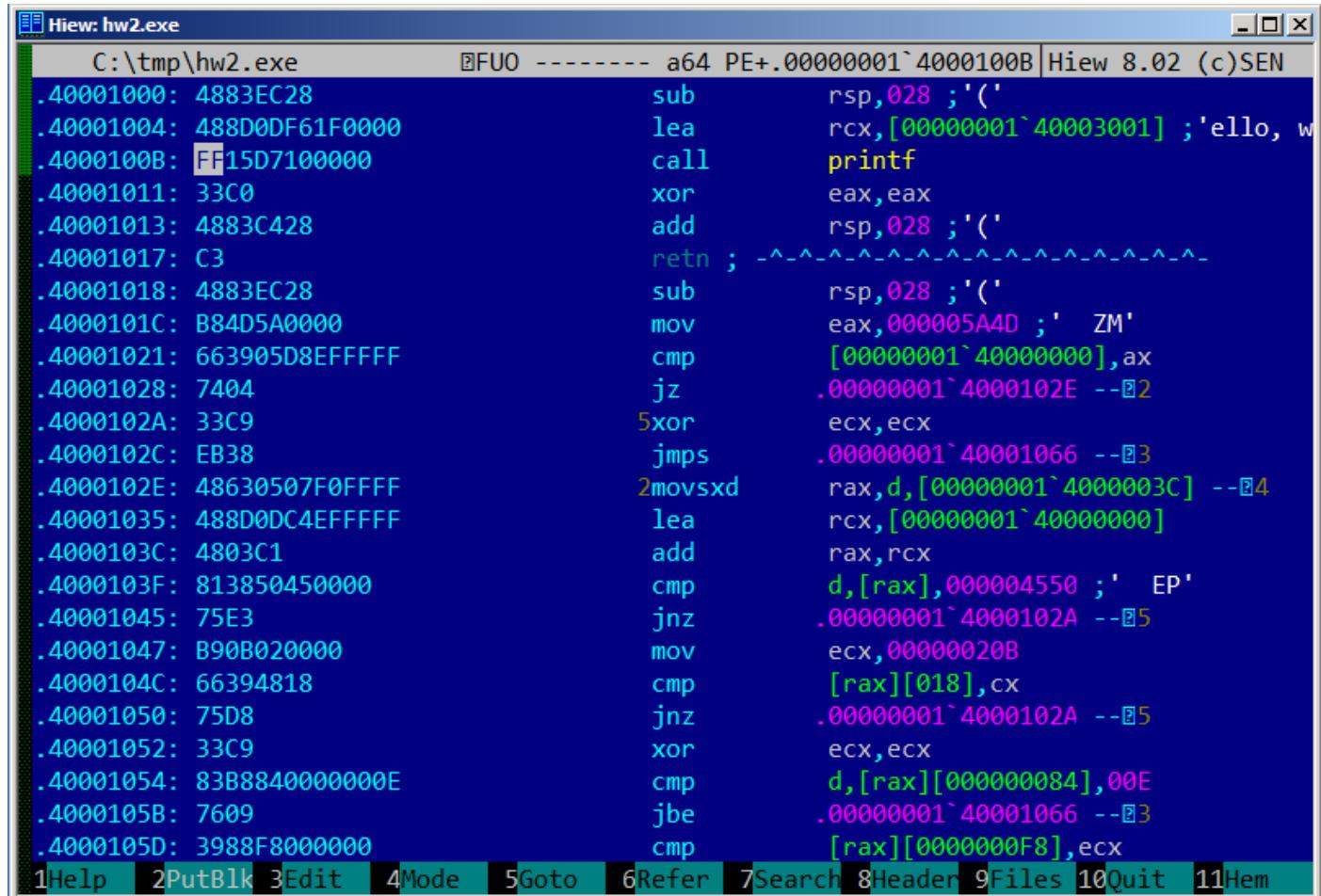


Figure 1.3: Hiew

As an experiment, we can **increment** address of the pointer by 1:

## 1.5. HELLO, WORLD!



The screenshot shows the Hiew debugger interface with the title bar "Hiew: hw2.exe". The main window displays assembly code for a 64-bit PE executable. The assembly listing includes several instructions, such as `sub`, `lea`, `call`, `xor`, `add`, and `ret`, along with memory addresses and register names. A portion of the string "Hello, world" is visible in the assembly output. The bottom of the window features a menu bar with options 1 through 11.

Figure 1.4: Hiew

Hiew shows “ello, world” string. And when we run patched executable, this very string is printed.

### Pick another string from binary image (Linux x64)

The binary file I've got when I compile our example using GCC 5.4.0 on Linux x64 box has many other text strings: they are mostly imported function names and library names.

I run objdump to get contents of all sections of the compiled file:

```
$ objdump -s a.out

a.out:      file format elf64-x86-64

Contents of section .interp:
400238 2f6c6962 36342f6c 642d6c69 6e75782d  /lib64/ld-linux-
400248 7838362d 36342e73 6f2e3200          x86-64.so.2.

Contents of section .note.ABI-tag:
400254 04000000 10000000 01000000 474e5500  ....GNU.
400264 00000000 02000000 06000000 20000000  .....

Contents of section .note.gnu.build-id:
400274 04000000 14000000 03000000 474e5500  ....GNU.
400284 fe461178 5bb710b4 bbf2aca8 5ec1ec10  .F.x[.....^...
400294 cf3f7ae4                      .?z.

...
```

It's not a problem to pass address of the text string “/lib64/ld-linux-x86-64.so.2” to `printf()` call:

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

## 1.5. HELLO, WORLD!

```
int main()
{
    printf(0x400238);
    return 0;
}
```

Hard to believe, this code prints mentioned string.

Change the address to `0x400260`, and the “GNU” string will be printed. The address is true for my specific GCC version, GNU toolset, etc. On your system, executable may be slightly different, and all addresses will also be different. Also, adding/removing code to/from this source code will probably shift all addresses back and forth.

### 1.5.3 GCC—one more thing

The fact that an *anonymous* C-string has *const* type ( [1.5.1 on page 9](#)), and that C-strings allocated in constants segment are guaranteed to be immutable, has an interesting consequence: the compiler may use a specific part of the string.

Let's try this example:

```
#include <stdio.h>

int f1()
{
    printf ("world\n");
}

int f2()
{
    printf ("hello world\n");
}

int main()
{
    f1();
    f2();
}
```

Common C/C++-compilers (including MSVC) allocate two strings, but let's see what GCC 4.8.1 does:

Listing 1.23: GCC 4.8.1 + IDA listing

```
f1          proc near
s           = dword ptr -1Ch
            sub    esp, 1Ch
            mov    [esp+1Ch+s], offset s ; "world\n"
            call   _puts
            add    esp, 1Ch
            retn
f1          endp

f2          proc near
s           = dword ptr -1Ch
            sub    esp, 1Ch
            mov    [esp+1Ch+s], offset aHello ; "hello "
            call   _puts
            add    esp, 1Ch
            retn
f2          endp

aHello      db 'hello '
s           db 'world',0xa,0
```

## 1.5. HELLO, WORLD!

Indeed: when we print the “hello world” string these two words are positioned in memory adjacently and `puts()` called from `f2()` function is not aware that this string is divided. In fact, it’s not divided; it’s divided only “virtually”, in this listing.

When `puts()` is called from `f1()`, it uses the “world” string plus a zero byte. `puts()` is not aware that there is something before this string!

This clever trick is often used by at least GCC and can save some memory. This is close to *string interning*.

Another related example is here: [3.2 on page 468](#).

### 1.5.4 ARM

For my experiments with ARM processors, several compilers were used:

- Popular in the embedded area: Keil Release 6/2013.
- Apple Xcode 4.6.3 IDE with the LLVM-GCC 4.2 compiler <sup>27</sup>.
- GCC 4.9 (Linaro) (for ARM64), available as win32-executables at <http://go.yurichev.com/17325>.

32-bit ARM code is used (including Thumb and Thumb-2 modes) in all cases in this book, if not mentioned otherwise. When we talk about 64-bit ARM here, we call it ARM64.

#### Non-optimizing Keil 6/2013 (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 <sup>28</sup>, but it is more important for us to see the instructions “as is” so let’s see the compiled result in [IDA](#).

Listing 1.24: Non-optimizing Keil 6/2013 (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
```

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}` <sup>29</sup>, works as an x86 `PUSH` instruction, writing the values of two registers (`R4` and `LR`) 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 that is not quite precise. The `PUSH` instruction is only available in Thumb mode. So, to make things less confusing, we’re doing this in [IDA](#).

This instruction first [decrements](#) the `SP`<sup>31</sup> so it points to the place in the stack that is free for new entries, then it saves the values of the `R4` and `LR` registers at the address stored in the modified `SP`.

This instruction (like the `PUSH` instruction in Thumb mode) is able to save several register values at once which can be very useful. By the way, this has no equivalent 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`. In other words, `STMFD` may be used for storing a set of registers at the specified memory address.

<sup>27</sup>It is indeed so: Apple Xcode 4.6.3 uses open-source GCC as front-end compiler and LLVM code generator

<sup>28</sup>e.g. ARM mode lacks `PUSH` / `POP` instructions

<sup>29</sup>`STMFD`<sup>30</sup>

<sup>31</sup>[stack pointer](#). `SP/ESP/RSP` in x86/x64. `SP` in ARM.

## 1.5. HELLO, WORLD!

The `ADR R0, aHelloWorld` instruction adds or subtracts the value in the `PC`<sup>32</sup> register to the offset where the `hello, world` string is located. How is the `PC` register used here, one might ask? This is called “position-independent code”<sup>33</sup>.

Such code can be executed at a non-fixed address in memory. In other words, this is `PC`-relative addressing. The `ADR` instruction takes into account the difference between the address of this instruction and the address where the string is located. This difference (offset) is always to be the same, no matter at what address our 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 memory address of our C-string.

`BL __2printf`<sup>34</sup> instruction calls the `printf()` function. Here’s how this instruction works:

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

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

That is a difference between “pure” `RISC`-processors like `ARM` and `CISC`<sup>35</sup>-processors like `x86`, where the return address is usually stored on the stack. Read more about this in next section ([1.7 on page 30](#)).

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. As we may recall, all `ARM`-mode instructions have a size of 4 bytes (32 bits). Hence, they can only be located on 4-byte boundary addresses. This implies that 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 encode  $current\_PC \pm \approx 32M$ .

Next, the `MOV R0, #0`<sup>36</sup> 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`<sup>37</sup>. It loads values from the stack (or any other memory place) in order to save them into `R4` and `PC`, and increments the `stack pointer SP`. It works like `POP` here.

N.B. The very first instruction `STMFD` saved the `R4` and `LR` registers pair on the stack, but `R4` and `PC` are restored during the `LDMFD` execution.

As we already know, the address of the place where each function must return control to is usually saved in the `LR` register. The very first instruction saves its value in the stack because the same register will be used by our `main()` function when calling `printf()`. In the function’s end, this value can be written directly to the `PC` register, thus passing control to where our function has been called.

Since `main()` is usually the primary function in `C/C++`, the control will be returned to the `OS` loader or to a point in a `CRT`, or something like that.

All that allows omitting the `BX LR` instruction at the end of the function.

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

### Non-optimizing Keil 6/2013 (Thumb mode)

Let’s compile the same example using Keil in Thumb mode:

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

We are getting (in `IDA`):

Listing 1.25: Non-optimizing Keil 6/2013 (Thumb mode) + `IDA`

```
.text:00000000          main
.text:00000000 10 B5      PUSH    {R4,LR}
```

<sup>32</sup>Program Counter. IP/EIP/RIP in `x86/64`. PC in `ARM`.

<sup>33</sup>Read more about it in relevant section ([6.4.1 on page 746](#))

<sup>34</sup>Branch with Link

<sup>35</sup>Complex instruction set computing

<sup>36</sup>Meaning MOVE

<sup>37</sup>`LDMFD`<sup>38</sup> is an inverse instruction of `STMFD`

## 1.5. HELLO, WORLD!

```
.text:00000002 C0 A0      ADR    R0, aHelloWorld ; "hello, world"
.text:00000004 06 F0 2E F9  BL     _2printf
.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 was already noted, 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 while using the small space in one 16-bit opcode. Therefore, 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 was noted, all instructions in Thumb mode have a size of 2 bytes (or 16 bits). This implies 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, the `BL` Thumb-instruction can encode an address in  $current\_PC \pm \approx 2M$ .

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

## Optimizing Xcode 4.6.3 (LLVM) (ARM mode)

Xcode 4.6.3 without optimization turned on produces a lot of redundant code so we'll study optimized output, where the instruction count is as small as possible, setting the compiler switch `-O3`.

Listing 1.26: Optimizing Xcode 4.6.3 (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 already 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 [iOS ABI Function Call Guide, (2010)]<sup>39</sup>) is a frame pointer. More on that below.

The `MOVT R0, #0` (`MOVE Top`) 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.

Keep in mind, all instruction opcodes in ARM mode are limited in size to 32 bits. Of course, this limitation is not related to moving data between registers. That's why an additional instruction `MOVT` exists for writing into the higher bits (from 16 to 31 inclusive). Its usage here, however, is redundant because the `MOV R0, #0x1686` instruction above cleared the higher part of the register. This is supposedly 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 the 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()`.

<sup>39</sup>Also available as <http://go.yurichev.com/17276>

## 1.5. HELLO, WORLD!

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

Almost, because the two functions are producing the same result only in case the string does not contain printf format identifiers starting with %. In case it does, the effect of these two functions would be different <sup>40</sup>.

Why did the compiler replace the `printf()` with `puts()`? Presumably because `puts()` is faster <sup>41</sup>.

Because it just passes characters to `stdout` without comparing every one of them with the % symbol.

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

### Optimizing Xcode 4.6.3 (LLVM) (Thumb-2 mode)

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

Listing 1.27: Optimizing Xcode 4.6.3 (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}

...
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 is obvious considering that the opcodes of the Thumb-2 instructions always begin with `0xFx` or `0xEx`.

But in the [IDA](#) listing the opcode bytes are swapped because for ARM processor the instructions are encoded as follows: last byte comes first and after that comes the first one (for Thumb and Thumb-2 modes) or for instructions in ARM mode the fourth byte comes first, then the third, then the second and finally the first (due to different [endianness](#)).

So that is how bytes are located in IDA listings:

- for ARM and ARM64 modes: 4-3-2-1;
- for Thumb mode: 2-1;
- for 16-bit instructions pair in Thumb-2 mode: 2-1-4-3.

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 stores a 16-bit value into the lower part of the `R0` register, clearing the higher bits.

Also, `MOVT.W R0, #0` works just like `MOVT` from the previous example only it works in Thumb-2.

Among the other differences, the `BLX` instruction is used in this case instead of the `BL`.

The difference is that, besides saving the `RA`<sup>42</sup> in the `LR` register and passing control to the `puts()` function, the processor is also switching from Thumb/Thumb-2 mode to ARM mode (or back).

This instruction is placed here since the instruction to which control is passed looks like (it is encoded in ARM mode):

<sup>40</sup>It has also to 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.

<sup>41</sup>[ciselant.de/projects/gcc\\_printf/gcc\\_printf.html](http://ciselant.de/projects/gcc_printf/gcc_printf.html)

<sup>42</sup>Return Address

## 1.5. HELLO, WORLD!

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

This is essentially a jump to the place where the address of `puts()` is written in the imports' section.

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). The dynamic libraries contain frequently used library functions, 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) imported from external modules along with the names of the modules themselves.

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 used by the [OS](#) loader to store the correct address of the function in an external library. Then the `LDR` instruction just reads 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 the [OS](#) loader needs for completing this procedure, it is good idea to write the address of each symbol only once, to a dedicated place.

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.

Therefore, the optimal solution is to allocate a separate function working in ARM mode with the sole goal of passing control to the dynamic library and then to jump to this short one-instruction function (the so-called [thunk function](#)) from the Thumb-code.

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

### More about thunk-functions

Thunk-functions are hard to understand, apparently, because of a misnomer. The simplest way to understand it as adaptors or converters of one type of jack to another. For example, an adaptor allowing the insertion of a British power plug into an American wall socket, or vice-versa. Thunk functions are also sometimes called *wrappers*.

Here are a couple more descriptions of these functions:

"A piece of coding which provides an address:", according to P. Z. Ingberman, who invented thunks in 1961 as a way of binding actual parameters to their formal definitions in Algol-60 procedure calls. If a procedure is called with an expression in the place of a formal parameter, the compiler generates a thunk which computes the expression and leaves the address of the result in some standard location.

...  
Microsoft and IBM have both defined, in their Intel-based systems, a "16-bit environment" (with bletcherous segment registers and 64K address limits) and a "32-bit environment" (with flat addressing and semi-real memory management). The two environments can both be running on the same computer and OS (thanks to what is called, in the Microsoft world, WOW which stands for Windows On Windows). MS and IBM have both decided that the process of getting from 16- to 32-bit and vice versa is called a "thunk"; for Windows 95, there is even a tool, THUNK.EXE, called a "thunk compiler".

### ( The Jargon File )

Another example we can find in LAPACK library—a "Linear Algebra PACKage" written in FORTRAN. C/C++ developers also want to use LAPACK, but it's insane to rewrite it to C/C++ and then maintain several

## 1.5. HELLO, WORLD!

versions. So there are short C functions callable from C/C++ environment, which are, in turn, call FORTRAN functions, and do almost anything else:

```
double Blas_Dot_Prod(const LaVectorDouble &dx, const LaVectorDouble &dy)
{
    assert(dx.size()==dy.size());
    integer n = dx.size();
    integer incx = dx.inc(), incy = dy.inc();

    return F77NAME(ddot)(&n, &dx(0), &incx, &dy(0), &incy);
}
```

Also, functions like that are called “wrappers”.

## ARM64

### GCC

Let's compile the example using GCC 4.8.1 in ARM64:

Listing 1.28: Non-optimizing GCC 4.8.1 + objdump

```
1 0000000000400590 <main>:
2  400590: a9bf7bfd      stp    x29, x30, [sp,#-16]!
3  400594: 910003fd      mov    x29, sp
4  400598: 90000000      adrp   x0, 4000000 <_init-0x3b8>
5  40059c: 91192000      add    x0, x0, #0x648
6  4005a0: 97ffffa0      bl     400420 <puts@plt>
7  4005a4: 52800000      mov    w0, #0x0           // #0
8  4005a8: a8c17bfd      ldp    x29, x30, [sp],#16
9  4005ac: d65f03c0      ret
10 ...
11 ...
12 ...
13 Contents of section .rodata:
14 400640 01000200 00000000 48656c6c 6f210a00 .....Hello!..
```

There are no Thumb and Thumb-2 modes in ARM64, only ARM, so there are 32-bit instructions only. The Register count is doubled: [2.4 on page 1022](#). 64-bit registers have **X-** prefixes, while its 32-bit parts—**W-**.

The **STP** instruction (*Store Pair*) saves two registers in the stack simultaneously: **X29** and **X30**.

Of course, this instruction is able to save this pair at an arbitrary place in memory, but the **SP** register is specified here, so the pair is saved in the stack.

ARM64 registers are 64-bit ones, each has a size of 8 bytes, so one needs 16 bytes for saving two registers.

The exclamation mark (“!”) after the operand means that 16 is to be subtracted from **SP** first, and only then are values from register pair to be written into the stack. This is also called *pre-index*. About the difference between *post-index* and *pre-index* read here: [1.32.2 on page 441](#).

Hence, in terms of the more familiar x86, the first instruction is just an analogue to a pair of **PUSH X29** and **PUSH X30**. **X29** is used as **FP**<sup>43</sup> in ARM64, and **X30** as **LR**, so that's why they are saved in the function prologue and restored in the function epilogue.

The second instruction copies **SP** in **X29** (or **FP**). This is made so to set up the function stack frame.

**ADRP** and **ADD** instructions are used to fill the address of the string “Hello!” into the **X0** register, because the first function argument is passed in this register. There are no instructions, whatsoever, in ARM that can store a large number into a register (because the instruction length is limited to 4 bytes, read more about it here: [1.32.3 on page 442](#)). So several instructions must be utilized. The first instruction (**ADRP**) writes the address of the 4KiB page, where the string is located, into **X0**, and the second one (**ADD**) just adds the remainder to the address. More about that in: [1.32.4 on page 444](#).

<sup>43</sup>Frame Pointer

## 1.5. HELLO, WORLD!

`0x400000 + 0x648 = 0x400648`, and we see our “Hello!” C-string in the `.rodata` data segment at this address.

`puts()` is called afterwards using the `BL` instruction. This was already discussed: [1.5.4 on page 21](#).

`MOV` writes 0 into `W0`. `W0` is the lower 32 bits of the 64-bit `X0` register:

High 32-bit part	low 32-bit part
	<code>X0</code>
	<code>W0</code>

The function result is returned via `X0` and `main()` returns 0, so that's how the return result is prepared. But why use the 32-bit part?

Because the `int` data type in ARM64, just like in x86-64, is still 32-bit, for better compatibility.

So if a function returns a 32-bit `int`, only the lower 32 bits of `X0` register have to be filled.

In order to verify this, let's change this example slightly and recompile it. Now `main()` returns a 64-bit value:

Listing 1.29: `main()` returning a value of `uint64_t` type

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

uint64_t main()
{
    printf ("Hello!\n");
    return 0;
}
```

The result is the same, but that's how `MOV` at that line looks like now:

Listing 1.30: Non-optimizing GCC 4.8.1 + objdump

4005a4:	d2800000	mov	x0, #0x0	// #0
---------	----------	-----	----------	-------

`LDP` (*Load Pair*) then restores the `X29` and `X30` registers.

There is no exclamation mark after the instruction: this implies that the values are first loaded from the stack, and only then is `SP` increased by 16. This is called *post-index*.

A new instruction appeared in ARM64: `RET`. It works just as `BX LR`, only a special *hint* bit is added, informing the `CPU` that this is a return from a function, not just another jump instruction, so it can execute it more optimally.

Due to the simplicity of the function, optimizing GCC generates the very same code.

## 1.5.5 MIPS

### A word about the “global pointer”

One important MIPS concept is the “global pointer”. As we may already know, each MIPS instruction has a size of 32 bits, so it's impossible to embed a 32-bit address into one instruction: a pair has to be used for this (like GCC did in our example for the text string address loading). It's possible, however, to load data from the address in the range of `register - 32768...register + 32767` using one single instruction (because 16 bits of signed offset could be encoded in a single instruction). So we can allocate some register for this purpose and also allocate a 64KiB area of most used data. This allocated register is called a “global pointer” and it points to the middle of the 64KiB area. This area usually contains global variables and addresses of imported functions like `printf()`, because the GCC developers decided that getting the address of some function must be as fast as a single instruction execution instead of two. In an ELF file this 64KiB area is located partly in sections `.sbss` (“small BSS<sup>44</sup>”) for uninitialized data and `.sdata` (“small data”) for initialized data. This implies that the programmer may choose what data he/she wants to be accessed fast and place it into `.sdata/.sbss`. Some old-school programmers may recall the MS-DOS

<sup>44</sup>Block Started by Symbol

## 1.5. HELLO, WORLD!

memory model [10.6 on page 990](#) or the MS-DOS memory managers like XMS/EMS where all memory was divided in 64KiB blocks.

This concept is not unique to MIPS. At least PowerPC uses this technique as well.

### Optimizing GCC

Lets consider the following example, which illustrates the “global pointer” concept.

Listing 1.31: Optimizing GCC 4.4.5 (assembly output)

```
1 $LC0:  
2 ; \000 is zero byte in octal base:  
3     .ascii  "Hello, world!\012\000"  
4 main:  
5 ; function prologue.  
6 ; set the GP:  
7     lui      $28,%hi(__gnu_local_gp)  
8     addiu   $sp,$sp,-32  
9     addiu   $28,$28,%lo(__gnu_local_gp)  
10 ; save the RA to the local stack:  
11    sw      $31,28($sp)  
12 ; load the address of the puts() function from the GP to $25:  
13    lw      $25,%call16(puts)($28)  
14 ; load the address of the text string to $4 ($a0):  
15    lui      $4,%hi($LC0)  
16 ; jump to puts(), saving the return address in the link register:  
17    jalr    $25  
18    addiu   $4,$4,%lo($LC0) ; branch delay slot  
19 ; restore the RA:  
20    lw      $31,28($sp)  
21 ; copy 0 from $zero to $v0:  
22    move    $2,$0  
23 ; return by jumping to the RA:  
24    j      $31  
25 ; function epilogue:  
26    addiu   $sp,$sp,32 ; branch delay slot + free local stack
```

As we see, the \$GP register is set in the function prologue to point to the middle of this area. The [RA](#) register is also saved in the local stack. [puts\(\)](#) is also used here instead of [printf\(\)](#). The address of the [puts\(\)](#) function is loaded into \$25 using [LW](#) the instruction (“Load Word”). Then the address of the text string is loaded to \$4 using [LUI](#) (“Load Upper Immediate”) and [ADDIU](#) (“Add Immediate Unsigned Word”) instruction pair. [LUI](#) sets the high 16 bits of the register (hence “upper” word in instruction name) and [ADDIU](#) adds the lower 16 bits of the address.

[ADDIU](#) follows [JALR](#) (haven’t you forgot *branch delay slots* yet?). The register \$4 is also called \$A0, which is used for passing the first function argument <sup>45</sup>.

[JALR](#) (“Jump and Link Register”) jumps to the address stored in the \$25 register (address of [puts\(\)](#)) while saving the address of the next instruction (LW) in [RA](#). This is very similar to ARM. Oh, and one important thing is that the address saved in [RA](#) is not the address of the next instruction (because it’s in a *delay slot* and is executed before the jump instruction), but the address of the instruction after the next one (after the *delay slot*). Hence,  $PC + 8$  is written to [RA](#) during the execution of [JALR](#), in our case, this is the address of the [LW](#) instruction next to [ADDIU](#).

[LW](#) (“Load Word”) at line 20 restores [RA](#) from the local stack (this instruction is actually part of the function epilogue).

[MOVE](#) at line 22 copies the value from the \$0 (\$ZERO) register to \$2 (\$V0).

MIPS has a *constant* register, which always holds zero. Apparently, the MIPS developers came up with the idea that zero is in fact the busiest constant in the computer programming, so let’s just use the \$0 register every time zero is needed.

<sup>45</sup>The MIPS registers table is available in appendix [3.1 on page 1023](#)

## 1.5. HELLO, WORLD!

Another interesting fact is that MIPS lacks an instruction that transfers data between registers. In fact, `MOVE DST, SRC` is `ADD DST, SRC, $ZERO` ( $DST = SRC + 0$ ), which does the same. Apparently, the MIPS developers wanted to have a compact opcode table. This does not mean an actual addition happens at each `MOVE` instruction. Most likely, the CPU optimizes these pseudo instructions and the ALU<sup>46</sup> is never used.

`J` at line 24 jumps to the address in `RA`, which is effectively performing a return from the function. `ADDIU` after `J` is in fact executed before `J` (remember *branch delay slots?*) and is part of the function epilogue. Here is also a listing generated by [IDA](#). Each register here has its own pseudo name:

Listing 1.32: Optimizing GCC 4.4.5 (IDA)

```
1 .text:00000000 main:
2 .text:00000000
3 .text:00000000 var_10          = -0x10
4 .text:00000000 var_4           = -4
5 .text:00000000
6 ; function prologue.
7 ; set the GP:
8 .text:00000000        lui      $gp, (__gnu_local_gp >> 16)
9 .text:00000004        addiu   $sp, -0x20
10 .text:00000008       la      $gp, (__gnu_local_gp & 0xFFFF)
11 ; save the RA to the local stack:
12 .text:0000000C       sw      $ra, 0x20+var_4($sp)
13 ; save the GP to the local stack:
14 ; for some reason, this instruction is missing in the GCC assembly output:
15 .text:00000010       sw      $gp, 0x20+var_10($sp)
16 ; load the address of the puts() function from the GP to $t9:
17 .text:00000014       lw      $t9, (puts & 0xFFFF)($gp)
18 ; form the address of the text string in $a0:
19 .text:00000018       lui      $a0, ($LC0 >> 16) # "Hello, world!"
20 ; jump to puts(), saving the return address in the link register:
21 .text:0000001C       jalr    $t9
22 .text:00000020       la      $a0, ($LC0 & 0xFFFF) # "Hello, world!"
23 ; restore the RA:
24 .text:00000024       lw      $ra, 0x20+var_4($sp)
25 ; copy 0 from $zero to $v0:
26 .text:00000028       move    $v0, $zero
27 ; return by jumping to the RA:
28 .text:0000002C       jr      $ra
29 ; function epilogue:
30 .text:00000030       addiu  $sp, 0x20
```

The instruction at line 15 saves the GP value into the local stack, and this instruction is missing mysteriously from the GCC output listing, maybe by a GCC error <sup>47</sup>. The GP value has to be saved indeed, because each function can use its own 64KiB data window. The register containing the `puts()` address is called `$T9`, because registers prefixed with T are called “temporaries” and their contents may not be preserved.

## Non-optimizing GCC

Non-optimizing GCC is more verbose.

Listing 1.33: Non-optimizing GCC 4.4.5 (assembly output)

```
1 $LC0:
2     .ascii  "Hello, world!\012\000"
3 main:
4 ; function prologue.
5 ; save the RA ($31) and FP in the stack:
6     addiu  $sp,$sp,-32
7     sw     $31,28($sp)
8     sw     $fp,24($sp)
9 ; set the FP (stack frame pointer):
10    move   $fp,$sp
11 ; set the GP:
```

<sup>46</sup>Arithmetic logic unit

<sup>47</sup>Apparently, functions generating listings are not so critical to GCC users, so some unfixed errors may still exist.

## 1.5. HELLO, WORLD!

```
12      lui      $28,%hi(__gnu_local_gp)
13      addiu   $28,$28,%lo(__gnu_local_gp)
14 ; load the address of the text string:
15      lui      $2,%hi($LC0)
16      addiu   $4,$2,%lo($LC0)
17 ; load the address of puts() using the GP:
18      lw       $2,%call16(puts)($28)
19      nop
20 ; call puts():
21      move    $25,$2
22      jalr    $25
23      nop ; branch delay slot
24
25 ; restore the GP from the local stack:
26      lw       $28,16($fp)
27 ; set register $2 ($V0) to zero:
28      move    $2,$0
29 ; function epilogue.
30 ; restore the SP:
31      move    $sp,$fp
32 ; restore the RA:
33      lw       $31,28($sp)
34 ; restore the FP:
35      lw       $fp,24($sp)
36      addiu   $sp,$sp,32
37 ; jump to the RA:
38      j       $31
39      nop ; branch delay slot
```

We see here that register FP is used as a pointer to the stack frame. We also see 3 NOPs. The second and third of which follow the branch instructions. Perhaps the GCC compiler always adds NOPs (because of *branch delay slots*) after branch instructions and then, if optimization is turned on, maybe eliminates them. So in this case they are left here.

Here is also [IDA](#) listing:

Listing 1.34: Non-optimizing GCC 4.4.5 ([IDA](#))

```
1 .text:00000000 main:
2 .text:00000000
3 .text:00000000 var_10          = -0x10
4 .text:00000000 var_8           = -8
5 .text:00000000 var_4           = -4
6 .text:00000000
7 ; function prologue.
8 ; save the RA and FP in the stack:
9 .text:00000000      addiu  $sp, -0x20
10 .text:00000004      sw     $ra, 0x20+var_4($sp)
11 .text:00000008      sw     $fp, 0x20+var_8($sp)
12 ; set the FP (stack frame pointer):
13 .text:0000000C      move   $fp, $sp
14 ; set the GP:
15 .text:00000010      la     $gp, __gnu_local_gp
16 .text:00000018      sw     $gp, 0x20+var_10($sp)
17 ; load the address of the text string:
18 .text:0000001C      lui    $v0, (aHelloWorld >> 16) # "Hello, world!"
19 .text:00000020      addiu $a0, $v0, (aHelloWorld & 0xFFFF) # "Hello, world!"
20 ; load the address of puts() using the GP:
21 .text:00000024      lw    $v0, (puts & 0xFFFF)($gp)
22 .text:00000028      or     $at, $zero ; NOP
23 ; call puts():
24 .text:0000002C      move  $t9, $v0
25 .text:00000030      jalr $t9
26 .text:00000034      or     $at, $zero ; NOP
27 ; restore the GP from local stack:
28 .text:00000038      lw    $gp, 0x20+var_10($fp)
29 ; set register $2 ($V0) to zero:
30 .text:0000003C      move  $v0, $zero
31 ; function epilogue.
32 ; restore the SP:
33 .text:00000040      move  $sp, $fp
```

## 1.5. HELLO, WORLD!

```
34 ; restore the RA:  
35 .text:00000044          lw      $ra, 0x20+var_4($sp)  
36 ; restore the FP:  
37 .text:00000048          lw      $fp, 0x20+var_8($sp)  
38 .text:0000004C          addiu $sp, 0x20  
39 ; jump to the RA:  
40 .text:00000050          jr      $ra  
41 .text:00000054          or      $at, $zero ; NOP
```

Interestingly, IDA recognized the LUI / ADDIU instructions pair and coalesced them into one LA ("Load Address") pseudo instruction at line 15. We may also see that this pseudo instruction has a size of 8 bytes! This is a pseudo instruction (or *macro*) because it's not a real MIPS instruction, but rather a handy name for an instruction pair.

Another thing is that IDA doesn't recognize NOP instructions, so here they are at lines 22, 26 and 41. It is OR \$AT, \$ZERO. Essentially, this instruction applies the OR operation to the contents of the \$AT register with zero, which is, of course, an idle instruction. MIPS, like many other ISAs, doesn't have a separate NOP instruction.

### Role of the stack frame in this example

The address of the text string is passed in the register. Why setup a local stack anyway? The reason for this lies in the fact that the values of registers RA and GP have to be saved somewhere (because printf() is called), and the local stack is used for this purpose. If this was a leaf function, it would have been possible to get rid of the function prologue and epilogue, for example: [1.4.3 on page 8](#).

### Optimizing GCC: load it into GDB

Listing 1.35: sample GDB session

```
root@debian-mips:~# gcc hw.c -O3 -o hw  
root@debian-mips:~# gdb hw  
GNU gdb (GDB) 7.0.1-debian  
...  
Reading symbols from /root/hw...(no debugging symbols found)...done.  
(gdb) b main  
Breakpoint 1 at 0x400654  
(gdb) run  
Starting program: /root/hw  
  
Breakpoint 1, 0x00400654 in main ()  
(gdb) set step-mode on  
(gdb) disas  
Dump of assembler code for function main:  
0x00400640 <main+0>: lui      gp,0x42  
0x00400644 <main+4>: addiu   sp,sp,-32  
0x00400648 <main+8>: addiu   gp,sp,-30624  
0x0040064c <main+12>: sw      ra,28(sp)  
0x00400650 <main+16>: sw      gp,16(sp)  
0x00400654 <main+20>: lw      t9,-32716(gp)  
0x00400658 <main+24>: lui      a0,0x40  
0x0040065c <main+28>: jalr    t9  
0x00400660 <main+32>: addiu   a0,a0,2080  
0x00400664 <main+36>: lw      ra,28(sp)  
0x00400668 <main+40>: move    v0,zero  
0x0040066c <main+44>: jr      ra  
0x00400670 <main+48>: addiu   sp,sp,32  
End of assembler dump.  
(gdb) s  
0x00400658 in main ()  
(gdb) s  
0x0040065c in main ()  
(gdb) s  
0x2ab2de60 in printf () from /lib/libc.so.6  
(gdb) x/s $a0
```

## 1.6. FUNCTION PROLOGUE AND EPILOGUE

```
0x400820:      "hello, world"
(gdb)
```

### 1.5.6 Conclusion

The main difference between x86/ARM and x64/ARM64 code is that the pointer to the string is now 64-bits in length. Indeed, modern CPUs are now 64-bit due to both the reduced cost of memory and the greater demand for it by modern applications. We can add much more memory to our computers than 32-bit pointers are able to address. As such, all pointers are now 64-bit.

### 1.5.7 Exercises

- <http://challenges.re/48>
- <http://challenges.re/49>

## 1.6 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: save the value in the `EBP` register, set the value of the `EBP` register to the value of the `ESP` and then allocate space on the stack for local variables.

The value in the `EBP` stays the same over the period of the function execution and is to be used for local variables and arguments access. For the same purpose one can use `ESP`, but since it changes over time this approach is not too convenient.

The function epilogue frees the allocated space in the stack, returns the value in the `EBP` register back to its initial state and returns the control flow to the [caller](#):

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

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

### 1.6.1 Recursion

Epilogues and prologues can negatively affect the recursion performance.

More about recursion in this book: [3.4.3 on page 480](#).

## 1.7 Stack

The stack is one of the most fundamental data structures in computer science <sup>48</sup>. AKA<sup>49</sup> LIFO<sup>50</sup>.

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 that block.

<sup>48</sup>[wikipedia.org/wiki/Call\\_stack](https://en.wikipedia.org/wiki/Call_stack)

<sup>49</sup> Also Known As

<sup>50</sup>Last In First Out

## 1.7. STACK

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

`POP` is the reverse operation: retrieve the data from the memory location that `SP` points to, load it into the instruction operand (often a register) and then add 4 (or 8) to the `stack pointer`.

After stack allocation, the `stack pointer` points at the bottom of the stack. `PUSH` decreases the `stack pointer` and `POP` increases it. The bottom 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.

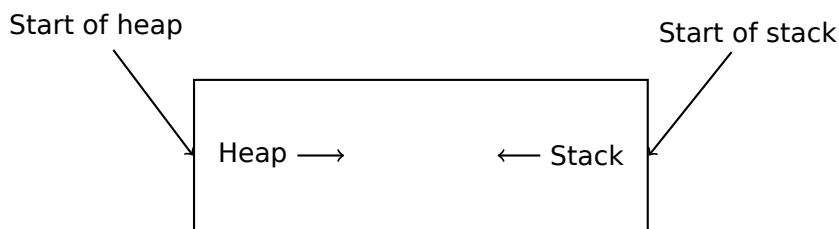
ARM supports both descending and ascending stacks.

For example the `STMFD/LDMFD`, `STMED51/LDMED52` instructions are intended to deal with a descending stack (grows downwards, starting with a high address and progressing to a lower one). The `STMFA53/LDMFA54`, `STMEA55/LDMEA56` instructions are intended to deal with an ascending stack (grows upwards, starting from a low address and progressing to a higher one).

### 1.7.1 Why does the stack grow backwards?

Intuitively, we might think that the stack grows upwards, i.e. towards higher addresses, like any other data structure.

The reason that the stack grows backward is probably historical. When the 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 [D. M. Ritchie and K. Thompson, *The UNIX Time Sharing System*, (1974)]<sup>57</sup> 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.

This reminds us how some students write two lecture notes using only one notebook: notes for the first lecture are written as usual, and notes for the second one are written from the end of notebook, by flipping it. Notes may meet each other somewhere in between, in case of lack of free space.

### 1.7.2 What is the stack used for?

#### Save the function's return address

#### x86

<sup>51</sup>Store Multiple Empty Descending (ARM instruction)

<sup>52</sup>Load Multiple Empty Descending (ARM instruction)

<sup>53</sup>Store Multiple Full Ascending (ARM instruction)

<sup>54</sup>Load Multiple Full Ascending (ARM instruction)

<sup>55</sup>Store Multiple Empty Ascending (ARM instruction)

<sup>56</sup>Load Multiple Empty Ascending (ARM instruction)

<sup>57</sup>Also available as <http://go.yurichev.com/17270>

## 1.7. STACK

When 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 —that 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 the compiler optimization (`/Ox` option) the optimized code will not overflow the stack and will work correctly<sup>58</sup> instead:

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

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

## ARM

ARM programs also use the stack for saving return addresses, but differently. As mentioned in “Hello, world!” (1.5.4 on page 19), the `RA` is saved to the `LR` (link register). If one needs, however, to call another function and use the `LR` register one more time, its value has to 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`.

<sup>58</sup>irony here

## 1.7. STACK

Nevertheless, if a function never calls any other function, in RISC terminology it is called a *leaf function*<sup>59</sup>. As a consequence, leaf functions do not save the `LR` register (because they don't modify it). If such 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, which can be faster than on older x86 machines because external RAM is not used for the stack<sup>60</sup>. This can be also useful for situations when memory for the stack is not yet allocated or not available.

Some examples of leaf functions: [1.10.3 on page 104](#), [1.10.3 on page 104](#), [1.277 on page 315](#), [1.293 on page 334](#), [1.22.5 on page 334](#), [1.185 on page 210](#), [1.183 on page 208](#), [1.202 on page 226](#).

## 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, 12 ; 4*3=12
```

Callee functions get their arguments via the stack pointer.

Therefore, this is how the argument values are located in the stack before the execution of the `f()` function's very first instruction:

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

For more information on other calling conventions see also section ([6.1 on page 732](#)).

By the way, the `callee` function does not have any information about how many arguments were passed. C functions with a variable number of arguments (like `printf()`) determine their number using format string specifiers (which begin with the % symbol).

If we write something like:

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

`printf()` will print 1234, and then two random numbers<sup>61</sup>, which were lying next to it in the stack.

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, the `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 are not used. If you declare `main()` as `main(int argc, char *argv[])`, you will be able to use first two arguments, and the third will remain “invisible” for your function. Even more, it is possible to declare `main(int argc)`, and it will work.

<sup>59</sup>[infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka13785.html](http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka13785.html)

<sup>60</sup>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 considered that having a big number of small functions is an *anti-pattern* [Eric S. Raymond, *The Art of UNIX Programming*, (2003)Chapter 4, Part II].

<sup>61</sup>Not random in strict sense, but rather unpredictable: [1.7.4 on page 38](#)

## Alternative ways of passing arguments

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.

A somewhat popular way among assembly language newbies is to pass arguments via global variables, like:

Listing 1.36: Assembly code

```
...
    mov     X, 123
    mov     Y, 456
    call    do_something

...
X      dd      ?
Y      dd      ?

do_something proc near
    ; take X
    ; take Y
    ; do something
    retn
do_something endp
```

But this method has obvious drawback: `do_something()` function cannot call itself recursively (or via another function), because it has to zap its own arguments. The same story with local variables: if you hold them in global variables, the function couldn't call itself. And this is also not thread-safe<sup>62</sup>. A method to store such information in stack makes this easier—it can hold as many function arguments and/or values, as much space it has.

[Donald E. Knuth, *The Art of Computer Programming*, Volume 1, 3rd ed., (1997), 189] mentions even weirder schemes particularly convenient on IBM System/360.

MS-DOS had a way of passing all function arguments via registers, for example, this is piece of code for ancient 16-bit MS-DOS prints "Hello, world!":

```
mov dx, msg      ; address of message
mov ah, 9        ; 9 means "print string" function
int 21h         ; DOS "syscall"

mov ah, 4ch      ; "terminate program" function
int 21h         ; DOS "syscall"

msg db 'Hello, World!\$'
```

This is quite similar to [6.1.3 on page 733](#) method. And also it's very similar to calling syscalls in Linux ([6.3.1 on page 746](#)) and Windows.

If a MS-DOS function is going to return a boolean value (i.e., single bit, usually indicating error state), `CF` flag was often used.

For example:

```
mov ah, 3ch      ; create file
lea dx, filename
mov cl, 1
int 21h
jc error
mov file_handle, ax
...
error:
...
```

In case of error, `CF` flag is raised. Otherwise, handle of newly created file is returned via `AX`.

<sup>62</sup>Correctly implemented, each thread would have its own stack with its own arguments/variables.

## 1.7. STACK

This method is still used by assembly language programmers. In Windows Research Kernel source code (which is quite similar to Windows 2003) we can find something like this (file `base/ntos/ke/i386/cpu.asm`):

```
public Get386Stepping
Get386Stepping proc

    call MultiplyTest          ; Perform multiplication test
    jnc short G3s00            ; if nc, muttest is ok
    mov ax, 0
    ret

G3s00:
    call Check386B0           ; Check for B0 stepping
    jnc short G3s05            ; if nc, it's B1/later
    mov ax, 100h                ; It is B0/earlier stepping
    ret

G3s05:
    call Check386D1           ; Check for D1 stepping
    jc short G3s10              ; if c, it is NOT D1
    mov ax, 301h                ; It is D1/later stepping
    ret

G3s10:
    mov ax, 101h                ; assume it is B1 stepping
    ret

    ...

MultiplyTest proc

    xor cx,cx                  ; 64K times is a nice round number
mlt00: push cx
    call Multiply               ; does this chip's multiply work?
    pop cx
    jc short mltx              ; if c, No, exit
    loop mlt00                  ; if nc, YES, loop to try again
    clc
mltx:
    ret

MultiplyTest endp
```

## Local variable storage

A function could allocate space in the stack for its local variables just by decreasing the [stack pointer](#) towards the stack bottom.

Hence, it's very fast, no matter how many local variables are defined. It is also not a requirement to store local variables in the stack. You could store local variables wherever you like, but traditionally this is how it's done.

## x86: `alloca()` function

It is worth noting the `alloca()` function <sup>63</sup>. This function works like `malloc()`, but allocates memory directly on the stack. The allocated memory chunk does not have to be freed via a `free()` function call, since the function epilogue ([1.6 on page 30](#)) returns `ESP` back to its initial state and the allocated memory is just *dropped*. 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:

<sup>63</sup>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`

67\$ & .

LIGHI BB\*18&BB  
LQFOXGH DOORFD K! \* &  
HOVH  
LQFOXGH PDOORF K! 069&  
HQGLI  
LQFOXGH VWGLR K!

YRLG I

^  
FKDU EXI FKDU DOORFD  
LIGHI BB\*18&BB  
VQSULQWI EXI KL G G?Q \* &&  
HOVH  
BVQSULQWI EXI KL G G?Q 069&  
HQGLI  
SXWV EXI

BVQSULQWXQFWLRQ ZRUN SULQWIOLXNM LQVWHDG RI GXPSLQJWWKGRUWW X O/W  
WHUPLQDO RU FRQVROH EXIEUXLQWV LWC SXWVHR SLHV WKH FR EX HWRWWGRIX W  
2IFRXUVH WKHVH WZR IXQFWLRQ FDOOV SULQWEHDUDOS QEXFHZHEK IRQHH WR LOO X  
EX@HU XVDJH

069 &

/HWUV FRPSLOH 069&

/LVWLQJ 069&

PRY HD[ +  
FDOO BBDOORFDDBSUREHB  
PRY HVL HVS

SXVK  
SXVK  
SXVK  
SXVK 2))6(7 6\*  
SXVK +  
SXVK HVL  
FDOO BBVQSULQWI  
  
SXVK HVL  
FDOO BSXWV  
DGG HVS

7KH VI DOORFD DUJXPHQW LV SD (\$;HGQALWDHDG RI SXVKLQJ LW LQWR WKH VWDF

\* & & , QWHO V\QWD[

\* & & GRHV WKH VDPH ZLWKRXW FDOOLQJ H[WHUQDO IXQFWLRQV

/LVWLQJ \* & &

/ &

,W LV EHFDXVH DOORFD LV UDWKHU D RQPSOLJH UMLQD/QUQ QRUFPDO IXQFWLRQ 2QH RI WKH UH  
VHSUDWH IXQFWLRQ LQVWHDG RI MXVWD FRXSOH RILQVWU&FWDQFRFDQ WPSHOFHPHQW D VEFHQX  
ZKLFK UHDGV IURP WKH PHPRU\ MXVW DOOREBWDHS SKQVRBQG UP MFRROHWW RWKHLRVQ \$IV DOORFD  
FD((63SRLQWV WR WKH EORFN RI E\WHV DQG ZH FDQ EXHDWW D\ PPHPRU\ IRU WKH

67\$&.

VWULQJ KL G G G?Q

I  
SXVK HES  
PRY HES HVS  
SXVK HE[  
VXE HVS  
OHD HE[ >HVS @  
DQG HE[ DOLJQ SRLQWHU E\ ELW ERUGHU  
PRY ':25' 375 >HVS@ HE[ V  
PRY ':25' 375 >HVS @  
PRY ':25' 375 >HVS @  
PRY ':25' 375 >HVS @  
PRY ':25' 375 >HVS @ 2))6(7 )/\$7 /& KL G G G?Q  
PRY ':25' 375 >HVS @ PD[OHQ  
FDOO BVQSULQWI  
PRY ':25' 375 >HVS@ HE[ V  
FDOO SXWV  
PRY HE[ ':25' 375 >HES @  
OHDYH  
UHW

\* & & \$7 7 V\QWD[

/HW\u00b5V VHH WKH VDPH FRGH EXW LQ \$7 7 V\QWD[

/LVWLQJ \* & &

/&  
VWULQJ KL G G G?Q  
I  
SXVKO HES  
PRYO HVS HES  
SXVKO HE[  
VXEO HVS  
OHD O HVS HE[  
DQGO HE[  
PRYO HE[ HVS  
PRYO HVS  
PRYO HVS  
PRYO HVS  
PRYO /& HVS  
PRYO HVS  
FDOO BVQSULQWI  
PRYO HE[ HVS  
FDOO SXWV  
PRYO HES HE[  
OHDYH  
UHW

7KH FRGH LV WKH VDPH DV LQ WKH SUHYLRXV OLVWLQJ

%\ WKH Z PRYO HVISRUUHVSR PRY ':25' 375 >HVS @@ , QWHO V\QWD[ ,  
\$7 7 V\QWD[ WKH UHJLWVHU R@VHW IRUPDW RIDG RIIVHW UHJLVRWUH ORRNV

:LQGRZV 6(+

6(+ UHFRUGV DUH DOVR VWRUHG RQ WKH VWDFN LI WKH\ DUH S@Q\SHDW 5HD

%X^H U RYHU-RZ SURWHFWLRQ

ORUH DERXW LW KH@Q SDJH

6WUXFWXUHG ([FHSWLRQ +DQGOLQJ

---

\$XWRPDWLF GHDOORFDWLRQ RI GDWD LQ VWDFN

3HUKDSV WKH UHDVRQ IRU VWRULQJ ORFDO YDULDEOHV DQG 6(+ UHFRUGV LQ  
LFDOO\ XSRQ IXQFWLRQ H[LW XVLQJ MXVW RQH LQWUXFWLR \$''VR )FXQFLUWHLFRQ  
DUJXPHQWV DV ZH FRXOG VD\ DUH DOVR GHDOORFDWHG DXWRPDWLFDOO\ D  
WKLQJ VWRUHKGDQWXWHEH GHDOORFDWHG H[SOLFLWO\

\$ W\SLFDO VWDFN OD\RDXW

\$ W\SLFDO VWDFN OD\RDXW LQ D ELW HQYLURQPHQW DW WKH VWDUWRIDIXQF  
ORRNV OLNH WKLV

(63 [&]	ORFDO YDULDEOH	'B	D YDUBLQ
(63)	ORFDO YDULDEOH	'B	D YDUBLQ
(63)	VDYHG YD (%3 RI		
(63)	5HWXUQ \$GGUHVV		
(63)	DUJXPHQW PDUNSHDG DUJB		
(63)	DUJXPHQW PDUNSHDG DUJB		
(63 [&]	DUJXPHQW PDUNSHDG DUJB		

1RLVH LQ VWDFN

:KHQ RQH VD\V WKDW VRPHWKLQJ VH  
UDQGRP ZKDWRQH XVXDOO\ PHDQV  
LV WKDW RQH FDQQRW VHH DQ\ UHJX

6WHSKHQ :ROIUDP \$ 1HZ .LQG RI

2IWHQ LQ WKLV ERRN pQRLVHj RU pJDUEDJHj YDOXHV LQ WKH VWDFN RU PH  
FRPH IURP" 7KHVH DUH ZKDWRQH OHQ WKHUH DIWHU RWKHU IXQFWLRQ

LQFOXGH VWGLR K!
YRLG I
^
LQW D E F
,
YRLG I
^
LQW D E F
SULQWI G G G?Q D E F
,
LQW PDLQ
^
I
,

&RPSLOLQJ

/LVWLQJ 1RQ RSWLPL]LQJ 069&

6*	'%	G	G	G	D+	+
BF				VLIH		
BE				VLIH		
BD				VLIH		
BI	352&					
	SXVK	HES				
	PRY	HES	HVS			

67\$&.

VXE HVS  
PRY ':25' 375 BD >HES@  
PRY ':25' 375 BE >HES@  
PRY ':25' 375 BF >HES@  
PRY HVS HES  
SRS HES  
UHW  
BI (1'3  
  
BF VL]H  
BE VL]H  
BD VL]H  
BI 352&  
SXVK HES  
PRY HES HVS  
VXE HVS  
PRY HD[ ':25' 375 BF >HES@  
SXVK HD[  
PRY HF[ ':25' 375 BE >HES@  
SXVK HF[  
PRY HG[ ':25' 375 BD >HES@  
SXVK HG[  
SXVK 2))6(7 6\* G G G  
FDOO ':25' 375 BBLPSBBSULQWI  
DGG HVS  
PRY HVS HES  
SRS HES  
UHW  
BI (1'3  
  
BPDLQ 352&  
SXVK HES  
PRY HES HVS  
FDOO BI  
FDOO BI  
[RU HD[ HD[  
SRS HES  
UHW  
BPDLQ (1'3

7KH FRPSLOHU ZLOO JUXPEOH D OLWWOH ELW

F ?3RO\JRK?F!FO VW F )DVW DVP 0'  
OLFURVRIW 5 ELW & & 2SWLPL]LQJ &RPSLOHU 9HUVLRQ IRU [  
&RS\ULJKW & OLFURVRIW &RUSRUDWLRQ \$OO ULJKWV UHVHUYHG  
  
VW F  
F ?SRO\JRK?F?VW F ZDUQLQJ & XQLQLWLDO]HG ORFDO YDULDEOH F XVH  
F ?SRO\JRK?F?VW F ZDUQLQJ & XQLQLWLDO]HG ORFDO YDULDEOH E XVH  
F ?SRO\JRK?F?VW F ZDUQLQJ & XQLQLWLDO]HG ORFDO YDULDEOH D XVH  
OLFURVRIW 5 ,QFUHPHQWDO /LQNHU 9HUVLRQ  
&RS\ULJKW & OLFURVRIW &RUSRUDWLRQ \$OO ULJKWV UHVHUYHG  
  
RXW VW H[H  
VW REM

%XW ZKHQ ZH UXQ WKH FRPSLOHG SURJUDP

F ?3RO\JRK?F!VW

2K ZKDWDZHLUG WKLQJ :H GLG QRW I D7QKHMHDHIDDEQHKR\QWVY YDOXHV Z  
LQ WKH VWDFN

67\$&.

/HWUV ORDG WKH H[DPSOH LQWR 2OO\ 'EJ

)LJXUH 2OO\ 'E I

:KHC I D V V L J Q V W K H Y D B U L O G E O M H K V H L U Y D O X H V D U H V W R U [ )) WDKQHGDVGRG RI @ V

)LJXUH 2OO\'E I

a b DQGR I DUH ORFDWHG DW WKH VDPH DGGUHVHV 1R RQH KDV RYHUZUL  
 SRLQW WKH\ DUH VWLOO XQWRXFKHG 6R IRU WKLV ZHLUG VLWXDWLRQ WR  
 RQH DIWHU DQRW **6BKBW** WORGEH WKH VDPH DW HDFK IXQFWLRQ HQWU\ L H WKH  
 DUJXPHQWV 7KHQ WKH ORFDO YDULDEOHV ZLOO EH ORFDWHG DW WKH VDPH  
 YDOXHV LQ WKH VWDFN DQG PHPRU\ FHOOV LQ JHQHUDO KDYH YDOXHV OHIW  
 7KH\ DUH QRW UDQGRP LQ WKH VWULFW VHQVH EXW UDWKHU KDYH XQSUHGL  
 ,W ZRXOG SUREDEO\ EH SRVVLEOH WR FOHDU SRUWLQV RI WKH VWDFN EHIRU  
 PXFK H[WUD DQG XQQHFHVVDU\ ZRUN

069&amp;

7KH H[DPSOH ZDV FRPSLOHG E\ 069& %XW WKH UHDGHU RI WKLV ERRN P  
 H[DPSOH LQ 069& UDQ LW DQG JRW DOO QXPEHUV UHYHUVHG

F ?3RO\JRQ?F!VW

:K\" , DOVR FRPSLOHG WKLV H[DPSOH LQ 069& DQG VDZ WKLV  
 /LVWLQJ 069&

BD	VL]H
BE	VL]H
BF	VL]H
BI	352&

BI (1'3

BF	VL]H
BE	VL]H
BD	VL]H
BI	352&

BI (1'3

8QOLNH 069& 069& DOORFDWHG D E F Y I DEQHWHYQUXQFRWIGRHOU \$ Q  
 FRPSOHWHO\ FRUUHFW EHFDXVH & & VWDQGDUGVKDVQRUXOH LQZKLFKR  
 LQ WKH ORFDO VWDFN LI DW DOO 7KH UHDVRQ RI GL@HUhQFH LV EHFDXVH 06  
 KDV VXSSRVHGO\ VRPHWKLQJ FKDQJHG LQVLGH RI FRPSLOHU JXWV VR LW

([ HUFLVHV

KWWS FKDOOHQJHV UH  
 KWWS FKDOOHQJHV UH

SULQWI ZLWK VHYHUDO DUJXPHQWV

1RZ OHWUVH [WHHQG R\KZR UORQ SDJHH[DPSON UH SULQWLQWPDLQ IXQFWLRQ  
 ERG\ ZLWK WKLV

LQFOXGH VWGLR K!

LQW PDLQ

^  
 SULQWI D G E G F G  
 ` UHWXUQ

[

[ DUJXPHQWV

069&amp;

:KHQ ZH FRPSLOHLW ZLWK 069&amp; ([SUHVV ZH JHW

6\* ' % D G E G F G +

SXVK  
 SXVK  
 SXVK  
 SXVK 2))6(7 6\*  
 FDOO BSULQWI  
 DGG HVS

+

\$OPRVW WKH VDPH EXW QRZ ; SULQWVDUHJWPKHQWV DUH SXVKHG RQWR WKH V  
 RUGHU 7KH UVW DUJXPHQW LV SXVKHG ODVW

%\ WKH ZD\ YDULIDOMAIS\ RQ ELW HQYLURQPHQW KDYH ELW ZLGWK WKDW  
 6R ZH KDYH DUJXPHQW\ 16H6WKH\ RFFXS\ H[DFWO\ E\WHV LQ WKH VWDFN  
 D VWULQJ DQG QXPEH\ QWRI W\SH

:KHQ WKWDFN SR (63WHLVWHU KDV FKDQJHG EDFN E\ WKH

\$'' (63 ; LQVWUXFWLRQ DIWHUDIXQFWLRQ FDOO RIWHQ WKH QXPEHU RI IX  
 E\ VLPSO\ GLYLGQJ ; E\

2I FRXUVH WKLV LV VSHGLHF DQWRLQKIRQYHQWLRQ DQGRQO\ IRU ELW HQYL

6HH DOVR WKH FDOOLQJ FRQYHQWLSDUJHVHFWRQ

,Q FHUWDLQ FDVHV ZKHUH VHYHUDO IXQFWLRQV UHWXUQ ULJKW DIWHURQH D  
þ\$'' (63 ;ÿ LQVWUXFWLRQV LQWR RQH DIWHU WKH ODVW FDOOSXVK D  
SXVK D  
FDOOSXVK D  
FDOOSXVK D  
SXVK D  
SXVK D  
FDOO  
DGG HVS

+HUH LV D UHDO ZRUOG H[DPSOH

/LVWLQJ [

WH[W	(	SXVK				
WH[W	(	FDOO	VXE <sub>B</sub>	%	WDNHV	RQH DUJXPHQW
WH[W	((	FDOO	VXE <sub>B</sub>	'	WDNHV	QR DUJXPHQWV DW DOO
WH[W	)	FDOO	VXE <sub>B</sub>	\$	WDNHV	QR DUJXPHQWV DW DOO
WH[W	)	SXVK				
WH[W	)\$	FDOO	VXE <sub>B</sub>	%	WDNHV	RQH DUJXPHQW
WH[W	))	DGG HVS		GURSV	WZR DUJXPHQWV IURP VWDFN DW RQFH	

35,17) :,7+6(9(5\$/ \$5\*80(176  
069& DQG 2OO\'EJ

1RZ OHWUV WU\ WR ORDG WKLV H[DPSOH LQ 2OO\'EJ ,W LV RQH RI WKH PRVW  
:H FDQ FRPSLOH RXU H[DPSOH LQ 0'&RSWZRQ KZKLFK PHDQV 069&5 'A/ZLRWK  
ZH FDQ VHH WKH LPSRUWHG IXQFWLRQV FOHDUO\ LQ WKH GHEXJJHU

7KHQ ORDG WKH H[HFXWDEOH LQ 2OO\'EJ 7KH YQWGOO VGSUOHMDNSR IUQW L7  
VHFRQG EUHDNS &57FRGM LQRZ ZH KDYH W PDLQ WKQHF WLRQ

)LQG WKLV FRGH E\ VFUROOLQJ WKH FRGH WR WKH PDLQ WRSF WLSR&Q DDOWO RWKDH  
EHJLQLQJ RI WKH FRGH VHFWLRQ

)LJXUH 2OO\'EJ WKH YHU\ V PDLQ RXQWFKWL RQ

&OLFN R 386+ (%BQVWUXFWLRQ SUHVV ) VHW EUHDNSRLQW DQG SUHVV )  
WKHVH DFWLRQV LQ &57FRUG W REVHNLDSX VH ZH DUHQW UHDOO\ LQWHUHVWHG LQ

)LJXUH 2OO\`EJ E+ SULQWIH[HFXWLRQ

1RZ W~~KH~~SRLQWV \&\$// SULQMQVWUXFWLRQ 2OO\`EJ OLNH RWKHU GHEXJJHU  
RI WKH UHJLVWHUV ZKLFK ZHUF FKDQJHG 6R (,3 FFK~~WQ~~PHV\~~DQ~~SULHWWYDOXH LV  
LQ UI (63 FKDQJHV DV ZHOO EHFDXVH WKH DUJXPHQWV YDOXHV DUH SXVKHG  
:KHUH DUH WKH YDOXHV LQ WKH VWDFN" 7DNH D ORRN DW WKH ULJKW ERWWRI

)LJXUH 2OO\`EJ VWDFN DIWHU WKH DUJXPHQW YDOXHV KDYH EHHQ SXVKHG  
DGGHG E\ WKH DXWKRU LQ DJUDSKLFV HGLWRU

:H FDQ VHH FROXPQV WKUH DGGUHVV LQ WKH VWDFN YDOXH LQ WKH VWD  
PHQWV 2OO\`EJ XQ(SULQW)~~I~~QGMH VWULQJV VR LW UHSRUWV WKH V~~WUWQ~~FKHG  
WR LW

,W LV SRVVLEOH WR ULJKW FOLFN RQ WKH IRUPDW VWULQJ FOLFN RQ b)ROOR  
LQ WKH GHEXJJHU OHIW ERWWRP ZLQGRZ ZKLFK DOZD\V GLVSOD\V VRPH SD  
YDOXHV FDQ EH HGLWHG ,W LV SRVVLEOH WR FKDQJH WKH IRUPDW VWULQJ  
ZRXOG EH GL®HUHQW ,W LV QRW YHU\ XVHIXO LQ WKLV SDUWLFXODU FDVH  
VWDUW EXLOGLQJD IHHO RI KRZ HYHU\WKLQJ ZRUNV KHUH

35,17) :,7+6(9(5\$/ \$5\*80(176

3UHVW ) VWHS RYHU

:H VHH WKH IROORZLQJ RXWSXW LQ WKH FRQVROH

D E F

/HWUV VHH KRZ WKH UHJLVWHUV DQG VWDFN VWDWH KDYH FKDQJHG

)LJXUH 200\ 'EJ D SULQWIH[HFXWLRQ

5HJLV (\$;UQRZ FRQV [ ' QV 7KDW LV FRUU SULQWJQHUXUQV WKH QXPEHU RI F  
SULQWHG 7KH (,3OKXHVRFKDQJHG LQGHHG QRZ LW FRQWDLQV WKH DGGUHV  
DIW &\$/// SULQ' (&; DQ ('; YDOXHV KDYH FKDQJHG DV ZHOC SULQWUHQWOLRQK  
KLGGHQ PDFKLQHUV XVHG WKHP IRU LWV RZQ QHHGV  
\$ YHU\ LPSRUWDQW IDFWRU (63DVWQXHLWQKRHWVWWKHHVWDFN VWDWH KDYH EHHQ I  
VHH WKDW WKH IRUPDW VWULQJ DQG FRUUHVSRQGLQJ YDOXHFGBIFQVWDFN  
FRQYHQWLWQ EFDKODQHUV QRW (63VWDQW WR LWV SUHYLRXVOVWDFN VSHRIQVLEO  
GR VR

35,17) :,7+6(9(5\$/ \$5\*80(176

3UHVV ) DJDLQ WI\$'' (63 ILQVWUXFWLRQ

)LJXUH 2OO\`EJ E\$'' (63 LQVWUXFWLRQ H[HFXWLRQ

(63 KDV FKDQJHG EXW WKH YDOXHV DUH VWLOO LQ WKH VWDFN <HV RI FRXU  
WR JHURV RU VRPHWKLQJ OLNH WKDW (YHU\WK 6Q JLQH WR VHWKEDDMQDFNDS/RQ RQ  
PHDQLQJ DW DOO ,W ZRXOG EH WLPH FRQVXPLQJ WR FOHDU WKH XQXVHG VW  
QHHGV WR

\* & &

1RZ OHWUV FRPSLOH WKH VDPH SURJUDPLQ /LQX[ XVLQJ \* & & DQG W\$ NH D

PDLQ SURF QHDU

YDUB GZRUG SWU K  
YDUB& GZRUG SWU &K  
YDUB GZRUG SWU  
YDUB GZRUG SWU

SXVK HES  
PRY HES HVS  
DQG HVS )))))) K  
VXE HVS K  
PRY HD[ RIIVHW D\$'%&' D G E G F G  
PRY >HVS K YDUB @  
PRY >HVS K YDUB @  
PRY >HVS K YDUB &@  
PRY >HVS K YDUB @ HD[  
FDOO BSULQWI  
PRY HD[  
OHDIH  
UHWQ  
PDLQ HQGS

,WV QRWLFHDEOH WKDW WKH GLRHUHQFH EHWZHHQ WKH 069& FRGH DQG WK  
JXPHQWV DUH VWRUHG RQ WKH VWDFN +HUH WKH \*&& LV ZRUNLQJ GLUHFW

386 323

\* & & DQG \*' %

\$ 5 \* 80 (176

H D Q O N Q X [Q

W K H F R P S L O H U W R L Q F O X G H G H E X J L Q I R U P D W L R Q L C

R P H G H Q Q L V S R O \ J R Q G R Q H

V L Q J O H W \ V V H W S U L Q W B R L Q W R Q

H X Q F W L R Q V R X U F H F R G % H F D Q U H W V R R Z L W E X W P D \ G R

P H G H Q Q L V S R O \ J R Q

I I R U P D W [ I D G E G F G D W S U L Q W I F  
I L O H R U G L U H F W R U \

Q W V 7 K H P R V W O H I W F R O X P Q F R Q W D L Q V D G G U H V V H V

[ I [ [

[ D H : H F D Q Y H U L I \ W K L V E \ G L V D V V H P E O L Q J W K

[ H D [

Q V D U H L G O H L Q V W W 1 U X 3 F W L R Q V D Q D O R J R X V W R

L V W K H I R U P D W V W U L Q J D G G U H V V

G

S U L Q W W D X U H J X P H Q W V 7 K H U H V W R I W K H H O H P H Q W V F R X  
K O G D O V R E H Y D O X H V I U R P R W K H U I X Q F W L R Q V W K H L U

D Q G L Q V W U X F W V \* ' % W R p H [ H F X W H D O O L Q V W U X F W L I  
S U L Q W K H H Q G R I

B S U L Q W I I R U P D W [ I D G E G F G D W S U L Q W I













































































































































































































































































































































































































































































































































































































































































































































































































































































































































































































































































































































































































































































































































































Figure 5.2: FAR: UTF-8

As you can see, the English language string looks the same as it is in ASCII.

The Hungarian language uses some Latin symbols plus symbols with diacritic marks.

These symbols are encoded using several bytes, these are underscored with red. It's the same story with the Icelandic and Polish languages.

There is also the “Euro” currency symbol at the start, which is encoded with 3 bytes.

The rest of the writing systems here have no connection with Latin.

At least in Russian, Arabic, Hebrew and Hindi we can see some recurring bytes, and that is not surprise: all symbols from a writing system are usually located in the same Unicode table, so their code begins with the same numbers.

At the beginning, before the “How much?” string we see 3 bytes, which are in fact the [BOM<sup>7</sup>](#). The [BOM](#) defines the encoding system to be used.

## UTF-16LE

Many win32 functions in Windows have the suffixes [-A](#) and [-W](#). The first type of functions works with normal strings, the other with UTF-16LE strings (*wide*).

In the second case, each symbol is usually stored in a 16-bit value of type *short*.

The Latin symbols in UTF-16 strings look in Hiew or FAR like they are interleaved with zero byte:

```
int wmain()
{
    wprintf (L"Hello, world!\n");
}
```

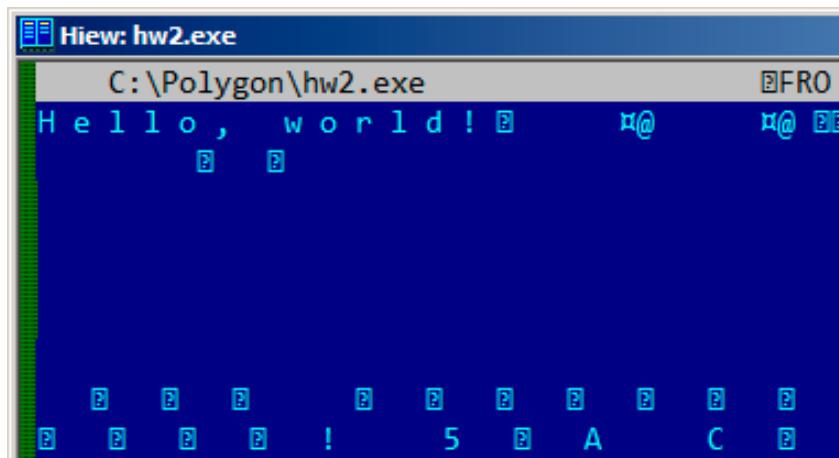


Figure 5.3: Hiew

<sup>7</sup>Byte order mark

## 5.4. *STRINGS*

We can see this often in Windows NT system files:

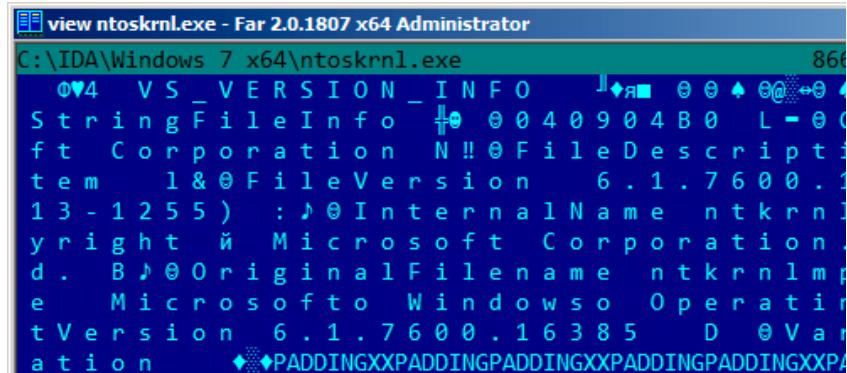


Figure 5.4: Hiew

Strings with characters that occupy exactly 2 bytes are called “Unicode” in IDA:

```
.data:0040E000 aHelloWorld:  
.data:0040E000                 unicode 0, <Hello, world!>  
.data:0040E000                 dw 0Ah, 0
```

Here is how the Russian language string is encoded in UTF-16LE:



Figure 5.5: Hiew: UTF-16LE

What we can easily spot is that the symbols are interleaved by the diamond character (which has the ASCII code of 4). Indeed, the Cyrillic symbols are located in the fourth Unicode plane<sup>8</sup>. Hence, all Cyrillic symbols in UTF-16LE are located in the `0x400-0x4FF` range.

Let's go back to the example with the string written in multiple languages. Here is how it looks like in UTF-16LE.

<sup>8</sup>wikipedia

## 5.4. *STRINGS*

Figure 5.6: FAR: UTF-16LE

Here we can also see the **BOM** at the beginning. All Latin characters are interleaved with a zero byte. Some characters with diacritic marks (Hungarian and Icelandic languages) are also underscored in red.

## Base64

The base64 encoding is highly popular for the cases when you have to transfer binary data as a text string. In essence, this algorithm encodes 3 binary bytes into 4 printable characters: all 26 Latin letters (both lower and upper case), digits, plus sign ("+") and slash sign ("/"), 64 characters in total.

One distinctive feature of base64 strings is that they often (but not always) end with 1 or 2 padding equality symbol(s) ("="), for example:

AVjbbVSVfcUMu1xvjaMgjNtueRwBbxnyJw8dpGnLW8ZW8aKG3v4Y0icuQT+qEJAp9lAOuWs=

wVjbbVSfcUMu1xvjaMgjNtueRwBbxnyJw8dpGnLW8ZW8aKG3v4Y0icuQT+qEJAp9lA0uQ==

The equality sign ("=") is never encountered in the middle of base64-encoded strings.

Now example of manual encoding. Let's encode 0x00, 0x11, 0x22, 0x33 hexadecimal bytes into base64 string:

```
$ echo -n "\x00\x11\x22\x33" | base64  
ABEiMw==
```

Let's put all 4 bytes in binary form, then regroup them into 6-bit groups:

```
| 00  || 11  || 22  || 33  ||      ||  
000000000000100010010001000110011?????????????????  
| A  || B  || E  || i  || M  || w  || =  || =  ||
```

Three first bytes (0x00, 0x11, 0x22) can be encoded into 4 base64 characters ("ABEi"), but the last one (0x33) — cannot be, so it's encoded using two characters ("Mw") and padding symbol ("=") is added twice to pad the last group to 4 characters. Hence, length of all correct base64 strings are always divisible by 4.

## 5.4. STRINGS

Base64 is often used when binary data needs to be stored in XML. "Armored" (i.e., in text form) PGP keys and signatures are encoded using base64.

Some people tries to use base64 to obfuscate strings: <http://blog.sec-consult.com/2016/01/deliberately.html><sup>9</sup>.

There are utilities for scanning an arbitrary binary files for base64 strings. One such utility is base64scanner<sup>10</sup>.

Another encoding system which was much more popular in UseNet and FidoNet is Uuencoding. It offers mostly the same features, but is different from base64 in the sense that file name is also stored in header.

By the way: there is also close sibling to base64: base32, alphabet of which has 10 digits and 26 Latin characters. One well-known usage of it is onion addresses<sup>11</sup>, like: <http://3g2upl4pq6kufc4m.onion/>. URL can't have mixed-case Latin characters, so apparently, this is why Tor developers used base32.

### 5.4.2 Finding strings in binary

Actually, the best form of Unix documentation is frequently running the **strings** command over a program's object code. Using **strings**, you can get a complete list of the program's hard-coded file name, environment variables, undocumented options, obscure error messages, and so forth.

The Unix-Haters Handbook

The standard UNIX *strings* utility is quick-n-dirty way to see strings in file. For example, these are some strings from OpenSSH 7.2 sshd executable file:

```
...
0123
0123456789
0123456789abcdefABCDEF.:/%
%02x
...
%.100s, line %lu: Bad permitopen specification <%.100s>
%.100s, line %lu: invalid criteria
%.100s, line %lu: invalid tun device
...
%.200s/.ssh/environment
...
2886173b9c9b6fdbdeda7a247cd636db38deaa.debug
$2a$06$r3.juUaHZDLIbQa02dS9FuYxL1W9M81R1Tc92PoSNmzvpEqLkLGrK
...
3des-cbc
...
Bind to port %s on %s.
Bind to port %s on %s failed: %.200s.
/bin/login
/bin/sh
/bin/sh /etc/ssh/sshd
...
D$4PQWR1
D$4PUj
D$4PV
D$4PVj
D$4PW
D$4PWj
D$4X
D$4XZj
D$4Y
...
diffie-hellman-group-exchange-sha1
diffie-hellman-group-exchange-sha256
```

<sup>9</sup> <http://archive.is/nDCas>

<sup>10</sup> <https://github.com/dennis714/base64scanner>

<sup>11</sup> <https://trac.torproject.org/projects/tor/wiki/doc/HiddenServiceNames>

## 5.4. STRINGS

```
digests
D$IPV
direct-streamlocal
direct-streamlocal@openssh.com
...
FFFFFFFFFFFFFC90FDAA22168C234C4C6628B80DC1CD129024E088A6...
...
```

There are options, error messages, file paths, imported dynamic modules and functions, some other strange strings (keys?) There is also unreadable noise—x86 code sometimes has chunks consisting of printable ASCII characters, up to 8 characters.

Of course, OpenSSH is open-source program. But looking at readable strings inside of some unknown binary is often a first step of analysis.

`grep` can be applied as well.

Hiew has the same capability (Alt-F6), as well as Sysinternals ProcessMonitor.

### 5.4.3 Error/debug messages

Debugging messages are very helpful if present. In some sense, the debugging messages are reporting what's going on in the program right now. Often these are `printf()`-like functions, which write to log-files, or sometimes do not writing anything but the calls are still present since the build is not a debug one but *release* one.

If local or global variables are dumped in debug messages, it might be helpful as well since it is possible to get at least the variable names. For example, one of such function in Oracle RDBMS is `ksdwrt()`.

Meaningful text strings are often helpful. The [IDA](#) disassembler may show from which function and from which point this specific string is used. Funny cases sometimes happen<sup>12</sup>.

The error messages may help us as well. In Oracle RDBMS, errors are reported using a group of functions. You can read more about them here: [blog.yurichev.com](http://blog.yurichev.com).

It is possible to find quickly which functions report errors and in which conditions.

By the way, this is often the reason for copy-protection systems to inarticulate cryptic error messages or just error numbers. No one is happy when the software cracker quickly understand why the copy-protection is triggered just by the error message.

One example of encrypted error messages is here: [8.5.2 on page 820](#).

### 5.4.4 Suspicious magic strings

Some magic strings which are usually used in backdoors looks pretty suspicious.

For example, there was a backdoor in the TP-Link WR740 home router<sup>13</sup>. The backdoor can activated using the following URL:

[http://192.168.0.1/userRpmNatDebugRpm26525557/start\\_art.html](http://192.168.0.1/userRpmNatDebugRpm26525557/start_art.html).

Indeed, the “userRpmNatDebugRpm26525557” string is present in the firmware.

This string was not googleable until the wide disclosure of information about the backdoor.

You would not find this in any [RFC](#)<sup>14</sup>.

You would not find any computer science algorithm which uses such strange byte sequences.

And it doesn't look like an error or debugging message.

So it's a good idea to inspect the usage of such weird strings.

Sometimes, such strings are encoded using base64.

<sup>12</sup>[blog.yurichev.com](http://blog.yurichev.com)

<sup>13</sup><http://sekurak.pl/tp-link-httptftp-backdoor/>

<sup>14</sup>Request for Comments

## 5.5. CALLS TO ASSERT()

So it's a good idea to decode them all and to scan them visually, even a glance should be enough.

More precise, this method of hiding backdoors is called “security through obscurity”.

## 5.5 Calls to assert()

Sometimes the presence of the `assert()` macro is useful too: commonly this macro leaves source file name, line number and condition in the code.

The most useful information is contained in the assert's condition, we can deduce variable names or structure field names from it. Another useful piece of information are the file names—we can try to deduce what type of code is there. Also it is possible to recognize well-known open-source libraries by the file names.

Listing 5.2: 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_nbBitsBit ; "sp->lzw_nbBits <= BITS_MAX"
.text:107D6771 call ds:_assert
```

It is advisable to “google” both the conditions and file names, which can lead us to an open-source library. For example, if we “google” “`sp->lzw_nbits <= BITS_MAX`”, this predictably gives us some open-source code that’s related to the LZW compression.

## 5.6 Constants

Humans, including programmers, often use round numbers like 10, 100, 1000, in real life as well as in the code.

The practicing reverse engineer usually know them well in hexadecimal representation: 0b10=0xA, 0b100=0x64, 0b1000=0x3E8, 0b10000=0x2710.

The constants `0xFFFFFFFF` (`0b101`) and `0xFFFFFFFF` (`0b01`) are also popular—those are composed of alternating bits.

That may help to distinguish some signal from a signal where all bits are turned on (0b1111 ...) or off (0b0000 ...). For example, the `0x55AA` constant is used at least in the boot sector, [MBR<sup>15</sup>](#), and in the [ROM](#) of IBM-compatible extension cards.

## **15 Master Boot Record**

## 5.6. CONSTANTS

Some algorithms, especially cryptographical ones use distinct constants, which are easy to find in code using [IDA](#).

For example, the MD5<sup>16</sup> algorithm initializes its own internal variables like this:

```
var int h0 := 0x67452301
var int h1 := 0xEFCDAB89
var int h2 := 0x98BADCCE
var int h3 := 0x10325476
```

If you find these four constants used in the code in a row, it is highly probable that this function is related to MD5.

Another example are the CRC16/CRC32 algorithms, whose calculation algorithms often use precomputed tables like this one:

Listing 5.3: 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,
    0xCC01, 0x0CC0, 0x0D80, 0xCD41, 0x0F00, 0xCFC1, 0xCE81, 0x0E40,
    ...
}
```

See also the precomputed table for CRC32: [3.5 on page 481](#).

In tableless CRC algorithms well-known polynomials are used, for example, 0xEDB88320 for CRC32.

### 5.6.1 Magic numbers

A lot of file formats define a standard file header where a *magic number(s)*<sup>17</sup> is used, single one or even several.

For example, all Win32 and MS-DOS executables start with the two characters “MZ”<sup>18</sup>.

At the beginning of a MIDI file the “MThd” signature must be present. If we have a program which uses MIDI files for something, it’s very likely that it must check the file for validity by checking at least the first 4 bytes.

This could be done like this: (*buf* points to the beginning of the loaded file in memory)

```
cmp [buf], 0x6468544D ; "MThd"
jnz _error_not_a_MIDI_file
```

...or by calling a function for comparing memory blocks like `memcmp()` or any other equivalent code up to a `CMPSB` ( [1.6 on page 1013](#)) instruction.

When you find such point you already can say where the loading of the MIDI file starts, also, we could see the location of the buffer with the contents of the MIDI file, what is used from the buffer, and how.

### Dates

Often, one may encounter number like `0x19870116`, which is clearly looks like a date (year 1987, 1th month (January), 16th day). This may be someone’s birthday (a programmer, his/her relative, child), or some other important date. The date may also be written in a reverse order, like `0x16011987`. American-style dates are also popular, like `0x01161987`.

Well-known example is `0x19540119` (magic number used in UFS2 superblock structure), which is a birth-day of Marshall Kirk McKusick, prominent FreeBSD contributor.

Stuxnet uses the number “19790509” (not as 32-bit number, but as string, though), and this led to speculation that the malware is connected to Israel<sup>19</sup>

<sup>16</sup>[wikipedia](#)

<sup>17</sup>[wikipedia](#)

<sup>18</sup>[wikipedia](#)

<sup>19</sup>This is a date of execution of Habib Elghanian, persian jew.

## 5.6. CONSTANTS

Also, numbers like those are very popular in amateur-grade cryptography, for example, excerpt from the *secret function* internals from HASP3 dongle <sup>20</sup>:

```
void xor_pwd(void)
{
    int i;

    pwd^=0x09071966;
    for(i=0;i<8;i++)
    {
        al_buf[i]= pwd & 7; pwd = pwd >> 3;
    }
};

void emulate_func2(unsigned short seed)
{
    int i, j;
    for(i=0;i<8;i++)
    {
        ch[i] = 0;

        for(j=0;j<8;j++)
        {
            seed *= 0x1989;
            seed += 5;
            ch[i] |= (tab[(seed>>9)&0x3f]) << (7-j);
        }
    }
}
```

## DHCP

This applies to network protocols as well. For example, the DHCP protocol's network packets contains the so-called *magic cookie*: `0x63538263`. Any code that generates DHCP packets somewhere must embed this constant into the packet. If we find it in the code we may find where this happens and, not only that. Any program which can receive DHCP packet must verify the *magic cookie*, comparing it with the constant.

For example, let's take the `dhcpcore.dll` file from Windows 7 x64 and search for the constant. And we can find it, twice: it seems that the constant is used in two functions with descriptive names

`DhcpExtractOptionsForValidation()` and `DhcpExtractFullOptions()`:

Listing 5.4: `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
```

And here are the places where these constants are accessed:

Listing 5.5: `dhcpcore.dll` (Windows 7 x64)

```
.text:000007FF6480875F mov     eax, [rsi]
.text:000007FF64808761 cmp     eax, cs:dword_7FF6483CBE8
.text:000007FF64808767 jnz     loc_7FF64817179
```

And:

Listing 5.6: `dhcpcore.dll` (Windows 7 x64)

```
.text:000007FF648082C7 mov     eax, [r12]
.text:000007FF648082CB cmp     eax, cs:dword_7FF6483CBEC
.text:000007FF648082D1 jnz     loc_7FF648173AF
```

<sup>20</sup><https://web.archive.org/web/20160311231616/http://www.woodmann.com/fravia/bayu3.htm>

## 5.6.2 Specific constants

Sometimes, there is a specific constant for some type of code. For example, the author once dug into a code, where number 12 was encountered suspiciously often. Size of many arrays is 12, or multiple of 12 (24, etc). As it turned out, that code takes 12-channel audio file at input and process it.

And vice versa: for example, if a program works with text field which has length of 120 bytes, there has to be a constant 120 or 119 somewhere in the code. If UTF-16 is used, then  $2 \cdot 120$ . If a code works with network packets of fixed size, it's good idea to search for this constant in the code as well.

This is also true for amateur cryptography (license keys, etc). If encrypted block has size of  $n$  bytes, you may want to try to find occurrences of this number throughout the code. Also, if you see a piece of code which is been repeated  $n$  times in loop during execution, this may be encryption/decryption routine.

## 5.6.3 Searching for constants

It is easy in [IDA](#): Alt-B or Alt-I. And for searching for a constant in a big pile of files, or for searching in non-executable files, there is a small utility called *binary grep*<sup>21</sup>.

## 5.7 Finding the right instructions

If the program is utilizing FPU instructions and there are very few of them in the code, one can try to check each one manually with a debugger.

For example, we may be interested how Microsoft Excel calculates the formulae entered by user. For example, the division operation.

If we load excel.exe (from Office 2010) version 14.0.4756.1000 into [IDA](#), make a full listing and to find every `FDIV` instruction (except the ones which use constants as a second operand—obviously, they do not suit us):

```
cat EXCEL.lst | grep fdiv | grep -v dbl_ > EXCEL.fdiv
```

...then we see that there are 144 of them.

We can enter a string like `= (1/3)` in Excel and check each instruction.

By checking each instruction in a debugger or [tracer](#) (one may check 4 instruction at a time), we get lucky and the sought-for instruction is just the 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)` holds the first argument (1) and second one is in `[EBX]`.

The instruction after `FDIV` (`FSTP`) writes the result in memory:

```
.text:3011E91B DD 1E          fstp    qword ptr [esi]
```

If we set a breakpoint on it, we can see the result:

<sup>21</sup>[GitHub](#)

## 5.8. SUSPICIOUS CODE PATTERNS

```
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 the 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 shows 666 in the cell, finally convincing us that we have found the right point.

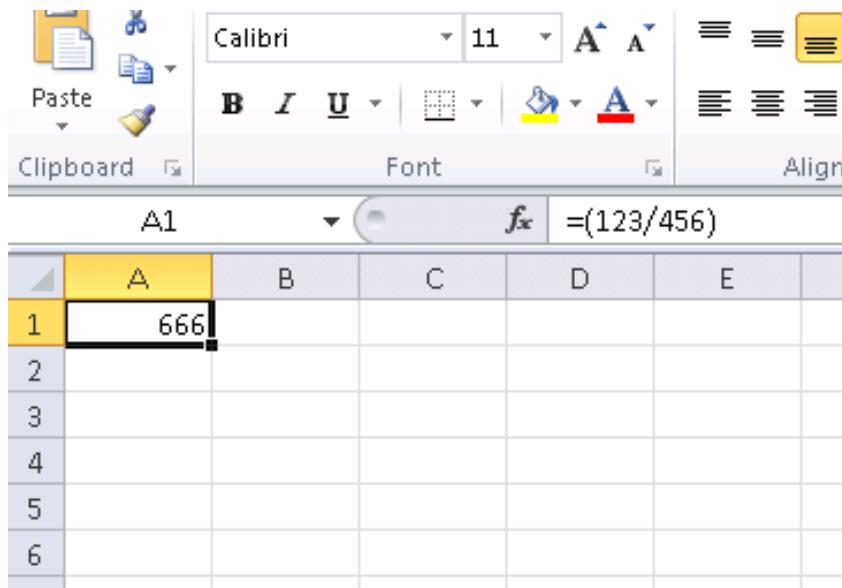


Figure 5.7: The practical joke worked

If we try the same Excel version, but in x64, we will find only 12 FDIV instructions there, and the one we looking for is the third one.

```
tracer.exe -l:excel.exe bpx=excel.exe!BASE+0x1B7FCC, set(st0,666)
```

It seems that a lot of division operations of *float* and *double* types, were replaced by the compiler with SSE instructions like DIVSD (DIVSD is present 268 times in total).

## 5.8 Suspicious code patterns

### 5.8.1 XOR instructions

Instructions like X0R op, op (for example, X0R EAX, EAX) are usually used for setting the register value to zero, but if the operands are different, the “exclusive or” operation is executed.

## 5.8. SUSPICIOUS CODE PATTERNS

This operation is rare in common programming, but widespread in cryptography, including amateur one. It's especially suspicious if the second operand is a big number.

This may point to encrypting/decrypting, checksum computing, etc.

One exception to this observation worth noting is the “canary” ([1.20.3 on page 283](#)). Its generation and checking are often done using the `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 noting that this kind of script can also match incorrectly disassembled code ([3.17 on page 537](#)).

### 5.8.2 Hand-written assembly code

Modern compilers do not emit the `LOOP` and `RCL` instructions. On the other hand, these instructions are well-known to coders who like to code directly in assembly language. If you spot these, it can be said that there is a high probability that this fragment of code was hand-written. Such instructions are marked as (M) in the instructions list in this appendix: [1.6 on page 1007](#).

Also the function prologue/epilogue are not commonly present in hand-written assembly.

Commonly there is no fixed system for passing arguments to functions in the hand-written code.

Example from the 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 in the `WRK22` v1.2 source code, this code can be found easily in file `WRK-v1.2\base\ntos\ke\i386\cpu.asm`.

<sup>22</sup>Windows Research Kernel

## 5.9 Using magic numbers while tracing

Often, our main goal is to understand how the program uses a value that has been either read from file or received via network. The manual tracing of a value is often a very labor-intensive task. One of the simplest techniques for this (although not 100% reliable) is to use your own *magic number*.

This resembles X-ray computed tomography in some sense: a radiocontrast agent is injected into the patient's blood, which is then used to improve the visibility of the patient's internal structure in to the X-rays. It is well known how the blood of healthy humans percolates in the kidneys and if the agent is in the blood, it can be easily seen on tomography, how blood is percolating, and are there any stones or tumors.

We can take a 32-bit number like `0x0badf00d`, or someone's birth date like `0x11101979` and write this 4-byte number to some point in a file used by the program we investigate.

Then, while tracing this program with [tracer](#) in code coverage mode, with the help of [grep](#) or just by searching in the text file (of tracing results), we can easily see where the value has been used and how.

Example of *grepable* [tracer](#) results in cc mode:

<code>0x150bf66 (_kziaia+0x14), e=</code>	<code>1 [MOV EBX, [EBP+8]] [EBP+8]=0xf59c934</code>
<code>0x150bf69 (_kziaia+0x17), e=</code>	<code>1 [MOV EDX, [69AEB08h]] [69AEB08h]=0</code>
<code>0x150bf6f (_kziaia+0x1d), e=</code>	<code>1 [FS: MOV EAX, [2Ch]]</code>
<code>0x150bf75 (_kziaia+0x23), e=</code>	<code>1 [MOV ECX, [EAX+EDX*4]] [EAX+EDX*4]=0xf1ac360</code>
<code>0x150bf78 (_kziaia+0x26), e=</code>	<code>1 [MOV [EBP-4], ECX] ECX=0xf1ac360</code>

This can be used for network packets as well. It is important for the *magic number* to be unique and not to be present in the program's code.

Aside of the [tracer](#), DosBox (MS-DOS emulator) in heavydebug mode is able to write information about all registers' states for each executed instruction of the program to a plain text file<sup>23</sup>, so this technique may be useful for DOS programs as well.

## 5.10 Other things

### 5.10.1 General idea

A reverse engineer should try to be in programmer's shoes as often as possible. To take his/her viewpoint and ask himself, how would one solve some task the specific case.

### 5.10.2 Order of functions in binary code

All functions located in a single .c or .cpp-file are compiled into corresponding object (.o) file. Later, linker puts all object files it needs together, not changing order or functions in them. As a consequence, if you see two or more consecutive functions, it means, that they were placed together in a single source code file (unless you're on border of two object files, of course.) This means these functions have something in common, that they are from the same [API](#) level, from same library, etc.

### 5.10.3 Tiny functions

Tiny functions like empty functions ([1.3 on page 5](#)) or function which returns just "true" (1) or "false" (0) ([1.4 on page 7](#)) are very common, and almost all decent compiler tends put only one such function into resulting executable code even if there was several similar functions in source code. So, whenever you see a tiny function consisting just of `mov eax, 1 / ret` which is referenced (and can be called) from many places, which are seems unconnected to each other, this may be a result of such optimization.

### 5.10.4 C++

[RTTI](#) ([3.19.1 on page 560](#))-data may be also useful for C++ class identification.

<sup>23</sup>See also my blog post about this DosBox feature: [blog.yurichev.com](http://blog.yurichev.com)

## 5.10.5 Some binary file patterns

All examples here were prepared on the Windows with active code page 437 <sup>24</sup> in console. Binary files internally may look visually different if another code page is set.

---

<sup>24</sup>[https://en.wikipedia.org/wiki/Code\\_page\\_437](https://en.wikipedia.org/wiki/Code_page_437)

# Arrays

Sometimes, we can clearly spot an array of 16/32/64-bit values visually, in hex editor.

Here is an example of array of 16-bit values. We see that the first byte in pair is 7 or 8, and the second looks random:

E:\...3affacde09fe21c28f1543db51145b.dat	h	1252	2175000	Col 0	23%	21:25
000007CA70:	EF 07 C6 07 D6 07 26 08	0C 08 CE 07 24 07 60 07	ï•Æ•Ö•&•♀•Î•\$•`•			
000007CA80:	CC 07 AA 07 A2 07 AC 07	E9 07 BF 07 D6 07 2C 08	Ì•¤•¢•--•é•ξ•Ö•,•			
000007CA90:	09 08 CA 07 31 07 5E 07	BC 07 9A 07 93 07 9E 07	ö•È•1•^•%•š•“•ž•			
000007CAA0:	E6 07 BD 07 D8 07 2F 08	06 08 CB 07 3E 07 5E 07	æ•%•ø•/•†•È•>•^•			
000007CAB0:	B3 07 91 07 8B 07 97 07	E1 07 BB 07 DB 07 32 08	³•‘•<•--•á•»•Û•2•			
000007CAC0:	03 08 CB 07 4C 07 61 07	AA 07 89 07 84 07 91 07	▼•È•L•a•¤•‰•„•‘•			
000007CAD0:	E0 07 BB 07 DC 07 33 08	01 08 CC 07 57 07 64 07	à•»•Ü•3•@•Ì•W•d•			
000007CAE0:	A4 07 84 07 81 07 90 07	DE 07 BB 07 DE 07 34 08	¤•„••¤•¤•þ•»•þ•4•			
000007CAF0:	FF 07 CD 07 65 07 69 07	A0 07 81 07 7F 07 90 07	ÿ•Í•e•i• •¤•¤•¤•			
000007CB00:	DE 07 BC 07 DF 07 33 08	FF 07 CE 07 70 07 6F 07	þ•%•ß•3•ÿ•Î•p•o•			
000007CB10:	9F 07 82 07 81 07 93 07	DD 07 BC 07 E0 07 34 08	Ý•,•¤•“•Ý•%•à•4•			
000007CB20:	FE 07 CE 07 7E 07 78 07	9F 07 84 07 84 07 96 07	þ•î•~•x•Ý•„•„•-			
000007CB30:	DE 07 BD 07 DF 07 32 08	FF 07 CE 07 87 07 7F 07	þ•%•ß•2•ÿ•Î•‡•¤•			
000007CB40:	A1 07 87 07 88 07 9B 07	E2 07 BF 07 DE 07 2F 08	í•‡•^••¤•é•þ•/•			
000007CB50:	02 08 CF 07 93 07 89 07	A4 07 8C 07 8D 07 9F 07	ø•Í•“•¤•¤•È•¤•Ý•			
000007CB60:	E4 07 C0 07 DD 07 2D 08	03 08 CF 07 9C 07 92 07	ä•À•Ý•-•▼•Í•¤•’•			
000007CB70:	A9 07 90 07 91 07 A3 07	E6 07 C3 07 DD 07 2B 08	Ø•¤•‘•£•æ•Ã•Ý•+•			
000007CB80:	04 08 D0 07 A7 07 9C 07	AE 07 96 07 96 07 A7 07	♦•Ð•§•¤•®•-•-•-•§•			
000007CB90:	E8 07 C7 07 DF 07 29 08	04 08 D3 07 B1 07 A7 07	ë•Ç•ß•)•†•Ó•±•§•			
000007CBA0:	B4 07 9B 07 9B 07 AB 07	E8 07 CA 07 E1 07 27 08	‘•»•»•«•è•É•á•’•			
000007CBB0:	03 08 D5 07 BB 07 B3 07	BB 07 A1 07 A0 07 AF 07	▼•Ø•»•³•»•í•-•-			
000007CBC0:	EA 07 CD 07 E3 07 25 08	03 08 D8 07 C4 07 BD 07	ë•Í•ã•%•ø•ð•Ä•%			
000007CBD0:	C1 07 A6 07 A5 07 B3 07	EA 07 D1 07 E6 07 22 08	Á•!•¥•³•ë•Ñ•æ•”•			
000007CBE0:	01 08 DC 07 CE 07 C8 07	C8 07 AD 07 AA 07 B7 07	ø•Ü•Î•È•È•-•¤•-			

Figure 5.8: FAR: array of 16-bit values

I used a file containing 12-channel signal digitized using 16-bit ADC<sup>25</sup>.

## <sup>25</sup>Analog-to-digital converter

## 5.10. OTHER THINGS

And here is an example of very typical MIPS code.

As we may recall, every MIPS (and also ARM in ARM mode or ARM64) instruction has size of 32 bits (or 4 bytes), so such code is array of 32-bit values.

By looking at this screenshot, we may see some kind of pattern.

Vertical red lines are added for clarity:

```

Hiew: FW96650A.bin
FW96650A.bin          FRO ----- 00005000
00005000: A0 B0 02 3C-04 00 BE AF-40 00 43 8C-21 F0 A0 03 aвв<п@ CM!Ёав
00005010: FF 1F 02 3C-21 E8 C0 03-FF FF 42 34-24 10 62 00 вв<!шв B4$вв
00005020: 00 A0 03 3C-25 10 43 00-04 00 BE 8F-08 00 E0 03 ав<%вв вв<Пв вв
00005030: 08 00 BD 27-F8 FF BD 27-A0 B0 02 3C-04 00 BE AF ввЛво Лв·ав<пв Лв
00005040: 48 00 43 8C-21 F0 A0 03-FF 1F 02 3C-21 E8 C0 03 Н CM!Ёав вв<!шв
00005050: FF FF 42 34-24 10 62 00-00 A0 03 3C-25 10 43 00 B4$вв ав<%вв
00005060: 04 00 BE 8F-08 00 E0 03-08 00 BD 27-F8 FF BD 27 вв<Пв ввЛво Лв
00005070: 21 10 00 00-04 00 BE AF-08 00 80 14-21 F0 A0 03 !вв вв<пв Ав!Ёав
00005080: A0 B0 03 3C-21 E8 C0 03-44 29 02 7C-3C 00 62 AC ав<!швD|< вв
00005090: 04 00 BE 8F-08 00 E0 03-08 00 BD 27-01 00 03 24 вв<Пв ввЛво вв$
000050A0: 44 29 62 7C-A0 B0 03 3C-21 E8 C0 03-3C 00 62 AC D|ав<!шв вв вв
000050B0: 04 00 BE 8F-08 00 E0 03-08 00 BD 27-F8 FF BD 27 вв<Пв ввЛво Лв
000050C0: A0 B0 02 3C-04 00 BE AF-84 00 43 8C-21 F0 A0 03 ав<пв СМ!Ёав
000050D0: 21 E8 C0 03-C4 FF 03 7C-84 00 43 AC-04 00 BE 8F !шв- вв|Д СМв вв
000050E0: 08 00 E0 03-08 00 BD 27-F8 FF BD 27-A0 B0 02 3C вв ввЛво Лв·ав<
000050F0: 04 00 BE AF-20 00 43 8C-21 F0 A0 03-01 00 04 24 вв<пв СМ!Ёав вв$
00005100: 21 E8 C0 03-44 08 83 7C-20 00 43 AC-04 00 BE 8F !швДвГ| СМв вв
00005110: 08 00 E0 03-08 00 BD 27-F8 FF BD 27-A0 B0 02 3C вв ввЛво Лв·ав<
00005120: 04 00 BE AF-20 00 43 8C-21 F0 A0 03-21 E8 C0 03 вв<пв СМ!Ёав!шв
00005130: 44 08 03 7C-20 00 43 AC-04 00 BE 8F-08 00 E0 03 D| СМв вв вв
00005140: 08 00 BD 27-F8 FF BD 27-A0 B0 03 3C-04 00 BE AF вв<Пв ввЛво Лв
00005150: 10 00 62 8C-01 00 08 24-04 A5 02 7D-08 00 09 24 вв вв{ } вв вв
00005160: 10 00 62 AC-04 7B 22 7D-04 48 02 7C-04 84 02 7D вв вв вв вв
00005170: 10 00 62 AC-21 F0 A0 03-21 18 00 00-A0 B0 0B 3C вв вв!Ёав! вв
00005180: 51 00 0A 24-02 00 88 94-00 00 89 94-00 44 08 00 Q вв ИФ ЙФ Д|
00005190: 25 40 09 01-01 00 63 24-14 00 68 AD-F9 FF 6A 14 %@ вв вв вв
000051A0: 04 00 84 24-21 18 00 00-A0 B0 0A 3C-07 00 09 24 вв вв вв вв
000051B0: 02 00 A4 94-00 00 A8 94-00 24 04 00-25 20 88 00 вв вв вв вв

```

1Global 2FilBlk 3CryBlk 4ReLoad 5 6String 7Direct 8Table 9 10Leave 11

Figure 5.9: Hiew: very typical MIPS code

Another example of such pattern here is book: [9.5 on page 967](#).

## Sparse files

This is sparse file with data scattered amidst almost empty file. Each space character here is in fact zero byte (which is looks like space). This is a file to program FPGA (Altera Stratix GX device). Of course, files like these can be compressed easily, but formats like this one are very popular in scientific and engineering software where efficient access is important while compactness is not.

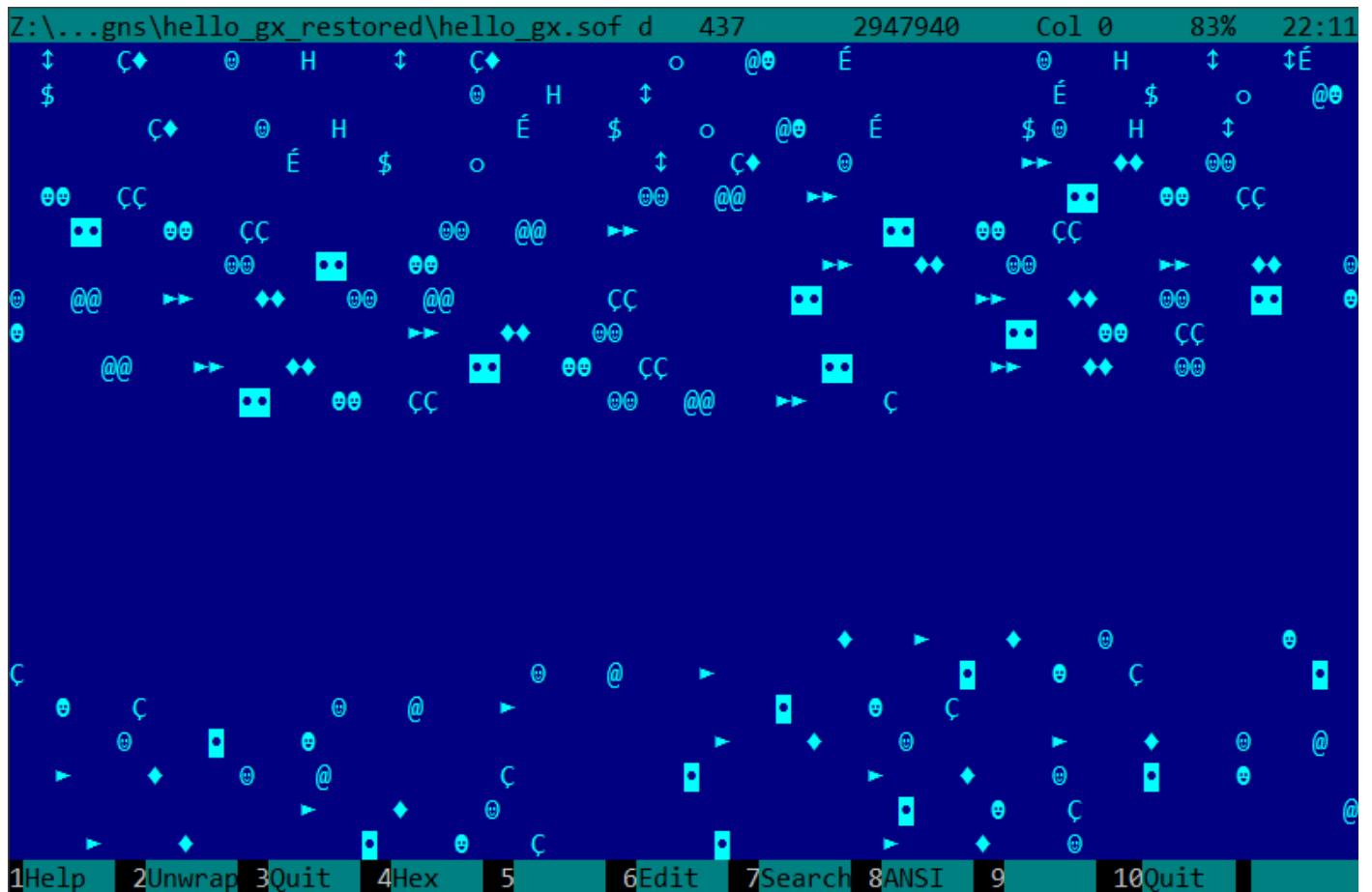


Figure 5.10: FAR: Sparse file

**Compressed file**

This file is just some compressed archive. It has relatively high entropy and visually looks just chaotic. This is how compressed and/or encrypted files looks like.



Figure 5.11: FAR: Compressed file

**CDFS<sup>26</sup>**

OS installations are usually distributed as ISO files which are copies of CD/DVD discs. Filesystem used is named CDFS, here is you see file names mixed with some additional data. This can be file sizes, pointers to another directories, file attributes, etc. This is how typical filesystems may look internally.

```
Z:\...\untu\ubuntu-15.10-desktop-i386.iso d 437 1226964992 Col 0 0% 21:59

ä # # • • s¤§»; @ @ 00 SP•@Jn
PX$@mA Am@ @ TF→@Jss¤§»; s¤§»; s¤§»; CEL@$ $ φ
φ ` # # • • s¤§»; @ @ 000PX$@mA Am@ @ TF→@Jss¤§»; s¤§»;
$¤§»; s¤§»; n % % • • s¤§»; @ @ 0+DISKPX$@mA Am@ @ TF
→@Jss¤§»;% s¤§»; s¤§»;% NM@ .diskn & & • • s¤§»; @ @ 0+BOOT PX$@mA Am@ @
@ TF→@Jss¤§»( s¤§»; s¤§»( NM@ boot r ( ( • • s¤§»; @ @ 0
↑CASPER PX$@mA Am@ @ TF→@Jss¤§»( s¤§»; s¤§»( NM@ casper n )
) • • s¤§»; @ @ 0+DISTSPX$@mA Am@ @ TF→@Jss¤§»; s¤§»; s¤§»;
s¤§»; NM@ distsr 1 1 • • s¤§»; @ @ 0+INSTALLPX$@mA Am@ @
TF→@Jss¤§»( s¤§»; s¤§»( NM@ installv 2 2 H H s¤§»; @ @ 0+ISOLINUX PX$@mA Am@ @
TF→@Jss¤§»( s¤§»( NM@ isolinux z p@ op` J
J s¤§»; @ @ 0+MD5SUM.TXT PX$@ü ü$@ @ TF→@Jss¤§»; s¤§»; s¤§»;
: NM@ md5sum.txt n ; ; • • s¤§»; @ @ 0+PICS PX$@mA Am@ @
TF→@Jss¤§»; s¤§»; s¤§»; NM@ pics n < < • • s¤§»; @ @ 0+POOL PX$@mA
Am@ @ TF→@Jss¤§»; s¤§»; s¤§»; NM@ pool r W W • • s¤§»; @ @
@ 0+PRESEEDPX$@mA Am@ @ TF→@Jss¤§»; s¤§»; s¤§»; NM@ preseed
è S@ 0S@ as¤§»; @ @ 0+README.DISKDEFINES PX$@ü ü$@ @
TF→@Jss¤§»; s¤§»; NM@ README.diskdefines x ≤ ≤ s¤§»; @ @ 0+UBUNT
U.PX$@mi im@ @ TF→@Jss¤§»; s¤§»; s¤§»; NM@ ubuntuSL@ @
```

1Help 2Unwrap 3Quit 4Hex 5 6Edit 7Search 8ANSI 9 10Quit

Figure 5.12: FAR: ISO file: Ubuntu 15 installation [CD<sup>27</sup>](#)<sup>26</sup>Compact Disc File System

## **32-bit x86 executable code**

This is how 32-bit x86 executable code looks like. It has not very high entropy, because some bytes occurred more often than others.

Figure 5.13: FAR: Executable 32-bit x86 code

## 5.10. OTHER THINGS

### BMP graphics files

BMP files are not compressed, so each byte (or group of bytes) describes each pixel. I've found this picture somewhere inside my installed Windows 8.1:



Figure 5.14: Example picture

You see that this picture has some pixels which unlikely can be compressed very good (around center), but there are long one-color lines at top and bottom. Indeed, lines like these also looks as lines during viewing the file:

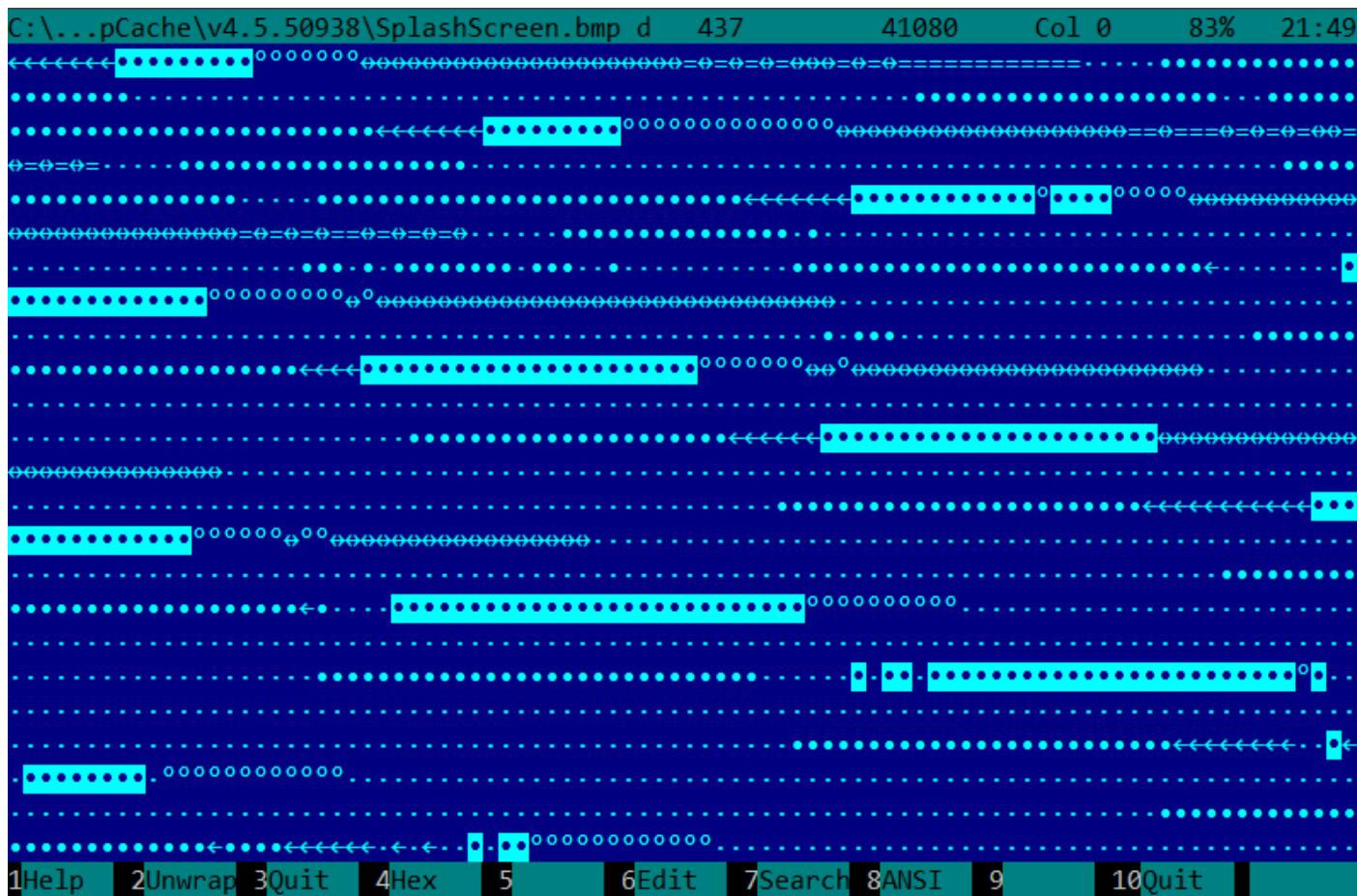


Figure 5.15: BMP file fragment

### 5.10.6 Memory “snapshots” comparing

The technique of the straightforward comparison of two memory snapshots in order to see changes was often used to hack 8-bit computer games and for hacking “high score” files.

## **5.10. OTHER THINGS**

For example, if you had a loaded game on an 8-bit computer (there isn't much memory on these, but the game 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 once, the bullet count goes to 99, do a second "snapshot" and then compare both: it must be a byte somewhere which has been 100 at the beginning, and now it is 99.

Considering the fact that 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 has holding the bullet count. If you searched for all references to the address in the disassembled game code, it was not very hard to find a piece of code **decrementing** the bullet count, then to write a **NOP** instruction there, or a couple of **NOP**-s, and then have a game with 100 bullets forever. Games on these 8-bit computers were commonly loaded at the constant address, also, there were not much different versions of each game (commonly just one version was popular for a long span of time), so enthusiastic gamers knew which bytes must be overwritten (using the BASIC's instruction **POKE**) at which address in order to hack it. This led to "cheat" lists that contained **POKE** instructions, published in magazines related to 8-bit games. See also: [wikipedia](#).

Likewise, it is easy to modify "high score" files, this does not work with just 8-bit games. Notice your score count and back up the file somewhere. When the "high score" count gets different, just compare the two files, it can even be done with the DOS utility FC<sup>28</sup> ("high score" files are often in binary form).

There will be a point where a couple of bytes are different and it is easy to see which ones are holding the score number. However, game developers are fully aware of such tricks and may defend the program against it.

Somewhat similar example in this book is: [9.3 on page 955](#).

## **Windows registry**

It is also possible to compare the Windows registry before and after a program installation.

It is a very popular method of finding which registry elements are used by the program. Perhaps, this is the reason why the "windows registry cleaner" shareware is so popular.

## **Blink-comparator**

Comparison of files or memory snapshots remind us blink-comparator <sup>29</sup>: a device used by astronomers in past, intended to find moving celestial objects.

Blink-comparator allows to switch quickly between two photographies shot in different time, so astronomer would spot the difference visually.

By the way, Pluto was discovered by blink-comparator in 1930.

---

<sup>28</sup>MS-DOS utility for comparing binary files

<sup>29</sup><http://go.yurichev.com/17348>

# Chapter 6

## OS-specific

### 6.1 Arguments passing methods (calling conventions)

#### 6.1.1 cdecl

This is the most popular method for passing arguments to functions in the C/C++ languages.

The caller also must return the value of the **stack pointer** (`ESP`) to its initial state after the **callee** function exits.

Listing 6.1: cdecl

```
push arg3  
push arg2  
push arg1  
call function  
add esp, 12 ; returns ESP
```

#### 6.1.2 stdcall

It's almost the same as *cdecl*, with the exception that the **callee** must set `ESP` to the initial state by executing the `RET x` instruction instead of `RET`, where `x = arguments number * sizeof(int)`<sup>1</sup>. The **caller** is not adjusting the **stack pointer**, there are no `add esp, x` instruction.

Listing 6.2: stdcall

```
push arg3  
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 can take the function from [1.85 on page 98](#) and change it slightly by adding the `__stdcall` modifier:

```
int __stdcall f2 (int a, int b, int c)  
{  
    return a*b+c;  
};
```

It is to be compiled in almost the same way as [1.86 on page 98](#), but you will see `RET 12` instead of `RET`. `SP` is not updated in the **caller**.

## 6.1. ARGUMENTS PASSING METHODS (CALLING CONVENTIONS)

As a consequence, the number of function arguments can be easily deduced from the `RETN n` instruction: just divide  $n$  by 4.

Listing 6.3: 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
_f2@12 ENDP

; ...
    push    3
    push    2
    push    1
    call    _f2@12
    push    eax
    push    OFFSET $SG81369
    call    _printf
    add    esp, 8
```

### Functions with variable number of arguments

`printf()`-like functions are, probably, the only case of functions with a variable number of arguments in C/C++, but it is easy to illustrate an important difference between *cdecl* and *stdcall* with their help. Let's start with the idea that the compiler knows the argument count of each `printf()` function call.

However, the called `printf()`, which is already compiled and located in MSVCRT.DLL (if we talk about Windows), does not have any information about how much arguments were passed, however it can determine it from the format string.

Thus, if `printf()` would be a *stdcall* function and restored *stack pointer* to its initial state by counting the number of arguments in the format string, this could be a dangerous situation, when one programmer's typo can provoke a sudden program crash. Thus it is not suitable for such functions to use *stdcall*, *cdecl* is better.

### 6.1.3 fastcall

That's the general naming for the method of passing some arguments via registers and the rest via the stack. It worked faster than *cdecl/stdcall* on older CPUs (because of smaller stack pressure). It may not help to gain any significant performance on latest (much more complex) CPUs, however.

It is not standardized, so the various compilers can do it differently. It's a well known caveat: if you have two DLLs and the one uses another one, and they are built by different compilers with different *fastcall* calling conventions, you can expect problems.

Both MSVC and GCC pass the first and second arguments via `ECX` and `EDX` and the rest of the arguments via the stack.

The *stack pointer* must be restored to its initial state by the *callee* (like in *stdcall*).

Listing 6.4: fastcall

```
push arg3
mov edx, arg2
mov ecx, arg1
call function

function:
```

## 6.1. ARGUMENTS PASSING METHODS (CALLING CONVENTIONS)

```
.. do something ..  
ret 4
```

For example, we may take the function from [1.85 on page 98](#) and change it slightly by adding a `__fastcall` modifier:

```
int __fastcall f3 (int a, int b, int c)  
{  
    return a*b+c;  
};
```

Here is how it is to be compiled:

Listing 6.5: Optimizing MSVC 2010 /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 the **callee** returns **SP** by using the `RETN` instruction with an operand.

Which implies that the number of arguments can be deduced easily here as well.

### GCC `regparm`

It is the evolution of `fastcall`<sup>2</sup> in some sense. With the `-mregparm` option it is possible to set how many arguments are to be passed via registers (3 is the maximum). Thus, the `EAX`, `EDX` and `ECX` registers are to be used.

Of course, if the number the of arguments is less than 3, not all 3 registers are to be used.

The **caller** restores the **stack pointer** to its initial state.

For example, see ([1.22.1 on page 306](#)).

### Watcom/OpenWatcom

Here it is called “register calling convention”. The first 4 arguments are passed via the `EAX`, `EDX`, `EBX` and `ECX` registers. All the rest—via the stack.

These functions has an underscore appended to the function name in order to distinguish them from those having a different calling convention.

<sup>2</sup><http://go.yurichev.com/17040>

**6.1.4 thiscall**

This is passing the object's *this* pointer to the function-method, in C++.

In MSVC, *this* is usually passed in the ECX register.

In GCC, the *this* pointer is passed as the first function-method argument. Thus it will be very visible that internally: all function-methods have an extra argument.

For an example, see ([3.19.1 on page 546](#)).

**6.1.5 x86-64****Windows x64**

The method of for passing arguments in Win64 somewhat resembles fastcall. The first 4 arguments are passed via RCX, RDX, R8 and R9, the rest—via the stack. The caller also must prepare space for 32 bytes or 4 64-bit values, so then the callee can save there the first 4 arguments. Short functions may use the arguments' values just from the registers, but larger ones may save their values for further use.

The caller also must return the stack pointer into its 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\n", a, b, c, d, e, f, g);
}

int main()
{
    f1(1,2,3,4,5,6,7);
}
```

Listing 6.6: MSVC 2012 /0b

```
$SG2937 DB      '%d %d %d %d %d %d', 0aH, 00H

main PROC
    sub    rsp, 72

    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
    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
```

## 6.1. ARGUMENTS PASSING METHODS (CALLING CONVENTIONS)

```

    mov    DWORD PTR [rsp+24], r8d
    mov    DWORD PTR [rsp+16], edx
    mov    DWORD PTR [rsp+8], ecx
    sub    rsp, 72

    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
    ret    0
f1    ENDP

```

Here we clearly see how 7 arguments are passed: 4 via registers and the remaining 3 via the stack.

The code of the f1() function's prologue saves the arguments in the “scratch space”—a space in the stack intended exactly for this purpose.

This is arranged so because the compiler cannot be sure that there will be enough registers to use without these 4, which will otherwise be occupied by the arguments until the function's execution end.

The “scratch space” allocation in the stack is the caller's duty.

Listing 6.7: Optimizing MSVC 2012 /O**b**

```

$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

    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
    ret    0
f1    ENDP

main  PROC
sub    rsp, 72

    mov    edx, 2
    mov    DWORD PTR [rsp+48], 7
    mov    DWORD PTR [rsp+40], 6

```

## 6.1. ARGUMENTS PASSING METHODS (CALLING CONVENTIONS)

```
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
ret    0
main  ENDP
```

If we compile the example with optimizations, it is to be almost the same, but the “scratch space” will not be used, because it won’t be needed.

Also take a look on how MSVC 2012 optimizes the loading of primitive values into registers by using LEA ([1.6 on page 1009](#)). MOV would be 1 byte longer here (5 instead of 4).

Another example of such thing is: [8.1.1 on page 795](#).

### Windows x64: Passing *this* (C/C++)

The *this* pointer is passed in RCX, the first argument of the method is in RDX, etc. For an example see: [3.19.1 on page 548](#).

### Linux x64

The way arguments are 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 is no “scratch space”, although the callee may save the register values in the stack, if it needs/wants to.

Listing 6.8: Optimizing GCC 4.7.3

```
.LC0:
    .string "%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 the values are written into the 32-bit parts of the registers (e.g., EAX) but not in the whole 64-bit register (RAX). This is because each write to the low 32-bit part of a register automatically clears the high 32 bits. Supposedly, it was decided in AMD to do so to simplify porting code to x86-64.

**6.1.6 Return values of *float* and *double* type**

In all conventions except in Win64, the values of type *float* or *double* are returned via the FPU register ST(0).

In Win64, the values of *float* and *double* types are returned in the low 32 or 64 bits of the XMM0 register.

**6.1.7 Modifying arguments**

Sometimes, C/C++ programmers (not limited to these PLs, though), may ask, what can happen if they modify the arguments?

The answer is simple: the arguments are stored in the stack, that is where the modification takes place.

The calling functions is not using them after the callee's exit (the author of these lines has never seen any such case in his practice).

```
#include <stdio.h>

void f(int a, int b)
{
    a=a+b;
    printf ("%d\n", a);
}
```

Listing 6.9: 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 can modify the arguments easily. Of course, if it is not references in C++ ([3.19.3 on page 562](#)), and if you don't modify data to which a pointer points to, then the effect will not propagate outside the current function.

Theoretically, after the callee's return, the caller could get the modified argument and use it somehow. Maybe if it is written directly in assembly language.

For example, code like this will be generated by usual C/C++ compiler:

```
push    456      ; will be b
push    123      ; will be a
call    f         ; f() modifies its first argument
add     esp, 2*4
```

We can rewrite this code like:

```
push    456      ; will be b
push    123      ; will be a
call    f         ; f() modifies its first argument
pop     eax
add     esp, 4
; EAX=1st argument of f() modified in f()
```

Hard to imagine, why anyone would need this, but this is possible in practice. Nevertheless, the C/C++ languages standards don't offer any way to do so.

## 6.1. ARGUMENTS PASSING METHODS (CALLING CONVENTIONS)

### 6.1.8 Taking a pointer to function argument

...even more than that, it's possible to take a pointer to the function's argument and pass it to another function:

```
#include <stdio.h>

// located in some other file
void modify_a (int *a);

void f (int a)
{
    modify_a (&a);
    printf ("%d\n", a);
}
```

It's hard to understand how it works until we can see the code:

Listing 6.10: Optimizing MSVC 2010

```
$SG2796 DB      '%d', 0aH, 00H

_a$ = 8
_f      PROC
    lea    eax, DWORD PTR _a$[esp-4] ; just get the address of value in local stack
    push   eax                      ; and pass it to modify_a()
    call   _modify_a
    mov    ecx, DWORD PTR _a$[esp]   ; reload it from the local stack
    push   ecx                      ; and pass it to printf()
    push   OFFSET $SG2796 ; '%d'
    call   _printf
    add    esp, 12
    ret    0
_f      ENDP
```

The address of the place in the stack where *a* has been passed is just passed to another function. It modifies the value addressed by the pointer and then `printf()` prints the modified value.

The observant reader might ask, what about calling conventions where the function's arguments are passed in registers?

That's a situation where the *Shadow Space* is used.

The input value is copied from the register to the *Shadow Space* in the local stack, and then this address is passed to the other function:

Listing 6.11: Optimizing MSVC 2012 x64

```
$SG2994 DB      '%d', 0aH, 00H

a$ = 48
_f      PROC
    mov    DWORD PTR [rsp+8], ecx ; save input value in Shadow Space
    sub    rsp, 40
    lea    rcx, QWORD PTR a$[rsp] ; get address of value and pass it to modify_a()
    call   modify_a
    mov    edx, DWORD PTR a$[rsp] ; reload value from Shadow Space and pass it to printf
    ↴ ()
    lea    rcx, OFFSET FLAT:$SG2994 ; '%d'
    call   printf
    add    rsp, 40
    ret    0
_f      ENDP
```

GCC also stores the input value in the local stack:

Listing 6.12: Optimizing GCC 4.9.1 x64

```
.LC0:
    .string "%d\n"
f:
```

## 6.2. THREAD LOCAL STORAGE

```
sub    rsp, 24
mov    DWORD PTR [rsp+12], edi ; store input value to the local stack
lea    rdi, [rsp+12]           ; take an address of the value and pass it to modify_a()
        ()
call   modify_a
mov    edx, DWORD PTR [rsp+12] ; reload value from the local stack and pass it to printf()
        ()
printf()
    mov    esi, OFFSET FLAT:.LC0      ; '%d'
    mov    edi, 1
    xor    eax, eax
    call   __printf_chk
    add    rsp, 24
    ret
```

GCC for ARM64 does the same, but this space is called *Register Save Area* here:

Listing 6.13: Optimizing GCC 4.9.1 ARM64

```
f:
stp    x29, x30, [sp, -32]!
add    x29, sp, 0          ; setup FP
add    x1, x29, 32         ; calculate address of variable in Register Save Area
str    w0, [x1,-4]!        ; store input value there
mov    x0, x1              ; pass address of variable to the modify_a()
bl    modify_a
ldr    w1, [x29,28]        ; load value from the variable and pass it to printf()
adrp  x0, .LC0 ; '%d'
add    x0, x0, :lo12:.LC0
bl    printf              ; call printf()
ldp    x29, x30, [sp], 32
ret
.LC0:
.string "%d\n"
```

By the way, a similar usage of the *Shadow Space* is also considered here: [3.14.1 on page 520](#).

## 6.2 Thread Local Storage

TLS is a data area, specific to each thread. Every thread can store what it needs there. One well-known example is the C standard global variable *errno*.

Multiple threads may simultaneously call functions which return an error code in *errno*, so a global variable will not work correctly here for multi-threaded programs, so *errno* must be stored in the [TLS](#).

In the C++11 standard, a new *thread\_local* modifier was added, showing that each thread has its own version of the variable, it can be initialized, and it is located in the [TLS](#)<sup>3</sup>:

Listing 6.14: C++11

```
#include <iostream>
#include <thread>

thread_local int tmp=3;

int main()
{
    std::cout << tmp << std::endl;
}
```

Compiled in MinGW GCC 4.8.1, but not in MSVC 2012.

If we talk about PE files, in the resulting executable file, the *tmp* variable is to be allocated in the section devoted to the [TLS](#).

<sup>3</sup> C11 also has thread support, optional though

## 6.2.1 Linear congruential generator revisited

The pseudorandom number generator we considered earlier [1.23 on page 339](#) has a flaw: it's not thread-safe, because it has an internal state variable which can be read and/or modified in different threads simultaneously.

### Win32

#### Uninitialized TLS data

One solution is to add `__declspec( thread )` modifier to the global variable, then it will be allocated in the [TLS](#) (line 9):

```

1 #include <stdint.h>
2 #include <windows.h>
3 #include <winnt.h>
4
5 // from the Numerical Recipes book:
6 #define RNG_a 1664525
7 #define RNG_c 1013904223
8
9 __declspec( thread ) uint32_t rand_state;
10
11 void my_srand (uint32_t init)
12 {
13     rand_state=init;
14 }
15
16 int my_rand ()
17 {
18     rand_state=rand_state*RNG_a;
19     rand_state=rand_state+RNG_c;
20     return rand_state & 0x7fff;
21 }
22
23 int main()
24 {
25     my_srand(0x12345678);
26     printf ("%d\n", my_rand());
27 }
```

Hiew shows us that there is a new PE section in the executable file: `.tls`.

Listing 6.15: Optimizing MSVC 2013 x86

```

_TLS    SEGMENT
_rand_state DD 01H DUP (?)
_TLS    ENDS

_DATA   SEGMENT
$SG84851 DB      '%d', 0Ah, 00H
_DATA   ENDS
_TEXT   SEGMENT

_init$ = 8      ; size = 4
_my_srand PROC
; FS:0=address of TIB
    mov     eax, DWORD PTR fs:_tls_array  ; displayed in IDA as FS:2Ch
; EAX=address of TLS of process
    mov     ecx, DWORD PTR _tls_index
    mov     ecx, DWORD PTR [eax+ecx*4]
; ECX=current TLS segment
    mov     eax, DWORD PTR _init$[esp-4]
    mov     DWORD PTR _rand_state[ecx], eax
    ret     0
_my_srand ENDP
```

## 6.2. THREAD LOCAL STORAGE

```
_my_rand PROC  
; FS:0=address of TIB  
    mov     eax, DWORD PTR fs:_tls_array ; displayed in IDA as FS:2Ch  
; EAX=address of TLS of process  
    mov     ecx, DWORD PTR __tls_index  
    mov     ecx, DWORD PTR [eax+ecx*4]  
; ECX=current TLS segment  
    imul    eax, DWORD PTR _rand_state[ecx], 1664525  
    add    eax, 1013904223           ; 3c6ef35fH  
    mov    DWORD PTR _rand_state[ecx], eax  
    and    eax, 32767            ; 00007ffffH  
    ret    0  
_my_rand ENDP  
  
_TEXT    ENDS
```

`rand_state` is now in the [TLS](#) segment, and each thread has its own version of this variable.

Here is how it's accessed: load the address of the [TIB](#) from FS:2Ch, then add an additional index (if needed), then calculate the address of the [TLS](#) segment.

Then it's possible to access the `rand_state` variable through the ECX register, which points to an unique area in each thread.

The `FS:` selector is familiar to every reverse engineer, it is specially used to always point to [TIB](#), so it would be fast to load the thread-specific data.

The `GS:` selector is used in Win64 and the address of the [TLS](#) is 0x58:

Listing 6.16: Optimizing MSVC 2013 x64

```
_TLS    SEGMENT  
rand_state DD  01H DUP (?)  
_TLS    ENDS  
  
_DATA   SEGMENT  
$SG85451 DB  '%d', 0ah, 00h  
_DATA   ENDS  
  
_TEXT   SEGMENT  
  
init$ = 8  
my_srand PROC  
    mov     edx, DWORD PTR __tls_index  
    mov     rax, QWORD PTR gs:88 ; 58h  
    mov     r8d, OFFSET FLAT:rand_state  
    mov     rax, QWORD PTR [rax+r8d*8]  
    mov     DWORD PTR [r8+rax], ecx  
    ret    0  
my_srand ENDP  
  
my_rand PROC  
    mov     rax, QWORD PTR gs:88 ; 58h  
    mov     ecx, DWORD PTR __tls_index  
    mov     edx, OFFSET FLAT:rand_state  
    mov     rcx, QWORD PTR [rax+rcx*8]  
    imul    eax, DWORD PTR [rcx+rdx], 1664525 ; 0019660dH  
    add    eax, 1013904223           ; 3c6ef35fH  
    mov    DWORD PTR [rcx+rdx], eax  
    and    eax, 32767            ; 00007ffffH  
    ret    0  
my_rand ENDP  
  
_TEXT   ENDS
```

### Initialized [TLS](#) data

## 6.2. THREAD LOCAL STORAGE

Let's say, we want to set some fixed value to `rand_state`, so in case the programmer forgets to, the `rand_state` variable would be initialized to some constant anyway (line 9):

```
1 #include <stdint.h>
2 #include <windows.h>
3 #include <winnt.h>
4
5 // from the Numerical Recipes book:
6 #define RNG_a 1664525
7 #define RNG_c 1013904223
8
9 __declspec( thread ) uint32_t rand_state=1234;
10
11 void my_srand (uint32_t init)
12 {
13     rand_state = init;
14 }
15
16 int my_rand ()
17 {
18     rand_state = rand_state * RNG_a;
19     rand_state = rand_state + RNG_c;
20     return rand_state & 0x7fff;
21 }
22
23 int main()
24 {
25     printf ("%d\n", my_rand());
26 }
```

The code is not different from what we already saw, but in IDA we see:

```
.tls:00404000 ; Segment type: Pure data
.tls:00404000 ; Segment permissions: Read/Write
.tls:00404000 _tls          segment para public 'DATA' use32
.tls:00404000             assume cs:_tls
.tls:00404000             ;org 404000h
.tls:00404000 TlsStart      db    0          ; DATA XREF: .rdata:TlsDirectory
.tls:00404001             db    0
.tls:00404002             db    0
.tls:00404003             db    0
.tls:00404004             dd   1234
.tls:00404008 TlsEnd        db    0          ; DATA XREF: .rdata:TlsEnd_ptr
...
...
```

1234 is there and every time a new thread starts, a new **TLS** is allocated for it, and all this data, including 1234, will be copied there.

This is a typical scenario:

- Thread A is started. A **TLS** is created for it, 1234 is copied to `rand_state`.
- The `my_rand()` function is called several times in thread A.  
`rand_state` is different from 1234.
- Thread B is started. A **TLS** is created for it, 1234 is copied to `rand_state`, while thread A has a different value in the same variable.

### TLS callbacks

But what if the variables in the **TLS** have to be filled with some data that must be prepared in some unusual way?

Let's say, we've got the following task: the programmer can forget to call the `my_srand()` function to initialize the **PRNG**, but the generator has to be initialized at start with something truly random, instead of 1234. This is a case in which **TLS** callbacks can be used.

## 6.2. THREAD LOCAL STORAGE

The following code is not very portable due to the hack, but nevertheless, you get the idea.

What we do here is define a function (`tls_callback()`) which is to be called *before* the process and/or thread start.

The function initializes the PRNG with the value returned by `GetTickCount()` function.

```
#include <stdint.h>
#include <windows.h>
#include <winnt.h>

// from the Numerical Recipes book:
#define RNG_a 1664525
#define RNG_c 1013904223

__declspec( thread ) uint32_t rand_state;

void my_srand (uint32_t init)
{
    rand_state=init;
}

void NTAPI tls_callback(PVOID a, DWORD dwReason, PVOID b)
{
    my_srand (GetTickCount());
}

#pragma data_seg(".CRT$XLB")
PIMAGE_TLS_CALLBACK p_thread_callback = tls_callback;
#pragma data_seg()

int my_rand ()
{
    rand_state=rand_state*RNG_a;
    rand_state=rand_state+RNG_c;
    return rand_state & 0x7fff;
}

int main()
{
    // rand_state is already initialized at the moment (using GetTickCount())
    printf ("%d\n", my_rand());
};
```

Let's see it in IDA:

Listing 6.17: Optimizing MSVC 2013

```
.text:00401020 TlsCallback_0    proc near          ; DATA XREF: .rdata:TlsCallbacks
.text:00401020                 call    ds:GetTickCount
.text:00401026                 push    eax
.text:00401027                 call    my_srand
.text:0040102C                 pop    ecx
.text:0040102D                 retn    0Ch
.text:0040102D TlsCallback_0    endp

...
.rdata:004020C0 TlsCallbacks    dd offset TlsCallback_0 ; DATA XREF: .rdata:TlsCallbacks_ptr
...
.rdata:00402118 TlsDirectory   dd offset TlsStart
.rdata:0040211C TlsEnd_ptr     dd offset TlsEnd
.rdata:00402120 TlsIndex_ptr   dd offset TlsIndex
.rdata:00402124 TlsCallbacks_ptr dd offset TlsCallbacks
.rdata:00402128 TlsSizeOfZeroFill dd 0
.rdata:0040212C TlsCharacteristics dd 300000h
```

TLS callback functions are sometimes used in unpacking routines to obscure their processing.

### 6.3. SYSTEM CALLS (SYSCALL-S)

Some people may be confused and be in the dark that some code executed right before the [OEP](#)<sup>4</sup>.

#### Linux

Here is how a thread-local global variable is declared in GCC:

```
_thread uint32_t rand_state=1234;
```

This is not the standard C/C++ modifier, but a rather GCC-specific one <sup>5</sup>.

The `GS:` selector is also used to access the [TLS](#), but in a somewhat different way:

Listing 6.18: Optimizing GCC 4.8.1 x86

```
.text:08048460 my_srand    proc  near
.text:08048460
.text:08048460 arg_0        = dword ptr  4
.text:08048460
.text:08048460             mov     eax, [esp+arg_0]
.text:08048464             mov     gs:0FFFFFFFCh, eax
.text:0804846A             retn
.text:0804846A my_srand    endp

.text:08048470 my_rand     proc  near
.text:08048470             imul   eax, gs:0FFFFFFFCh, 19660Dh
.text:0804847B             add    eax, 3C6EF35Fh
.text:08048480             mov     gs:0FFFFFFFCh, eax
.text:08048486             and    eax, 7FFFh
.text:0804848B             retn
.text:0804848B my_rand    endp
```

More about it: [Ulrich Drepper, *ELF Handling For Thread-Local Storage*, (2013)]<sup>6</sup>.

## 6.3 System calls (syscall-s)

As we know, all running processes inside an [OS](#) are divided into two categories: those having full access to the hardware (“kernel space”) and those that do not (“user space”).

The [OS](#) kernel and usually the drivers are in the first category.

All applications are usually in the second category.

For example, Linux kernel is in *kernel space*, but Glibc in *user space*.

This separation is crucial for the safety of the [OS](#): it is very important not to give to any process the possibility to screw up something in other processes or even in the [OS](#) kernel. On the other hand, a failing driver or error inside the [OS](#)’s kernel usually leads to a kernel panic or [BSOD](#)<sup>7</sup>.

The protection in the x86 processors 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) are a point where these two areas are connected.

It can be said that this is the main [API](#) provided to applications.

As in [Windows NT](#), the syscalls table resides in the [SSDT](#)<sup>8</sup>.

The 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, but it is easier to use syscalls. However, much more code has to be written due to the lower level of abstraction of the [API](#).

It is also worth noting that the syscall numbers may be different in various OS versions.

<sup>4</sup>Original Entry Point

<sup>5</sup><http://go.yurichev.com/17062>

<sup>6</sup>Also available as <http://go.yurichev.com/17272>

<sup>7</sup>Blue Screen of Death

<sup>8</sup>System Service Dispatch Table

## 6.3.1 Linux

In Linux, a syscall is usually called via `int 0x80`. The call's number is passed in the `EAX` register, and any other parameters —in the other registers.

Listing 6.19: A simple example of the usage of two syscalls

```
section .text
global _start

_start:
    mov     edx,len ; buffer len
    mov     ecx,msg ; buffer
    mov     ebx,1    ; file descriptor. 1 is for stdout
    mov     eax,4    ; syscall number. 4 is for sys_write
    int     0x80

    mov     eax,1    ; syscall number. 1 is for sys_exit
    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://go.yurichev.com/17319>.

For system calls interception and tracing in Linux, strace( [7.2.3 on page 789](#)) can be used.

## 6.3.2 Windows

Here they are called via `int 0x2e` or using the special x86 instruction `SYSENTER`.

The full list of syscalls in Windows: <http://go.yurichev.com/17320>.

Further reading:

“Windows Syscall Shellcode” by Piotr Bania: <http://go.yurichev.com/17321>.

## 6.4 Linux

### 6.4.1 Position-independent code

While analyzing Linux shared (.so) libraries, one may frequently spot this code pattern:

Listing 6.20: 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
```

## 6.4. LINUX

```

.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 - 1AF000h)[ebx] ; "__gen_tempname"
.text:000579F6          mov     [esp+0ACh+var_A0], eax
.text:000579FA          lea     eax, (a_SysdepsPosix - 1AF000h)[ebx] ; "../sysdeps/✓
    ↳ posix/tempname.c"
.text:00057A00          mov     [esp+0ACh+var_A8], eax
.text:00057A04          lea     eax, (aInvalidKindIn_ - 1AF000h)[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 some constants and the value in `EBX`, which is calculated at the beginning of each function.

This is the so-called [PIC](#), it is intended to be executable if 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 is still crucial today in embedded systems without virtual memory support (where all processes are placed in a single continuous memory block).

It is also still used in \*NIX systems for shared libraries, since they are shared across many processes while loaded in memory only once. But all these processes can map the same shared library at different addresses, so that is why a shared library has to work correctly without using any absolute addresses.

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 the resulting .so file in [IDA](#):

```
gcc -fPIC -shared -O3 -o 1.so 1.c
```

Listing 6.21: 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:00000573          call   __x86_get_pc_thunk_bx

```

## 6.4. LINUX

```
.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: the pointers to «returning %d\n» and *global\_variable* are to be corrected at each function execution.

The `__x86_get_pc_thunk_bx()` function returns in `EBX` the address of the point after a call to itself (`0x57C` here).

That's a simple way to get the value of the program counter (`EIP`) at some point. The `0x1A84` constant is related to the difference between this function's start and the so-called *Global Offset Table Procedure Linkage Table* (GOT PLT), the section right after the *Global Offset Table* (GOT), where the pointer to *global\_variable* is. [IDA](#) shows these offsets in their processed form to make them easier to understand, 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]
```

Here `EBX` points to the `GOT PLT` section and to calculate a pointer to *global\_variable* (which is 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 the AMD64 instruction set supports RIP<sup>9</sup>-relative addressing — to simplify PIC-code.

Let's compile the same C code using the same GCC version, but for x64.

[IDA](#) would simplify the resulting code but would suppress the RIP-relative addressing details, so we are going to use `objdump` instead of IDA to see everything:

```
00000000000000720 <f1>:
720: 48 8b 05 b9 08 20 00  mov    rax,QWORD PTR [rip+0x2008b9]      # 200fe0 <_DYNAMIC+0>
     ↓ x1d0>
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 the address of the 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.

<sup>9</sup>program counter in AMD64

## 6.4. LINUX

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 care about performance [see: Agner Fog, *Optimizing software in C++* (2015)].

## Windows

The PIC mechanism is not used in Windows DLLs. If the Windows loader needs to load DLL on another base address, it “patches” the DLL in memory (at the *FIXUP* places) in order to correct all addresses.

This implies that several Windows processes cannot share an once loaded DLL at different addresses in different process’ memory blocks — since each instance that’s loaded in memory is *fixed* to work only at these addresses..

### 6.4.2 *LD\_PRELOAD* hack in Linux

This allows us to load our own dynamic libraries before others, even before system ones, like `libc.so.6`.

This, in turn, allows us to “substitute” our written functions before the original ones in the system libraries. For example, it is easy to intercept all calls to `time()`, `read()`, `write()`, etc.

Let’s see if we can fool the *uptime* utility. As we know, it tells how long the computer has been working. With the help of `strace` ([7.2.3 on page 789](#)), it is possible to see that the utility takes this information 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 and its contents are generated on fly in the Linux kernel. There are just two numbers:

```
$ cat /proc/uptime
416690.91 415152.03
```

What we can learn from Wikipedia <sup>[10](#)</sup>:

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.

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 the name of the file to be opened with what we need and if it is so, it will write down the descriptor of the file opened.

Second, `read()`, if called for this file descriptor, will substitute the output, and in the rest of the cases will call the original `read()` from `libc.so.6`. And also `close()`, will note if the file we are currently following is to be closed.

We are going to use the `dlopen()` and `dlsym()` functions to determine the original function addresses in `libc.so.6`.

We need them because we must pass control to the “real” functions.

On the other hand, if we intercepted `strcmp()` and monitored each string comparisons in the program, then we would have to implement our version of `strcmp()`, and not use the original function <sup>[11](#)</sup>, that would be easier.

<sup>10</sup>[wikipedia](#)

<sup>11</sup>For example, here is how simple `strcmp()` interception works in this article <sup>[12](#)</sup> written by Yong Huang

## 6.4. LINUX

```
#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 initied = 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 (initied)
        return;

    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");

    initied = 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);
```

## 6.5. WINDOWS NT

```
};

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 sprintf (buf, count, "%d %d", 0x7fffffff, 0x7fffffff)+1;
    };
    // not our file, go to real read() function
    return (*read_ptr)(fd, buf, count);
};
```

( [Source code at GitHub](#) )

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 the 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 always points to the filename and path of our library, it is to be loaded for all starting programs.

More examples:

- Very simple interception of the strcmp() (Yong Huang) <http://go.yurichev.com/17143>
- Kevin Pulo—Fun with LD\_PRELOAD. A lot of examples and ideas. [yurichev.com](http://yurichev.com)
- File functions interception for compression/decompression files on fly (zlibc). <http://go.yurichev.com/17146>

## 6.5 Windows NT

### 6.5.1 CRT (win32)

Does the program execution start right at the `main()` function? No, it does not.

If we would open any executable file in IDA or HIEW, we can see OEP pointing to some another code block.

This code is doing some maintenance and preparations before passing control flow to our code. It is called startup-code or CRT code (C RunTime).

The `main()` function takes an array of the arguments passed on the command line, and also one with environment variables. But in fact a generic string is passed to the program, the CRT code finds the spaces in it and cuts it in parts. The CRT code also prepares the environment variables array `envp`.

As for GUI<sup>13</sup> win32 applications, `WinMain` is used instead of `main()`, having its own arguments:

```
int CALLBACK WinMain(
    _In_ HINSTANCE hInstance,
    _In_ HINSTANCE hPrevInstance,
    _In_ LPSTR lpCmdLine,
    _In_ int nCmdShow
);
```

<sup>13</sup>Graphical user interface

## 6.5. WINDOWS NT

The CRT code prepares them as well.

Also, the number returned by the `main()` function is the exit code.

It may be passed in CRT to the `ExitProcess()` function, which takes the exit code as an argument.

Usually, each compiler has its own CRT code.

Here is a typical CRT code for MSVC 2008.

```
1  __tmainCRTStartup proc near
2
3  var_24 = dword ptr -24h
4  var_20 = dword ptr -20h
5  var_1C = dword ptr -1Ch
6  ms_exc = CPPEH_RECORD ptr -18h
7
8      push    14h
9      push    offset stru_4092D0
10     call    __SEH_prolog4
11     mov     eax, 5A4Dh
12     cmp     ds:400000h, ax
13     jnz    short loc_401096
14     mov     eax, ds:40003Ch
15     cmp     dword ptr [eax+400000h], 4550h
16     jnz    short loc_401096
17     mov     ecx, 10Bh
18     cmp     [eax+400018h], cx
19     jnz    short loc_401096
20     cmp     dword ptr [eax+400074h], 0Eh
21     jbe    short loc_401096
22     xor     ecx, ecx
23     cmp     [eax+4000E8h], ecx
24     setnz  cl
25     mov     [ebp+var_1C], ecx
26     jmp    short loc_40109A
27
28
29 loc_401096: ; CODE XREF: __tmainCRTStartup+18
30             ; __tmainCRTStartup+29 ...
31     and    [ebp+var_1C], 0
32
33 loc_40109A: ; CODE XREF: __tmainCRTStartup+50
34     push   1
35     call    _heap_init
36     pop    ecx
37     test   eax, eax
38     jnz    short loc_4010AE
39     push   1Ch
40     call    _fast_error_exit
41     pop    ecx
42
43 loc_4010AE: ; CODE XREF: __tmainCRTStartup+60
44     call    __mtinit
45     test   eax, eax
46     jnz    short loc_4010BF
47     push   10h
48     call    _fast_error_exit
49     pop    ecx
50
51 loc_4010BF: ; CODE XREF: __tmainCRTStartup+71
52     call    sub_401F2B
53     and    [ebp+ms_exc.disabled], 0
54     call    __ioinit
55     test   eax, eax
56     jge    short loc_4010D9
57     push   1Bh
58     call    __amsg_exit
59     pop    ecx
60
```

## 6.5. WINDOWS NT

```
61 loc_4010D9: ; CODE XREF: __tmainCRTStartup+8B
62     call    ds:GetCommandLineA
63     mov     dword_40B7F8, eax
64     call    __crtGetEnvironmentStringsA
65     mov     dword_40AC60, eax
66     call    __setargv
67     test   eax, eax
68     jge    short loc_4010FF
69     push   8
70     call    __amsg_exit
71     pop    ecx
72
73 loc_4010FF: ; CODE XREF: __tmainCRTStartup+B1
74     call    __setenvp
75     test   eax, eax
76     jge    short loc_401110
77     push   9
78     call    __amsg_exit
79     pop    ecx
80
81 loc_401110: ; CODE XREF: __tmainCRTStartup+C2
82     push   1
83     call    __cinit
84     pop    ecx
85     test   eax, eax
86     jz    short loc_401123
87     push   eax
88     call    __amsg_exit
89     pop    ecx
90
91 loc_401123: ; CODE XREF: __tmainCRTStartup+D6
92     mov    eax, envp
93     mov    dword_40AC80, eax
94     push  eax          ; envp
95     push  argv          ; argv
96     push  argc          ; argc
97     call  _main
98     add   esp, 0Ch
99     mov    [ebp+var_20], eax
100    cmp   [ebp+var_1C], 0
101    jnz  short $LN28
102    push  eax          ; uExitCode
103    call  $LN32
104
105 $LN28:    ; CODE XREF: __tmainCRTStartup+105
106    call  __cexit
107    jmp   short loc_401186
108
109
110 $LN27:    ; DATA XREF: .rdata:stru_4092D0
111    mov    eax, [ebp+ms_exc.exc_ptr] ; Exception filter 0 for function 401044
112    mov    ecx, [eax]
113    mov    ecx, [ecx]
114    mov    [ebp+var_24], ecx
115    push  eax
116    push  ecx
117    call  __XcptFilter
118    pop   ecx
119    pop   ecx
120
121 $LN24:
122     retn
123
124
125 $LN14:    ; DATA XREF: .rdata:stru_4092D0
126    mov    esp, [ebp+ms_exc.old_esp] ; Exception handler 0 for function 401044
127    mov    eax, [ebp+var_24]
128    mov    [ebp+var_20], eax
129    cmp   [ebp+var_1C], 0
130    jnz  short $LN29
```

## 6.5. WINDOWS NT

```
131     push    eax          ; int
132     call    __exit
133
134
135 $LN29:   ; CODE XREF: __tmainCRTStartup+135
136     call    __c_exit
137
138 loc_401186: ; CODE XREF: __tmainCRTStartup+112
139     mov     [ebp+ms_exc.disabled], 0FFFFFFFEh
140     mov     eax, [ebp+var_20]
141     call    __SEH_epilog4
142     retn
```

Here we can see calls to `GetCommandLineA()` (line 62), then to `setargv()` (line 66) and `setenvp()` (line 74), which apparently fill the global variables `argc`, `argv`, `envp`.

Finally, `main()` is called with these arguments (line 97).

There are also calls to functions with self-describing names like `heap_init()` (line 35), `ioinit()` (line 54).

The `heap` is indeed initialized in the [CRT](#). If you try to use `malloc()` in a program without CRT, it will exit abnormally with the following error:

```
runtime error R6030
- CRT not initialized
```

Global object initializations in C++ is also occur in the [CRT](#) before the execution of `main()`: [3.19.4 on page 567](#).

The value that `main()` returns is passed to `cexit()`, or in `$LN32`, which in turn calls `doexit()`.

Is it possible to get rid of the [CRT](#)? Yes, if you know what you are doing.

The [MSVC](#)'s linker has the `/ENTRY` option for setting an 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 are getting a runnable .exe with size 2560 bytes, that has a PE header in it, instructions calling `MessageBox`, two strings in the data segment, the `MessageBox` function imported from `user32.dll` and nothing else.

This works, but you cannot write `WinMain` with its 4 arguments instead of `main()`.

To be precise, you can, but the arguments are not prepared at the moment of execution.

By the way, it is possible to make the .exe even shorter by aligning the PE sections at less than the default 4096 bytes.

```
cl no_crt.c user32.lib /link /entry:main /align:16
```

Linker says:

```
LINK : warning LNK4108: /ALIGN specified without /DRIVER; image may not run
```

We get an .exe that's 720 bytes. It can be executed in Windows 7 x86, but not in x64 (an error message will be shown when you try to execute it).

With even more efforts, it is possible to make the executable even shorter, but as you can see, compatibility problems arise quickly.

## 6.5.2 Win32 PE

PE is an executable file format used in Windows. The difference between .exe, .dll and .sys is that .exe and .sys usually do not have exports, only imports.

A [DLL<sup>14</sup>](#), just like any other PE-file, has an entry point ([OEP](#)) (the function `DllMain()` is located there) but this function usually does nothing. .sys is usually a device driver. As of drivers, Windows requires the checksum to be present in the PE file and for it to be correct [<sup>15</sup>](#).

Starting at Windows Vista, a driver's files must also be signed with a digital signature. It will fail to load otherwise.

Every PE file begins with tiny DOS program that prints a message like "This program cannot be run in DOS mode."—if you run this program in DOS or Windows 3.1 ([OS](#)-es which are not aware of the PE format), this message will be printed.

### Terminology

- Module—a separate file, .exe or .dll.
- Process—a program loaded into memory and currently running. Commonly consists of one .exe file and bunch of .dll files.
- Process memory—the memory a process works with. Each process has its own. There usually are loaded modules, memory of the stack, [heap\(s\)](#), etc.
- [VA<sup>16</sup>](#)—an address which is to be used in program while runtime.
- Base address (of module)—the address within the process memory at which the module is to be loaded. [OS](#) loader may change it, if the base address is already occupied by another module just loaded before.
- [RVA<sup>17</sup>](#)—the [VA](#)-address minus the base address.  
Many addresses in PE-file tables use [RVA](#)-addresses.
- [IAT<sup>18</sup>](#)—an array of addresses of imported symbols [<sup>19</sup>](#). Sometimes, the `IMAGE_DIRECTORY_ENTRY_IAT` data directory points at the [IAT](#). It is worth noting that [IDA](#) (as of 6.1) may allocate a pseudo-section named `.idata` for [IAT](#), even if the [IAT](#) is a part of another section!
- [INT<sup>20</sup>](#)—an array of names of symbols to be imported<sup>21</sup>.

### Base address

The problem is that several module authors can prepare DLL files for others to use and it is not possible to reach an agreement which addresses is to be assigned to whose modules.

So that is why if two necessary DLLs for a process have the same base address, one of them will be loaded at this base address, and the other—at some other free space in process memory, and each virtual addresses in the second DLL will be corrected.

With [MSVC](#) the linker often generates the .exe files with a base address of `0x400000` [<sup>22</sup>](#), and with the code section starting at `0x401000`. This means that the [RVA](#) of the start of the code section is `0x1000`.

DLLs are often generated by MSVC's linker with a base address of `0x10000000` [<sup>23</sup>](#).

There is also another reason to load modules at various base addresses, in this case random ones. It is [ASLR<sup>24</sup>](#).

<sup>14</sup>Dynamic-link library

<sup>15</sup>For example, Hiew ([7.1 on page 787](#)) can calculate it

<sup>16</sup>Virtual Address

<sup>17</sup>Relative Virtual Address

<sup>18</sup>Import Address Table

<sup>19</sup>Matt Pietrek, *An In-Depth Look into the Win32 Portable Executable File Format*, (2002)]

<sup>20</sup>Import Name Table

<sup>21</sup>Matt Pietrek, *An In-Depth Look into the Win32 Portable Executable File Format*, (2002)]

<sup>22</sup>The origin of this address choice is described here: [MSDN](#)

<sup>23</sup>This can be changed by the /BASE linker option

<sup>24</sup>wikipedia

## 6.5. WINDOWS NT

A shellcode trying to get executed on a compromised system must call system functions, hence, know their addresses.

In older OS (in Windows NT line: before Windows Vista), system DLL (like kernel32.dll, user32.dll) were always loaded at known addresses, and if we also recall that their versions rarely changed, the addresses of functions were fixed and shellcode could call them directly.

In order to avoid this, the ASLR method loads your program and all modules it needs at random base addresses, different every time.

ASLR support is denoted in a PE file by setting the flag

`IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE` [see Mark Russinovich, *Microsoft Windows Internals*].

## Subsystem

There is also a *Subsystem* field, usually it is:

- native<sup>25</sup> (.sys-driver),
- console (console application) or
- GUI (non-console).

## OS version

A PE file also specifies the minimal Windows version it needs in order to be loadable.

The table of version numbers stored in the PE file and corresponding Windows codenames is here<sup>26</sup>.

For example, MSVC 2005 compiles .exe files for running on Windows NT4 (version 4.00), but MSVC 2008 does not (the generated files have a version of 5.00, at least Windows 2000 is needed to run them).

MSVC 2012 generates .exe files of version 6.00 by default, targeting at least Windows Vista. However, by changing the compiler's options<sup>27</sup>, it is possible to force it to compile for Windows XP.

## Sections

Division in sections, as it seems, is present in all executable file formats.

It is devised in order to separate code from data, and data—from constant data.

- Either the `IMAGE_SCN_CNT_CODE` or `IMAGE_SCN_MEM_EXECUTE` flags will be set on the 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`.
- On a constant data section (one that's protected from writing), the flags `IMAGE_SCN_CNT_INITIALIZED_DATA` and `IMAGE_SCN_MEM_READ` can be set, but not `IMAGE_SCN_MEM_WRITE`. A process going to crash if it tries to write to this section.

Each section in PE-file may have a name, however, it is not very important. Often (but not always) the code section is named `.text`, the data section—`.data`, the constant data section — `.rdata` (readable data). Other popular section names are:

- `.idata` —imports section. IDA may create a pseudo-section named like this: [6.5.2 on the preceding page](#).
- `.edata` —exports section (rare)

<sup>25</sup>Meaning, the module use Native API instead of Win32

<sup>26</sup>wikipedia

<sup>27</sup>MSDN

## 6.5. WINDOWS NT

- `.pdata` — section holding all information about exceptions in Windows NT for MIPS, IA64 and x64: [6.5.3 on page 781](#)
- `.reloc` — relocs section
- `.bss` — uninitialized data ([BSS](#))
- `.tls` — thread local storage ([TLS](#))
- `.rsrc` — resources
- `.CRT` — may present in binary files compiled by ancient MSVC versions

PE file packers/encryptors often garble section names or replace the names with their own.

MSVC allows you to declare data in arbitrarily named section <sup>28</sup>.

Some compilers and linkers can add a section with debugging symbols and other debugging information (MinGW for instance). However it is not so in latest versions of MSVC (separate PDB files are used there for this purpose).

That is how a PE section is 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;  
    WORD NumberOfLinenumbers;  
    DWORD Characteristics;  
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

<sup>29</sup>

A word about terminology: *PointerToRawData* is called “Offset” in Hiew and *VirtualAddress* is called “RVA” there.

## Data section

Data section in file can be smaller than in memory. For example, some variables can be initialized, some are not. Compiler and linker will collect them all into one section, but the first part of it is initialized and allocated in file, while another is absent in file (of course, to make it smaller). *VirtualSize* will be equal to the size of section in memory, and *SizeOfRawData* — to size of section in file.

IDA can show the border between initialized and not initialized parts like that:

```
...  
.data:10017FFA db 0  
.data:10017FFB db 0  
.data:10017FFC db 0  
.data:10017FFD db 0  
.data:10017FFE db 0  
.data:10017FFF db 0  
.data:10018000 db ? ;  
.data:10018001 db ? ;  
.data:10018002 db ? ;  
.data:10018003 db ? ;  
.data:10018004 db ? ;  
.data:10018005 db ? ;
```

<sup>28</sup>[MSDN](#)

<sup>29</sup>[MSDN](#)

## Relocations (relocs)

AKA FIXUP-s (at least in Hiew).

They are also present in almost all executable file formats <sup>30</sup>. Exceptions are shared dynamic libraries compiled with PIC, or any other PIC-code.

What are they for?

Obviously, modules can be loaded on various base addresses, but how to deal with global variables, for example? They must be accessed by address. One solution is position-independent code ([6.4.1 on page 746](#)). But it is not always convenient.

That is why a relocations table is present. There the addresses of points that must be corrected are enumerated, in case of loading at a different base address.

For example, there is a global variable at address `0x410000` and this is how it is accessed:

<code>A1 00 00 41 00</code>	<code>mov</code>	<code>eax, [00041000]</code>
-----------------------------	------------------	------------------------------

The base address of the module is `0x400000`, the RVA of the global variable is `0x10000`.

If the module is loaded at base address `0x500000`, the real address of the global variable must be `0x510000`.

As we can see, the address of variable is encoded in the instruction `MOV`, after the byte `0xA1`.

That is why the address of the 4 bytes after `0xA1`, is written in the relocs table.

If the module is loaded at a different base address, the OS loader enumerates all addresses in the table, finds each 32-bit word the address points to, subtracts the original base address from it (we get the RVA here), and adds the new base address to it.

If a module is loaded at its original base address, nothing happens.

All global variables can 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.1.21](#).

OllyDbg underlines the places in memory to which relocs are to be applied, for example: [fig.1.52](#).

## Exports and imports

As we all know, any executable program must use the OS's services and other DLL-libraries somehow.

It can be said that functions from one module (usually DLL) must be connected somehow to the points of their calls in other modules (.exe-file or another DLL).

For this, each DLL has an "exports" table, which consists of functions plus their addresses in a module.

And every .exe file or DLL has "imports", a table of functions it needs for execution including list of DLL filenames.

After loading the main .exe-file, the OS loader processes imports table: it loads the additional DLL-files, finds function names among the DLL exports and writes their addresses down in the IAT of the main .exe-module.

As we can see, during loading the loader must compare a lot of function names, but string comparison is not a very fast procedure, so there is a support for "ordinals" or "hints", which are function numbers stored in the table, instead of their names.

That is how they can be located faster when loading a DLL. Ordinals are always present in the "export" table.

---

<sup>30</sup>Even in .exe files for MS-DOS

## 6.5. WINDOWS NT

For example, a program using the [MFC<sup>31</sup>](#) library usually loads mfc\*.dll by ordinals, and in such programs there are no MFC function names in INT.

When loading such programs in [IDA](#), it will ask for a path to the mfc\*.dll files in order to determine the function names.

If you don't tell [IDA](#) the path to these DLLs, there will be *mfc80\_123* instead of function names.

### Imports section

Often a separate section is allocated for the imports table and everything related to it (with name like `.idata`), however, this is not a strict rule.

Imports are also a confusing subject because of the terminological mess. Let's try to collect all information in one place.

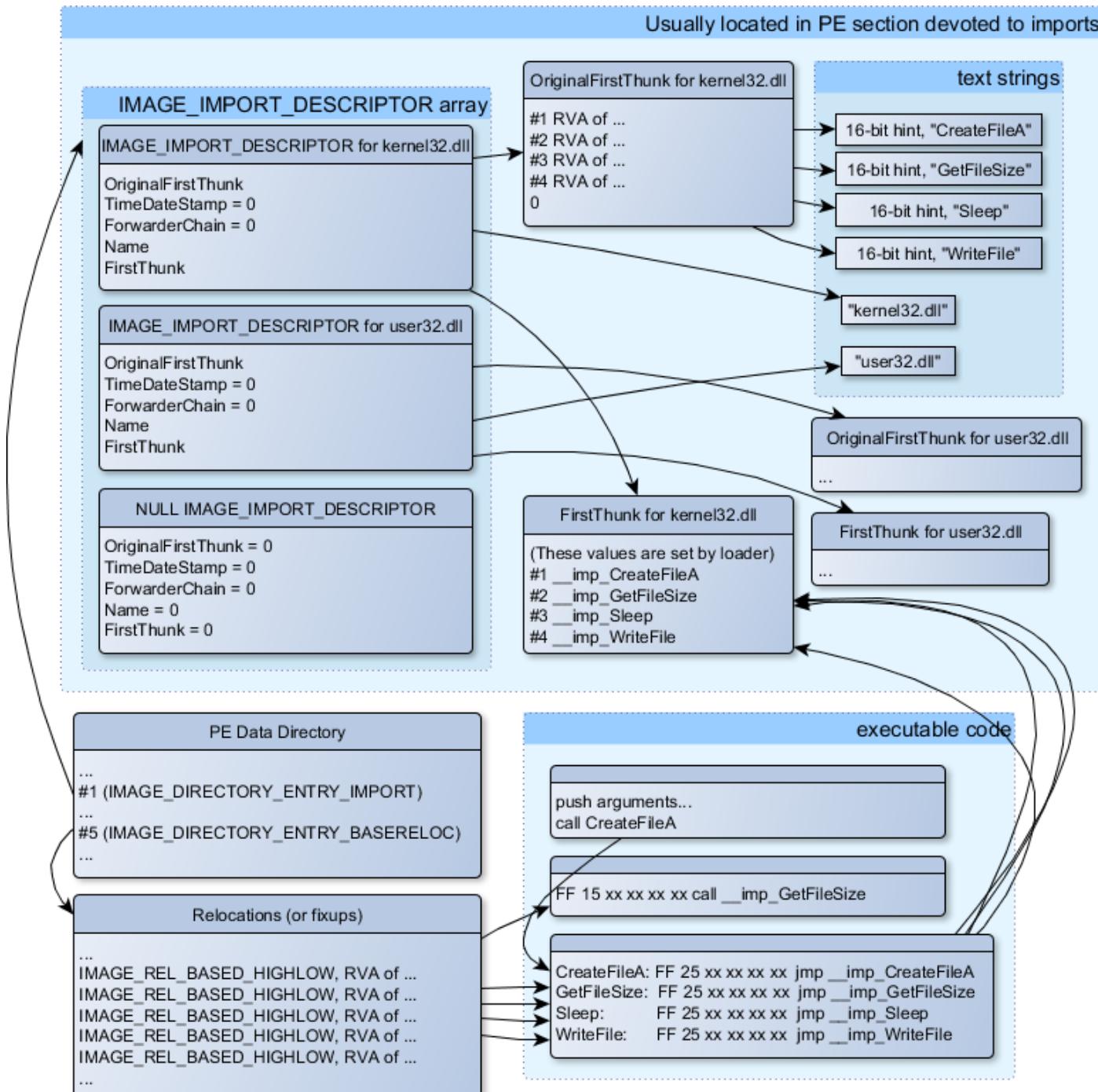


Figure 6.1: A scheme that unites all PE-file structures related to imports

<sup>31</sup>Microsoft Foundation Classes

## 6.5. WINDOWS NT

The main structure is the array *IMAGE\_IMPORT\_DESCRIPTOR*. Each element for each DLL being imported.

Each element holds the **RVA** address of the text string (DLL name) (*Name*).

*OriginalFirstThunk* is the **RVA** address of the **INT** table. This is an array of **RVA** addresses, each of which points to a text string with a function name. Each string is prefixed by a 16-bit integer (“hint”)—“ordinal” of function.

While loading, if it is possible to find a function by ordinal, then the strings comparison will not occur. The array is terminated by zero.

There is also a pointer to the **IAT** table named *FirstThunk*, it is just the **RVA** address of the place where the loader writes the addresses of the resolved functions.

The points where the loader writes addresses are marked by **IDA** like this: *\_imp\_CreateFileA*, etc.

There are at least two ways to use the addresses written by the loader.

- The code will have instructions like *call \_imp\_CreateFileA*, and since the field with the address of the imported function is a global variable in some sense, the address of the *call* instruction (plus 1 or 2) is to be added to the relocs table, for the case when the module is loaded at a 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 slows down the process of loading modules.

- For each imported function, there is only one jump allocated, using the **JMP** instruction plus a reloc to it. 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 have relative addresses and do not need to be corrected.

These two methods can be combined.

Possible, the linker creates individual “thunk”s if there are too many calls to the function, but not done by default.

By the way, the array of function addresses to which *FirstThunk* is pointing is not necessary to be located in the **IAT** section. For example, the author of these lines once wrote the **PE\_add\_import**<sup>32</sup> utility for adding imports 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 with a call to another DLL, my utility wrote the following code:

```
MOV EAX, [yourdll.dll!function]
JMP EAX
```

*FirstThunk* points to the first instruction. In other words, when loading *yourdll.dll*, the loader writes the address of the *function* function right in the code.

It also worth noting that a code section is usually write-protected, so my utility adds the *IMAGE\_SCN\_MEM\_WRITE* flag for code section. Otherwise, the program to crash while loading with error code 5 (access denied).

One might ask: what if I supply a program with a set of DLL files which is not supposed to change (including addresses of all DLL functions), is it possible to speed up the loading process?

Yes, it is possible to write the addresses of the functions to be imported into the *FirstThunk* arrays in advance. The *Timestamp* field is present in the *IMAGE\_IMPORT\_DESCRIPTOR* structure.

If a value is present there, then the loader compares this value with the date-time of the DLL file.

If the values are equal, then the loader does not do anything, and the loading of the process can be faster. This is called “old-style binding”<sup>33</sup>.

<sup>32</sup>[yurichev.com](http://yurichev.com)

<sup>33</sup>[MSDN](#). There is also the “new-style binding”.

## 6.5. WINDOWS NT

The BIND.EXE utility in Windows SDK is for for this. For speeding up the loading of your program, Matt Pietrek in Matt Pietrek, *An In-Depth Look into the Win32 Portable Executable File Format*, (2002)]<sup>34</sup>, suggests to do the binding shortly after your program installation on the computer of the end user.

PE-files packers/encryptors may also compress/encrypt imports table.

In this case, the Windows loader, of course, will not load all necessary DLLs.

Therefore, the packer/encryptor does this on its own, with the help of *LoadLibrary()* and the *GetProcAddress()* functions.

That is why these two functions are often present in **IAT** in packed files.

In the standard DLLs from the Windows installation, **IAT** often is located right at the beginning of the PE file. Supposedly, it is made so for optimization.

While loading, the .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 in parts as they are accessed.

Probably, Microsoft developers decided it will be faster.

## Resources

Resources in a PE file are 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 the **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, by using the ResHack editor, for example ([6.5.2](#)).

## .NET

.NET programs are not compiled into machine code but into a special bytecode. Strictly speaking, there is bytecode instead of the usual x86 code in the .exe file, however, the entry point (**OEP**) points to this tiny fragment of x86 code:

```
jmp mscoree.dll!_CorExeMain
```

The .NET loader is located in mscoree.dll, which processes the PE file.

It was so in all pre-Windows XP **OSes**. Starting from XP, the **OS** loader is able to detect the .NET file and run it without executing that **JMP** instruction <sup>35</sup>.

## TLS

This section holds initialized data for the **TLS** ([6.2 on page 740](#)) (if needed). When a new thread start, its **TLS** data is initialized using the data from this section.

Aside from that, the PE file specification also provides initialization of the **TLS** section, the so-called TLS callbacks.

If they are present, they are to be called before the control is passed to the main entry point (**OEP**).

This is used widely in the PE file packers/encryptors.

<sup>34</sup>Also available as <http://go.yurichev.com/17318>

<sup>35</sup>MSDN

---

**Tools**

- objdump (present in cygwin) for dumping all PE-file structures.
- Hiew ([7.1 on page 787](#)) as editor.
- pefile—Python-library for PE-file processing <sup>[36](#)</sup>.
- ResHack AKA Resource Hacker—resources editor<sup>[37](#)</sup>.
- PE\_add\_import<sup>[38](#)</sup>— simple tool for adding symbol(s) to PE executable import table.
- PE\_patcher<sup>[39](#)</sup>—simple tool for patching PE executables.
- PE\_search\_str\_refs<sup>[40](#)</sup>—simple tool for searching for a function in PE executables which use some text string.

**Further reading**

- Daniel Pistelli—The .NET File Format <sup>[41](#)</sup>

**6.5.3 Windows SEH****Let's forget about MSVC**

In Windows, the [SEH](#) is intended for exceptions handling, nevertheless, it is language-agnostic, not related to C++ or [OOP](#) in any way.

Here we are going to take a look at [SEH](#) in its isolated (from C++ and MSVC extensions) form.

Each running process has a chain of [SEH](#) handlers, [TIB](#) has the address of the last handler.

When an exception occurs (division by zero, incorrect address access, user exception triggered by calling the `RaiseException()` function), the [OS](#) finds the last handler in the [TIB](#) and calls it, passing all information about the [CPU](#) state (register values, etc.) at the moment of the exception.

The exception handler considering the exception, does it see something familiar? If so, it handles the exception.

If not, it signals to the [OS](#) that it cannot handle it and the [OS](#) calls the next handler in the chain, until a handler which is able to handle the exception is found.

At the very end of the chain there a standard handler that shows the well-known dialog box, informing the user about a process crash, some technical information about the [CPU](#) state at the time of the crash, and offering to collect all information and send it to developers in Microsoft.

---

<sup>36</sup><http://go.yurichev.com/17052>

<sup>37</sup><http://go.yurichev.com/17052>

<sup>38</sup><http://go.yurichev.com/17049>

<sup>39</sup>[yurichev.com](http://yurichev.com)

<sup>40</sup>[yurichev.com](http://yurichev.com)

<sup>41</sup><http://go.yurichev.com/17056>

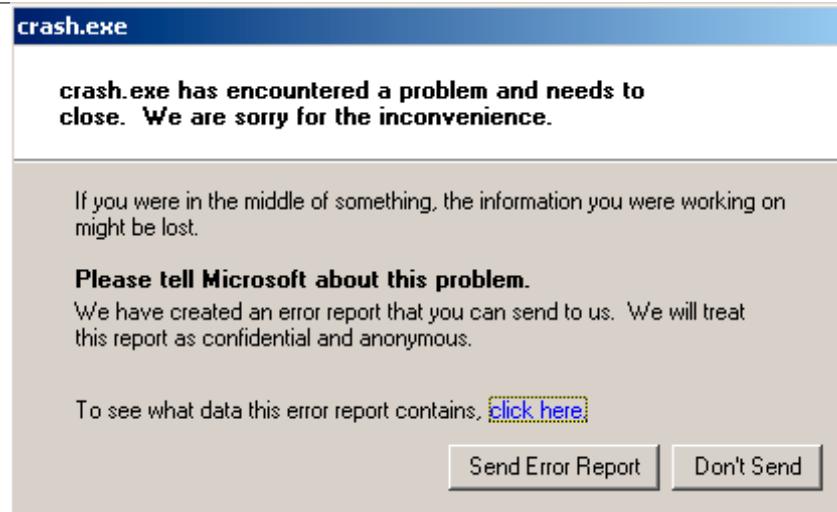


Figure 6.2: Windows XP

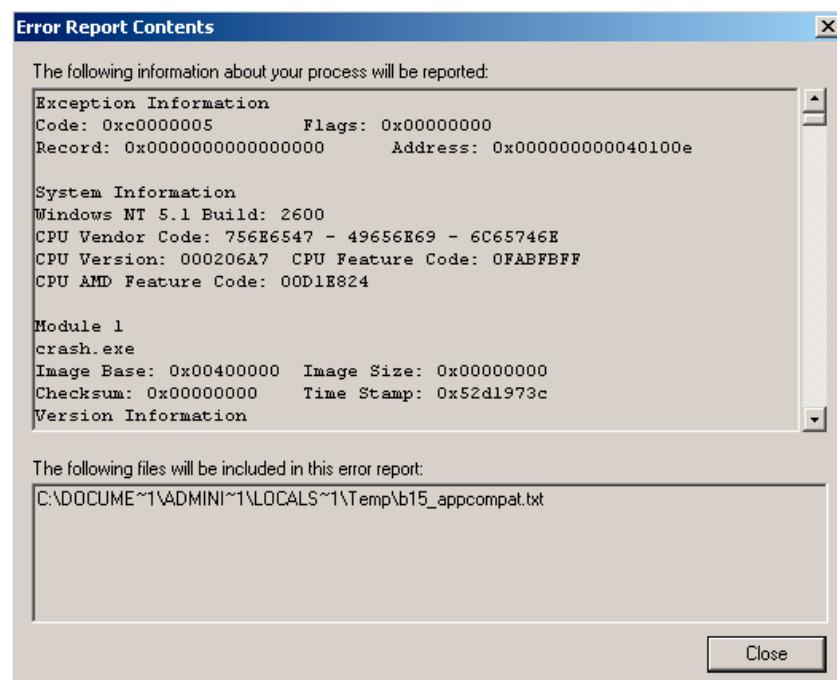


Figure 6.3: Windows XP

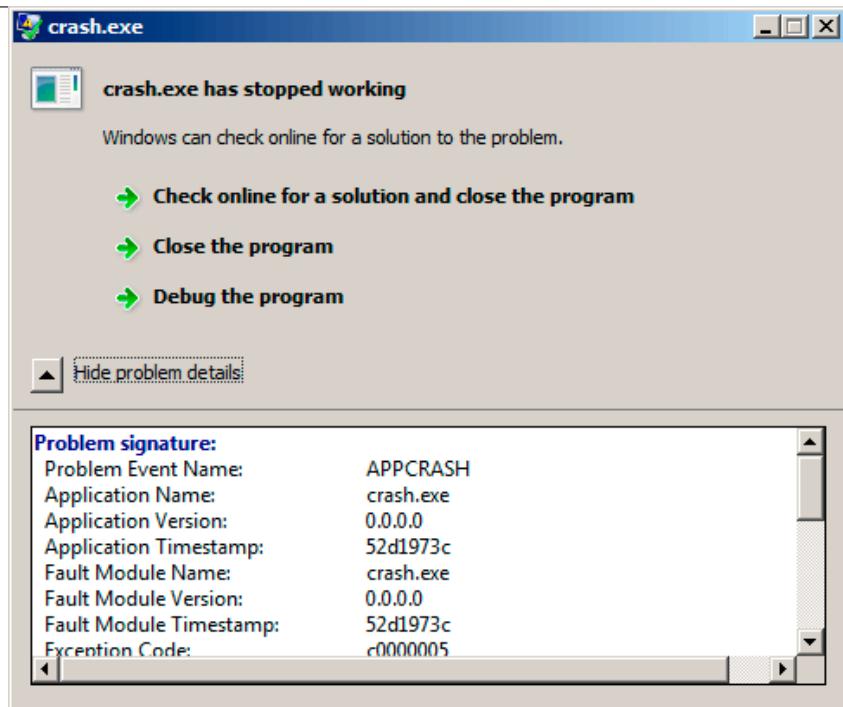


Figure 6.4: Windows 7

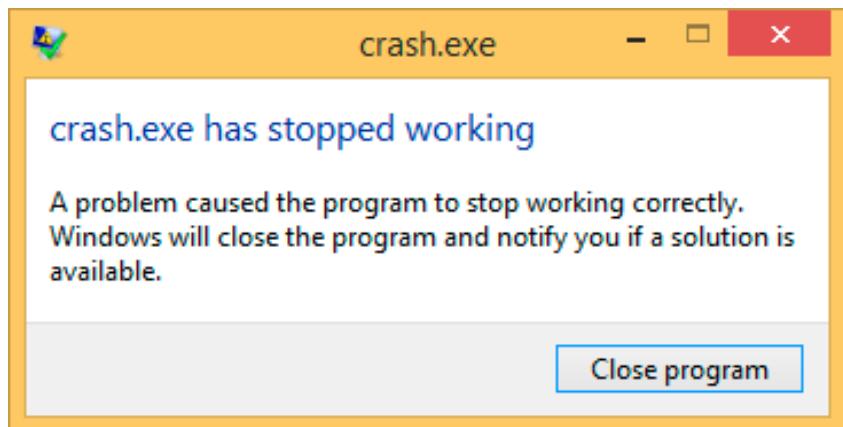


Figure 6.5: Windows 8.1

Earlier, this handler was called Dr. Watson <sup>42</sup>.

By the way, some developers make their own handler that sends information about the program crash to themselves. It is registered with the help of `SetUnhandledExceptionFilter()` and to be called if the OS does not have any other way to handle the exception. An example is Oracle RDBMS—it saves huge dumps reporting all possible information about the CPU and memory state.

Let's write our own primitive exception handler. This example is based on the example from [Matt Pietrek, *A Crash Course on the Depths of Win32™ Structured Exception Handling*, (1997)]<sup>43</sup>. It must be compiled with the SAFESEH option: `cl seh1.cpp /link /safe seh: no`. More about SAFESEH here: [MSDN](#).

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

DWORD new_value=1234;

EXCEPTION_DISPOSITION __cdecl except_handler(
    struct _EXCEPTION_RECORD *ExceptionRecord,
    void * EstablisherFrame,
    struct _CONTEXT *ContextRecord,
```

<sup>42</sup>[wikipedia](#)

<sup>43</sup>Also available as <http://go.yurichev.com/17293>

## 6.5. WINDOWS NT

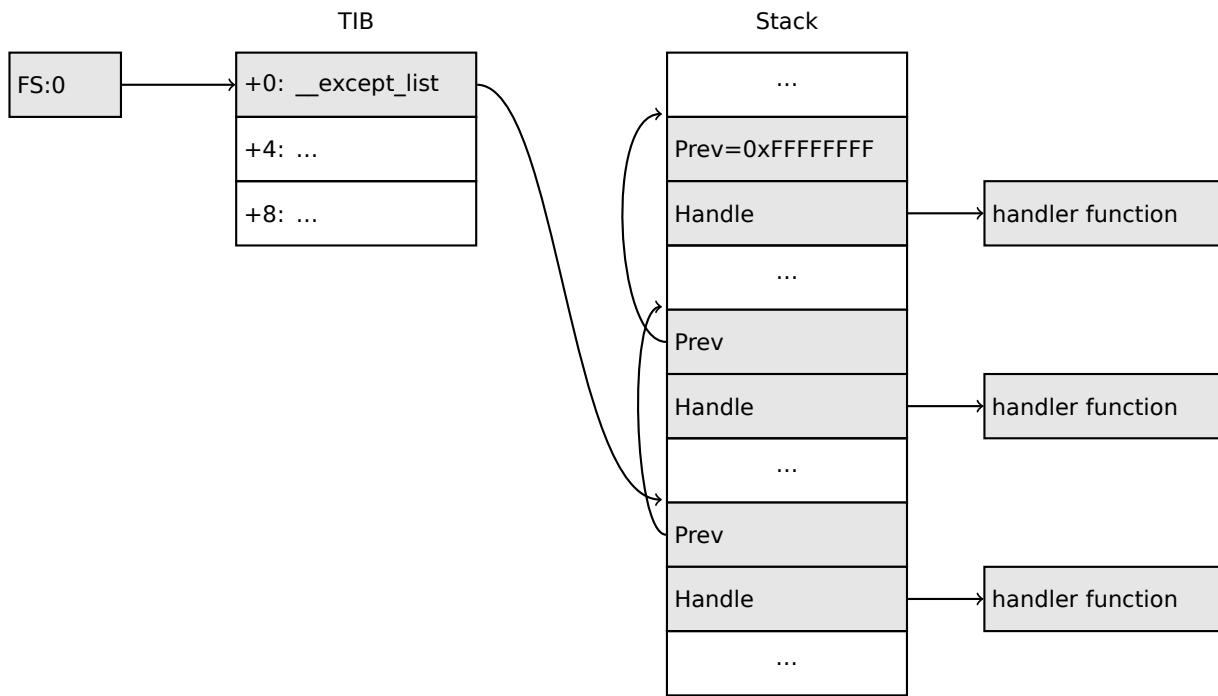
```
        void * DispatcherContext )  
{  
    unsigned i;  
  
    printf ("%s\n", __FUNCTION__);  
    printf ("ExceptionRecord->ExceptionCode=0x%p\n", ExceptionRecord->ExceptionCode);  
    printf ("ExceptionRecord->ExceptionFlags=0x%p\n", ExceptionRecord->ExceptionFlags);  
    printf ("ExceptionRecord->ExceptionAddress=0x%p\n", 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");  
        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  
    {  
        push    handler           // make EXCEPTION_REGISTRATION record:  
        push    FS:[0]            // address of handler function  
        mov     FS:[0],ESP         // address of previous handler  
    }  
  
    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  
    {  
        mov    eax,[ESP]          // remove our EXCEPTION_REGISTRATION record  
        mov    FS:[0], EAX          // get pointer to previous record  
        add    esp, 8              // install previous record  
        add    esp, 8              // clean our EXCEPTION_REGISTRATION off stack  
    }  
  
    return 0;  
}
```

The FS: segment register is pointing to the [TIB](#) in win32.

The very first element in the [TIB](#) is a pointer to the last handler in the chain. We save it in the stack and store the address of our handler there. The structure is named `_EXCEPTION_REGISTRATION`, it is a simple singly-linked list and its elements are stored right in the stack.

```
\_EXCEPTION\REGISTRATION struc
    prev    dd      ?
    handler dd      ?
\_EXCEPTION\REGISTRATION ends
```

So each “handler” field points to a handler and an each “prev” field points to the previous record in the stack. The last record has `0xFFFFFFFF` (-1) in the “prev” field.



After our handler is installed, we call `RaiseException()`<sup>44</sup>. This is an user exception. The handler checks the code. If the code is `0xE1223344`, it returning `ExceptionContinueExecution`, which means that handler corrected the CPU state (it is usually a correction of the EIP/ESP registers) and the OS can resume the execution of the. If you alter slightly the code so the handler returns `ExceptionContinueSearch`,

then the OS will call the other handlers, and it's unlikely that one who can handle it will be found, since no one will have any information about it (rather about its code). You will see the standard Windows dialog about a process crash.

What is the difference between a system exceptions and a user one? Here are the system ones:

<sup>44</sup>MSDN

## 6.5. WINDOWS NT

as defined in WinBase.h	as defined in ntstatus.h	value
EXCEPTION_ACCESS_VIOLATION	STATUS_ACCESS_VIOLATION	0xC0000005
EXCEPTION_DATATYPE_MISALIGNMENT	STATUS_DATATYPE_MISALIGNMENT	0x80000002
EXCEPTION_BREAKPOINT	STATUS_BREAKPOINT	0x80000003
EXCEPTION_SINGLE_STEP	STATUS_SINGLE_STEP	0x80000004
EXCEPTION_ARRAY_BOUNDS_EXCEEDED	STATUS_ARRAY_BOUNDS_EXCEEDED	0xC000008C
EXCEPTION_FLT_DENORMAL_OPERAND	STATUS_FLOAT_DENORMAL_OPERAND	0xC000008D
EXCEPTION_FLT_DIVIDE_BY_ZERO	STATUS_FLOAT_DIVIDE_BY_ZERO	0xC000008E
EXCEPTION_FLT_INEXACT_RESULT	STATUS_FLOAT_INEXACT_RESULT	0xC000008F
EXCEPTION_FLT_INVALID_OPERATION	STATUS_FLOAT_INVALID_OPERATION	0xC0000090
EXCEPTION_FLT_OVERFLOW	STATUS_FLOAT_OVERFLOW	0xC0000091
EXCEPTION_FLT_STACK_CHECK	STATUS_FLOAT_STACK_CHECK	0xC0000092
EXCEPTION_FLT_UNDERFLOW	STATUS_FLOAT_UNDERFLOW	0xC0000093
EXCEPTION_INT_DIVIDE_BY_ZERO	STATUS_INTEGER_DIVIDE_BY_ZERO	0xC0000094
EXCEPTION_INT_OVERFLOW	STATUS_INTEGER_OVERFLOW	0xC0000095
EXCEPTION_PRIV_INSTRUCTION	STATUS_PRIVILEGED_INSTRUCTION	0xC0000096
EXCEPTION_IN_PAGE_ERROR	STATUS_IN_PAGE_ERROR	0xC0000006
EXCEPTION_ILLEGAL_INSTRUCTION	STATUS_ILLEGAL_INSTRUCTION	0xC000001D
EXCEPTION_NONCONTINUABLE_EXCEPTION	STATUS_NONCONTINUABLE_EXCEPTION	0xC0000025
EXCEPTION_STACK_OVERFLOW	STATUS_STACK_OVERFLOW	0xC00000FD
EXCEPTION_INVALID_DISPOSITION	STATUS_INVALID_DISPOSITION	0xC0000026
EXCEPTION_GUARD_PAGE	STATUS_GUARD_PAGE_VIOLATION	0x80000001
EXCEPTION_INVALID_HANDLE	STATUS_INVALID_HANDLE	0xC0000008
EXCEPTION_POSSIBLE_DEADLOCK	STATUS_POSSIBLE_DEADLOCK	0xC0000194
CONTROL_C_EXIT	STATUS_CONTROL_C_EXIT	0xC000013A

That is how the code is defined:

31	29	28	27	16	15	0
S	U	0	Facility code		Error code	

S is a basic status code: 11—error; 10—warning; 01—informational; 00—success. U—whether the code is user code.

That is why we chose 0xE1223344—E<sub>16</sub> (1110<sub>2</sub>) 0xE (1110b) means that it is 1) user exception; 2) error. But to be honest, this example works fine without these high bits.

Then we try to read a value from memory at address 0.

Of course, there is nothing at this address in win32, so an exception is raised.

The very first handler is to be called—yours, and it will know about it first, by checking the code if it's equal to the EXCEPTION\_ACCESS\_VIOLATION constant.

The code that's reading from memory at address 0 looks like this:

Listing 6.23: 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 this error “on the fly” and to continue with program execution?

Yes, our exception handler can fix the EAX value and let the OS execute this instruction once again. So that is what we do. printf() prints 1234, because after the execution of our handler EAX is not 0, but contains the address of the global variable new\_value. The execution will resume.

That is what is going on: the memory manager in the CPU signals about an error, the CPU suspends the thread, finds the exception handler in the Windows kernel, which, in turn, starts to call all handlers in the SEH chain, one by one.

We use MSVC 2010 here, but of course, there is no any guarantee that EAX will be used for this pointer.

This address replacement trick is showy, and we considering it here as an illustration of SEH's internals. Nevertheless, it's hard to recall any case where it is used for “on-the-fly” error fixing.

## 6.5. WINDOWS NT

Why SEH-related records are stored right in the stack instead of some other place?

Supposedly because the OS is not needing to care about freeing this information, these records are simply disposed when the function finishes its execution. This is somewhat like alloca(): ([1.7.2 on page 35](#)).

### Now let's get back to MSVC

Supposedly, Microsoft programmers needed exceptions in C, but not in C++, so they added a non-standard C extension to MSVC<sup>45</sup>. It is not related to C++ PL exceptions.

```
_try
{
    ...
}
_except(filter code)
{
    handler code
}
```

"Finally" block may be instead of handler code:

```
_try
{
    ...
}
_finally
{
    ...
}
```

The filter code is an expression, telling whether this handler code corresponds to the exception raised.

If your code is too big and cannot fit into one expression, a separate filter function can be defined.

There are a lot of such constructs in the Windows kernel. Here are a couple of examples from there ([WRK](#)):

Listing 6.24: 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 6.25: 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 ) ) {
```

<sup>45</sup>[MSDN](#)

## 6.5. WINDOWS NT

Here is also a filter code example:

Listing 6.26: WRK-v1.2/base/ntos/cache/copysup.c

```
LONG
CcCopyReadExceptionFilter(
    IN PEXCEPTION_POINTERS ExceptionPointer,
    IN PNTSTATUS ErrorCode
)

/*++

Routine Description:

    This routine serves as an 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.

    ErrorCode - A pointer to an NTSTATUS that is to receive the real
                status.

Return Value:

    EXCEPTION_EXECUTE_HANDLER

--*/

{

    *ErrorCode = 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 the OS-supported exceptions. But the handler function is `_except_handler3` (for SEH3) or `_except_handler4` (for SEH4).

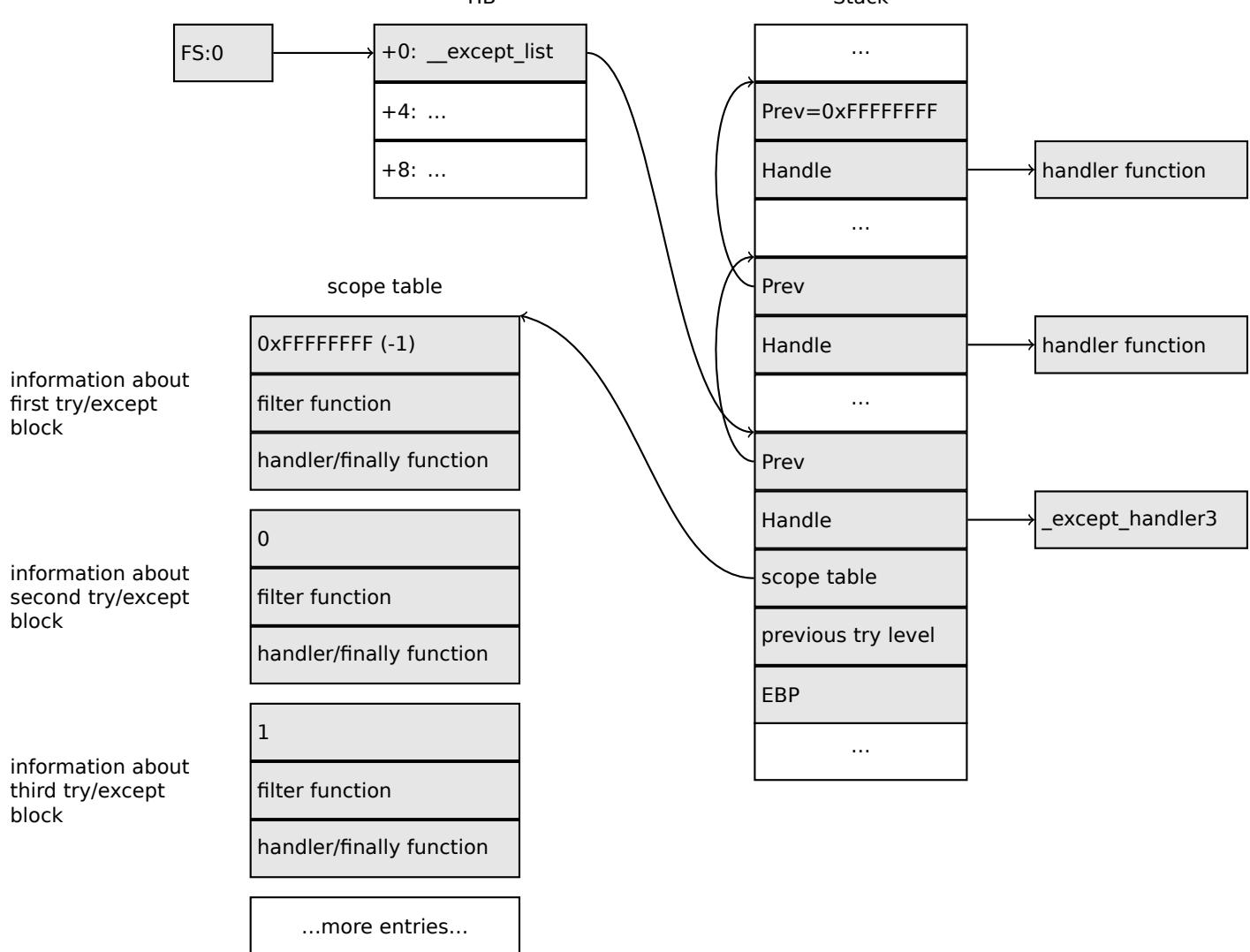
The code of this handler is MSVC-related, it is located in its libraries, or in `msvcr*.dll`. It is very important to know that SEH is a MSVC thing.

Other win32-compilers may offer something completely different.

### SEH3

SEH3 has `_except_handler3` as a handler function, and extends the `_EXCEPTION_REGISTRATION` table, adding a pointer to the *scope table* and *previous try level* variable. SEH4 extends the *scope table* by 4 values for buffer overflow protection.

The *scope table* is a table that consists of pointers to the filter and handler code blocks, for each nested level of *try/except*.



Again, it is very important to understand that the OS takes care only of the `prev/Handle` fields, and nothing more.

It is the job of the `_except_handler3` function to read the other fields and `scope table`, and decide which handler to execute and when.

The source code of the `_except_handler3` function is closed.

However, Sanos OS, which has a win32 compatibility layer, has the same functions reimplemented, which are somewhat equivalent to those in Windows<sup>46</sup>. Another reimplementation is present in Wine<sup>47</sup> and ReactOS<sup>48</sup>.

If the `filter` pointer is NULL, the `handler` pointer is the pointer to the `finally` code block.

During execution, the `previous try level` value in the stack changes, so `_except_handler3` can get information about the current level of nestedness, in order to know which `scope table` entry to use.

### SEH3: one try/except block example

```
#include <stdio.h>
#include <windows.h>
#include <excpt.h>

int main()
{
    int* p = NULL;
```

<sup>46</sup><http://go.yurichev.com/17058>

<sup>47</sup>[GitHub](#)

<sup>48</sup><http://go.yurichev.com/17060>

## 6.5. WINDOWS NT

```

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 6.27: 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
$T74622   DD      0xffffffffH      ; previous try level
            DD      FLAT:$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:$T74622        ; scope table
    push    OFFSET FLAT:_except_handler3 ; handler
    mov     eax, DWORD PTR fs:_except_list
    push    eax                      ; prev
    mov     DWORD PTR fs:_except_list, esp
    add    esp, -16
; 3 registers to be saved:
    push    ebx
    push    esi
    push    edi
    mov     DWORD PTR __$SEHRec$[ebp], esp
    mov     DWORD PTR _p$[ebp], 0
    mov     DWORD PTR __$SEHRec$[ebp+20], 0 ; 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

```

## 6.5. WINDOWS NT

```
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 the SEH frame is constructed in the stack. The *scope table* is located in the CONST segment—indeed, these fields are not to be changed. An interesting thing is how the *previous try level* variable has changed. The initial value is 0xFFFFFFFF (-1). The moment when the body of the try statement is opened is marked with an instruction that writes 0 to the variable. The moment when the body of the try statement is closed, -1 is written back to it. We also see the addresses of filter and handler code.

Thus we can easily see the structure of the try/except constructs in the function.

Since the SEH setup code in the function prologue may be shared between many functions, sometimes the compiler inserts a call to the SEH\_prolog() function in the prologue, which does just that.

The SEH cleanup code is in the SEH\_epilog() function.

Let's try to run this example in tracer:

```
tracer.exe -l:2.exe --dump-seh
```

Listing 6.28: 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=0xffffffff GSCookieXOROffset=0x0
                EHCookieOffset=0xffffffff EHCookieXOROffset=0x0
scopetable entry[0]. previous try level=-2, filter=0x771f74d0 (ntdll.dll! ↴
    ↴ __safe_se_handler_table+0x20) handler=0x771f90eb (ntdll.dll!_TppTerminateProcess@4+0x43)
* SEH frame at 0x18ffe4 prev=0xffffffff handler=0x77247428 (ntdll.dll!_FinalExceptionHandler@16 ↴
    ↴ )
```

## 6.5. WINDOWS NT

We see that the SEH chain consists of 4 handlers.

The first two are located in our example. Two? But we made only one? Yes, another one has been set up in the CRT function `_mainCRTStartup()`, and as it seems that it handles at least FPU exceptions. Its source code can be found in the MSVC installation: `crt/src/winxfltr.c`.

The third is the SEH4 one in ntdll.dll, and the fourth handler is not MSVC-related and is located in ntdll.dll, and has a 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 has two entries, one for each block. *Previous try level* changes as execution flow enters or exits the `try` block.

Listing 6.29: MSVC 2003

```
$SG74606 DB      'in filter. code=0x%08X', 0Ah, 00H
$SG74608 DB      'yes, that is our exception', 0Ah, 00H
```

## 6.5. WINDOWS NT

```
$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
$T74644  DD      0xffffffffH ; previous try level for outer block
            DD      FLAT:$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
_main    ENDP
```

## 6.5. WINDOWS NT

```
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
```

## 6.5. WINDOWS NT

```
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 we set a breakpoint on the `printf()` function, which is called from the handler, we can also see how yet another SEH handler is added.

Perhaps it's another machinery inside the 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 6.30: tracer.exe output

```
(0) 3.exe!printf
EAX=0x00000001b EBX=0x000000000 ECX=0x0040cc58 EDX=0x0008e3c8
ESI=0x000000000 EDI=0x000000000 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:   GSCookie0ffset=0xffffffff GSCookieXOROffset=0x0
               EHCookie0ffset=0xffffffffcc EHCookieXOROffset=0x0
scopetable entry[0]. previous try level=-2, filter=0x771f74d0 (ntdll.dll! ↴
  ↴ __safe_se_handler_table+0x20) handler=0x771f90eb (ntdll.dll!_TppTerminateProcess@4+0x43)
* SEH frame at 0x18ffe4 prev=0xffffffff handler=0x77247428 (ntdll.dll!_FinalExceptionHandler@16 ↴
  ↴ )
```

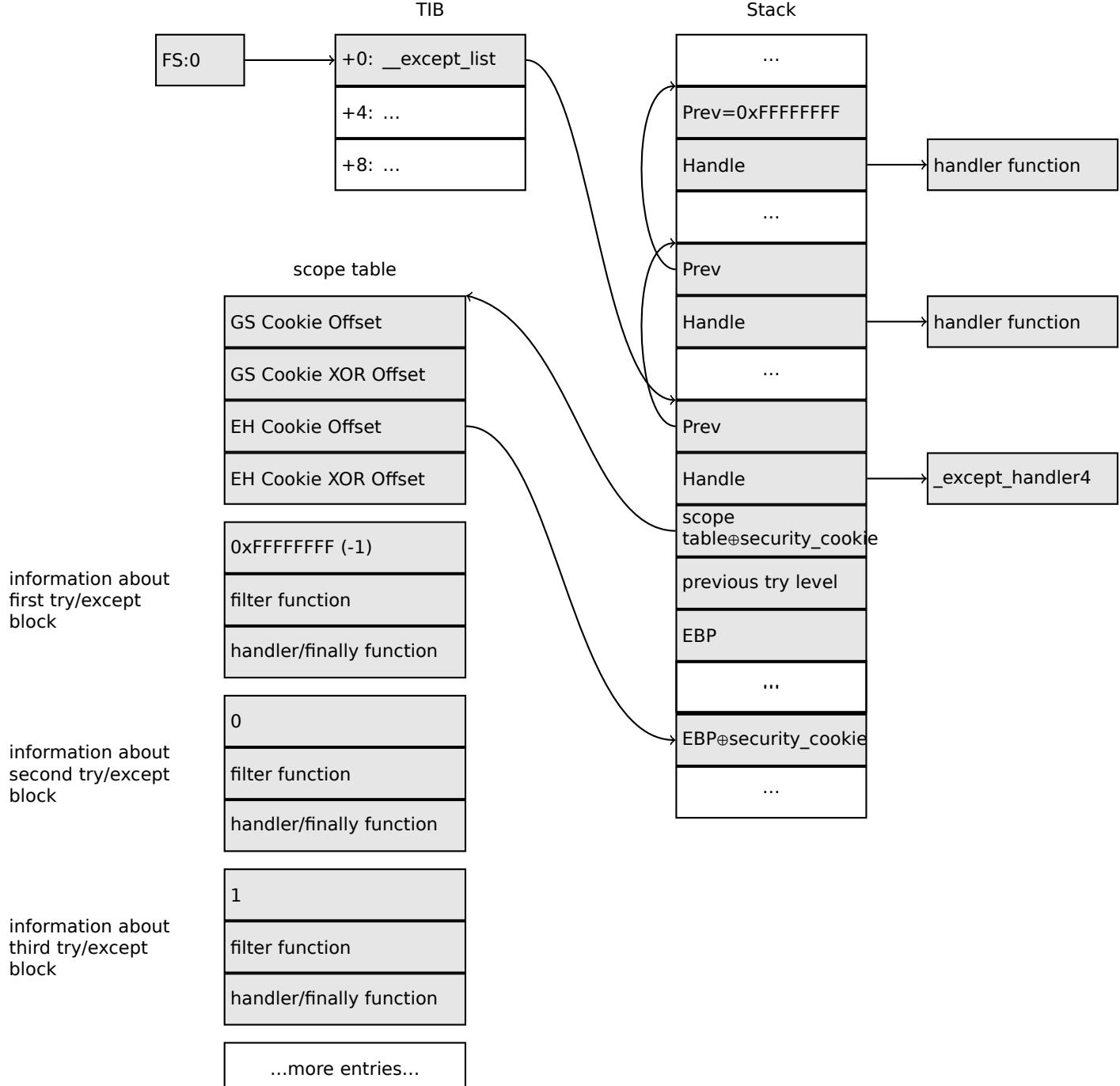
## SEH4

During a buffer overflow ([1.20.2 on page 275](#)) attack, the address of the *scope table* can be rewritten, so starting from MSVC 2005, SEH3 was upgraded to SEH4 in order to have buffer overflow protection. The pointer to the *scope table* is now *xored* with a *security cookie*. The *scope table* was extended to have a header consisting of two pointers to *security cookies*.

Each element has an offset inside the stack of another value: the address of the *stack frame* (EBP) *xored* with the *security\_cookie*, placed in the stack.

This value will be read during exception handling and checked for correctness. The *security cookie* in the stack is random each time, so hopefully a remote attacker can't predict it.

The initial *previous try level* is -2 in SEH4 instead of -1.



Here are both examples compiled in MSVC 2012 with SEH4:

Listing 6.31: 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 0xffffffffeH ; GS Cookie Offset
                   00H           ; GS Cookie XOR Offset
                   0xffffffffccH ; EH Cookie Offset
                   00H           ; EH Cookie XOR Offset
                   0xfffffffffeH ; previous try level
                   FLAT:$LN12@main ; filter
                   FLAT:$LN8@main  ; handler
xdata$x          ENDS

$T2 = -36        ; size = 4
_p$ = -32        ; size = 4
tv68 = -28       ; size = 4
__$SEHRec$ = -24 ; size = 24
```

## 6.5. WINDOWS NT

```
_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
```

## 6.5. WINDOWS NT

Listing 6.32: 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 0xfffffffffeH          ; GS Cookie Offset
                  DD 00H                 ; GS Cookie XOR Offset
                  DD 0xffffffffc8H        ; EH Cookie Offset
                  DD 00H                 ; EH Cookie Offset
                  DD 0xfffffffffeH        ; 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:
```

## 6.5. WINDOWS NT

```
$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
```

## 6.5. WINDOWS NT

```
_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 the meaning of the *cookies*: **Cookie Offset** is the difference between the address of the saved EBP value in the stack and the *EBP + security\_cookie* value in the stack. **Cookie XOR Offset** is an additional difference between the *EBP + security\_cookie* value and what is stored in the stack.

If this equation is not true, the process is to halt due to stack corruption:

$\text{security\_cookie} \oplus (\text{CookieXOROffset} + \text{address\_of\_saved\_EBP}) == \text{stack}[\text{address\_of\_saved\_EBP} + \text{CookieOffset}]$

If **Cookie Offset** is -2, this implies that it is not present.

Cookies checking is also implemented in my [tracer](#), see [GitHub](#) for details.

It is still possible to fall back to SEH3 in the compilers after (and including) MSVC 2005 by setting the **/GS-** option, however, the [CRT](#) code use SEH4 anyway.

## Windows x64

As you might think, it is not very fast to set up the SEH frame at each function prologue. Another performance problem is changing the *previous try level* value many times during the function's 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**, and from there the **OS**'s exception handler takes all the information.

Here are the two examples from the previous section compiled for x64:

Listing 6.33: 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
```

## 6.5. WINDOWS NT

```

xdata SEGMENT
$unwind$main DD 020609H
    DD 030023206H
    DD imagerel __C_specific_handler
    DD 01H
    DD imagerel $LN9+8
    DD imagerel $LN9+40
    DD imagerel main$filter$0
    DD imagerel $LN9+40
$unwind$main$filter$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 ; align next label
$LN8@main:
    xor  eax, eax
    add  rsp, 32
    pop  rbx
    ret  0
main ENDP
_TEXT ENDS

text$x SEGMENT
main$filter$0 PROC
    push rbp
    sub  rsp, 32
    mov  rbp, rdx
$LN5@main$filter$0:
    mov  rax, QWORD PTR [rcx]
    xor  ecx, ecx
    cmp  DWORD PTR [rax], -1073741819; c0000005H
    sete cl
    mov  eax, ecx
$LN7@main$filter$0:
    add  rsp, 32
    pop  rbp
    ret  0
    int  3
main$filter$0 ENDP
text$x ENDS

```

Listing 6.34: 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

```

## 6.5. WINDOWS NT

```
$pdata$main DD imagerel $LN14
    DD imagerel $LN14+95
    DD imagerel $unwind$main
pdata ENDS
pdata SEGMENT
$pdata$main$filter$0 DD imagerel main$filter$0
    DD imagerel main$filter$0+32
    DD imagerel $unwind$main$filter$0
$pdata$main$filter$1 DD imagerel main$filter$1
    DD imagerel main$filter$1+30
    DD imagerel $unwind$main$filter$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 $LN14+59
    DD imagerel main$filter$0
    DD imagerel $LN14+59
    DD imagerel $LN14+8
    DD imagerel $LN14+74
    DD imagerel main$filter$1
    DD imagerel $LN14+74
$unwind$main$filter$0 DD 020601H
    DD 050023206H
$unwind$main$filter$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 ; align next label
$LN13@main:
    jmp  SHORT $LN9@main
$LN7@main:
    lea   rcx, OFFSET FLAT:$SG86294 ; 'user exception caught'
    call printf
    npad 1 ; align next label
$LN9@main:
    xor  eax, eax
    add  rsp, 32
    pop  rbx
    ret  0
main ENDP

text$x SEGMENT
main$filter$0 PROC
```

## 6.5. WINDOWS NT

```
push    rbp
sub    rsp, 32
mov    rbp, rdx
$LN10@main$filter$:
    mov    rax, QWORD PTR [rcx]
    xor    ecx, ecx
    cmp    DWORD PTR [rax], -1073741819; c0000005H
    sete   cl
    mov    eax, ecx
$LN12@main$filter$:
    add    rsp, 32
    pop    rbp
    ret    0
    int    3
main$filter$0 ENDP

main$filter$1 PROC
    push   rbp
    sub    rsp, 32
    mov    rbp, rdx
$LN6@main$filter$:
    mov    rax, QWORD PTR [rcx]
    mov    rdx, rcx
    mov    ecx, DWORD PTR [rax]
    call   filter_user_exceptions
    npad   1 ; align next label
$LN8@main$filter$:
    add    rsp, 32
    pop    rbp
    ret    0
    int    3
main$filter$1 ENDP
text$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 [Igor Skochinsky, *Compiler Internals: Exceptions and RTTI*, (2012)]<sup>49</sup> for more detailed information about this.

Aside from exception information, `.pdata` is a section that contains the addresses of almost all function starts and ends, hence it may be useful for a tools targeted at automated analysis.

<sup>49</sup>Also available as <http://go.yurichev.com/17294>

[Matt Pietrek, *A Crash Course on the Depths of Win32™ Structured Exception Handling*, (1997)]<sup>50</sup>, [Igor Skochinsky, *Compiler Internals: Exceptions and RTTI*, (2012)]<sup>51</sup>.

### 6.5.4 Windows NT: Critical section

Critical sections in any OS are very important in multithreaded environment, mostly for giving a guarantee that only one thread can access some data in a single moment of time, while blocking other threads and interrupts.

That is how a `CRITICAL_SECTION` structure is declared in [Windows NT](#) line OS:

Listing 6.35: (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 6.36: 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
```

<sup>50</sup>Also available as <http://go.yurichev.com/17293>

<sup>51</sup>Also available as <http://go.yurichev.com/17294>

## 6.5. WINDOWS NT

The most important instruction in this code fragment is `BTR` (prefixed with `LOCK`):  
the zeroth bit is stored in the `CF` flag and cleared in memory. This is an [atomic operation](#),  
blocking all other CPUs' access to this piece of memory (see the `LOCK` prefix before the `BTR` instruction).  
If the bit at `LockCount` is 1,  
fine, reset it and return from the function: we are in a critical section.  
If not—the critical section is already occupied by other thread, so wait.  
The wait is performed there using `WaitForSingleObject()`.

And here is how the `LeaveCriticalSection()` function works:

Listing 6.37: 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], 0FFFFFFFh ; 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, 0FFFFFFFh
        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 adds 1 to `LockCount`, meanwhile saves initial value of `LockCount` in the `EBX` register.  
However, value in `EBX` is to be incremented with a help of subsequent `INC EBX`, and it also will be equal to  
the updated value of `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 in memory.

The `LOCK` prefix is very important:

without it two threads, each of which works on separate CPU or CPU core can try to enter a critical section  
and to modify the value in memory, which will result in non-deterministic behavior.

# Chapter 7

## Tools

Now that Dennis Yurichev has made this book free (*libre*), it is a contribution to the world of free knowledge and free education. However, for our freedom's sake, we need free (*libre*) reverse engineering tools to replace the proprietary tools described in this book.

---

Richard M. Stallman

### 7.1 Binary analysis

Tools you use when you don't run any process.

- (Free, open-source) *ent*<sup>1</sup>: entropy analyzing tool. Read more about entropy: [9.2 on page 945](#).
- *Hiew*<sup>2</sup>: for small modifications of code in binary files. Has dssembler/issasember.
- (Free, open-source) *GHex*<sup>3</sup>: simple hexadecimal editor for Linux.
- (Free, open-source) *xxd* and *od*: standard UNIX utilities for dumping.
- (Free, open-source) *strings*: \*NIX tool for searching for ASCII strings in binary files, including executable ones. Sysinternals has alternative<sup>4</sup> supporting wide char strings (UTF-16, widely used in Windows).
- (Free, open-source) *Binwalk*<sup>5</sup>: analyzing firmware images.
- (Free, open-source) *binary grep*: a small utility for searching any byte sequence in a big pile of files, including non-executable ones: [GitHub](#).

#### 7.1.1 Disassemblers

- *IDA*. An older freeware version is available for download<sup>6</sup>. Hot-keys cheatsheet: [.6.1 on page 1025](#)
- *Binary Ninja*<sup>7</sup>
- (Free, open-source) *zynamics BinNavi*<sup>8</sup>
- (Free, open-source) *objdump*: simple command-line utility for dumping and disassembling.
- (Free, open-source) *readelf*<sup>9</sup>: dump information about ELF file.

---

<sup>1</sup><http://www.fourmilab.ch/random/>

<sup>2</sup>[hiew.ru](http://hiew.ru)

<sup>3</sup><https://wiki.gnome.org/Apps/Ghex>

<sup>4</sup><https://technet.microsoft.com/en-us/sysinternals/strings>

<sup>5</sup><http://binwalk.org/>

<sup>6</sup>[hex-rays.com/products/ida/support/download\\_freeware.shtml](http://hex-rays.com/products/ida/support/download_freeware.shtml)

<sup>7</sup><http://binary.ninja/>

<sup>8</sup><https://www.zynamics.com/binnavi.html>

<sup>9</sup><https://sourceware.org/binutils/docs/binutils/readelf.html>

## 7.1.2 Decompilers

There is only one known, publicly available, high-quality decompiler to C code: *Hex-Rays*: [hex-rays.com/products/decompiler/](http://hex-rays.com/products/decompiler/)

## 7.1.3 Patch comparison/diffing

You may want to use it when you compare original version of some executable and patched one, in order to find what has been patched and why.

- (Free) *zynamics BinDiff*<sup>10</sup>
- (Free, open-source) *Diaphora*<sup>11</sup>

## 7.2 Live analysis

Tools you use on a live system or during running of a process.

### 7.2.1 Debuggers

- (Free) *OllyDbg*. Very popular user-mode win32 debugger<sup>12</sup>. Hot-keys cheatsheet: [.6.2 on page 1025](#)
- (Free, open-source) *GDB*. Not quite popular debugger among reverse engineers, because it's intended mostly for programmers. Some commands: [.6.5 on page 1026](#). There is a visual interface for GDB, "GDB dashboard"<sup>13</sup>.
- (Free, open-source) *LLDB*<sup>14</sup>.
- *WinDbg*<sup>15</sup>: kernel debugger for Windows.
- *IDA* has internal debugger.
- (Free, open-source) *Radare AKA rada.re AKA r2*<sup>16</sup>. A GUI also exists: *ragui*<sup>17</sup>.
- (Free, open-source) *tracer*. The author often uses *tracer*<sup>18</sup> instead of a debugger.

The author of these lines stopped using a debugger eventually, since all he needs from it is to spot function arguments while executing, or registers state at some point. Loading a debugger each time is too much, so a small utility called *tracer* was born. It works from command line, allows intercepting function execution, setting breakpoints at arbitrary places, reading and changing registers state, etc.

N.B.: the *tracer* isn't evolving, because it was developed as a demonstration tool for this book, not as everyday tool.

### 7.2.2 Library calls tracing

*ltrace*<sup>19</sup>.

---

<sup>10</sup><https://www.zynamics.com/software.html>

<sup>11</sup><https://github.com/joxeankoret/diaphora>

<sup>12</sup>[ollydbg.de](http://ollydbg.de)

<sup>13</sup><https://github.com/cyrus-and/gdb-dashboard>

<sup>14</sup><http://lldb.llvm.org/>

<sup>15</sup><https://developer.microsoft.com/en-us/windows/hardware/windows-driver-kit>

<sup>16</sup><http://rada.re/r/>

<sup>17</sup><http://radare.org/ragui/>

<sup>18</sup>[yurichev.com](http://yurichev.com)

<sup>19</sup><http://www.ltrace.org/>

## 7.2. LIVE ANALYSIS

### **7.2.3 System calls tracing**

## **strace / dtruss**

It shows which system calls (syscalls( 6.3 on page 745)) are called by a process right now.

For example:

Mac OS X has dtruss for doing the same.

Cygwin also has strace, but as far as it's known, it works only for .exe-files compiled for the cygwin environment itself.

#### 7.2.4 Network sniffing

*Sniffing* is intercepting some information you may be interested in.

(Free, open-source) *Wireshark*<sup>20</sup> for network sniffing. It has also capability for USB sniffing<sup>21</sup>.

Wireshark has a younger (or older) brother *tcpdump*<sup>22</sup>, simpler command-line tool.

### 7.2.5 Sysinternals

(Free) Sysinternals (developed by Mark Russinovich)<sup>23</sup>. At least these tools are important and worth studying: Process Explorer, Handle, VMMap, TCPView, Process Monitor.

## 7.2.6 Valgrind

(Free, open-source) a powerful tool for detecting memory leaks: <http://valgrind.org/>. Due to its powerful JIT mechanism, Valgrind is used as a framework for other tools.

### 7.2.7 Emulators

- (Free, open-source) *QEMU*<sup>24</sup>: emulator for various CPUs and architectures.
  - (Free, open-source) *DosBox*<sup>25</sup>: MS-DOS emulator, mostly used for retrogaming.
  - (Free, open-source) *SimH*<sup>26</sup>: emulator of ancient computers, mainframes, etc.

<sup>20</sup> <https://www.wireshark.org/>

<sup>21</sup> <https://wiki.wireshark.org/CaptureSetup/USB>

<sup>22</sup><http://www.tcpdump.org/>

<sup>23</sup><https://technet.microsoft.com/en-us/sysinternals/bb842062>

<sup>24</sup> <http://gemu.org>

25 <https://www.dosbox.com/>

[26](http://simh.trailing-edge.com/)

## 7.3 Other tools

*Microsoft Visual Studio Express* <sup>27</sup>: Stripped-down free version of Visual Studio, convenient for simple experiments.

Some useful options: [.6.3 on page 1025](#).

There is a website named “Compiler Explorer”, allowing to compile small code snippets and see output in various GCC versions and architectures (at least x86, ARM, MIPS): <http://gcc.betab.godbolt.org/>—I would use it myself for the book if I would know about it!

## 7.4 Something missing here?

If you know a great tool not listed here, please drop me a note:

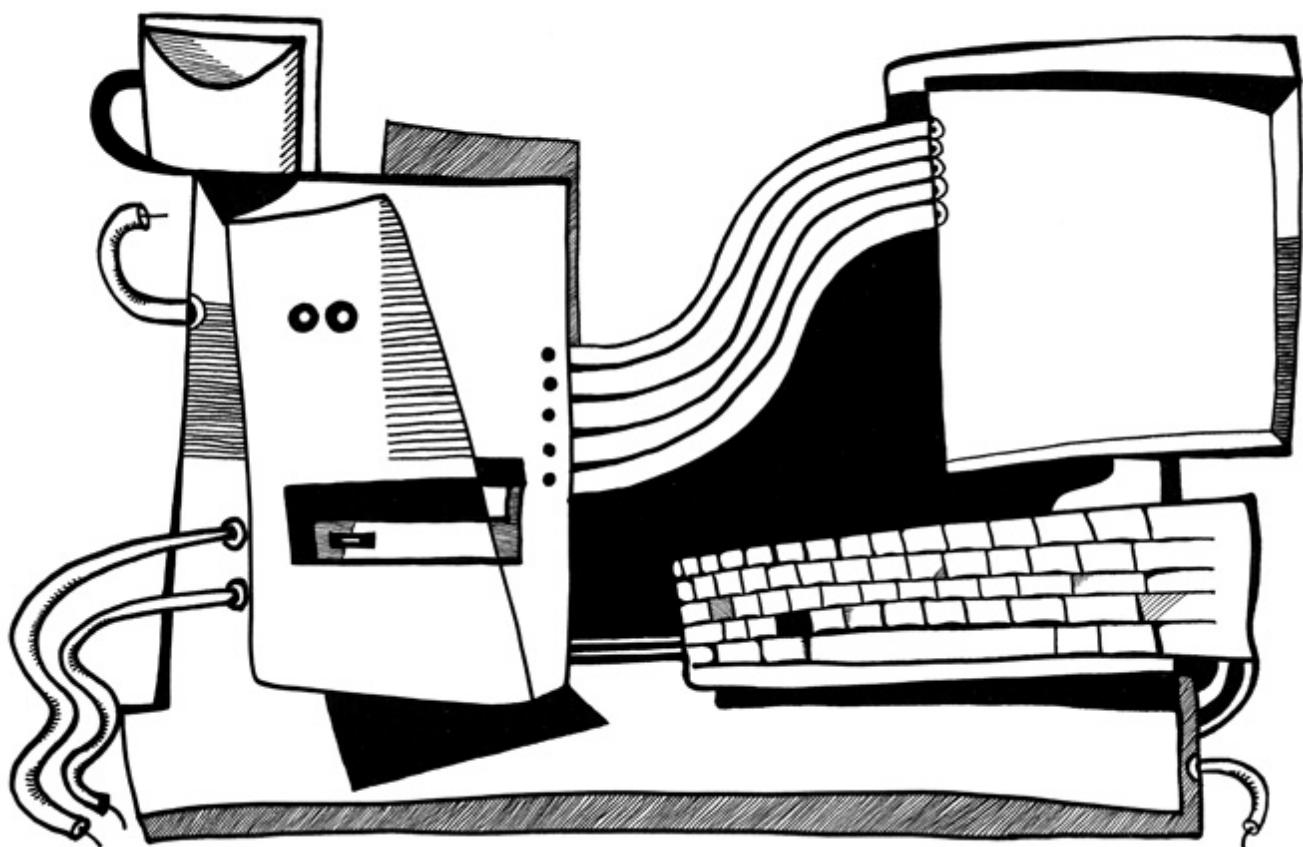
dennis(a)yurichev.com

---

<sup>27</sup>[visualstudio.com/en-US/products/visual-studio-express-vs](http://visualstudio.com/en-US/products/visual-studio-express-vs)

# **Chapter 8**

## **Case studies**



## 8.1 Task manager practical joke (Windows Vista)

Let's see if it's possible to hack Task Manager slightly so it would detect more CPU cores.

Let us first think, how does the Task Manager know the number of cores?

There is the `GetSystemInfo()` win32 function present in win32 userspace which can tell us this. But it's not imported in `taskmgr.exe`.

There is, however, another one in `NTAPI`, `NtQuerySystemInformation()`, which is used in `taskmgr.exe` in several places.

To get the number of cores, one has to call this function with the `SystemBasicInformation` constant as a first argument (which is zero<sup>1</sup>).

The second argument has to point to the buffer which is getting all the information.

So we have to find all calls to the

`NtQuerySystemInformation(0, ?, ?, ?)` function. Let's open `taskmgr.exe` in IDA.

What is always good about Microsoft executables is that IDA can download the corresponding `PDB` file for this executable and show all function names.

It is visible that Task Manager is written in C++ and some of the function names and classes are really speaking for themselves. There are classes `CAdapter`, `CNetPage`, `CPerfPage`, `CProcInfo`, `CProcPage`, `CSvcPage`, `CTaskPage`, `CUserPage`.

Apparently, each class corresponds to each tab in Task Manager.

Let's visit each call and add comment with the value which is passed as the first function argument. We will write "not zero" at some places, because the value there was clearly not zero, but something really different (more about this in the second part of this chapter).

And we are looking for zero passed as argument, after all.

xrefs to __imp_NtQuerySystemInformation			
Dire...	T.	Address	Text
↳ Up	p	wWinMain+50E	call cs:__imp_NtQuerySystemInformation; 0
↳ Up	p	wWinMain+542	call cs:__imp_NtQuerySystemInformation; 2
↳ Up	p	CPerfPage::TimerEvent(void)+200	call cs:__imp_NtQuerySystemInformation; not zero
↳	p	InitPerfInfo(void)+2C	call cs:__imp_NtQuerySystemInformation; 0
↳ D...	p	InitPerfInfo(void)+F0	call cs:__imp_NtQuerySystemInformation; 8
↳ D...	p	CalcCpuTime(int)+5F	call cs:__imp_NtQuerySystemInformation; 8
↳ D...	p	CalcCpuTime(int)+248	call cs:__imp_NtQuerySystemInformation; 2
↳ D...	p	CPerfPage::CalcPhysicalMem(unsigned ...)	call cs:__imp_NtQuerySystemInformation; not zero
↳ D...	p	CPerfPage::CalcPhysicalMem(unsigned ...)	call cs:__imp_NtQuerySystemInformation; not zero
↳ D...	p	CProcPage::GetProcessInfo(void)+2B	call cs:__imp_NtQuerySystemInformation; 5
↳ D...	p	CProcPage::UpdateProcInfoArray(void)+...	call cs:__imp_NtQuerySystemInformation; 0
↳ D...	p	CProcPage::UpdateProcInfoArray(void)+...	call cs:__imp_NtQuerySystemInformation; 2
↳ D...	p	CProcPage::Initialize(HWND__ *)+201	call cs:__imp_NtQuerySystemInformation; 0
↳ D...	p	CProcPage::GetTaskListEx(void)+3C	call cs:__imp_NtQuerySystemInformation; 5

Figure 8.1: IDA: cross references to `NtQuerySystemInformation()`

Yes, the names are really speaking for themselves.

When we closely investigate each place where

`NtQuerySystemInformation(0, ?, ?, ?)` is called, we quickly find what we need in the `InitPerfInfo()` function:

<sup>1</sup>MSDN

## 8.1. TASK MANAGER PRACTICAL JOKE (WINDOWS VISTA)

Listing 8.1: taskmgr.exe (Windows Vista)

```
.text:10000B4B3 xor    r9d, r9d
.text:10000B4B6 lea    rdx, [rsp+0C78h+var_C58] ; buffer
.text:10000B4BB xor    ecx, ecx
.text:10000B4BD lea    ebp, [r9+40h]
.text:10000B4C1 mov    r8d, ebp
.text:10000B4C4 call   cs:_imp_NtQuerySystemInformation ; 0
.text:10000B4CA xor    ebx, ebx
.text:10000B4CC cmp    eax, ebx
.text:10000B4CE jge    short loc_10000B4D7
.text:10000B4D0
.text:10000B4D0 loc_10000B4D0:           ; CODE XREF: InitPerfInfo(void)+97
.text:10000B4D0                 ; InitPerfInfo(void)+AF
xor    al, al
jmp   loc_10000B5EA
.text:10000B4D7 ; -----
.text:10000B4D7 loc_10000B4D7:           ; CODE XREF: InitPerfInfo(void)+36
mov    eax, [rsp+0C78h+var_C50]
.text:10000B4DB mov    esi, ebx
.text:10000B4DD mov    r12d, 3E80h
.text:10000B4E3 mov    cs:?g_PageSize@@3KA, eax ; ulong g_PageSize
.text:10000B4E9 shr    eax, 0Ah
.text:10000B4EC lea    r13, __ImageBase
.text:10000B4F3 imul   eax, [rsp+0C78h+var_C4C]
.text:10000B4F8 cmp    [rsp+0C78h+var_C20], bpl
.text:10000B4FD mov    cs:?g_MEMMax@@3_JA, rax ; __int64 g_MEMMax
.text:10000B504 movzx  eax, [rsp+0C78h+var_C20] ; number of CPUs
.text:10000B509 cmova  eax, ebp
.text:10000B50C cmp    al, bl
.text:10000B50E mov    cs:?g_cProcessors@@3EA, al ; uchar g_cProcessors
```

`g_cProcessors` is a global variable, and this name has been assigned by IDA according to the [PDB](#) loaded from Microsoft's symbol server.

The byte is taken from `var_C20`. And `var_C58` is passed to

`NtQuerySystemInformation()` as a pointer to the receiving buffer. The difference between `0xC20` and `0xC58` is `0x38` (56).

Let's take a look at format of the return structure, which we can find in MSDN:

```
typedef struct _SYSTEM_BASIC_INFORMATION {
    BYTE Reserved1[24];
    PVOID Reserved2[4];
    CCHAR NumberOfProcessors;
} SYSTEM_BASIC_INFORMATION;
```

This is a x64 system, so each `PVOID` takes 8 byte.

All `reserved` fields in the structure take  $24 + 4 * 8 = 56$  bytes.

Oh yes, this implies that `var_C20` is the local stack is exactly the `NumberOfProcessors` field of the `SYSTEM_BASIC_INFORMATION` structure.

Let's check our guess. Copy `taskmgr.exe` from `C:\Windows\System32` to some other folder (so the `Windows Resource Protection` will not try to restore the patched `taskmgr.exe`).

Let's open it in Hiew and find the place:

### 8.1. TASK MANAGER PRACTICAL JOKE (WINDOWS VISTA)

01`0000B4F8: 40386C2458	cmp [rsp][058], bp
01`0000B4FD: 48890544A00100	mov [00000001, 00025548], rax
01`0000B504: 0FB6442458	movzx eax, b, [rsp][058]
01`0000B509: 0F47C5	cmova eax, ebp
01`0000B50C: 3AC3	cmp al, bl
01`0000B50E: 880574950100	mov [00000001, 00024A88], al
01`0000B514: 7645	jbe .00000001, 0000B55B --B
01`0000B516: 488BFB	mov rdi, rbx
01`0000B519: 498BD4	mov rdx, r12
01`0000B51C: 8BCD	mov ecx, ebp

Figure 8.2: Hiew: find the place to be patched

Let's replace the MOVZX instruction with ours. Let's pretend we've got 64 CPU cores.

Add one additional NOP (because our instruction is shorter than the original one):

00`0000A8F8: 40386C2458	cmp [rsp][058], bp
00`0000A8FD: 48890544A00100	mov [000024948], rax
00`0000A904: 66B84000	mov ax, 00040 ; @'
00`0000A908: 90	nop
00`0000A909: 0F47C5	cmova eax, ebp
00`0000A90C: 3AC3	cmp al, bl
00`0000A90E: 880574950100	mov [000023E88], al
00`0000A914: 7645	jbe 00000A95B
00`0000A916: 488BFB	mov rdi, rbx
00`0000A919: 498BD4	mov rdx, r12
00`0000A91C: 8BCD	mov ecx, ebp

Figure 8.3: Hiew: patch it

And it works! Of course, the data in the graphs is not correct.

At times, Task Manager even shows an overall CPU load of more than 100%.

## 8.1. TASK MANAGER PRACTICAL JOKE (WINDOWS VISTA)

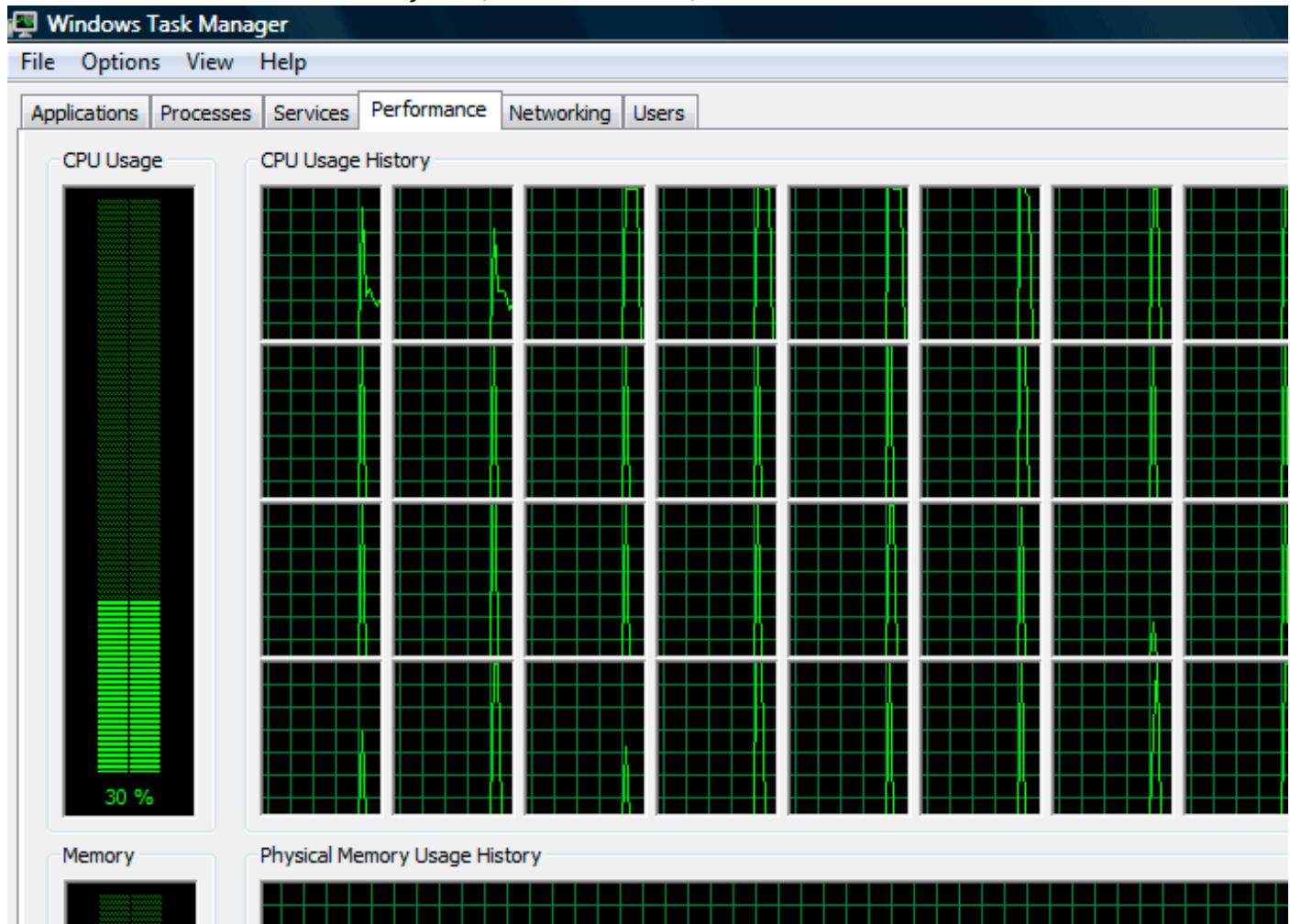


Figure 8.4: Fooled Windows Task Manager

The biggest number Task Manager does not crash with is 64.

Apparently, Task Manager in Windows Vista was not tested on computers with a large number of cores. So there are probably some static data structure(s) inside it limited to 64 cores.

### 8.1.1 Using LEA to load values

Sometimes, `LEA` is used in `taskmgr.exe` instead of `MOV` to set the first argument of `NtQuerySystemInformation()`:

Listing 8.2: taskmgr.exe (Windows Vista)

```
xor    r9d, r9d
div    dword ptr [rsp+4C8h+WndClass.lpfnWndProc]
lea    rdx, [rsp+4C8h+VersionInformation]
lea    ecx, [r9+2]      ; put 2 to ECX
mov    r8d, 138h
mov    ebx, eax
; ECX=SystemPerformanceInformation
call   cs:_imp_NtQuerySystemInformation ; 2
...
mov    r8d, 30h
lea    r9, [rsp+298h+var_268]
lea    rdx, [rsp+298h+var_258]
lea    ecx, [r8-2Dh]    ; put 3 to ECX
; ECX=SystemTimeOfDayInformation
```

### 8.1. TASK MANAGER PRACTICAL JOKE (WINDOWS VISTA)

```
call    cs:_imp_NtQuerySystemInformation ; not zero
...
mov    rbp, [rsi+8]
mov    r8d, 20h
lea    r9, [rsp+98h+arg_0]
lea    rdx, [rsp+98h+var_78]
lea    ecx, [r8+2Fh] ; put 0x4F to ECX
mov    [rsp+98h+var_60], ebx
mov    [rsp+98h+var_68], rbp
; ECX=SystemSuperfetchInformation
call    cs:_imp_NtQuerySystemInformation ; not zero
```

Perhaps MSVC did so because machine code of LEA is shorter than MOV REG, 5 (would be 5 instead of 4).

LEA with offset in -128..127 range (offset will occupy 1 byte in opcode) with 32-bit registers is even shorter (for lack of REX prefix)—3 bytes.

Another example of such thing is: [6.1.5 on page 737](#).

## 8.2 Color Lines game practical joke

This is a very popular game with several implementations in existence. We can take one of them, called BallTriX, from 1997, available freely at <http://go.yurichev.com/17311><sup>2</sup>. Here is how it looks:

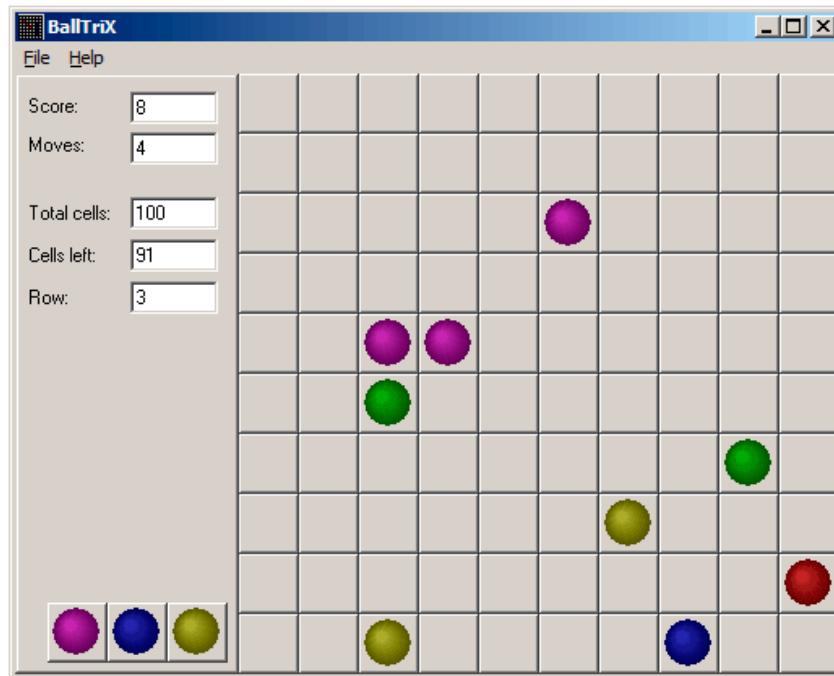


Figure 8.5: This is how the game is usually looks like

<sup>2</sup>Or at <http://go.yurichev.com/17365> or <http://go.yurichev.com/17366>.

## 8.2. COLOR LINES GAME PRACTICAL JOKE

So let's see, is it be possible to find the random generator and do some trick with it. [IDA](#) quickly recognize the standard `_rand` function in `balltrix.exe` at `0x00403DA0`. [IDA](#) also shows that it is called only from one place:

```
.text:00402C9C sub_402C9C    proc near                ; CODE XREF: sub_402ACA+52
.text:00402C9C
.text:00402C9C
.text:00402C9C arg_0        = dword ptr  8
.text:00402C9C
.text:00402C9C             push    ebp
.text:00402C9D             mov     ebp, esp
.text:00402C9F             push    ebx
.text:00402CA0             push    esi
.text:00402CA1             push    edi
.text:00402CA2             mov     eax, dword_40D430
.text:00402CA7             imul   eax, dword_40D440
.text:00402CAE             add    eax, dword_40D5C8
.text:00402CB4             mov     ecx, 32000
.text:00402CB9             cdq
.text:00402CBA             idiv   ecx
.text:00402CBC             mov     dword_40D440, edx
.text:00402CC2             call   _rand
.text:00402CC7             cdq
.text:00402CC8             idiv   [ebp+arg_0]
.text:00402CCB             mov     dword_40D430, edx
.text:00402CD1             mov     eax, dword_40D430
.text:00402CD6             jmp    $+5
.text:00402CDB             pop    edi
.text:00402CDC             pop    esi
.text:00402CDD             pop    ebx
.text:00402CDE             leave
.text:00402CDF             retn
.text:00402CDF sub_402C9C    endp
```

We'll call it "random". Let's not to dive into this function's code yet.

This function is referred from 3 places.

Here are the first two:

```
.text:00402B16             mov     eax, dword_40C03C ; 10 here
.text:00402B1B             push   eax
.text:00402B1C             call   random
.text:00402B21             add    esp, 4
.text:00402B24             inc    eax
.text:00402B25             mov    [ebp+var_C], eax
.text:00402B28             mov    eax, dword_40C040 ; 10 here
.text:00402B2D             push   eax
.text:00402B2E             call   random
.text:00402B33             add    esp, 4
```

Here is the third one:

```
.text:00402BBB             mov     eax, dword_40C058 ; 5 here
.text:00402BC0             push   eax
.text:00402BC1             call   random
.text:00402BC6             add    esp, 4
.text:00402BC9             inc    eax
```

So the function has only one argument.

10 is passed in first two cases and 5 in third. We can also notice that the board has a size of  $10 \times 10$  and there are 5 possible colors. This is it! The standard `rand()` function returns a number in the `0..0x7FFF` range and this is often inconvenient, so many programmers implement their own random functions which returns a random number in a specified range. In our case, the range is  $0..n - 1$  and  $n$  is passed as the sole argument of the function. We can quickly check this in any debugger.

So let's fix the third function call to always return zero. First, we will replace three instructions (`PUSH/CALL/ADD`) by `NOPs`. Then we'll add `XOR EAX, EAX` instruction, to clear the `EAX` register.

## 8.2. COLOR LINES GAME PRACTICAL JOKE

```
.00402BB8: 83C410      add    esp,010
.00402BBB: A158C04000  mov    eax,[00040C058]
.00402BC0: 31C0        xor    eax, eax
.00402BC2: 90          nop
.00402BC3: 90          nop
.00402BC4: 90          nop
.00402BC5: 90          nop
.00402BC6: 90          nop
.00402BC7: 90          nop
.00402BC8: 90          nop
.00402BC9: 40          inc    eax
.00402BCA: 8B4DF8      mov    ecx,[ebp][-8]
.00402BCD: 8D0C49      lea    ecx,[ecx][ecx]*2
.00402BD0: 8B15F4D54000 mov    edx,[00040D5F4]
```

So what we did is we replaced a call to the `random()` function by a code which always returns zero.

### 8.3. MINESWEEPER (WINDOWS XP)

Let's run it now:

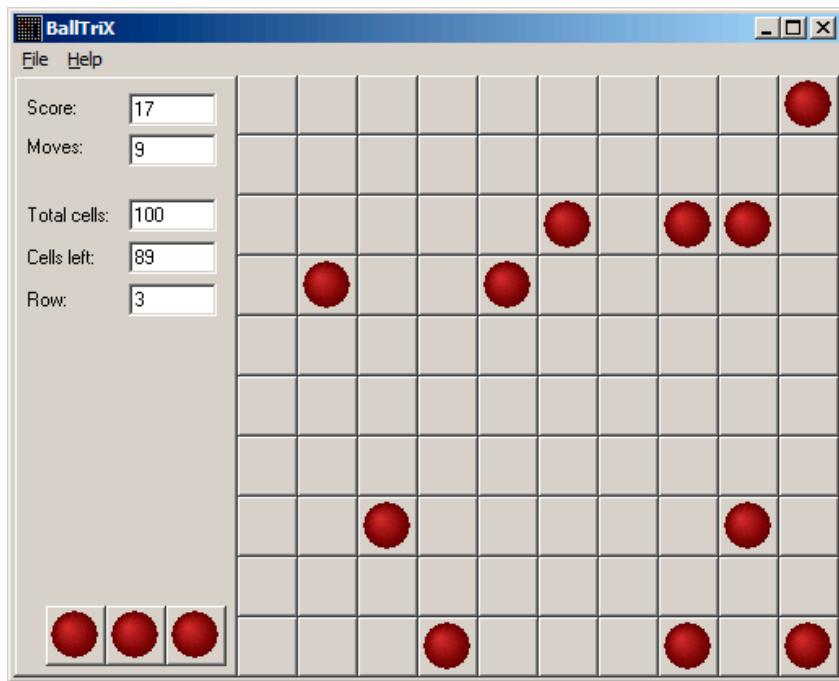


Figure 8.6: Practical joke works

Oh yes, it works<sup>3</sup>.

But why are the arguments to the `random()` functions global variables? That's just because it's possible to change the board size in the game's settings, so these values are not hardcoded. The 10 and 5 values are just defaults.

## 8.3 Minesweeper (Windows XP)

For those who are not very good at playing Minesweeper, we could try to reveal the hidden mines in the debugger.

As we know, Minesweeper places mines randomly, so there has to be some kind of random number generator or a call to the standard `rand()` C-function.

What is really cool about reversing Microsoft products is that there are **PDB** file with symbols (function names, etc). When we load `winmine.exe` into **IDA**, it downloads the **PDB** file exactly for this executable and shows all names.

So here it is, the only call to `rand()` is this function:

```
.text:01003940 ; __stdcall Rnd(x)
.text:01003940 _Rnd@4          proc near             ; CODE XREF: StartGame()+53
.text:01003940                                         ; StartGame()+61
.text:01003940
.text:01003940 arg_0           = dword ptr  4
.text:01003940
.text:01003940                                         call    ds:_imp__rand
.text:01003946                                         cdq
.text:01003947                                         idiv   [esp+arg_0]
.text:0100394B                                         mov    eax, edx
.text:0100394D                                         retn   4
.text:0100394D _Rnd@4          endp
```

**IDA** named it so, and it was the name given to it by Minesweeper's developers.

The function is very simple:

<sup>3</sup>Author of this book once did this as a joke for his coworkers with the hope that they would stop playing. They didn't.

### 8.3. MINESWEEPER (WINDOWS XP)

```
int Rnd(int limit)
{
    return rand() % limit;
};
```

(There is no “limit” name in the PDB file; we manually named this argument like this.)

So it returns a random value from 0 to a specified limit.

Rnd() is called only from one place, a function called StartGame(), and as it seems, this is exactly the code which place the mines:

```
.text:010036C7          push    _xBoxMac
.text:010036CD          call    _Rnd@4           ; Rnd(x)
.text:010036D2          push    _yBoxMac
.text:010036D8          mov     esi, eax
.text:010036DA          inc    esi
.text:010036DB          call    _Rnd@4           ; Rnd(x)
.text:010036E0          inc    eax
.text:010036E1          mov     ecx, eax
.text:010036E3          shl    ecx, 5            ; ECX=ECX*32
.text:010036E6          test   _rgBlk[ecx+esi], 80h
.text:010036EE          jnz    short loc_10036C7
.text:010036F0          shl    eax, 5            ; EAX=EAX*32
.text:010036F3          lea    eax, _rgBlk[eax+esi]
.text:010036FA          or     byte ptr [eax], 80h
.text:010036FD          dec    _cBombStart
.text:01003703          jnz    short loc_10036C7
```

Minesweeper allows you to set the board size, so the X (xBoxMac) and Y (yBoxMac) of the board are global variables. They are passed to Rnd() and random coordinates are generated. A mine is placed by the OR instruction at 0x010036FA. And if it has been placed before (it's possible if the pair of Rnd() generates a coordinates pair which has been already generated), then TEST and JNZ at 0x010036E6 jumps to the generation routine again.

cBombStart is the global variable containing total number of mines. So this is loop.

The width of the array is 32 (we can conclude this by looking at the SHL instruction, which multiplies one of the coordinates by 32).

The size of the rgBlk global array can be easily determined by the difference between the rgBlk label in the data segment and the next known one. It is 0x360 (864):

```
.data:01005340 _rgBlk          db 360h dup(?)           ; DATA XREF: MainWndProc(x,x,x,x)+574
.data:01005340                  ; DisplayBlk(x,x)+23
.data:010056A0 _Preferences     dd ?                   ; DATA XREF: FixMenus()+
...
```

864/32 = 27.

So the array size is  $27 * 32$ ? It is close to what we know: when we try to set board size to  $100 * 100$  in Minesweeper settings, it fallbacks to a board of size  $24 * 30$ . So this is the maximal board size here. And the array has a fixed size for any board size.

So let's see all this in OllyDbg. We will ran Minesweeper, attaching OllyDbg to it and now we can see the memory dump at the address of the rgBlk array (0x01005340)<sup>4</sup>.

So we got this memory dump of the array:

Address	Hex dump
01005340	10 10 10 10 10 10 10 10 10 10 10 10 0F 0F 0F 0F 0F
01005350	0F
01005360	10 0F 10 0F 0F 0F 0F 0F 0F 0F
01005370	0F
01005380	10 0F 10 0F 0F 0F 0F 0F 0F 0F
01005390	0F
010053A0	10 0F 0F 0F 0F 0F 0F 0F 8F 0F 10 0F 0F 0F 0F 0F 0F 0F

<sup>4</sup>All addresses here are for Minesweeper for Windows XP SP3 English. They may differ for other service packs.

### 8.3. MINESWEEPER (WINDOWS XP)

```

010053B0 0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|
010053C0 10 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|10 0F 0F 0F 0F 0F 0F 0F|
010053D0 0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F 0F 0F 0F 0F|
010053E0 10 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|10 0F 0F 0F 0F 0F 0F 0F|
010053F0 0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F 0F 0F 0F 0F|
01005400 10 0F 0F 8F|0F 0F 8F 0F|0F 0F 0F 10 0F 0F 0F 0F 0F 0F 0F 0F|
01005410 0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F 0F 0F 0F 0F|
01005420 10 8F 0F 0F|8F 0F 0F 0F|0F 0F 0F 10 0F 0F 0F 0F 0F 0F 0F 0F|
01005430 0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F 0F 0F 0F 0F|
01005440 10 8F 0F 0F|0F 0F 0F 0F|8F 0F 0F 8F 10 0F 0F 0F 0F 0F 0F 0F|
01005450 0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F 0F 0F 0F 0F|
01005460 10 0F 0F 0F|0F 0F 0F 0F|8F 0F 0F 8F 10 0F 0F 0F 0F 0F 0F 0F|
01005470 0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F 0F 0F 0F 0F|
01005480 10 10 10 10|10 10 10 10|10 10 10 10 10 0F 0F 0F 0F 0F 0F 0F|
01005490 0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F 0F 0F 0F 0F|
010054A0 0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F 0F 0F 0F 0F|
010054B0 0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F 0F 0F 0F 0F|
010054C0 0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F|0F 0F 0F 0F 0F 0F 0F 0F|

```

OllyDbg, like any other hexadecimal editor, shows 16 bytes per line. So each 32-byte array row occupies exactly 2 lines here.

This is beginner level (9\*9 board).

There is some square structure can be seen visually (0x10 bytes).

We will click “Run” in OllyDbg to unfreeze the Minesweeper process, then we’ll click randomly at the Minesweeper window and trapped into mine, but now all mines are visible:

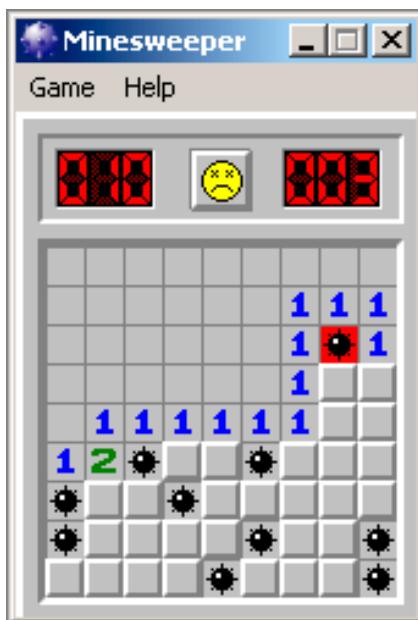


Figure 8.7: Mines

By comparing the mine places and the dump, we can conclude that 0x10 stands for border, 0x0F—empty block, 0x8F—mine.

Now we’ll add comments and also enclose all 0x8F bytes into square brackets:

```

border:
01005340 10 10 10 10 10 10 10 10 10 10 0F 0F 0F 0F 0F
01005350 0F 0F
line #1:
01005360 10 0F 0F 0F 0F 0F 0F 0F 0F 10 0F 0F 0F 0F 0F 0F
01005370 0F 0F
line #2:
01005380 10 0F 0F 0F 0F 0F 0F 0F 0F 10 0F 0F 0F 0F 0F 0F
01005390 0F 0F
line #3:
010053A0 10 0F 0F 0F 0F 0F 0F [8F]0F 10 0F 0F 0F 0F 0F 0F
010053B0 0F 0F

```

### 8.3. MINESWEEPER (WINDOWS XP)

```
line #4:  
010053C0  10 0F 0F 0F 0F 0F 0F 0F 0F 10 0F 0F 0F 0F 0F 0F  
010053D0  0F  
line #5:  
010053E0  10 0F 0F 0F 0F 0F 0F 0F 0F 10 0F 0F 0F 0F 0F 0F  
010053F0  0F  
line #6:  
01005400  10 0F 0F[8F]0F 0F[8F]0F 0F 0F 10 0F 0F 0F 0F 0F  
01005410  0F  
line #7:  
01005420  10[8F]0F 0F[8F]0F 0F 0F 0F 0F 10 0F 0F 0F 0F 0F  
01005430  0F  
line #8:  
01005440  10[8F]0F 0F 0F 0F[8F]0F 0F[8F]10 0F 0F 0F 0F 0F  
01005450  0F  
line #9:  
01005460  10 0F 0F 0F 0F[8F]0F 0F 0F[8F]10 0F 0F 0F 0F 0F  
01005470  0F  
border:  
01005480  10 10 10 10 10 10 10 10 10 10 10 10 0F 0F 0F 0F  
01005490  0F 0F
```

Now we'll remove all *border bytes* (0x10) and what's beyond those:

```
0F 0F 0F 0F 0F 0F 0F 0F  
0F 0F 0F 0F 0F 0F 0F 0F  
0F 0F 0F 0F 0F 0F[8F]0F  
0F 0F 0F 0F 0F 0F 0F 0F  
0F 0F 0F 0F 0F 0F 0F 0F  
0F 0F[8F]0F 0F[8F]0F 0F 0F  
[8F]0F 0F[8F]0F 0F 0F 0F 0F  
[8F]0F 0F 0F 0F[8F]0F 0F[8F]  
0F 0F 0F 0F[8F]0F 0F 0F[8F]
```

Yes, these are mines, now it can be clearly seen and compared with the screenshot.

### 8.3. MINESWEEPER (WINDOWS XP)

What is interesting is that we can modify the array right in OllyDbg. We can remove all mines by changing all 0x8F bytes by 0x0F, and here is what we'll get in Minesweeper:

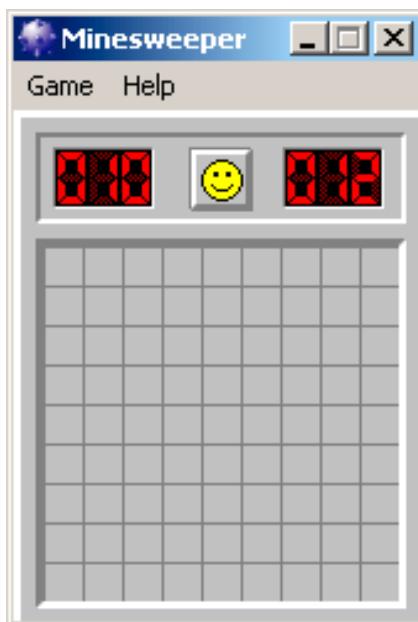


Figure 8.8: All mines are removed in debugger

We can also move all of them to the first line:

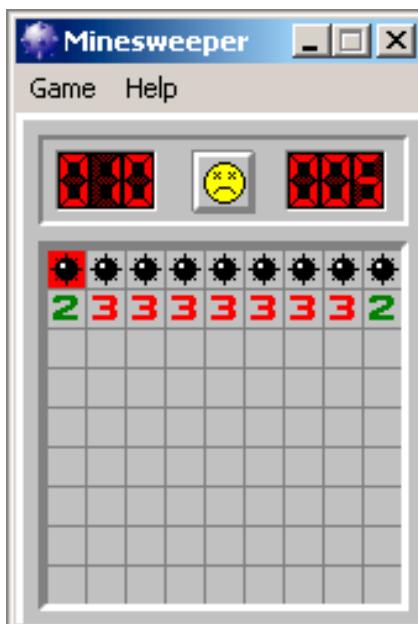


Figure 8.9: Mines set in debugger

Well, the debugger is not very convenient for eavesdropping (which is our goal anyway), so we'll write a small utility to dump the contents of the board:

```
// Windows XP MineSweeper cheater
// written by dennis(a)yurichev.com for http://beginners.re/ book
#include <windows.h>
#include <assert.h>
#include <stdio.h>

int main (int argc, char * argv[])
{
    int i, j;
    HANDLE h;
    DWORD PID, address, rd;
    BYTE board[27][32];
```

### 8.3. MINESWEEPER (WINDOWS XP)

```
if (argc!=3)
{
    printf ("Usage: %s <PID> <address>\n", argv[0]);
    return 0;
};

assert (argv[1]!=NULL);
assert (argv[2]!=NULL);

assert (sscanf (argv[1], "%d", &PID)==1);
assert (sscanf (argv[2], "%x", &address)==1);

h=OpenProcess (PROCESS_VM_OPERATION | PROCESS_VM_READ | PROCESS_VM_WRITE, FALSE, PID);

if (h==NULL)
{
    DWORD e=GetLastError();
    printf ("OpenProcess error: %08X\n", e);
    return 0;
};

if (ReadProcessMemory (h, (LPVOID)address, board, sizeof(board), &rd )!=TRUE)
{
    printf ("ReadProcessMemory() failed\n");
    return 0;
};

for (i=1; i<26; i++)
{
    if (board[i][0]==0x10 && board[i][1]==0x10)
        break; // end of board
    for (j=1; j<31; j++)
    {
        if (board[i][j]==0x10)
            break; // board border
        if (board[i][j]==0x8F)
            printf ("*");
        else
            printf (" ");
    };
    printf ("\n");
};

CloseHandle (h);
};
```

Just set the [PID<sup>5</sup>](#) <sup>6</sup> and the address of the array (`0x01005340` for Windows XP SP3 English) and it will dump it <sup>7</sup>.

It attaches itself to a win32 process by [PID](#) and just reads process memory at the address.

#### 8.3.1 Exercises

- Why do the *border bytes* (0x10) exist in the array?

What they are for if they are not visible in Minesweeper's interface? How could it work without them?

- As it turns out, there are more values possible (for open blocks, for flagged by user, etc). Try to find the meaning of each one.
- Modify my utility so it can remove all mines or set them in a fixed pattern that you want in the Minesweeper process currently running.

<sup>5</sup>Program/process ID

<sup>6</sup>PID it can be seen in Task Manager (enable it in "View → Select Columns")

<sup>7</sup>The compiled executable is here: [beginners.re](#)

#### 8.4. HACKING WINDOWS CLOCK

- Modify my utility so it can work without the array address specified and without a PDB file.
- Yes, it's possible to find board information in the data segment of Minesweeper's running process automatically.

## 8.4 Hacking Windows clock

Sometimes I did some kind of first April prank for my coworkers.

Let's find, if we could do something with Windows clock? Can we force to go clock hands backwards?

First of all, when you click on date/time in status bar, a C:\WINDOWS\SYSTEM32\TIMEDATE.CPL module gets executed, which is usual executable PE file.

Let's see, how it draw hands? When I open the file (from Windows 7) in Resource Hacker, there are clock faces, but with no hands:

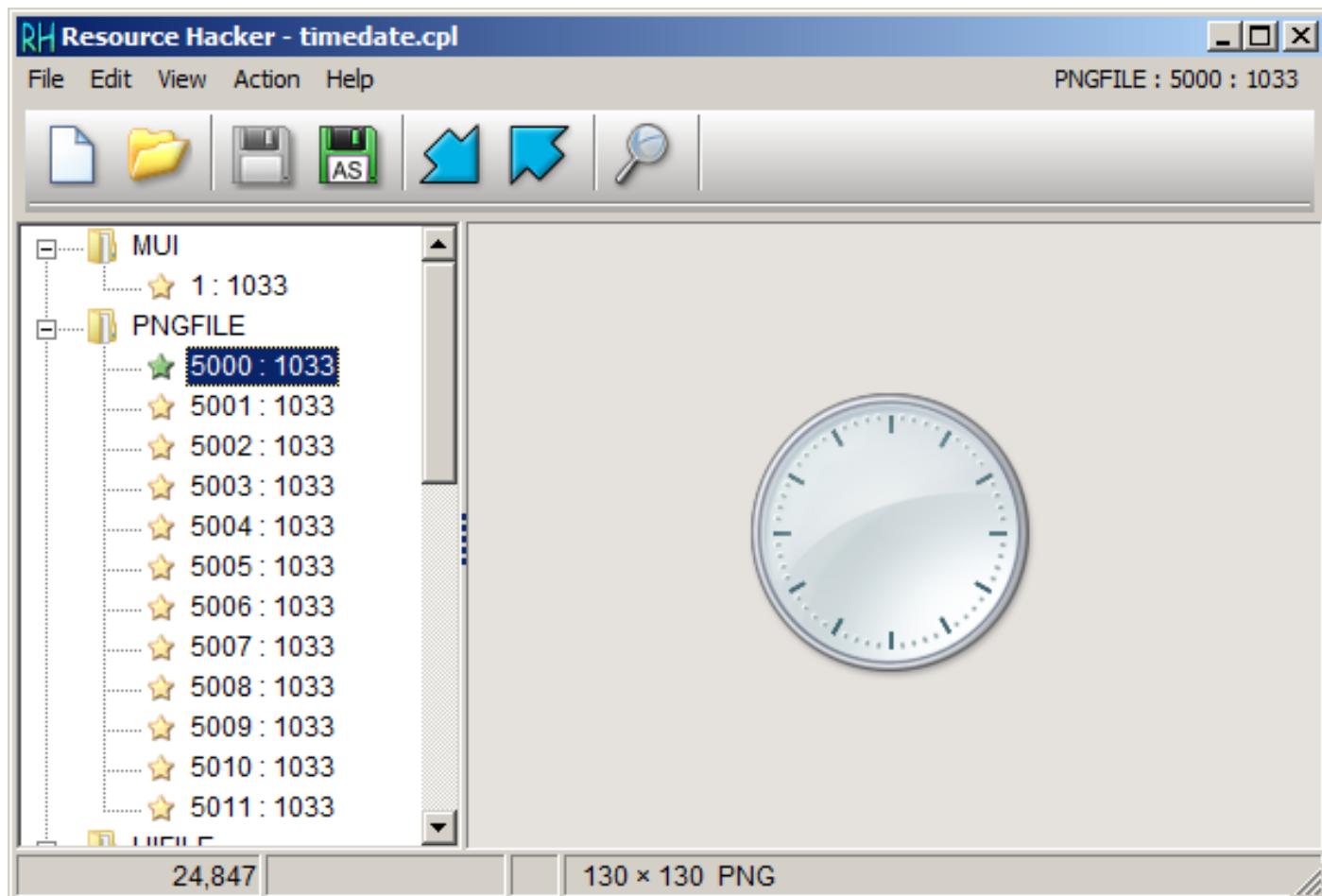


Figure 8.10: Resource Hacker

OK, what we know? How to draw a clock hand? All they are started at the middle of circle, ending with its border. Hence, we must calculate coordinates of a point on circle's border. From school-level mathematics we may recall that we have to use sine/cosine functions to draw circle, or at least square root. There are no such things in TIMEDATE.CPL, at least at first glance. But, thanks to Microsoft debugging PDB files, I can find a function named *CAnalogClock::DrawHand()*, which calls *Gdiplus::Graphics::DrawLine()* at least twice.

Here is its code:

```
.text:6EB9DBC7 ; private: enum Gdiplus::Status __thiscall CAnalogClock::DrawHand(class ↴
    ↴ Gdiplus::Graphics *, int, struct ClockHand const &, class Gdiplus::Pen *)
.text:6EB9DBC7 ?_DrawHand@CAnalogClock@@AAE?↗
    ↴ AW4Status@Gdiplus@@PAVGraphics@3@HABUClockHand@@PAVPen@3@@Z proc near
```

#### 8.4. HACKING WINDOWS CLOCK

```

.text:6EB9DBC7 ; CODE XREF: CAalogClock::_ClockPaint(✓
    ↳ HDC__ *)+163
.text:6EB9DBC7 ; CAalogClock::_ClockPaint(HDC__ *)+18B
.text:6EB9DBC7
.text:6EB9DBC7 var_10 = dword ptr -10h
.text:6EB9DBC7 var_C = dword ptr -0Ch
.text:6EB9DBC7 var_8 = dword ptr -8
.text:6EB9DBC7 var_4 = dword ptr -4
.text:6EB9DBC7 arg_0 = dword ptr 8
.text:6EB9DBC7 arg_4 = dword ptr 0Ch
.text:6EB9DBC7 arg_8 = dword ptr 10h
.text:6EB9DBC7 arg_C = dword ptr 14h
.text:6EB9DBC7
.text:6EB9DBC7 mov edi, edi
.text:6EB9DBC7 push ebp
.text:6EB9DBCA mov ebp, esp
.text:6EB9DBCC sub esp, 10h
.text:6EB9DBCF mov eax, [ebp+arg_4]
.text:6EB9DBD2 push ebx
.text:6EB9DBD3 push esi
.text:6EB9DBD4 push edi
.text:6EB9DBD5 cdq
.text:6EB9DBD6 push 3Ch
.text:6EB9DBD8 mov esi, ecx
.text:6EB9DBDA pop ecx
.text:6EB9DBDB idiv ecx
.text:6EB9DBDD push 2
.text:6EB9DBDF lea ebx, table[edx*8]
.text:6EB9DBE6 lea eax, [edx+1Eh]
.text:6EB9DBE9 cdq
.text:6EB9DBEA idiv ecx
.text:6EB9DBEC mov ecx, [ebp+arg_0]
.text:6EB9DBEF mov [ebp+var_4], ebx
.text:6EB9DBF2 lea eax, table[edx*8]
.text:6EB9DBF9 mov [ebp+arg_4], eax
.text:6EB9DBFC call ?SetInterpolationMode@Graphics@Gdiplus@@QAE?✓
    ↳ AW4Status@2@W4InterpolationMode@2@Z ; Gdiplus::Graphics::SetInterpolationMode(Gdiplus::✓
    ↳ InterpolationMode)
.text:6EB9DC01 mov eax, [esi+70h]
.text:6EB9DC04 mov edi, [ebp+arg_8]
.text:6EB9DC07 mov [ebp+var_10], eax
.text:6EB9DC0A mov eax, [esi+74h]
.text:6EB9DC0D mov [ebp+var_C], eax
.text:6EB9DC10 mov eax, [edi]
.text:6EB9DC12 sub eax, [edi+8]
.text:6EB9DC15 push 8000 ; nDenominator
.text:6EB9DC1A push eax ; nNumerator
.text:6EB9DC1B push dword ptr [ebx+4] ; nNumber
.text:6EB9DC1E mov ebx, ds:_imp_MulDiv@12 ; MulDiv(x,x,x)
.text:6EB9DC24 call ebx ; MulDiv(x,x,x) ; MulDiv(x,x,x)
.text:6EB9DC26 add eax, [esi+74h]
.text:6EB9DC29 push 8000 ; nDenominator
.text:6EB9DC2E mov [ebp+arg_8], eax
.text:6EB9DC31 mov eax, [edi]
.text:6EB9DC33 sub eax, [edi+8]
.text:6EB9DC36 push eax ; nNumerator
.text:6EB9DC37 mov eax, [ebp+var_4]
.text:6EB9DC3A push dword ptr [eax] ; nNumber
.text:6EB9DC3C call ebx ; MulDiv(x,x,x) ; MulDiv(x,x,x)
.text:6EB9DC3E add eax, [esi+70h]
.text:6EB9DC41 mov ecx, [ebp+arg_0]
.text:6EB9DC44 mov [ebp+var_8], eax
.text:6EB9DC47 mov eax, [ebp+arg_8]
.text:6EB9DC4A mov [ebp+var_4], eax
.text:6EB9DC4D lea eax, [ebp+var_8]
.text:6EB9DC50 push eax
.text:6EB9DC51 lea eax, [ebp+var_10]
.text:6EB9DC54 push eax
.text:6EB9DC55 push [ebp+arg_C]
.text:6EB9DC58 call ?DrawLine@Graphics@Gdiplus@@QAE?✓

```

#### 8.4. HACKING WINDOWS CLOCK

```

    ↳ AW4Status@2@PBVPen@2@ABVPoint@2@1@Z ; Gdiplus::Graphics::DrawLine(Gdiplus::Pen const *, ↳
    ↳ Gdiplus::Point const &,Gdiplus::Point const &)
.text:6EB9DC5D          mov     ecx, [edi+8]
.text:6EB9DC60          test    ecx, ecx
.text:6EB9DC62          jbe    short loc_6EB9DCAA
.text:6EB9DC64          test    eax, eax
.text:6EB9DC66          jnz    short loc_6EB9DCAA
.text:6EB9DC68          mov     eax, [ebp+arg_4]
.text:6EB9DC6B          push    8000      ; nDenominator
.text:6EB9DC70          push    ecx       ; nNumerator
.text:6EB9DC71          push    dword ptr [eax+4] ; nNumber
.call    ebx ; MulDiv(x,x,x) ; MulDiv(x,x,x)
.text:6EB9DC74          add     eax, [esi+74h]
.text:6EB9DC76          push    8000      ; nDenominator
.text:6EB9DC79          push    dword ptr [edi+8] ; nNumerator
.text:6EB9DC7E          mov     [ebp+arg_8], eax
.text:6EB9DC81          mov     eax, [ebp+arg_4]
.text:6EB9DC84          push    dword ptr [eax] ; nNumber
.call    ebx ; MulDiv(x,x,x) ; MulDiv(x,x,x)
.text:6EB9DC89          add     eax, [esi+70h]
.text:6EB9DC8B          mov     ecx, [ebp+arg_0]
.text:6EB9DC91          mov     [ebp+var_8], eax
.text:6EB9DC94          mov     eax, [ebp+arg_8]
.text:6EB9DC97          mov     [ebp+var_4], eax
.text:6EB9DC9A          lea     eax, [ebp+var_8]
.text:6EB9DC9D          push    eax
.text:6EB9DC9E          lea     eax, [ebp+var_10]
.text:6EB9DCA1          push    eax
.text:6EB9DCA2          push    [ebp+arg_C]
.text:6EB9DCA5          call    ?DrawLine@Graphics@Gdiplus@@QAE?`v
    ↳ AW4Status@2@PBVPen@2@ABVPoint@2@1@Z ; Gdiplus::Graphics::DrawLine(Gdiplus::Pen const *, ↳
    ↳ Gdiplus::Point const &,Gdiplus::Point const &)
.text:6EB9DCAA
.text:6EB9DCAA loc_6EB9DCAA:           ; CODE XREF: CAnalogClock::_DrawHand( ↳
    ↳ Gdiplus::Graphics *,int,ClockHand const &,Gdiplus::Pen *)+9B
.text:6EB9DCAA           ; CAnalogClock::_DrawHand(Gdiplus:: ↳
    ↳ Graphics *,int,ClockHand const &,Gdiplus::Pen *)+9F
.pop    edi
.pop    esi
.pop    ebx
.leave
.ret    10h
.text:6EB9DCAE ?_DrawHand@CAnalogClock@AAE?`v
    ↳ AW4Status@Gdiplus@@PAVGraphics@3@HABUClockHand@@PAVPen@3@Z endp
.text:6EB9DCAE

```

We can see that *DrawLine()* arguments are dependent on result of *MulDiv()* function and a *table[]* table (name is mine), which has 8-byte elements (look at **LEA**'s second operand).

What is inside of *table[]*?

```

.text:6EB87890 ; int table[]
.text:6EB87890 table        dd 0
.text:6EB87894        dd 0FFFFE0C1h
.text:6EB87898        dd 344h
.text:6EB8789C        dd 0FFFFE0ECh
.text:6EB878A0        dd 67Fh
.text:6EB878A4        dd 0xFFFFE16Fh
.text:6EB878A8        dd 9A8h
.text:6EB878AC        dd 0xFFFFE248h
.text:6EB878B0        dd 0CB5h
.text:6EB878B4        dd 0FFFFE374h
.text:6EB878B8        dd 0F9Fh
.text:6EB878BC        dd 0FFFFE4F0h
.text:6EB878C0        dd 125Eh
.text:6EB878C4        dd 0FFFFE6B8h
.text:6EB878C8        dd 14E9h
...

```

## 8.4. HACKING WINDOWS CLOCK

It's referenced only from *DrawHand()* function at has 120 32-bit words or 60 32-bit pairs... wait, 60? Let's take a closer look at these values. First of all, I'll zap 6 pairs or 12 32-bit words with zeros, and then I'll put patched *TIMEDATE.CPL* into *C:\WINDOWS\SYSTEM32*. (You may need to set owner of the \**TIMEDATE.CPL*\* file to your primary user account (instead of *TrustedInstaller*), and also, boot in safe mode with command prompt so you can copy the file, which is usually locked.)

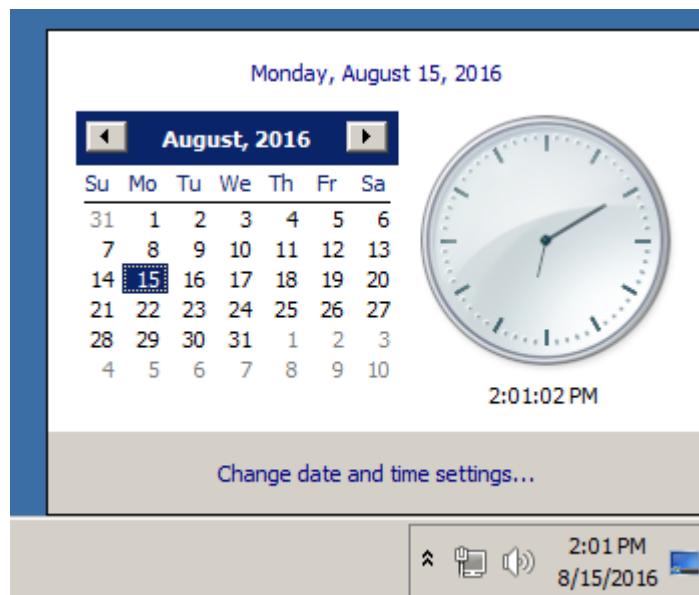


Figure 8.11: Attempt to run

Now when any hand is located at 0-5 seconds/minutes, it's invisible! However, opposite (shorter) part of second hand is visible and moving. When any hand is outside of this area, hand is visible as usual.

Let's take even closer look at the table in Mathematica. I have copied/pasted table from the *TIMEDATE.CPL* to a *tbl* file (480 bytes). We will take for granted the fact that these are signed values, because half of elements are below zero (0xFFFFE0C1h, etc.). If these values would be unsigned, they would be suspiciously huge.

```
In[]:= tbl = BinaryReadList["~/.../tbl", "Integer32"]

Out[]={0, -7999, 836, -7956, 1663, -7825, 2472, -7608, 3253, -7308, 3999, \
-6928, 4702, -6472, 5353, -5945, 5945, -5353, 6472, -4702, 6928, \
-4000, 7308, -3253, 7608, -2472, 7825, -1663, 7956, -836, 8000, 0, \
7956, 836, 7825, 1663, 7608, 2472, 7308, 3253, 6928, 4000, 6472, \
4702, 5945, 5353, 5353, 5945, 4702, 6472, 3999, 6928, 3253, 7308, \
2472, 7608, 1663, 7825, 836, 7956, 0, 7999, -836, 7956, -1663, 7825, \
-2472, 7608, -3253, 7308, -4000, 6928, -4702, 6472, -5353, 5945, \
-5945, 5353, -6472, 4702, -6928, 3999, -7308, 3253, -7608, 2472, \
-7825, 1663, -7956, 836, -7999, 0, -7956, -836, -7825, -1663, -7608, \
-2472, -7308, -3253, -6928, -4000, -6472, -4702, -5945, -5353, -5353, \
-5945, -4702, -6472, -3999, -6928, -3253, -7308, -2472, -7608, -1663, \
-7825, -836, -7956}

In[]:= Length[tbl]
Out[]= 120
```

Let's treat two consecutive 32-bit values as pair:

```
In[]:= pairs = Partition[tbl, 2]
Out[]={{0, -7999}, {836, -7956}, {1663, -7825}, {2472, -7608}, \
{3253, -7308}, {3999, -6928}, {4702, -6472}, {5353, -5945}, {5945, \
-5353}, {6472, -4702}, {6928, -4000}, {7308, -3253}, {7608, -2472}, \
{7825, -1663}, {7956, -836}, {8000, 0}, {7956, 836}, {7825, \
1663}, {7608, 2472}, {7308, 3253}, {6928, 4000}, {6472, \
4702}, {5945, 5353}, {5353, 5945}, {4702, 6472}, {3999, \
6928}, {3253, 7308}, {2472, 7608}, {1663, 7825}, {836, 7956}, {0, \
7999}, {-836, 7956}, {-1663, 7825}, {-2472, 7608}, {-3253, \
7308}, {-4000, 6928}, {-4702, 6472}, {-5353, 5945}, {-5945, \
5353}, {-6472, 4702}, {-6928, 3999}, {-7308, 3253}, {-7608, \
-1663}, {-7825, -836}, {-7956, -836}}
```

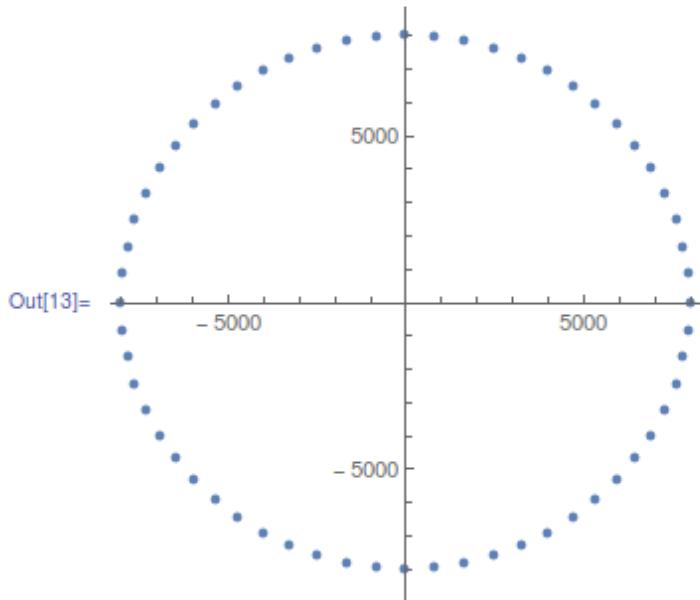
#### 8.4. HACKING WINDOWS CLOCK

```
2472}, {-7825, 1663}, {-7956, 836}, {-7999,
0}, {-7956, -836}, {-7825, -1663}, {-7608, -2472}, {-7308, -3253}, \
{-6928, -4000}, {-6472, -4702}, {-5945, -5353}, {-5353, -5945}, \
{-4702, -6472}, {-3999, -6928}, {-3253, -7308}, {-2472, -7608}, \
{-1663, -7825}, {-836, -7956}}
```

```
In[]:= Length[pairs]
Out[] = 60
```

Let's try to treat each pair as X/Y coordinate and draw all 60 pairs, and also first 15 pairs:

```
In[13]:= ListPlot[pairs, AspectRatio → Full, ImageSize → {300, 300}]
```



```
In[27]:= ListPlot[pairs[[1 ;; 15]], AspectRatio → Full, ImageSize → {300, 300}]
```

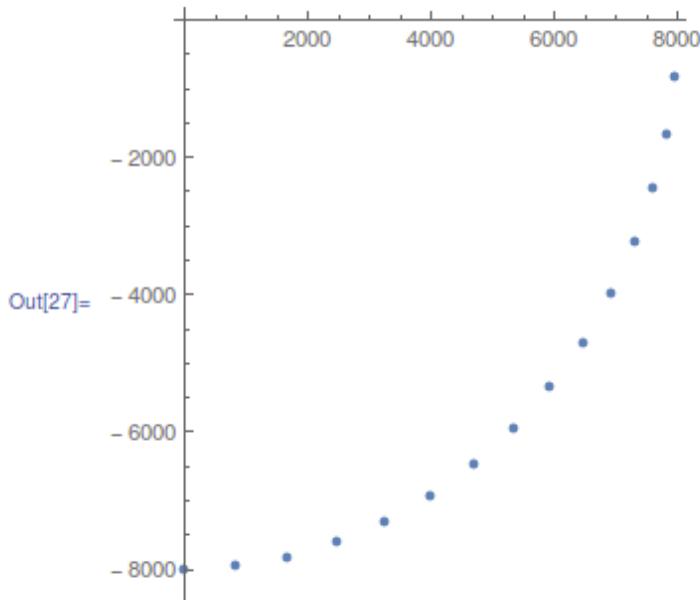


Figure 8.12: Mathematica

Now this is something! Each pair is just coordinate. First 15 pairs are coordinates for  $\frac{1}{4}$  of circle.

Perhaps, Microsoft developers precalculated all coordinates and put them into table.

Now I can understand why when I zapped 6 pairs, hands were invisible at that area: in fact, hands were drawn, they just had zero length, because hand started at 0:0 coordinate and ended there.

## 8.4. HACKING WINDOWS CLOCK

### The prank (practical joke)

Given all that, how would we force hands to go counterclockwise? In fact, this is simple, we need just to rotate the table, so each hand, instead of drawing at place of first second, would be drawing at place of 59th second.

I made the patcher a long time ago, at the very beginning of 2000s, for Windows 2000. Hard to believe, it still works for Windows 7, perhaps, the table hasn't been changed since then!

Patcher source code: [https://github.com/dennis714/random\\_notes/blob/master/timedate/time\\_pt.c](https://github.com/dennis714/random_notes/blob/master/timedate/time_pt.c).

Now I can see all hands goes backwards:

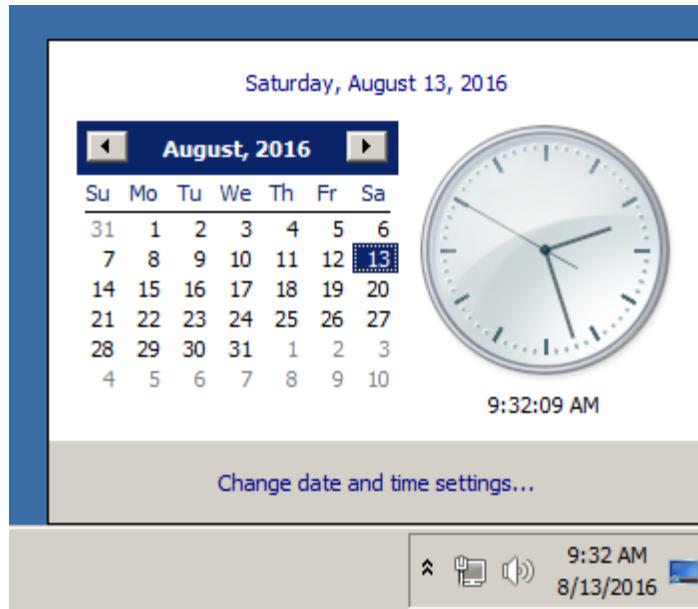


Figure 8.13: Now it works

Well, there is no animation in this article, but if you look closer, you can see, that hands are in fact shows correct time, but the whole clock face is rotated vertically, like we see it from the inside of clock.

### Windows 2000 leaked source code

So I did the patcher and then Windows 2000 source code has been leaked (I can't force you to trust me, though). Let's take a look on source code if that function and table.

The file is *win2k/private/shell/cpls/utc/clock.c*:

```
//
// Array containing the sine and cosine values for hand positions.
//
POINT rCircleTable[] =
{
    { 0,      -7999},
    { 836,    -7956},
    { 1663,   -7825},
    { 2472,   -7608},
    { 3253,   -7308},
    ...
    { -4702,  -6472},
    { -3999,  -6928},
    { -3253,  -7308},
    { -2472,  -7608},
    { -1663,  -7825},
    { -836,   -7956},
};

///////////////////////////////
//
```

## 8.5. DONGLES

```
// DrawHand
//
// Draws the hands of the clock.
//
////////////////////////////////////////////////////////////////
void DrawHand(
    HDC hDC,
    int pos,
    HPEN hPen,
    int scale,
    int patMode,
    PCLOCKSTR np)
{
    LPPOINT lppt;
    int radius;

    MoveTo(hDC, np->clockCenter.x, np->clockCenter.y);
    radius = MulDiv(np->clockRadius, scale, 100);
    lppt = rCircleTable + pos;
    SetROP2(hDC, patMode);
    SelectObject(hDC, hPen);

    LineTo( hDC,
            np->clockCenter.x + MulDiv(lppt->x, radius, 8000),
            np->clockCenter.y + MulDiv(lppt->y, radius, 8000) );
}
```

Now it's clear: coordinates has been precalculated as if clock face has height and width of  $2 \cdot 8000$ , and then it's rescaled to current clock face radius using *MulDiv()* function.

POINT structure<sup>8</sup> is a structure of two 32-bit values, first is x, second is y.

## 8.5 Dongles

The author of these lines, occasionally did software copy-protection **dongle** replacements, or “dongle emulators” and here are couple examples of how it’s happening.

About one of the cases about Rocket and Z3 that is not present here, you can read here: [http://yurichev.com/tmp/SAT\\_SMT\\_DRAFT.pdf](http://yurichev.com/tmp/SAT_SMT_DRAFT.pdf).

### 8.5.1 Example #1: MacOS Classic and PowerPC

Here is an example of a program for MacOS Classic <sup>9</sup>, for PowerPC. The company who developed the software product has disappeared a long time ago, so the (legal) customer was afraid of physical dongle damage.

While running without a dongle connected, a message box with the text “Invalid Security Device” appeared.

Luckily, this text string could easily be found in the executable binary file.

Let’s pretend we are not very familiar both with Mac OS Classic and PowerPC, but will try anyway.

IDA opened 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 the error message, we’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
```

<sup>8</sup>[https://msdn.microsoft.com/en-us/library/windows/desktop/dd162805\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/dd162805(v=vs.85).aspx)

<sup>9</sup>pre-UNIX MacOS

## 8.5. DONGLES

```
seg000:000C880C 40 82 00 40    bne      OK
seg000:000C8810 80 62 9F D8    lwz       %r3, TC_aInvalidSecurityDevice
...
...
```

Yes, this is PowerPC code.

The CPU is a very typical 32-bit [RISC](#) of 1990s era.

Each instruction occupies 4 bytes (just as in MIPS and ARM) and the names somewhat resemble MIPS instruction names.

`check1()` is a function name we'll give to it later. `BL` is *Branch Link* instruction, e.g., intended for calling subroutines.

The crucial point is the `BNE` instruction which jumps if the dongle protection check passes or not if an error occurs: then the address of the text string gets loaded into the `r3` register for the subsequent passing into a message box routine.

From the [Steve Zucker, SunSoft and Kari Karhi, IBM, *SYSTEM V APPLICATION BINARY INTERFACE: PowerPC Processor Supplement*, (1995)]<sup>10</sup> we will find out that the `r3` register is used for return values (and `r4`, in case of 64-bit values).

Another yet unknown instruction is `CLRLWI`. From [*PowerPC(tm) Microprocessor Family: The Programming Environments for 32-Bit Microprocessors*, (2000)]<sup>11</sup> we'll learn that this instruction does both clearing and loading. In our case, it clears the 24 high bits from the value in `r3` and puts them in `r0`, so it is analogical to `MOVZX` in x86 ([1.17.1 on page 202](#)), but it also sets the flags, so `BNE` can check them afterwards.

Let's take a look into the `check1()` function:

```
seg000:00101B40          check1: # CODE XREF: seg000:00063E7Cp
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 you can see in [IDA](#), that function is called from many places in the program, but only the `r3` register's value is checked after each call.

All this function does is to call the other function, so it is a [thunk function](#): there are function prologue and epilogue, but the `r3` register is not touched, so `check1()` returns what `check2()` returns.

`BLR`<sup>12</sup> looks like the return from the function, but since [IDA](#) does the function layout, we probably do not need to care about this.

Since it is a typical [RISC](#), it seems that subroutines are called using a [link register](#), just like in ARM.

The `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
```

<sup>10</sup>Also available as [http://yurichev.com/mirrors/PowerPC/elfspec\\_ppc.pdf](http://yurichev.com/mirrors/PowerPC/elfspec_ppc.pdf)

<sup>11</sup>Also available as [http://yurichev.com/mirrors/PowerPC/6xx\\_pem.pdf](http://yurichev.com/mirrors/PowerPC/6xx_pem.pdf)

<sup>12</sup>(PowerPC) Branch to Link Register

## 8.5. DONGLES

```

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

```

We are lucky again: some function names are left in the executable (debug symbols section)? Hard to say while we are not very familiar with the file format, maybe it is some kind of PE exports? ([6.5.2 on page 758](#)), like `.RBEFINDNEXT()` and `.RBEFINDFIRST()`.

Eventually these functions call other functions with names like `.GetNextDeviceViaUSB()`, `.USBSendPKT()`, so these are clearly dealing with an USB device.

## 8.5. DONGLES

There is even a function named `.GetNextEve3Device()` —sounds familiar, there was a Sentinel Eve3 dongle for ADB port (present on Macs) in 1990s.

Let's first take a look on how the r3 register is set before return, while ignoring everything else.

We know that a “good” r3 value has to be non-zero, zero r3 leads the execution flow to the message box with an error message.

There are two `li %r3, 1` instructions present in the function and one `li %r3, 0` (*Load Immediate*, i.e., loading a value into a register). The first instruction is at `0x001186B0`—and frankly speaking, it's hard to say what it means.

What we see next is, however, easier to understand: `.RBEFINDFIRST()` is called: if it fails, 0 is written into r3 and we jump to `exit`, otherwise another function is called (`check3()`)—if it fails too, `.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 a 16-bit value.

`B` (stands for *branch*) unconditional jump.

`BEQ` is the 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
```

## 8.5. DONGLES

```

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, 0xFEAO
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

```

## 8.5. DONGLES

```

seg000:00118890 38 60 00 01 li      %r3, 1
seg000:00118894 48 00 01 DC b       exit
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

```

## 8.5. DONGLES

```

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

```

## 8.5. DONGLES

```
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 `.RBREAD()`.

Perhaps, the function returns some values from the dongle, so they are compared here with some hard-coded variables using `CMPLWI`.

We also see that the r3 register is also filled before each call to `.RBREAD()` with one of these values: 0, 1, 8, 0xA, 0xB, 0xC, 0xD, 4, 5. Probably a memory address or something like that?

Yes, indeed, by googling these function names it is easy to find the Sentinel Eve3 dongle manual!

Perhaps we don't even have to learn any other PowerPC instructions: all this function does is just call `.RBREAD()`, compare its results with the constants and returns 1 if the comparisons are fine or 0 otherwise.

OK, all we've got is that `check1()` has always to return 1 or any other non-zero value.

But since we are not very confident in our knowledge of PowerPC instructions, we are going to be careful: we will patch the jumps in `check2()` at `0x001186FC` and `0x00118718`.

At `0x001186FC` we'll write bytes 0x48 and 0 thus converting the `BEQ` instruction in an `B` (unconditional jump): we can spot its opcode in the code without even referring to [PowerPC(tm) Microprocessor Family: The Programming Environments for 32-Bit Microprocessors, (2000)]<sup>13</sup>.

At `0x00118718` we'll write 0x60 and 3 zero bytes, thus converting it to a `NOP` instruction: Its opcode we could spot in the code too.

And now it all works without a dongle connected.

In summary, such small modifications can be done with `IDA` and minimal assembly language knowledge.

<sup>13</sup>Also available as [http://yurichev.com/mirrors/PowerPC/6xx\\_pem.pdf](http://yurichev.com/mirrors/PowerPC/6xx_pem.pdf)

## 8.5.2 Example #2: SCO OpenServer

An ancient software for SCO OpenServer from 1997 developed by a company that disappeared a long time ago.

There is a special dongle driver to be installed in the system, that contains the following text strings: "Copyright 1989, Rainbow Technologies, Inc., Irvine, CA" and "Sentinel Integrated Driver Ver. 3.0".

After the installation of the driver in SCO OpenServer, these device files appear in the /dev filesystem:

```
/dev/rbsl8
/dev/rbsl9
/dev/rbsl10
```

The program reports an error without dongle connected, but the error string cannot be found in the executables.

Thanks to [IDA](#), it is easy to load the COFF executable used in SCO OpenServer.

Let's also try to find "rbsl" string 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, 0FFFFFFFh
.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
```

## 8.5. DONGLES

```
.text:00022B45      add    esp, 0Ch
.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      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      loc_22B6B: ; CODE XREF: SSQC+47j
.text:00022B6B      mov    eax, 0FFFFFFFh
.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      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      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      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      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%$"
.text:00022BBB7      push   eax
.text:00022BBE      call   nl_sprintf
```

## 8.5. DONGLES

```
.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 needs to communicate with the driver somehow.

The only place where the `SSQC()` function is 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()` can be called from at least 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 we gave to the labels:
.data:004016C0 answers1      dd 6B05h      ; DATA XREF: sys_info+E7r
.data:004016C4      dd 3D87h      ; DATA XREF: sys_info+F2r
.data:004016C8 answers2      dd 3Ch       ; DATA XREF: sys_info+F2r
```

## 8.5. DONGLES

```
.data:004016CC          dd 832h
.data:004016D0 _C_and_B db 0Ch           ; DATA XREF: sys_info+BAr
.data:004016D0          db 0Kr           ; 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, 0xFFFFFFFFh
.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
```

## 8.5. DONGLES

```
.text:0000D6F7          pop     ebp
.text:0000D6F8          retn
.text:0000D6F8  sys_info    endp
```

“3C” and “3E” sound familiar: there was a Sentinel Pro dongle by Rainbow with no memory, providing only one crypto-hashing secret function.

You can read a short description of what hash function is here: [2.10 on page 465](#).

But let's get back to the program.

So the program can only check the presence or absence of a connected dongle.

No other information can be written to such dongle, as it has no memory. The two-character codes are commands (we can see how the commands are handled in the `SSQC()` function) and all other strings are hashed inside the dongle, being transformed into a 16-bit number. The algorithm was secret, so it was not possible to write a driver replacement or to remake the dongle hardware that would emulate it perfectly.

However, it is always possible to intercept all accesses to it and to find what constants the hash function results are compared to.

But we need to say that it is possible to build a robust software copy protection scheme based on secret cryptographic hash-function: let it encrypt/decrypt the data files your software uses.

But let's get back to the code.

Codes 51/52/53 are used for LPT printer port selection. 3x/4x are used for “family” selection (that's how Sentinel Pro dongles are differentiated from each other: more than one dongle can be connected to a 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 the global variable `ctl_model`.

Another text string that gets passed is “PRESS ANY KEY TO CONTINUE: ”, but the result is not checked. Hard to say why, probably by mistake <sup>14</sup>.

Let's see where the value from the global variable `ctl_mode` is used.

One such place 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 aIeCvulnv0kgT_ ; "Ie-cvulnvV\\b0KG]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
```

<sup>14</sup>What a strange feeling: to find bugs in such ancient software.

## 8.5. DONGLES

```
.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 to a decryption routine and printed.

The error string decryption routine seems a simple [xor-ing](#):

```
.text:0000A43C err_warn          proc near               ; CODE XREF: prep_sys+E8p
.text:0000A43C
.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]
```

## 8.5. DONGLES

```
.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 we were unable to find the error messages in the executable files, because they are encrypted (which is popular practice).

Another call to the `SSQ()` hashing function passes the “offln” string to it and compares the result with `0xFE81` and `0x12A9`.

If they don't match, it works with some `timer()` function (maybe waiting for a poorly connected dongle to be reconnected and check again?) and then decrypts another error message to dump.

```
.text:0000DA55 loc_DA55:                                ; CODE XREF: sync_sys+24Cj
.text:0000DA55          push    offset a0ffln   ; "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 we 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
```

## 8.5. DONGLES

```

.text:0000D9B0          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 we 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, 1Fh, 7, 1Eh, 1Ah

```

Bypassing the dongle is pretty straightforward: just patch all jumps after the relevant `CMP` instructions.

Another option is to write our own SCO OpenServer driver, containing a table of questions and answers, all of those which present in the program.

### Decrypting error messages

By the way, we can also try to decrypt all error messages. The algorithm that is located in the `err_warn()` function is very simple, indeed:

Listing 8.3: Decryption 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 is supplied to the decryption 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 we gave to label manually:
.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

```

## 8.5. DONGLES

```
.text:0000D9C6          push    ecx
.text:0000D9C7          push    18h
.text:0000D9C9          call    err_warn
```

The algorithm is a simple **xoring**: each byte is xored with a key, but the key is increased by 3 after the processing of each byte.

We can write a simple Python script to check our hypothesis:

Listing 8.4: 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 the decrypted message.

There are also other encrypted messages with their corresponding keys. But needless to say, it is possible to decrypt them without their keys. First, we can see that the key is in fact a byte. It is because the core decryption instruction (**XOR**) works on byte level. The key is located in the **ESI** register, but only one byte part of **ESI** is used. Hence, a key may be greater than 255, but its value is always to be rounded.

As a consequence, we can just try brute-force, trying all possible keys in the 0..255 range. We are also going to skip the messages that has unprintable characters.

Listing 8.5: 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, 0x5C,
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, 0x50,
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):
```

## 8.5. DONGLES

```
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 get:

Listing 8.6: 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#gghtme?!#!'!#
message #3
key= 7 value= Bk<waoqNUpu$`yreoa\wpmpusj,bkIjh
key= 8 value= Mj?vfnr0jqv%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 is some garbage, but we can quickly find the English-language messages!

By the way, since the algorithm is a simple xoring encryption, the very same function can be used to encrypt messages. If needed, we can encrypt our own messages, and patch the program by inserting them.

### 8.5.3 Example #3: MS-DOS

Another very old software for MS-DOS from 1995 also developed by a company that disappeared a long time ago.

In the pre-DOS extenders era, all the software for MS-DOS mostly relied on 16-bit 8086 or 80286 CPUs, so the code was 16-bit en masse.

The 16-bit code is mostly same as you already saw in this book, but all registers are 16-bit and there are less instructions available.

The MS-DOS environment has no system drivers, and any program can deal with the bare hardware via ports, so here you can see the OUT / IN instructions, which are present in mostly in drivers in our times (it is impossible to access ports directly in user mode on all modern OSes).

Given that, the MS-DOS program which works with a dongle has to access the LPT printer port directly.

So we can just search for such instructions. And yes, here they are:

```
seg030:0034          out_port proc far ; CODE XREF: sent_prot+22p
seg030:0034                      ; sent_prot+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]
```

## 8.5. DONGLES

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

```

## 8.5. DONGLES

```

seg030:00AD EB 40          jmp     short loc_35A4F
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

```

## 8.5. DONGLES

```

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

```

This is again a Sentinel Pro “hashing” dongle as in the previous example. It is noticeably because text strings are passed here, too, and 16 bit values are returned and compared with others.

So that is how Sentinel Pro is accessed via ports.

The output port address is usually 0x378, i.e., the printer port, where the data to the old printers in pre-USB era was passed to.

The port is uni-directional, because when it was developed, no one imagined that someone will need to transfer information from the printer <sup>15</sup>.

The only way to get information from the printer is a status register on port 0x379, which contains such bits as “paper out”, “ack”, “busy”—thus the printer may signal to the host computer if it is ready or not and if paper is present in it.

So the dongle returns information from one of these bits, one bit at each iteration.

`_in_port_2` contains the address of the status word (0x379) and `_in_port_1` contains the control register address (0x37A).

It seems that the dongle returns information via the “busy” flag at `seg030:00B9`: each bit is stored in the `DI` register, which is returned at the end of the function.

What do all these bytes sent to output port mean? Hard to say. Perhaps, 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 the 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+_0   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

```

<sup>15</sup>If we consider Centronics only. The following IEEE 1284 standard allows the transfer of information from the printer.

## 8.5. DONGLES

```

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
loc_35B0A: ; CODE XREF: check_dongle+1Fj
seg030:01AA
; check_dongle+5Ej
seg030:01AA 33 C0      xor    ax, ax
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 can be called very frequently, e.g., before the execution of each important software feature, and accessing the dongle is generally slow (because of the slow printer port and also slow [MCU](#) in the dongle), they probably added a way to skip some dongle checks, by checking the current time in the `biostime()` function.

The `get_rand()` function uses the 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

```

## 8.6. “QR9”: RUBIK’S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

```
seg030:01E0 CB          retf  
seg030:01E0      get_rand endp
```

So the text string is selected randomly, passed into the dongle, and then the result of the hashing is compared with the correct value.

The text strings seem to be constructed randomly as well, during software development.

And this 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 ..."  
seg033:08A3 1E              push  ds  
seg033:08A4 68 60 E9      push  offset byte_6C7E0 ; dest  
seg033:08A7 9A CD 64 00+  call  _strcat
```

Bypassing the dongle 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
```

The observant reader might recall that the `strcpy()` C function usually requires two pointers in its arguments, but we see that 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
```

This is related to MS-DOS’ memory model. You can read more about it here: [10.6 on page 990](#).

So as you may see, `strcpy()` and any other function that take pointer(s) in arguments work with 16-bit pairs.

Let’s get back to our example. `DS` is currently set to the data segment located in the executable, that is where the text string is stored.

In the `sent_pro()` function, each byte of the string is loaded at

`seg030:00EF`: the `LES` instruction loads the `ES:BX` pair simultaneously from the passed argument.

The `MOV` at `seg030:00F5` loads the byte from the memory at which the `ES:BX` pair points.

## 8.6 “QR9”: Rubik’s cube inspired amateur crypto-algorithm

Sometimes amateur cryptosystems appear to be pretty bizarre.

The author of this book was once asked to reverse engineer an amateur cryptoalgorithm of some data encryption utility, the source code for which was lost<sup>16</sup>.

<sup>16</sup>He also got permission from the customer to publish the algorithm’s details

## 8.6. “QR9”: RUBIK’S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

Here is the listing exported from [IDA](#) for the original encryption 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]
.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]
```

## 8.6. “QR9”: RUBIK’S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

```

.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, 0FFFFFFFh
.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: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

```

## 8.6. “QR9”: RUBIK’S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

```

.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, 0FFFFFFFh
.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 ; ===== S U B R O U T I N E =====
.text:00541150
.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

```

## 8.6. “QR9”: RUBIK’S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

```

.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, 0FFFFFFFh
.text:0054119E          jg     short loc_54118B
.text:005411A0          inc     ebx
.text:005411A1          cmp     ebx, 8
.text:005411A1          jl     short loc_541186
.text:005411A4          pop     edi
.text:005411A6          pop     esi
.text:005411A7          pop     ebp
.text:005411A8          pop     ebx
.text:005411A9          pop     esp, 40h
.text:005411AA          add
.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 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'

```

## 8.6. “QR9”: RUBIK’S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

```

.jtext:005411DA          jg      short next_character_in_password
.jtext:005411DC          movsx   ecx, al
.jtext:005411DF          sub     ecx, 'a'
.jtext:005411E2          cmp     ecx, 24
.jtext:005411E5          jle    short skip_subtracting
.jtext:005411E7          sub     ecx, 24
.jtext:005411EA
.jtext:005411EA skip_subtracting: ; CODE XREF: rotate_all_with_password+35
.jtext:005411EA           mov     eax, 55555556h
.jtext:005411EF           imul   ecx
.jtext:005411F1           mov     eax, edx
.jtext:005411F3           shr     eax, 1Fh
.jtext:005411F6           add     edx, eax
.jtext:005411F8           mov     eax, ecx
.jtext:005411FA           mov     esi, edx
.jtext:005411FC           mov     ecx, 3
.jtext:00541201           cdq
.jtext:00541202           idiv   ecx
.jtext:00541204           sub     edx, 0
.jtext:00541207           jz    short call_rotate1
.jtext:00541209           dec     edx
.jtext:0054120A           jz    short call_rotate2
.jtext:0054120C           dec     edx
.jtext:0054120D           jnz    short next_character_in_password
.jtext:0054120F           test   ebx, ebx
.jtext:00541211           jle    short next_character_in_password
.jtext:00541213           mov     edi, ebx
.jtext:00541215
.jtext:00541215 call_rotate3: ; CODE XREF: rotate_all_with_password+6F
.jtext:00541215           push   esi
.jtext:00541216           call   rotate3
.jtext:0054121B           add    esp, 4
.jtext:0054121E           dec    edi
.jtext:0054121F           jnz    short call_rotate3
.jtext:00541221           jmp    short next_character_in_password
.jtext:00541223
.jtext:00541223 call_rotate2: ; CODE XREF: rotate_all_with_password+5A
.jtext:00541223           test   ebx, ebx
.jtext:00541225           jle    short next_character_in_password
.jtext:00541227           mov     edi, ebx
.jtext:00541229 loc_541229: ; CODE XREF: rotate_all_with_password+83
.jtext:00541229           push   esi
.jtext:0054122A           call   rotate2
.jtext:0054122F           add    esp, 4
.jtext:00541232           dec    edi
.jtext:00541233           jnz    short loc_541229
.jtext:00541235           jmp    short next_character_in_password
.jtext:00541237
.jtext:00541237 call_rotate1: ; CODE XREF: rotate_all_with_password+57
.jtext:00541237           test   ebx, ebx
.jtext:00541239           jle    short next_character_in_password
.jtext:0054123B           mov     edi, ebx
.jtext:0054123D loc_54123D: ; CODE XREF: rotate_all_with_password+97
.jtext:0054123D           push   esi
.jtext:0054123E           call   rotate1
.jtext:00541243           add    esp, 4
.jtext:00541246           dec    edi
.jtext:00541247           jnz    short loc_54123D
.jtext:00541249 next_character_in_password: ; CODE XREF: rotate_all_with_password+26
.jtext:00541249           ; rotate_all_with_password+2A ...
.jtext:00541249           mov     al, [ebp+1]
.jtext:0054124C           inc    ebp
.jtext:0054124D           test   al, al
.jtext:0054124F           jnz    loop_begin
.jtext:00541255           pop    edi
.jtext:00541256           pop    esi
.jtext:00541257           pop    ebx

```

## 8.6. “QR9”: RUBIK’S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

```
.text:00541258 ; CODE XREF: rotate_all_with_password+A
.text:00541258 exit:          pop     ebp
.text:00541258                 retn
.text:00541259 rotate_all_with_password endp
.text:00541259
.text:0054125A align 10h
.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 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 ; ===== 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    __strupd
.text:005412B9 push    eax           ; Str
```

## 8.6. "QR9": RUBIK'S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

```

.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
.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

```

## 8.6. "QR9": RUBIK'S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

```

.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, 0FFFFFC0h ; 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 encrypted 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

```

## 8.6. “QR9”: RUBIK’S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

```

.text:005413FF align 10h
.text:00541400 ; ===== S U B R O U T I N E =====
.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 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 encrypted!\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

```

## 8.6. "QR9": RUBIK'S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

```
.text:00541489          ; CODE XREF: decrypt_file+75
.text:00541489 loc_541489:
.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 were given by me during the analysis.

Let's start from the top. Here is a function that takes 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 the file and report if an error occurs:

```
.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 loc_541348:
```

Get the 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
```

## 8.6. “QR9”: RUBIK’S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

```
.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 the file size aligned on a 64-byte boundary. This is because this cryptographic algorithm works with only 64-byte blocks. The operation is pretty straightforward: divide the file size by 64, forget about the remainder and add 1, then multiply by 64. The following code removes the remainder as if the value has already been divided by 64 and adds 64. It is almost the same.

```
.text:00541369 mov     esi, [esp+2Ch+Str]
; reset all lowest 6 bits
.text:0054136D and    esi, 0FFFFFFC0h
; align size to 64-byte border
.text:00541370 add    esi, 40h
```

Allocate buffer with aligned size:

```
.text:00541373         push    esi          ; Size
.text:00541374         call    _malloc
```

Call `memset()`, e.g., clear the allocated buffer<sup>17</sup>.

```
.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    stobs       ; memset (buffer, 0, aligned_size)
```

Read file via the 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 a buffer, buffer size (aligned) and a 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 the output file. By the way, the developer forgot to check if it has been created correctly! The file opening result is being checked, though.

<sup>17</sup> `malloc() + memset()` could be replaced by `calloc()`

## 8.6. “QR9”: RUBIK’S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

```
.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
```

The 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 the 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 the encrypted 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 the file and free the 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 the 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=f.tell (f);
    fseek (f, 0, SEEK_SET);

    flen_aligned=(flen&0xFFFFFC0)+0x40;

    buf=(BYTE*)malloc (flen_aligned);
```

## 8.6. "QR9": RUBIK'S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

```

        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);
    };
}

```

The decryption 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

```

## 8.6. "QR9": RUBIK'S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

```
.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 the signature is absent:

```
.text:00541477          push   offset aFileIsNotCrypt ; "File is not encrypted!\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          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 the 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)
    {
```

## 8.6. “QR9”: RUBIK’S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

```

        printf ("Cannot open input file!\n");
        return;
    };

    fseek (f, 0, SEEK_END);
    flen=f.tell (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 encrypted!\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 loc_54126A:

```

This fragment of code copies a part of the input buffer to an internal array we later name “cube64”. The size is in the `ECX` register. `MOVSD` stands for *move 32-bit dword*, so, 16 32-bit dwds 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 encrypted contents back from “cube64” to buffer:

## 8.6. “QR9”: RUBIK’S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

```
.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 = amount of encrypted data.
.text:0054129F          rep     movsd
```

If **EBP** is not bigger than the size input argument, then continue to the 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 in function **rotate\_all\_with\_password()**. It takes two arguments: password string and a number.

In **crypt()**, the number 1 is used, and in the **decrypt()** function (where **rotate\_all\_with\_password()** function is called too), the 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 the current character in the 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          loop_begin:
```

Call **tolower()**, a 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
```

## 8.6. “QR9”: RUBIK’S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

Hmm, if the password has non-Latin character, it is skipped! Indeed, when we run the encryption utility and try non-Latin characters in the 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 the value of “a” (97) from the 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 implies that the 26 Latin alphabet symbols become values in the range of 0..23, (24 in total).

```
.text:005411EA          ; CODE XREF: rotate_all_with_password+35  
.text:005411EA skip_subtracting:
```

This is actually division via multiplication. You can read more about it in the “Division using multiplication” section ([3.9 on page 496](#)).

The code actually divides the password character’s 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 the 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 ; .. if 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 the remainder is 2, call `rotate3()`. EDI is the second argument of the `rotate_all_with_password()` function. As we already noted, 1 is for the encryption operations and 3 is for the decryption. So, here is a loop. When encrypting, rotate1/2/3 are to 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
```

## 8.6. “QR9”: RUBIK’S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

```
.text:00541227          mov     edi, ebx
.text:00541229          push    esi
.text:00541229 loc_541229:
.text:00541229          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 the next character from the password string.

```
.text:00541249 next_character_in_password:
.text:00541249          mov    al, [ebp+1]
```

Increment the character pointer in the 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 the 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;
            };
        }
    }
}
```

## 8.6. “QR9”: RUBIK’S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

```

    );
    p++;
};

}

```

Now let's go deeper and investigate the rotate1/2/3 functions. Each function calls another two functions. We eventually will name them `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 cube64 array: `arg_4 + arg_0 * 8`. Then shift a byte from the array by `arg_8` bits right. Isolate the 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
```

The value in the `DL` is 1 here. It gets shifted left by `arg_8`. For example, if `arg_8` is 4, the value in the `DL` register is to be `0x10` or `1000b` in binary form.

```
.text:00541017             shl    dl, cl
.text:00541019             mov    cl, cube64[eax+esi*8]
```

Get a bit from array and explicitly set it.

```
.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]
```

## 8.6. “QR9”: RUBIK’S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

...invert DL. For example, if DL’s state after the shift is 0x10 or 0b1000, there is 0xEF to be after the NOT instruction (or 0b11101111b).

```
.text:00541034          not     dl
```

This instruction clears the bit, in other words, it saves all bits in CL which are also set in DL except those in DL which are cleared. This implies that if DL is 11101111b in binary form, all bits are to be saved except the 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 the specific bit in the array, or sets it otherwise.

We also know that the array’s size is 64. The first two arguments both in the `set_bit()` and `get_bit()` functions could be seen as 2D coordinates. Then the array is to be an 8\*8 matrix.

Here is a C representation of what we know up to now:

```
#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 shift, int bit)
{
    if (bit)
        SET_BIT (cube[x][y], 1<<shift);
    else
        REMOVE_BIT (cube[x][y], 1<<shift);
};

bool 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 the rotate1/2/3 functions.

```
.text:00541070 rotate1      proc near
.text:00541070
```

Internal array allocation in the local stack, with size of 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 the internal array:

```
.text:0054107D          lea     ebx, [esp+50h+internal_array_64]
.text:00541081
```

## 8.6. “QR9”: RUBIK’S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

Here we have two nested loops:

```
.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

; increment internal array pointer by 8 at each loop 1 iteration
.text:00541098     add     ebx, 8
.text:0054109B     cmp     edi, 8
.text:0054109E     jl      short first_loop1_begin
```

...we see that both loops' counters are in the range of 0..7. Also they are used as the first and second argument for the `get_bit()` function. The third argument to `get_bit()` is the only argument of `rotate1()`. The return value from `get_bit()` is placed in the internal array.

Prepare a pointer to the 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, 0FFFFFFFh
.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 the contents of the internal array to the cube global array via the `set_bit()` function, *but* in a different order! Now the counter of the first loop is in the range of 7 to 0, **decrementing** at each iteration!

The C code representation looks like:

```
void rotate1 (int v)
{
    bool tmp[8][8]; // internal array
    int i, j;

    for (i=0; i<8; i++)
```

## 8.6. "QR9": RUBIK'S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

```

        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 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, 0FFFFFFFh
.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

```

## 8.6. “QR9”: RUBIK’S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

It is almost the same, except the order of the arguments of the `get_bit()` and `set_bit()` is different.  
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 the `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 a 3D cube of size 8\*8\*8, where each element is a bit, `get_bit()` and `set_bit()` take just the coordinates of a bit as input.

The `rotate1/2/3` functions are in fact rotating all bits in a specific plane. These three functions are one for each cube side and the `v` argument sets the plane in the range of 0..7.

Maybe the algorithm's author was thinking of a 8\*8\*8 Rubik's cube <sup>18</sup>?!

Yes, indeed.

Let's look closer into the `decrypt()` function, here is its rewritten version:

```
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 as for `crypt()`, but the password string is reversed by the `strrev()` standard C function and `rotate_all()` is called with argument 3.

This implies that in case of decryption, each corresponding `rotate1/2/3` call is to be performed thrice.

<sup>18</sup>[wikipedia](#)

<sup>19</sup>[MSDN](#)

## 8.6. “QR9”: RUBIK’S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

This is almost as in Rubik’s cube! If you want to get back, do the same in reverse order and direction! If you want to undo the effect of rotating one place in clockwise direction, rotate it once in counter-clockwise direction, or thrice in clockwise direction.

`rotate1()` is apparently for rotating the “front” plane. `rotate2()` is apparently for rotating the “top” plane. `rotate3()` is apparently for rotating the “left” plane.

Let’s get back to the core of the `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 a side (one of three) and a plane (one of 8).  $3 \times 8 = 24$ , that is why two the last two characters of the Latin alphabet are remapped to fit an alphabet of exactly 24 elements.

The algorithm is clearly weak: in case of short passwords you can see that in the encrypted file there are the original bytes of the original file in a binary file editor.

Here is the whole source code reconstructed:

```
#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++) rotate_f (quotient); break;
                case 1: for (int i=0; i<v; i++) rotate_t (quotient); break;
                case 2: for (int i=0; i<v; i++) rotate_l (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);
    }
}

```

## 8.6. "QR9": RUBIK'S CUBE INSPIRED AMATEUR CRYPTO-ALGORITHM

```

        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=f.tell (f);
    fseek (f, 0, SEEK_SET);

    flen_aligned=(flen&0xFFFFFFFFC0)+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;

```

## 8.7. ENCRYPTED DATABASE CASE #1

```
f=fopen(fin, "rb");

if (f==NULL)
{
    printf ("Cannot open input file!\n");
    return;
};

fseek (f, 0, SEEK_END);
flen=f.tell (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 encrypted!\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;
};
```

## 8.7 Encrypted database case #1

(This part has been first appeared in my blog at 26-Aug-2015. Some discussion: <https://news.ycombinator.com/item?id=10128684>.)

## 8.7.1 Base64 and entropy

I've got the XML file containing some encrypted data. Perhaps, it's related to some orders and/or customers information.

```
<?xml version = "1.0" encoding = "UTF-8"?>
<Orders>
    <Order>
        <OrderID>1</OrderID>
        <Data>yjmjhXUbhB/5MV45chPsXZWAJwIh1S0aD9lFn3XuJMSxJ3/E+UE3hsnH</Data>
    </Order>
    <Order>
        <OrderID>2</OrderID>
        <Data>0KGe/wnypFBjsy+U0C2P9fC5nDZP3XDZLMPCRaiBw90jIk6Tu5U=</Data>
    </Order>
    <Order>
        <OrderID>3</OrderID>
        <Data>mqkXfdzvQKvEArdzh+zD9oETVGBFvcTBLs2ph1b5bYddExzp</Data>
    </Order>
    <Order>
        <OrderID>4</OrderID>
        <Data>FCx6JhIDqnESyT3HAePyE1BJ3cJd7wCk+APCRUeuNtZdpCvQ2MR/7kLXtfUHuA==</Data>
    </Order>
...
...
```

The file is available [here](#).

This is clearly base64-encoded data, because all strings consisting of Latin characters, digits, plus (+) and slash (/) symbols. There can be 1 or 2 padding symbols (=), but they are never occurred in the middle of string. Keeping in mind these base64 properties, it's very easy to recognize them.

Let's decode them and calculate entropies ([9.2 on page 945](#)) of these blocks in Wolfram Mathematica:

```
In[]:= ListOfBase64Strings =
  Map[First#[[3]] &, Cases[Import["encrypted.xml"], XMLElement["Data", _, _], Infinity]];

In[]:= BinaryStrings =
  Map[ImportString[#, {"Base64", "String"}] &, ListOfBase64Strings];

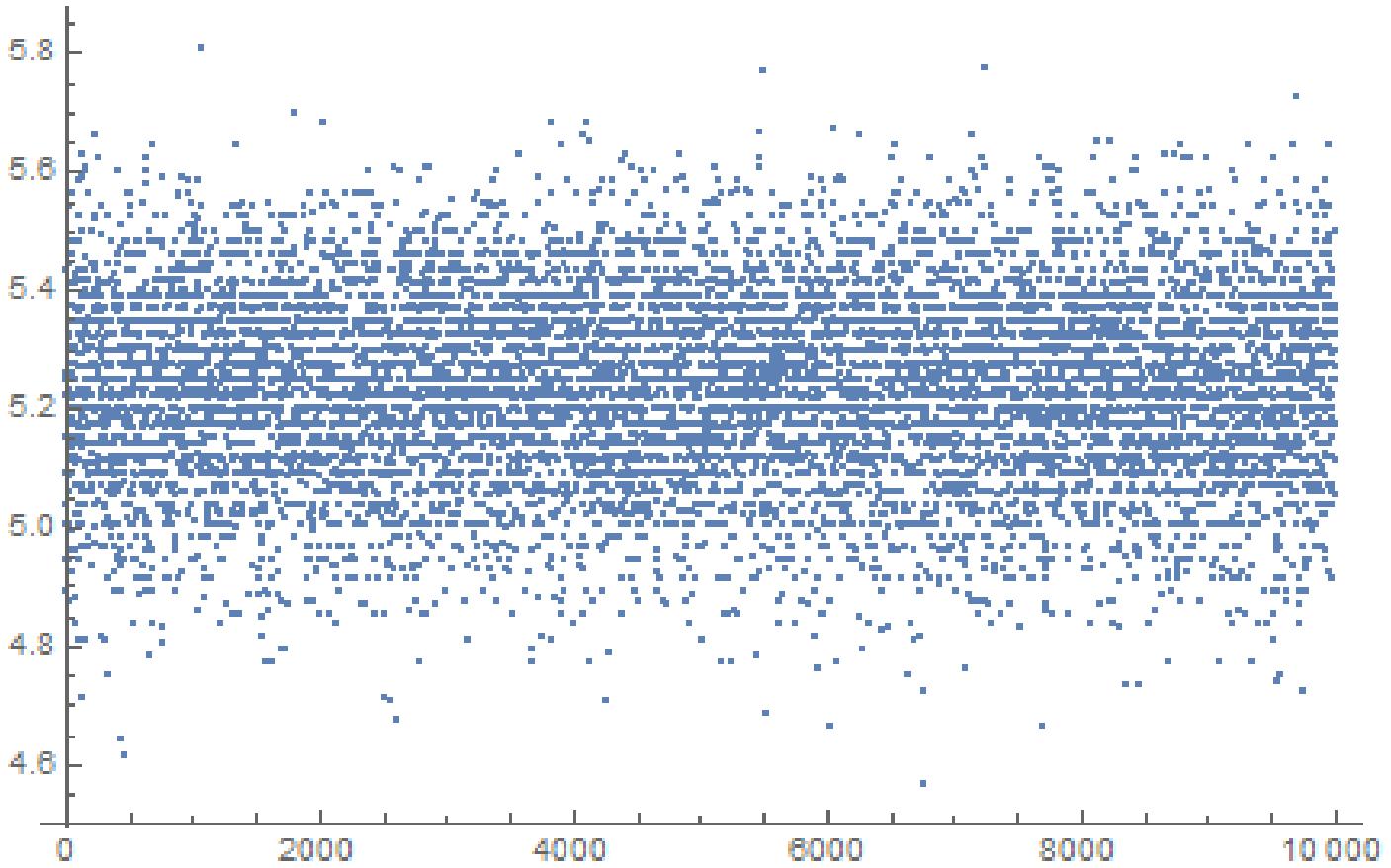
In[]:= Entropies = Map[N[Entropy[2, #]] &, BinaryStrings];

In[]:= Variance[Entropies]
Out[] = 0.0238614
```

Variance is low. This means the entropy values are not very different from each other. This is visible on graph:

```
In[]:= ListPlot[Entropies]
```

## 8.7. ENCRYPTED DATABASE CASE #1

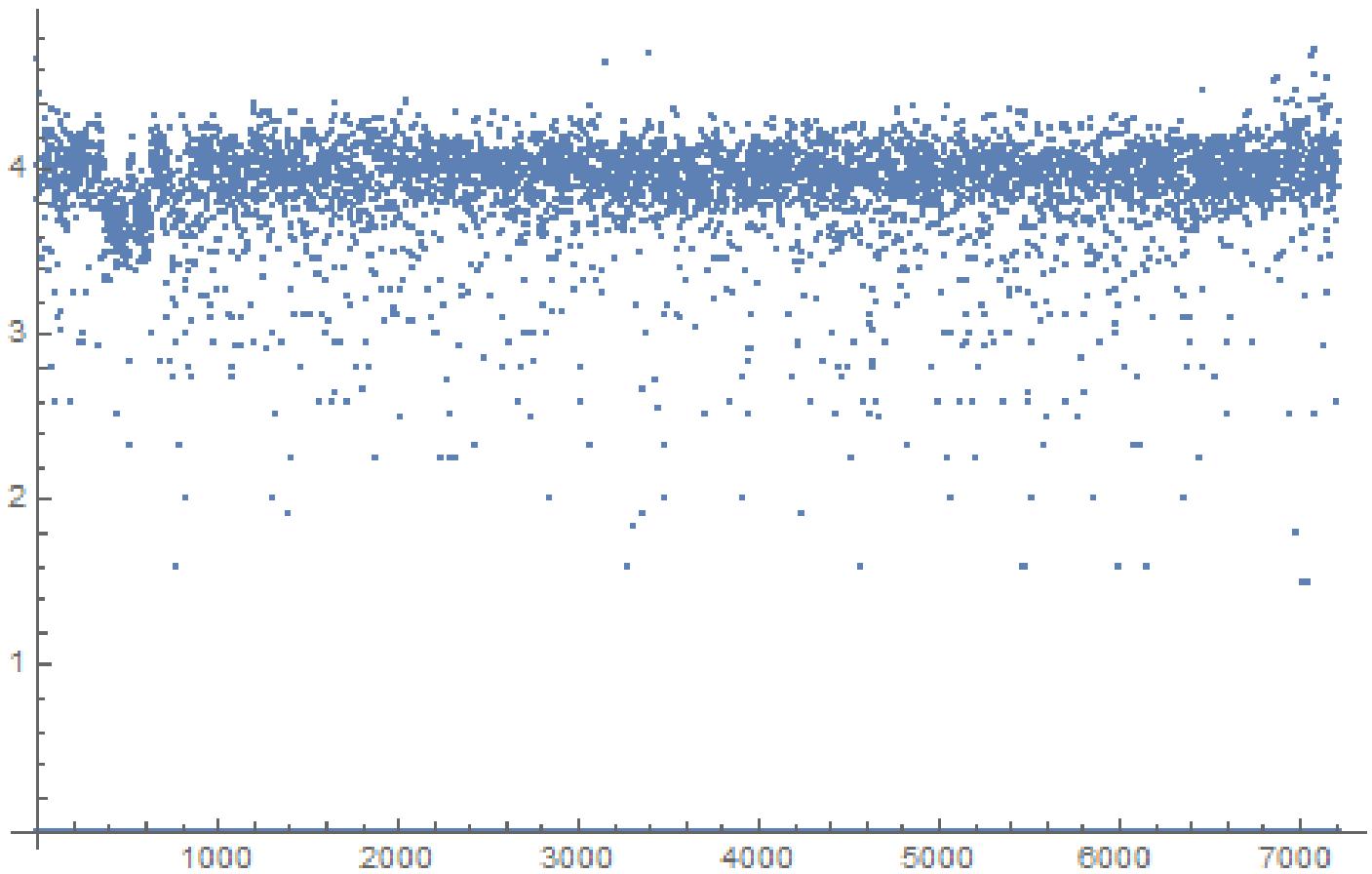


Most values are between 5.0 and 5.4. This is a sign that the data is compressed and/or encrypted.

To understand variance, let's calculate entropies of all lines in Conan Doyle's *The Hound of the Baskervilles* book:

```
In[]:= BaskervillesLines = Import["http://www.gutenberg.org/cache/epub/2852/pg2852.txt", "List"];
In[]:= EntropiesT = Map[N[Entropy[2, #]] &, BaskervillesLines];
In[]:= Variance[EntropiesT]
Out[] = 2.73883
In[]:= ListPlot[EntropiesT]
```

## 8.7. ENCRYPTED DATABASE CASE #1



Most values are gathered around value of 4, but there are also values which are smaller, and they are influenced final variance value.

Perhaps, shortest strings has smaller entropy, let's take short string from the Conan Doyle's book:

```
In[]:= Entropy[2, "Yes, sir."] // N  
Out[] = 2.9477
```

Let's try even shorter:

```
In[]:= Entropy[2, "Yes"] // N  
Out[] = 1.58496  
  
In[]:= Entropy[2, "No"] // N  
Out[] = 1.
```

### 8.7.2 Is it compressed?

OK, so our data is compressed and/or encrypted. Is it compressed? Almost all data compressors put some header at the start, signature, or something like that. As we can see, there are no consistent data at the start. It's still possible that this is DIY handmade data compressor, but they are very rare. Handmade cryptoalgorithm is very easy implement because it's very easy to make it work. Even primitive keyless cryptosystems like `memfrob()`<sup>20</sup> and ROT13 working fine without errors. It's a challenge to write data compressor from scratch using only fantasy and imagination in a way so it will have no evident bugs. Some programmers implements data compression functions by reading textbooks, but this is also rare. The most popular two ways are: 1) just take open-source library like zlib; 2) copy&paste something from somewhere. Open-source data compressions algorithms usually puts some kind of header, and so are popular code snippets from the sites like <http://www.codeproject.com/>.

<sup>20</sup><http://linux.die.net/man/3/memfrob>

### 8.7.3 Is it encrypted?

All major data encryption algorithms process data in blocks. DES—8 bytes, AES—16 bytes. If the input buffer is not divided evenly by block, it's padded, so encrypted data is aligned by cryptoalgorithm's block size. This is not our case.

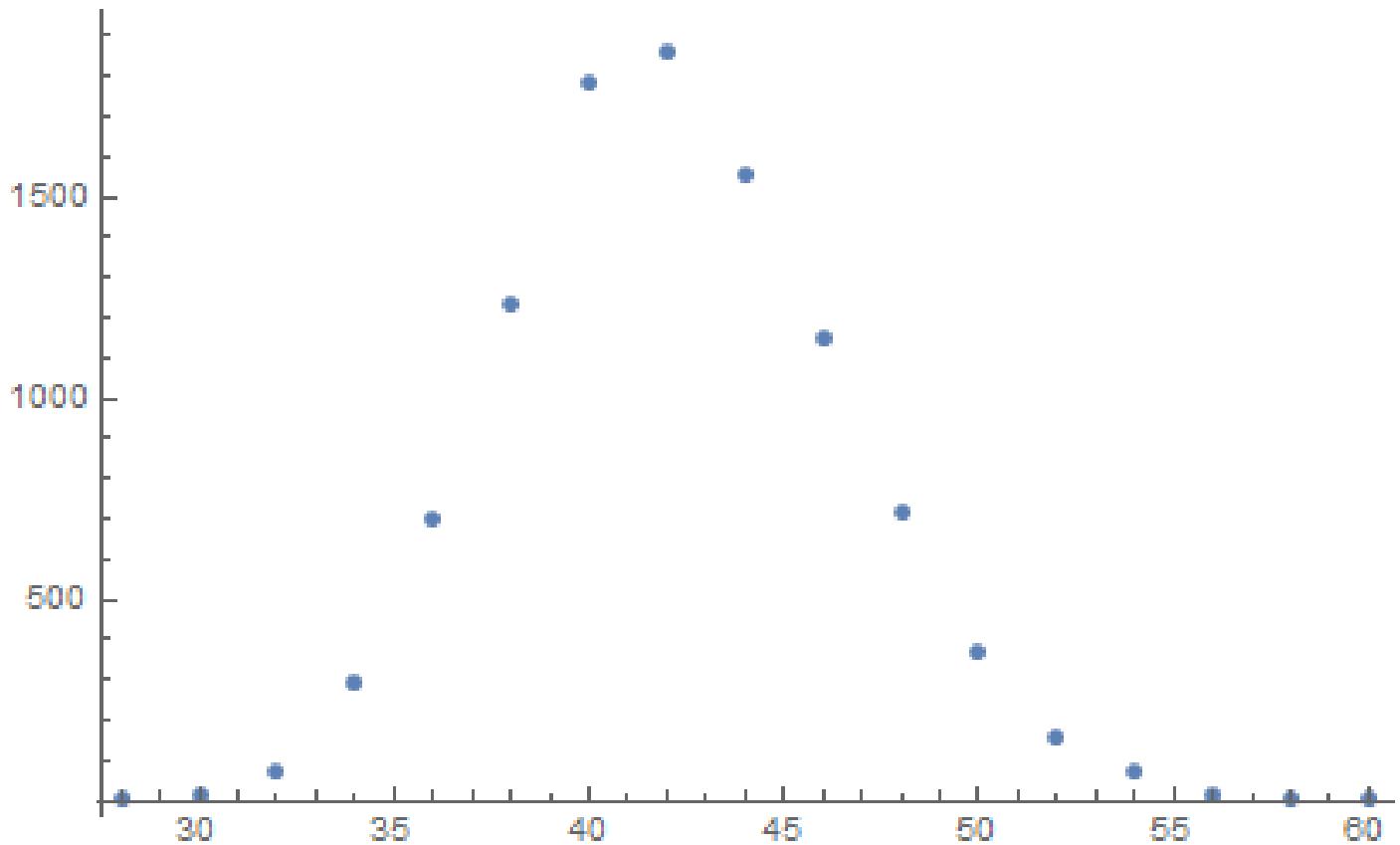
Using Wolfram Mathematica, I analyzed data block's lengths:

```
In[]:= Counts[Map[StringLength[#] &, BinaryStrings]]
Out[]=<|42 -> 1858, 38 -> 1235, 36 -> 699, 46 -> 1151, 40 -> 1784,
44 -> 1558, 50 -> 366, 34 -> 291, 32 -> 74, 56 -> 15, 48 -> 716,
30 -> 13, 52 -> 156, 54 -> 71, 60 -> 3, 58 -> 6, 28 -> 4|>
```

1858 blocks has size of 42 bytes, 1235 blocks has size of 38 bytes, etc.

I made a graph:

```
ListPlot[Counts[Map[StringLength[#] &, BinaryStrings]]]
```



So, most blocks has size between 36 and 48. There is also another thing to notice: all block sizes are even. No single block with odd size.

There are, however, stream ciphers which can operate on byte or bit-level.

### 8.7.4 CryptoPP

The program which can browse this encrypted database is written C# and the .NET code is heavily obfuscated. Nevertheless, there is DLL with x86 code, which, after brief examination, has parts of the CryptoPP popular open-source library! (I just spotted "CryptoPP" strings inside.) Now it's very easy to find all functions inside of DLL because CryptoPP library is open-source.

CryptoPP library has a lot of crypto-functions, including AES (AKA Rijndael). Newer x86 CPUs has AES helper instructions like AESENC, AESDEC and AESKEYGENASSIST<sup>21</sup>. They are not performing encryption/decryption completely, but they do significant amount of job. And newer CryptoPP versions use them. For example, here: 1, 2. To my surprise, during decryption, AESENC is executed, while AESDEC

<sup>21</sup>[https://en.wikipedia.org/wiki/AES\\_instruction\\_set](https://en.wikipedia.org/wiki/AES_instruction_set)

## 8.7. ENCRYPTED DATABASE CASE #1

is not (I just checked with my tracer utility, but any debugger can be used). I checked, if my CPU really supports AES instructions. Some Intel i3 CPUs are not. And if not, CryptoPP library falling back to AES functions implemented in old way <sup>22</sup>. But my CPU supports them. Why AESDEC is still not executed? Why the program use AES encryption in order to decrypt database?

OK, it's not a problem to find a function which encrypts block. It is called *CryptoPP::Rijndael::Enc::ProcessAndXorBlock()*. <https://github.com/mmoss/cryptopp/blob/2772f7b57182b31a41659b48d5f35a7b6cedd34d/src/rijndael.cpp#L349>, and it has references to another function: *Rijndael::Enc::AdvancedProcessBlocks()*. <https://github.com/mmoss/cryptopp/blob/2772f7b57182b31a41659b48d5f35a7b6cedd34d/src/rijndael.cpp#L1179>, which, in turn, has references to two functions (*AESNI\_Enc\_Block* and *AESNI\_Enc\_4\_Blocks*) which has AESENC instructions.

So, judging by CryptoPP internals,

*CryptoPP::Rijndael::Enc::ProcessAndXorBlock()* encrypts one 16-byte block. Let's set breakpoint on it and see, what happens during decryption. I use my simple tracer tool again. The software must decrypt first data block now. Oh, by the way, here is the first data block converted from base64 encoding to hexadecimal data, let's have it at hand:

```
00000000: CA 39 B1 85 75 1B 84 1F F9 31 5E 39 72 13 EC 5D .9..u....1^9r..]
00000010: 95 80 27 02 21 D5 2D 1A 0F D9 45 9F 75 EE 24 C4 ..'!.!....E.u.$.
00000020: B1 27 7F 84 FE 41 37 86 C9 C0 .'.!....A7....
```

This is also arguments of the function from CryptoPP source files:

```
size_t Rijndael::Enc::AdvancedProcessBlocks(const byte *inBlocks, const byte *xorBlocks, byte *outBlocks, size_t length, word32 flags);
```

So it has 5 arguments. Possible flags are:

```
enum {BT_InBlockIsCounter=1, BT_DontIncrementInOutPointers=2, BT_XorInput=4, ↴
      BT_ReverseDirection=8, BT_AllowParallel=16} FlagsForAdvancedProcessBlocks;
```

OK, run tracer on *ProcessAndXorBlock()* function:

```
... tracer.exe -l:filename.exe bpf=filename.exe!0x4339a0,args:5,dump_args:0x10

Warning: no tracer.cfg file.
PID=1984|New process software.exe
no module registered with image base 0x77320000
no module registered with image base 0x76e20000
no module registered with image base 0x77320000
no module registered with image base 0x77220000
Warning: unknown (to us) INT3 breakpoint at ntdll.dll!LdrVerifyImageMatchesChecksum+0x96c (0x776c103b)
(0) software.exe!0x4339a0(0x38b920, 0x0, 0x38b978, 0x10, 0x0) (called from software.exe!.text+0x33c0d (0x13e4c0d))
Argument 1/5
0038B920: 01 00 00 00 FF FF FF-79 C1 69 0B 67 C1 04 7D ".....y.i.g.."
Argument 3/5
0038B978: CD CD CD CD CD CD CD-CD CD CD CD CD CD CD "....."
(0) software.exe!0x4339a0() -> 0x0
Argument 3/5 difference
00000000: C7 39 4E 7B 33 1B D6 1F-B8 31 10 39 39 13 A5 5D ".9N{3....1.99..]"
(0) software.exe!0x4339a0(0x38a828, 0x38a838, 0x38bb40, 0x0, 0x8) (called from software.exe!.text+0x3a407 (0x13eb407))
Argument 1/5
0038A828: 95 80 27 02 21 D5 2D 1A-0F D9 45 9F 75 EE 24 C4 "...!....E.u.$."
Argument 2/5
0038A838: B1 27 7F 84 FE 41 37 86-C9 C0 00 CD CD CD CD CD CD ".'....A7....."
Argument 3/5
0038BB40: CD CD CD CD CD CD CD-CD CD CD CD CD CD CD CD "....."
(0) software.exe!0x4339a0() -> 0x0
(0) software.exe!0x4339a0(0x38b920, 0x38a828, 0x38bb30, 0x10, 0x0) (called from software.exe!.text+0x33c0d (0x13e4c0d))
Argument 1/5
0038B920: CA 39 B1 85 75 1B 84 1F-F9 31 5E 39 72 13 EC 5D ".9..u....1^9r..]"
Argument 2/5
0038A828: 95 80 27 02 21 D5 2D 1A-0F D9 45 9F 75 EE 24 C4 "...!....E.u.$."
```

<sup>22</sup><https://github.com/mmoss/cryptopp/blob/2772f7b57182b31a41659b48d5f35a7b6cedd34d/src/rijndael.cpp#L355>

## 8.7. ENCRYPTED DATABASE CASE #1

```
Argument 3/5
0038BB30: CD "....."
(0) software.exe!0x4339a0() -> 0x0
Argument 3/5 difference
00000000: 45 00 20 00 4A 00 4F 00-48 00 4E 00 53 00 00 00 "E. .J.O.H.N.S..."
(0) software.exe!0x4339a0(0x38b920, 0x0, 0x38b978, 0x10, 0x0) (called from software.exe!.text+0x38b920)
    ↳ x33c0d (0x13e4c0d))
Argument 1/5
0038B920: 95 80 27 02 21 D5 2D 1A-0F D9 45 9F 75 EE 24 C4 "...!....E.u.$."
Argument 3/5
0038B978: 95 80 27 02 21 D5 2D 1A-0F D9 45 9F 75 EE 24 C4 "...!....E.u.$."
(0) software.exe!0x4339a0() -> 0x0
Argument 3/5 difference
00000000: B1 27 7F E4 9F 01 E3 81-CF C6 12 FB B9 7C F1 BC ".'.....|..."
PID=1984|Process software.exe exited. ExitCode=0 (0x0)
```

Here we can see inputs and outputs to the *ProcessAndXorBlock()* function.

This is output from the first call of encryption operation:

```
00000000: C7 39 4E 7B 33 1B D6 1F-B8 31 10 39 39 13 A5 5D ".9N{3....1.99..]"
```

Then the *ProcessAndXorBlock()* is called with 0-length block, but with 8 flag (*BT\_ReverseDirection*).

Second:

```
00000000: 45 00 20 00 4A 00 4F 00-48 00 4E 00 53 00 00 00 "E. .J.O.H.N.S..."
```

Wow, there is some string familiar to us!

Third:

```
00000000: B1 27 7F E4 9F 01 E3 81-CF C6 12 FB B9 7C F1 BC ".'.....|..."
```

The first output is very similar to the first 16 bytes of the encrypted buffer.

Output of the first call of *ProcessAndXorBlock()*:

```
00000000: C7 39 4E 7B 33 1B D6 1F-B8 31 10 39 39 13 A5 5D ".9N{3....1.99..]"
```

First 16 bytes of encrypted buffer:

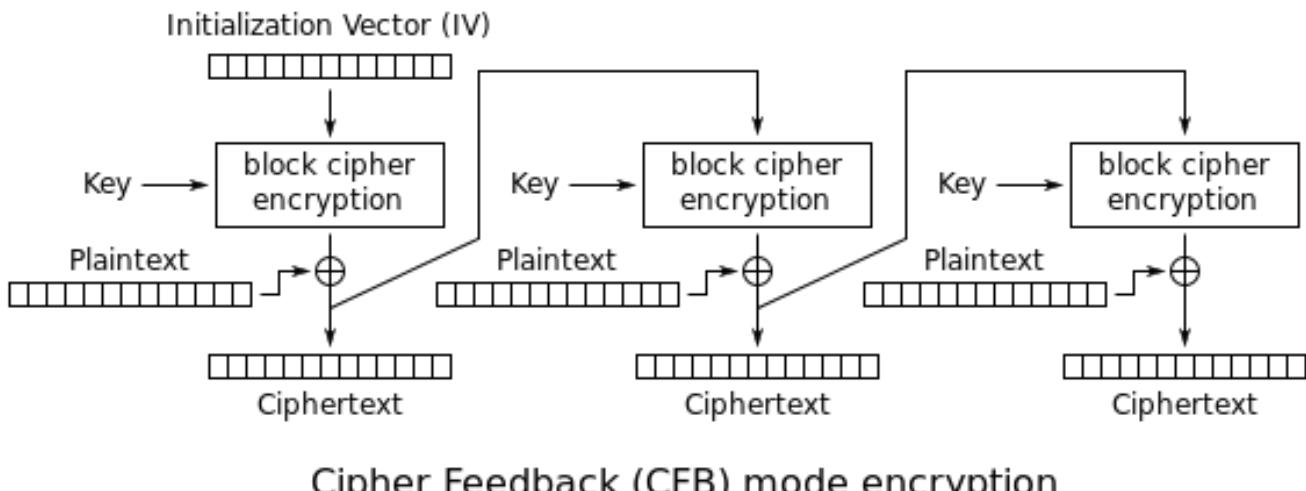
```
00000000: CA 39 B1 85 75 1B 84 1F F9 31 5E 39 72 13 EC 5D .9..u....1^9r..]
```

There are too much equal bytes! How AES encryption result can be very similar to the encrypted buffer while this is not encryption but rather decryption?!

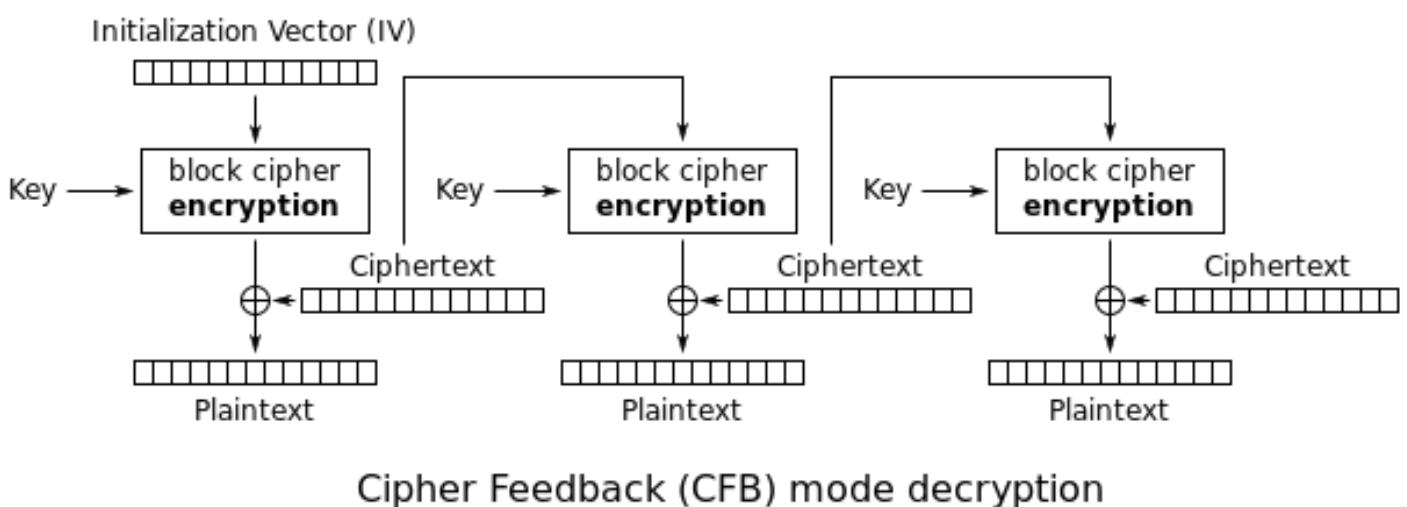
### 8.7.5 Cipher Feedback mode

The answer is CFB (Cipher Feedback mode): In this mode, AES algorithms used not as encryption algorithm, but as a device which generates cryptographically secure random data. The actual encryption is happens using simple XOR operation.

Here is encryption algorithm (images are taken from Wikipedia):



And decryption:



Now let's see: AES encryption operation generates 16 bytes (or 128 bits) or *random* data to be used while XOR-ing, who forces us to use all 16 bytes? If at the last stage we've got 1 byte of data, let's xor 1 byte of data with 1 byte of generated *random* data? This leads to important property of CFB mode: data must not be padded, data of arbitrary size can be encrypted and decrypted.

Oh, that's why all encrypted blocks are not padded. And that's why AESDEC instruction is never called.

Let's try to decrypt first block manually, using Python. CFB mode also use IV (initialization vector), as a seed to *random generator*. In our case, IV is the block which is encrypted at first stage:

```
0038B920: 01 00 00 00 FF FF FF FF-79 C1 69 0B 67 C1 04 7D ".....y.i.g..}"
```

Oh, and we also have to recover encryption key. There is AESKEYGENASSIST is DLL, and it is called, and it is used in the

*Rijndael::Base::UncheckedSetKey()* function:

<https://github.com/mross/cryptopp/blob/2772f7b57182b31a41659b48d5f35a7b6cedd34d/src/rijndael.cpp#L198> It's easy to find it in IDA and set breakpoint. Let's see:

```
... tracer.exe -l:filename.exe bpf=filename.exe!0x435c30,args:3,dump_args:0x10
```

Warning: no tracer.cfg file.

PID=2068|New process software.exe

no module registered with image base 0x77320000

no module registered with image base 0x76e20000

no module registered with image base 0x77320000

## 8.7. ENCRYPTED DATABASE CASE #1

```
no module registered with image base 0x77220000
Warning: unknown (to us) INT3 breakpoint at ntdll.dll!LdrVerifyImageMatchesChecksum+0x96c (0x776c103b)
(0) software.exe!0x435c30(0x15e8000, 0x10, 0x14f808) (called from software.exe!.text+0x22fa1 (0x13d3fa1))
Argument 1/3
015E8000: CD C5 7E AD 28 5F 6D E1-CE 8F CC 29 B1 21 88 8E "...~.(_m....).!.."
Argument 3/3
0014F808: 38 82 58 01 C8 B9 46 00-01 D1 3C 01 00 F8 14 00 "8.X...F...<...."
Argument 3/3 +0x0: software.exe!.rdata+0x5238
Argument 3/3 +0x8: software.exe!.text+0x1c101
(0) software.exe!0x435c30() -> 0x13c2801
PID=2068|Process software.exe exited. ExitCode=0 (0x0)
```

So this is the key: *CD C5 7E AD 28 5F 6D E1-CE 8F CC 29 B1 21 88 8E*.

During manual decryption we've got this:

```
00000000: 0D 00 FF FE 46 00 52 00 41 00 4E 00 4B 00 49 00 ....F.R.A.N.K.I.
00000010: 45 00 20 00 4A 00 4F 00 48 00 4E 00 53 00 66 66 E..J.O.H.N.S.ff
00000020: 66 66 66 9E 61 40 D4 07 06 01 fff.a@....
```

Now this is something readable! And now we can see why there were so many equal bytes at the first decryption stage: because plaintext has so many zero bytes!

Let's decrypt the second block:

```
00000000: 17 98 D0 84 3A E9 72 4F DB 82 3F AD E9 3E 2A A8 .....r0..?..>*.
00000010: 41 00 52 00 52 00 4F 00 4E 00 CD CC CC CC CC CC A.R.R.O.N.....
00000020: 1B 40 D4 07 06 01 .@....
```

Third, fourth and fifth:

```
00000000: 5D 90 59 06 EF F4 96 B4 7C 33 A7 4A BE FF 66 AB ].Y.....|3.J..f.
00000010: 49 00 47 00 47 00 53 00 00 00 00 00 00 C0 65 40 I.G.G.S.....e@
00000020: D4 07 06 01 ....
```

```
00000000: D3 15 34 5D 21 18 7C 6E AA F8 2D FE 38 F9 D7 4E ..4]!.|n...8..N
00000010: 41 00 20 00 44 00 4F 00 48 00 45 00 52 00 54 00 A..D.O.H.E.R.T.
00000020: 59 00 48 E1 7A 14 AE FF 68 40 D4 07 06 02 Y.H.z...h@....
```

```
00000000: 1E 8B 90 0A 17 7B C5 52 31 6C 4E 2F DE 1B 27 19 .....{.R1lN...'.
00000010: 41 00 52 00 43 00 55 00 53 00 00 00 00 00 00 60 A.R.C.U.S.....
00000020: 66 40 D4 07 06 03 f@....
```

All blocks decrypted seems correctly except of first 16 byte part.

### 8.7.6 Initializing Vector

What can affect first 16 bytes?

Let's back to CFB decryption algorithm again: [8.7.5 on the previous page](#).

We can see that IV can affect to first block decryption operation, but not the second, because the second stage used ciphertext from the first stage, and in case of decryption, it's the same, no matter what IV has!

So probably, IV is different each time. Using my tracer, I'll take a look at the first input during decryption of the second block of XML file:

```
0038B920: 02 00 00 00 FE FF FF FF-79 C1 69 0B 67 C1 04 7D ".....y.i.g..}"
```

... third:

```
0038B920: 03 00 00 00 FD FF FF FF-79 C1 69 0B 67 C1 04 7D ".....y.i.g..}"
```

## 8.7. ENCRYPTED DATABASE CASE #1

It seems, first and fifth byte are changed each time. I finally concluded that the first 32-bit integer is just OrderID from the XML file, and the second is also OrderID, but negated. All other 8 bytes are static. Now I have decrypted the whole database: [https://raw.githubusercontent.com/dennis714/yurichev.com/master/blog/encrypted\\_DB\\_case\\_1/decrypted.full.txt](https://raw.githubusercontent.com/dennis714/yurichev.com/master/blog/encrypted_DB_case_1/decrypted.full.txt).

The Python script used for this is: [https://github.com/dennis714/yurichev.com/blob/master/blog/encrypted\\_DB\\_case\\_1/decrypt\\_blocks.py](https://github.com/dennis714/yurichev.com/blob/master/blog/encrypted_DB_case_1/decrypt_blocks.py).

Perhaps, author wanted each block encrypted differently, so he/she used OrderID as part of key. It would be also possible to make different AES key instead of IV.

So now we know that IV only affects first block during decryption in CFB mode, this is feature of it. All other blocks can be decrypted without knowledge IV, but using the key.

OK, so why CFB mode? Apparently, because the very first AES example on CryptoPP wiki uses CFB mode: [http://www.cryptopp.com/wiki/Advanced\\_Encryption\\_Standard#Encrypting\\_and\\_Decrypting\\_Using\\_AES](http://www.cryptopp.com/wiki/Advanced_Encryption_Standard#Encrypting_and_Decrypting_Using_AES). Supposedly, CryptoPP developers choose it for simplicity: the example can encrypt/decrypt text strings with arbitrary lengths, without padding.

It is very likely, my program's programmer(s) just copypasted the example from CryptoPP wiki page. Many programmers do so.

The only difference that IV is chosen randomly in CryptoPP wiki example, while this indeterminism wasn't allowable to programmers of the software we are dissecting now, so they choose to initialize IV using Order ID.

Now we can proceed to analyzing matter of each byte in the decrypted block.

### 8.7.7 Structure of the buffer

Let's take first four decrypted blocks:

00000000: 0D 00 FF FE 46 00 52 00	41 00 4E 00 4B 00 49 00	....F.R.A.N.K.I.
00000010: 45 00 20 00 4A 00 4F 00	48 00 4E 00 53 00 66 66	E. .J.O.H.N.S.ff
00000020: 66 66 66 9E 61 40 D4 07	06 01	fff.a@....
00000000: 0B 00 FF FE 4C 00 4F 00	52 00 49 00 20 00 42 00	....L.O.R.I. .B.
00000010: 41 00 52 00 52 00 4F 00	4E 00 CD CC CC CC CC	A.R.R.O.N.....
00000020: 1B 40 D4 07 06 01		.@....
00000000: 0A 00 FF FE 47 00 41 00	52 00 59 00 20 00 42 00	....G.A.R.Y. .B.
00000010: 49 00 47 00 47 00 53 00	00 00 00 00 00 00 C0 65 40	I.G.G.S.....e@
00000020: D4 07 06 01		....
00000000: 0F 00 FF FE 4D 00 45 00	4C 00 49 00 4E 00 44 00	....M.E.L.I.N.D.
00000010: 41 00 20 00 44 00 4F 00	48 00 45 00 52 00 54 00	A. .D.O.H.E.R.T.
00000020: 59 00 48 E1 7A 14 AE FF	68 40 D4 07 06 02	Y.H.z...h@....

UTF-16 encoded text strings are clearly visible, these are names and surnames. The first byte (or 16-bit word) is seems string length, we can visually check it. FF FE is seems Unicode BOM.

There are 12 more bytes after each string.

Using this script ([https://github.com/dennis714/yurichev.com/blob/master/blog/encrypted\\_DB\\_case\\_1/dump\\_buffer\\_rest.py](https://github.com/dennis714/yurichev.com/blob/master/blog/encrypted_DB_case_1/dump_buffer_rest.py)) I've got random selection of the block tails:

dennis@...:\$ python decrypt.py encrypted.xml   shuf   head -20	
00000000: 48 E1 7A 14 AE 5F 62 40	DD 07 05 08 H.z.._b@....
00000000: 00 00 00 00 00 40 5A 40	DC 07 08 18 .....@Z@....
00000000: 00 00 00 00 00 80 56 40	D7 07 0B 04 .....V@....
00000000: 00 00 00 00 00 60 61 40	D7 07 0C 1C .....a@....
00000000: 00 00 00 00 00 20 63 40	D9 07 05 18 .....c@....
00000000: 3D 0A D7 A3 70 FD 34 40	D7 07 07 11 =....p.4@....
00000000: 00 00 00 00 A0 63 40	D5 07 05 19 .....c@....
00000000: CD CC CC CC CC 3C 5C 40	D7 07 08 11 .....@....
00000000: 66 66 66 66 FE 62 40	D4 07 06 05 fffff.b@....
00000000: 1F 85 EB 51 B8 FE 40 40	D6 07 09 1E ...Q..@....
00000000: 00 00 00 00 40 5F 40	DC 07 02 18 .....@_@....
00000000: 48 E1 7A 14 AE 9F 67 40	D8 07 05 12 H.z...g@....
00000000: CD CC CC CC CC 3C 5E 40	DC 07 01 07 .....^@....

## 8.7. ENCRYPTED DATABASE CASE #1

00000000: 00 00 00 00 00 00 67 40	D4 07 0B 0E	.....g@....
00000000: 00 00 00 00 00 40 51 40	DC 07 04 0B	.....@Q@....
00000000: 00 00 00 00 00 40 56 40	D7 07 07 0A	.....@V@....
00000000: 8F C2 F5 28 5C 7F 55 40	DB 07 01 16	....(.U@....
00000000: 00 00 00 00 00 00 32 40	DB 07 06 09	.....2@....
00000000: 66 66 66 66 7E 66 40	D9 07 0A 06	fffff~f@....
00000000: 48 E1 7A 14 AE DF 68 40	D5 07 07 16	H.z...h@....

We first see the 0x40 and 0x07 bytes present in each *tail*. The very last byte is always in 1..0x1F (1..31) range, as I checked. The penultimate byte is always in 1..0xC (1..12) range. Wow, that looks like a date! Year can be represented as 16-bit value, and maybe last 4 bytes is date (16 bits for year, 8 bits for month and day)? 0x7DD is 2013, 0x7D5 is 2005, etc. Seems fine. This is a date. There are 8 more bytes. Judging by the fact this is database named *orders*, maybe some kind of sum is present here? I made attempt to interpret it as double-precision IEEE 754 floating point and dump all values!

Some are:

```
71.0
134.0
51.95
53.0
121.99
96.95
98.95
15.95
85.95
184.99
94.95
29.95
85.0
36.0
130.99
115.95
87.99
127.95
114.0
150.95
```

Looks like real!

Now we can dump names, sums and dates.

```
plain:
00000000: 0D 00 FF FE 46 00 52 00 41 00 4E 00 4B 00 49 00 ....F.R.A.N.K.I.
00000010: 45 00 20 00 4A 00 4F 00 48 00 4E 00 53 00 66 66 E. .J.O.H.N.S.ff
00000020: 66 66 66 9E 61 40 D4 07 06 01 fff.a@....
OrderID= 1 name= FRANKIE JOHNS sum= 140.95 date= 2004 / 6 / 1

plain:
00000000: 0B 00 FF FE 4C 00 4F 00 52 00 49 00 20 00 42 00 ....L.O.R.I. .B.
00000010: 41 00 52 00 52 00 4F 00 4E 00 CD CC CC CC CC CC A.R.R.O.N. .....
00000020: 1B 40 D4 07 06 01 .@....
OrderID= 2 name= LORI BARRON sum= 6.95 date= 2004 / 6 / 1

plain:
00000000: 0A 00 FF FE 47 00 41 00 52 00 59 00 20 00 42 00 ....G.A.R.Y. .B.
00000010: 49 00 47 00 47 00 53 00 00 00 00 00 00 C0 65 40 I.G.G.S. ....e@
00000020: D4 07 06 01 .....
OrderID= 3 name= GARY BIGGS sum= 174.0 date= 2004 / 6 / 1

plain:
00000000: 0F 00 FF FE 4D 00 45 00 4C 00 49 00 4E 00 44 00 ....M.E.L.I.N.D.
00000010: 41 00 20 00 44 00 4F 00 48 00 45 00 52 00 54 00 A. .D.O.H.E.R.T.
00000020: 59 00 48 E1 7A 14 AE FF 68 40 D4 07 06 02 Y.H.z...h@....
OrderID= 4 name= MELINDA DOHERTY sum= 199.99 date= 2004 / 6 / 2

plain:
00000000: 0B 00 FF FE 4C 00 45 00 4E 00 41 00 20 00 4D 00 ....L.E.N.A. .M.
00000010: 41 00 52 00 43 00 55 00 53 00 00 00 00 00 00 60 A.R.C.U.S. .....
00000020: 66 40 D4 07 06 03 f@....
```

## 8.7. ENCRYPTED DATABASE CASE #1

```
OrderID= 5 name= LENA MARCUS sum= 179.0 date= 2004 / 6 / 3
```

See more: [https://raw.githubusercontent.com/dennis714/yurichev.com/master/blog/encrypted\\_DB\\_case\\_1/decrypted.full.with\\_data.txt](https://raw.githubusercontent.com/dennis714/yurichev.com/master/blog/encrypted_DB_case_1/decrypted.full.with_data.txt). Or filtered: [https://github.com/dennis714/yurichev.com/blob/master/blog/encrypted\\_DB\\_case\\_1/decrypted.short.txt](https://github.com/dennis714/yurichev.com/blob/master/blog/encrypted_DB_case_1/decrypted.short.txt). Seems correct.

This is some kind of OOP serialization, i.e., packing differently typed values into binary buffer for storing and/or transmitting.

### 8.7.8 Noise at the end

The only thing is that sometimes, tail is bigger:

```
00000000: 0E 00 FF FE 54 00 48 00 45 00 52 00 45 00 53 00 ....T.H.E.R.E.S.  
00000010: 45 00 20 00 54 00 55 00 54 00 54 00 4C 00 45 00 E. .T.U.T.T.L.E.  
00000020: 66 66 66 66 1E 63 40 D4 07 07 1A 00 07 07 19 fffff.c@.....  
OrderID= 172 name= THERESE TUTTLE sum= 152.95 date= 2004 / 7 / 26
```

(00 07 07 19 bytes are not used and is ballast).

```
00000000: 0C 00 FF FE 4D 00 45 00 4C 00 41 00 4E 00 49 00 ....M.E.L.A.N.I.  
00000010: 45 00 20 00 4B 00 49 00 52 00 4B 00 00 00 00 00 E. .K.I.R.K.....  
00000020: 00 20 64 40 D4 07 09 02 00 02 . d@.....  
OrderID= 286 name= MELANIE KIRK sum= 161.0 date= 2004 / 9 / 2
```

(00 02 are not used).

After close examination, we can see, that the noise at the end of tail is just left from previous encryption!

Here are two subsequent buffers:

```
00000000: 10 00 FF FE 42 00 4F 00 4E 00 4E 00 49 00 45 00 ....B.O.N.N.I.E.  
00000010: 20 00 47 00 4F 00 4C 00 44 00 53 00 54 00 45 00 .G.O.L.D.S.T.E.  
00000020: 49 00 4E 00 9A 99 99 99 99 79 46 40 D4 07 07 19 I.N.....yF@....  
OrderID= 171 name= BONNIE GOLDSTEIN sum= 44.95 date= 2004 / 7 / 25  
  
00000000: 0E 00 FF FE 54 00 48 00 45 00 52 00 45 00 53 00 ....T.H.E.R.E.S.  
00000010: 45 00 20 00 54 00 55 00 54 00 54 00 4C 00 45 00 E. .T.U.T.T.L.E.  
00000020: 66 66 66 66 1E 63 40 D4 07 07 1A 00 07 07 19 fffff.c@.....  
OrderID= 172 name= THERESE TUTTLE sum= 152.95 date= 2004 / 7 / 26
```

(The last 07 07 19 bytes are copied from the previous plaintext buffer).

Another two subsequent buffers:

```
00000000: 0D 00 FF FE 4C 00 4F 00 52 00 45 00 4E 00 45 00 ....L.O.R.E.N.E.  
00000010: 20 00 4F 00 54 00 4F 00 4F 00 4C 00 45 00 CD CC .O.T.O.O.L.E...  
00000020: CC CC CC 3C 5E 40 D4 07 09 02 ...<^@....  
OrderID= 285 name= LORENE OTOOLE sum= 120.95 date= 2004 / 9 / 2  
  
00000000: 0C 00 FF FE 4D 00 45 00 4C 00 41 00 4E 00 49 00 ....M.E.L.A.N.I.  
00000010: 45 00 20 00 4B 00 49 00 52 00 4B 00 00 00 00 00 E. .K.I.R.K.....  
00000020: 00 20 64 40 D4 07 09 02 00 02 . d@.....  
OrderID= 286 name= MELANIE KIRK sum= 161.0 date= 2004 / 9 / 2
```

The last 02 byte is copied from the previous plaintext buffer.

It's possible if the buffer used while encrypting is global and/or isn't cleared before each encryption. The final buffer size value is also biased somehow, nevertheless, the bug left uncaught because it doesn't affect decrypting software, it's just ignores noise at the end. This is common mistake. It even affected OpenSSL (Heartbleed bug).

### 8.7.9 Conclusion

Summary: every practicing reverse engineer should be familiar with major crypto algorithms and also major cryptographical modes. Some books about it: [11.1.9 on page 995](#).

## **8.8. OVERCLOCKING COINTERRA BITCOIN MINER**

Encrypted database contents has been artificially constructed by me for the sake of demonstration. I've got most popular USA names and surnames from there: <http://stackoverflow.com/questions/1803628/raw-list-of-person-names>, and combined them randomly. Dates and sums were also generated randomly.

All files used in this article are here: [https://github.com/dennis714/yurichev.com/tree/master/blog/encrypted\\_DB\\_case\\_1](https://github.com/dennis714/yurichev.com/tree/master/blog/encrypted_DB_case_1).

Nevertheless, many features like these I've observed in real-world software applications. This example is based on them.

### **8.7.10 Post Scriptum: brute-forcing IV**

The case you saw is artificially constructed, but based on some real applications I've observed. When I've been working on one of such real applications, I first noticed that Initializing Vector (IV) is generating using some 32-bit number, I wasn't able to find a link between this value and OrderID. So I prepared to use brute-force, which is indeed possible here. It's not a problem to enumerate all 32-bit values and try each IV based on each. Then you decrypt 16-byte block and check for zero bytes, which are always at fixed places.

## **8.8 Overclocking Cointerra Bitcoin miner**

There was Cointerra Bitcoin miner, looking like that:

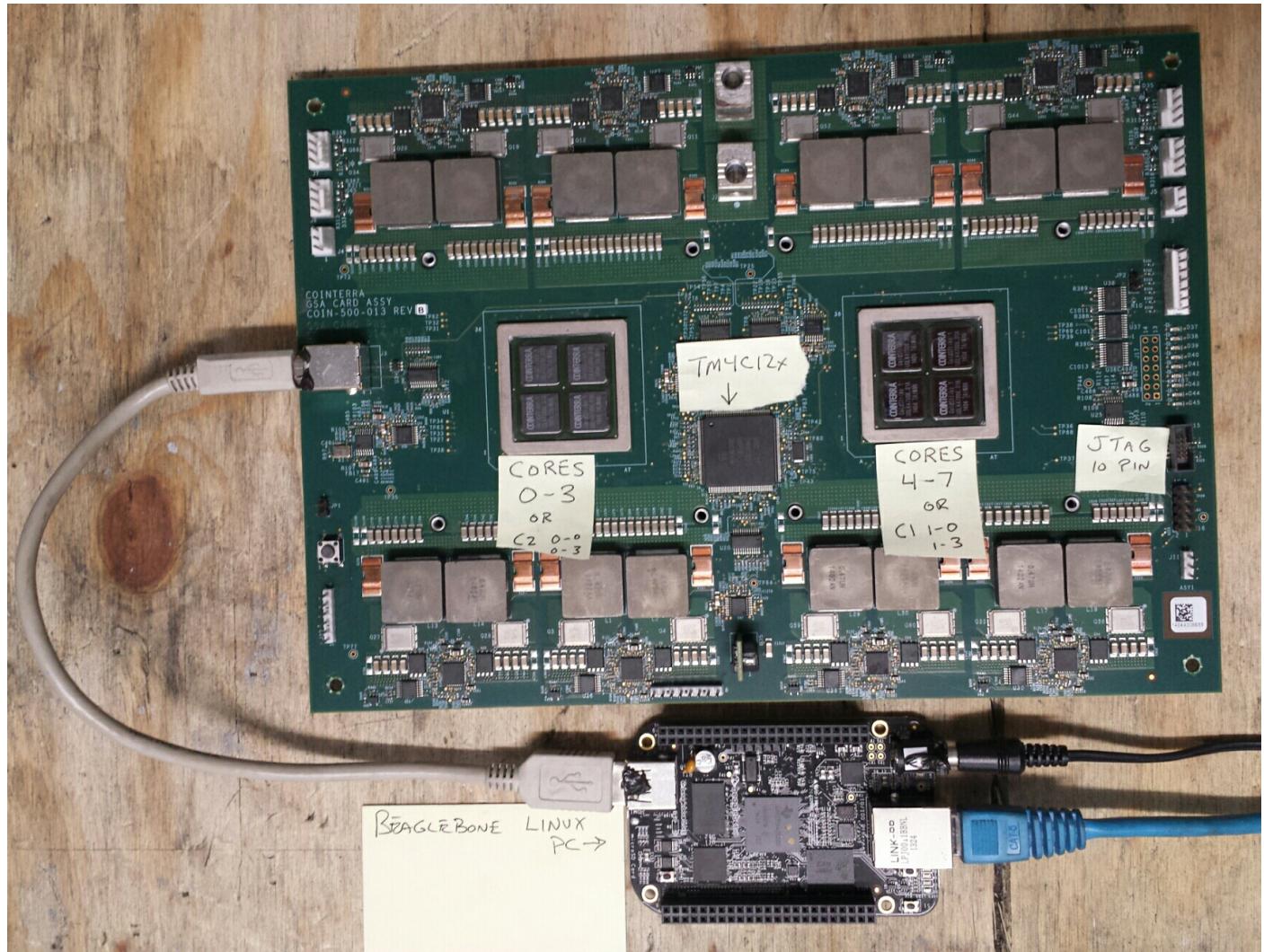


Figure 8.14: Board

## 8.8. OVERCLOCKING COINTERRA BITCOIN MINER

And there was also (possibly leaked) utility<sup>23</sup> which can set clock rate for the board. It runs on additional BeagleBone Linux ARM board (small board at bottom of the picture).

And the author was once asked, is it possible to hack this utility to see, which frequency can be set and which are not. And it is possible to tweak it?

The utility must be executed like that: `./cointool-overclock 0 0 900`, where 900 is frequency in MHz. If the frequency is too high, utility will print “Error with arguments” and exit.

This is a fragment of code around reference to “Error with arguments” text string:

```
...  
.text:0000ABC4      STR      R3, [R11,#var_28]  
.text:0000ABC8      MOV      R3, #optind  
.text:0000ABD0      LDR      R3, [R3]  
.text:0000ABD4      ADD      R3, R3, #1  
.text:0000ABD8      MOV      R3, R3,LSL#2  
.text:0000ABDC      LDR      R2, [R11,#argv]  
.text:0000ABE0      ADD      R3, R2, R3  
.text:0000ABE4      LDR      R3, [R3]  
.text:0000ABE8      MOV      R0, R3 ; nptr  
.text:0000ABEC      MOV      R1, #0 ; endptr  
.text:0000ABF0      MOV      R2, #0 ; base  
.text:0000ABF4      BL       strtoll  
.text:0000ABF8      MOV      R2, R0  
.text:0000ABFC      MOV      R3, R1  
.text:0000AC00      MOV      R3, R2  
.text:0000AC04      STR      R3, [R11,#var_2C]  
.text:0000AC08      MOV      R3, #optind  
.text:0000AC10      LDR      R3, [R3]  
.text:0000AC14      ADD      R3, R3, #2  
.text:0000AC18      MOV      R3, R3,LSL#2  
.text:0000AC1C      LDR      R2, [R11,#argv]  
.text:0000AC20      ADD      R3, R2, R3  
.text:0000AC24      LDR      R3, [R3]  
.text:0000AC28      MOV      R0, R3 ; nptr  
.text:0000AC2C      MOV      R1, #0 ; endptr  
.text:0000AC30      MOV      R2, #0 ; base  
.text:0000AC34      BL       strtoll  
.text:0000AC38      MOV      R2, R0  
.text:0000AC3C      MOV      R3, R1  
.text:0000AC40      MOV      R3, R2  
.text:0000AC44      STR      R3, [R11,#third_argument]  
.text:0000AC48      LDR      R3, [R11,#var_28]  
.text:0000AC4C      CMP      R3, #0  
.text:0000AC50      BLT     errors_with_arguments  
.text:0000AC54      LDR      R3, [R11,#var_28]  
.text:0000AC58      CMP      R3, #1  
.text:0000AC5C      BGT     errors_with_arguments  
.text:0000AC60      LDR      R3, [R11,#var_2C]  
.text:0000AC64      CMP      R3, #0  
.text:0000AC68      BLT     errors_with_arguments  
.text:0000AC6C      LDR      R3, [R11,#var_2C]  
.text:0000AC70      CMP      R3, #3  
.text:0000AC74      BGT     errors_with_arguments  
.text:0000AC78      LDR      R3, [R11,#third_argument]  
.text:0000AC7C      CMP      R3, #0x31  
.text:0000AC80      BLE     errors_with_arguments  
.text:0000AC84      LDR      R2, [R11,#third_argument]  
.text:0000AC88      MOV      R3, #950  
.text:0000AC8C      CMP      R2, R3  
.text:0000AC90      BGT     errors_with_arguments  
.text:0000AC94      LDR      R2, [R11,#third_argument]  
.text:0000AC98      MOV      R3, #0x51EB851F  
.text:0000ACA0      SMULL   R1, R3, R3, R2  
.text:0000ACA4      MOV      R1, R3,ASR#4  
.text:0000ACA8      MOV      R3, R2,ASR#31
```

<sup>23</sup>Can be downloaded here: [http://yurichev.com/blog/bitcoin\\_miner/files/cointool-overclock](http://yurichev.com/blog/bitcoin_miner/files/cointool-overclock)

## 8.8. OVERCLOCKING COINTERRA BITCOIN MINER

```

.text:0000ACAC      RSB      R3, R3, R1
.text:0000ACB0      MOV      R1, #50
.text:0000ACB4      MUL      R3, R1, R3
.text:0000ACB8      RSB      R3, R3, R2
.text:0000ACBC      CMP      R3, #0
.text:0000ACC0      BEQ      loc_ACEC
.text:0000ACC4
.text:0000ACC4 errors_with_arguments
.text:0000ACC4
.text:0000ACC4      LDR      R3, [R11,#argv]
.text:0000ACC8      LDR      R3, [R3]
.text:0000ACCC      MOV      R0, R3 ; path
.text:0000ACD0      BL       __xpg_basename
.text:0000ACD4      MOV      R3, R0
.text:0000ACD8      MOV      R0, #aSErrorWithArgu ; format
.text:0000ACE0      MOV      R1, R3
.text:0000ACE4      BL       printf
.text:0000ACE8      B       loc_ADD4
.text:0000ACEC ; -----
.text:0000ACEC loc_ACEC           ; CODE XREF: main+66C
.text:0000ACEC      LDR      R2, [R11,#third_argument]
.text:0000ACF0      MOV      R3, #499
.text:0000ACF4      CMP      R2, R3
.text:0000ACF8      BGT      loc_AD08
.text:0000ACFC      MOV      R3, #0x64
.text:0000AD00      STR      R3, [R11,#unk_constant]
.text:0000AD04      B       jump_to_write_power
.text:0000AD08 ; -----
.text:0000AD08
.text:0000AD08 loc_AD08           ; CODE XREF: main+6A4
.text:0000AD08      LDR      R2, [R11,#third_argument]
.text:0000AD0C      MOV      R3, #799
.text:0000AD10      CMP      R2, R3
.text:0000AD14      BGT      loc_AD24
.text:0000AD18      MOV      R3, #0x5F
.text:0000AD1C      STR      R3, [R11,#unk_constant]
.text:0000AD20      B       jump_to_write_power
.text:0000AD24 ; -----
.text:0000AD24
.text:0000AD24 loc_AD24           ; CODE XREF: main+6C0
.text:0000AD24      LDR      R2, [R11,#third_argument]
.text:0000AD28      MOV      R3, #899
.text:0000AD2C      CMP      R2, R3
.text:0000AD30      BGT      loc_AD40
.text:0000AD34      MOV      R3, #0x5A
.text:0000AD38      STR      R3, [R11,#unk_constant]
.text:0000AD3C      B       jump_to_write_power
.text:0000AD40 ; -----
.text:0000AD40
.text:0000AD40 loc_AD40           ; CODE XREF: main+6DC
.text:0000AD40      LDR      R2, [R11,#third_argument]
.text:0000AD44      MOV      R3, #999
.text:0000AD48      CMP      R2, R3
.text:0000AD4C      BGT      loc_AD5C
.text:0000AD50      MOV      R3, #0x55
.text:0000AD54      STR      R3, [R11,#unk_constant]
.text:0000AD58      B       jump_to_write_power
.text:0000AD5C ; -----
.text:0000AD5C
.text:0000AD5C loc_AD5C           ; CODE XREF: main+6F8
.text:0000AD5C      LDR      R2, [R11,#third_argument]
.text:0000AD60      MOV      R3, #1099
.text:0000AD64      CMP      R2, R3
.text:0000AD68      BGT      jump_to_write_power
.text:0000AD6C      MOV      R3, #0x50
.text:0000AD70      STR      R3, [R11,#unk_constant]
.text:0000AD74      jump_to_write_power          ; CODE XREF: main+6B0
                                         ; main+6CC ...

```

## 8.8. OVERCLOCKING COINTERRA BITCOIN MINER

```

.text:0000AD74      LDR    R3, [R11,#var_28]
.text:0000AD78      UXTB   R1, R3
.text:0000AD7C      LDR    R3, [R11,#var_2C]
.text:0000AD80      UXTB   R2, R3
.text:0000AD84      LDR    R3, [R11,#unk_constant]
.text:0000AD88      UXTB   R3, R3
.text:0000AD8C      LDR    R0, [R11,#third_argument]
.text:0000AD90      UXTH   R0, R0
.text:0000AD94      STR    R0, [SP,#0x44+var_44]
.text:0000AD98      LDR    R0, [R11,#var_24]
.text:0000AD9C      BL     write_power
.text:0000ADA0      LDR    R0, [R11,#var_24]
.text:0000ADA4      MOV    R1, #0x5A
.text:0000ADA8      BL     read_loop
.text:0000ADAC      B      loc_ADD4

...
.rodata:0000B378 aSErrorWithArgu DCB "%s: Error with arguments",0xA,0 ; DATA XREF: main+684
...

```

Function names were present in debugging information of the original binary, like `write_power`, `read_loop`. But labels inside functions were named by me.

`optind` name looks familiar. It is from `getopt` \*NIX library intended for command-line parsing—well, this is exactly what happens in this utility. Then, the 3rd argument (where frequency value is to be passed) is converted from string to number using call to `strtol()` function.

The value is then checked against various constants. At `0xACEC`, it's checked, if it is lesser or equal to `499`, `0x64` is to be passed finally to `write_power()` function (which sends a command through USB using `send_msg()`). If it is greater than `499`, jump to `0xAD08` is occurred.

At `0xAD08` it's checked, if it's lesser or equal to `799`. `0x5F` is then passed to `write_power()` function in case of success.

There are more checks: for `899` at `0xAD24`, for `0x999` at `0xAD40` and finally, for `1099` at `0xAD5C`. If the input frequency is lesser or equal to `1099`, `0x50` will be passed (at `0xAD6C`) to `write_power()` function. And there is some kind of bug. If the value is still greater than `1099`, the value itself is passed into `write_power()` function. Oh, it's not a bug, because we can't get here: value is checked first against `950` at `0xAC88`, and if it is greater, error message will be displayed and the utility will finish.

Now the table between frequency in MHz and value passed to `write_power()` function:

MHz	hexadecimal	decimal
499MHz	0x64	100
799MHz	0x5f	95
899MHz	0x5a	90
999MHz	0x55	85
1099MHz	0x50	80

As it seems, value passed to the board is gradually decreasing during frequency increasing.

Now we see that value of `950MHz` is a hardcoded limit, at least in this utility. Can we trick it?

Let's back to this piece of code:

```

.text:0000AC84      LDR    R2, [R11,#third_argument]
.text:0000AC88      MOV    R3, #950
.text:0000AC8C      CMP    R2, R3
.text:0000AC90      BGT   errors_with_arguments ; I've patched here to 00 00 00 00

```

We must disable `BGT` branch instruction at `0xAC90` somehow. And this is ARM in ARM mode, because, as we see, all addresses are increasing by 4, i.e., each instruction has size of 4 bytes. `NOP` (no operation)

## 8.9. BREAKING SIMPLE EXECUTABLE CRYPTOR

instruction in ARM mode is just four zero bytes: `00 00 00 00`. So by writing four zeros at 0xAC90 address (or physical offset in file 0x2C90) will disable this check.

Not it's possible to set frequencies up to 1050MHz. Even more is possible, but due to the bug, if input value is greater than 1099, raw value in MHz will be passed to the board, which is incorrect.

I didn't go further, but if I had to, I would try to decrease a value which is passed to `write_power()` function.

Now the scary piece of code which I skipped at first:

```
.text:0000AC94    LDR      R2, [R11,#third_argument]
.text:0000AC98    MOV      R3, #0x51EB851F
.text:0000ACA0    SMULL   R1, R3, R3, R2 ; R3=3rg_arg/3.125
.text:0000ACA4    MOV      R1, R3, ASR#4 ; R1=R3/16=3rg_arg/50
.text:0000ACA8    MOV      R3, R2, ASR#31 ; R3=MSB(3rg_arg)
.text:0000ACAC    RSB      R3, R3, R1 ; R3=3rd_arg/50
.text:0000ACB0    MOV      R1, #50
.text:0000ACB4    MUL      R3, R1, R3 ; R3=50*(3rd_arg/50)
.text:0000ACB8    RSB      R3, R3, R2
.text:0000ACBC    CMP      R3, #0
.text:0000ACC0    BEQ      loc_ACEC
.text:0000ACC4    errors_with_arguments
```

Multiplication via division is used here, and constant is 0x51EB851F. I wrote a simple programmer's calculator<sup>24</sup> for myself. And I have there a feature to calculate modulo inverse.</p>

```
modinv32(0x51EB851F)
Warning, result is not integer: 3.125000
(unsigned) dec: 3 hex: 0x3 bin: 11
```

That means that `SMULL` instruction at 0xACA0 is basically divides 3rd argument by 3.125. In fact, all `modinv32()` function in my calculator does, is this:

$$\frac{1}{\frac{\text{input}}{2^{32}}} = \frac{2^{32}}{\text{input}}$$

Then there are additional shifts and now we see than 3rg argument is just divided by 50. And then it's multiplied by 50 again. Why? This is simplest check, if the input value is can be divided by 50 evenly. If the value of this expression is non-zero,  $x$  can't be divided by 50 evenly:

$$x - ((\frac{x}{50}) \cdot 50)$$

This is in fact simple way to calculate remainder of division.

And then, if the remainder is non-zero, error message is displayed. So this utility takes frequency values in form like 850, 900, 950, 1000, etc., but not 855 or 911.

That's it! If you do something like that, please be warned that you may damage your board, just as in case of overclocking other devices like CPUs, GPUs, etc. If you have a Cointerra board, do this on your own risk!

## 8.9 Breaking simple executable cryptor

I've got an executable file which is encrypted by relatively simple encryption. [Here is it](#) (only executable section is left here).

First, all encryption function does is just adds number of position in buffer to the byte. Here is how this can be encoded in Python:

<sup>24</sup><https://github.com/dennis714/progcalc>

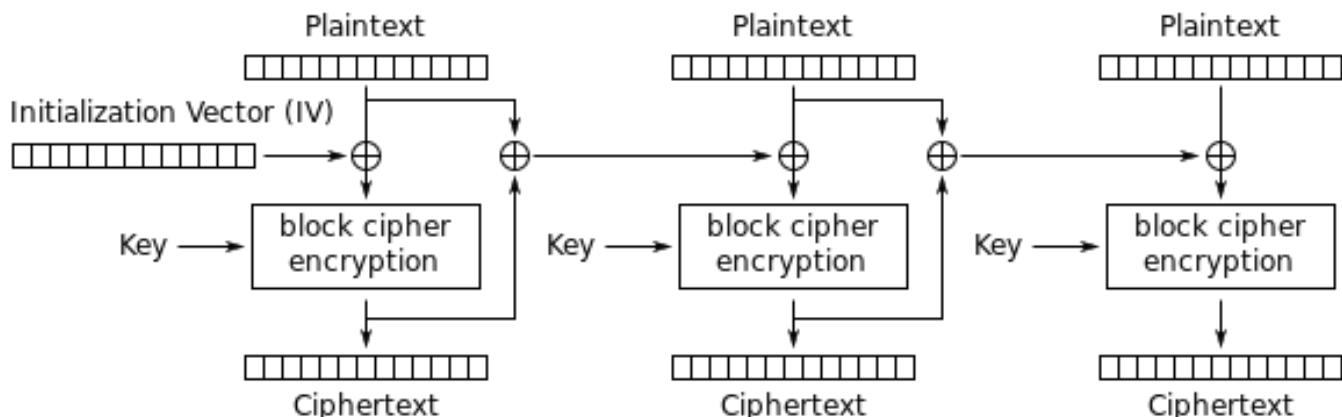
Listing 8.7: Python script

```
#!/usr/bin/env python
def e(i, k):
    return chr ((ord(i)+k) % 256)

def encrypt(buf):
    return e(buf[0], 0)+ e(buf[1], 1)+ e(buf[2], 2) + e(buf[3], 3)+ e(buf[4], 4)+ e(buf[5], 5)+
    ↴ e(buf[6], 6)+ e(buf[7], 7)+
    ↴ e(buf[8], 8)+ e(buf[9], 9)+ e(buf[10], 10)+ e(buf[11], 11)+ e(buf[12], 12)+ e(buf[13], 13)+ e(buf[14], 14)+ e(buf[15], 15)
```

Hence, if you encrypt buffer with 16 zeros, you'll get 0, 1, 2, 3 ... 12, 13, 14, 15.

Propagating Cipher Block Chaining (PCBC) is also used, here is how it works:



Propagating Cipher Block Chaining (PCBC) mode encryption

Figure 8.15: Propagating Cipher Block Chaining encryption (image is taken from Wikipedia article)

The problem is that it's too boring to recover IV (Initialization Vector) each time. Brute-force is also not an option, because IV is too long (16 bytes). Let's see, if it's possible to recover IV for arbitrary encrypted executable file?

Let's try simple frequency analysis. This is 32-bit x86 executable code, so let's gather statistics about most frequent bytes and opcodes. I tried huge oracle.exe file from Oracle RDBMS version 11.2 for windows x86 and I've found that the most frequent byte (no surprise) is zero ( 10%). The next most frequent byte is (again, no surprise) 0xFF ( 5%). The next is 0x8B ( 5%).

0x8B is opcode for `MOV`, this is indeed one of the most busy x86 instructions. Now what about popularity of zero byte? If compiler needs to encode value bigger than 127, it has to use 32-bit displacement instead of 8-bit one, but large values are very rare, so it is padded by zeros. This is at least in `LEA`, `MOV`, `PUSH`, `CALL`.

For example:

8D B0 28 01 00 00	lea      esi, [eax+128h]
8D BF 40 38 00 00	lea      edi, [edi+3840h]

Displacements bigger than 127 are very popular, but they are rarely exceeds 0x10000 (indeed, such large memory buffers/structures are also rare).

Same story with `MOV`, large constants are rare, the most heavily used are 0, 1, 10, 100,  $2^n$ , and so on. Compiler has to pad small constants by zeros to represent them as 32-bit values:

BF 02 00 00 00	mov      edi, 2
BF 01 00 00 00	mov      edi, 1

Now about 00 and FF bytes combined: jumps (including conditional) and calls can pass execution flow forward or backwards, but very often, within the limits of the current executable module. If forward,

## 8.9. BREAKING SIMPLE EXECUTABLE CRYPTOR

displacement is not very big and also padded with zeros. If backwards, displacement is represented as negative value, so padded with FF bytes. For example, transfer execution flow forward:

E8 43 0C 00 00	call _function1
E8 5C 00 00 00	call _function2
0F 84 F0 0A 00 00	jz loc_4F09A0
0F 84 EB 00 00 00	jz loc_4EFBB8

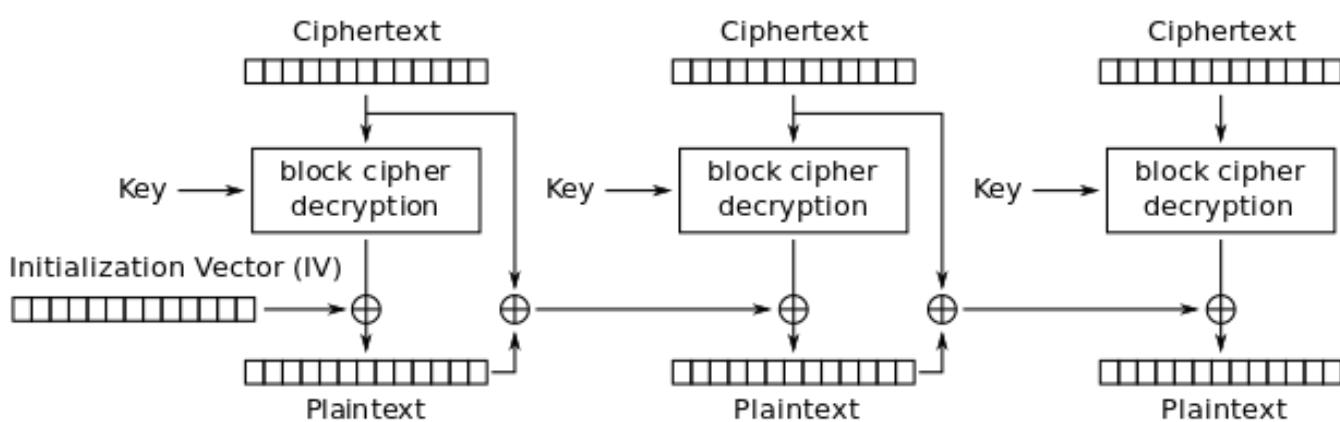
Backwards:

E8 79 0C FE FF	call _function1
E8 F4 16 FF FF	call _function2
0F 84 F8 FB FF FF	jz loc_8212BC
0F 84 06 FD FF FF	jz loc_FF1E7D

FF byte is also very often occurred in negative displacements like these:

8D 85 1E FF FF FF	lea eax, [ebp-0E2h]
8D 95 F8 5C FF FF	lea edx, [ebp-0A308h]

So far so good. Now we have to try various 16-byte keys, decrypt executable section and measure how often 00, FF ad 8B bytes are occurred. Let's also keep in sight how PCBC decryption works:



Propagating Cipher Block Chaining (PCBC) mode decryption

Figure 8.16: Propagating Cipher Block Chaining decryption (image is taken from Wikipedia article)

The good news is that we don't really have to decrypt whole piece of data, but only slice by slice, this is exactly how I did in my previous example: [9.1.4 on page 939](#).

Now I'm trying all possible bytes (0..255) for each byte in key and just pick the byte producing maximal amount of 00/FF/8B bytes in a decrypted slice:

```

#!/usr/bin/env python
import sys, hexdump, array, string, operator

KEY_LEN=16

def chunks(l, n):
    # split n by l-byte chunks
    # http://stackoverflow.com/questions/312443/how-do-you-split-a-list-into-evenly-sized-chunks-in-python
    n = max(1, n)
    return [l[i:i + n] for i in range(0, len(l), n)]

def read_file(fname):
    file=open(fname, mode='rb')
    content=file.read()
    file.close()
    return content

```

```

def decrypt_byte (c, key):
    return chr((ord(c)-key) % 256)

def XOR_PCBC_step (IV, buf, k):
    prev=IV
    rt=""
    for c in buf:
        new_c=decrypt_byte(c, k)
        plain=chr(ord(new_c)^ord(prev))
        prev=chr(ord(c)^ord(plain))
        rt=rt+plain
    return rt

each_Nth_byte=[""]*KEY_LEN

content=read_file(sys.argv[1])
# split input by 16-byte chunks:
all_chunks=chunks(content, KEY_LEN)
for c in all_chunks:
    for i in range(KEY_LEN):
        each_Nth_byte[i]=each_Nth_byte[i] + c[i]

# try each byte of key
for N in range(KEY_LEN):
    print "N=", N
    stat={}
    for i in range(256):
        tmp_key=chr(i)
        tmp=XOR_PCBC_step(tmp_key,each_Nth_byte[N], N)
        # count 0, FFs and 8Bs in decrypted buffer:
        important_bytes=tmp.count('\x00')+tmp.count('\xFF')+tmp.count('\x8B')
        stat[i]=important_bytes
    sorted_stat = sorted(stat.iteritems(), key=operator.itemgetter(1), reverse=True)
    print sorted_stat[0]

```

(Source code can downloaded [here](#).)

I run it and here is a key for which 00/FF/8B bytes presence in decrypted buffer is maximal:

```

N= 0
(147, 1224)
N= 1
(94, 1327)
N= 2
(252, 1223)
N= 3
(218, 1266)
N= 4
(38, 1209)
N= 5
(192, 1378)
N= 6
(199, 1204)
N= 7
(213, 1332)
N= 8
(225, 1251)
N= 9
(112, 1223)
N= 10
(143, 1177)
N= 11
(108, 1286)
N= 12
(10, 1164)
N= 13
(3, 1271)
N= 14
(128, 1253)

```

## 8.9. BREAKING SIMPLE EXECUTABLE CRYPTOR

```
N= 15  
(232, 1330)
```

Let's write decryption utility with the key we got:

```
#!/usr/bin/env python  
import sys, hexdump, array  
  
def xor_strings(s,t):  
    # https://en.wikipedia.org/wiki/XOR_cipher#Example_implementation  
    """xor two strings together"""  
    return "".join(chr(ord(a)^ord(b)) for a,b in zip(s,t))  
  
IV=array.array('B', [147, 94, 252, 218, 38, 192, 199, 213, 225, 112, 143, 108, 10, 3, 128, ↴  
↳ 232]).tostring()  
  
def chunks(l, n):  
    n = max(1, n)  
    return [l[i:i + n] for i in range(0, len(l), n)]  
  
def read_file(fname):  
    file=open(fname, mode='rb')  
    content=file.read()  
    file.close()  
    return content  
  
def decrypt_byte(i, k):  
    return chr ((ord(i)-k) % 256)  
  
def decrypt(buf):  
    return "".join(decrypt_byte(buf[i], i) for i in range(16))  
  
fout=open(sys.argv[2], mode='wb')  
  
prev=IV  
content=read_file(sys.argv[1])  
tmp=chunks(content, 16)  
for c in tmp:  
    new_c=decrypt(c)  
    p=xor_strings (new_c, prev)  
    prev=xor_strings(c, p)  
    fout.write(p)  
fout.close()
```

(Source code can downloaded [here](#).)

Let's check resulting file:

```
$ objdump -b binary -m i386 -D decrypted.bin  
  
...  
  
5: 8b ff          mov    %edi,%edi  
7: 55             push   %ebp  
8: 8b ec          mov    %esp,%ebp  
a: 51             push   %ecx  
b: 53             push   %ebx  
c: 33 db          xor    %ebx,%ebx  
e: 43             inc    %ebx  
f: 84 1d a0 e2 05 01 test   %bl,0x105e2a0  
15: 75 09          jne    0x20  
17: ff 75 08          pushl  0x8(%ebp)  
1a: ff 15 b0 13 00 01 call   *0x10013b0  
20: 6a 6c          push   $0x6c  
22: ff 35 54 d0 01 01 pushl  0x101d054  
28: ff 15 b4 13 00 01 call   *0x10013b4  
2e: 89 45 fc          mov    %eax,-0x4(%ebp)  
31: 85 c0          test   %eax,%eax  
33: 0f 84 d9 00 00 00 je    0x112  
39: 56             push   %esi
```

## 8.10. SAP

```
3a: 57          push %edi
3b: 6a 00       push $0x0
3d: 50          push %eax
3e: ff 15 b8 13 00 01 call *0x10013b8
44: 8b 35 bc 13 00 01 mov 0x10013bc,%esi
4a: 8b f8       mov %eax,%edi
4c: a1 e0 e2 05 01 mov 0x105e2e0,%eax
51: 3b 05 e4 e2 05 01 cmp 0x105e2e4,%eax
57: 75 12       jne 0x6b
59: 53          push %ebx
5a: 6a 03       push $0x3
5c: 57          push %edi
5d: ff d6       call *%esi
```

...

Yes, this is seems correctly disassembled piece of x86 code. The whole dectypted file can be downloaded [here](#).

In fact, this is text section from regedit.exe from Windows 7. But this example is based on a real case I encountered, so just executable is different (and key), algorithm is the same.

### 8.9.1 Other ideas to consider

What if I would fail with such simple frequency analysis? There are other ideas on how to measure correctness of decrypted/decompressed x86 code:

- Many modern compilers aligns functions on 0x10 border. So the space left before is filled with NOPs (0x90) or other NOP instructions with known opcodes: [1.7 on page 1019](#).
- Perhaps, the most frequent pattern in any assembly language is function call:  
PUSH chain / CALL / ADD ESP, X. This sequence can easily detected and found. I've even gathered statistics about average number of function arguments: [10.2 on page 986](#). (Hence, this is average length of PUSH chain.)

By the way, it is interesting to know that the fact that function calls ( PUSH / CALL / ADD ) and MOV instructions are the most frequently executed pieces of code in almost all programs we use. In other words, CPU is very busy passing information between levels of abstractions, or, it can be said, it's very busy switching between these levels. This is a cost of splitting problems into several levels of abstractions.

## 8.10 SAP

### 8.10.1 About SAP client network traffic compression

(Tracing the connection between the TDW\_NOCOMPRESS SAPGUI<sup>25</sup> environment variable and the pesky annoying pop-up window and the actual data compression routine.)

It is known that the network traffic between SAPGUI and SAP is not encrypted by default, but compressed (see here<sup>26</sup> and here<sup>27</sup>).

It is also known that by setting the environment variable TDW\_NOCOMPRESS to 1, it is possible to turn the network packet compression off.

But you will see an annoying pop-up window that cannot be closed:

<sup>25</sup>SAP GUI client

<sup>26</sup><http://go.yurichev.com/17221>

<sup>27</sup>[blog.yurichev.com](http://blog.yurichev.com)

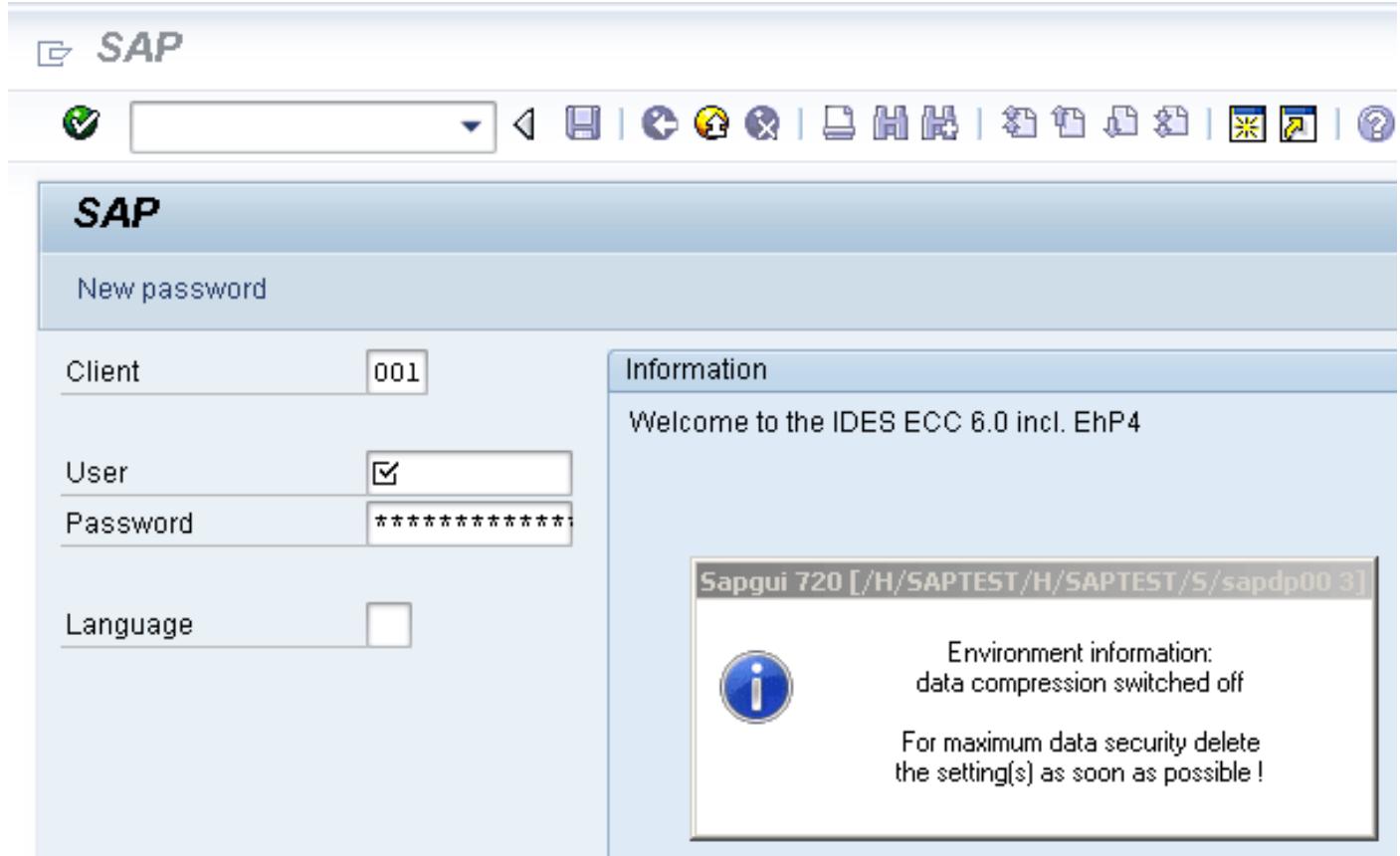


Figure 8.17: Screenshot

Let's see if we can remove the window somehow.

But before this, let's see what we already know.

First: we know that the environment variable `TDW_NOCOMPRESS` is checked somewhere inside the SAPGUI client.

Second: a string like "data compression switched off" must be present somewhere in it.

With the help of the FAR file manager<sup>28</sup> we can find that both of these strings are stored in the SAPguilib.dll file.

So let's open SAPguilib.dll in [IDA](#) and search for the "`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
```

<sup>28</sup><http://go.yurichev.com/17347>

## 8.10. SAP

```
.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
```

The string returned by `chk_env()` via its second argument is then handled by the MFC string functions and then `atoi()`<sup>29</sup> is called. After that, the numerical value is stored in `edi+15h`.

Also take a look at the `chk_env()` function (we gave this name to it manually):

```
.text:64413F20 ; int __cdecl chk_env(char *VarName, int)
.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
```

<sup>29</sup>standard C library function that converts the digits in a string to a number

## 8.10. SAP

```

.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   0xFFFFFFFFh
.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. The `getenv_s()`<sup>30</sup>

function is a Microsoft security-enhanced version of `getenv()`<sup>31</sup>.

There are also some MFC string manipulations.

Lots of other environment variables are checked as well. Here is a list of all variables that are being checked and what SAPGUI would write to its 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 Screen On"
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_FULLSCREEN	"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"

The settings for each variable are written in the array via a pointer in the `EDI` register. `EDI` is 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

```

<sup>30</sup>[MSDN](#)

<sup>31</sup>Standard C library returning environment variable

## 8.10. SAP

```
.text:6440EE0F          cmp    edi, ebx
.text:6440EE11          jz     short loc_6440EE42
.text:6440EE13          push   edi
.text:6440EE14          push   offset aSapguiStoppedA ; "Sapgui stopped after ↴
    ↴ commandline interp"...
.text:6440EE19          push   dword_644F93E8
.text:6440EE1F          call   FEWTraceError
```

Now, can we find the “*data record mode switched on*” string?

Yes, and the only reference is in

```
CDwsGui::PrepareInfoWindow().
```

How do we get know the class/method names? There are a lot of special debugging calls that write to the 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 the contents of this pesky annoying pop-up window's 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]
```

## 8.10. SAP

```

.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 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 loc_64404FD3:
.text:64404FD3          xor    ebx, ebx
.text:64404FD5          inc    ebx
.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 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 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"
     ↴ n"
.text:64405012          mov    ecx, edi

; demangled name: ATL::CStringT::operator+=(char const *)

```

## 8.10. SAP

```
.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\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          mov    ecx, edi
.text:6440503C          cmp    [ebp+var_10], ebx
.text:6440503F          jnz   loc_64405142
.text:64405045          push   offset aForMaximumData ; "\nFor maximum data security\n
                           \ 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], 0FFFFFFFh
.text:6440518C          lea     ecx, [ebp+var_14]

; demangled name: ATL::CStringT::~CStringT()
.text:6440518F          call    ds:mfc90_601
.text:64405195          call    __EH_eptLog3

```

## 8.10. SAP

```
.text:6440519A          retn
.text:6440519A CDwsGui__PrepareInfoWindow endp
```

At the start of the function `ECX` has a pointer to the object (since it is a `thiscall` ([3.19.1 on page 546](#))-type of function). In our case, the object obviously has class type of `CDwsGui`. Depending on the option turned on in the object, a specific message part is to be concatenated with the resulting message.

If the value at address `this+0x3D` is not zero, the compression is off:

```
.text:64405007 loc_64405007:
.text:64405007          cmp     byte ptr [esi+3Dh], 0
.text:6440500B          jz      short bypass
.text:6440500D          push    offset aDataCompression ; "data compression switched off\n"
    ↳ 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 the `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 \n
    ↳ 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 we get SAPGUI working without the pesky annoying pop-up window appearing!

Now let's dig deeper and find a connection between the `0x15` offset in the `load_command_line()` (we gave it this name) function and the `this+0x3D` variable in `CDwsGui::PrepareInfoWindow`. Are we sure the value is the same?

## 8.10. SAP

We are starting to search for all occurrences of the `0x15` value in code. For a small programs like SAPGUI, it sometimes works. Here is the first occurrence we've got:

```
.text:64404C19 sub_64404C19    proc near
.text:64404C19                = dword ptr 4
.text:64404C19 arg_0          push    ebx
.text:64404C19                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 has been called from the function named `CDwsGui::CopyOptions!` And thanks again for debugging information.

But the real answer is in `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: the array filled in the `load_command_line()` function is actually placed in the `CDwsGui` class, but at address `this+0x28`. `0x15 + 0x28` is exactly `0x3D`. OK, we found the point where the value is copied to.

Let's also find the rest of the places where the `0x3D` offset is used. Here is one of them in the `CDwsGui::SapguiRun` function (again, thanks to the 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
```

## 8.10. SAP

.text:64409D7D	lea	ecx, [esi+8]
.text:64409D80	call	CConnectionContext__CreateNetwork

Let's check our findings.

Replace the `setz al` here with the `xor eax, eax / nop` instructions, clear the `TDW_NOCOMPRESS` environment variable and run SAPGUI. Wow! There pesky annoying window is no more (just as expected, because the variable is not set) but in Wireshark we can see that the network packets are not compressed anymore! Obviously, this is the point where the compression flag is to be set in the `CConnectionContext` object.

So, the compression flag is passed in the 5th argument of `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
```

The compression flag is passed here in the 5th argument to the `CNetwork::CNetwork` constructor.

And here is how the `CNetwork` constructor sets the flag in the `CNetwork` object according to its 5th argument *and* another variable which probably could also affect network packets compression.

```
.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 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 address `this+0x3A4`.

Now let's dig through `SAPguilib.dll` for the `0x3A4` value. And here is the second occurrence in `CDws-Gui::OnClientMessageWrite` (endless thanks for the 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
```

## 8.10. SAP

```
.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 in `sub_644055C5`. In it we can only see the call to `memcpy()` and another function named (by IDA) `sub_64417440`.

And, let's take a look inside `sub_64417440`. What we see is:

```
.text:6441747C      push    offset aErrorCsSrcCompre ; "\nERROR: CsRCompress: invalid ↵
    ↪ handle"
.text:64417481      call    eax ; dword_644F94C8
.text:64417483      add     esp, 4
```

Voilà! We've found the function that actually compresses the data. As it was shown in past [32](#), this function is used in SAP and also the open-source MaxDB project. So it is available in source form.

Doing the last check here:

```
.text:64406F79      cmp     dword ptr [ecx+3A4h], 1
.text:64406F80      jnz    compression_flag_is_zero
```

Replace `JNZ` here for an unconditional `JMP`. Remove the environment variable `TDW_NOCOMPRESS`. Voilà!

In Wireshark we see that the client messages are not compressed. The server responses, however, are compressed.

So we found exact connection between the environment variable and the point where data compression routine can be called or bypassed.

## 8.10.2 SAP 6.0 password checking functions

One time when the author of this book have returned again to his SAP 6.0 IDES installed in a VMware box, he figured out that he forgot the password for the SAP\* account, then he have recalled it, but then he got this error message «*Password logon no longer possible - too many failed attempts*», since he've made all these attempts in attempt to recall it.

The first extremely good news was that the full `disp+work.pdb` PDB file is supplied with SAP, and it contain almost everything: function names, structures, types, local variable and argument names, etc. What a lavish gift!

There is `TYPEINFODUMP`<sup>33</sup> utility for converting PDB files into something readable and grepable.

Here is an example of a function information + its arguments + its local variables:

```
FUNCTION ThVmcsEvent
  Address: 10143190  Size: 675 bytes Index: 60483 TypeIndex: 60484
  Type: int NEAR_C ThVmcsEvent (unsigned int, unsigned char, unsigned short*)
Flags: 0
PARAMETER events
  Address: Reg335+288  Size: 4 bytes Index: 60488 TypeIndex: 60489
```

<sup>32</sup><http://go.yurichev.com/17312>

<sup>33</sup><http://go.yurichev.com/17038>

## 8.10. SAP

```
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: *debugging* calls (there are plenty of them) are very useful.

Here you can also notice the *ct\_level* global variable<sup>34</sup>, that reflects the current trace level.

There are a lot of debugging inserts 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:hd1         ; hd1
lea    rdx, aSDpreadmemvalu ; "%s: DpReadMemValue (%d)"
mov    r9d, ebx
call   DpTrcErr
call   DpUnlock
```

If the current trace level is bigger or equal to threshold defined in the code here, a debugging message is to be written to the log files like *dev\_w0*, *dev\_disp*, and other *dev\** files.

Let's try grepping in the file that we have got with the help of the TYPEINFODUMP utility:

<sup>34</sup>More about trace level: <http://go.yurichev.com/17039>

## 8.10. SAP

```
cat "disp+work.pdb.d" | grep FUNCTION | grep -i password
```

We have 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
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 the 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 strings that this function can write to a 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, we noticed that the problem is exactly in it. It is called from the *chckpass()* function —one of the password checking functions.

First, we would like to make sure that we are at the correct point:

Run [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
```

The call path is: *syssigni() -> DylSigni() -> dychkusr() -> usrexist() -> chckpass()*.

The number 0x35 is an error returned in *chckpass()* at that point:

## 8.10. SAP

```
.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! We can successfully login now.

By the way, we can pretend we forgot the password, fixing the *chckpass()* function to return a value of 0 is enough to bypass the 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 0x1402f1060 (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 the *password\_attempt\_limit\_exceeded()* function is that at the very beginning of it, this call can 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()* is used to query the value of some configuration parameter. This function can be called from 1768 different places. It seems that with the help of this information, we can easily find the places in code, the control flow of which can be affected by specific configuration parameters.

It is really sweet. The function names are very clear, much clearer than in the Oracle RDBMS.

It seems that the *disp+work* process is written in C++. Has it been rewritten some time ago?

## 8.11 Oracle RDBMS

### 8.11.1 V\$VERSION table in the Oracle RDBMS

Oracle RDBMS 11.2 is a huge program, its main module `oracle.exe` contain approx. 124,000 functions. For comparison, the Windows 7 x86 kernel (`ntoskrnl.exe`) contains approx. 11,000 functions and the Linux 3.9.8 kernel (with default drivers compiled)—31,000 functions.

Let's start with an easy question. Where does Oracle RDBMS get all this information, when we execute this simple statement in SQL\*Plus:

```
SQL> select * from V$VERSION;
```

And we get:

```
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 can we find the string `V$VERSION`?

In the win32-version, `oracle.exe` file contains the string, it's easy to see. But we can also use the object (.o) files from the Linux version of Oracle RDBMS since, unlike the win32 version `oracle.exe`, the function names (and global variables as well) are preserved there.

So, the `kqf.o` file contains the `V$VERSION` string. The object file is in the main Oracle-library `libserver11.a`.

A reference to this text string can find in the `kqfviw` table stored in the same file, `kqf.o`:

Listing 8.8: `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$WAITSTAT"
.rodata:0800C4A8      dd 4
.rodata:0800C4AC      dd offset _2__STRING_10103_0 ; "NULL"
.rodata:0800C4B0      dd 3
.rodata:0800C4B4      dd 0
.rodata:0800C4B8      dd 195h
.rodata:0800C4BC      dd 4
.rodata:0800C4C0      dd 0
.rodata:0800C4C4      dd 0FFFC1CBh
.rodata:0800C4C8      dd 3
.rodata:0800C4CC      dd 0
.rodata:0800C4D0      dd 0Ah
.rodata:0800C4D4      dd offset _2__STRING_10104_0 ; "V$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 0FFFC003h
.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
```

## 8.11. ORACLE RDBMS

```
.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 analyzing Oracle RDBMS's internals, you may ask yourself, why are the names of the functions and global variable so weird.

Probably, because Oracle RDBMS is a very old product and was developed in C in the 1980s.

And that was a time when the C standard guaranteed that the function names/variables can support only up to 6 characters inclusive: «6 significant initial characters in an external identifier»<sup>35</sup>

Probably, the table `kqfviw` contains most (maybe even all) views prefixed with V\$, these are *fixed views*, present all the time. Superficially, by noticing the 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., in it are described 1023 of all possible *fixed views*.

We are going to return to this number later.

As we can see, there is not much information in these numbers in the fields. The first number is always equals to the name of the view (without the terminating zero). This is correct for each element. But this information is not very useful.

We also know that the information about all fixed views can be retrieved from a *fixed view* named `V$FIXED_VIEW_DEFINITION` (by the way, the information for this view is also taken from the `kqfviw` and `kqfvip` tables.) By the way, there are 1023 elements in those too. Coincidence? No.

```
SQL> select * from V$FIXED_VIEW_DEFINITION where view_name='V$VERSION';

VIEW_NAME
-----
VIEW_DEFINITION
-----

V$VERSION
select BANNER from GV$VERSION where inst_id = USERENV('Instance')
```

So, `V$VERSION` is some kind of a *thunk view* for another view, named `GV$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
```

The tables prefixed with X\$ in the Oracle RDBMS are service tables too, undocumented, cannot be changed by the user and are refreshed dynamically.

<sup>35</sup>Draft ANSI C Standard (ANSI X3J11/88-090) (May 13, 1988) ([yurichev.com](http://yurichev.com))

## 8.11. ORACLE RDBMS

If we search for the text

```
select BANNER from GV$VERSION where inst_id =  
USERENV('Instance')
```

... in the `kqf.o` file, we find it in the `kqfvip` table:

Listing 8.9: `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(index,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(bitandcfflg,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 in it, again, the number we already know.

The second field points to another table that contains the table fields for this *fixed view*. As for `V$VERSION`, this table has only two elements, the first is 6 and the second is the `BANNER` string (the number 6 is this string's length) and after, a *terminating* element that contains 0 and a *null* C string:

Listing 8.10: `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 can get the SQL statements which are executed when the user wants to query information from a specific *fixed view*.

So we can write an oracle tables<sup>36</sup> program, to gather all this information from Oracle RDBMS for Linux's object files. For `V$VERSION`, we find this:

Listing 8.11: Result of oracle tables

```
kqfviw_element.viewname: [V$VERSION] ?: 0x3 0x43 0x1 0xfffffc085 0x4  
kqfvip_element.statement: [select BANNER from GV$VERSION where inst_id = USERENV('Instance')]  
kqfvip_element.params:  
[BANNER]
```

And:

Listing 8.12: Result of oracle tables

```
kqfviw_element.viewname: [GV$VERSION] ?: 0x3 0x26 0x2 0xfffffc192 0x1  
kqfvip_element.statement: [select inst_id, banner from x$version]  
kqfvip_element.params:  
[INST_ID] [BANNER]
```

<sup>36</sup>[yurichev.com](http://yurichev.com)

## 8.11. ORACLE RDBMS

The `GV$VERSION` fixed view is different from `V$VERSION` only in that it has one more field with the identifier *instance*.

Anyway, we are going to stick with the `X$VERSION` table. Just like any other X\$-table, 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 some additional fields, like `ADDR` and `INDX`.

While scrolling `kqf.o` in [IDA](#) we can spot another table that contains a pointer to the `X$VERSION` string, this is `kqftab`:

Listing 8.13: 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 the X\$-table names, apparently, to all Oracle RDBMS 11.2 X\$-tables. But again, we don't have enough information.

It's not clear what does the `kqvt` string stands for.

The `kq` prefix may mean *kernel* or *query*.

`v` apparently stands for *version* and `t` —*type*? Hard to say.

A table with a similar name can be found in `kqf.o`:

Listing 8.14: 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"
```

## 8.11. ORACLE RDBMS

```
.rodata:0808C3CC          kqftap_param <6, offset _2__STRING_5017_0, 601h, 0, 0, 0, 0, 50h, 0, 0> ↵
    ↴ ; "BANNER"
.rodata:0808C3F0          kqftap_param <0, offset _2__STRING_0_0, 0, 0, 0, 0, 0, 0, 0, 0>
```

It contains information about all fields in the `X$VERSION` table. The only reference to this table is in the `kqftap` table:

Listing 8.15: `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 like the pointer to the `X$VERSION` string in the `kqftab` table.

Probably, the `kqftap` and `kqftab` tables complement each other, just like `kqfvip` and `kqfviw`.

We also see a pointer to the `kqvrow()` function. Finally, we got something useful!

So we will add these tables to our oracle tables<sup>37</sup> utility too. For `X$VERSION` we get:

Listing 8.16: Result of oracle tables

```
kqftab_element.name: [X$VERSION] ?: [kqvt] 0x4 0x4 0x4 0xc 0xfffffc075 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 is called 6 times in row (from the `qerfxFetch()` function) while querying the `X$VERSION` table.

Let's run `tracer` in `cc` mode (it comments 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
    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
```

<sup>37</sup>[yurichev.com](http://yurichev.com)

```

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, 0x2ce127a
    jmp    edx               ; EDX=0x2ce1116, 0x2ce11ac, 0x2ce11db, 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]=0xc98cd58
    call   _kghalf            ; tracing nested maximum level (1) reached, skipping this CALL
    mov    esi, ds:_imp_vsnum ; [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

```

## 8.11. ORACLE RDBMS

```

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]=0xacce190
    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

```

## 8.11. ORACLE RDBMS

```

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]=0xc98cd58
    call   _kgfrf              ; 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
_kqvrow_ endp
; EAX=0

```

Now it is easy to see that the row number is passed from outside. The function returns the string, constructing it as follows:

String 1	Using <code>vsnstr</code> , <code>vsnnum</code> , <code>vsnban</code> global variables. Calls <code>sprintf()</code> .
String 2	Calls <code>kkxvsn()</code> .
String 3	Calls <code>lmxver()</code> .
String 4	Calls <code>npinli()</code> , <code>nrtnsvrs()</code> .
String 5	Calls <code>lxvers()</code> .

That's how the corresponding functions are called for determining each module's version.

## 8.11.2 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, the contents of this table are cleared each time it's queried. Are we able to find why? Let's get back to tables we already know: `kqftab` and `kqftap` which were generated with oracle tables<sup>38</sup>'s help, that has all information about the X\$-tables. We can see here that the `ksmlrs()` function is called to prepare this table's elements:

Listing 8.17: Result of oracle tables

```

kqftab_element.name: [X$KSMLRU] ?: [ksmlr] 0x4 0x64 0x11 0xc 0xfffffc0bb 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=[KSMLRSHRP00L] ?: 0xb02 0x0 0x0 0x0 0x4 0x8 0x0
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 `tracer`'s help it is easy to see that 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 calls the `ksmsplu()` at the end. At the end of the `ksmsplu()` function we see a call to `memset()`:

Listing 8.18: ksm.o

```

...
.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

```

<sup>38</sup>[yurichev.com](http://yurichev.com)

## 8.11. ORACLE RDBMS

```
.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 take a risk, block the `memset()` call and see what happens?

Let's run `tracer` with the following options: set breakpoint at `0x434C7A` (the point where the arguments to `memset()` are to be passed), so that `tracer` will set program counter `EIP` to the point where the arguments passed to `memset()` are to be cleared (at `0x434C8A`) It can be said that we just simulate an unconditional jump from address `0x434C7A` to `0x434C8A`.

```
tracer -a:oracle.exe bpx=oracle.exe!0x00434C7A, set(eip,0x00434C8A)
```

(Important: all these addresses are valid only for the win32 version of Oracle RDBMS 11.2)

Indeed, now we can query the `X$KSMLRU` table as many times as we want and it is not being cleared 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 behavior, but as an experiment for locating a piece of code that we need, it perfectly suits our needs!

### 8.11.3 V\$TIMER table in Oracle RDBMS

`V$TIMER` is another *fixed view* that reflects 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 <sup>39</sup>)

It is interesting that the periods are different for Oracle for win32 and for Linux. Will we be able to find the function that generates this value?

As we can see, this information is finally taken from the `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
```

<sup>39</sup><http://go.yurichev.com/17088>

## 8.11. ORACLE RDBMS

```
-----  
VIEW_DEFINITION  
-----  
  
GV$TIMER  
select inst_id,ksutmtim from x$ksutm
```

Now we are stuck in a small problem, there are no references to value generating function(s) in the tables `kqftab` / `kqftap`:

Listing 8.19: Result of oracle tables

```
kqftab_element.name: [X$KSUTM] ?: [ksutm] 0x1 0x4 0x4 0x0 0xfffffc09b 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
```

When we try to find the string `KSUTMTIM`, we see 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  
ret  
kqfd_DRN_ksutm_c endp
```

The `kqfd_DRN_ksutm_c()` function is mentioned in the

`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 is a function `ksugtm()` referenced here. Let's see what's in it (Linux x86):

Listing 8.20: 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]
```

## 8.11. ORACLE RDBMS

```
mov    [edx], eax
mov    eax, 4
mov    esp, ebp
pop    ebp
retn
ksugtm endp
```

The code in the win32 version is almost the same.

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 8.21: [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 returned via the 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
```

## 8.11. ORACLE RDBMS

```
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 ([3.9 on page 496](#))

And win32-version:

```
_slgcs proc near ; CODE XREF: _dbgefgHtElResetCount+15
           ; _dbgcrunActions+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, 0CCCCCCCCDh
    mul   edx
    shr   edx, 3
    mov   eax, edx
    mov   esp, ebp
    pop   ebp
    retn
_slgcs endp
```

It is just the result of `GetTickCount()` <sup>40</sup> divided by 10 ([3.9 on page 496](#)).

Voilà! That's why the win32 version and the Linux x86 version show different results, because they are generated by different OS functions.

*Drain* apparently implies *connecting* a specific table column to a specific function.

We will add support of the table `kqfd_tabl_registry_0` to oracle tables<sup>41</sup>, now we can see how the table column's variables are *connected* to a 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 stands for, *open*, and *DRN*, apparently, for *drain*.

<sup>40</sup>[MSDN](#)

<sup>41</sup>[yurichev.com](#)

## 8.12 Handwritten assembly code

### 8.12.1 EICAR test file

This .COM-file is intended for testing antivirus software, it is possible to run in MS-DOS and it prints this string: "EICAR-STANDARD-ANTIVIRUS-TEST-FILE!"<sup>42</sup>.

Its important property is that it's consists entirely of printable ASCII-symbols, which, in turn, makes it possible to create it in any text editor:

```
X50!P%@AP[4\PZX54(P^)7CC]7\$EICAR-STANDARD-ANTIVIRUS-TEST-FILE!\$H+H*
```

Let's decompile it:

```
; initial conditions: SP=0FFFFh, 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 = FFFFh and SS:[FFFF] = 214Fh
0105 25 40 41    and     ax, 4140h
; AX = 140h, SP = FFFFh and SS:[FFFF] = 214Fh
0108 50          push    ax
; AX = 140h, SP = FFFCh, SS:[FFFC] = 140h and SS:[FFFF] = 214Fh
0109 5B          pop     bx
; AX = 140h, BX = 140h, SP = FFFFh and SS:[FFFF] = 214Fh
010A 34 5C          xor    al, 5Ch
; AX = 11Ch, BX = 140h, SP = FFFFh and SS:[FFFF] = 214Fh
010C 50          push    ax
010D 5A          pop     dx
; AX = 11Ch, BX = 140h, DX = 11Ch, SP = FFFFh and SS:[FFFF] = 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
```

We will add comments about the 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, the address of which is passed in DS:DX. By the way, the string has to be terminated with the '\$' sign. Apparently, it's inherited from CP/M and this function was left 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 the register (AH and DX) values that we need;
- preparing INT 21 and INT 20 opcodes in memory;

<sup>42</sup>wikipedia

## 8.13. DEMOS

- executing INT 21 and INT 20.

By the way, this technique is widely used in shellcode construction, when one have to pass x86 code in string form.

Here is also a list of all x86 instructions which have printable opcodes: [.1.6 on page 1018](#).

## 8.13 Demos

Demos (or demomaking) were an excellent exercise in mathematics, computer graphics programming and very tight x86 hand coding.

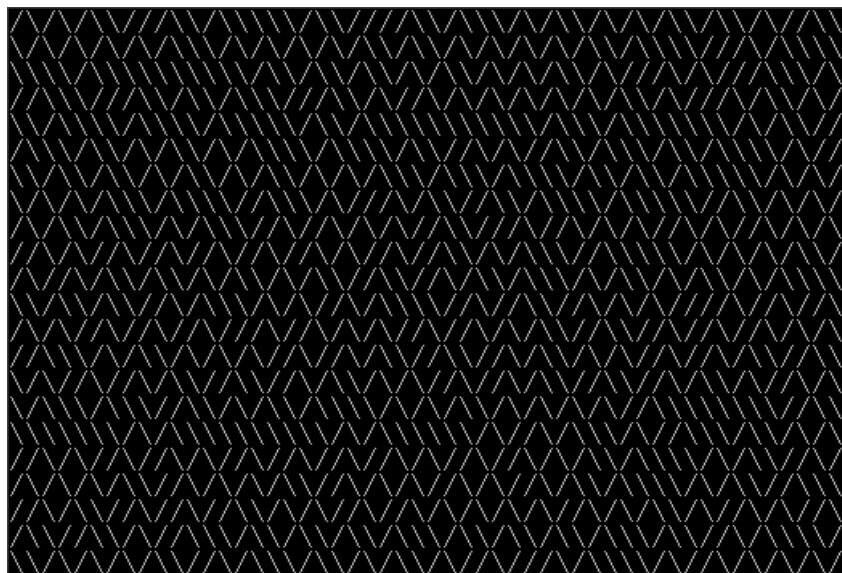
### 8.13.1 10 PRINT CHR\$(205.5+RND(1)); : GOTO 10

All examples here are MS-DOS .COM files.

In [Nick Montfort et al, *10 PRINT CHR\$(205.5+RND(1)); : GOTO 10*, (The MIT Press:2012)] <sup>43</sup>

we can read about one of the most simple possible random maze generators.

It just prints a slash or backslash characters randomly and endlessly, resulting in something like this:



There are a few known implementations for 16-bit x86.

#### Trixter's 42 byte version

The listing was taken from his website<sup>44</sup>, but the comments are mine.

```
00000000: B001      mov     al,1      ; set 40x25 video mode
00000002: CD10      int     010
00000004: 30FF      xor     bh,bh    ; set video page 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
00000011: 88C4      mov     ah,al
00000013: E440      in     al,040
00000015: 9D        popf   ; enable interrupts by restoring IF flag
```

<sup>43</sup>Also available as <http://go.yurichev.com/17286>

<sup>44</sup><http://go.yurichev.com/17305>

### 8.13. DEMOS

```
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        00000022
; 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     00000009 ; loop 2000 times
00000028: CD20      int       020      ; exit to DOS
```

The pseudo-random value here is in fact the time that has passed from the system's boot, taken from the 8253 time chip, the value increases by one 18.2 times per second.

By writing zero to port `43h`, we send the command "select counter 0", "counter latch", "binary counter" (not a **BCD** value).

The interrupts are enabled back with the `POPF` instruction, which restores the `IF` flag as well.

It is not possible to use the `IN` instruction with registers other than `AL`, hence the shuffling.

#### My attempt to reduce Trixter's version: 27 bytes

We can say that since we use the timer not to get a precise time value, but a pseudo-random one, we do not need to spend time (and code) to disable the interrupts.

Another thing we can say is that we need only one bit from the low 8-bit part, so let's read only it.

We can reduced the code slightly and we'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 '\'
0000000F: 7202    jc      00000013
00000011: B02F    mov     al,02F  ; prepare '/'
; output character to screen
00000013: B40E    mov     ah,00E
00000015: CD10    int     010
00000017: E2EA    loop    00000003
; exit to DOS
00000019: CD20    int     020
```

#### Taking random memory garbage as a source of randomness

Since it is MS-DOS, there is no memory protection at all, we can read from whatever address we want. Even more than that: a simple `LDSB` instruction reads a byte from the `DS:SI` address, but it's not a problem if the registers' values are not set up, let it read 1) random bytes; 2) from a random place in memory!

It is suggested in Trixter's webpage<sup>45</sup> to use `LDSB` without any setup.

It is also suggested that the `SCASB`

instruction can be used instead, because it sets a flag according to the byte it reads.

Another idea to minimize the code is to use the `INT 29h` DOS syscall, which just prints the character stored in the `AL` register.

<sup>45</sup><http://go.yurichev.com/17305>

## 8.13. DEMOS

That is what Peter Ferrie and Andrey "herm1t" Baranovich did (11 and 10 bytes) <sup>46</sup>:

Listing 8.22: 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      00000007  
00000005: B02F      mov     al,02F    ; '/'  
00000007: CD29      int     029      ; output AL to screen  
00000009: EBF5      jmp     00000000 ; loop endlessly
```

SCASB also uses the value in the AL register, it subtract a random memory byte's value from the 5Ch value in AL. JP is a rare instruction, here it used for checking the 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 a random memory byte, but by a sum of bits, this (hopefully) makes the result more distributed.

It is possible to make this even shorter by using the undocumented x86 instruction SALC (AKA SETALC) ("Set AL CF"). It was introduced in the NEC V20 CPU and sets AL to 0xFF if CF is 1 or to 0 if otherwise.

Listing 8.23: Peter Ferrie: 10 bytes

```
; AL is random at this point  
00000000: AE        scasb  
; CF is set according 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    00000000 ; loop endlessly
```

So it is possible to get rid of conditional jumps at all. The ASCII code of backslash ("\") is 0x5C and 0x2F for slash ("/"). So we have to convert one (pseudo-random) bit in the CF flag to a value of 0x5C or 0x2F.

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 we just output it to the screen.

## Conclusion

It is also worth mentioning that the result may be different in DOSBox, Windows NT and even MS-DOS, due to different conditions: the timer chip can be emulated differently and the initial register contents may be different as well.

<sup>46</sup><http://go.yurichev.com/17087>

## 8.13.2 Mandelbrot set

You know, if you magnify the coastline, it still looks like a coastline, and a lot of other things have this property. Nature has recursive algorithms that it uses to generate clouds and Swiss cheese and things like that.

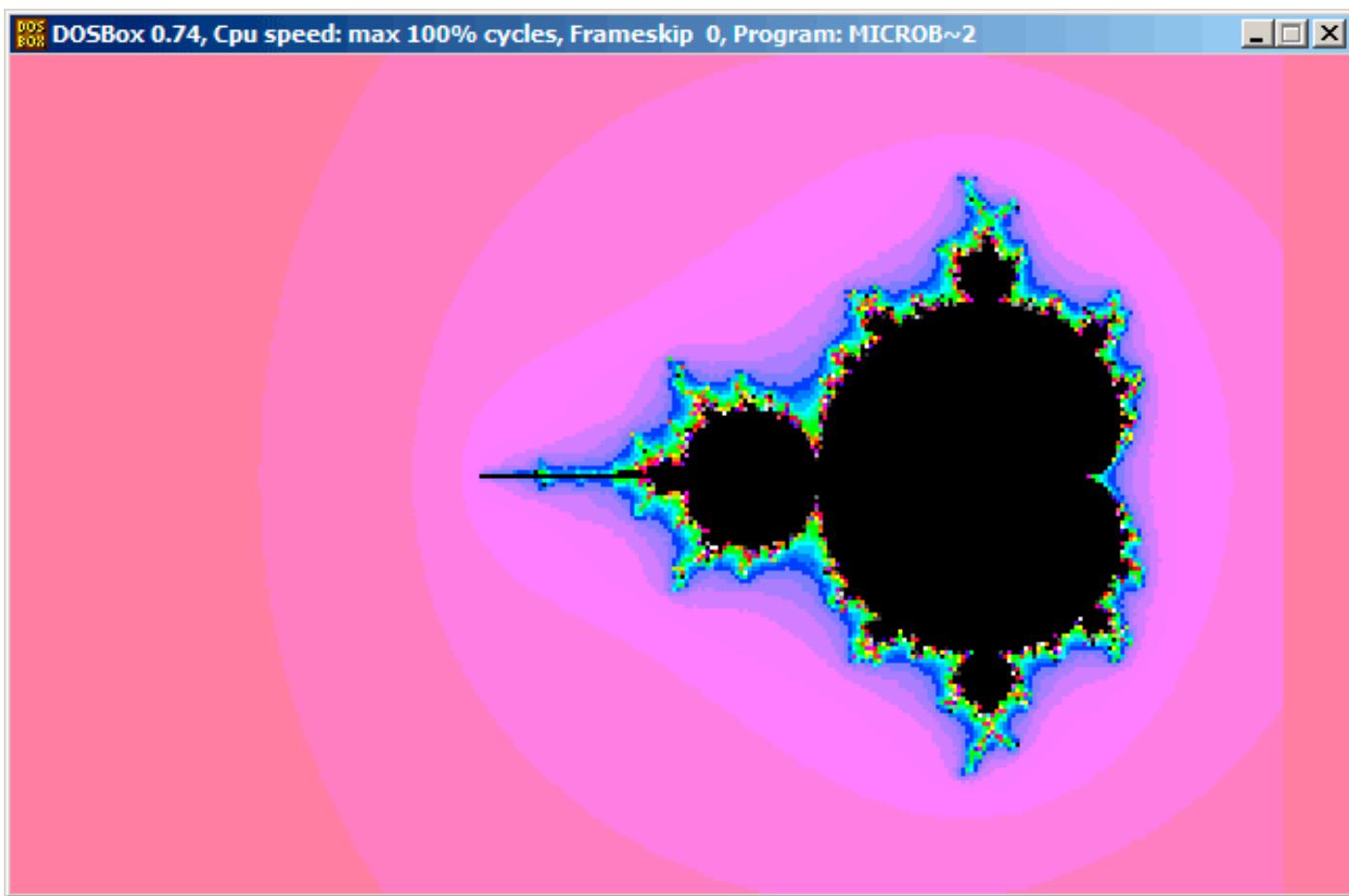
Donald Knuth, interview (1993)

Mandelbrot set is a fractal, which exhibits self-similarity.

When you increase scale, you see that this characteristic pattern repeating infinitely.

Here is a demo<sup>47</sup> written by “Sir\_Lagsalot” in 2009, that draws the Mandelbrot set, which is just a x86 program with executable file size of only 64 bytes. There are only 30 16-bit x86 instructions.

Here it is what it draws:



Let's try to understand how it works.

### Theory

#### A word about complex numbers

A complex number is a number that consists of two parts—real (Re) and imaginary (Im).

The complex plane is a two-dimensional plane where any complex number can be placed: the real part is one coordinate and the imaginary part is the other.

Some basic rules we have to keep in mind:

- Addition:  $(a + bi) + (c + di) = (a + c) + (b + d)i$

In other words:

<sup>47</sup>Download it [here](#),

### 8.13. DEMOS

$$\operatorname{Re}(sum) = \operatorname{Re}(a) + \operatorname{Re}(b)$$

$$\operatorname{Im}(sum) = \operatorname{Im}(a) + \operatorname{Im}(b)$$

- Multiplication:  $(a + bi)(c + di) = (ac - bd) + (bc + ad)i$

In other words:

$$\operatorname{Re}(product) = \operatorname{Re}(a) \cdot \operatorname{Re}(c) - \operatorname{Re}(b) \cdot \operatorname{Re}(d)$$

$$\operatorname{Im}(product) = \operatorname{Im}(b) \cdot \operatorname{Im}(c) + \operatorname{Im}(a) \cdot \operatorname{Im}(d)$$

- Square:  $(a + bi)^2 = (a + bi)(a + bi) = (a^2 - b^2) + (2ab)i$

In other words:

$$\operatorname{Re}(square) = \operatorname{Re}(a)^2 - \operatorname{Im}(a)^2$$

$$\operatorname{Im}(square) = 2 \cdot \operatorname{Re}(a) \cdot \operatorname{Im}(a)$$

## How to draw the Mandelbrot set

The Mandelbrot set is a set of points for which the  $z_{n+1} = z_n^2 + c$  recursive sequence (where  $z$  and  $c$  are complex numbers and  $c$  is the starting value) does not approach infinity.

In plain English language:

- Enumerate all points on screen.
- Check if the specific point is in the Mandelbrot set.
- Here is how to check it:
  - Represent the point as a complex number.
  - Calculate the square of it.
  - Add the starting value of the point to it.
  - Does it go off limits? If yes, break.
  - Move the point to the new place at the coordinates we just calculated.
  - Repeat all this for some reasonable number of iterations.
- The point is still in limits? Then draw the point.
- The point has eventually gone off limits?
  - (For a black-white image) do not draw anything.
  - (For a colored image) transform the number of iterations to some color. So the color shows the speed with which point has gone off limits.

Here is Pythonesque algorithm for both complex and integer number representations:

Listing 8.24: For complex numbers

```
def check_if_is_in_set(P):
    P_start=P
    iterations=0

    while True:
        if (P>bounds):
            break
        P=P^2+P_start
        if iterations > max_iterations:
            break
        iterations++

    return iterations

# black-white
for each point on screen P:
    if check_if_is_in_set (P) < max_iterations:
        draw point
```

## 8.13. DEMOS

```
# colored
for each point on screen P:
    iterations = if check_if_is_in_set (P)
    map iterations to color
    draw color point
```

The integer version is where the operations on complex numbers are replaced with integer operations according to the rules which were explained above.

Listing 8.25: For integer numbers

```
def check_if_is_in_set(X, Y):
    X_start=X
    Y_start=Y
    iterations=0

    while True:
        if (X^2 + Y^2 > bounds):
            break
        new_X=X^2 - Y^2 + X_start
        new_Y=2*X*Y + Y_start
        if iterations > max_iterations:
            break
        iterations++

    return iterations

# black-white
for X = min_X to max_X:
    for Y = min_Y to max_Y:
        if check_if_is_in_set (X,Y) < max_iterations:
            draw point at X, Y

# colored
for X = min_X to max_X:
    for Y = min_Y to max_Y:
        iterations = if check_if_is_in_set (X,Y)
        map iterations to color
        draw color point at X,Y
```

Here is also a C# source which is present in the Wikipedia article<sup>48</sup>, but we'll modify it so it will print the iteration numbers instead of some symbol <sup>49</sup>:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Mnobj
{
    class Program
    {
        static void Main(string[] args)
        {
            double realCoord, imagCoord;
            double realTemp, imagTemp, realTemp2, arg;
            int iterations;
            for (imagCoord = 1.2; imagCoord >= -1.2; imagCoord -= 0.05)
            {
                for (realCoord = -0.6; realCoord <= 1.77; realCoord += 0.03)
                {
                    iterations = 0;
                    realTemp = realCoord;
                    imagTemp = imagCoord;
                    arg = (realCoord * realCoord) + (imagCoord * imagCoord);
                    while ((arg < 2*2) && (iterations < 40))
                    {

```

<sup>48</sup>[wikipedia](#)

<sup>49</sup>Here is also the executable file: [beginners.re](#)

## 8.13. DEMOS

```
    realTemp2 = (realTemp * realTemp) - (imagTemp * imagTemp) - realCoord;
    imagTemp = (2 * realTemp * imagTemp) - imagCoord;
    realTemp = realTemp2;
    arg = (realTemp * realTemp) + (imagTemp * imagTemp);
    iterations += 1;
}
Console.WriteLine("{0,2:D} ", iterations);
}
Console.WriteLine("\n");
}
Console.ReadKey();
}
}
```

Here is the resulting file, which is too wide to be included here:

[beginners.re](#).

The maximal number of iterations is 40, so when you see 40 in this dump, it means that this point has been wandering for 40 iterations but never got off limits.

A number  $n$  less than 40 means that point remained inside the bounds only for  $n$  iterations, then it went outside them.

### 8.13. DEMOS

There is a cool demo available at <http://go.yurichev.com/17309>, which shows visually how the point moves on the plane at each iteration for some specific point. Here are two screenshots.

First, we've clicked inside the yellow area and saw that the trajectory (green line) eventually swirls at some point inside:

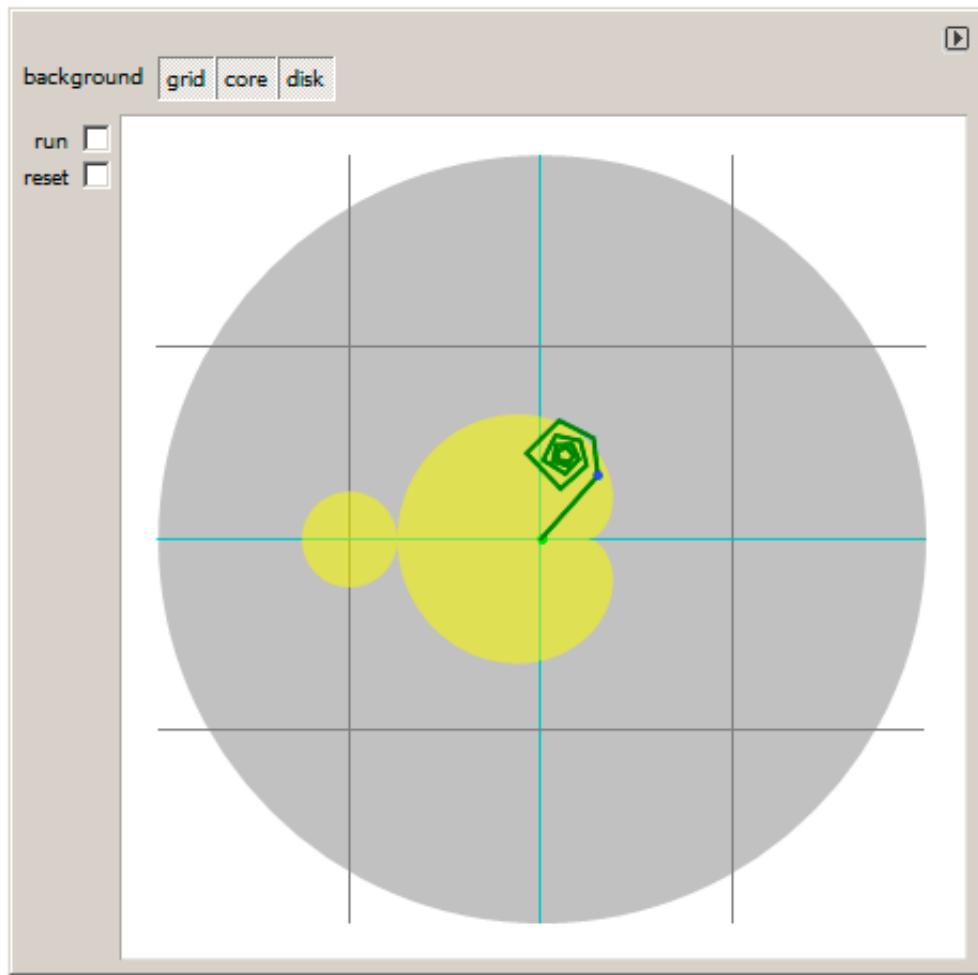


Figure 8.18: Click inside yellow area

This implies that the point we've clicked belongs to the Mandelbrot set.

### 8.13. DEMOS

Then we've clicked outside the yellow area and saw a much more chaotic point movement, which quickly went off bounds:

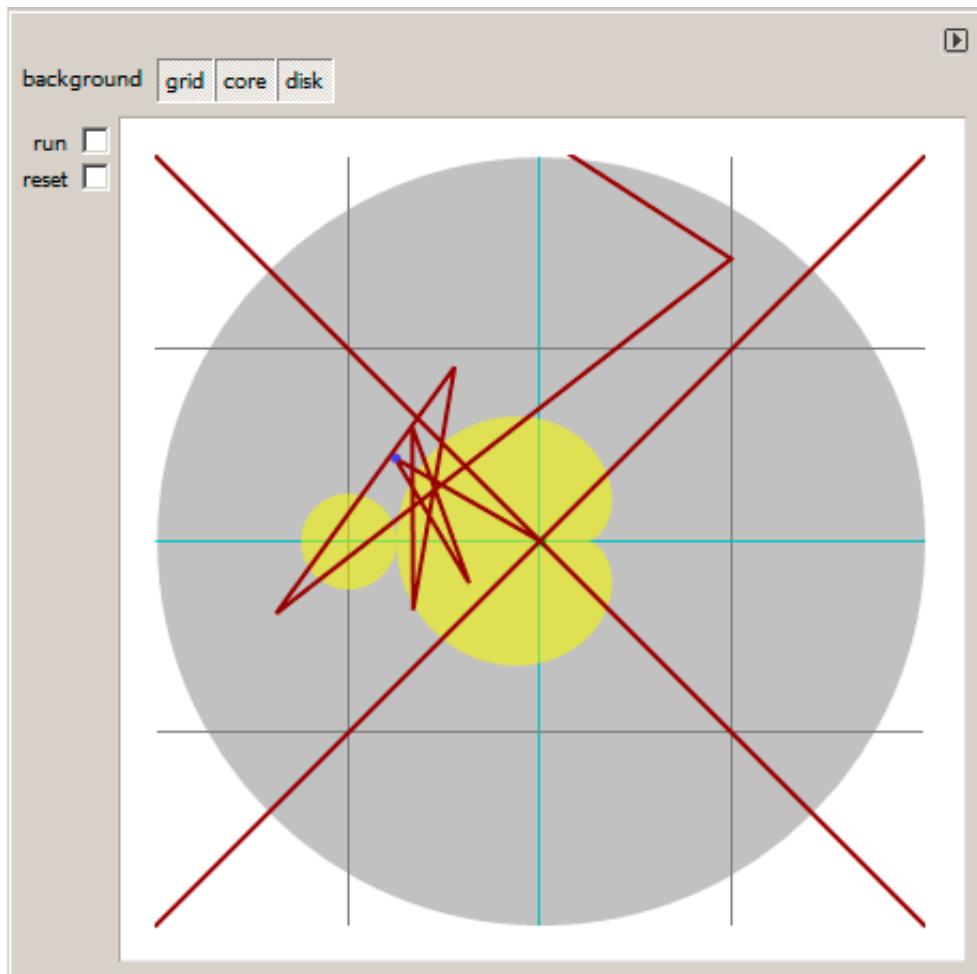


Figure 8.19: Click outside yellow area

This means the point doesn't belong to Mandelbrot set.

Another good demo is available here: <http://go.yurichev.com/17310>.

## 8.13. DEMOS

### Let's get back to the demo

The demo, although very tiny (just 64 bytes or 30 instructions), implements the common algorithm described here, but using some coding tricks.

The source code is easily downloadable, so here is it, but let's also add comments:

Listing 8.26: Commented source code

```
1 ; X is column on screen
2 ; Y is row on screen
3
4
5 ; X=0, Y=0           X=319, Y=0
6 ; +----->
7 ;
8 ;
9 ;
10 ;
11 ;
12 ;
13 ; v
14 ; X=0, Y=199       X=319, Y=199
15
16
17 ; switch to VGA 320*200*256 graphics mode
18 mov al,13h
19 int 10h
20 ; initial BX is 0
21 ; initial DI is 0xFFFFE
22 ; DS:BX (or DS:0) is pointing to Program Segment Prefix at this moment
23 ; ... first 4 bytes of which are CD 20 FF 9F
24 les ax,[bx]
25 ; ES:AX=9FFF:20CD
26
27 FillLoop:
28 ; set DX to 0. CWD works as: DX:AX = sign_extend(AX).
29 ; AX here 0x20CD (at startup) or less than 320 (when getting back after loop),
30 ; so DX will always be 0.
31 cwd
32 mov ax,di
33 ; AX is current pointer within VGA buffer
34 ; divide current pointer by 320
35 mov cx,320
36 div cx
37 ; DX (start_X) - remainder (column: 0..319); AX - result (row: 0..199)
38 sub ax,100
39 ; AX=AX-100, so AX (start_Y) now is in range -100..99
40 ; DX is in range 0..319 or 0x0000..0x013F
41 dec dh
42 ; DX now is in range 0xFF00..0x003F (-256..63)
43
44 xor bx,bx
45 xor si,si
46 ; BX (temp_X)=0; SI (temp_Y)=0
47
48 ; get maximal number of iterations
49 ; CX is still 320 here, so this is also maximal number of iteration
50 MandelLoop:
51 mov bp,si      ; BP = temp_Y
52 imul si,bx    ; SI = temp_X*temp_Y
53 add si,si     ; SI = SI*2 = (temp_X*temp_Y)*2
54 imul bx,bx    ; BX = BX^2 = temp_X^2
55 jo MandelBreak ; overflow?
56 imul bp,bp    ; BP = BP^2 = temp_Y^2
57 jo MandelBreak ; overflow?
58 add bx,bp     ; BX = BX+BP = temp_X^2 + temp_Y^2
59 jo MandelBreak ; overflow?
60 sub bx,bp     ; BX = BX-BP = temp_X^2 + temp_Y^2 - temp_Y^2 = temp_X^2
61 sub bx,bp     ; BX = BX-BP = temp_X^2 - temp_Y^2
```

## 8.13. DEMOS

```
63 ; correct scale:
64 sar bx,6      ; BX=BX/64
65 add bx,dx     ; BX=BX+start_X
66 ; now temp_X = temp_X^2 - temp_Y^2 + start_X
67 sar si,6      ; SI=SI/64
68 add si,ax     ; SI=SI+start_Y
69 ; now temp_Y = (temp_X*temp_Y)*2 + start_Y
70
71 loop MandelLoop
72
73 MandelBreak:
74 ; CX=iterations
75 xchg ax,cx
76 ; AX=iterations. store AL to VGA buffer at ES:[DI]
77 stosb
78 ; stosb also increments DI, so DI now points to the next point in VGA buffer
79 ; jump always, so this is eternal loop here
80 jmp FillLoop
```

### Algorithm:

- Switch to 320\*200 VGA video mode, 256 colors.  $320 * 200 = 64000$  (0xFA00).

Each pixel is encoded by one byte, so the buffer size is 0xFA00 bytes. It is addressed using the ES:DI registers pair.

ES must be 0xA000 here, because this is the segment address of the VGA video buffer, but storing 0xA000 to ES requires at least 4 bytes (`PUSH 0A000h / POP ES`). You can read more about the 16-bit MS-DOS memory model here: [10.6 on page 990](#).

Assuming that BX is zero here, and the Program Segment Prefix is at the zeroth address, the 2-byte `LES AX, [BX]` instruction stores 0x20CD to AX and 0xFFFF to ES.

So the program starts to draw 16 pixels (or bytes) before the actual video buffer. But this is MS-DOS, there is no memory protection, so a write happens into the very end of conventional memory, and usually, there is nothing important. That's why you see a red strip 16 pixels wide at the right side. The whole picture is shifted left by 16 pixels. This is the price of saving 2 bytes.

- An infinite loop processes each pixel.

Probably, the most common way to enumerate all pixels on the screen is with two loops: one for the X coordinate, another for the Y coordinate. But then you'll need to multiply the coordinates to address a byte in the VGA video buffer.

The author of this demo decided to do it otherwise: enumerate all bytes in the video buffer by using one single loop instead of two, and get the coordinates of the current point using division. The resulting coordinates are: X in the range of -256..63 and Y in the range of -100..99. You can see on the screenshot that the picture is somewhat shifted to the right part of screen.

That's because the biggest heart-shaped black hole usually appears on coordinates 0,0 and these are shifted here to right. Could the author just subtract 160 from the value to get X in the range of -160..159? Yes, but the instruction `SUB DX, 160` takes 4 bytes, while `DEC DH` —2 bytes (which subtracts 0x100 (256) from DX). So the whole picture is shifted for the cost of another 2 bytes of saved space.

- Check, if the current point is inside the Mandelbrot set. The algorithm is the one that has been described here.
- The loop is organized using the `LOOP` instruction, which uses the CX register as counter.

The author could set the number of iterations to some specific number, but he didn't: 320 is already present in CX (has been set at line 35), and this is good maximal iteration number anyway. We save here some space by not the reloading CX register with another value.

- `IMUL` is used here instead of `MUL`, because we work with signed values: keep in mind that the 0,0 coordinates has to be somewhere near the center of the screen.

It's the same with `SAR` (arithmetic shift for signed values): it's used instead of `SHR`.

- Another idea is to simplify the bounds check. We must check a coordinate pair, i.e., two variables. What the author does is to checks thrice for overflow: two squaring operations and one addition.

## 8.13. DEMOS

Indeed, we use 16-bit registers, which hold signed values in the range of -32768..32767, so if any of the coordinates is greater than 32767 during the signed multiplication, this point is definitely out of bounds: we jump to the `MandelBreak` label.

- There is also a division by 64 (SAR instruction). 64 sets scale.

Try to increase the value and you can get a closer look, or to decrease if for a more distant look.

- We are at the `MandelBreak` label, there are two ways of getting here: the loop ended with CX=0 (the point is inside the Mandelbrot set); or because an overflow has happened (CX still holds some value). Now we write the low 8-bit part of CX (CL) to the video buffer.

The default palette is rough, nevertheless, 0 is black: hence we see black holes in the places where the points are in the Mandelbrot set. The palette can be initialized at the program's start, but keep in mind, this is only a 64 bytes program!

- The program runs in an infinite loop, because an additional check where to stop, or any user interface will result in additional instructions.

Some other optimization tricks:

- The 1-byte CWD is used here for clearing DX instead of the 2-byte `XOR DX, DX` or even the 3-byte `MOV DX, 0`.
- The 1-byte `XCHG AX, CX` is used instead of the 2-byte `MOV AX,CX`. The current value of AX is not needed here anyway.
- DI (position in video buffer) is not initialized, and it is 0xFFFF at the start <sup>50</sup>.

That's OK, because the program works for all DI in the range of 0..0xFFFF eternally, and the user can't notice that it is started off the screen (the last pixel of a 320\*200 video buffer is at address 0xF9FF). So some work is actually done off the limits of the screen.

Otherwise, you'll need an additional instructions to set DI to 0 and check for the video buffer's end.

## My “fixed” version

Listing 8.27: My “fixed” version

```
1 org 100h
2 mov al,13h
3 int 10h
4
5 ; set palette
6 mov dx, 3c8h
7 mov al, 0
8 out dx, al
9 mov cx, 100h
10 inc dx
11 l00:
12 mov al, cl
13 shl ax, 2
14 out dx, al ; red
15 out dx, al ; green
16 out dx, al ; blue
17 loop l00
18
19 push 0a000h
20 pop es
21
22 xor di, di
23
24 FillLoop:
25 cwd
26 mov ax,di
27 mov cx,320
28 div cx
29 sub ax,100
30 sub dx,160
```

<sup>50</sup>More information about initial register values: <http://go.yurichev.com/17004>

## 8.13. DEMOS

```
31
32 xor bx,bx
33 xor si,si
34
35 MandelLoop:
36 mov bp,si
37 imul si,bx
38 add si,si
39 imul bx,bx
40 jo MandelBreak
41 imul bp,bp
42 jo MandelBreak
43 add bx,bp
44 jo MandelBreak
45 sub bx,bp
46 sub bx,bp
47
48 sar bx,6
49 add bx,dx
50 sar si,6
51 add si,ax
52
53 loop MandelLoop
54
55 MandelBreak:
56 xchg ax,cx
57 stosb
58 cmp di, 0FA00h
59 jb FillLoop
60
61 ; wait for keypress
62 xor ax,ax
63 int 16h
64 ; set text video mode
65 mov ax, 3
66 int 10h
67 ; exit
68 int 20h
```

The author of these lines made an attempt to fix all these oddities: now the palette is smooth grayscale, the video buffer is at the correct place (lines 19..20), the picture is drawn on center of the screen (line 30), the program eventually ends and waits for the user's keypress (lines 58..68).

But now it's much bigger: 105 bytes (or 54 instructions) <sup>51</sup>.

<sup>51</sup>You can experiment by yourself: get DosBox and NASM and compile it as: `nasm fiole.asm -fbin -o file.com`

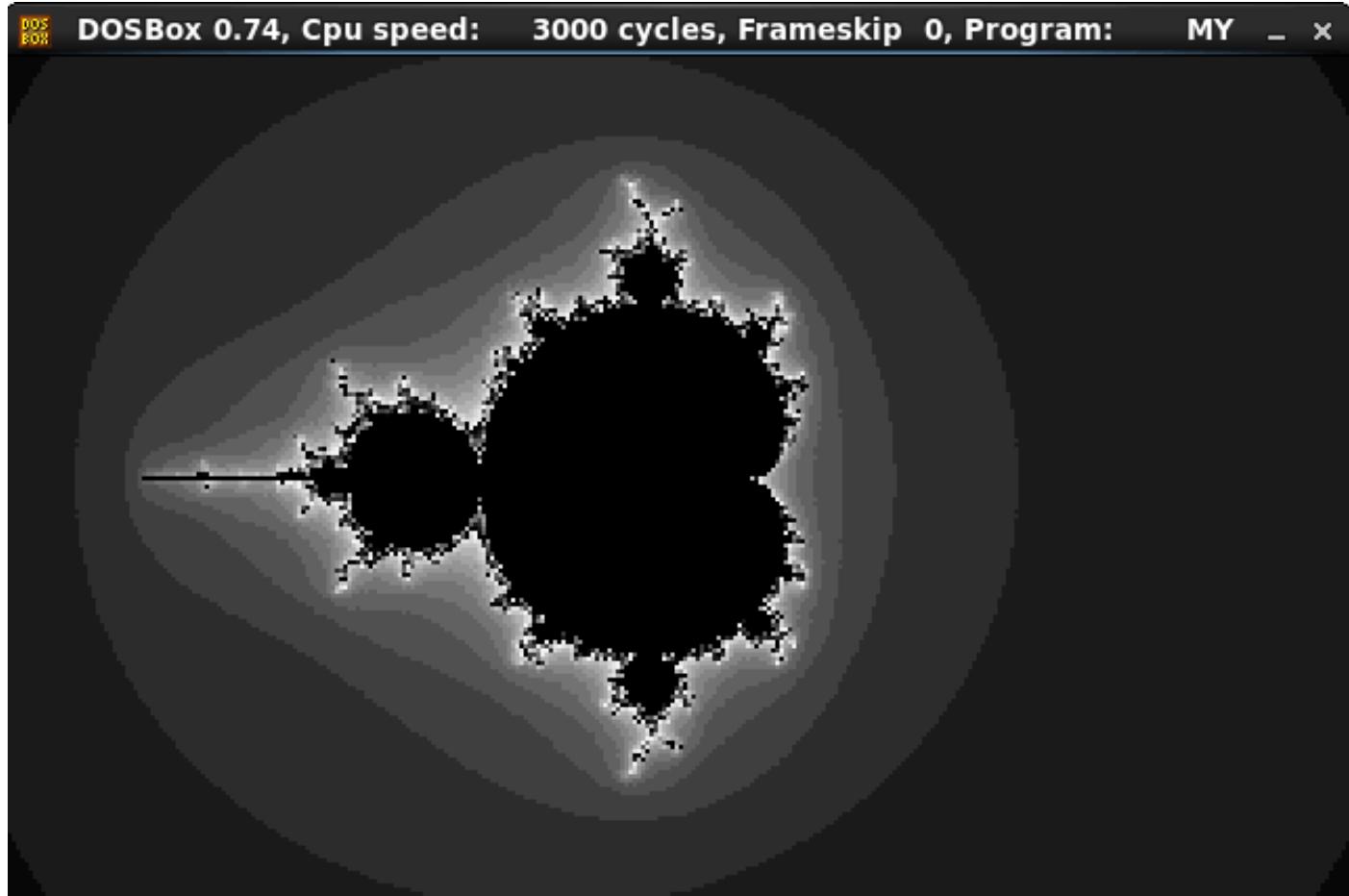


Figure 8.20: My “fixed” version

## 8.14 Other examples

An example about Z3 and manual decompilation was here. It is (temporarily) moved there: [http://yurichev.com/tmp/SAT\\_SMT\\_DRAFT.pdf](http://yurichev.com/tmp/SAT_SMT_DRAFT.pdf).

# **Chapter 9**

## **Examples of reversing proprietary file formats**

### **9.1 Primitive XOR-encryption**

### 9.1.1 Norton Guide: simplest possible 1-byte XOR encryption

Norton Guide<sup>1</sup> was popular in the epoch of MS-DOS, it was a resident program that worked as a hypertext reference manual.

Norton Guide's databases are files with the extension .ng, the contents of which look encrypted:

Figure 9.1: Very typical look

Why did we think that it's encrypted but not compressed?

We see that the 0x1A byte (looking like “→”) occurs often, it would not be possible in a compressed file.

We also see long parts that consist only of Latin letters, and they look like strings in an unknown language.

<sup>1</sup>wikipedia

## 9.1. PRIMITIVE XOR-ENCRYPTION

Since the 0x1A byte occurs so often, we can try to decrypt the file, assuming that it's encrypted by the simplest XOR-encryption.

If we apply XOR with the 0x1A constant to each byte in Hiew, we can see familiar English text strings:

The screenshot shows the Hiew X86.NG editor interface. The title bar says "Hiew: X86.NG". The status bar at the bottom shows memory addresses from 1 to 11. The main window displays memory dump and assembly code. The assembly code is in Intel syntax. The memory dump shows many 0x1A bytes, which are replaced by readable ASCII characters in the assembly code. The assembly code includes CPU instructions like "CPU Instruction set Registers, privilege Exceptions, Addressing modes, Opcodes", FPU instructions, MMX instructions, and other assembly language constructs. The assembly code is mostly in Russian, with some English words like "privilege Exceptions" and "Addressing modes". The memory dump shows the raw binary data, with many 0x1A bytes appearing where they were replaced by ASCII characters in the assembly code.

Figure 9.2: Hiew XORing with 0x1A

XOR encryption with one single constant byte is the simplest possible encryption method, which is, nevertheless, encountered sometimes.

Now we understand why the 0x1A byte is occurring so often: because there are so many zero bytes and they were replaced by 0x1A in encrypted form.

But the constant might be different. In this case, we could try every constant in the 0..255 range and look for something familiar in the decrypted file. 256 is not so much.

More about Norton Guide's file format: <http://go.yurichev.com/17317>.

## Entropy

A very important property of such primitive encryption systems is that the information entropy of the encrypted/decrypted block is the same.

Here is my analysis in Wolfram Mathematica 10.

## 9.1. PRIMITIVE XOR-ENCRYPTION

Listing 9.1: Wolfram Mathematica 10

```
In[1]:= input = BinaryReadList["X86.NG"];  
In[2]:= Entropy[2, input] // N  
Out[2]= 5.62724  
  
In[3]:= decrypted = Map[BitXor[#, 16^^1A] &, input];  
  
In[4]:= Export["X86_decrypted.NG", decrypted, "Binary"];  
  
In[5]:= Entropy[2, decrypted] // N  
Out[5]= 5.62724  
  
In[6]:= Entropy[2, ExampleData[{"Text", "ShakespearesSonnets"}]] // N  
Out[6]= 4.42366
```

What we do here is load the file, get its entropy, decrypt it, save it and get the entropy again (the same!).

Mathematica also offers some well-known English language texts for analysis.

So we also get the entropy of Shakespeare's sonnets, and it is close to the entropy of the file we just analyzed.

The file we analyzed consists of English language sentences, which are close to the language of Shakespeare.

And the XOR-ed bitwise English language text has the same entropy.

However, this is not true when the file is XOR-ed with a pattern larger than one byte.

The file we analyzed can be downloaded here: <http://go.yurichev.com/17350>.

### One more word about base of entropy

Wolfram Mathematica calculates entropy with base of  $e$  (base of the natural logarithm), and the UNIX `ent` utility<sup>2</sup> uses base 2.

So we set base 2 explicitly in `Entropy` command, so Mathematica will give us the same results as the `ent` utility.

<sup>2</sup><http://www.fourmilab.ch/random/>

## 9.1.2 Simplest possible 4-byte XOR encryption

If a longer pattern was used for XOR-encryption, for example a 4 byte pattern, it's easy to spot as well. For example, here is the beginning of the kernel32.dll file (32-bit version from Windows Server 2008):

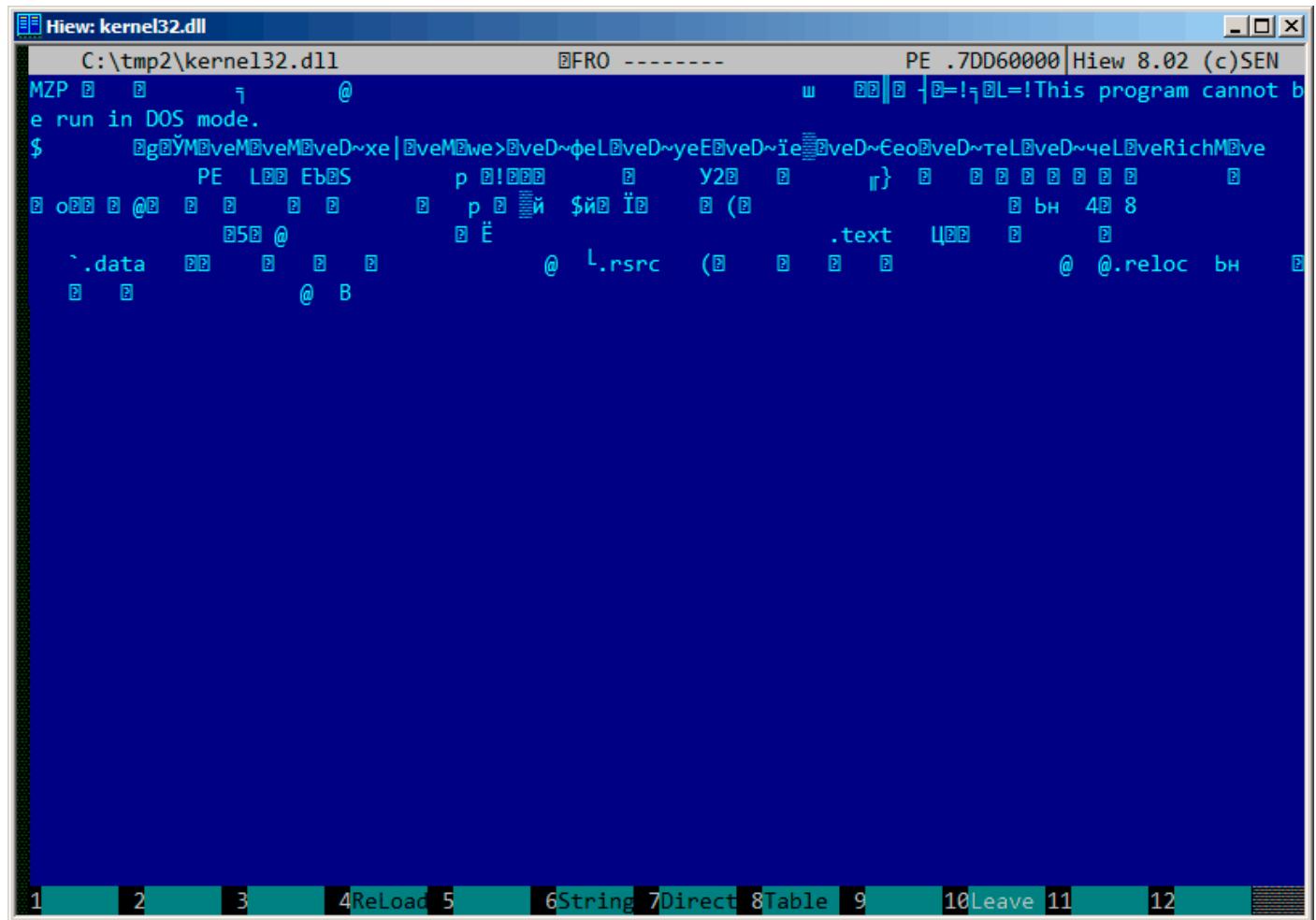


Figure 9.3: Original file

## 9.1. PRIMITIVE XOR-ENCRYPTION

Here it is “encrypted” with a 4-byte key:

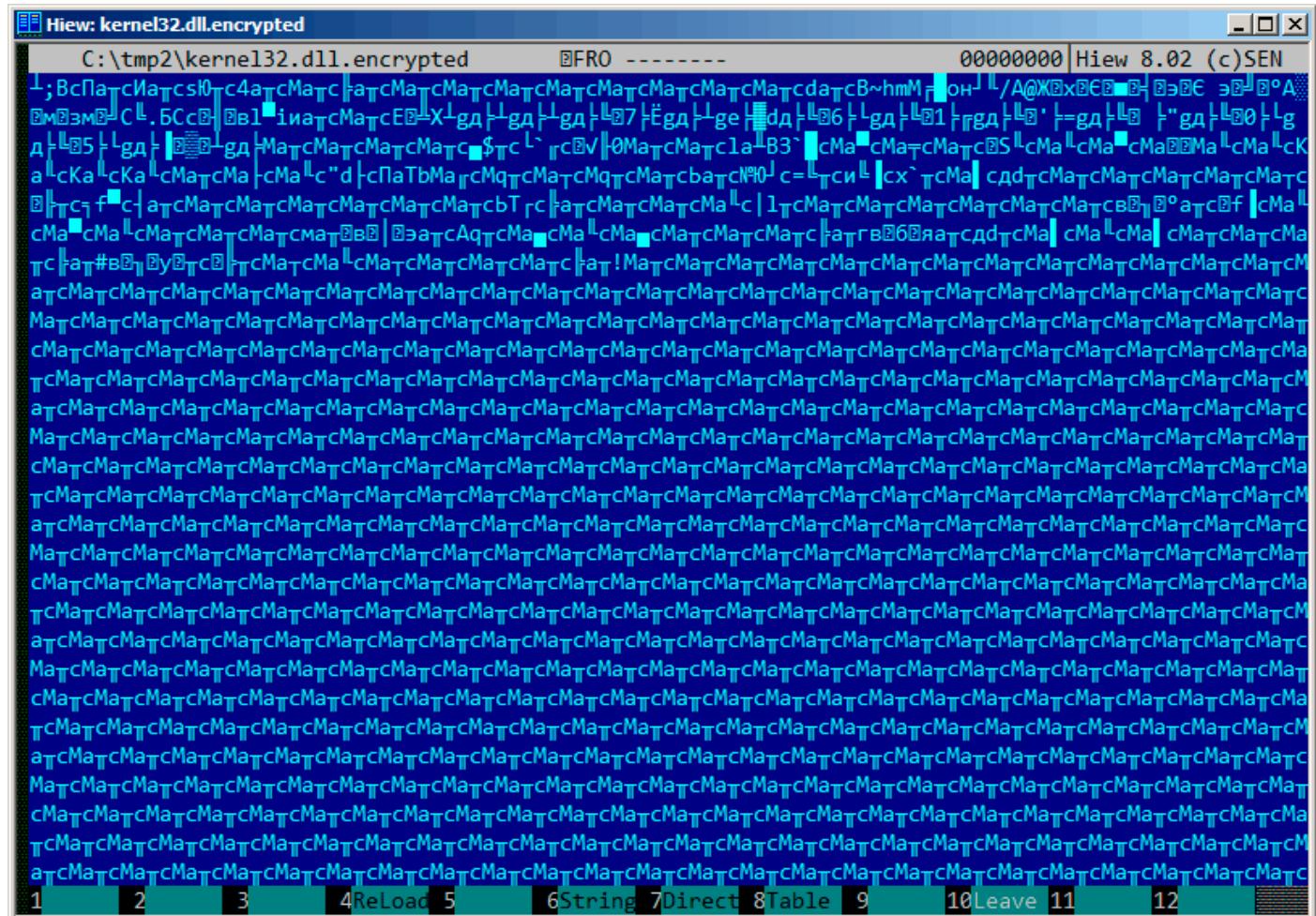


Figure 9.4: “Encrypted” file

It's very easy to spot the recurring 4 symbols.

Indeed, the header of a PE-file has a lot of long zero areas, which are the reason for the key to become visible.

## 9.1. PRIMITIVE XOR-ENCRYPTION

Here is the beginning of a PE-header in hexadecimal form:

```

Hiew: kernel32.dll
C:\tmp2\kernel32.dll          FRO -----  PE .7DD60290
.7DD600E0: 00 00 00 00-00 00 00 00-50 45 00 00-4C 01 04 00  PE L@@
.7DD600F0: 85 9A 15 53-00 00 00 00-00 00 00 00-E0 00 02 21  E@@S p @@
.7DD60100: 0B 01 09 00-00 00 0D 00-00 00 03 00-00 00 00 00  @@  @@
.7DD60110: 93 32 01 00-00 00 01 00-00 00 0D 00-00 00 D6 7D  Y2@  @  } 
.7DD60120: 00 00 01 00-00 00 01 00-06 00 01 00-06 00 01 00  @  @  @  @  @  @
.7DD60130: 06 00 01 00-00 00 00 00-00 00 11 00-00 00 01 00  @  @  @  @  @
.7DD60140: AE 05 11 00-03 00 40 01-00 00 04 00-00 10 00 00  @@ @@ @  @  @
.7DD60150: 00 00 10 00-00 10 00 00-00 00 00 00-10 00 00 00  @  @  @  @
.7DD60160: 70 FF 0B 00-B1 A9 00 00-24 A9 0C 00-F4 01 00 00  p @@ $@@ @@ 
.7DD60170: 00 00 0F 00-28 05 00 00-00 00 00 00-00 00 00 00  @ ( @
.7DD60180: 00 00 00 00-00 00 00 00-00 00 10 00-9C AD 00 00  @ @n
.7DD60190: 34 07 0D 00-38 00 00 00-00 00 00 00-00 00 00 00 4@ 8
.7DD601A0: 00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
.7DD601B0: 10 35 08 00-40 00 00 00-00 00 00 00-00 00 00 00  @@ @
.7DD601C0: 00 00 01 00-F0 0D 00 00-00 00 00 00-00 00 00 00  @ @
.7DD601D0: 00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
.7DD601E0: 2E 74 65 78-74 00 00 00-96 07 0C 00-00 00 01 00  .text  @@  @
.7DD601F0: 00 00 0D 00-00 00 01 00-00 00 00 00-00 00 00 00  @
.7DD60200: 00 00 00 00-20 00 00 60-2E 64 61 74-61 00 00 00  ` .data
.7DD60210: 0C 10 00 00-00 00 0E 00-00 00 01 00-00 00 0E 00  @@  @  @  @
.7DD60220: 00 00 00 00-00 00 00 00-00 00 00 00-40 00 00 C0  @ L
.7DD60230: 2E 72 73 72-63 00 00 00-28 05 00 00-00 00 0F 00  .rsrc  (@  @
.7DD60240: 00 00 01 00-00 00 0F 00-00 00 00 00-00 00 00 00  @  @
.7DD60250: 00 00 00 00-40 00 00 40-2E 72 65 6C-6F 63 00 00  @ @.reloc
.7DD60260: 9C AD 00 00-00 00 10 00-00 00 01 00-00 00 10 00  b@  @  @  @
.7DD60270: 00 00 00 00-00 00 00 00-00 00 00 00-40 00 00 42  @ B
.7DD60280: 00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00 00
.7DD60290: 00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00 00

```

Figure 9.5: PE-header

## 9.1. PRIMITIVE XOR-ENCRYPTION

Here it is “encrypted”:

Address	Value	Content
000000E0:	8C 61 D2 63-8C 61 D2 63-DC 24 D2 63-C0 60 D6 63	Ma[cMaTc\$cTcL`c
000000F0:	09 FB C7 30-8C 61 D2 63-8C 61 D2 63-6C 61 D0 42	0MaTcMatclalB
00000100:	87 60 DB 63-8C 61 DF 63-8C 61 D1 63-8C 61 D2 63	3`cMa[cMaTcMaTc
00000110:	1F 53 D3 63-8C 61 D3 63-8C 61 DF 63-8C 61 04 1E	BS[cMa[cMa[cMa@
00000120:	8C 61 D3 63-8C 61 D3 63-8A 61 D3 63-8A 61 D3 63	Ma[cMa[cKa[cKaLc
00000130:	8A 61 D3 63-8C 61 D2 63-8C 61 C3 63-8C 61 D3 63	KaLcMaTcMa[cMaLc
00000140:	22 64 C3 63-8F 61 92 62-8C 61 D6 63-8C 71 D2 63	"d[cPlaTbMa[cMqTc
00000150:	8C 61 C2 63-8C 71 D2 63-8C 61 D2 63-9C 61 D2 63	Ma[cMqTcMaTcbaTc
00000160:	FC 9E D9 63-3D C8 D2 63-A8 C8 DE 63-78 60 D2 63	N@0c=cTcИLcx`Tc
00000170:	8C 61 DD 63-A4 64 D2 63-8C 61 D2 63-8C 61 D2 63	Ma[cddTcMatcMatc
00000180:	8C 61 D2 63-8C 61 D2 63-8C 61 C2 63-10 CC D2 63	Ma[cMatcMatc@Tc
00000190:	B8 66 DF 63-B4 61 D2 63-8C 61 D2 63-8C 61 D2 63	7f[c]atcMatcMatc
000001A0:	8C 61 D2 63-8C 61 D2 63-8C 61 D2 63-8C 61 D2 63	Ma[cMatcMatcMatc
000001B0:	9C 54 DA 63-CC 61 D2 63-8C 61 D2 63-8C 61 D2 63	bT[c]atcMatcMatc
000001C0:	8C 61 D3 63-7C 6C D2 63-8C 61 D2 63-8C 61 D2 63	Ma[c lTcMatcMatc
000001D0:	8C 61 D2 63-8C 61 D2 63-8C 61 D2 63-8C 61 D2 63	Ma[cMatcMatcMatc
000001E0:	A2 15 B7 1B-F8 61 D2 63-1A 66 DE 63-8C 61 D3 63	в@0a[c@f[cMaLc
000001F0:	8C 61 DF 63-8C 61 D3 63-8C 61 D2 63-8C 61 D2 63	Ma[cMa[cMa[cMatc
00000200:	8C 61 D2 63-AC 61 D2 03-A2 05 B3 17-ED 61 D2 63	MaTcma@в@#в@Tc
00000210:	80 71 D2 63-8C 61 DC 63-8C 61 D3 63-8C 61 DC 63	AqTcMa[cMa[cMaLc
00000220:	8C 61 D2 63-8C 61 D2 63-8C 61 D2 63-CC 61 D2 A3	Ma[cMatcMatc#атг
00000230:	A2 13 A1 11-EF 61 D2 63-A4 64 D2 63-8C 61 DD 63	в@бяаTcddTcMaLc
00000240:	8C 61 D3 63-8C 61 DD 63-8C 61 D2 63-8C 61 D2 63	Ma[cMa[cMatcMatc
00000250:	8C 61 D2 63-CC 61 D2 23-A2 13 B7 0F-E3 02 D2 63	MaTc#ат#в@#в@Tc
00000260:	10 CC D2 63-8C 61 C2 63-8C 61 D3 63-8C 61 C2 63	@TcMatcMa[cMatc
00000270:	8C 61 D2 63-8C 61 D2 63-8C 61 D2 63-CC 61 D2 21	Ma[cMatcMatc#ат!
00000280:	8C 61 D2 63-8C 61 D2 63-8C 61 D2 63-8C 61 D2 63	Ma[cMatcMatcMatc
00000290:	8C 61 D2 63-8C 61 D2 63-8C 61 D2 63-8C 61 D2 63	Ma[cMatcMatcMatc

Figure 9.6: “Encrypted” PE-header

It's easy to spot that the key is the following 4 bytes: `8C 61 D2 63`.

With this information, it's easy to decrypt the whole file.

So it is important to keep in mind these properties of PE-files: 1) PE-header has many zero-filled areas; 2) all PE-sections are padded with zeros at a page boundary (4096 bytes), so long zero areas are usually present after each section.

Some other file formats may contain long zero areas.

It's typical for files used by scientific and engineering software.

For those who want to inspect these files on their own, they are downloadable here: <http://go.yurichev.com/17352>.

## Exercise

- <http://challenges.re/50>

### 9.1.3 Simple encryption using XOR mask

I've found an old interactive fiction game while diving deep into *if-archive*<sup>3</sup>:

```
The New Castle v3.5 - Text/Adventure Game
in the style of the original Infocom (tm)
type games, Zork, Colossal Cave (Adventure),
etc. Can you solve the mystery of the
abandoned castle?
Shareware from Software Customization.
Software Customization [ASP] Version 3.5 Feb. 2000
```

It's downloadable here: [http://yurichev.com/blog/XOR\\_mask\\_1/files/newcastle.tgz](http://yurichev.com/blog/XOR_mask_1/files/newcastle.tgz).

There is a file inside (named *castle.dbf*) which is clearly encrypted, but not by a real crypto algorithm, nor it's compressed, this is something rather simpler. I wouldn't even measure entropy level ([9.2 on page 945](#)) of the file. Here is how it looks like in Midnight Commander:

```
/home/dennis/P/RE-book/decrypt_dat_file/castle.dbf
Pg.tqfv.c...t}k.cmrf.yS. uqo...ze.n..lj..hw.m.j a
c.w.iubgv.^az..rn..}c~wf.z.uP.J1q13'\.\Qt>9P.(r$K8!L 78;QA-.<7]'Z.lj..hw.m.j a
c.w.iubgv.^az..rn..}c~wf.z.uPd.tqfv.c...t}k.cmrf.yS. uqo...ze.n..lj..hw.m.j a
c.w.iubgv.^az..rn..}c~wf.z.uP.J;q-8V4[.0<<?.&;*..5&MB&q&K.+T?e@+0@(9,[.wE(Tu a
c.w.iubgv.^az..rn..}c~wf.z.uP.J1q.>X'GD.??$N0!rf.yS. uqo...ze.n..lj..hw.m.j a
c.w.iubgv.^az..rn..}c~wf.z.uPd.tqfv.c...t}k.cmrf.yS. uqo...ze.n..lj..hw.m.j a
c.w.iubgv.^az..rn..}c~wf.z.uPd.tqfv.c...t}k.cmrf.yS. uqo...ze.n..lj..hw.m.j a
c.w.iubgv.^az..rn..}c~wf.z.uP..<003.7@V.<8*K7m=.8s\R=<+.ze.n..lj..hw.m.j a
c.w.iubgv.^az..rn..}c~wf.z.uP..?4#&.*.^.^:)*.$!35!y9[u!>.I&.> [%.lj..hw.m.j a
c.w.iubgv.^az..rn..}c~wf.z.uPd.tqfv.c...t}k.cmrf.yS. uqo...ze.n..lj..hw.m.j a
c.w.iubgv.^az..rn..}c~wf.z.uP.G"449Wg...t}k.cmrf.yS. uqo...ze.n..lj..hw.m.j a
c.w.iubgv.^az..rn..}c~wf.z.uP.J1##Vm.M.d<</V4m>/K*). uqo...ze.n..lj..hw.m.j a
c.w.iubgv.^az..rn..}c~wf.z.uPg.tqfv.c...t}k.cmrf.yS. uqo...ze.n..lj..hw.m.j a
c.w.iubgv.^az..rn..}c~wf.z.uP.N8q69J*\ZX:4%^c$f!f!..~)...,.
.E1Xz+G:.s...x..mc.j a
c.w.iubgv.^az..rn..}c~wf.z.uPg.tqfv.c...t}k.cmrf.yS. uqo...ze.n..lj..hw.m.j a
c.w.iubgv.^az..rn..}c~wf.z.uP.J=f?JcI.\8(/~m&).42TLu%"LW.0.0X'J,("Y2/w_"H!.a
c.w.iubgv.^az..rn..}c~wf.z.uPg.tqfv.c...t}k.cmrf.yS. uqo...ze.n..lj..hw.m.j a
c.w.iubgv.^az..rn..}c~wf.z.uP.G80>z. 2255$kP0m=(B.s..yqz..s.iq.nm'8)[..]K&IjH6L:.w.iubgv.^az..rn..}c~wf.z.uPg.tqf
c.w.iubgv.^az..rn..}c~wf.z.uP.G6$!1P-0.g&2,K"!fG*sY\;po
w0.36.<[\2#^Sh?M.^g0[.0[w!]!-g457?*zFN>"P.Itc~wf.z.uPd.tqfv.c...t}k.cmrf.yS. uqo...ze.n..lj..hw.m.j a
c.w.iubgv.^az..rn..}c~wf.z.uP.L0q $V..D^ 5" We9:#.-<RKu>).P7Yz0F*K80f.B):$H/Za0&H62i!*?"S~[$;B[r/P,])c~wf.z.uP" M&%3
v.^az..rn..}c~wf.z.uPg.tqfv.c...t}k.cmrf.yS. uqo...ze.n..lj..hw.m.j a
c.w.iubgv.^az..rn..}c~wf.z.uP.W05#8U:.R.'4%P0974.y$MH<%'IM4Yz#A)(@9jQD82I?Ijk$K,J2.0:7kvR;X,3_Hr:L,))c~wf.z.uP0N;02
\{.k.&%
"J?<Z,))c~wf.z.uPBC $43.,N.C<8kN,?>".12L 69.KC:XveI J,("U.,"F*_%GaN"R9[=u "vG1H/>..r.0I))c~wf.z.uP;W%f8V4.CV"
J!nJK)2c~wf.z.uP"J1q5&K&IW^:;k]"?9(K*,.u!.L60z5D/MWI+@D-6Z>,+ 5L0[2R<9.>vM;I5?CJ6nP;3c~wf.z.uP$G55/8^y...t}k.cmrf.
c.w.iubgv.^az..rn..}c~wf.z.uPg.tqfv.c...t}k.cmrf.yS. uqo...ze.n..lj..hw.m.j a
c.w.iubgv.^az..rn..}c~wf.z.uPg.tqfv.c...t}k.cm...g
.e...qm.rz.i.bq...hw.m.j a
```

Figure 9.7: Encrypted file in Midnight Commander

The encrypted file can be downloaded [here](#).

Will it be possible to decrypt it without accessing to the program, using just this file?

There is a clearly visible pattern of repeating string. If a simple encryption by XOR mask was applied, such repeating strings is a prominent signature, because, probably, there were a long lacunas of zero bytes, which, in turn, are present in many executable files as well as in binary data files.

Here I'll dump the file's beginning using *xxd* UNIX utility:

```
...
0000030: 09 61 0d 63 0f 77 14 69 75 62 67 76 01 7e 1d 61 .a.c.w.iubgv.^.
0000040: 7a 11 0f 72 6e 03 05 7d 7d 63 7e 77 66 1e 7a 02 z..rn..}c~wf.z.
0000050: 75 50 02 4a 31 71 31 33 5c 27 08 5c 51 74 3e 39 uP.J1q13'\.\Qt>9
0000060: 50 2e 28 72 24 4b 38 21 4c 09 37 38 3b 51 41 2d P.(r$K8!L.78;QA-
0000070: 1c 3c 37 5d 27 5a 1c 7c 6a 10 14 68 77 08 6d 1a .<7]'Z.|j..hw.m.

0000080: 6a 09 61 0d 63 0f 77 14 69 75 62 67 76 01 7e 1d j.a.c.w.iubgv.^.
0000090: 61 7a 11 0f 72 6e 03 05 7d 7d 63 7e 77 66 1e 7a az..rn..}c~wf.z
00000a0: 02 75 50 64 02 74 71 66 76 19 63 08 13 17 74 7d .uPd.tqfv.c...t}
```

<sup>3</sup><http://www.ifarchive.org/>

## 9.1. PRIMITIVE XOR-ENCRYPTION

```

00000b0: 6b 19 63 6d 72 66 0e 79 73 1f 09 75 71 6f 05 04 k.cmrf.y..uqo..
00000c0: 7f 1c 7a 65 08 6e 0e 12 7c 6a 10 14 68 77 08 6d ..ze.n..|j..hw.m

00000d0: 1a 6a 09 61 0d 63 0f 77 14 69 75 62 67 76 01 7e .j.a.c.w.iubgv.~
00000e0: 1d 61 7a 11 0f 72 6e 03 05 7d 7d 63 7e 77 66 1e .az..rn..}c~wf.
00000f0: 7a 02 75 50 01 4a 3b 71 2d 38 56 34 5b 13 40 3c z.uP.J;q-8V4[.@<
0000100: 3c 3f 19 26 3b 3b 2a 0e 35 26 4d 42 26 71 26 4b <?.&;*.*.5&MB&q&K
0000110: 04 2b 54 3f 65 40 2b 4f 40 28 39 10 5b 2e 77 45 .+T?e@+0@(9.[.wE

0000120: 28 54 75 09 61 0d 63 0f 77 14 69 75 62 67 76 01 (Tu.a.c.w.iubgv.
0000130: 7e 1d 61 7a 11 0f 72 6e 03 05 7d 7d 63 7e 77 66 ~.az..rn..}c~wf
0000140: 1e 7a 02 75 50 02 4a 31 71 15 3e 58 27 47 44 17 .z.uP.J1q.>X'GD.
0000150: 3f 33 24 4e 30 6c 72 66 0e 79 73 1f 09 75 71 6f ?3$N0lrf.y..uqo
0000160: 05 04 7f 1c 7a 65 08 6e 0e 12 7c 6a 10 14 68 77 ....ze.n..|j..hw

...

```

Let's stick at visible repeating "iubgv" string. By looking at this dump, we can clearly see that the period of the string occurrence is 0x51 or 81. Probably, 81 is size of block? Size of the file is 1658961, and it can be divided evenly by 81 (and there are 20481 blocks then).

Now I'll use Mathematica to analyze, are there repeating 81-byte blocks in the file? I'll split input file by 81-byte blocks and then I'll use *Tally[]*<sup>4</sup> function which just calculates, how many times some item has been occurred in the input list. Tally's output is not sorted, so I also add *Sort[]* function to sort it by number of occurrences in descending order.

```

input = BinaryReadList["/home/dennis/.../castle.dbf"];
blocks = Partition[input, 81];
stat = Sort[Tally[blocks], #1[[2]] > #2[[2]] &]

```

And here is output:

```

{{{80, 103, 2, 116, 113, 102, 118, 25, 99, 8, 19, 23, 116, 125, 107,
25, 99, 109, 114, 102, 14, 121, 115, 31, 9, 117, 113, 111, 5, 4,
127, 28, 122, 101, 8, 110, 14, 18, 124, 106, 16, 20, 104, 119, 8,
109, 26, 106, 9, 97, 13, 99, 15, 119, 20, 105, 117, 98, 103, 118,
1, 126, 29, 97, 122, 17, 15, 114, 110, 3, 5, 125, 125, 99, 126,
119, 102, 30, 122, 2, 117}, 1739},
{{{80, 100, 2, 116, 113, 102, 118, 25, 99, 8, 19, 23, 116,
125, 107, 25, 99, 109, 114, 102, 14, 121, 115, 31, 9, 117, 113,
111, 5, 4, 127, 28, 122, 101, 8, 110, 14, 18, 124, 106, 16, 20,
104, 119, 8, 109, 26, 106, 9, 97, 13, 99, 15, 119, 20, 105, 117,
98, 103, 118, 1, 126, 29, 97, 122, 17, 15, 114, 110, 3, 5, 125,
125, 99, 126, 119, 102, 30, 122, 2, 117}, 1422},
{{{80, 101, 2, 116, 113, 102, 118, 25, 99, 8, 19, 23, 116,
125, 107, 25, 99, 109, 114, 102, 14, 121, 115, 31, 9, 117, 113,
111, 5, 4, 127, 28, 122, 101, 8, 110, 14, 18, 124, 106, 16, 20,
104, 119, 8, 109, 26, 106, 9, 97, 13, 99, 15, 119, 20, 105, 117,
98, 103, 118, 1, 126, 29, 97, 122, 17, 15, 114, 110, 3, 5, 125,
125, 99, 126, 119, 102, 30, 122, 2, 117}, 1012},
{{{80, 120, 2, 116, 113, 102, 118, 25, 99, 8, 19, 23, 116,
125, 107, 25, 99, 109, 114, 102, 14, 121, 115, 31, 9, 117, 113,
111, 5, 4, 127, 28, 122, 101, 8, 110, 14, 18, 124, 106, 16, 20,
104, 119, 8, 109, 26, 106, 9, 97, 13, 99, 15, 119, 20, 105, 117,
98, 103, 118, 1, 126, 29, 97, 122, 17, 15, 114, 110, 3, 5, 125,
125, 99, 126, 119, 102, 30, 122, 2, 117}, 377},
...
{{{80, 2, 74, 49, 113, 21, 62, 88, 39, 71, 68, 23, 63, 51, 36, 78, 48,
108, 114, 102, 14, 121, 115, 31, 9, 117, 113, 111, 5, 4, 127, 28,
122, 101, 8, 110, 14, 18, 124, 106, 16, 20, 104, 119, 8, 109, 26,
106, 9, 97, 13, 99, 15, 119, 20, 105, 117, 98, 103, 118, 1, 126,
29, 97, 122, 17, 15, 114, 110, 3, 5, 125, 125, 99, 126, 119, 102,
30, 122, 2, 117}, 1},
{{{80, 1, 74, 59, 113, 45, 56, 86, 52, 91, 19, 64, 60, 60, 63,

```

<sup>4</sup><https://reference.wolfram.com/language/ref/Tally.html>

## 9.1. PRIMITIVE XOR-ENCRYPTION

```

25, 38, 59, 59, 42, 14, 53, 38, 77, 66, 38, 113, 38, 75, 4, 43, 84,
63, 101, 64, 43, 79, 64, 40, 57, 16, 91, 46, 119, 69, 40, 84, 117,
9, 97, 13, 99, 15, 119, 20, 105, 117, 98, 103, 118, 1, 126, 29,
97, 122, 17, 15, 114, 110, 3, 5, 125, 125, 99, 126, 119, 102, 30,
122, 2, 117}, 1},
{{80, 2, 74, 49, 113, 49, 51, 92, 39, 8, 92, 81, 116, 62, 57,
80, 46, 40, 114, 36, 75, 56, 33, 76, 9, 55, 56, 59, 81, 65, 45, 28,
60, 55, 93, 39, 90, 28, 124, 106, 16, 20, 104, 119, 8, 109, 26,
106, 9, 97, 13, 99, 15, 119, 20, 105, 117, 98, 103, 118, 1, 126,
29, 97, 122, 17, 15, 114, 110, 3, 5, 125, 125, 99, 126, 119, 102,
30, 122, 2, 117}, 1}}

```

Tally's output is list pairs, each pair has 81-byte block and number of times it has been occurred in the file. We see that the most frequent block is the first, it has been occurred 1739 times. The second one has been occurred 1422 times. There are others: 1012 times, 377 times, etc. 81-byte blocks which has been occurred just once are at the end of output.

Let's try to compare these blocks? The first and the second? Is there a function in Mathematica which compares lists/arrays? Certainly is, but for educational purposes, I'll use XOR operation for comparison. Indeed: if bytes in two input arrays are identical, XOR result is 0. If they are non-equal, result will be non-zero.

Let's compare first block (occurred 1739 times) and the second (occurred 1422 times):

They are differ only in the second byte.

Let's compare second (occurred 1422 times) and third (occurred 1012 times):

They are also differ only in the second byte.

Anyway, let's try to use the most occurred block as a XOR key and try to decrypt four first 81-byte blocks in the file:

## 9.1. PRIMITIVE XOR-ENCRYPTION

(I've replaced unprintable characters by "?".)

So we see that first and third blocks are empty (or almost empty), but second and fourth has clearly visible English language words/phrases. It seems that our assumption about key is correct (at least partially). This means that the most occurred 81-block in the file can be found at places of lacunas of zero bytes or something like that.

Let's try to decrypt the whole file:

```
DecryptBlock[blk_] := BitXor[key, blk]
decrypted = Map[DecryptBlock[#] &, blocks];
BinaryWrite["/home/dennis/.../tmp", Flatten[decrypted]]
Close["/home/dennis/.../tmp"]
```

### 9.1. PRIMITIVE XOR-ENCRYPTION

RE-book/decrypt\_dat\_file/tmp

4011/1620K

eHE.WEED.OF

0%

```

TTER.FRUIT..... fHO.KNOWS.WHAT.EVIL.LURKS.IN.THE.HE
..... eHE.sHADOW.KNOWS.

..... x.HAVE.THE.HEART.OF.A.CHILD.
P.IT.IN.A.GLASS.JAR.ON.MY.DESK..... uEVERON.
..... fHERE.THE.sHADOW.LIES.

..... pLL.POSITIONING.IS.relative.AND.NOT.absolute.

..... eHIS.IS.A.KLUDGE.TO.MAKE.THIS.STUPID.THING.WORK.
..... cELAX

Y..... cLOCK.tICKS.AWAY..... uEBUGGING.pROGRAMS.IS.FUN...s
RD
K.WALLS.

..... pND.FROM.WITHIN.THE.TOMB.OF.THE.UNDEAD..VAMPIRES.BEGAN.THEIR.FEA
..... ECTORIED.CRIES.RANG.OUT..... tASTES.GREAT..IESS.FILLING.

..... bUDDENL
RAITHLIKE.FIGURE.APPEARS.BEFORE.YOU..SEEMING.TO..... wLOAT.IN.THE.AIR...
WFUL.VOICE.HE.SAYS...aLAS..THE.VERY....._ATURE.OF.THE.WORLD.HAS.CHANGED
DN.CANNOT.BE.FOUND...aLL.....\UST.NOW.PASS.AWAY....rAISING.HIS.DAKEN.STA
HE.FADES.INTO.....eHE.SPREADING.DARKNESS...iN.HIS.PLACE.APPEARS.A.TASTEFU
GN.....CEADING...

```

Figure 9.8: Decrypted file in Midnight Commander, 1st attempt

Looks like some kind of English phrases for some game, but something wrong. First of all, cases are inverted: phrases and some words are started with lowercase characters, while other characters are in upper case. Also, some phrases started with wrong letters. Take a look at the very first phrase: "eHE WEED OF CRIME BEARS BITTER FRUIT". What is "eHE"? Isn't "tHE" have to be here? Is it possible that our decryption key has wrong byte at this place?

Let's look again at the second block in the file, at key and at decryption result:

```

In[]:= blocks[[2]]
Out[]={80, 2, 74, 49, 113, 49, 51, 92, 39, 8, 92, 81, 116, 62, \
57, 80, 46, 40, 114, 36, 75, 56, 33, 76, 9, 55, 56, 59, 81, 65, 45, \
28, 60, 55, 93, 39, 90, 28, 124, 106, 16, 20, 104, 119, 8, 109, 26, \
106, 9, 97, 13, 99, 15, 119, 20, 105, 117, 98, 103, 118, 1, 126, 29, \
97, 122, 17, 15, 114, 110, 3, 5, 125, 125, 99, 126, 119, 102, 30, \
122, 2, 117}

In[]:= key
Out[]={80, 103, 2, 116, 113, 102, 118, 25, 99, 8, 19, 23, 116, \
125, 107, 25, 99, 109, 114, 102, 14, 121, 115, 31, 9, 117, 113, 111, \
5, 4, 127, 28, 122, 101, 8, 110, 14, 18, 124, 106, 16, 20, 104, 119, \
8, 109, 26, 106, 9, 97, 13, 99, 15, 119, 20, 105, 117, 98, 103, 118, \
1, 126, 29, 97, 122, 17, 15, 114, 110, 3, 5, 125, 125, 99, 126, 119, \
102, 30, 122, 2, 117}

In[]:= BitXor[key, blocks[[2]]]
Out[]={0, 101, 72, 69, 0, 87, 69, 69, 68, 0, 79, 70, 0, 67, 82, \
73, 77, 69, 0, 66, 69, 65, 82, 83, 0, 66, 73, 84, 84, 69, 82, 0, 70, \
82, 85, 73, 84, 14, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \
0, 0, 0, 0}
```

## 9.1. PRIMITIVE XOR-ENCRYPTION

Encrypted byte is 2, byte from key is  $103$ ,  $2 \oplus 103 = 101$  and  $101$  is ASCII code for "e" character. What byte a key must be equal to, so the resulting ASCII code will be  $116$  (for "t" character)?  $2 \oplus 116 = 118$ , let's put  $118$  in key at the second byte...

```
key = {80, 118, 2, 116, 113, 102, 118, 25, 99, 8, 19, 23, 116, 125,
 107, 25, 99, 109, 114, 102, 14, 121, 115, 31, 9, 117, 113, 111, 5,
 4, 127, 28, 122, 101, 8, 110, 14, 18, 124, 106, 16, 20, 104, 119, 8,
 109, 26, 106, 9, 97, 13, 99, 15, 119, 20, 105, 117, 98, 103, 118,
 1, 126, 29, 97, 122, 17, 15, 114, 110, 3, 5, 125, 125, 99, 126, 119,
 102, 30, 122, 2, 117}
```

... and decrypt the whole file again.

```
/home/dennis/P/RE-book/decrypt_dat_file/tmp 4011/1620K 0%
.....THE.WEED.OF
.CRIME.BEARS.BITTER.FRUIT.....WHO.KNOWS.WHAT.EVIL.LURKS.IN.THE.HE
ARTS.OF.MEN.....tHE.sHADOW.KNOWS.....
.....i.HAVE.THE.HEART.OF.A.CHILD.....
.....i.KEEP.IT.IN.A.GLASS.JAR.ON.MY.DESK.....
.....dEVERON.....
.....wHERE.THE.sHADOW.LIES.....
.....aLL.POSITIONING.IS.relative.AND.NOT.absolute.....
.....tHIS.IS.A.KLUDGE.TO.MAKE.THIS.STUPID.THING.WORK.....
.....rELAX.....
.....fRIDAY.IS.ONLY.....cLOCK.tICKS.AWAY.....
.....dEBUGGING.pROGRAMS.IS.FUN...s
D.IS.RUNNING.HEAD
FIRST.INTO.BRICK.WALLS...
.....aND.FROM.WITHIN.THE.TOMB.OF.THE.UNDEAD..VAMPIRES.BEGAN.THEIR.FEA
ST.AS.....
.....TORTURED.CRIES.RANG.OUT.....
.....tASTES.GREAT..IESS.FILLING.....
.....sUDDENL
Y.A.SINISTER..WRAITHLIKE.FIGURE.APPEARS.BEFORE.YOU..SEEMING.TO.....FLOAT.IN.THE.AIR...
IN.A.LOW..SORROWFUL.VOICE.HE.SAYS...aLAS..THE.VERY.....NATURE.OF.THE.WORLD.HAS.CHANGED
..AND.THE.DUNGEON.CANNOT.BE.FOUND...aL.....MUST.NOW.PASS.AWAY...rAISING.HIS.OAKEN.STA
FF.IN.FAIRWELL..HE.FADES.INTO.....THE.SPREADING.DARKNESS...iN.HIS.PLACE.APPEARS.A.TASTEFU
LLY.LETTERED.SIGN.....
.....READING.....
```

Figure 9.9: Decrypted file in Midnight Commander, 2nd attempt

Wow, now the grammar is correct, all phrases started with correct letters. But still, case inversion is suspicious. Why would game's developer write them in such a manner? Maybe our key is still incorrect?

While observing ASCII table in Wikipedia article<sup>5</sup> we can notice that uppercase and lowercase letter's ASCII codes are differ in just one bit (6th bit starting at 1st,  $0b100000$ ). This bit in decimal form is  $32\dots 32$ ? But  $32$  is ASCII code for space!

Indeed, one can switch case just by XOR-ing ASCII character code with  $32$ .

It is possible that the empty lacunas in the file are not zero bytes, but rather spaces? Let's modify XOR key one more time (I'll XOR each byte of key by  $32$ ):

```
(* "32" is scalar and "key" is vector, but that's OK *)
In[]:= key3 = BitXor[32, key]
Out[]={112, 86, 34, 84, 81, 70, 86, 57, 67, 40, 51, 55, 84, 93, 75, \
57, 67, 77, 82, 70, 46, 89, 83, 63, 41, 85, 81, 79, 37, 36, 95, 60, \
90, 69, 40, 78, 46, 50, 92, 74, 48, 52, 72, 87, 40, 77, 58, 74, 41, \
65, 45, 67, 47, 87, 52, 73, 85, 66, 71, 86, 33, 94, 61, 65, 90, 49, \
47, 82, 78, 35, 37, 93, 93, 67, 94, 87, 70, 62, 90, 34, 85}
In[]:= DecryptBlock[blk_] := BitXor[key3, blk]
```

<sup>5</sup><https://en.wikipedia.org/wiki/ASCII>

## 9.1. PRIMITIVE XOR-ENCRYPTION

Let's decrypt the input file again:

```
/home/dennis/P/RE-book/decrypt_dat_file/tmp3
1
2
in the hearts of men?                                The Shadow knows!
2
I keep it in a glass jar on my desk.                  I H
Deveron:
Where the Shadow lies.                               1
All positioning is RELATIVE and not ABSOLUTE.
This is a kludge to make this
1
1
(So is running head-first into brick walls!!)      2
And from within the tomb of the undead, vampires began their feast as      tom
g!"                                              10
hlike figure appears before you, seeming to          float in the air. In a low, s
nature of the world has changed, and the dungeon cannot be found. All      mus
well, he fades into          the spreading darkness. In his place appears a tasteful
INITIALIZATION FAILURE
The darkness becomes all encompassing, and your vision fo
Lick My User Port!!!
1
CRATCH Paper.           1
hem you were playing GAMES all day...                1
Keep it up and we'll both go out for a beer.        1
No, odd addresses don't occur on the South side of the st
Did you really expect me to re
1
1
```

Figure 9.10: Decrypted file in Midnight Commander, final attempt

(Decrypted file is available [here](#).) This is undoubtedly a correct source file. Oh, and we see numbers at the start of each block. It has to be a source of our erroneous XOR key. As it seems, the most occurred 81-byte block in the file is a block filled with spaces and containing "1" character at the place of second byte. Indeed, somehow, many blocks here are interleaved with this one. Maybe it's some kind of padding for short phrases/messages? Other highly occurred 81-byte blocks are also space-filled blocks, but with different digits, hence, they are differ only at the second byte.

That's all! Now we can write utility to encrypt the file back, and maybe modify it before.

Mathematica notebook file is downloadable [here](#).

Summary: XOR encryption like that is not robust at all. It has been intended by game's developer(s), probably, just to prevent gamer(s) to peek into internals of game, nothing else. Still, encryption like that is extremely popular due to its simplicity and many reverse engineers are usually familiar with it.

### 9.1.4 Simple encryption using XOR mask, case II

I've got another encrypted file, which is clearly encrypted by something simple, like XOR-ing:

## 9.1. PRIMITIVE XOR-ENCRYPTION

0x00000000																		
00000000	DD	D2	0F	70	1C	E7	9E	8D	E9	EC	AC	3D	61	5A	15	95	·P·聘	=aZ.
00000010	5C	F5	D3	0D	70	38	E7	94	DF	F2	E2	BC	76	34	61	0F	\ .p8	v4a.
00000020	98	5D	FC	D9	01	26	2A	FD	82	DF	E9	E2	BB	33	61	7B	J .&*	3aC
00000030	14	D9	45	F8	C5	01	3D	20	FD	95	96	EB	E4	BC	7A	61	. E . =	za
00000040	61	1B	8F	54	9D	AA	54	20	20	E1	DB	8B	ED	EC	BC	33	a. T T . ^	3
00000050	61	7C	15	8D	11	F9	CE	47	22	2A	FE	8E	9A	EB	F7	EF	al. . G"*	
00000060	39	22	71	1B	8A	58	FF	CE	52	70	38	E7	9E	91	A5	EB	9"q. X Rp8 聽	
00000070	AA	76	36	73	09	D9	44	E0	80	40	3C	23	AF	95	96	E2	v6s. D @<#	
00000080	EB	BB	7A	61	65	1B	8A	11	E3	C5	40	24	2A	EB	F6	F5	zae. . @\$*	
00000090	E4	F7	EF	22	29	77	5A	9B	43	F5	C1	4A	36	2E	FC	8F	" )wZ C J6,	
000000A0	DF	F1	E2	AD	3A	24	3C	5A	B0	11	E3	D4	4E	3F	2B	AF	: \$<Z . N?+	
000000B0	8E	8F	EA	ED	EF	22	29	77	5A	91	54	F1	D2	55	38	62	" )wZ T U8b	
000000C0	FD	BE	98	A5	E2	A1	32	61	62	13	9A	5A	F5	C4	01	25	2ab. Z . %	
000000D0	3F	AF	8F	97	E0	8E	C5	25	35	7B	19	92	11	E7	C8	48	? %5( . . H	
000000E0	33	27	AF	94	8A	F7	A3	B9	3F	32	7B	0E	96	43	B0	C8	3' ?2( . C	
000000F0	40	34	6F	E3	9E	99	F1	A3	AD	33	29	7B	14	9D	11	F8	@4o 聽 3)C. .	
00000100	C9	4C	70	3B	E7	9E	DF	EB	EA	A8	3E	35	32	18	9C	57	Lp; >52. W	
00000110	FF	D2	44	7E	6F	C6	8F	DF	F2	E2	BC	76	20	1F	70	9F	D~o@ v . p	
00000120	58	FE	C5	0D	70	3B	E7	92	9C	EE	A3	BF	3F	24	71	1F	X . p; 瑞 □ ?\$q.	
00000130	D9	5E	F6	80	56	3F	20	EB	D7	DF	E7	F6	R3	34	2E	67	^ V? 4.g	
00000140	09	D4	59	F5	C1	45	35	2B	A3	DB	90	E3	A3	BB	3E	24	. Y E5+ <del>xs</del> >\$	
00000150	32	09	96	43	E4	80	56	3B	26	EC	93	DF	EC	F0	EF	3D	2. C V8& =	
00000160	2F	7D	0D	97	11	F1	D3	2C	5A	2E	AF	D9	AF	E0	ED	AE	/J. . , Z. .	
00000170	38	26	32	16	98	46	E9	C5	53	7E	6D	AF	B1	8A	F6	F7	8&2. F S~m	
00000180	EF	23	2F	76	1F	8B	11	E4	C8	44	70	27	EA	9A	9B	A5	#/v. . Dp'.	
00000190	F4	AE	25	61	73	5A	9B	43	FF	C1	45	70	3C	E6	97	89	%asZ C Ep< 聽	
000001A0	E0	F1	EF	34	20	7C	1E	D9	5F	F5	C1	53	3C	36	82	F1	4 I. _ S<6	
000001B0	9E	EB	A3	A6	38	22	7A	5A	98	52	E2	CF	52	23	61	AF	尋 8"z2 R R#a	
000001C0	D9	AB	EA	A3	85	37	2C	77	09	D9	7C	FF	D2	55	39	22	, . 7.w. I U9"	
000001D0	EA	89	D3	A5	CE	E1	04	6F	51	54	AA	1F	BC	80	47	22	Ü . oQT . G"	
000001E0	20	E2	DB	97	EC	F0	EF	30	33	7B	1F	97	55	E3	80	4E	03( . U N	
000001F0	36	6F	FB	93	9A	88	89	8C	78	02	3C	32	D7	1D	B2	80	6o x.<2 .	

Figure 9.11: Encrypted file in Midnight Commander

The encrypted file can be downloaded [here](#).

`ent` Linux utility reports about 7.5 bits per byte, and this is high level of entropy ([9.2 on page 945](#)), close to compressed or properly encrypted file. But still, we clearly see some pattern, there are some blocks with size of 17 bytes, hence, you see some kind of ladder, shifting by 1 byte at each 16-byte line.

It's also known that the plain text is just English language text.

Now let's assume that this piece of text is encrypted by simple XOR-ing with 17-byte key.

I tried to find some repeating 17-byte blocks in Mathematica, like I did before in my previous example ([9.1.3 on page 933](#)):

Listing 9.2: Mathematica

```
In]:=input = BinaryReadList["/home/dennis/tmp/cipher.txt"];
In]:=blocks = Partition[input, 17];
In]:=Sort[Tally[blocks], #1[[2]] > #2[[2]] &]
Out}:={{248,128,88,63,58,175,159,154,232,226,161,50,97,127,3,217,80},1},
{{226,207,67,60,42,226,219,150,246,163,166,56,97,101,18,144,82},1},
{{228,128,79,49,59,250,137,154,165,236,169,118,53,122,31,217,65},1},
{{252,217,1,39,39,238,143,223,241,235,170,91,75,119,2,152,82},1},
{{244,204,88,112,59,234,151,147,165,238,170,118,49,126,27,144,95},1},
{{241,196,78,112,54,224,142,223,242,236,186,58,37,50,17,144,95},1},
{{176,201,71,112,56,230,143,151,234,246,187,118,44,125,8,156,17},1},
```

## 9.1. PRIMITIVE XOR-ENCRYPTION

```
...
{{255, 206, 82, 112, 56, 231, 158, 145, 165, 235, 170, 118, 54, 115, 9, 217, 68}, 1},
{{249, 206, 71, 34, 42, 254, 142, 154, 235, 247, 239, 57, 34, 113, 27, 138, 88}, 1},
{{157, 170, 84, 32, 32, 225, 219, 139, 237, 236, 188, 51, 97, 124, 21, 141, 17}, 1},
{{248, 197, 1, 61, 32, 253, 149, 150, 235, 228, 188, 122, 97, 97, 27, 143, 84}, 1},
{{252, 217, 1, 38, 42, 253, 130, 223, 233, 226, 187, 51, 97, 123, 20, 217, 69}, 1},
{{245, 211, 13, 112, 56, 231, 148, 223, 242, 226, 188, 118, 52, 97, 15, 152, 93}, 1},
{{221, 210, 15, 112, 28, 231, 158, 141, 233, 236, 172, 61, 97, 90, 21, 149, 92}, 1}}
```

No luck, each 17-byte block is unique within the file and occurred only once. Perhaps, there are no 17-byte zero (or space) lacunas. It is possible indeed: such long space indentation and padding may be absent in tightly typeset text.

The first idea is to try all possible 17-byte keys and find those, which will result in printable plain text after decryption. Bruteforce is not an option, because there are  $256^{17}$  possible keys ( $10^{40}$ ), that's too much. But there are good news: who said we have to test 17-byte key as a whole, why can't we test each byte of key separately? It is possible indeed.

Now the algorithm is:

- try all 256 bytes for 1st byte of key;
- decrypt 1st byte of each 17-byte blocks in the file;
- are all decrypted bytes we got are printable? keep tabs on it;
- do so for all 17 bytes of key.

I wrote with the following Python script to check this idea:

Listing 9.3: Python script

```
each_Nth_byte=[""]*KEY_LEN

content=read_file(sys.argv[1])
# split input by 17-byte chunks:
all_chunks=chunks(content, KEY_LEN)
for c in all_chunks:
    for i in range(KEY_LEN):
        each_Nth_byte[i]=each_Nth_byte[i] + c[i]

# try each byte of key
for N in range(KEY_LEN):
    print "N=", N
    possible_keys=[]
    for i in range(256):
        tmp_key=chr(i)*len(each_Nth_byte[N])
        tmp=xor_strings(tmp_key,each_Nth_byte[N])
        # are all characters in tmp[] are printable?
        if is_string_printable(tmp)==False:
            continue
        possible_keys.append(i)
    print possible_keys, "len=", len(possible_keys)
```

(Full version of the source code is [here](#).)

Here is its output:

```
N= 0
[144, 145, 151] len= 3
N= 1
[160, 161] len= 2
N= 2
[32, 33, 38] len= 3
N= 3
[80, 81, 87] len= 3
N= 4
[78, 79] len= 2
N= 5
[142, 143] len= 2
N= 6
[250, 251] len= 2
N= 7
```

## 9.1. PRIMITIVE XOR-ENCRYPTION

```
[254, 255] len= 2
N= 8
[130, 132, 133] len= 3
N= 9
[130, 131] len= 2
N= 10
[206, 207] len= 2
N= 11
[81, 86, 87] len= 3
N= 12
[64, 65] len= 2
N= 13
[18, 19] len= 2
N= 14
[122, 123] len= 2
N= 15
[248, 249] len= 2
N= 16
[48, 49] len= 2
```

So there are 2 or 3 possible bytes for each byte of 17-byte key. This is much better than 256 possible bytes for each byte, but still too much. There are up to 1 million of possible keys:

Listing 9.4: Mathematica

```
In[]:= 3*2*3*3*2*2*2*3*2*2*3*2*2*2*2*2*2
Out[]= 995328
```

It's possible to check all of them, but then we must check visually, if the decrypted text is looks like English language text.

Let's also take into consideration the fact that we deal with 1) natural language; 2) English language. Natural languages has some prominent statistical features. First of all, punctuation and word lengths. What is average word length in English language? Let's just count spaces in some well-known English language texts using Mathematica.

Here is "[The Complete Works of William Shakespeare](#)" text file from Gutenberg Library:

Listing 9.5: Mathematica

```
In[]:= input = BinaryReadList["/home/dennis/tmp/pg100.txt"];
In[]:= Tally[input]
Out[]={{239, 1}, {187, 1}, {191, 1}, {84, 39878}, {104,
218875}, {101, 406157}, {32, 1285884}, {80, 12038}, {114,
209907}, {111, 282560}, {106, 2788}, {99, 67194}, {116,
291243}, {71, 11261}, {117, 115225}, {110, 216805}, {98,
46768}, {103, 57328}, {69, 42703}, {66, 15450}, {107, 29345}, {102,
69103}, {67, 21526}, {109, 95890}, {112, 46849}, {108, 146532}, {87,
16508}, {115, 215605}, {105, 199130}, {97, 245509}, {83,
34082}, {44, 83315}, {121, 85549}, {13, 124787}, {10, 124787}, {119,
73155}, {100, 134216}, {118, 34077}, {46, 78216}, {89, 9128}, {45,
8150}, {76, 23919}, {42, 73}, {79, 33268}, {82, 29040}, {73,
55893}, {72, 18486}, {68, 15726}, {58, 1843}, {65, 44560}, {49,
982}, {50, 373}, {48, 325}, {91, 2076}, {35, 3}, {93, 2068}, {74,
2071}, {57, 966}, {52, 107}, {70, 11770}, {85, 14169}, {78,
27393}, {75, 6206}, {77, 15887}, {120, 4681}, {33, 8840}, {60,
468}, {86, 3587}, {51, 343}, {88, 608}, {40, 643}, {41, 644}, {62,
440}, {39, 31077}, {34, 488}, {59, 17199}, {126, 1}, {95, 71}, {113,
2414}, {81, 1179}, {63, 10476}, {47, 48}, {55, 45}, {54, 73}, {64,
3}, {53, 94}, {56, 47}, {122, 1098}, {90, 532}, {124, 33}, {38,
21}, {96, 1}, {125, 2}, {37, 1}, {36, 2}}
```

```
In[]:= Length[input]/1285884 // N
Out[]= 4.34712
```

There are 1285884 spaces in the whole file, and the frequency of space occurrence is 1 space per 4.3 characters.

Now here is [Alice's Adventures in Wonderland, by Lewis Carroll](#) from the same library:

## 9.1. PRIMITIVE XOR-ENCRYPTION

Listing 9.6: Mathematica

```
In[]:= input = BinaryReadList["/home/dennis/tmp/pg11.txt"];
In[]:= Tally[input]
Out[]={{239, 1}, {187, 1}, {191, 1}, {80, 172}, {114, 6398}, {111,
9243}, {106, 222}, {101, 15082}, {99, 2815}, {116, 11629}, {32,
27964}, {71, 193}, {117, 3867}, {110, 7869}, {98, 1621}, {103,
2750}, {39, 2885}, {115, 6980}, {65, 721}, {108, 5053}, {105,
7802}, {100, 5227}, {118, 911}, {87, 256}, {97, 9081}, {44,
2566}, {121, 2442}, {76, 158}, {119, 2696}, {67, 185}, {13,
3735}, {10, 3735}, {84, 571}, {104, 7580}, {66, 125}, {107,
1202}, {102, 2248}, {109, 2245}, {46, 1206}, {89, 142}, {112,
1796}, {45, 744}, {58, 255}, {68, 242}, {74, 13}, {50, 12}, {53,
13}, {48, 22}, {56, 10}, {91, 4}, {69, 313}, {35, 1}, {49, 68}, {93,
4}, {82, 212}, {77, 222}, {57, 11}, {52, 10}, {42, 88}, {83,
288}, {79, 234}, {70, 134}, {72, 309}, {73, 831}, {85, 111}, {78,
182}, {75, 88}, {86, 52}, {51, 13}, {63, 202}, {40, 76}, {41,
76}, {59, 194}, {33, 451}, {113, 135}, {120, 170}, {90, 1}, {122,
79}, {34, 135}, {95, 4}, {81, 85}, {88, 6}, {47, 24}, {55, 6}, {54,
7}, {37, 1}, {64, 2}, {36, 2}}
```

```
In[]:= Length[input]/27964 // N
Out[]= 5.99049
```

The result is different probably because of different formatting of these texts (maybe indentation and/or padding).

OK, so let's assume the average frequency of space in English language is 1 space per 4..7 characters.

Now the good news again: we can measure frequency of spaces while decrypting our file gradually. Now I count spaces in each *slice* and throw away 1-byte keys which produce buffers with too small number of spaces (or too large, but this is almost impossible given so short key):

Listing 9.7: Python script

```
each_Nth_byte=[""]*KEY_LEN

content=read_file(sys.argv[1])
# split input by 17-byte chunks:
all_chunks=chunks(content, KEY_LEN)
for c in all_chunks:
    for i in range(KEY_LEN):
        each_Nth_byte[i]=each_Nth_byte[i] + c[i]

# try each byte of key
for N in range(KEY_LEN):
    print "N=", N
    possible_keys=[]
    for i in range(256):
        tmp_key=chr(i)*len(each_Nth_byte[N])
        tmp=xor_strings(tmp_key,each_Nth_byte[N])
        # are all characters in tmp[] are printable?
        if is_string_printable(tmp)==False:
            continue
        # count spaces in decrypted buffer:
        spaces=tmp.count(' ')
        if spaces==0:
            continue
        spaces_ratio=len(tmp)/spaces
        if spaces_ratio<4:
            continue
        if spaces_ratio>7:
            continue
        possible_keys.append(i)
print possible_keys, "len=", len(possible_keys)
```

(Full version of the source code is [here](#).)

This reports just one single possible byte for each byte of key:

N= 0

## 9.1. PRIMITIVE XOR-ENCRYPTION

```
[144] len= 1
N= 1
[160] len= 1
N= 2
[33] len= 1
N= 3
[80] len= 1
N= 4
[79] len= 1
N= 5
[143] len= 1
N= 6
[251] len= 1
N= 7
[255] len= 1
N= 8
[133] len= 1
N= 9
[131] len= 1
N= 10
[207] len= 1
N= 11
[86] len= 1
N= 12
[65] len= 1
N= 13
[18] len= 1
N= 14
[122] len= 1
N= 15
[249] len= 1
N= 16
[49] len= 1
```

Let's check this key in Mathematica:

Listing 9.8: Mathematica

```
In[]:= input = BinaryReadList["/home/dennis/tmp/cipher.txt"];
In[]:= blocks = Partition[input, 17];
In[]:= key = {144, 160, 33, 80, 79, 143, 251, 255, 133, 131, 207, 86, 65, 18, 122, 249, 49};
In[]:= EncryptBlock[blk_] := BitXor[key, blk]
In[]:= encrypted = Map[EncryptBlock[#] &, blocks];
In[]:= BinaryWrite["/home/dennis/tmp/plain2.txt", Flatten[encrypted]]
In[]:= Close["/home/dennis/tmp/plain2.txt"]
```

And the plain text is:

Mr. Sherlock Holmes, who was usually very late in the mornings, save upon those not infrequent occasions when he was up all night, was seated at the breakfast table. I stood upon the hearth-rug and picked up the stick which our visitor had left behind him the night before. It was a fine, thick piece of wood, bulbous-headed, of the sort which is known as a "Penang lawyer." Just under the head was a broad silver band nearly an inch across. "To James Mortimer, M.R.C.S., from his friends of the C.C.H.," was engraved upon it, with the date "1884." It was just such a stick as the old-fashioned family practitioner used to carry--dignified, solid, and reassuring.

"Well, Watson, what do you make of it?"

Holmes was sitting with his back to me, and I had given him no sign of my occupation.

...

(Full version of the text is [here](#).)

The text looks correct. Yes, I made up this example and choose well-known text of Conan Doyle, but it's very close to what I had in my practice some time ago.

### Other ideas to consider

If we would fail with space counting, there are other ideas to try:

- Take into consideration the fact that lowercase letters are much more frequent than uppercase ones.
- Frequency analysis.
- There is also a good technique to detect language of a text: trigrams. Each language has some very frequent letter triplets, these may be "the" and "tha" for English. Read more about it: [N-Gram-Based Text Categorization](#), <http://code.activestate.com/recipes/326576/>. Interestingly enough, trigrams detection can be used when you decrypt a ciphertext gradually, like in this example (you just have to test 3 adjacent decrypted characters).

For non-Latin writing systems encoded in UTF-8, things may be easier. For example, Russian text encoded in UTF-8 has each byte interleaved with 0xD0/0xD1 byte. It is because Cyrillic characters are placed in 4th block of Unicode. Other writing systems has their own blocks.

## 9.2 Analyzing unknown binary files using information entropy

For the sake of simplification, I would say, information entropy is a measure, how tightly some piece of data can be compressed. For example, it is usually not possible to compress already compressed archive file, so it has high entropy. On the other hand, one megabyte of zero bytes can be compressed to a tiny output file. Indeed, in plain English language, one million of zeros can be described just as "resulting file is one million zero bytes". Compressed files are usually a list of instructions to decompressor, like this: "put 1000 zeros, then 0x23 byte, then 0x45 byte, then put a block of size 10 bytes which we've seen 500 bytes back, etc."

Texts written in natural languages are also can be compressed tightly, because natural languages has a lot of redundancy (otherwise, a tiny typo will always lead to misunderstanding, like any toggled bit in compressed archive make decompression nearly impossible), some words are used very often, etc. It's possible to drop some words and text will be still readable.

Code for CPUs is also can be compressed, because some ISA instructions are used much more often than others. In x86, most used instructions are MOV/PUSH/CALL—indeed, most of the time, computer CPU is just shuffling data and switching between levels of abstractions. If to consider data shuffling as moving data between levels of abstractions, this is also part of switching.

Data compressors and encryptors tend to produce very high entropy results. Good pseudorandom number generators also produce data which cannot be compressed (it is possible to measure their quality by this sign).

So, in other words, entropy is a measure which can help to probe unknown data block.

### 9.2.1 Analyzing entropy in Mathematica

(This part has been first appeared in my blog at 13-May-2015. Some discussion: <https://news.ycombinator.com/item?id=9545276>.)

It is possible to slice some file by blocks, probe each and draw a graph a graph. I did this in Wolfram Mathematica for demonstration and here is a source code (Mathematica 10):

```
(* loading the file *)
input=BinaryReadList["file.bin"];

(* setting block sizes *)
BlockSize=4096;BlockSizeToShow=256;
```

## 9.2. ANALYZING USING INFORMATION ENTROPY

```
(* slice blocks by 4k *)
blocks=Partition[input,BlockSize];

(* how many blocks we've got? *)
Length[blocks]

(* calculate entropy for each block. 2 in Entropy[] (base) is set with the intention so Entropy[
   ↴ []
function will produce the same results as Linux ent utility does *)
entropies=Map[N[Entropy[2,#]]&,blocks];

(* helper functions *)
fBlockToShow[input_,offset_]:=Take[input,{1+offset,1+offset+BlockSizeToShow}]
fToASCII[val_]:=FromCharacterCode[val,"PrintableASCII"]
fToHex[val_]:=IntegerString[val,16]
fPutASCIIWindow[data_]:=Framed[Grid[Partition[Map[fToASCII,data],16]]]
fPutHexWindow[data_]:=Framed[Grid[Partition[Map[fToHex,data],16],Alignment->Right]]

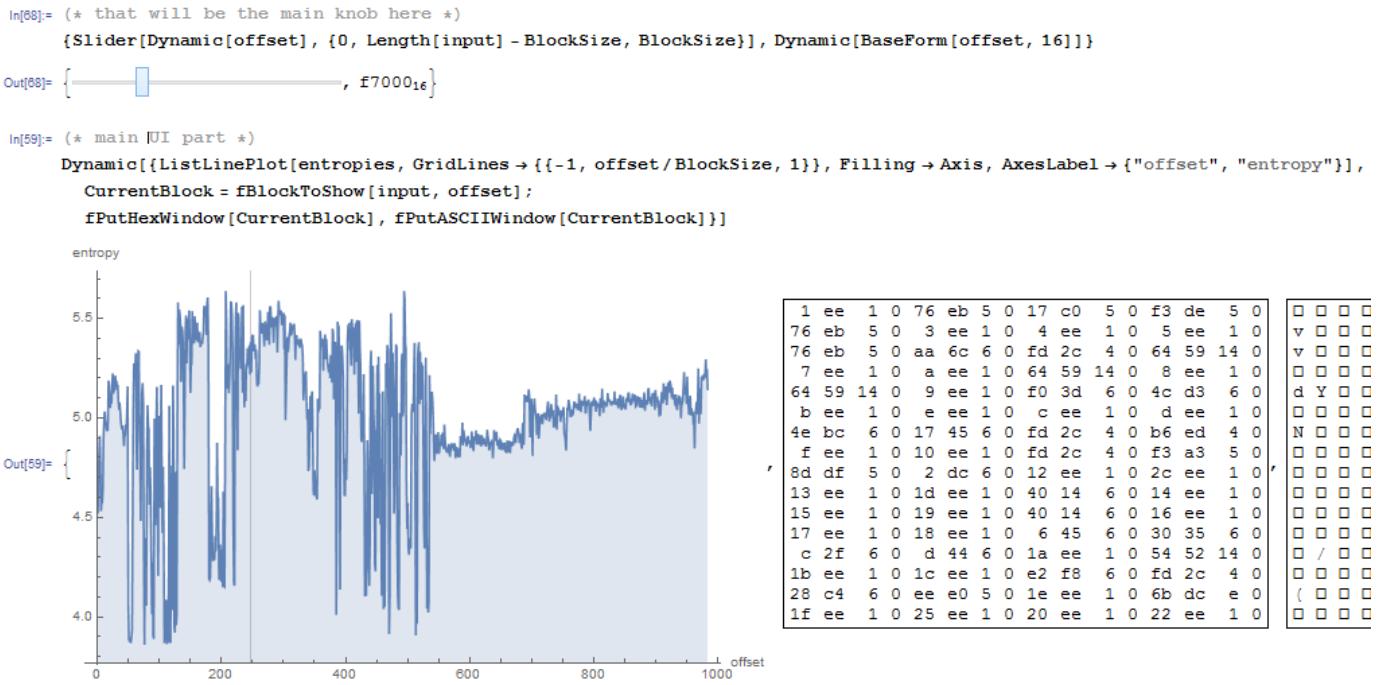
(* that will be the main knob here *)
{Slider[Dynamic[offset],{0,Length[input]-BlockSize,BlockSize}],Dynamic[BaseForm[offset,16]]}

(* main UI part *)
Dynamic[{ListLinePlot[entropies,GridLines->{{-1,offset/BlockSize,1}},Filling->Axis,AxesLabel[
   ↴ ->{"offset","entropy"}],
CurrentBlock=fBlockToShow[input,offset];
fPutHexWindow[CurrentBlock],
fPutASCIIWindow[CurrentBlock]}]
```

## GeoIP ISP database

Let's start with the [GeoIP](#) file (which assigns ISP to the block of IP addresses). This binary file (*GeoIPISP.dat*) has some tables (which are IP address ranges perhaps) plus some text blob at the end of the file (containing ISP names).

When I load it to Mathematica, I see this:

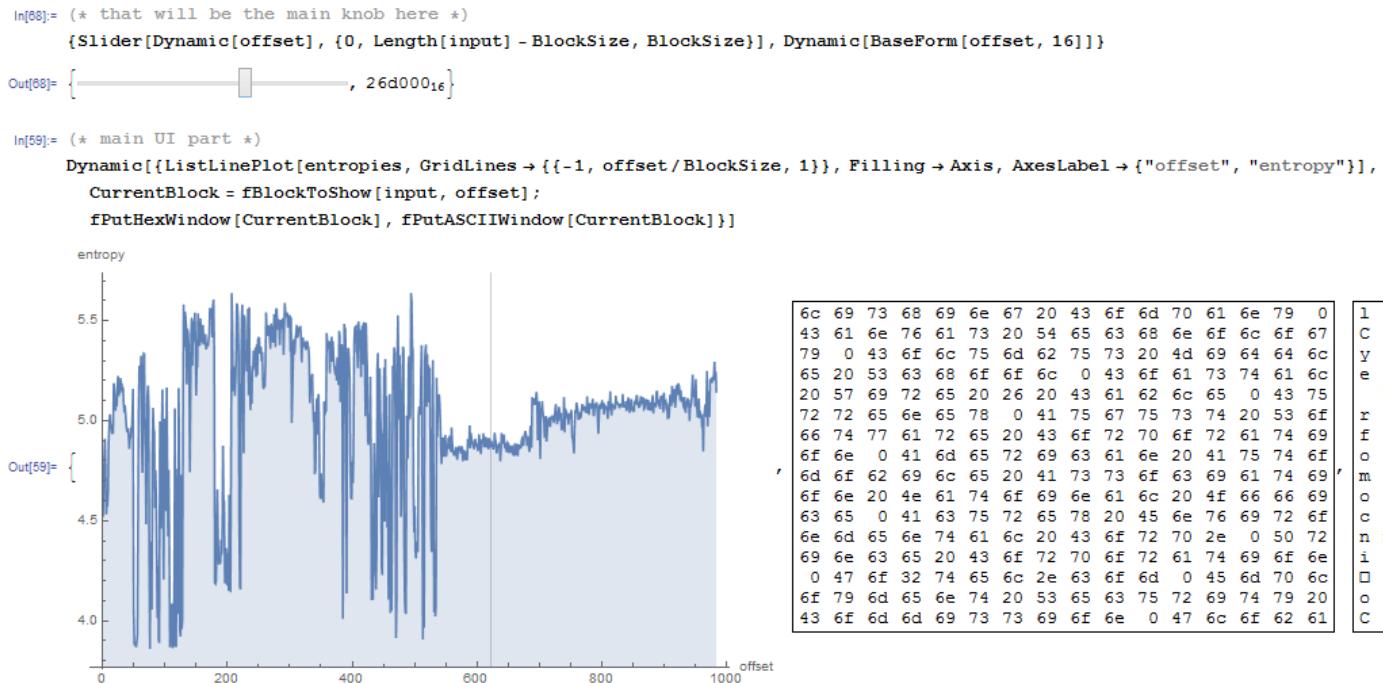


There are two parts in graph: first is somewhat chaotic, second is more steady.

0 in horizontal axis in graph means lowest entropy (the data which can be compressed very tightly, *ordered* in other words) and 8 is highest (cannot be compressed at all, *chaotic* or *random* in other words). Why 0 and 8? 0 means 0 bits per byte (byte slot is not filled at all) and 8 means 8 bits per byte, i.e., the whole byte slot is filled with the information tightly.

## 9.2. ANALYZING USING INFORMATION ENTROPY

So I put slider to point in the middle of the first block, and I clearly see some array of 32-bit integers. Now I put slider in the middle of the second block and I see English text:



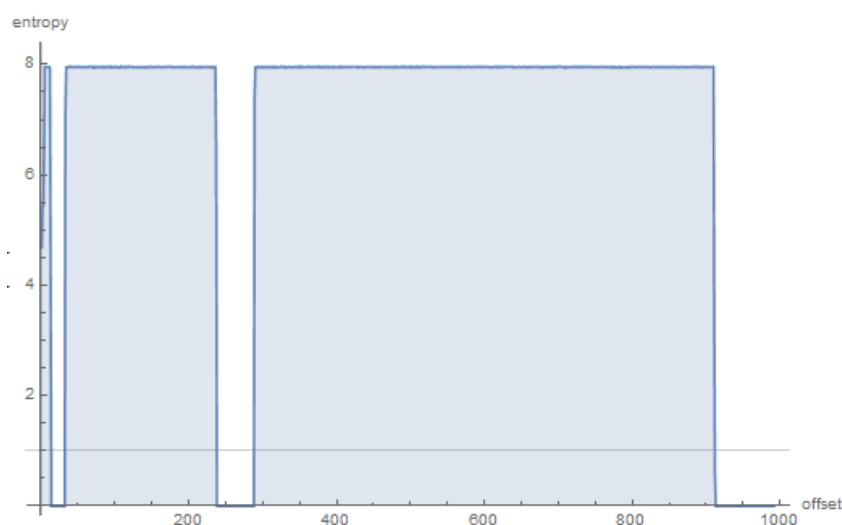
Indeed, this are names of ISPs. So, entropy of English text is 4.5-5.5 bits per byte? Yes, something like this. Wolfram Mathematica has some well-known English literature corpus embedded, and we can see entropy of Shakespeare's sonnets:

```
In[]:= Entropy[2,ExampleData[{"Text","ShakespearesSonnets"}]]//N
Out[]= 4.42366
```

4.4 is close to what we've got (4.7-5.3). Of course, classic English literature texts are somewhat different from ISP names and other English texts we can find in binary files (debugging/logging/error messages), but this value is close.

### TP-Link WR941 firmware

Now advanced example. I've got firmware for TP-Link WR941 router:



We see here 3 blocks with empty lacunas. The first block (started at address 0) is small, second (address somewhere at 0x22000) is bigger and third (address 0x123000) is biggest. I can't be sure about exact entropy of the first block, but 2nd and 3rd has very high entropy, meaning that these blocks are either compressed and/or encrypted.

## 9.2. ANALYZING USING INFORMATION ENTROPY

I tried [binwalk](#) for this firmware file:

DECIMAL	HEXADECIMAL	DESCRIPTION
0	0x0	TP-Link firmware header, firmware version: 0.-15221.3, image ↴ ↳ version: "", product ID: 0x0, product version: 155254789, kernel load address: 0x0, ↴ ↳ kernel entry point: 0x-7FFE000, kernel offset: 4063744, kernel length: 512, rootfs ↴ ↳ offset: 837431, rootfs length: 1048576, bootloader offset: 2883584, bootloader length: 0
14832	0x39F0	U-Boot version string, "U-Boot 1.1.4 (Jun 27 2014 - 14:56:49)"
14880	0x3A20	CRC32 polynomial table, big endian
16176	0x3F30	uImage header, header size: 64 bytes, header CRC: 0x3AC66E95, ↴ ↳ created: 2014-06-27 06:56:50, image size: 34587 bytes, Data Address: 0x80010000, Entry ↴ ↳ Point: 0x80010000, data CRC: 0xDF2DBA0B, OS: Linux, CPU: MIPS, image type: Firmware Image ↴ ↳ , compression type: lzma, image name: "u-boot image"
16240	0x3F70	LZMA compressed data, properties: 0x5D, dictionary size: 33554432 ↴ ↳ bytes, uncompressed size: 90000 bytes
131584	0x20200	TP-Link firmware header, firmware version: 0.0.3, image version: ↴ ↳ "", product ID: 0x0, product version: 155254789, kernel load address: 0x0, kernel entry ↴ ↳ point: 0x-7FFE000, kernel offset: 3932160, kernel length: 512, rootfs offset: 837431, ↴ ↳ rootfs length: 1048576, bootloader offset: 2883584, bootloader length: 0
132096	0x20400	LZMA compressed data, properties: 0x5D, dictionary size: 33554432 ↴ ↳ bytes, uncompressed size: 2388212 bytes
1180160	0x120200	Squashfs filesystem, little endian, version 4.0, compression:lzma ↴ ↳ , size: 2548511 bytes, 536 inodes, blocksize: 131072 bytes, created: 2014-06-27 07:06:52

Indeed: there are some stuff at the beginning, but two large LZMA compressed blocks are started at 0x20400 and 0x120200. These are roughly addresses we have seen in Mathematica. Oh, and by the way, binwalk can show entropy information as well (-E option):

DECIMAL	HEXADECIMAL	ENTROPY
0	0x0	Falling entropy edge (0.419187)
16384	0x4000	Rising entropy edge (0.988639)
51200	0xC800	Falling entropy edge (0.000000)
133120	0x20800	Rising entropy edge (0.987596)
968704	0xEC800	Falling entropy edge (0.508720)
1181696	0x120800	Rising entropy edge (0.989615)
3727360	0x38E000	Falling entropy edge (0.732390)

Rising edges are corresponding to rising edges of block on our graph. Falling edges are the points where empty lacunas are started.

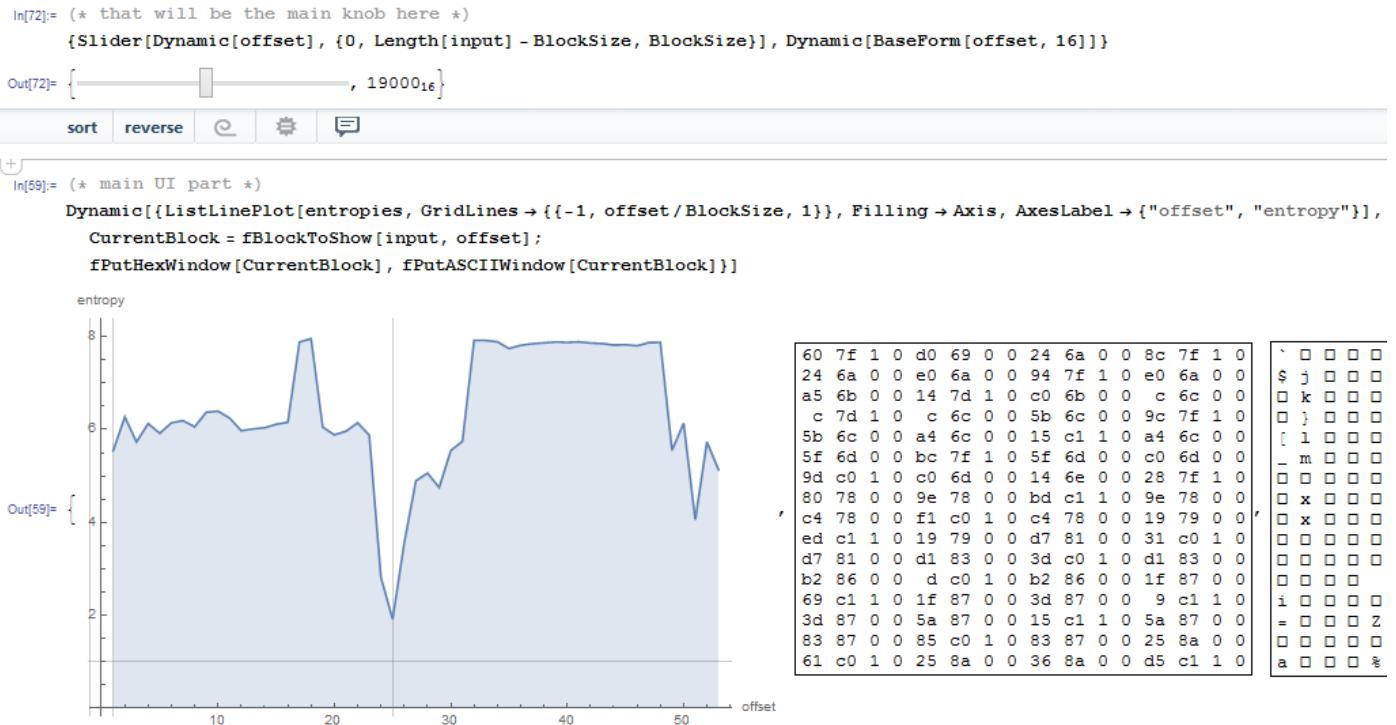
I wasn't able to force binwalk to generate PNG graphs (due to absence of some Python library), but here is an example how binwalk can do them: [http://binwalk.org/wp-content/uploads/2013/12/lg\\_dtv.png](http://binwalk.org/wp-content/uploads/2013/12/lg_dtv.png).

What can we say about lacunas? By looking in hex editor, we see that these are just filled with 0xFF bytes. Why developers put them? Perhaps, because they weren't able to calculate precise compressed blocks sizes, so they allocated space for them with some reserve.

## Notepad

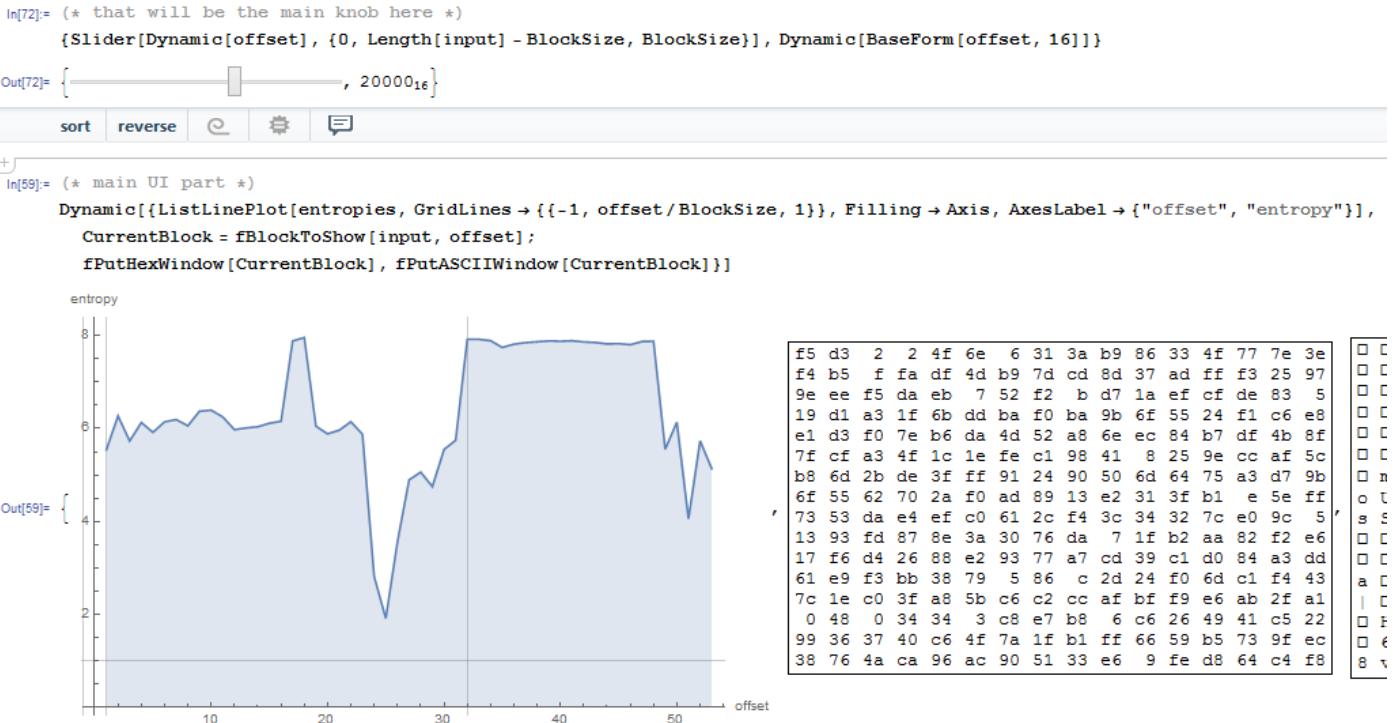
Another example is notepad.exe I've picked in Windows 8.1:

## 9.2. ANALYZING USING INFORMATION ENTROPY



There is cavity at 0x19000 (absolute file offset). I opened the executable file in hex editor and found imports table there (which has lower entropy than x86-64 code in the first half of graph).

There are also high entropy block started at 0x20000:

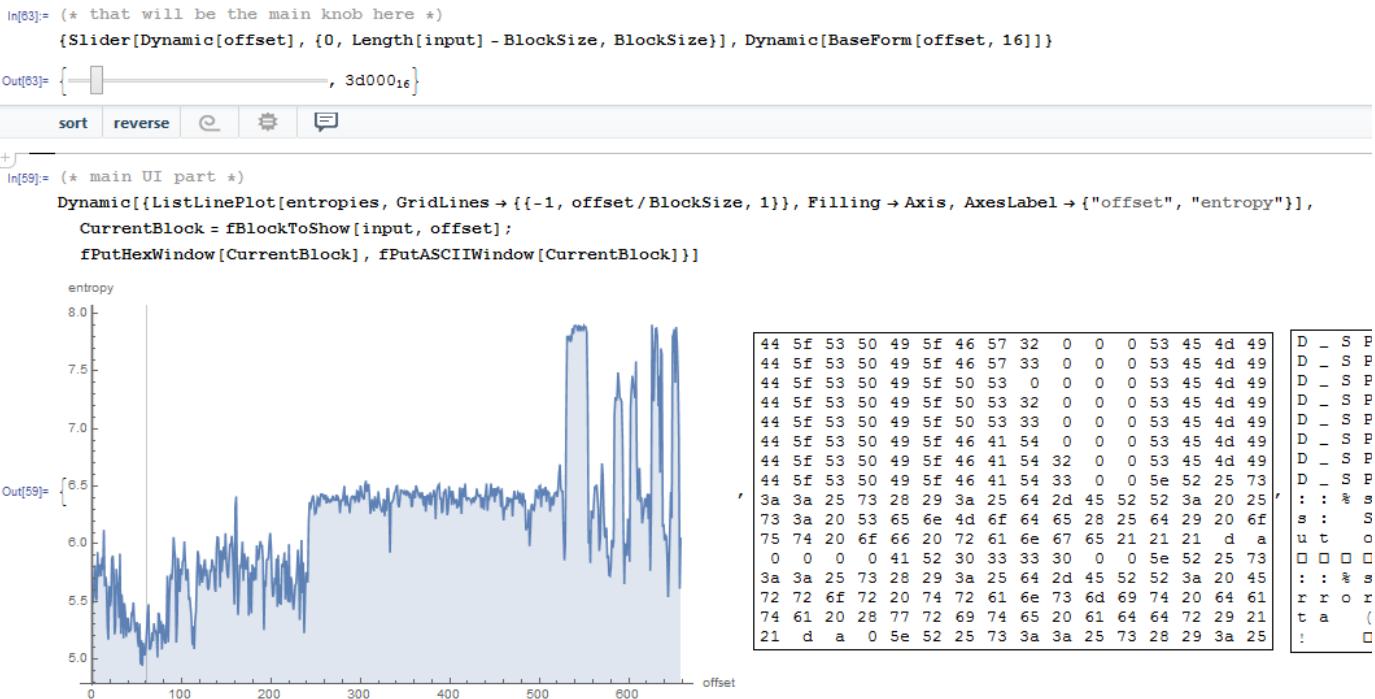


In hex editor I can see PNG file here, embedded in the PE file resource section (it is a large image of notepad icon). PNG files are compressed, indeed.

### Unnamed dashcam

Now the most advanced example in this article is the firmware of some unnamed dashcam I've received from friend:

## 9.2. ANALYZING USING INFORMATION ENTROPY



The cavity at the very beginning is an English text: debugging messages. I checked various ISAs and I found that the first third of the whole file (with the text segment inside) is in fact MIPS (little-endian) code!

For instance, this is very distinctive MIPS function epilogue:

ROM:000013B0	move	\$sp, \$fp
ROM:000013B4	lw	\$ra, 0x1C(\$sp)
ROM:000013B8	lw	\$fp, 0x18(\$sp)
ROM:000013BC	lw	\$s1, 0x14(\$sp)
ROM:000013C0	lw	\$s0, 0x10(\$sp)
ROM:000013C4	jr	\$ra
ROM:000013C8	addiu	\$sp, 0x20

From our graph we can see that MIPS code has entropy of 5-6 bits per byte. Indeed, I once measured various ISAs entropy and I've got these values:

- x86: .text section of ntoskrnl.exe file from Windows 2003: 6.6
- x64: .text section of ntoskrnl.exe file from Windows 7 x64: 6.5
- ARM (thumb mode), Angry Birds Classic: 7.05
- ARM (ARM mode) Linux Kernel 3.8.0: 6.03
- MIPS (little endian), .text section of user32.dll from Windows NT 4: 6.09

So the entropy of executable code is higher than of English text, but still can be compressed.

Now the second third is started at 0xF5000. I don't know what this is. I tried different ISAs but without success. The entropy of the block looks even steadier than for executable one. Maybe some kind of data?

There is also a spike at 0x213000. I checked it in hex editor and I found JPEG file there (which, of course, compressed)! I also don't know what is at the end. Let's try Binwalk for this file:

```
dennis@ubuntu:~/P/entropy$ binwalk FW96650A.bin
```

DECIMAL	HEXADECIMAL	DESCRIPTION
-----		
167698	0x28F12	Unix path: /15/20/24/25/30/60/120/240fps can be served..
280286	0x446DE	Copyright string: "Copyright (c) 2012 Novatek Microelectronic ↴ Corp."
2169199	0x21196F	JPEG image data, JFIF standard 1.01
2300847	0x231BAF	MySQL MISAM compressed data file Version 3

```
dennis@ubuntu:~/P/entropy$ binwalk -E FW96650A.bin
```

## 9.2. ANALYZING USING INFORMATION ENTROPY

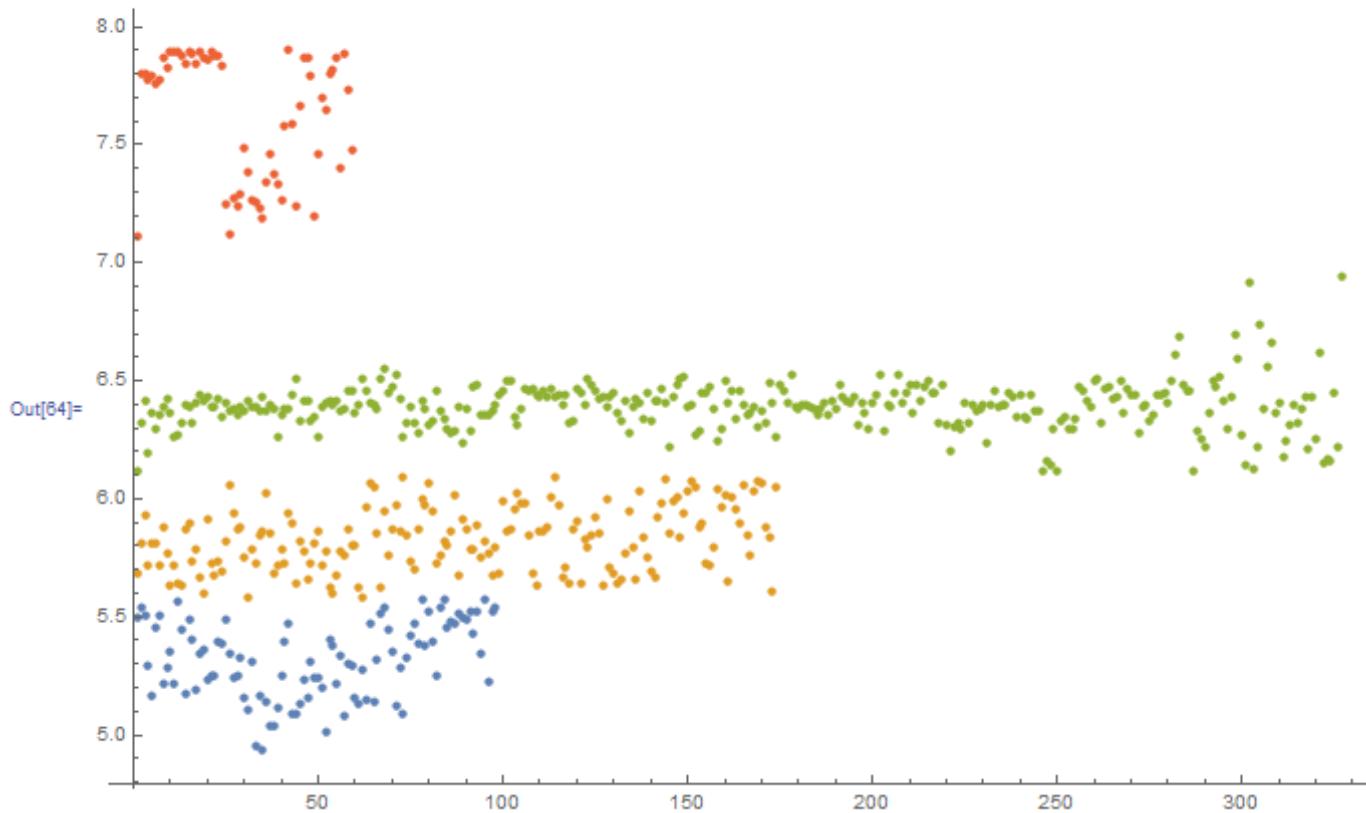
WARNING: pyqtgraph not found, visual entropy graphing will be disabled

DECIMAL	HEXADECIMAL	ENTROPY
0	0x0	Falling entropy edge (0.579792)
2170880	0x212000	Rising entropy edge (0.967373)
2267136	0x229800	Falling entropy edge (0.802974)
2426880	0x250800	Falling entropy edge (0.846639)
2490368	0x260000	Falling entropy edge (0.849804)
2560000	0x271000	Rising entropy edge (0.974340)
2574336	0x274800	Rising entropy edge (0.970958)
2588672	0x278000	Falling entropy edge (0.763507)
2592768	0x279000	Rising entropy edge (0.951883)
2596864	0x27A000	Falling entropy edge (0.712814)
2600960	0x27B000	Rising entropy edge (0.968167)
2607104	0x27C800	Rising entropy edge (0.958582)
2609152	0x27D000	Falling entropy edge (0.760989)
2654208	0x288000	Rising entropy edge (0.954127)
2670592	0x28C000	Rising entropy edge (0.967883)
2676736	0x28D800	Rising entropy edge (0.975779)
2684928	0x28F800	Falling entropy edge (0.744369)

Yes, it found JPEG file and even MySQL data! But I'm not sure if it's true—I didn't check it yet.

It's also interesting to try clusterization in Mathematica:

```
In[64]:= (* let also take a look on clustering attempt of Mathematica *)
ListPlot[FindClusters[entropies]]
```



Here is an example of how Mathematica grouped various entropy values into distinctive groups. Indeed, there is something credible. Blue dots in range of 5.0-5.5 are supposedly related to English text. Yellow dots in 5.5-6 are MIPS code. A lot of green dots in 6.0-6.5 is the unknown second third. Orange dots close to 8.0 are related to compressed JPEG file. Other orange dots are supposedly related to the end of the firmware (unknown to us data).

**Links**

Binary files used while writing: <http://yurichev.com/blog/entropy/files/>. Wolfram Mathematica notebook file: [http://yurichev.com/blog/entropy/files/binary\\_file\\_entropy.nb](http://yurichev.com/blog/entropy/files/binary_file_entropy.nb) (all cells must be evaluated to start things working).

**9.2.2 Conclusion**

Information entropy can be used as a quick-n-dirty method for inspecting unknown binary files. In particular, it is a very quick way to find compressed/encrypted pieces of data. Someone say it's possible to find RSA (and other asymmetric cryptographic algorithms) public/private keys in executable code (which has high entropy as well), but I didn't try this myself.

**9.2.3 Tools**

Handy Linux `ent` utility to measure entropy of a file<sup>6</sup>.

There is a great online entropy visualizer made by Aldo Cortesi, which I tried to mimic using Mathematica: <http://binvis.io>. His articles about entropy visualization are worth reading: <http://corte.si/posts/visualisation/entropy/index.html>, <http://corte.si/posts/visualisation/malware/index.html>, <http://corte.si/posts/visualisation/binvis/index.html>.

`radare2` framework has `#entropy` command for this.

A tool for IDA: `IDAtropy`<sup>7</sup>.

**9.2.4 A word about primitive encryption like XORing**

It's interesting that simple XOR encryption doesn't affect entropy of data. I've shown this in *Norton Guide* example in the book ([9.1.1 on page 926](#)).

Generalizing: encryption by substitution cipher also doesn't affect entropy of data (and XOR can be viewed as substitution cipher). The reason of that is because entropy calculation algorithm view data on byte-level. On the other hand, the data encrypted by 2 or 4-byte XOR pattern will result in another entropy.

Nevertheless, low entropy is usually a good sign of weak amateur cryptography (which is also used in license keys, license files, etc.).

**9.2.5 More about entropy of executable code**

It is quickly noticeable that probably a biggest source of high-entropy in executable code are relative offsets encoded in opcodes. For example, these two consequent instructions will produce different relative offsets in their opcodes, while they are in fact pointing to the same function:

```
function proc
...
function endp
...
CALL function
...
CALL function
```

Ideal executable code compressor would encode information like this: *there is a CALL to a "function" at address X and the same CALL at address Y without necessity to encode address of the function twice*.

To deal with this, executable compressors are sometimes able to reduce entropy here. One example is UPX: <http://sourceforge.net/p/upx/code/ci/default/tree/doc/filter.txt>.

<sup>6</sup><http://www.fourmilab.ch/random/>

<sup>7</sup><https://github.com/danigargu/IDAtropy>

## 9.2.6 Random number generators

When I run GnuPG to generate new secret key, it asking for some entropy...

We need to generate a lot of random bytes. It is a good idea to perform some other action (type on the keyboard, move the mouse, utilize the disks) during the prime generation; this gives the random number generator a better chance to gain enough entropy.

Not enough random bytes available. Please do some other work to give the OS a chance to collect more entropy! (Need 169 more bytes)

This means that good a PRNG produces long high-entropy results, and this is what the secret asymmetrical cryptographical key needs. But [CPRNG<sup>8</sup>](#) is tricky (because computer is highly deterministic device itself), so the GnuPG asking for some additional randomness from the user.

Here is a case where I made attempt to calculate entropy of some unknown blocks: [8.7 on page 861](#).

### 9.2.7 Entropy of various files

Entropy of random bytes is close to 8:

```
$ dd bs=1M count=1 if=/dev/urandom | ent
Entropy = 7.999803 bits per byte.
```

This means, almost all available space inside of byte is filled with information.

256 bytes in range of 0..255 gives exact value of 8:

```
#!/usr/bin/env python
import sys

for i in range(256):
    sys.stdout.write(chr(i))
```

```
python 1.py | ent
Entropy = 8.000000 bits per byte.
```

Order of bytes doesn't matter. This means, all available space inside of byte is filled.

Entropy of all zero bytes is 0:

```
$ dd bs=1M count=1 if=/dev/zero | ent
Entropy = 0.000000 bits per byte.
```

Entropy of a string consisting of a single (any) byte is 0:

```
$ echo -n "aaaaaaaaaaaaaaaaaa" | ent
Entropy = 0.000000 bits per byte.
```

Entropy of base64 string is the same as entropy of source data, but multiplied by  $\frac{3}{4}$ . This is because base64 encoding uses 64 symbols instead of 256.

```
$ dd bs=1M count=1 if=/dev/urandom | base64 | ent
Entropy = 6.022068 bits per byte.
```

Perhaps, 6.02 not that close to 6 because padding symbols (=) spoils our statistics for a little.

Uuencode, also uses 64 symbols:

```
$ dd bs=1M count=1 if=/dev/urandom | uuencode - | ent
Entropy = 6.013162 bits per byte.
```

This means, any base64 and Uuencode strings can be transmitted using 6-bit bytes or characters.

Any random information in hexadecimal form has entropy of 4 bits per byte:

<sup>8</sup>Cryptographically secure Pseudorandom Number Generator

## 9.2. ANALYZING USING INFORMATION ENTROPY

```
$ openssl rand -hex $(( 2**16 )) | ent
Entropy = 4.000013 bits per byte.
```

Entropy of randomly picked English language text from Gutenberg library has entropy 4.5. The reason of this is because English texts uses mostly 26 symbols, and  $\log_2(26) = 4.7$ , i.e., you would need 5-bit bytes to transmit uncompressed English texts, that would be enough (it was indeed so in teletype era).

Randomly chosen Russian language text from <http://lib.ru> library is F.M.Dostoevsky "Idiot"<sup>9</sup>, internally encoded in CP1251 encoding.

And this file has entropy of 4.98. Russian language has 33 characters, and  $\log_2(33) = 5.04$ . But it has unpopular and rare “ё” character<sup>10</sup>. And  $\log_2(32) = 5$  (Russian alphabet without this rare character) — now this close to what we've got.

In fact, the text we studying uses “ё” letter, but, probably, it's still rarely used there.

The very same file transcoded from CP1251 to UTF-8 gave entropy of 4.23. Each Cyrillic character encoded in UTF-8 is usually encoded as a pair, and the first byte is always one of: 0xD0 or 0xD1. Perhaps, this caused bias.

Let's generate random bits and output them as “T” or “F” characters:

```
#!/usr/bin/env python
import random, sys

rt=""
for i in range(102400):
    if random.randint(0,1)==1:
        rt=rt+"T"
    else:
        rt=rt+"F"
print rt
```

Sample: ...TTTFTFTTTFFFFTTTFTTTTTFTTTFFTTFTFTTTFTTTFFFF... .

Entropy is very close to 1 (i.e., 1 bit per byte).

Let's generate random decimal numbers:

```
#!/usr/bin/env python
import random, sys

rt=""
for i in range(102400):
    rt=rt,"%d" % random.randint(0,9)
print rt
```

Sample: ...52203466119390328807552582367031963888032.... .

Entropy will be close to 3.32, indeed, this is  $\log_2(10)$ .

### 9.2.8 Making lower level of entropy

The author of these lines once saw a software which stored each byte of encrypted data in 3 bytes: each has roughly  $\frac{byte}{3}$  value, so reconstructing encrypted byte back involving summing up 3 consecutive bytes. Looks absurdly.

But some people say this was done in order to conceal the very fact the data has something encrypted inside: measuring entropy of such block will show much lower level of it.

<sup>9</sup>[http://az.lib.ru/d/dostoevskij\\_f\\_m/text\\_0070.shtml](http://az.lib.ru/d/dostoevskij_f_m/text_0070.shtml)

<sup>10</sup>When I typed it here in text, I've needed to look down to my keyboard and find it.

## 9.3 Millennium game save file

The “Millenium Return to Earth” is an ancient DOS game (1991), that allows you to mine resources, build ships, equip them on other planets, and so on<sup>11</sup>.

Like many other games, it allows you to save all game state into a file.

Let's see if we can find something in it.

---

<sup>11</sup>It can be downloaded for free [here](#)

### 9.3. MILLENIUM GAME SAVE FILE

So there is a mine in the game. Mines at some planets work faster, or slower on others. The set of resources is also different.

Here we can see what resources are mined at the time:



Figure 9.12: Mine: state 1

Let's save a game state. This is a file of size 9538 bytes.

Let's wait some "days" here in the game, and now we've got more resources from the mine:

### 9.3. MILLENIUM GAME SAVE FILE



Figure 9.13: Mine: state 2

Let's save game state again.

Now let's try to just do binary comparison of the save files using the simple DOS/Windows FC utility:

```
...> FC /b 2200save.i.v1 2200SAVE.I.V2

Comparing files 2200save.i.v1 and 2200SAVE.I.V2
00000016: 0D 04
00000017: 03 04
0000001C: 1F 1E
00000146: 27 3B
00000BDA: 0E 16
00000BDC: 66 9B
00000BDE: 0E 16
00000BE0: 0E 16
00000BE6: DB 4C
00000BE7: 00 01
00000BE8: 99 E8
00000BEC: A1 F3
00000BEE: 83 C7
00000BFB: A8 28
00000BFD: 98 18
00000BFF: A8 28
00000C01: A8 28
00000C07: D8 58
00000C09: E4 A4
00000COD: 38 B8
00000COF: E8 68
...
```

The output is incomplete here, there are more differences, but we will cut result to show the most interesting.

In the first state, we have 14 “units” of hydrogen and 102 “units” of oxygen.

### *9.3. MILLENIUM GAME SAVE FILE*

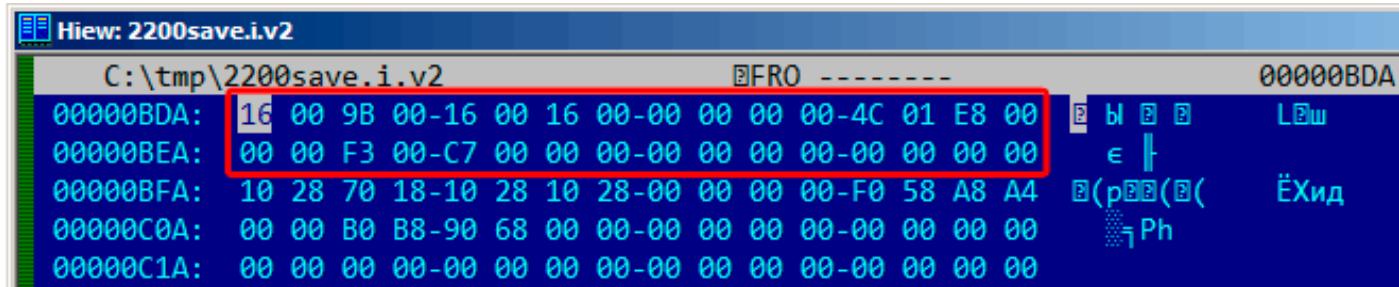
---

We have 22 and 155 “units” respectively in the second state. If these values are saved into the save file, we would see this in the difference. And indeed we do. There is 0x0E (14) at position 0xBDA and this value is 0x16 (22) in the new version of the file. This is probably hydrogen. There is 0x66 (102) at position 0xBDC in the old version and 0x9B (155) in the new version of the file. This seems to be the oxygen.

Both files are available on the website for those who wants to inspect them (or experiment) more: [beginners.re](#).

### 9.3. MILLENIUM GAME SAVE FILE

Here is the new version of file opened in Hiew, we marked the values related to the resources mined in the game:



C:\tmp\2200save.i.v2	00000BDA
00000BDA: 16 00 9B 00-16 00 16 00-00 00 00 00-4C 01 E8 00	ы ў ў Лш
00000BEA: 00 00 F3 00-C7 00 00 00-00 00 00 00-00 00 00 00	е
00000BFA: 10 28 70 18-10 28 10 28-00 00 00 00-F0 58 A8 A4	Б(рББ(Б( ЁХид
00000C0A: 00 00 B0 B8-90 68 00 00-00 00 00 00-00 00 00 00	Ph
00000C1A: 00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00	

Figure 9.14: Hiew: state 1

Let's check each, and these are.

These are clearly 16-bit values: not a strange thing for 16-bit DOS software where the *int* type has 16-bit width.

### 9.3. MILLENIUM GAME SAVE FILE

Let's check our assumptions. We will write the 1234 (0x4D2) value at the first position (this must be hydrogen):

C:\tmp\2200save.i.v2 EDITMODE 00000BDC  
00000BDA: D2 04 9B 00-16 00 16 00-00 00 00 00-4C 01 E8 00  
00000BEA: 00 00 F3 00-C7 00 00 00-00 00 00 00-00 00 00 00  
00000BFA: 10 28 70 18-10 28 10 28-00 00 00 00-F0 58 A8 A4  
00000C0A: 00 00 B0 B8-90 68 00 00-00 00 00 00-00 00 00 00  
00000C1A: 00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00

Figure 9.15: Hiew: let's write 1234 (0x4D2) there

Then we will load the changed file in the game and took a look at mine statistics:

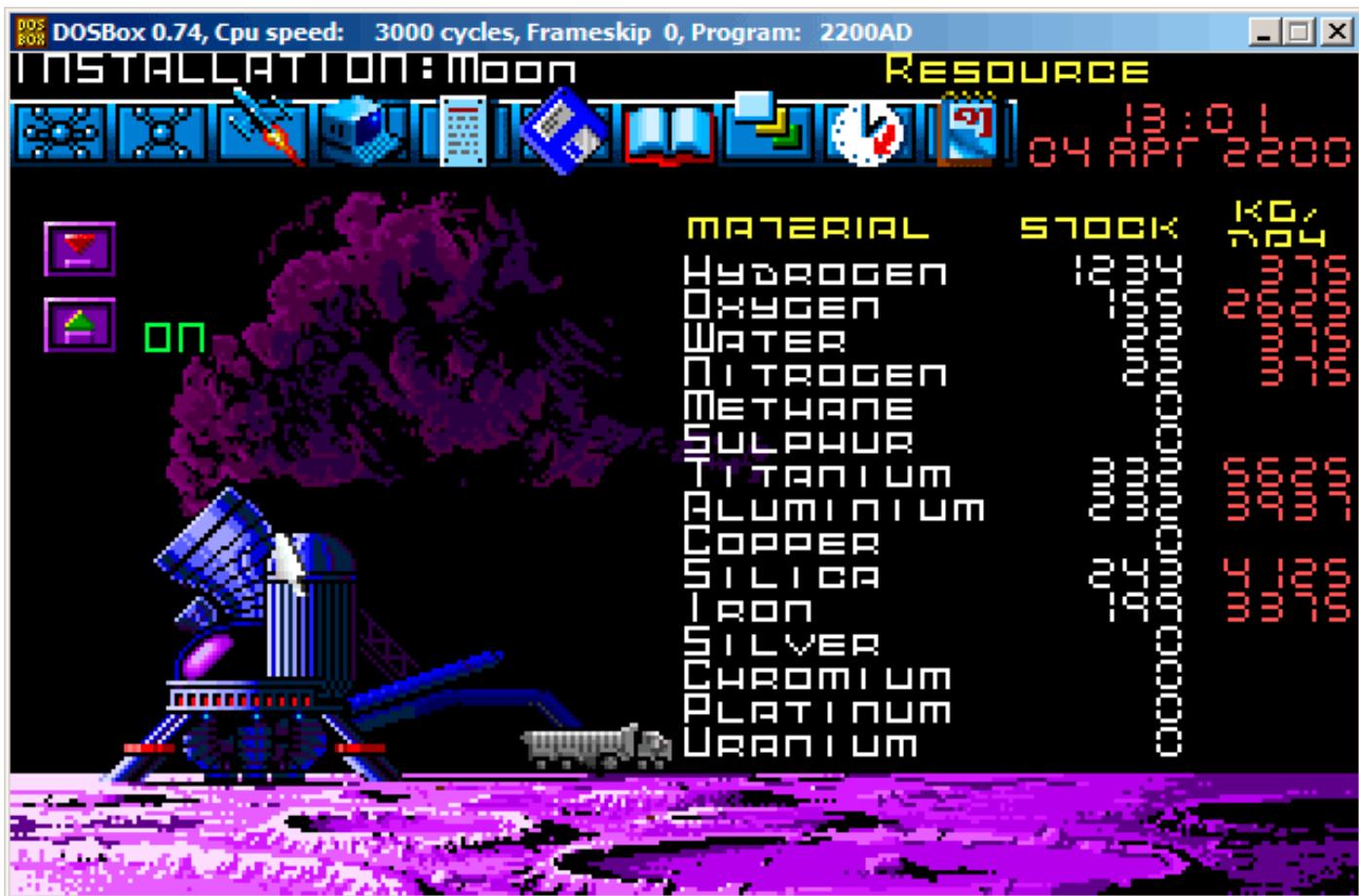
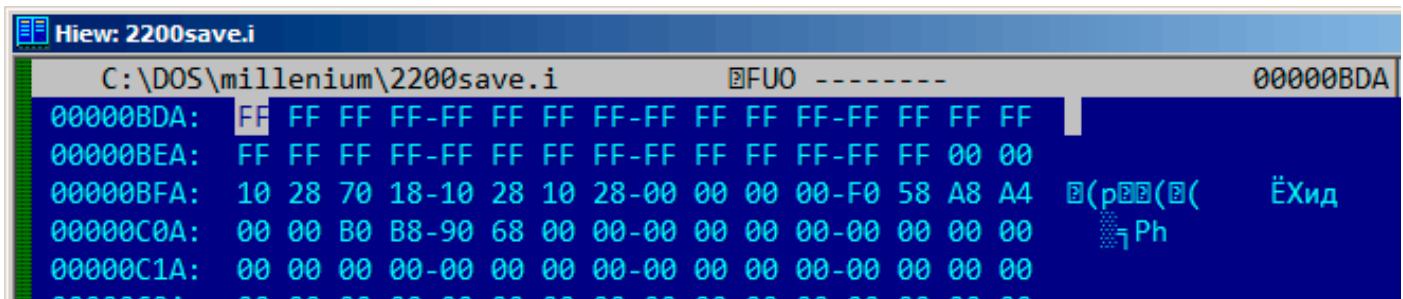


Figure 9.16: Let's check for hydrogen value

So yes, this is it.

### 9.3. MILLENIUM GAME SAVE FILE

Now let's try to finish the game as soon as possible, set the maximal values everywhere:



Hiew: 2200save.i

C:\DOS\millenium\2200save.i FU0 ----- 00000BDA

00000BDA:	FF	FF	FF	FF-FF	FF	FF	FF-FF	FF	FF	FF-FF	FF	FF	FF	FF	FF
00000BEA:	FF	FF	FF	FF-FF	FF	FF	FF-FF	FF	FF	FF-FF	FF	FF	FF	00	00
00000BFA:	10	28	70	18-10	28	10	28-00	00	00	00-F0	58	A8	A4	0(р00(0(	ЁХид
00000C0A:	00	00	B0	B8-90	68	00	00-00	00	00	00-00	00	00	00	00	00
00000C1A:	00	00	00	00-00	00	00	00-00	00	00	00-00	00	00	00	00	00
00000C2A:	00	00	00	00-00	00	00	00-00	00	00	00-00	00	00	00	00	00

Figure 9.17: Hiew: let's set maximal values

0xFFFF is 65535, so yes, we now have a lot of resources:



Figure 9.18: All resources are 65535 (0xFFFF) indeed

## 9.4. FORTUNE PROGRAM INDEXING FILE

Let's skip some "days" in the game and oops! We have a lower amount of some resources:



Figure 9.19: Resource variables overflow

That's just overflow.

The game's developer supposedly didn't think about such high amounts of resources, so there are probably no overflow checks, but the mine is "working" in the game, resources are added, hence the overflows. Apparently, it is a bad idea to be that greedy.

There are probably a lot of more values saved in this file.

So this is very simple method of cheating in games. High score files often can be easily patched like that. More about files and memory snapshots comparing: [5.10.6 on page 730](#).

## 9.4 Reverse engineering of simple *fortune* program indexing file

(This part was first appeared in my blog at 25-Apr-2015.)

*fortune* is well-known UNIX program which shows random phrase from a collection. Some geeks are often set up their system in such way, so *fortune* can be called after logging on. *fortune* takes phrases from the text files laying in `/usr/share/games/fortunes` (as of Ubuntu Linux). Here is example (“fortunes” text file):

A day for firm decisions!!!!!! Or is it?  
%  
A few hours grace before the madness begins again.  
%  
A gift of a flower will soon be made to you.  
%  
A long-forgotten loved one will appear soon.  
  
Buy the negatives at any price.  
%

#### 9.4. FORTUNE PROGRAM INDEXING FILE

A tall, dark stranger will have more fun than you.

%

...

So it is just phrases, sometimes multiline ones, divided by percent sign. The task of *fortune* program is to find random phrase and to print it. In order to achieve this, it must scan the whole text file, count phrases, choose random and print it. But the text file can get bigger, and even on modern computers, this naive algorithm is a bit uneconomical to computer resources. The straightforward way is to keep binary index file containing offset of each phrase in text file. With index file, *fortune* program can work much faster: just to choose random index element, take offset from there, set offset in text file and read phrase from it. This is actually done in *fortune* file. Let's inspect what is in its index file inside (these are .dat files in the same directory) in hexadecimal editor. This program is open-source of course, but intentionally, I will not peek into its source code.

```
$ od -t x1 --address-radix=x fortunes.dat
000000 00 00 00 02 00 00 01 af 00 00 00 bb 00 00 00 0f
000010 00 00 00 00 25 00 00 00 00 00 00 00 00 00 00 00 2b
000020 00 00 00 60 00 00 00 8f 00 00 00 df 00 00 01 14
000030 00 00 01 48 00 00 01 7c 00 00 01 ab 00 00 01 e6
000040 00 00 02 20 00 00 02 3b 00 00 02 7a 00 00 02 c5
000050 00 00 03 04 00 00 03 3d 00 00 03 68 00 00 03 a7
000060 00 00 03 e1 00 00 04 19 00 00 04 2d 00 00 04 7f
000070 00 00 04 ad 00 00 04 d5 00 00 05 05 00 00 05 3b
000080 00 00 05 64 00 00 05 82 00 00 05 ad 00 00 05 ce
000090 00 00 05 f7 00 00 06 1c 00 00 06 61 00 00 06 7a
0000a0 00 00 06 d1 00 00 07 0a 00 00 07 53 00 00 07 9a
0000b0 00 00 07 f8 00 00 08 27 00 00 08 59 00 00 08 8b
0000c0 00 00 08 a0 00 00 08 c4 00 00 08 e1 00 00 08 f9
0000d0 00 00 09 27 00 00 09 43 00 00 09 79 00 00 09 a3
0000e0 00 00 09 e3 00 00 0a 15 00 00 0a 4d 00 00 0a 5e
0000f0 00 00 0a 8a 00 00 0a a6 00 00 0a bf 00 00 0a ef
000100 00 00 0b 18 00 00 0b 43 00 00 0b 61 00 00 0b 8e
000110 00 00 0b cf 00 00 0b fa 00 00 0c 3b 00 00 0c 66
000120 00 00 0c 85 00 00 0c b9 00 00 0c d2 00 00 0d 02
000130 00 00 0d 3b 00 00 0d 67 00 00 0d ac 00 00 0d e0
000140 00 00 0e 1e 00 00 0e 67 00 00 0e a5 00 00 0e da
000150 00 00 0e ff 00 00 0f 43 00 00 0f 8a 00 00 0f bc
000160 00 00 0f e5 00 00 10 1e 00 00 10 63 00 00 10 9d
000170 00 00 10 e3 00 00 11 10 00 00 11 46 00 00 11 6c
000180 00 00 11 99 00 00 11 cb 00 00 11 f5 00 00 12 32
000190 00 00 12 61 00 00 12 8c 00 00 12 ca 00 00 13 87
0001a0 00 00 13 c4 00 00 13 fc 00 00 14 1a 00 00 14 6f
0001b0 00 00 14 ae 00 00 14 de 00 00 15 1b 00 00 15 55
0001c0 00 00 15 a6 00 00 15 d8 00 00 16 0f 00 00 16 4e
...
...
```

Without any special aid we could see that there are four 4-byte elements on each 16-byte line. Perhaps, it's our index array. I'm trying to load the whole file in Wolfram Mathematica as 32-bit integer array:

```
In[]:= BinaryReadList["c:/tmp1/fortunes.dat", "UnsignedInteger32"]
Out[]={33554432, 2936078336, 3137339392, 251658240, 0, 37, 0, \
721420288, 1610612736, 2399141888, 3741319168, 335609856, 1208025088, \
2080440320, 2868969472, 3858825216, 537001984, 989986816, 2046951424, \
3305242624, 67305472, 1023606784, 1745027072, 2801991680, 3775070208, \
419692544, 755236864, 2130968576, 2902720512, 3573809152, 84213760, \
990183424, 1678049280, 2181365760, 2902786048, 3456434176, \
4144300032, 470155264, 1627783168, 2047213568, 3506831360, 168230912, \
1392967680, 2584150016, 4161208320, 654835712, 1493696512, \
2332557312, 2684878848, 3288858624, 3775397888, 4178051072, \
...
...
```

Nope, something wrong. Numbers are suspiciously big. But let's back to *od* output: each common 4-byte element has two zero bytes and two non-zero bytes, so the offsets (at least at the beginning of the file) are 16-bit at maximum. Probably different endianness is used in the file? Default endianness in Mathematica is little-endian, as used in Intel CPUs. Now I'm changing it to big-endian:

```
In[]:= BinaryReadList["c:/tmp1/fortunes.dat", "UnsignedInteger32",
ByteOrdering -> 1]
```

## 9.4. FORTUNE PROGRAM INDEXING FILE

```
Out[] = {2, 431, 187, 15, 0, 620756992, 0, 43, 96, 143, 223, 276, \
328, 380, 427, 486, 544, 571, 634, 709, 772, 829, 872, 935, 993, \
1049, 1069, 1151, 1197, 1237, 1285, 1339, 1380, 1410, 1453, 1486, \
1527, 1564, 1633, 1658, 1745, 1802, 1875, 1946, 2040, 2087, 2137, \
2187, 2208, 2244, 2273, 2297, 2343, 2371, 2425, 2467, 2531, 2581, \
2637, 2654, 2698, 2726, 2751, 2799, 2840, 2883, 2913, 2958, 3023, \
3066, 3131, 3174, 3205, 3257, 3282, 3330, 3387, 3431, 3500, 3552, \
...}
```

Yes, this is something readable. I choose random element (3066) which is 0xBFA in hexadecimal form. I'm opening 'fortunes' text file in hex editor, I'm setting 0xBFA as offset and I see this phrase:

```
$ od -t x1 -c --skip-bytes=0xbfa --address-radix=x fortunes
000bfa 44 6f 20 77 68 61 74 20 63 6f 6d 65 73 20 6e 61
      D   o     w   h   a   t     c   o   m   e   s     n   a
000c0a 74 75 72 61 6c 6c 79 2e 20 20 53 65 65 74 68 65
      t   u   r   a   l   l   y   .     S   e   e   t   h   e
000c1a 20 61 6e 64 20 66 75 6d 65 20 61 6e 64 20 74 68
      a   n   d     f   u   m   e     a   n   d     t   h
....
```

Or:

```
Do what comes naturally. Seethe and fume and throw a tantrum.
%
```

Other offset are also can be checked, yes, they are valid offsets.

I can also check in Mathematica that each subsequent element is bigger than previous. I.e., the array is ascending. In mathematics lingo, this is called *strictly increasing monotonic function*.

```
In]:= Differences[input]
```

```
Out[] = {429, -244, -172, -15, 620756992, -620756992, 43, 53, 47, \
80, 53, 52, 52, 47, 59, 58, 27, 63, 75, 63, 57, 43, 63, 58, 56, 20, \
82, 46, 40, 48, 54, 41, 30, 43, 33, 41, 37, 69, 25, 87, 57, 73, 71, \
94, 47, 50, 50, 21, 36, 29, 24, 46, 28, 54, 42, 64, 50, 56, 17, 44, \
28, 25, 48, 41, 43, 30, 45, 65, 43, 65, 43, 31, 52, 25, 48, 57, 44, \
69, 52, 62, 73, 62, 53, 37, 68, 71, 50, 41, 57, 69, 58, 70, 45, 54, \
38, 45, 50, 42, 61, 47, 43, 62, 189, 61, 56, 30, 85, 63, 48, 61, 58, \
81, 50, 55, 63, 83, 80, 49, 42, 94, 54, 67, 81, 52, 57, 68, 43, 28, \
120, 64, 53, 81, 33, 82, 88, 29, 61, 32, 75, 63, 70, 47, 101, 60, 79, \
33, 48, 65, 35, 59, 47, 55, 22, 43, 35, 102, 53, 80, 65, 45, 31, 29, \
69, 32, 25, 38, 34, 35, 49, 59, 39, 41, 18, 43, 41, 83, 37, 31, 34, \
59, 72, 72, 81, 77, 53, 53, 50, 51, 45, 53, 39, 70, 54, 103, 33, 70, \
51, 95, 67, 54, 55, 65, 61, 54, 54, 53, 45, 100, 63, 48, 65, 71, 23, \
28, 43, 51, 61, 101, 65, 39, 78, 66, 43, 36, 56, 40, 67, 92, 65, 61, \
31, 45, 52, 94, 82, 82, 91, 46, 76, 55, 19, 58, 68, 41, 75, 30, 67, \
92, 54, 52, 108, 60, 56, 76, 41, 79, 54, 65, 74, 112, 76, 47, 53, 61, \
66, 53, 28, 41, 81, 75, 69, 89, 63, 60, 18, 18, 50, 79, 92, 37, 63, \
88, 52, 81, 60, 80, 26, 46, 80, 64, 78, 70, 75, 46, 91, 22, 63, 46, \
34, 81, 75, 59, 62, 66, 74, 76, 111, 55, 73, 40, 61, 55, 38, 56, 47, \
78, 81, 62, 37, 41, 60, 68, 40, 33, 54, 34, 41, 36, 49, 44, 68, 51, \
50, 52, 36, 53, 66, 46, 41, 45, 51, 44, 44, 33, 72, 40, 71, 57, 55, \
39, 66, 40, 56, 68, 43, 88, 78, 30, 54, 64, 36, 55, 35, 88, 45, 56, \
76, 61, 66, 29, 76, 53, 96, 36, 46, 54, 28, 51, 82, 53, 60, 77, 21, \
84, 53, 43, 104, 85, 50, 47, 39, 66, 78, 81, 94, 70, 49, 67, 61, 37, \
51, 91, 99, 58, 51, 49, 46, 68, 72, 40, 56, 63, 65, 41, 62, 47, 41, \
43, 30, 43, 67, 78, 80, 101, 61, 73, 70, 41, 82, 69, 45, 65, 38, 41, \
57, 82, 66}
```

As we can see, except of the very first 6 values (which is probably belongs to index file header), all numbers are in fact length of all text phrases (offset of the next phrase minus offset of the current phrase is in fact length of the current phrase).

It's very important to keep in mind that bit-endianess can be confused with incorrect array start. Indeed, from *od* output we see that each element started with two zeros. But when shifted by two bytes in either side, we can interpret this array as little-endian:

#### 9.4. FORTUNE PROGRAM INDEXING FILE

```
$ od -t x1 --address-radix=x --skip-bytes=0x32 fortunes.dat
000032 01 48 00 00 01 7c 00 00 01 ab 00 00 01 e6 00 00
000042 02 20 00 00 02 3b 00 00 02 7a 00 00 02 c5 00 00
000052 03 04 00 00 03 3d 00 00 03 68 00 00 03 a7 00 00
000062 03 e1 00 00 04 19 00 00 04 2d 00 00 04 7f 00 00
000072 04 ad 00 00 04 d5 00 00 05 05 00 00 05 3b 00 00
000082 05 64 00 00 05 82 00 00 05 ad 00 00 05 ce 00 00
000092 05 f7 00 00 06 1c 00 00 06 61 00 00 06 7a 00 00
0000a2 06 d1 00 00 07 0a 00 00 07 53 00 00 07 9a 00 00
0000b2 07 f8 00 00 08 27 00 00 08 59 00 00 08 8b 00 00
0000c2 08 a0 00 00 08 c4 00 00 08 e1 00 00 08 f9 00 00
0000d2 09 27 00 00 09 43 00 00 09 79 00 00 09 a3 00 00
0000e2 09 e3 00 00 0a 15 00 00 0a 4d 00 00 0a 5e 00 00
...
...
```

If we would interpret this array as little-endian, the first element is 0x4801, second is 0x7C01, etc. High 8-bit part of each of these 16-bit values are seems random to us, and the lowest 8-bit part is seems ascending.

But I'm sure that this is big-endian array, because the very last 32-bit element of the file is big-endian (00 00 5f c4 here):

```
$ od -t x1 --address-radix=x fortunes.dat
...
000660 00 00 59 0d 00 00 59 55 00 00 59 7d 00 00 59 b5
000670 00 00 59 f4 00 00 5a 35 00 00 5a 5e 00 00 5a 9c
000680 00 00 5a cb 00 00 5a f4 00 00 5b 1f 00 00 5b 3d
000690 00 00 5b 68 00 00 5b ab 00 00 5b f9 00 00 5c 49
0006a0 00 00 5c ae 00 00 5c eb 00 00 5d 34 00 00 5d 7a
0006b0 00 00 5d a3 00 00 5d f5 00 00 5e 3a 00 00 5e 67
0006c0 00 00 5e a8 00 00 5e ce 00 00 5e f7 00 00 5f 30
0006d0 00 00 5f 82 00 00 5f c4
0006d8
```

Perhaps, *fortune* program developer had big-endian computer or maybe it was ported from something like it.

OK, so the array is big-endian, and, judging by common sense, the very first phrase in the text file must be started at zeroth offset. So zero value should be present in the array somewhere at the very beginning. We've got couple of zero elements at the beginning. But the second is most appealing: 43 is going right after it and 43 is valid offset to valid English phrase in the text file.

The last array element is 0x5FC4, and there are no such byte at this offset in the text file. So the last array element is pointing behind the end of file. It's supposedly done because phrase length is calculated as difference between offset to the current phrase and offset to the next phrase. This can be faster than traversing phrase string for percent character. But this wouldn't work for the last element. So the *dummy* element is also added at the end of array.

So the first 6 32-bit integer values are supposedly some kind of header.

Oh, I forgot to count phrases in text file:

```
$ cat fortunes | grep % | wc -l
432
```

The number of phrases can be present in index, but may be not. In case of very simple index files, number of elements can be easily deduced from index file size. Anyway, there are 432 phrases in the text file. And we see something very familiar at the second element (value 431). I've checked other files (*literature.dat* and *riddles.dat* in Ubuntu Linux) and yes, the second 32-bit element is indeed number of phrases minus 1. Why *minus 1*? Perhaps, this is not number of phrases, but rather index of the last phrase (starting at zero)?

And there are some other elements in the header. In Mathematica, I'm loading each of three available files and I'm taking a look on the header:

## 9.4. FORTUNE PROGRAM INDEXING FILE

```
In[14]:= input = BinaryReadList["c:/tmp1/fortunes.dat", "UnsignedInteger32",
    ByteOrdering → 1];
]

In[18]:= BaseForm[Take[input, {1, 6}], 16]
Out[18]/BaseForm=
{216, 1af16, bb16, f16, 016, 2500000016}}
```

```
In[19]:= input = BinaryReadList["c:/tmp1/literature.dat", "UnsignedInteger32",
    ByteOrdering → 1];
]

In[20]:= BaseForm[Take[input, {1, 6}], 16]
Out[20]/BaseForm=
{216, 10616, 98316, 1a16, 016, 2500000016}}
```

```
In[21]:= input = BinaryReadList["c:/tmp1/riddles.dat", "UnsignedInteger32", ByteOrdering → 1];
]

In[22]:= BaseForm[Take[input, {1, 6}], 16]
Out[22]/BaseForm=
{216, 8016, 7f216, 2416, 016, 2500000016}}
```

I have no idea what other values mean, except the size of index file. Some fields are the same for all files, some are not. From my own experience, there could be:

- file signature;
- file version;
- checksum;
- some other flags;
- maybe even text language identifier;
- text file timestamp, so *fortune* program will regenerate index file if user added some new phrase(s) to it.

For example, Oracle .SYM files ([9.5 on the next page](#)) which contain symbols table for DLL files, also contain timestamp of corresponding DLL file, so to be sure it is still valid. But there are still possibility that the index file is regenerated if modification time of index file is older than text file. On the other hand, text file and index file timestamps can gone out of sync after archiving/unarchiving/installing/deploying/etc.

But there are no timestamp, in my opinion. The most compact way of representing date and time is UNIX time value, which is big 32-bit number. We don't see any of such here. Other ways are even less compact.

So here is algorithm, how *fortune* supposedly works:

- take number of last phrase from the second element;
- generate random number in range of 0..number\_of\_last\_phrase;
- find corresponding element in array of offsets, take also following offset;
- output to *stdout* all characters from the text file starting at current offset until the next offset minus 2 (so to ignore terminating percent sign and character of the following phrase).

### 9.4.1 Hacking

Let's try to check some of our assumptions. I will create this text file under the path and name */usr/share/games/fortunes/fortunes*:

```
Phrase one.
%
Phrase two.
%
```

## 9.5. ORACLE RDBMS: .SYM-FILES

Then this fortunes.dat file. I take header from the original fortunes.dat, I changed second field (count of all phrases) to zero and I left two elements in the array: 0 and 0x1c, because the whole length of the text fortunes file is 28 (0x1c) bytes:

```
$ od -t x1 --address-radix=x fortunes.dat
000000 00 00 00 02 00 00 00 00 00 00 bb 00 00 00 00 0f
000010 00 00 00 00 25 00 00 00 00 00 00 00 00 00 00 1c
```

Now I run it:

```
$ /usr/games/fortune
fortune: no fortune found
```

Something wrong. Let's change the second field to 1:

```
$ od -t x1 --address-radix=x fortunes.dat
000000 00 00 00 02 00 00 00 01 00 00 00 bb 00 00 00 00 0f
000010 00 00 00 00 25 00 00 00 00 00 00 00 00 00 00 1c
```

Now it works. It's always shows only the first phrase:

```
$ /usr/games/fortune
Phrase one.
```

Hmmm. Let's leave only one element in array (0) without terminating one:

```
$ od -t x1 --address-radix=x fortunes.dat
000000 00 00 00 02 00 00 00 01 00 00 00 bb 00 00 00 00 0f
000010 00 00 00 00 25 00 00 00 00 00 00 00 00 00 00 00
00001c
```

Fortune program always shows only first phrase.

From this experiment we got to know that percent sign is parsed inside of fortune file and the size is not calculated as I deduced before, perhaps, even terminal array element is not used. However, it still can be used. And probably it was used in past?

### 9.4.2 The files

For the sake of demonstration, I still didn't take a look in *fortune* source code. If you want to try to understand meaning of other values in index file header, you may try to achieve it without looking into source code as well. Files I took from Ubuntu Linux 14.04 are here: <http://beginners.re/examples/fortune/>, hacked files are also here.

Oh, and I took the files from x64 version of Ubuntu, but array elements are still has size of 32 bit. It is because *fortune* text files are probably never exceeds 4GiB!<sup>12</sup> size. But if it will, all elements must have size of 64 bit so to be able to store offset to the text file larger than 4GB.

For impatient readers, the source code of *fortune* is [here](#).

## 9.5 Oracle RDBMS: .SYM-files

When an Oracle RDBMS process experiences some kind of crash, it writes a lot of information into log files, including stack trace, like this:

```
----- Call Stack Trace -----
calling          call   entry           argument values in hex
location        type   point           (? means dubious value)
-----
_kqvrow()          00000000
_opifch2()+2729    CALLptr 00000000      23D4B914 E47F264 1F19AE2
_kpoal8()+2832    CALLrel _opifch2()     EB1C8A8 1
_opiopr()+1248    CALLreg 00000000      89 5 EB1CC74
                                         5E 1C EB1F0A0
```

<sup>12</sup>GiB!

## 9.5. ORACLE RDBMS: .SYM-FILES

_ttcpip() + 1051	CALLreg	00000000	5E 1C EB1F0A0 0
_opitsk() + 1404	CALL???	00000000	C96C040 5E EB1F0A0 0 EB1ED30
			EB1F1CC 53E52E 0 EB1F1F8
_opiino() + 980	CALLrel	_opitsk()	0 0
_opiodr() + 1248	CALLreg	00000000	3C 4 EB1FBF4
_opidrv() + 1201	CALLrel	_opiodr()	3C 4 EB1FBF4 0
_sou2o() + 55	CALLrel	_opidrv()	3C 4 EB1FBF4
_opimai_real() + 124	CALLrel	_sou2o()	EB1FC04 3C 4 EB1FBF4
_opimai() + 125	CALLrel	_opimai_real()	2 EB1FC2C
_OracleThreadStart@	CALLrel	_opimai()	2 EB1FF6C 7C88A7F4 EB1FC34 0
4() + 830			EB1FD04
77E6481C	CALLreg	00000000	E41FF9C 0 0 E41FF9C 0 EB1FFC4
00000000	CALL???	00000000	

But of course, Oracle RDBMS's executables must have some kind of debug information or map files with symbol information included or something like that.

Windows NT Oracle RDBMS has symbol information in files with .SYM extension, but the format is proprietary. (Plain text files are good, but needs additional parsing, hence offer slower access.)

Let's see if we can understand its format.

We will pick the shortest `orawtc8.sym` file that comes with the `orawtc8.dll` file in Oracle 8.1.7 <sup>13</sup>.

---

<sup>13</sup>We can chose an ancient Oracle RDBMS version intentionally due to the smaller size of its modules

## 9.5. ORACLE RDBMS: .SYM-FILES

Here is the file opened in Hiew:

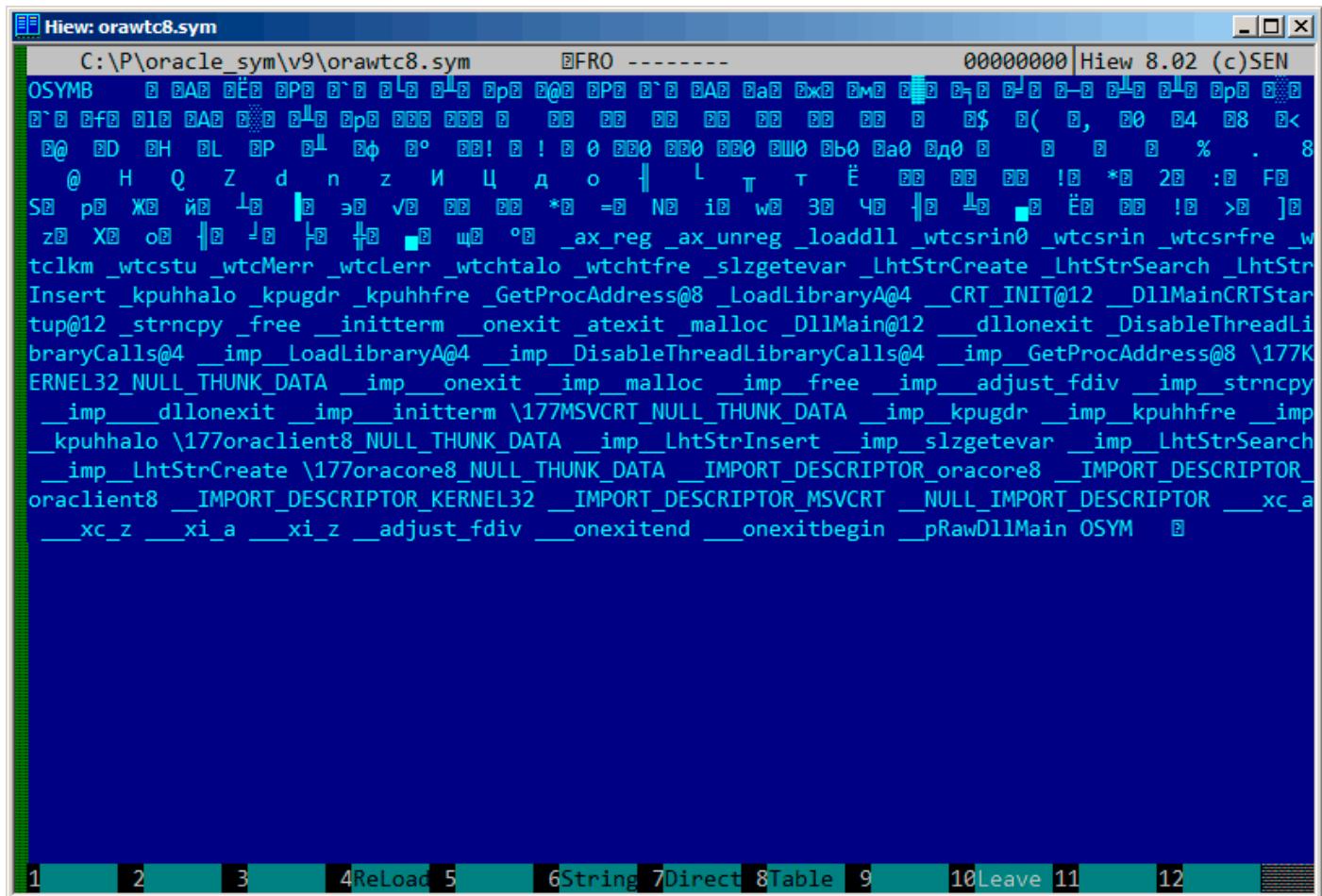


Figure 9.20: The whole file in Hiew

By comparing the file with other .SYM files, we can quickly see that `OSYM` is always header (and footer), so this is maybe the file's signature.

We also see that basically, the file format is: `OSYM` + some binary data + zero delimited text strings + `OSYM`. The strings are, obviously, function and global variable names.

## 9.5. ORACLE RDBMS: .SYM-FILES

We will mark the OSYM signatures and strings here:

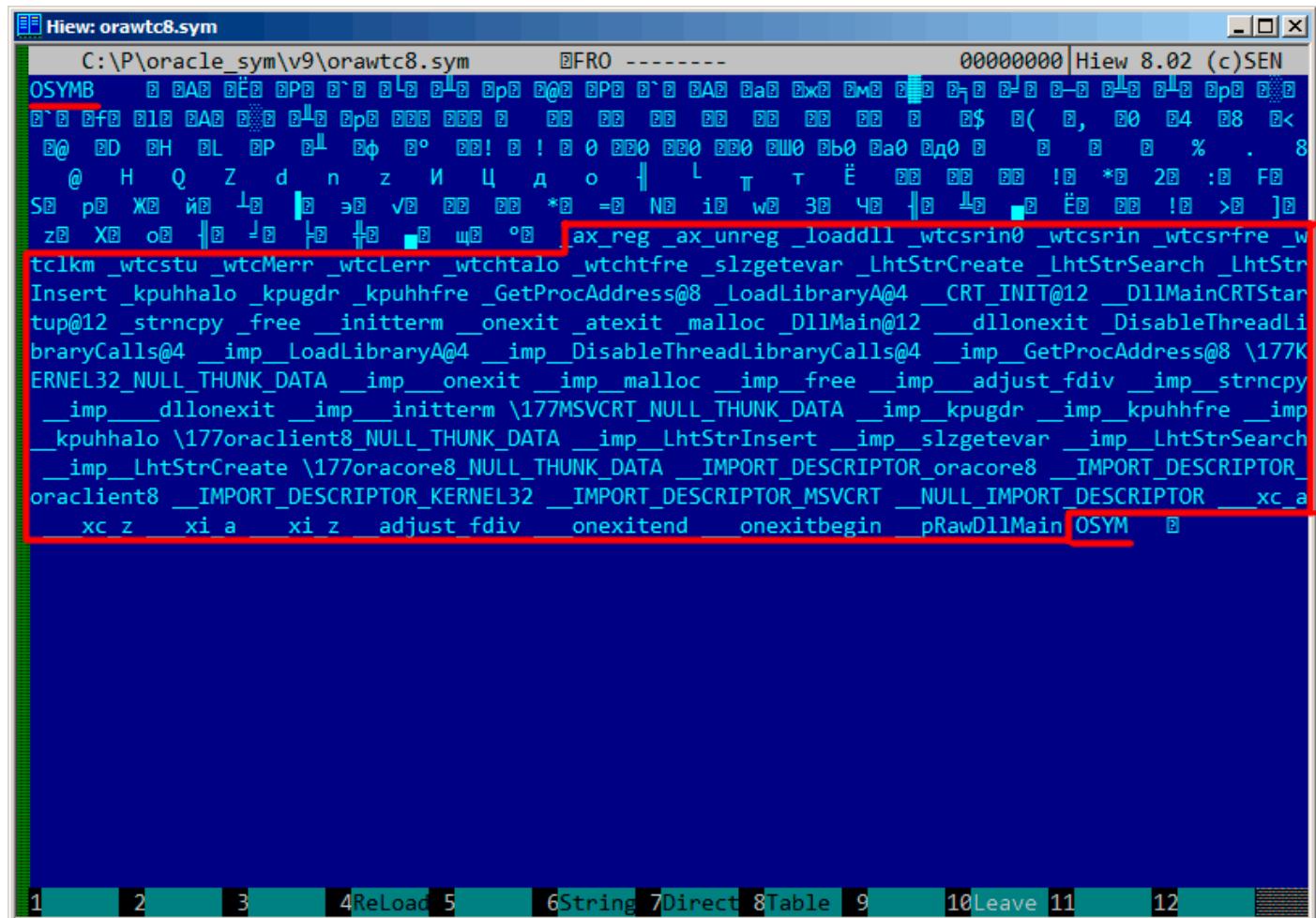


Figure 9.21: OSYM signature and text strings

Well, let's see. In Hiew, we will mark the whole strings block (except the trailing OSYM signatures) and put it into a separate file. Then we run UNIX *strings* and *wc* utilities to count the text strings:

```
strings strings_block | wc -l  
66
```

So there are 66 text strings. Please note that number.

We can say, in general, as a rule, the number of *anything* is often stored separately in binary files.

It's indeed so, we can find the 66 value (0x42) at the file's start, right after the OSYM signature:

```
$ hexdump -C orawtc8.sym
00000000  4f 53 59 4d 42 00 00 00  00 10 00 10 80 10 00 10  |OSYMB.....
00000010  f0 10 00 10 50 11 00 10  60 11 00 10 c0 11 00 10  |.....P.....
00000020  d0 11 00 10 70 13 00 10  40 15 00 10 50 15 00 10  |.....p...@..P...
00000030  60 15 00 10 80 15 00 10  a0 15 00 10 a6 15 00 10  |`.....'.
....
```

Of course, 0x42 here is not a byte, but most likely a 32-bit value packed as little-endian, hence we see 0x42 and then at least 3 zero bytes.

Why do we believe it's 32-bit? Because, Oracle RDBMS's symbol files may be pretty big.

The oracle.sym file for the main oracle.exe (version 10.2.0.4) executable contains **0x3A38E** (238478) symbols. A 16-bit value isn't enough here.

We can check other .SYM files like this and it proves our guess: the value after the 32-bit OSYM signature always reflects the number of text strings in the file.

## 9.5. ORACLE RDBMS: .SYM-FILES

It's a general feature of almost all binary files: a header with a signature plus some other information about the file.

Now let's investigate closer what this binary block is.

Using Hiew again, we put the block starting at address 8 (i.e., after the 32-bit *count* value) ending at the strings block, into a separate binary file.

## 9.5. ORACLE RDBMS: .SYM-FILES

Let's see the binary block in Hiew:

	FRO	-----	00000000
00000000:	00 10 00 10-80 10 00 10-F0 10 00 10-50 11 00 10	Б БАБ БЁБ БРБ Б	
00000010:	60 11 00 10-C0 11 00 10-D0 11 00 10-70 13 00 10	Б БЛБ БЛБ БРБ Б	
00000020:	40 15 00 10-50 15 00 10-60 15 00 10-80 15 00 10	@Б БРБ Б`Б БАБ Б	
00000030:	A0 15 00 10-A6 15 00 10-AC 15 00 10-B2 15 00 10	аБ БжБ БМБ Б`Б Б	
00000040:	B8 15 00 10-BE 15 00 10-C4 15 00 10-CA 15 00 10	Б Б`Б Б-Б Б-Б Б	
00000050:	D0 15 00 10-E0 15 00 10-B0 16 00 10-60 17 00 10	Б БрБ БББ Б`Б Б	
00000060:	66 17 00 10-6C 17 00 10-80 17 00 10-B0 17 00 10	fБ Б1Б БАБ БББ Б	
00000070:	D0 17 00 10-E0 17 00 10-10 18 00 10-16 18 00 10	Б БрБ БББ БББ Б	
00000080:	00 20 00 10-04 20 00 10-08 20 00 10-0C 20 00 10	ББ ББ ББ ББ Б	
00000090:	10 20 00 10-14 20 00 10-18 20 00 10-1C 20 00 10	Б ББ ББ ББ ББ Б	
000000A0:	20 20 00 10-24 20 00 10-28 20 00 10-2C 20 00 10	Б\$ Б( Б, Б	
000000B0:	30 20 00 10-34 20 00 10-38 20 00 10-3C 20 00 10	0 Б4 Б8 Б< Б	
000000C0:	40 20 00 10-44 20 00 10-48 20 00 10-4C 20 00 10	@ БD БH БL Б	
000000D0:	50 20 00 10-D0 20 00 10-E4 20 00 10-F8 20 00 10	Р БЛ Бф Б° Б	
000000E0:	0C 21 00 10-20 21 00 10-00 30 00 10-04 30 00 10	Б! Б ! Б 0 ББ0 Б	
000000F0:	08 30 00 10-0C 30 00 10-98 30 00 10-9C 30 00 10	Б0 ББ0 БШ0 ББ0 Б	
00000100:	A0 30 00 10-A4 30 00 10-00 00 00 00-08 00 00 00	а0 Бд0 Б Б	
00000110:	12 00 00 00-1B 00 00 00-25 00 00 00-2E 00 00 00	Б Б % .	
00000120:	38 00 00 00-40 00 00 00-48 00 00 00-51 00 00 00	8 @ Н Q	
00000130:	5A 00 00 00-64 00 00 00-6E 00 00 00-7A 00 00 00	Z d n z	
00000140:	88 00 00 00-96 00 00 00-A4 00 00 00-AE 00 00 00	И Ц Д о	
00000150:	B6 00 00 00-C0 00 00 00-D2 00 00 00-E2 00 00 00	Б Л Т	
00000160:	F0 00 00 00-07 01 00 00-10 01 00 00-16 01 00 00	Ё ББ ББ ББ	
00000170:	21 01 00 00-2A 01 00 00-32 01 00 00-3A 01 00 00	!Б *Б 2Б :Б	
00000180:	46 01 00 00-53 01 00 00-70 01 00 00-86 01 00 00	FБ SБ pБ ЖБ	
00000190:	A9 01 00 00-C1 01 00 00-DE 01 00 00-ED 01 00 00	йБ ЛБ Б эБ	
000001A0:	FB 01 00 00-07 02 00 00-1B 02 00 00-2A 02 00 00	√Б ББ ББ *Б	

Figure 9.22: Binary block

There is a clear pattern in it.

## 9.5. ORACLE RDBMS: .SYM-FILES

We will add red lines to divide the block:

C:\P\oracle_sym\v9\asd2											EFRO			00000000		
00000000:	00	10	00	10	80	10	00	10	F0	10	00	10	50	11	00	10
00000010:	60	11	00	10	C0	11	00	10	D0	11	00	10	70	13	00	10
00000020:	40	15	00	10	50	15	00	10	60	15	00	10	80	15	00	10
00000030:	A0	15	00	10	A6	15	00	10	AC	15	00	10	B2	15	00	10
00000040:	B8	15	00	10	BE	15	00	10	C4	15	00	10	CA	15	00	10
00000050:	D0	15	00	10	E0	15	00	10	B0	16	00	10	60	17	00	10
00000060:	66	17	00	10	6C	17	00	10	80	17	00	10	B0	17	00	10
00000070:	D0	17	00	10	E0	17	00	10	10	18	00	10	16	18	00	10
00000080:	00	20	00	10	04	20	00	10	08	20	00	10	0C	20	00	10
00000090:	10	20	00	10	14	20	00	10	18	20	00	10	1C	20	00	10
000000A0:	20	20	00	10	24	20	00	10	28	20	00	10	2C	20	00	10
000000B0:	30	20	00	10	34	20	00	10	38	20	00	10	3C	20	00	10
000000C0:	40	20	00	10	44	20	00	10	48	20	00	10	4C	20	00	10
000000D0:	50	20	00	10	D0	20	00	10	E4	20	00	10	F8	20	00	10
000000E0:	0C	21	00	10	20	21	00	10	00	30	00	10	04	30	00	10
000000F0:	08	30	00	10	0C	30	00	10	98	30	00	10	9C	30	00	10
00000100:	A0	30	00	10	A4	30	00	10	00	00	00	00	08	00	00	00
00000110:	12	00	00	00	1B	00	00	00	25	00	00	00	2E	00	00	00
00000120:	38	00	00	00	40	00	00	00	48	00	00	00	51	00	00	00
00000130:	5A	00	00	00	64	00	00	00	6E	00	00	00	7A	00	00	00
00000140:	88	00	00	00	96	00	00	00	A4	00	00	00	AE	00	00	00
00000150:	B6	00	00	00	C0	00	00	00	D2	00	00	00	E2	00	00	00
00000160:	F0	00	00	00	07	01	00	00	10	01	00	00	16	01	00	00
00000170:	21	01	00	00	2A	01	00	00	32	01	00	00	3A	01	00	00
00000180:	46	01	00	00	53	01	00	00	70	01	00	00	86	01	00	00
00000190:	A9	01	00	00	C1	01	00	00	DE	01	00	00	ED	01	00	00
000001A0:	FB	01	00	00	07	02	00	00	1B	02	00	00	2A	02	00	00
000001B0:	3D	02	00	00	4E	02	00	00	69	02	00	00	77	02	00	00

Figure 9.23: Binary block patterns

Hiew, like almost any other hexadecimal editor, shows 16 bytes per line. So the pattern is clearly visible: there are 4 32-bit values per line.

The pattern is visually visible because some values here (till address `0x104`) are always in `0x1000xxxx` form, started with `0x10` and zero bytes.

Other values (starting at 0x108) are in 0x0000xxxx form, so always started with two zero bytes.

Let's dump the block as an array of 32-bit values:

Listing 9.9: first column is address

```
$ od -v -t x4 binary_block
00000000 10001000 10001080 100010f0 10001150
00000020 10001160 100011c0 100011d0 10001370
00000040 10001540 10001550 10001560 10001580
00000060 100015a0 100015a6 100015ac 100015b2
00000100 100015b8 100015be 100015c4 100015ca
00000120 100015d0 100015e0 100016b0 10001760
00000140 10001766 1000176c 10001780 100017b0
00000160 100017d0 100017e0 10001810 10001816
00000200 10002000 10002004 10002008 1000200c
00000220 10002010 10002014 10002018 1000201c
```

## 9.5. ORACLE RDBMS: .SYM-FILES

```
0000240 10002020 10002024 10002028 1000202c
0000260 10002030 10002034 10002038 1000203c
0000300 10002040 10002044 10002048 1000204c
0000320 10002050 100020d0 100020e4 100020f8
0000340 1000210c 10002120 10003000 10003004
0000360 10003008 1000300c 10003098 1000309c
0000400 100030a0 100030a4 00000000 00000008
0000420 00000012 0000001b 00000025 0000002e
0000440 00000038 00000040 00000048 00000051
0000460 0000005a 00000064 0000006e 0000007a
0000500 00000088 00000096 000000a4 000000ae
0000520 000000b6 000000c0 000000d2 000000e2
0000540 000000f0 00000107 00000110 00000116
0000560 00000121 0000012a 00000132 0000013a
0000600 00000146 00000153 00000170 00000186
0000620 000001a9 000001c1 000001de 000001ed
0000640 000001fb 00000207 0000021b 0000022a
0000660 0000023d 0000024e 00000269 00000277
0000700 00000287 00000297 000002b6 000002ca
0000720 000002dc 000002f0 00000304 00000321
0000740 0000033e 0000035d 0000037a 00000395
0000760 000003ae 000003b6 000003be 000003c6
0001000 000003ce 000003dc 000003e9 000003f8
0001020
```

There are 132 values, that's  $66 \times 2$ . Probably, there are two 32-bit values for each symbol, but maybe there are two arrays? Let's see.

Values starting with `0x1000` may be addresses.

This is a .SYM file for a DLL after all, and the default base address of win32 DLLs is `0x10000000`, and the code usually starts at `0x10001000`.

When we open the orawtc8.dll file in [IDA](#), the base address is different, but nevertheless, the first function is:

```
.text:60351000 sub_60351000    proc near
.text:60351000
.text:60351000 arg_0     = dword ptr  8
.text:60351000 arg_4     = dword ptr  0Ch
.text:60351000 arg_8     = dword ptr  10h
.text:60351000
.text:60351000         push   ebp
.text:60351001         mov    ebp, esp
.text:60351003         mov    eax, dword_60353014
.text:60351008         cmp    eax, 0FFFFFFFh
.text:6035100B         jnz    short loc_6035104F
.text:6035100D         mov    ecx, hModule
.text:60351013         xor    eax, eax
.text:60351015         cmp    ecx, 0FFFFFFFh
.text:60351018         mov    dword_60353014, eax
.text:6035101D         jnz    short loc_60351031
.text:6035101F         call   sub_603510F0
.text:60351024         mov    ecx, eax
.text:60351026         mov    eax, dword_60353014
.text:6035102B         mov    hModule, ecx
.text:60351031
.text:60351031 loc_60351031: ; CODE XREF: sub_60351000+1D
.text:60351031         test   ecx, ecx
.text:60351033         jbe    short loc_6035104F
.text:60351035         push   offset ProcName ; "ax_reg"
.text:6035103A         push   ecx           ; hModule
.text:6035103B         call   ds:GetProcAddress
...
```

Wow, "ax\_reg" string sounds familiar.

It's indeed the first string in the strings block! So the name of this function seems to be "ax\_reg".

The second function is:

## 9.5. ORACLE RDBMS: .SYM-FILES

```
.text:60351080 sub_60351080    proc near
.text:60351080
.text:60351080 arg_0      = dword ptr  8
.text:60351080 arg_4      = dword ptr  0Ch
.text:60351080
.text:60351080          push   ebp
.text:60351081          mov    ebp, esp
.text:60351083          mov    eax, dword_60353018
.text:60351088          cmp    eax, 0FFFFFFFh
.text:6035108B          jnz    short loc_603510CF
.text:6035108D          mov    ecx, hModule
.text:60351093          xor    eax, eax
.text:60351095          cmp    ecx, 0FFFFFFFh
.text:60351098          mov    dword_60353018, eax
.text:6035109D          jnz    short loc_603510B1
.text:6035109F          call   sub_603510F0
.text:603510A4          mov    ecx, eax
.text:603510A6          mov    eax, dword_60353018
.text:603510AB          mov    hModule, ecx
.text:603510B1
.text:603510B1 loc_603510B1: ; CODE XREF: sub_60351080+1D
.text:603510B1          test   ecx, ecx
.text:603510B3          jbe    short loc_603510CF
.text:603510B5          push   offset aAx_unreg ; "ax_unreg"
.text:603510BA          push   ecx           ; hModule
.text:603510BB          call   ds:GetProcAddress
...
...
```

The “ax\_unreg” string is also the second string in the strings block!

The starting address of the second function is `0x60351080`, and the second value in the binary block is `10001080`. So this is the address, but for a DLL with the default base address.

We can quickly check and be sure that the first 66 values in the array (i.e., the first half of the array) are just function addresses in the DLL, including some labels, etc. Well, what's the other part of array then? The other 66 values that start with `0x0000`? These seem to be in range `[0...0x3F8]`. And they do not look like bitfields: the series of numbers is increasing.

The last hexadecimal digit seems to be random, so, it's unlikely the address of something (it would be divisible by 4 or maybe 8 or `0x10` otherwise).

Let's ask ourselves: what else Oracle RDBMS's developers would save here, in this file?

Quick wild guess: it could be the address of the text string (function name).

It can be quickly checked, and yes, each number is just the position of the first character in the strings block.

This is it! All done.

We will write an utility to convert these .SYM files into [IDA](#) script, so we can load the .idc script and it sets the function names:

```
#include <stdio.h>
#include <stdint.h>
#include <io.h>
#include <assert.h>
#include <malloc.h>
#include <fcntl.h>
#include <string.h>

int main (int argc, char *argv[])
{
    uint32_t sig, cnt, offset;
    uint32_t *d1, *d2;
    int     h, i, remain, file_len;
    char   *d3;
    uint32_t array_size_in_bytes;

    assert (argc[1]); // file name
    assert (argc[2]); // additional offset (if needed)
```

```

// additional offset
assert (sscanf (argv[2], "%X", &offset)==1);

// get file length
assert ((h=open (argv[1], _O_RDONLY | _O_BINARY, 0))!=-1);
assert ((file_len=lseek (h, 0, SEEK_END))!=-1);
assert (lseek (h, 0, SEEK_SET)!=-1);

// read signature
assert (read (h, &sig, 4)==4);
// read count
assert (read (h, &cnt, 4)==4);

assert (sig==0x4D59534F); // OSYM

// skip timestamp (for 11g)
//_lseek (h, 4, 1);

array_size_in_bytes=cnt*sizeof(uint32_t);

// load symbol addresses array
d1=(uint32_t*)malloc (array_size_in_bytes);
assert (d1);
assert (read (h, d1, array_size_in_bytes)==array_size_in_bytes);

// load string offsets array
d2=(uint32_t*)malloc (array_size_in_bytes);
assert (d2);
assert (read (h, d2, array_size_in_bytes)==array_size_in_bytes);

// calculate strings block size
remain=file_len-(8+4)-(cnt*8);

// load strings block
assert (d3=(char*)malloc (remain));
assert (read (h, d3, remain)==remain);

printf ("#include <idc.idc>\n\n");
printf ("static main() {\n");

for (i=0; i<cnt; i++)
    printf ("\tMakeName(0x%08X, \"%s\");\n", offset + d1[i], &d3[d2[i]]);

printf ("}\n");

close (h);
free (d1); free (d2); free (d3);
};

}

```

Here is an example of its work:

```

#include <idc.idc>

static main() {
    MakeName(0x60351000, "_ax_reg");
    MakeName(0x60351080, "_ax_unreg");
    MakeName(0x603510F0, "_loaddll");
    MakeName(0x60351150, "_wtcsrin0");
    MakeName(0x60351160, "_wtcsrin");
    MakeName(0x603511C0, "_wtcsrfre");
    MakeName(0x603511D0, "_wtclkm");
    MakeName(0x60351370, "_wtcstu");
...
}

```

The example files were used in this example are here: [beginners.re](#).

## 9.6. ORACLE RDBMS: .MSB-FILES

Oh, let's also try Oracle RDBMS for win64. There has to be 64-bit addresses instead, right?

The 8-byte pattern is visible even easier here:

	FRO	SYMM64	m@
00000000:	4F 53 59 4D-41 4D 36 34-BD 6D 05 00-00 00 00 00 00	OSYMM64	m@
00000010:	CD 21 2A 47-00 00 00 00-00 00 00 00-00 00 00 00 00	=!*G	
00000020:	00 00 00 00-00 00 00 00-00 00 40 00-00 00 00 00 00		@
00000030:	00 10 40 00-00 00 00 00-6C 10 40 00-00 00 00 00 00	Б@	1Б@
00000040:	04 11 40 00-00 00 00 00-80 13 40 00-00 00 00 00 00	ББ@	AB@
00000050:	E3 13 40 00-00 00 00 00-01 14 40 00-00 00 00 00 00	yБ@	ББ@
00000060:	1F 14 40 00-00 00 00 00-3E 14 40 00-00 00 00 00 00	ББ@	>Б@
00000070:	54 14 40 00-00 00 00 00-1E 18 40 00-00 00 00 00 00	ТБ@	ББ@
00000080:	97 1B 40 00-00 00 00 00-C1 1B 40 00-00 00 00 00 00	ЧБ@	ЛБ@
00000090:	0A 1C 40 00-00 00 00 00-4C 1C 40 00-00 00 00 00 00	ББ@	ЛБ@
000000A0:	7A 1C 40 00-00 00 00 00-98 1C 40 00-00 00 00 00 00	zБ@	ШБ@
000000B0:	E7 25 40 00-00 00 00 00-11 26 40 00-00 00 00 00 00	Ч%@	Б&%@
000000C0:	80 26 40 00-00 00 00 00-C4 26 40 00-00 00 00 00 00	A&%@	-&%@
000000D0:	F4 26 40 00-00 00 00 00-24 27 40 00-00 00 00 00 00	Ї&%@	\$'@
000000E0:	50 27 40 00-00 00 00 00-78 27 40 00-00 00 00 00 00	P'@	x'@
000000F0:	A0 27 40 00-00 00 00 00-4E 28 40 00-00 00 00 00 00	a'@	N(@
00000100:	26 29 40 00-00 00 00 00-B4 2C 40 00-00 00 00 00 00	&)@	Л,@
00000110:	66 2D 40 00-00 00 00 00-A6 2D 40 00-00 00 00 00 00	f-@	ж-@
00000120:	30 2E 40 00-00 00 00 00-BA 2E 40 00-00 00 00 00 00	0.%@	.%@
00000130:	F2 30 40 00-00 00 00 00-84 31 40 00-00 00 00 00 00	€0@	Д1@
00000140:	F0 31 40 00-00 00 00 00-5E 32 40 00-00 00 00 00 00	Ё1@	^2@
00000150:	CC 32 40 00-00 00 00 00-3A 33 40 00-00 00 00 00 00	¶2@	:3@
00000160:	A8 33 40 00-00 00 00 00-16 34 40 00-00 00 00 00 00	и3@	Б4@
00000170:	84 34 40 00-00 00 00 00-F2 34 40 00-00 00 00 00 00	Д4@	€4@
00000180:	60 35 40 00-00 00 00 00-CC 35 40 00-00 00 00 00 00	~5@	Л5@
00000190:	3A 36 40 00-00 00 00 00-A8 36 40 00-00 00 00 00 00	:6@	и6@
000001A0:	16 37 40 00-00 00 00 00-84 37 40 00-00 00 00 00 00	Б7@	Д7@

Figure 9.24: .SYM-file example from Oracle RDBMS for win64

So yes, all tables now have 64-bit elements, even string offsets!

The signature is now `OSYMM64`, to distinguish the target platform, apparently.

This is it!

Here is also library which has functions to access Oracle RDBMS.SYM-files: [GitHub](#).

## 9.6 Oracle RDBMS: .MSB-files

When working toward the solution of a problem, it always helps if you know the answer.

Murphy's Laws, Rule of Accuracy

This is a binary file that contains error messages with their corresponding numbers. Let's try to understand its format and find a way to unpack it.

## 9.6. ORACLE RDBMS: .MSB-FILES

There are Oracle RDBMS error message files in text form, so we can compare the text and packed binary files <sup>14</sup>.

This is the beginning of the ORAUS.MSG text file with some irrelevant comments stripped:

Listing 9.10: Beginning of ORAUS.MSG file without comments

```
00000, 00000, "normal, successful completion"
00001, 00000, "unique constraint (%s.%s) violated"
00017, 00000, "session requested to set trace event"
00018, 00000, "maximum number of sessions exceeded"
00019, 00000, "maximum number of session licenses exceeded"
00020, 00000, "maximum number of processes (%s) exceeded"
00021, 00000, "session attached to some other process; cannot switch session"
00022, 00000, "invalid session ID; access denied"
00023, 00000, "session references process private memory; cannot detach session"
00024, 00000, "logins from more than one process not allowed in single-process mode"
00025, 00000, "failed to allocate %s"
00026, 00000, "missing or invalid session ID"
00027, 00000, "cannot kill current session"
00028, 00000, "your session has been killed"
00029, 00000, "session is not a user session"
00030, 00000, "User session ID does not exist."
00031, 00000, "session marked for kill"
...
...
```

The first number is the error code. The second is perhaps maybe some additional flags.

<sup>14</sup>Open-source text files don't exist in Oracle RDBMS for every .MSB file, so that's why we will work on their file format

## 9.6. ORACLE RDBMS: .MSB-FILES

Now let's open the ORAUS.MSB binary file and find these text strings. And there are:

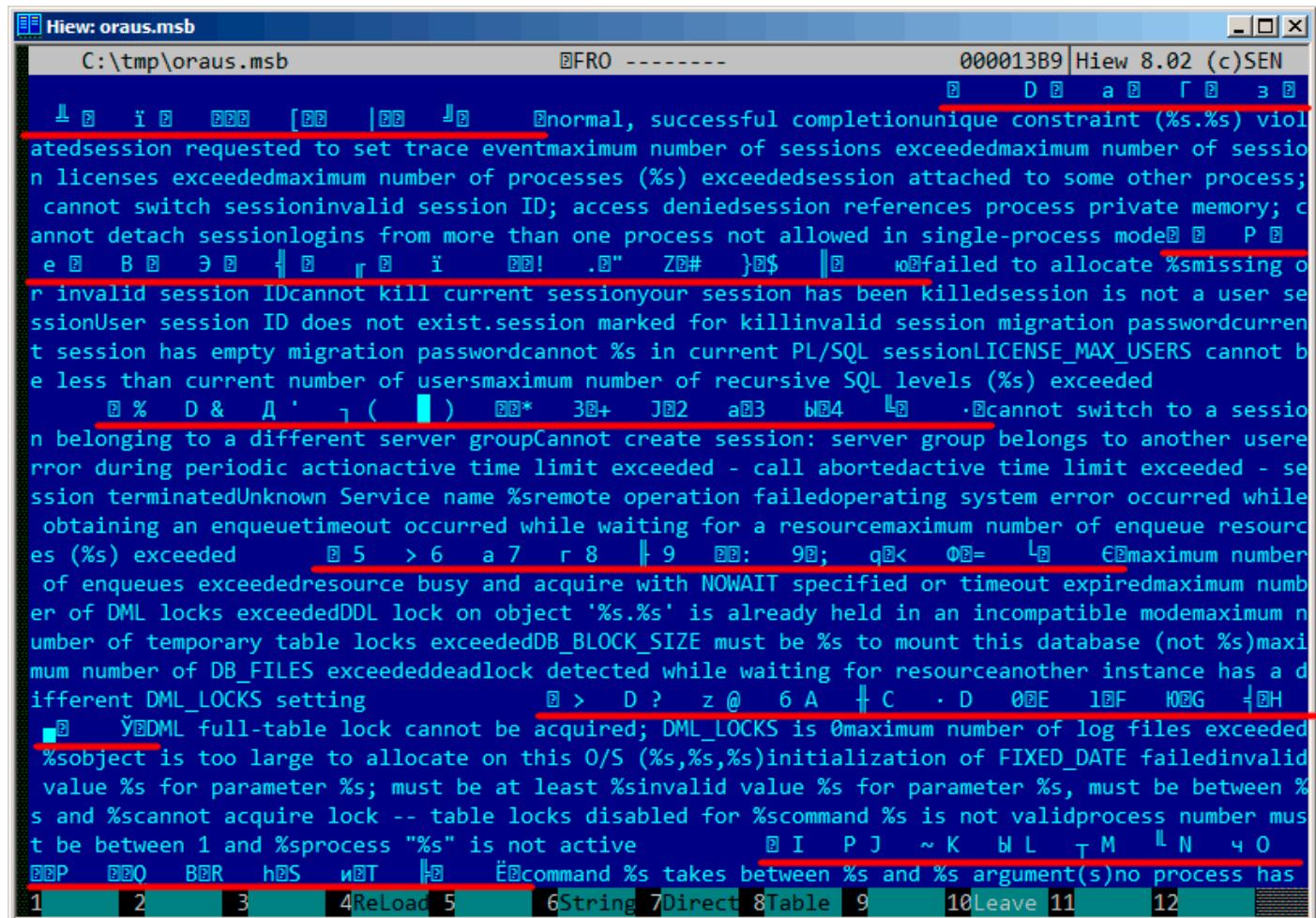


Figure 9.25: Hiew: first block

We see the text strings (including those from the beginning of the ORAUS.MSG file) interleaved with some binary values. By quick investigation, we can see that main part of the binary file is divided by blocks of size 0x200 (512) bytes.

## 9.6. ORACLE RDBMS: .MSB-FILES

Let's see the contents of the first block:

		EFRO	-----	00001400						
00001400:	0A 00 00 00-00 00 44 00-01 00 00 00-61 00 11 00	E D a	Г з І							
00001410:	00 00 83 00-12 00 00 00-A7 00 13 00-00 00 CA 00									
00001420:	14 00 00 00-F5 00 15 00-00 00 1E 01-16 00 00 00	ї	І Ї							
00001430:	5B 01 17 00-00 00 7C 01-18 00 00 00-BC 01 00 00	[І]	І							
00001440:	00 00 00 02-6E 6F 72 6D-61 6C 2C 20-73 75 63 63	Normal, succ	essful completio							
00001450:	65 73 73 66-75 6C 20 63-6F 6D 70 6C-65 74 69 6F	nique constrai	nt (%s.%s) viola							
00001460:	6E 75 6E 69-71 75 65 20-63 6F 6E 73-74 72 61 69	ted session reque	sted to set trac							
00001470:	6E 74 20 28-25 73 2E 25-73 29 20 76-69 6F 6C 61	ed maximum n	umber of sessi							
00001480:	74 65 64 73-65 73 73 69-6F 6E 20 72-65 71 75 65	on licenses exc	eeded maximum num							
00001490:	73 74 65 64-20 74 6F 20-73 65 74 20-74 72 61 63	ber of processes	er exceeded sessi							
000014A0:	65 20 65 76-65 6E 74 6D-61 78 69 6D-75 6D 20 6E	(%s) exceede	on attached to some other pro							
000014B0:	75 6D 62 65-72 20 6F 66-20 73 65 73-73 69 6F 6E	cess; cannot swit	cess; cannot switc							
000014C0:	73 20 65 78-63 65 65 64-65 64 6D 61-78 69 6D 75	ch session invalid	h session invalid							
000014D0:	6D 20 6E 75-6D 62 65 72-20 6F 66 20-73 65 73 73	id session ID; access denied	session references p							
000014E0:	69 6F 6E 20-6C 69 63 65-6E 73 65 73-20 65 78 63	cess denied session	rocess private memory; cannot de							
000014F0:	65 65 64 65-64 6D 61 78-69 6D 75 6D-20 6E 75 6D	ach session logi	tach session logi							
00001500:	62 65 72 20-6F 66 20 70-72 6F 63 65-73 73 65 73									
00001510:	20 28 25 73-29 20 65 78-63 65 65 64-65 64 73 65									
00001520:	73 73 69 6F-6E 20 61 74-74 61 63 68-65 64 20 74									
00001530:	6F 20 73 6F-6D 65 20 6F-74 68 65 72-20 70 72 6F									
00001540:	63 65 73 73-3B 20 63 61-6E 6E 6F 74-20 73 77 69									
00001550:	74 63 68 20-73 65 73 73-69 6F 6E 69-6E 76 61 6C									
00001560:	69 64 20 73-65 73 73 69-6F 6E 20 49-44 3B 20 61									
00001570:	63 63 65 73-73 20 64 65-6E 69 65 64-73 65 73 73									
00001580:	69 6F 6E 20-72 65 66 65-72 65 6E 63-65 73 20 70									
00001590:	72 6F 63 65-73 73 20 70-72 69 76 61-74 65 20 6D									
000015A0:	65 6D 6F 72-79 3B 20 63-61 6E 6E 6F-74 20 64 65									
000015B0:	74 61 63 68-20 73 65 73-73 69 6F 6E-6C 6F 67 69									
1Global	2FillBlk	3CryBlk	4Reload	5String	6Direct	7Table	8Leave	9	10	11

Figure 9.26: Hiew: first block

Here we see the texts of the first messages errors. What we also see is that there are no zero bytes between the error messages. This implies that these are not null-terminated C strings. As a consequence, the length of each error message must be encoded somehow. Let's also try to find the error numbers. The ORAUS.MSG file starts with these: 0, 1, 17 (0x11), 18 (0x12), 19 (0x13), 20 (0x14), 21 (0x15), 22 (0x16), 23 (0x17), 24 (0x18)... We will find these numbers at the beginning of the block and mark them with red lines. The period between error codes is 6 bytes.

This implies that there are probably 6 bytes of information allocated for each error message.

The first 16-bit value (0xA here or 10) means the number of messages in each block: this can be checked by investigating other blocks. Indeed: the error messages have arbitrary size. Some are longer, some are shorter. But block size is always fixed, hence, you never know how many text messages can be packed in each block.

As we already noted, since these are not null-terminated C strings, their size must be encoded somewhere. The size of the first string "normal, successful completion" is 29 (0x1D) bytes. The size of the second string "unique constraint (%s.%s) violated" is 34 (0x22) bytes. We can't find these values (0x1D or/and 0x22) in the block.

There is also another thing. Oracle RDBMS has to determine the position of the string it needs to load in the block, right? The first string "normal, successful completion" starts at position 0x1444 (if we count starting at the beginning of the file) or at 0x44 (from the block's start). The second string "unique constraint

## 9.6. ORACLE RDBMS: .MSB-FILES

(%s.%s) violated" starts at position 0x1461 (from the file's start) or at 0x61 (from the at the block's start). These numbers (0x44 and 0x61) are familiar somehow! We can clearly see them at the start of the block.

So, each 6-byte block is:

- 16-bit error number;
- 16-bit zero (maybe additional flags);
- 16-bit starting position of the text string within the current block.

We can quickly check the other values and be sure our guess is correct. And there is also the last "dummy" 6-byte block with an error number of zero and starting position beyond the last error message's last character. Probably that's how text message length is determined? We just enumerate 6-byte blocks to find the error number we need, then we get the text string's position, then we get the position of the text string by looking at the next 6-byte block! This way we determine the string's boundaries! This method allows to save some space by not saving the text string's size in the file!

It's not possible to say it saves a lot of space, but it's a clever trick.

## 9.6. ORACLE RDBMS: .MSB-FILES

Let's back to the header of .MSB-file:

		FRO	-----	00000000
00000000:	15 13 22 01-13 03 09 09-00 00 00 00-00 00 00 00		EE"EEEE	
00000010:	00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00			
00000020:	00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00			
00000030:	00 00 00 00-02 00 00 00-01 00 00 00-01 00 00 00		Э Э Э	
00000040:	08 00 00 00-84 07 00 00-00 00 00 00-9C EA 00 00		Э ДВ бъ	
00000050:	60 40 00 00-14 4D 0C 00-12 04 02 0D-13 00 00 00		^@ ЭМВ ЭФФ	
00000060:	00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00			
00000070:	00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00			
00000080:	00 00 00 00-00 00 00 00-00 00 0F 12-01 00 00 00		FFF	
00000090:	00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00			
000000A0:	00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00			
000000B0:	00 00 00 00-00 00 00 00-00 00 00 00-15 13 00 00		FF	
000000C0:	00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00			
000000D0:	00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00			
000000E0:	00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00			
000000F0:	03 00 00 00-01 00 00 00-01 00 00 00-01 00 00 00		Э Э Э Э	
00000100:	00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00			
00000110:	00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00			
00000120:	00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00			
00000130:	00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00			
00000140:	00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00			
00000150:	00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00			
00000160:	00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00			
00000170:	00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00			
00000180:	00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00			
00000190:	00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00			
000001A0:	00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00			
000001B0:	00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00			
1Global 2FilBlk 3CryBlk 4ReLoad 5	6String 7Direct 8Table 9	10Leave 11		

Figure 9.27: Hiew: file header

Now we can quickly find the number of blocks in the file (marked by red). We can checked other .MSB-files and we see that it's true for all of them.

There are a lot of other values, but we will not investigate them, since our job (an unpacking utility) is done.

If we have to write a .MSB file packer, we would probably have to understand the meaning of the other values.

## 9.6. ORACLE RDBMS: .MSB-FILES

There is also a table that came after the header which probably contains 16-bit values:

Hiew: oraus.msb												00000800
	FRO -----											00000800
00000800:	83	34	8F	34	9B	34	AA	34	BE	34	C7	34
00000810:	E3	34	EB	34	24	35	2C	35	32	35	39	35
00000820:	4E	35	56	35	5D	35	84	35	8A	35	8F	35
00000830:	C6	35	CE	35	D8	35	E4	35	04	36	0F	36
00000840:	2C	36	52	36	5B	36	94	36	A2	36	B4	36
00000850:	CE	36	D7	36	DF	36	E7	36	ED	36	F5	36
00000860:	0C	37	13	37	1A	37	21	37	29	37	31	37
00000870:	4E	37	55	37	5E	37	68	37	6E	37	75	37
00000880:	A2	37	AF	37	B7	37	BD	37	C5	37	CC	37
00000890:	E0	37	E8	37	F2	37	F9	37	45	38	73	38
000008A0:	B1	38	B7	38	BC	38	C6	38	0A	39	0F	39
000008B0:	23	39	29	39	2F	39	35	39	3E	39	46	39
000008C0:	AE	39	9A	3A	A5	3A	B1	3A	BC	3A	C7	3A
000008D0:	E5	3A	F4	3A	00	3B	0B	3B	15	3B	2E	3B
000008E0:	51	3B	5E	3B	68	3B	74	3B	84	3B	8E	3B
000008F0:	65	3C	6E	3C	77	3C	8F	3C	96	3C	C0	3C
00000900:	F5	3C	53	3D	88	3E	90	3E	96	3E	9E	3E
00000910:	BA	3E	C4	3E	CF	3E	D9	3E	E1	3E	EA	3E
00000920:	07	3F	12	3F	1B	3F	23	3F	2B	3F	34	3F
00000930:	4D	3F	56	3F	61	3F	6C	3F	78	3F	80	3F
00000940:	99	3F	16	40	1F	40	26	40	2F	40	80	40
00000950:	AA	40	B6	40	C0	40	CA	40	D4	40	DC	40
00000960:	FA	40	02	41	0B	41	15	41	1D	41	44	41
00000970:	5F	41	66	41	6E	41	7B	41	86	41	8D	41
00000980:	A7	41	AF	41	B7	41	BD	41	3B	42	60	44
00000990:	DD	44	55	46	5E	46	42	4A	4E	4A	56	4A
000009A0:	AA	4A	B3	4A	B7	4A	BB	4A	BD	4A	BF	4A
000009B0:	C6	4A	CA	4A	CD	4A	D1	4A	DA	4A	E0	4A

Figure 9.28: Hiew: last\_errnos table

Their size can be determined visually (red lines are drawn here).

While dumping these values, we have found that each 16-bit number is the last error code for each block.

So that's how Oracle RDBMS quickly finds the error message:

- load a table we will call `last_errnos` (that contains the last error number for each block);
- find a block that contains the error code we need, assuming all error codes increase across each block and across the file as well;
- load the specific block;
- enumerate the 6-byte structures until the specific error number is found;
- get the position of the first character from the current 6-byte block;
- get the position of the last character from the next 6-byte block;
- load all characters of the message in this range.

This is C program that we wrote which unpacks .MSB-files: [beginners.re](#).

There are also the two files which were used in the example (Oracle RDBMS 11.1.0.6): [beginners.re](#), [beginners.ms](#).

---

## 9.6.1 Summary

The method is probably too old-school for modern computers. Supposedly, this file format was developed in the mid-80's by someone who also coded for *big iron* with memory/disk space economy in mind. Nevertheless, it has been an interesting and yet easy task to understand a proprietary file format without looking into Oracle RDBMS's code.

## 9.7 Exercise

Try to reverse engineer of any binary files of your favorite game, including high-score files, resources, etc. There are also binary files with known structure: utmp/wtmp files.

The EXIF header in JPEG file is documented, but you can try to understand its structure without help, just shoot photos at various date/time, places, and try to find date/time and GPS location in EXIF. Try to patch GPS location, upload JPEG file to Facebook and see, how it will put your picture on the map.

Try to patch any information in MP3 file and see how your favorite MP3-player will react.

# Chapter 10

## Other things

### 10.1 Executable files patching

#### 10.1.1 Text strings

The C strings are the thing that is the easiest to patch (unless they are encrypted) in any hex editor. This technique is available even for those who are not aware of machine code and executable file formats. The new string has not to be bigger than the old one, because there's a risk of overwriting another value or code there.

Using this method, a lot of software was *localized* in the MS-DOS era, at least in the ex-USSR countries in 80's and 90's. It was the reason why some weird abbreviations were present in the *localized* software: there was no room for longer strings.

As for Delphi strings, the string's size must also be corrected, if needed.

#### 10.1.2 x86 code

Frequent patching tasks are:

- One of the most frequent jobs is to disable some instruction. It is often done by filling it using byte `0x90` (`NOP`).
- Conditional jumps, which have an opcode like `74 xx` (`JZ`), can be filled with two `NOPs`.  
It is also possible to disable a conditional jump by writing 0 at the second byte (*jump offset*).
- Another frequent job is to make a conditional jump to always trigger: this can be done by writing `0xEB` instead of the opcode, which stands for `JMP`.
- A function's execution can be disabled by writing `RETN` (0xC3) at its beginning. This is true for all functions excluding `stdcall` ([6.1.2 on page 732](#)). While patching `stdcall` functions, one has to determine the number of arguments (for example, by finding `RETN` in this function), and use `RETN` with a 16-bit argument (0xC2).
- Sometimes, a disabled function has to return 0 or 1. This can be done by `MOV EAX, 0` or `MOV EAX, 1`, but it's slightly verbose.  
A better way is `XOR EAX, EAX` (2 bytes `0x31 0xC0`) or `XOR EAX, EAX / INC EAX` (3 bytes `0x31 0xC0 0x41`).

A software may be protected against modifications.

This protection is often done by reading the executable code and calculating a checksum. Therefore, the code must be read before protection is triggered.

This can be determined by setting a breakpoint on reading memory.

`tracer` has the BPM option for this.

PE executable file relocs ([6.5.2 on page 758](#)) must not to be touched while patching, because the Windows loader may overwrite your new code. (They are grayed in Hiew, for example: [fig.1.21](#)).

## 10.2. FUNCTION ARGUMENTS STATISTICS

As a last resort, it is possible to write jumps that circumvent the relocations, or you will have to edit the relocations table.

## **10.2 Function arguments statistics**

I've always been interesting in what is average number of function arguments.

Just analyzed many Windows 7 32-bit DLLs

(crypt32.dll, mfc71.dll, msrv100.dll, shell32.dll, user32.dll, d3d11.dll, mshtml.dll, msxml6.dll, sqlncli11.dll, wininet.dll, mfc120.dll, msvbvm60.dll, ole32.dll, themeui.dll, wmp.dll) (because they use stdcall convention, and so it is to grep disassembly output just by RETN X ).

- no arguments: 29%
- 1 argument: 23%
- 2 arguments: 20%
- 3 arguments: 11%
- 4 arguments: 7%
- 5 arguments: 3%
- 6 arguments: 2%
- 7 arguments: 1%

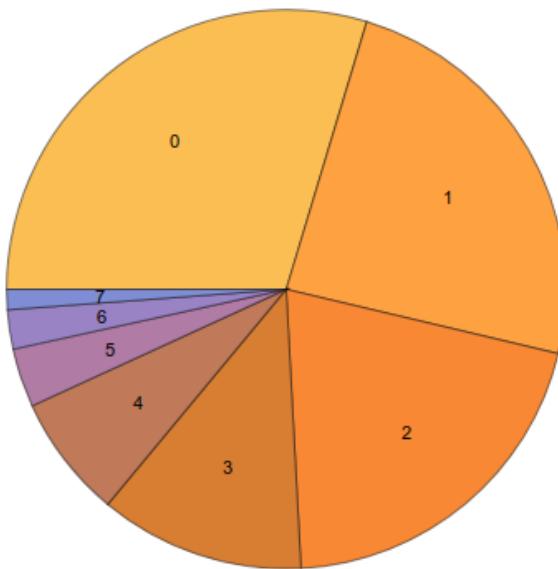


Figure 10.1: Function arguments statistics

This is heavily dependent on programming style and may be very different for other software products.

## **10.3 Compiler intrinsic**

A function specific to a compiler which is not an usual library function. The compiler generates a specific machine code instead of a 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 they are present in most [CPUs](#).

## 10.4. COMPILER'S ANOMALIES

For programmer's convenience, at least MSVC has pseudofunctions `_rotl()` and `_rotr()`<sup>1</sup> which are translated by the compiler directly to the ROL/ROR x86 instructions.

Another example are functions to generate SSE-instructions right in the code.

Full list of MSVC intrinsics: [MSDN](#).

## 10.4 Compiler's anomalies

### 10.4.1 Oracle RDBMS 11.2 and Intel C++ 10.1

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`. The second `JZ` is thus meaningless.

Listing 10.1: kdli.o from libserver11.a

```
.text:08114CF1          loc_8114CF1: ; CODE XREF: _PGOSF539_kdlimemSer+89A
.text:08114CF1
.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 10.2: from the same code

```
.text:0811A2A5          loc_811A2A5: ; CODE XREF: kdliSerLengths+11C
.text:0811A2A5
.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 supposedly a code generator bug that was not found by tests, because resulting code works correctly anyway.

### 10.4.2 MSVC 6.0

Just found in some old code:

```
fabs
fld   [esp+50h+var_34]
fabs
fxch st(1) ; first instruction
fxch st(1) ; second instruction
faddp st(1), st
```

<sup>1</sup>[MSDN](#)

```
fcomp [esp+50h+var_3C]
fnstsw ax
test ah, 41h
jz short loc_100040B7
```

The first `FXCH` instruction swaps `ST(0)` and `ST(1)`, the second do the same, so both do nothing. This is a program uses MFC42.dll, so it could be MSVC 6.0, 5.0 or maybe even MSVC 4.2 from 1990s.

This pair do nothing, so it probably wasn't caught by MSVC compiler tests. Or maybe I wrong?

### 10.4.3 Summary

Other compiler anomalies here in this book: [1.22.2 on page 315](#), [3.7.3 on page 492](#), [3.15.7 on page 531](#), [1.20.7 on page 302](#), [1.14.4 on page 148](#), [1.22.5 on page 333](#).

Such cases are demonstrated here in this book, to show that such compilers errors are possible and sometimes one should not to rack one's brain while thinking why did the compiler generate such strange code.

## 10.5 Itanium

Although almost failed, Intel Itanium ([IA64](#)) is a very interesting architecture.

While [OOE](#) CPUs decides how to rearrange their instructions and execute them in parallel, [EPIC<sup>2</sup>](#) was an attempt to shift these decisions to the compiler: to let it group the instructions at the compile stage.

This resulted in notoriously complex compilers.

Here is one sample of [IA64](#) code: simple cryptographic algorithm from the Linux kernel:

Listing 10.3: 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];

    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:

<sup>2</sup>Explicitly parallel instruction computing

Listing 10.4: 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 = 0xFFFFFFFF9E3779B9;; // 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 a size of 16 bytes (128 bits) and consists of template code (5 bits) + 3 instructions (41 bits for each).

[IDA](#) shows the bundles as 6+6+4 bytes —you can easily spot the pattern.

All 3 instructions from each bundle usually executes simultaneously, unless one of instructions has a “stop bit”.

Supposedly, Intel and HP engineers gathered statistics on most frequent instruction patterns and decided to bring bundle types ([AKA](#) “templates”): a bundle code defines the instruction types in the bundle. There are 12 of them.

For example, the zeroth bundle type is [MII](#), which implies the first instruction is Memory (load or store), the second and third ones are I (integer instructions).

Another example is the bundle of type [0x1d](#): [MFB](#): the first instruction is Memory (load or store), the second one is Float ([FPU](#) instruction), and the third is Branch (branch instruction).

If the compiler cannot pick a suitable instruction for the relevant bundle slot, it may insert a [NOP](#): you can

## 10.6. 8086 MEMORY MODEL

see here the `nop.i` instructions (**NOP** at the place where the integer instruction might be) or `nop.m` (a memory instruction might be at this slot).

**NOPs** are inserted automatically when one uses assembly language manually.

And that is not all. Bundles are also grouped.

Each bundle may have a “stop bit”, so all the consecutive bundles with a terminating bundle which has the “stop bit” can be executed simultaneously.

In practice, Itanium 2 can execute 2 bundles at once, resulting in the execution of 6 instructions at once.

So all instructions inside a bundle and a bundle group cannot interfere with each other (i.e., must not have data hazards).

If they do, the results are to be undefined.

Each stop bit is marked in assembly language as two semicolons (`;;`) after the instruction.

So, the instructions at [90-ac] may be executed simultaneously: they do not interfere. The next group is [b0-cc].

We also see a stop bit at 10c. The next instruction at 110 has a stop bit too.

This implies that these instructions must be executed isolated from all others (as in **CISC**).

Indeed: the next instruction at 110 uses the result from the previous one (the value in register r26), so they cannot be executed at the same time.

Apparently, the compiler was not able to find a better way to parallelize the instructions, in other words, to load **CPU** as much as possible, hence too much stop bits and **NOPs**.

Manual assembly programming is a tedious job as well: the programmer has to group the instructions manually.

The programmer is still able to add stop bits to each instructions, but this will degrade the performance that Itanium was made for.

An interesting examples of manual **IA64** assembly code can be found in the Linux kernel’s sources:

<http://go.yurichev.com/17322>.

Another introductory paper on Itanium assembly: [Mike Burrell, *Writing Efficient Itanium 2 Assembly Code* (2010)]<sup>3</sup>, [papasutra of haquebright, *WRITING SHELLCODE FOR IA-64* (2001)]<sup>4</sup>.

Another very interesting Itanium feature is the *speculative execution* and the **NaT** (“not a thing”) bit, somewhat resembling **NaN** numbers:

[MSDN](#).

## 10.6 8086 memory model

When dealing with 16-bit programs for MS-DOS or Win16 ( [8.5.3 on page 829](#) or [3.22.5 on page 620](#)), we can see that the pointers consist of two 16-bit values. What do they mean? Oh yes, that is another weird MS-DOS and 8086 artifact.

8086/8088 was a 16-bit CPU, but was able to address 20-bit address in RAM (thus being able to access 1MB of external memory).

The external memory address space was divided between **RAM** (640KB max), **ROM**, windows for video memory, EMS cards, etc.

Let’s also recall that 8086/8088 was in fact an inheritor of the 8-bit 8080 CPU.

The 8080 has a 16-bit memory space, i.e., it was able to address only 64KB.

And probably because of reason of old software porting<sup>5</sup>, 8086 can support many 64KB windows simultaneously, placed within the 1MB address space.

This is some kind of a toy-level virtualization.

<sup>3</sup>Also available as <http://yurichev.com/mirrors/RE/itanium.pdf>

<sup>4</sup>Also available as <http://phrack.org/issues/57/5.html>

<sup>5</sup>The author is not 100% sure here

## 10.7. BASIC BLOCKS REORDERING

All 8086 registers are 16-bit, so to address more, special segment registers (CS, DS, ES, SS) were introduced.

Each 20-bit pointer is calculated using the values from a segment register and an address register pair (e.g. DS:BX) as follows:

$$\text{real\_address} = (\text{segment\_register} \ll 4) + \text{address\_register}$$

For example, the graphics ([EGA<sup>6</sup>](#), [VGA<sup>7</sup>](#)) video [RAM](#) window on old IBM PC-compatibles has a size of 64KB.

To access it, a value of 0xA000 has to be stored in one of the segment registers, e.g. into DS.

Then DS:0 will address the first byte of video [RAM](#) and DS:0xFFFF — the last byte of RAM.

The real address on the 20-bit address bus, however, will range from 0xA0000 to 0xFFFF.

The program may contain hard-coded addresses like 0x1234, but the [OS](#) may need to load the program at arbitrary addresses, so it recalculates the segment register values in a way that the program does not have to care where it's placed in the RAM.

So, any pointer in the old MS-DOS environment in fact consisted of the segment address and the address inside segment, i.e., two 16-bit values. 20-bit was enough for that, though, but we needed to recalculate the addresses very often: passing more information on the stack seemed a better space/convenience balance.

By the way, because of all this it was not possible to allocate a memory block larger than 64KB.

The segment registers were reused at 80286 as selectors, serving a different function.

When the 80386 CPU and computers with bigger [RAM](#) were introduced, MS-DOS was still popular, so the DOS extenders emerged: these were in fact a step toward a “serious” [OS](#), switching the CPU in protected mode and providing much better memory [APIs](#) for the programs which still needed to run under MS-DOS.

Widely popular examples include DOS/4GW (the DOOM video game was compiled for it), Phar Lap, PMODE.

By the way, the same way of addressing memory was used in the 16-bit line of Windows 3.x, before Win32.

## 10.7 Basic blocks reordering

### 10.7.1 Profile-guided optimization

This optimization method can move some [basic blocks](#) to another section of the executable binary file.

Obviously, there are parts of a function which are executed more frequently (e.g., loop bodies) and less often (e.g., error reporting code, exception handlers).

The compiler adds instrumentation code into the executable, then the developer runs it with a lot of tests to collect statistics.

Then the compiler, with the help of the statistics gathered, prepares final the 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 usage.

An example from Oracle RDBMS code, which was compiled with Intel C++:

Listing 10.5: orageneric11.dll (win32)

```
_skgfsync    public _skgfsync
              proc near
; address 0x6030D86A
              db      66h
              nop
              push    ebp
              mov     ebp, esp
              mov     edx, [ebp+0Ch]
              test   edx, edx
```

<sup>6</sup>Enhanced Graphics Adapter

<sup>7</sup>Video Graphics Array

## 10.7. BASIC BLOCKS REORDERING

```
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 ...
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 between these two code fragments is almost 9 MB.

All infrequently executed code was placed at the end of the code section of the DLL file, among all function parts.

This part of the function was marked by the Intel C++ compiler with the `VInfreq` prefix.

Here we see that a part of the function that writes to a log file (presumably in case of error or warning or something like that) which was probably not executed very often when Oracle's developers gathered statistics (if it was executed at all).

The writing to log basic block eventually returns the control flow to the “hot” part of the function.

Another “infrequent” part is the `basic block` returning error code 27050.

In Linux ELF files, all infrequently executed code is moved by Intel C++ into the 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 into its core and error handling parts.

# Chapter 11

## Books/blogs worth reading

### 11.1 Books and other materials

#### 11.1.1 Reverse Engineering

- Eldad Eilam, *Reversing: Secrets of Reverse Engineering*, (2005)
- Bruce Dang, Alexandre Gazet, Elias Bachaalany, Sebastien Josse, *Practical Reverse Engineering: x86, x64, ARM, Windows Kernel, Reversing Tools, and Obfuscation*, (2014)
- Michael Sikorski, Andrew Honig, *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*, (2012)
- Chris Eagle, *IDA Pro Book*, (2011)

#### 11.1.2 Windows

- Mark Russinovich, *Microsoft Windows Internals*

Blogs:

- Microsoft: Raymond Chen
- nynaeve.net

#### 11.1.3 C/C++

- [Brian W. Kernighan, Dennis M. Ritchie, *The C Programming Language*, (1988)]
- ISO/IEC 9899:TC3 (C C99 standard), (2007)<sup>1</sup>
- Bjarne Stroustrup, *The C++ Programming Language, 4th Edition*, (2013)
- C++11 standard<sup>2</sup>
- Agner Fog, *Optimizing software in C++* (2015)<sup>3</sup>
- Marshall Cline, *C++ FAQ*<sup>4</sup>
- Dennis Yurichev, *C/C++ programming language notes*<sup>5</sup>

<sup>1</sup>Also available as <http://go.yurichev.com/17274>

<sup>2</sup>Also available as <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3690.pdf>.

<sup>3</sup>Also available as [http://agner.org/optimize/optimizing\\_cpp.pdf](http://agner.org/optimize/optimizing_cpp.pdf).

<sup>4</sup>Also available as <http://go.yurichev.com/17291>

<sup>5</sup>Also available as <http://yurichev.com/C-book.html>

**11.1.4 x86 / x86-64**

- Intel manuals<sup>6</sup>
- AMD manuals<sup>7</sup>
- Agner Fog, *The microarchitecture of Intel, AMD and VIA CPUs*, (2016)<sup>8</sup>
- Agner Fog, *Calling conventions* (2015)<sup>9</sup>
- [*Intel® 64 and IA-32 Architectures Optimization Reference Manual*, (2014)]
- [*Software Optimization Guide for AMD Family 16h Processors*, (2013)]

Somewhat outdated, but still interesting to read:

Michael Abrash, *Graphics Programming Black Book*, 1997<sup>10</sup> (he is known for his work on low-level optimization for such projects as Windows NT 3.1 and id Quake).

**11.1.5 ARM**

- ARM manuals<sup>11</sup>
- *ARM(R) Architecture Reference Manual, ARMv7-A and ARMv7-R edition*, (2012)
- [*ARM Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile*, (2013)]<sup>12</sup>
- Advanced RISC Machines Ltd, *The ARM Cookbook*, (1994)<sup>13</sup>

**11.1.6 Java**

[Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, *The Java(R) Virtual Machine Specification / Java SE 7 Edition*] <sup>14</sup>.

**11.1.7 UNIX**

Eric S. Raymond, *The Art of UNIX Programming*, (2003)

**11.1.8 Programming in general**

- Brian W. Kernighan, Rob Pike, *Practice of Programming*, (1999)
- Henry S. Warren, *Hacker's Delight*, (2002).
- (For hard-core geeks with computer science and mathematical background) Donald E. Knuth, *The Art of Computer Programming*.

<sup>6</sup>Also available as <http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html>

<sup>7</sup>Also available as <http://developer.amd.com/resources/developer-guides-manuals/>

<sup>8</sup>Also available as <http://agner.org/optimize/microarchitecture.pdf>

<sup>9</sup>Also available as [http://www.agner.org/optimize/calling\\_conventions.pdf](http://www.agner.org/optimize/calling_conventions.pdf)

<sup>10</sup>Also available as <https://github.com/jagregory/abrush-black-book>

<sup>11</sup>Also available as <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.architecture.reference/index.html>

<sup>12</sup>Also available as [http://yurichev.com/mirrors/ARMv8-A\\_Architecture\\_Reference\\_Manual\\_\(Issue\\_A.a\).pdf](http://yurichev.com/mirrors/ARMv8-A_Architecture_Reference_Manual_(Issue_A.a).pdf)

<sup>13</sup>Also available as <http://go.yurichev.com/17273>

<sup>14</sup>Also available as <https://docs.oracle.com/javase/specs/jvms/se7/jvms7.pdf>; <http://docs.oracle.com/javase/specs/jvms/se7/html/>

## 11.1.9 Cryptography

- Bruce Schneier, *Applied Cryptography*, (John Wiley & Sons, 1994)
- (Free) Ivh, *Crypto 101*<sup>15</sup>
- (Free) Dan Boneh, Victor Shoup, *A Graduate Course in Applied Cryptography*<sup>16</sup>.

---

<sup>15</sup>Also available as <https://www.cryptol101.io/>

<sup>16</sup>Also available as <https://crypto.stanford.edu/~dabo/cryptobook/>

# Chapter 12

## Communities

There are two excellent RE<sup>1</sup>-related subreddits on reddit.com: [reddit.com/r/ReverseEngineering/](https://www.reddit.com/r/ReverseEngineering/) and [reddit.com/r/remath](https://www.reddit.com/r/remath) (on the topics for the intersection of RE and mathematics).

There is also a RE part of the Stack Exchange website: [reverseengineering.stackexchange.com](https://reverseengineering.stackexchange.com).

On IRC there's a ##re channel on FreeNode<sup>2</sup>.

---

<sup>1</sup>Reverse Engineering

<sup>2</sup>[freenode.net](https://freenode.net)

# **Afterword**

## 12.1 Questions?

Do not hesitate to mail any questions to the author:

`<dennis(a)yurichev.com>`. Do you have any suggestion on new content for the book? Please do not hesitate to send any corrections (including grammar (you see how horrible my English is?)), etc.

The author is working on the book a lot, so the page and listing numbers, etc., are changing very rapidly. Please do not refer to page and listing numbers in your emails to me. There is a much simpler method: make a screenshot of the page, in a graphics editor underline the place where you see the error, and send it to me. He'll fix it much faster. And if you familiar with git and  $\text{\LaTeX}$  you can fix the error right in the source code:

[GitHub](#).

Do not worry to bother me while writing me about any petty mistakes you found, even if you are not very confident. I'm writing for beginners, after all, so beginners' opinions and comments are crucial for my job.

# **Appendix**

## **.1.1 Terminology**

Common for 16-bit (8086/80286), 32-bit (80386, etc.), 64-bit.

**byte** 8-bit. The DB assembly directive is used for defining variables and arrays of bytes. Bytes are passed in the 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 the 16-bit part of the 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 the 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 environment, 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.

## **.1.2 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 the 8-bit 8080) still supported for backward compatibility. Older 8-bit CPUs (8080) had 16-bit registers divided by two. Programs written for 8080 could access the low byte part of 16-bit registers, high byte part or the whole 16-bit register. Perhaps, this feature was left in 8086 as a helper for easier porting. This feature is usually not present in RISC CPUs.

Registers prefixed with R- appeared in x86-64, 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 the byte parts of these registers are prefixed by L, e.g.: R8L, but IDA names these registers by adding the B suffix, e.g.: R8B.

### **RAX/EAX/AX/AL**

Byte number:							
7th	6th	5th	4th	3rd	2nd	1st	0th
RAX <sup>x64</sup>							
EAX							
AX							
AH AL							

AKA accumulator. The result of a function is usually returned via this register.

### **RBX/EBX/BX/BL**

Byte number:							
7th	6th	5th	4th	3rd	2nd	1st	0th
RBX <sup>x64</sup>							
EBX							
BX							
BH BL							

Byte number:							
7th	6th	5th	4th	3rd	2nd	1st	0th
RCX <sup>x64</sup>							
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).

### **RDX/EDX/DX/DL**

Byte number:							
7th	6th	5th	4th	3rd	2nd	1st	0th
RDX <sup>x64</sup>							
EDX							
DX							
DH DL							

### **RSI/ESI/SI/SIL**

Byte number:							
7th	6th	5th	4th	3rd	2nd	1st	0th
RSI <sup>x64</sup>							
ESI							
SI							
SIL <sup>x64</sup>							

**AKA** “source index”. Used as source in the instructions REP MOVSx, REP CMPSx.

### **RDI/EDI/DI/DIL**

Byte number:							
7th	6th	5th	4th	3rd	2nd	1st	0th
RDI <sup>x64</sup>							
EDI							
DI							
DIL <sup>x64</sup>							

**AKA** “destination index”. Used as a pointer to the destination in the instructions REP MOVSx, REP STOSx.

### **R8/R8D/R8W/R8L**

Byte number:							
7th	6th	5th	4th	3rd	2nd	1st	0th
R8							
R8D							
R8W							
R8L							

### **R9/R9D/R9W/R9L**

Byte number:							
7th	6th	5th	4th	3rd	2nd	1st	0th
R9							
R9D							
R9W							
R9L							

**R10/R10D/R10W/R10L**

Byte number:							
7th	6th	5th	4th	3rd	2nd	1st	0th
				R10			
					R10D		
						R10W	
							R10L

**R11/R11D/R11W/R11L**

Byte number:							
7th	6th	5th	4th	3rd	2nd	1st	0th
				R11			
					R11D		
						R11W	
							R11L

**R12/R12D/R12W/R12L**

Byte number:							
7th	6th	5th	4th	3rd	2nd	1st	0th
				R12			
					R12D		
						R12W	
							R12L

**R13/R13D/R13W/R13L**

Byte number:							
7th	6th	5th	4th	3rd	2nd	1st	0th
				R13			
					R13D		
						R13W	
							R13L

**R14/R14D/R14W/R14L**

Byte number:							
7th	6th	5th	4th	3rd	2nd	1st	0th
				R14			
					R14D		
						R14W	
							R14L

**R15/R15D/R15W/R15L**

Byte number:							
7th	6th	5th	4th	3rd	2nd	1st	0th
				R15			
					R15D		
						R15W	
							R15L

Byte number:							
7th	6th	5th	4th	3rd	2nd	1st	0th
				RSP			
					ESP		
						SP	
							SPL

**AKA stack pointer.** Usually points to the current stack except in those cases when it is not yet initialized.

### **RBP/EBP/BP/BPL**

Byte number:							
7th	6th	5th	4th	3rd	2nd	1st	0th
			RBP				
				EBP			
					BP		
						BPL	

**AKA frame pointer.** Usually used for local variables and accessing the arguments of the function. More about it: ([1.9.1 on page 68](#)).

### **RIP/EIP/IP**

Byte number:							
7th	6th	5th	4th	3rd	2nd	1st	0th
			RIP <sup>x64</sup>				
				EIP			
					IP		

**AKA “instruction pointer”**<sup>3</sup>. Usually always points to the instruction to be executed right now. Cannot be modified, however, it is possible to do this (which is equivalent):

```
MOV EAX, ...
JMP EAX
```

Or:

```
PUSH value
RET
```

### **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 made so for faster access to the [TLS](#) and other structures like the [TIB](#).

In the past, these registers were used as segment registers ([10.6 on page 990](#)).

### **Flags register**

**AKA EFLAGS.**

<sup>3</sup>Sometimes also called “program counter”

## .1. X86

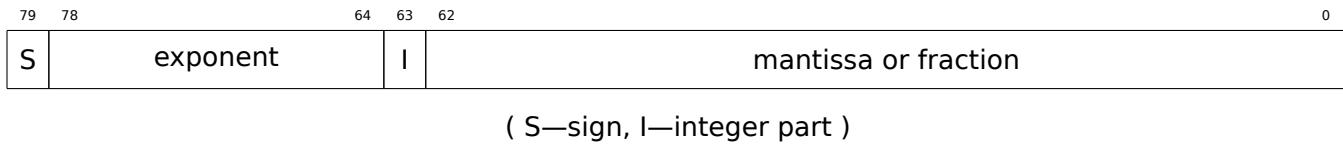
Bit (mask)	Abbreviation (meaning)	Description
0 (1)	CF (Carry)	The CLC/STC/CMC instructions are used for setting/resetting/toggling this flag ( <a href="#">1.19.7 on page 235</a> ).
2 (4)	PF (Parity)	Exist solely for work with <a href="#">BCD-numbers</a>
4 (0x10)	AF (Adjust)	
6 (0x40)	ZF (Zero)	Setting to 0 if the last operation's result is equal to 0.
7 (0x80)	SF (Sign)	
8 (0x100)	TF (Trap)	Used for debugging. If turned on, an exception is to be generated after each instruction's execution.
9 (0x200)	IF (Interrupt enable)	Are interrupts enabled. The CLI/STI instructions are used for setting/resetting the flag
10 (0x400)	DF (Direction)	A direction is set for the REP MOVSx/CMPsx/LODSx/SCASx instructions. The CLD/STD instructions are used for setting/resetting the flag See also: <a href="#">3.26 on page 652</a> .
11 (0x800)	OF (Overflow)	
12, 13 (0x3000)	IOPL (I/O privilege level) <sup>i286</sup>	
14 (0x4000)	NT (Nested task) <sup>i286</sup>	
16 (0x10000)	RF (Resume) <sup>i386</sup>	Used for debugging. The CPU ignores the hardware breakpoint in DRx if the flag is set.
17 (0x20000)	VM (Virtual 8086 mode) <sup>i386</sup>	
18 (0x40000)	AC (Alignment check) <sup>i486</sup>	
19 (0x80000)	VIF (Virtual interrupt) <sup>i586</sup>	
20 (0x100000)	VIP (Virtual interrupt pending) <sup>i586</sup>	
21 (0x200000)	ID (Identification) <sup>i586</sup>	

All the rest flags are reserved.

### .1.3 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:



### Control Word

Register controlling the behavior of the [FPU](#).

## 1. X86

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 define if to generate exception in the case of a corresponding error.

## 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 signal about exceptions.

About the C3, C2, C1, C0 you can read more here: ([1.19.7 on page 234](#)).

N.B.: When ST(x) is used, the FPU adds  $x$  to TOP (by modulo 8) and that is how it gets the internal register's number.

## Tag Word

The register has current information about the usage of numbers registers.

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)

Each tag contains information about a physical FPU register (R(x)), not logical (ST(x)).

For each tag:

- 00 — The register contains a non-zero value

## .1. X86

---

- 01 — The register contains 0
- 10 — The register contains a special value ([NAN<sup>4</sup>](#),  $\infty$ , or denormal)
- 11 — The register is empty

## .1.4 SIMD registers

### MMX registers

8 64-bit registers: MM0..MM7.

### 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.

## .1.5 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

### DR6

Bit (mask)	Description
0 (1)	B0 — breakpoint #1 has been triggered
1 (2)	B1 — breakpoint #2 has been triggered
2 (4)	B2 — breakpoint #3 has been triggered
3 (8)	B3 — breakpoint #4 has been triggered
13 (0x2000)	BD — modification attempt of one of the DRx registers. may be raised if GD is enabled
14 (0x4000)	BS — single step breakpoint (TF flag has been set in EFLAGS). Highest priority. Other bits may also be set.
15 (0x8000)	BT (task switch flag)

N.B. A single step breakpoint is a breakpoint which occurs after each instruction. It can be enabled by setting TF in EFLAGS ([.1.2 on page 1003](#)).

### DR7

Breakpoint types are set here.

---

<sup>4</sup>Not a Number

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 is to be raised if any MOV instruction tries to modify one of the 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

The 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

## .1.6 Instructions

Instructions marked as (M) are not usually generated by the compiler: if you see one of them, it is probably a hand-written piece of assembly code, or a compiler intrinsic ([10.3 on page 986](#)).

Only the most frequently used instructions are listed here. You can read [11.1.4 on page 994](#) for a full documentation.

Do you have to know all instruction's opcodes by heart? No, only those which are used for code patching ([10.1.2 on page 985](#)). All the rest of the opcodes don't need to be memorized.

### Prefixes

**LOCK** forces CPU to make exclusive access to the RAM in multiprocessor environment. For the sake of simplification, it can be said that when an instruction with this prefix is executed, all other CPUs in a multiprocessor system are stopped. Most often it is used for critical sections, semaphores, mutexes. Commonly used with ADD, AND, BTR, BTS, CMPXCHG, OR, XADD, XOR. You can read more about critical sections here ([6.5.4 on page 785](#)).

**REP** is used with the MOVSx and STOSx instructions: execute the instruction in a loop, the counter is located in the CX/ECX/RCX register. For a detailed description, read more about the MOVSx ([.1.6 on page 1010](#)) and STOSx ([.1.6 on page 1011](#)) instructions.

The instructions prefixed by REP are sensitive to the DF flag, which is used to set the direction.

**REPE/REPNE** (AKA REPZ/REPNZ) used with CMPSx and SCASx instructions: execute the last instruction in a loop, the count is set in the `CX / ECX / RCX` register. It terminates prematurely if ZF is 0 (REPE) or if ZF is 1 (REPNE).

For a detailed description, you can read more about the CMPSx ([.1.6 on page 1013](#)) and SCASx ([.1.6 on page 1011](#)) instructions.

Instructions prefixed by REPE/REPNE are sensitive to the DF flag, which is used to set the direction.

## Most frequently used instructions

These can be memorized in the first place.

**ADC** (add with carry) add values, [increment](#) the result if the CF flag is set. ADC is often used for the addition of large values, for example, to add two 64-bit values in a 32-bit environment using two ADD and ADC instructions. For example:

```
; work with 64-bit values: add val1 to val2.
; .lo means 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: [1.28 on page 397](#).

**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 without writing the result

**DEC** [decrement](#).Unlike other arithmetic instructions, `DEC` doesn't modify CF flag.

**IMUL** signed multiply `IMUL` often used instead of `MUL`, read more about it: [2.2.1](#).

**INC** [increment](#).Unlike other arithmetic instructions, `INC` doesn't modify CF flag.

**JCXZ, JCXZ, JRCXZ** (M) jump if CX/ECX/RCX=0

**JMP** jump to another address. The opcode has a [jump offset](#).

**Jcc** (where cc—condition code)

A lot of these instructions have synonyms (denoted with AKA), this was done for convenience. Synonymous instructions are translated into the same opcode. The opcode has a [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.

## 1. X86

**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

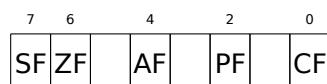
**JPO** jump if PF flag is cleared (Jump Parity Odd)

**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 **EBC** register to its initial state.

**LEA (Load Effective Address)** form an address

This instruction was intended not for summing values and multiplication but for forming an address, e.g., for calculating the address of an array element by adding the array address, element index, with multiplication of element size<sup>5</sup>.

So, the difference between **MOV** and **LEA** is that **MOV** forms a memory address and loads a 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 performed by it does not alter **CPU** flags. This may be very important for **OOE** processors (to create less data dependencies).

Aside from this, starting at least at Pentium, LEA instruction is executed in 1 cycle.

```
int f(int a, int b)
{
    return a*8+b;
};
```

Listing 1: Optimizing MSVC 2010

```
_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;
};
```

<sup>5</sup>See also: [wikipedia](#)

```
_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 performs faster than one IMUL.

**MOVSB/MOVSW/MOVSD/MOVSQ** copy byte/ 16-bit word/ 32-bit word/ 64-bit word from the address which is in SI/ESI/RSI into the address which is in DI/EDI/RDI.

Together with the REP prefix, it is to be repeated in a loop, the count is to be stored in the CX/ECX/RCX register: it works like memcpy() in C. If the block size is known to the compiler in the compile stage, memcpy() is often inlined into a short code fragment using REP MOVSw, sometimes even as several instructions.

The 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 works faster than copying 15 bytes using just one REP MOVSb).

**MOVSX** load with sign extension see also: ([1.17.1 on page 201](#))

**MOVZX** load and clear all other bits see also: ([1.17.1 on page 202](#))

**MOV** load value. this instruction name is misnomer, resulting in some confusion (data is not moved but copied), in other architectures the same instructions is usually named “LOAD” and/or “STORE” or something like that.

One important thing: if you set the low 16-bit part of a 32-bit register in 32-bit mode, the high 16 bits remains as they were. But if you modify the low 32-bit part of the register in 64-bit mode, the high 32 bits of the register will be cleared.

Supposedly, it was done to simplify porting code to x86-64.

**MUL** unsigned multiply. **IMUL** often used instead of **MUL**, read more about it: [2.2.1](#).

**NEG** negation:  $op = -op$  Same as **NOT op / ADD op, 1**.

**NOP** **NOP**. Its opcode is 0x90, it is in fact the **XCHG EAX,EAX** idle instruction. This implies that x86 does not have a dedicated **NOP** instruction (as in many RISC). This book has at least one listing where GDB shows NOP as 16-bit XCHG instruction: [1.8.1 on page 48](#).

More examples of such operations: ([1.7 on page 1019](#)).

**NOP** may be generated by the compiler for aligning labels on a 16-byte boundary. Another very popular usage of **NOP** is to replace manually (patch) some instruction like a conditional jump to **NOP** in order to disable its execution.

**NOT**  $op1: op1 = \neg op1$ . logical inversion Important feature—the instruction doesn’t change flags.

**OR** logical “or”

**POP** get a value from the stack: **value=SS:[ESP]; ESP=ESP+4 (or 8)**

**PUSH** push a value into the stack: **ESP=ESP-4 (or 8); SS:[ESP]=value**

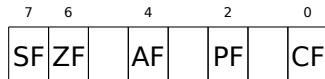
**RET** return from subroutine: **POP tmp; JMP tmp**.

In fact, RET is an assembly language macro, in Windows and \*NIX environment it is translated into RETN (“return near”) or, in MS-DOS times, where the memory was addressed differently ([10.6 on page 990](#)), into RETF (“return far”).

**RET** can have an operand. Then it works like this:

**POP tmp; ADD ESP op1; JMP tmp**. **RET** with an operand usually ends functions in the *stdcall* calling convention, see also: [6.1.2 on page 732](#).

**SAHF** copy bits from AH to CPU flags:



**SBB** (*subtraction with borrow*) subtract values, [decrement](#) the result if the CF flag is set. SBB is 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 means 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: [1.28 on page 397](#).

**SCASB/SCASW/SCASD/SCASQ** (M) compare byte/ 16-bit word/ 32-bit word/ 64-bit word that's stored in AX/EAX/RAX with a variable whose address is in DI/EDI/RDI. Set flags as [CMP](#) does.

This instruction is often used with the REPNE prefix: continue to scan the buffer until a special value stored in AX/EAX/RAX is found. Hence “NE” in REPNE: continue to scan while the compared values are not equal and stop when equal.

It is often used like the `strlen()` C standard function, to determine an [ASCIIIZ](#) string's length:

Example:

```
lea    edi, string
mov   ecx, 0xFFFFFFFFh ; scan  $2^{32} - 1$  bytes, i.e., almost infinitely
xor   eax, eax        ; 0 is the terminator
repne scasb
add   edi, 0xFFFFFFFFh ; correct it

; now EDI points to the last character of the ASCIIIZ string.

; lets determine string length
; current ECX = -1-strlen

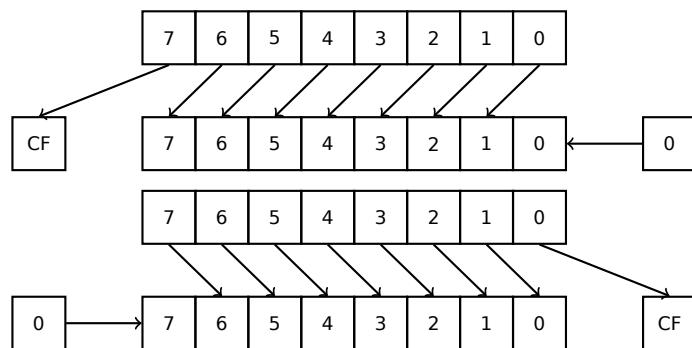
not   ecx
dec   ecx

; now ECX contain string length
```

If we use a different AX/EAX/RAX value, the function acts like the `memchr()` standard C function, i.e., it finds a specific byte.

**SHL** shift value left

**SHR** shift value right:



These instructions are frequently used for multiplication and division by  $2^n$ . Another very frequent application is processing bit fields: [1.22 on page 304](#).

**SHRD** op1, op2, op3: shift value in op2 right by op3 bits, taking bits from op1.

Example: [1.28 on page 397](#).

**STOSB/STOSW/STOSD/STOSQ** store byte/ 16-bit word/ 32-bit word/ 64-bit word from AX/EAX/RAX into the address which is in DI/EDI/RDI.

## 1. X86

Together with the REP prefix, it is to be repeated in a loop, the counter is in the CX/ECX/RCX register: it works like `memset()` in C. If the block size is known to the compiler on compile stage, `memset()` is often inlined into a 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 works faster than storing 15 bytes using just one REP STOSB).

**SUB** subtract values. A frequently occurring pattern is `SUB reg, reg`, which implies zeroing of `reg`.

**TEST** same as AND but without saving the result, see also: [1.22 on page 304](#)

**XOR** op1, op2: [`XOR`<sup>6</sup>](#) values.  $op1 = op1 \oplus op2$ . A frequently occurring pattern is `XOR reg, reg`, which implies zeroing of `reg`. See also: [2.5 on page 461](#).

## Less frequently used instructions

**BSF** bit scan forward, see also: [1.29.2 on page 420](#)

**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 the value's sign, extending it to high part of the newly constructed value. See also: [1.28.5 on page 406](#).

Interestingly to know these instructions was initially named as `SEX` (*Sign EXtend*), as Stephen P. Morse (one of Intel 8086 CPU designers) wrote in [Stephen P. Morse, *The 8086 Primer*, (1980)]<sup>7</sup>:

The process of stretching numbers by extending the sign bit is called sign extension. The 8086 provides instructions (Fig. 3.29) to facilitate the task of sign extension. These instructions were initially named SEX (sign extend) but were later renamed to the more conservative CBW (convert byte to word) and CWD (convert word to double word).

**CLD** clear DF flag.

**CLI** (M) clear IF flag

**CMC** (M) toggle CF flag

**CMOVcc** conditional MOV: load if the condition is true. The condition codes are the same as in the Jcc instructions ( [1.6 on page 1008](#)).

<sup>6</sup>eXclusive OR

<sup>7</sup>Also available as <https://archive.org/details/The8086Primer>

**CMPSB/CMPSW CMPSD/CMPSQ** (M) compare byte/ 16-bit word/ 32-bit word/ 64-bit word from the address which is in SI/ESI/RSI with the variable at the address stored in DI/EDI/RDI. Set flags as **CMP** does.

Together with the REP prefix, it is to be repeated in a loop, the counter is stored in the CX/ECX/RCX register, the process will run until the ZF flag is zero (e.g., until the compared values are equal to each other, hence "E" in REPE).

It works like memcmp() in C.

Example from the Windows NT kernel ([WRK v1.2](#)):

Listing 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 a 32-bit word comparison (CMPSD) if the block size is a multiple of 4, or per-byte comparison (CMPSB) otherwise.

**CPUID** get information about the CPU's features. see also: ([1.24.6 on page 370](#)).

**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 a syscall vector. The registers AX/BX/CX/DX/SI/DI were filled with the arguments and then the flow jumped to the address in the Interrupt Vector Table (located at the beginning of the address space). It was popular because INT has a short opcode (2 bytes) and the program which needs some MS-DOS services is not bother to determine the address of the service's entry point. The interrupt handler returns the control flow to caller using the IRET instruction.

The most busy MS-DOS interrupt number was 0x21, serving a huge part of its [API](#). See also: [[Ralf Brown Ralf Brown's Interrupt List](#)], for the most comprehensive interrupt lists and other MS-DOS information.

In the post-MS-DOS era, this instruction was still used as syscall both in Linux and Windows ([6.3 on page 745](#)), but was later replaced by the SYSENTER or SYSCALL instructions.

**INT 3** (M): this instruction is somewhat close to `INT`, it has its own 1-byte opcode (`0xCC`), and is actively used while debugging. Often, the debuggers just write the `0xCC` byte at the address of the breakpoint to be set, and when an exception is raised, the original byte is restored and the original instruction at this address is re-executed.

As of [Windows NT](#), an `EXCEPTION_BREAKPOINT` exception is to be raised when the CPU executes this

## 1. X86

instruction. This debugging event may be intercepted and handled by a host debugger, if one is loaded. If it is not loaded, Windows offers to run one of the registered system debuggers. If [MSVS<sup>8</sup>](#) 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 check integrity of the loaded code.

MSVC has [compiler intrinsic](#) for the instruction: `__debugbreak()`<sup>9</sup>.

There is also a win32 function in kernel32.dll named `DebugBreak()`<sup>10</sup>, which also executes `INT 3`.

**IN** (M) input data from port. The instruction usually can be seen in OS drivers or in old MS-DOS code, for example ([8.5.3 on page 829](#)).

**IRET** : was used in the MS-DOS environment for returning from an interrupt handler after it was called by the INT instruction. Equivalent to `POP tmp; POPF; JMP tmp`.

**LOOP** (M) [decrement](#) CX/ECX/RCX, jump if it is still not zero.

LOOP instruction was often used in DOS-code which works with external devices. To add small delay, this was done:

MOV	CX, nnnn
LABEL:	LOOP
	LABEL

Drawback is obvious: length of delay depends on [CPU speed](#).

**OUT** (M) output data to port. The instruction usually can be seen in OS drivers or in old MS-DOS code, for example ([8.5.3 on page 829](#)).

**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 the stack.

**POPCNT** population count. Counts the number of 1 bits in the value.

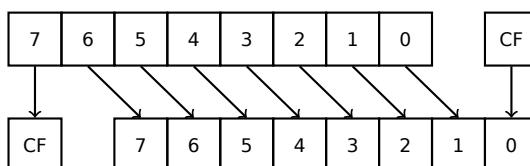
See: [2.6 on page 463](#).

**POPF** restore flags from the stack ([AKA EFLAGS register](#))

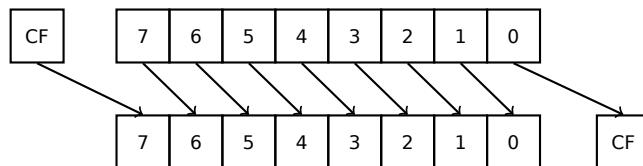
**PUSHA** (M) pushes the values of the (R|E)AX, (R|E)CX, (R|E)DX, (R|E)BX, (R|E)BP, (R|E)SI, (R|E)DI registers to the stack.

**PUSHF** push flags ([AKA EFLAGS register](#))

**RCL** (M) rotate left via CF flag:

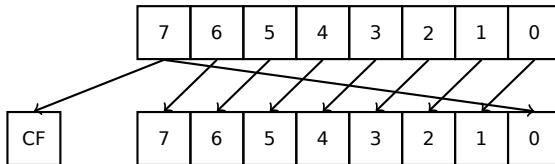


**RCR** (M) rotate right via CF flag:



**ROL/ROR** (M) cyclic shift

ROL: rotate left:

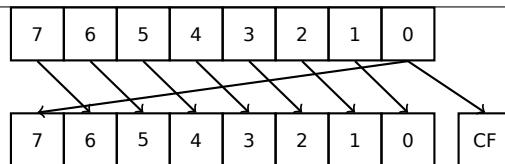


ROR: rotate right:

<sup>8</sup>Microsoft Visual Studio

<sup>9</sup>[MSDN](#)

<sup>10</sup>[MSDN](#)

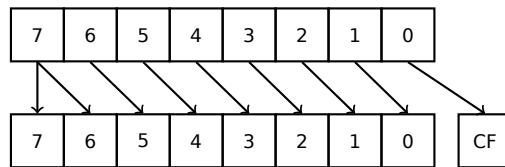


Despite the fact that almost all CPUs have these instructions, there are no corresponding operations in C/C++, so the compilers of these PLs usually do not generate these instructions.

For the programmer's convenience, at least **MSVC** has the pseudofunctions (compiler intrinsics) `_rotl()` and `_rotr()`<sup>11</sup>, which are translated by the compiler directly to these instructions.

**SAL** Arithmetic shift left, synonymous to **SHL**

**SAR** Arithmetic shift right



Hence, the sign bit always stays at the place of the **MSB**.

**SETcc** op: load 1 to operand (byte only) if the condition is true or zero otherwise. The condition codes are the same as in the Jcc instructions ( [.1.6 on page 1008](#)).

**STC** (M) set CF flag

**STD** (M) set DF flag. This instruction is not generated by compilers and generally rare. For example, it can be found in the `ntoskrnl.exe` Windows kernel file, in the hand-written memory copy routines.

**STI** (M) set IF flag

**SYSCALL** (AMD) call syscall ( [6.3 on page 745](#))

**SYSENTER** (Intel) call syscall ( [6.3 on page 745](#))

**UD2** (M) undefined instruction, raises exception. Used for testing.

**XCHG** (M) exchange the values in the operands

This instruction is rare: compilers don't generate it, because starting at Pentium, XCHG with address in memory in operand executes as if it has LOCK prefix ([Michael Abrash, *Graphics Programming Black Book*, 1997 chapter 19]). Perhaps, Intel engineers did so for compatibility with synchronizing primitives. Hence, XCHG starting at Pentium can be slow. On the other hand, XCHG was very popular in assembly language programmers. So if you see XCHG in code, it can be a sign that this piece of code is written manually. However, at least Borland Delphi compiler generates this instruction.

## FPU instructions

-R suffix in the mnemonic usually implies that the operands are reversed, -P suffix implies that one element is popped from the stack after the instruction's execution, -PP suffix implies that two elements are popped.

-P instructions are often useful when we do not need the value in the FPU stack to be present anymore after the operation.

**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., the values in the stack are replaced by their sum

**FCHS** ST(0)=-ST(0)

**FCOM** compare ST(0) with ST(1)

<sup>11</sup>[MSDN](#)

---

**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., the 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 ([.1.3 on page 1004](#)) 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 ([.1.3 on page 1004](#)) into 16-bit op after checking for pending exceptions.

**FNSTCW** op: store FPU control word ([.1.3 on page 1004](#)) into 16-bit op.

**FSTSW** op: store FPU status word ([.1.3 on page 1005](#)) into 16-bit op after checking for pending exceptions.

**FNSTSW** op: store FPU status word ([.1.3 on page 1005](#)) 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., the value in the stack is replaced by the 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., the value in the stack is replaced by the 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 perform just like FCOM, but an exception is raised only if one of the 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)

### Instructions having printable ASCII opcode

(In 32-bit mode).

These can be suitable for shellcode construction. See also: [8.12.1 on page 910](#).

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
a	60	PUSHA
f	61	POPA
	66	(in 32-bit mode) switch to

g	67	16-bit operand size 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	

In summary: AAA, AAS, CMP, DEC, IMUL, INC, JA, JAE, JB, JBE, JE, JNE, JNO, JNS, JO, JP, JS, POP, POPA, PUSH, PUSH, XOR.

## .1.7 npad

It is an assembly language macro for aligning labels on a specific boundary.

That's often needed for the busy labels to where the control flow is often passed, e.g., loop body starts. So the CPU can load the data or code from the memory effectively, through the memory bus, cache lines, etc.

Taken from `listing.inc` (MSVC):

By the way, it is a curious example of the different `NOP` variations. All these instructions have no effects whatsoever, but have a different size.

Having a single idle instruction instead of couple of NOP-s, is accepted to be better for CPU performance.

```
; 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
```

## 2. ARM

.2 ARM

## .2.1 Terminology

ARM was initially developed as 32-bit CPU, so that's why a *word* here, unlike x86, is 32-bit.

**byte** 8-bit. The DB assembly directive is used for defining variables and arrays of bytes.

**halfword** 16-bit. DCW assembly directive —"—.

## .2. ARM

---

**word** 32-bit. DCD assembly directive —"——.

**doubleword** 64-bit.

**quadword** 128-bit.

### .2.2 Versions

- ARMv4: Thumb mode introduced.
- ARMv6: used in iPhone 1st gen., iPhone 3G (Samsung 32-bit RISC ARM 1176JZ(F)-S that supports Thumb-2)
- ARMv7: Thumb-2 was added (2003). was used in iPhone 3GS, iPhone 4, iPad 1st gen. (ARM Cortex-A8), iPad 2 (Cortex-A9), iPad 3rd gen.
- ARMv7s: New instructions added. Was used in iPhone 5, iPhone 5c, iPad 4th gen. (Apple A6).
- ARMv8: 64-bit CPU, [AKA](#) ARM64 [AKA](#) AArch64. Was used in iPhone 5S, iPad Air (Apple A7). There is no Thumb mode in 64-bit mode, only ARM (4-byte instructions).

### .2.3 32-bit ARM (AArch32)

#### General purpose registers

- R0— function result is usually returned using R0
- R1...R12—[GPRs](#)
- R13—[AKA](#) SP ([stack pointer](#))
- R14—[AKA](#) LR ([link register](#))
- R15—[AKA](#) PC (program counter)

R0 - R3 are also called “scratch registers”: the function’s arguments are usually passed in them, and the values in them are not required to be restored upon the function’s exit.

#### 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

#### VFP (floating point) and NEON registers

0..31 <sup>bits</sup>	32..64	65..96	97..127
Q0 <sup>128 bits</sup>			
D0 <sup>64 bits</sup>			D1
S0 <sup>32 bits</sup>	S1	S2	S3

## .2. ARM

---

S-registers are 32-bit, used for the storage of single precision numbers.

D-registers are 64-bit ones, used for the storage of double precision numbers.

D- and S-registers share the same physical space in the CPU—it is possible to access a D-register via the S-registers (it is senseless though).

Likewise, the **NEON** Q-registers are 128-bit ones and share the same physical space in the CPU with the other floating point registers.

In VFP 32 S-registers are present: S0..S31.

In VFPv2 there 16 D-registers are added, which in fact occupy the same space as S0..S31.

In VFPv3 (**NEON** or “Advanced SIMD”) there are 16 more D-registers, D0..D31, but the D16..D31 registers are not sharing space with any other S-registers.

In **NEON** or “Advanced SIMD” another 16 128-bit Q-registers were added, which share the same space as D0..D31.

## .2.4 64-bit ARM (AArch64)

### General purpose registers

The number of registers was doubled since AArch32.

- X0— function result is usually returned using X0
- X0...X7—Function arguments are passed here.
- X8
- X9...X15—are temporary registers, the callee function can use and not restore them.
- X16
- X17
- X18
- X19...X29—callee function can use them, but must restore them upon exit.
- X29—used as **FP** (at least GCC)
- X30—“Procedure Link Register” **AKA LR (link register)**.
- X31—register always contains zero **AKA XZR or “Zero Register”**. It’s 32-bit part is called **WZR**.
- **SP**, not a general purpose register anymore.

See also: [*Procedure Call Standard for the ARM 64-bit Architecture (AArch64)*, (2013)]<sup>12</sup>.

The 32-bit part of each X-register is also accessible via W-registers (W0, W1, etc.).

High 32-bit part	low 32-bit part
X0	
	W0

## .2.5 Instructions

There is a **-S** suffix for some instructions in ARM, indicating that the instruction sets the flags according to the result. Instructions which lacks this suffix are not modify flags. For example **ADD** unlike **ADDS** will add two numbers, but the flags will not be touched. Such instructions are convenient to use between **CMP** where the flags are set and, e.g. conditional jumps, where the flags are used. They are also better in terms of data dependency analysis (because less number of registers are modified during execution).

<sup>12</sup>Also available as <http://go.yurichev.com/17287>

**Conditional codes table**

Code	Description	Flags
EQ	Equal	Z == 1
NE	Not equal	Z == 0
CS AKA HS (Higher or Same)	Carry set / Unsigned, Greater than, equal	C == 1
CC AKA LO (LOwer)	Carry clear / Unsigned, Less than	C == 0
MI	Minus, negative / Less than	N == 1
PL	Plus, positive or zero / Greater than, equal	N == 0
VS	Overflow	V == 1
VC	No overflow	V == 0
HI	Unsigned higher / Greater than	C == 1 and Z == 0
LS	Unsigned lower or same / Less than or equal	C == 0 or Z == 1
GE	Signed greater than or equal / Greater than or equal	N == V
LT	Signed less than / Less than	N != V
GT	Signed greater than / Greater than	Z == 0 and N == V
LE	Signed less than or equal / Less than, equal	Z == 1 or N != V
None / AL	Always	Any

## .3 MIPS

### .3.1 Registers

( O32 calling convention )

#### General purpose registers GPR

Number	Pseudoname	Description
\$0	\$ZERO	Always zero. Writing to this register is like <a href="#">NOP</a> .
\$1	\$AT	Used as a temporary register for assembly macros and pseudo instructions.
\$2 ...\$3	\$V0 ...\$V1	Function result is returned here.
\$4 ...\$7	\$A0 ...\$A3	Function arguments.
\$8 ...\$15	\$T0 ...\$T7	Used for temporary data.
\$16 ...\$23	\$S0 ...\$S7	Used for temporary data*.
\$24 ...\$25	\$T8 ...\$T9	Used for temporary data.
\$26 ...\$27	\$K0 ...\$K1	Reserved for <a href="#">OS</a> kernel.
\$28	\$GP	Global Pointer**.
\$29	\$SP	<a href="#">SP</a> *.
\$30	\$FP	<a href="#">FP</a> *
\$31	\$RA	<a href="#">RA</a> .
n/a	PC	<a href="#">PC</a> .
n/a	HI	high 32 bit of multiplication or division remainder***.
n/a	LO	low 32 bit of multiplication and division remainder***.

#### Floating-point registers

Name	Description
\$F0..\$F1	Function result returned here.
\$F2..\$F3	Not used.
\$F4..\$F11	Used for temporary data.
\$F12..\$F15	First two function arguments.
\$F16..\$F19	Used for temporary data.
\$F20..\$F31	Used for temporary data*.

\*—Callee must preserve the value.

\*\*—Callee must preserve the value ( except in [PIC](#) code).

#### 4. SOME GCC LIBRARY FUNCTIONS

\*\*\*—accessible using the MFHI and MFL0 instructions.

### .3.2 Instructions

There are 3 kinds of instructions:

- R-type: those which have 3 registers. R-instruction usually have the following form:

instruction destination, source1, source2
---

One important thing to keep in mind is that when the first and second register are the same, IDA may show the instruction in its shorter form:

instruction destination/source1, source2
--

That somewhat reminds us of the Intel syntax for x86 assembly language.

- I-type: those which have 2 registers and a 16-bit immediate value.
- J-type: jump/branch instructions, have 26 bits for encoding the offset.

### Jump instructions

What is the difference between B- instructions (BEQ, B, etc.) and J- ones (JAL, JALR, etc.)?

The B-instructions have an I-type, hence, the B-instructions' offset is encoded as a 16-bit immediate. JR and JALR are R-type and jump to an absolute address specified in a register. J and JAL are J-type, hence the offset is encoded as a 26-bit immediate.

In short, B-instructions can encode a condition (B is in fact pseudo instruction for BEQ \$ZERO, \$ZERO, LABEL), while J-instructions can't.

## 4 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

## 5 Some MSVC library functions

ll in function name stands for “long long”, e.g., a 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

## .6. CHEATSHEETS

Multiplication and shift left procedures are the same for both signed and unsigned numbers, hence there is only one function for each operation here. .

The source code of these function can be found in the installed [MSVS](#), in `VC/crt/src/intel/*.asm` `VC/crt/src/intel/*.asm`.

## .6 Cheatsheets

### .6.1 IDA

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 identifier
?	calculator
G	jump to address
:	add comment
Ctrl-X	show references 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.)Text suchen (inkl. Anweisungen, usw.)
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<sup>13</sup>](#) for hiding some often used patterns of inline code. .

### .6.2 OllyDbg

Hot-keys cheatsheet:

hot-key	meaning
F7	trace into
F8	step over
F9	run
Ctrl-F2	restart

### .6.3 MSVC

Some useful options which were used through this book. .

<sup>13</sup>[GitHub](#)

## .6. CHEATSHEETS

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 <i>n</i> -byte boundary
/MD	produced executable will use MSVCR*.DLL MSVCR*.DLL

Some information about MSVC versions: [5.1.1 on page 704](#).

## .6.4 GCC

Some useful options which were used through this book.

option	meaning
-Os	code size optimization
-O3	maximum optimization
-regparm=	how many arguments are to 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
-fno-inline	do not inline functions

## .6.5 GDB

Some of commands we used in this book:

## .6. CHEATSHEETS

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	—
disas/r	disassemble with opcodes
info registers	print all registers
info float	print FPU-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
x/10w ...	at address in RDI
x/s ...	dump 10 memory words
x/i ...	dump memory as string
x/10c ...	dump memory as code
x/b ...	dump 10 characters
x/h ...	dump bytes
x/g ...	dump 16-bit halfwords
finish	dump giant (64-bit) words
next	execute till the end of function
step	next instruction (don't dive into functions)
set step-mode on	next instruction (dive into functions)
frame n	do not use line number information while stepping
info break	switch stack frame
del n	list of breakpoints
set args ...	delete breakpoint
	set command-line arguments

## **Acronyms used**

<b>.6. CHEATSHEETS</b>	
<b>OS</b> Operating System .....	xv
<b>OOP</b> Object-Oriented Programming.....	546
<b>PL</b> Programming language.....	xiii
<b>PRNG</b> Pseudorandom number generator .....	ix
<b>ROM</b> Read-only memory .....	81
<b>ALU</b> Arithmetic logic unit.....	27
<b>PID</b> Program/process ID.....	805
<b>LF</b> Line feed (10 or '\n' in C/C++).....	525
<b>CR</b> Carriage return (13 or '\r' in C/C++).....	525
<b>LIFO</b> Last In First Out .....	30
<b>MSB</b> Most significant bit .....	317
<b>LSB</b> Least significant bit	
<b>RA</b> Return Address .....	22
<b>PE</b> Portable Executable .....	5
<b>SP</b> stack pointer. SP/ESP/RSP in x86/x64. SP in ARM.....	19
<b>DLL</b> Dynamic-link library.....	755
<b>PC</b> Program Counter. IP/EIP/RIP in x86/64. PC in ARM.....	20
<b>LR</b> Link Register .....	6
<b>IDA</b> Interactive Disassembler and debugger developed by Hex-Rays .....	6
<b>IAT</b> Import Address Table.....	755
<b>INT</b> Import Name Table.....	755
<b>RVA</b> Relative Virtual Address .....	755
<b>VA</b> Virtual Address .....	755
<b>OEP</b> Original Entry Point .....	745
<b>MSVC</b> Microsoft Visual C++	
<b>MSVS</b> Microsoft Visual Studio .....	1015

.6. CHEATSHEETS	
<b>ASLR</b> Address Space Layout Randomization.....	636
<b>MFC</b> Microsoft Foundation Classes.....	758
<b>TLS</b> Thread Local Storage.....	xiii
<b>AKA</b> Also Known As .....	30
<b>CRT</b> C runtime library.....	10
<b>CPU</b> Central processing unit .....	xv
<b>FPU</b> Floating-point unit .....	v
<b>CISC</b> Complex instruction set computing .....	20
<b>RISC</b> Reduced instruction set computing.....	2
<b>GUI</b> Graphical user interface .....	751
<b>RTTI</b> Run-time type information.....	561
<b>BSS</b> Block Started by Symbol .....	25
<b>SIMD</b> Single instruction, multiple data .....	196
<b>BSOD</b> Blue Screen of Death .....	745
<b>DBMS</b> Database management systems .....	xiii
<b>ISA</b> Instruction Set Architecture.....	2
<b>HPC</b> High-Performance Computing.....	517
<b>SEH</b> Structured Exception Handling .....	37
<b>ELF</b> Executable file format widely used in *NIX systems including Linux .....	xiii
<b>TIB</b> Thread Information Block.....	284
<b>PIC</b> Position Independent Code: <a href="#">6.4.1 on page 746</a> .....	xiii
<b>NAN</b> Not a Number .....	1006
<b>NOP</b> No OPeration.....	6
<b>BEQ</b> (PowerPC, ARM) Branch if Equal.....	96
<b>BNE</b> (PowerPC, ARM) Branch if Not Equal .....	209
<b>BLR</b> (PowerPC) Branch to Link Register.....	813

.6. CHEATSHEETS	
<b>XOR</b> eXclusive OR.....	1012
<b>MCU</b> Microcontroller unit.....	494
<b>RAM</b> Random-access memory .....	81
<b>GCC</b> GNU Compiler Collection .....	3
<b>EGA</b> Enhanced Graphics Adapter .....	991
<b>VGA</b> Video Graphics Array .....	991
<b>API</b> Application programming interface .....	645
<b>ASCII</b> American Standard Code for Information Interchange .....	450
<b>ASCIIZ</b> ASCII Zero (null-terminated ASCII string ).....	93
<b>IA64</b> Intel Architecture 64 (Itanium): <a href="#">10.5 on page 988</a> .....	464
<b>EPIC</b> Explicitly parallel instruction computing.....	988
<b>OOE</b> Out-of-order execution .....	465
<b>MSDN</b> Microsoft Developer Network.....	639
<b>STL</b> (C++) Standard Template Library: <a href="#">3.19.4 on page 563</a> .....	567
<b>PODT</b> (C++) Plain Old Data Type .....	579
<b>HDD</b> Hard disk drive.....	590
<b>VM</b> Virtual Memory	
<b>WRK</b> Windows Research Kernel .....	721
<b>GPR</b> General Purpose Registers.....	2
<b>SSDT</b> System Service Dispatch Table.....	745
<b>RE</b> Reverse Engineering.....	996
<b>RAID</b> Redundant Array of Independent Disks.....	vii
<b>BCD</b> Binary-coded decimal .....	449
<b>BOM</b> Byte order mark.....	711
<b>GDB</b> GNU debugger.....	48
<b>FP</b> Frame Pointer.....	24

## .6. CHEATSHEETS

<b>MBR</b> Master Boot Record .....	716
<b>JPE</b> Jump Parity Even (x86 instruction) .....	239
<b>CIDR</b> Classless Inter-Domain Routing .....	484
<b>STMFD</b> Store Multiple Full Descending (ARM instruction)	
<b>LDMFD</b> Load Multiple Full Descending (ARM instruction)	
<b>STMED</b> Store Multiple Empty Descending (ARM instruction).....	31
<b>LDMED</b> Load Multiple Empty Descending (ARM instruction).....	31
<b>STMFA</b> Store Multiple Full Ascending (ARM instruction).....	31
<b>LDMFA</b> Load Multiple Full Ascending (ARM instruction) .....	31
<b>STMEA</b> Store Multiple Empty Ascending (ARM instruction) .....	31
<b>LDMEA</b> Load Multiple Empty Ascending (ARM instruction).....	31
<b>APSR</b> (ARM) Application Program Status Register .....	262
<b>FPSCR</b> (ARM) Floating-Point Status and Control Register.....	262
<b>RFC</b> Request for Comments .....	715
<b>TOS</b> Top Of Stack .....	666
<b>LVA</b> (Java) Local Variable Array .....	672
<b>JVM</b> Java virtual machine.....	ix
<b>JIT</b> Just-in-time compilation .....	665
<b>CDFS</b> Compact Disc File System .....	728
<b>CD</b> Compact Disc	
<b>ADC</b> Analog-to-digital converter.....	724
<b>EOF</b> End of file .....	85
<b>DIY</b> Do It Yourself.....	641
<b>MMU</b> Memory management unit.....	635
<b>CPRNG</b> Cryptographically secure Pseudorandom Number Generator .....	953
<b>DES</b> Data Encryption Standard .....	450

## .6. CHEATSHEETS

---

<b>MIME</b> Multipurpose Internet Mail Extensions.....	450
<b>DBI</b> Dynamic Binary Instrumentation.....	523
<b>XML</b> Extensible Markup Language .....	650
<b>JSON</b> JavaScript Object Notation .....	650
<b>URL</b> Uniform Resource Locator.....	4

# Glossary

**heap** usually, a big chunk of memory provided by the OS so that applications can divide it by themselves as they wish. malloc()/free() work with the heap. [31](#), [349](#), [564](#), [566](#), [579](#), [580](#), [596](#), [754](#), [755](#)

**real number** numbers which may contain a dot. this is *float* and *double* in C/C++. [218](#)

**decrement** Decrease by 1. [19](#), [185](#), [203](#), [442](#), [731](#), [855](#), [1008](#), [1011](#), [1015](#)

**increment** Increase by 1. [16](#), [20](#), [185](#), [189](#), [203](#), [209](#), [327](#), [330](#), [442](#), [852](#), [1008](#)

**integral data type** usual numbers, but not a real ones. may be used for passing variables of boolean data type and enumerations. [232](#)

**product** Multiplication result. [99](#), [224](#), [227](#), [408](#), [433](#), [456](#)

**arithmetic mean** a sum of all values divided by their count . [518](#)

**stack pointer** A register pointing to a place in the stack. [9](#), [11](#), [20](#), [31](#), [35](#), [42](#), [54](#), [56](#), [73](#), [100](#), [547](#), [616](#), [732–735](#), [1003](#), [1009](#), [1021](#), [1029](#)

**tail call** It is when the compiler (or interpreter) transforms the recursion (with which it is possible: *tail recursion*) into an iteration for efficiency: [wikipedia](#). [480](#)

**quotient** Division result. [218](#), [220](#), [222](#), [223](#), [227](#), [432](#), [496](#), [520](#)

**anti-pattern** Generally considered as bad practice. [33](#), [76](#), [464](#)

**atomic operation** “*ατομός*” stands for “indivisible” in Greek, so an atomic operation is guaranteed not to be interrupted by other threads. [661](#), [786](#)

**basic block** a group of instructions that do not have jump/branch instructions, and also don't have jumps inside the block from the outside. In IDA it looks just like as a list of instructions without empty lines . [695](#), [991](#), [992](#)

**callee** A function being called by another . [33](#), [46](#), [67](#), [86](#), [98](#), [100](#), [102](#), [421](#), [465](#), [547](#), [616](#), [732–735](#), [737](#), [738](#), [1023](#)

**caller** A function calling another . [5–8](#), [10](#), [30](#), [46](#), [86](#), [98](#), [99](#), [101](#), [109](#), [156](#), [421](#), [468](#), [547](#), [732](#), [734](#), [735](#), [738](#)

**compiler intrinsic** A function specific to a compiler which is not an usual library function. The compiler generates a specific machine code instead of a call to it. Often, it's a pseudofunction for a specific CPU instruction. Read more: ([10.3 on page 986](#)). [1015](#)

**CP/M** Control Program for Microcomputers: a very basic disk OS used before MS-DOS. [910](#)

**dongle** Dongle is a small piece of hardware connected to LPT printer port (in past) or to USB. Its function was similar to a security token, it has some memory and, sometimes, a secret (crypto-)hashing algorithm. [812](#)

**endianness** Byte order: [2.7 on page 463](#). [22](#), [78](#), [347](#), [1012](#)

**GiB** Gibibyte:  $2^{30}$  or 1024 mebibytes or 1073741824 bytes. [15](#)

**jump offset** a part of the JMP or Jcc instruction's opcode, to be added to the address of the next instruction, and this is how the new PC is calculated. May be negative as well. [94](#), [134](#), [1008](#)

**kernel mode** A restrictions-free CPU mode in which the OS kernel and drivers execute. cf. [user mode](#). [1035](#)

**leaf function** A function which does not call any other function. [29](#), [32](#)

**link register** (RISC) A register where the return address is usually stored. This makes it possible to call leaf functions without using the stack, i.e., faster. [32](#), [813](#), [1021](#), [1022](#)

**loop unwinding** It is when a compiler, instead of generating loop code for  $n$  iterations, generates just  $n$  copies of the loop body, in order to get rid of the instructions for loop maintenance. [187](#)

**name mangling** used at least in C++, where the compiler needs to encode the name of class, method and argument types in one string, which will become the internal name of the function. You can read more about it here: [3.19.1 on page 546](#). [546](#), [705](#), [706](#)

**NaN** not a number: a special cases for floating point numbers, usually signaling about errors . [235](#), [257](#), [990](#)

**NEON** AKA “Advanced SIMD”—SIMD from ARM. [1022](#)

**NOP** “no operation”, idle instruction. [731](#)

**NTAPI** API available only in the Windows NT line. Largely not documented by Microsoft. [792](#)

**padding** *Padding* in English language means to stuff a pillow with something to give it a desired (bigger) form. In computer science, padding means to add more bytes to a block so it will have desired size, like  $2^n$  bytes. . [713](#)

**PDB** (Win32) Debugging information file, usually just function names, but sometimes also function arguments and local variables names. [704](#), [757](#), [792](#), [793](#), [800](#), [801](#), [806](#), [893](#)

**POKE** BASIC language instruction for writing a byte at a specific address. [731](#)

**register allocator** The part of the compiler that assigns CPU registers to local variables. [202](#), [307](#), [421](#)

**reverse engineering** act of understanding how the thing works, sometimes in order to clone it. [v](#), [1015](#)

**security cookie** A random value, different at each execution. You can read more about it here: [1.20.3 on page 283](#). [776](#)

**stack frame** A part of the stack that contains information specific to the current function: local variables, function arguments, RA, etc.. [68](#), [98](#), [99](#), [476](#), [776](#)

**stdout** standard output. [22](#), [36](#), [156](#)

**thunk function** Tiny function with a single role: call another function. [23](#), [394](#), [813](#), [822](#)

**tracer** My own simple debugging tool. You can read more about it here: [7.2.1 on page 788](#). [190–192](#), [708](#), [719](#), [722](#), [772](#), [781](#), [895](#), [901](#), [905](#), [906](#), [908](#), [985](#)

**user mode** A restricted CPU mode in which it all application software code is executed. cf. [kernel mode](#). [829](#), [1035](#)

**Windows NT** Windows NT, 2000, XP, Vista, 7, 8, 10. [293](#), [419](#), [615](#), [712](#), [745](#), [756](#), [785](#), [913](#), [1014](#)

**word** data type fitting in [GPR](#). In the computers older than PCs, the memory size was often measured in words rather than bytes. [449](#), [451](#), [452](#), [456](#), [457](#), [569](#), [652](#)

**xoring** often used in the English language, which implying applying the [XOR](#) operation. [776](#), [825](#), [828](#)

# Index

.NET, 761  
0x0BADF00D, 76  
0xCCCCCCCC, 76

Ada, 107  
AES, 865  
Alpha AXP, 2  
AMD, 737  
Angry Birds, 263, 264  
Apollo Guidance Computer, 211  
ARM, 209, 539, 813, 1020  
    Addressing modes, 441  
    ARM mode, 2  
    ARM1, 452  
    armel, 228  
    armhf, 228  
    Condition codes, 137  
    D-registers, 227, 1021  
    Data processing instructions, 498  
    DCB, 20  
    hard float, 228  
    if-then block, 263  
Instructions  
    ADC, 400  
    ADD, 21, 106, 137, 193, 321, 334, 498, 1022  
    ADDAL, 137  
    ADDCC, 175  
    ADDS, 104, 400, 1022  
    ADR, 19, 137  
    ADRcc, 137, 164, 165, 465  
    ADRP/ADD pair, 24, 55, 82, 290, 303, 444  
    ANDcc, 536  
    ASR, 337  
    ASRS, 315, 499  
    B, 54, 137, 138  
    Bcc, 96, 97, 149  
    BCS, 138, 265  
    BEQ, 95, 164  
    BGE, 138  
    BIC, 315, 316, 320, 339  
    BL, 20–24, 137, 445  
    BLcc, 137  
    BLE, 138  
    BLS, 138  
    BLT, 193  
    BLX, 22  
    BNE, 138  
    BX, 104, 177  
    CMP, 95, 96, 137, 165, 175, 193, 334, 1022  
    CSEL, 146, 151, 153, 335  
    EOR, 320  
    FCMPE, 265  
    FCSEL, 265  
    FMOV, 443  
    FMRS, 321  
    IT, 153, 263, 286  
    LDMccFD, 137  
    LDMEA, 31  
    LDMED, 31  
    LDMFA, 31  
    LDMFD, 20, 31, 137  
    LDP, 25  
    LDR, 56, 73, 81, 272, 289, 441  
    LDRB, 365  
    LDRB.W, 209  
    LDRSB, 209  
    LEA, 465  
    LSL, 334, 337  
    LSL.W, 334  
    LSLR, 536  
    LSLS, 273, 321, 536  
    LSR, 337  
    LSRS, 321  
    MADD, 104  
    MLA, 104  
    MOV, 8, 20, 21, 334, 498  
    MOVcc, 149, 153  
    MOVK, 443  
    MOVT, 21, 498  
    MOVT.W, 22  
    MOVW, 22  
    MUL, 106  
    MULS, 104  
    MVNS, 210  
    NEG, 506  
    ORR, 315  
    POP, 19–21, 30, 32  
    PUSH, 21, 30, 32  
    RET, 25  
    RSB, 143, 299, 334, 505  
    SBC, 400  
    SMMUL, 498  
    STMEA, 31  
    STMED, 31  
    STMFA, 31, 57  
    STMFD, 19, 31  
    STMIA, 56  
    STMIB, 57  
    STP, 24, 55  
    STR, 55, 272  
    SUB, 56, 299, 334  
    SUBcc, 536  
    SUBEQ, 210  
    SUBS, 400  
    SXTB, 366  
    SXTW, 303  
    TEST, 202

- 
- TST, 308, 334
  - VADD, 227
  - VDIV, 227
  - VLDR, 227
  - VMOV, 227, 262
  - VMOVGT, 262
  - VMRS, 262
  - VMUL, 227
  - XOR, 143, 321
  - Leaf function, 32
  - Mode switching, 104, 177
  - mode switching, 22
  - Optional operators
    - ASR, 334, 498
    - LSL, 272, 299, 334, 443
    - LSR, 334, 498
    - ROR, 334
    - RRX, 334
  - Pipeline, 175
  - Registers
    - APSR, 262
    - FPSCR, 262
    - Link Register, 20, 32, 54, 177, 1021
    - R0, 107, 1021
    - scratch registers, 209, 1021
    - X0, 1022
    - Z, 96, 1021
  - S-registers, 227, 1021
  - soft float, 228
  - Thumb mode, 2, 138, 176
  - Thumb-2 mode, 2, 176, 263, 264
  - ARM64
    - lo12, 55
  - ASLR, 755
  - AT&T syntax, 12, 37
  - AWK, 721
  - Base address, 755
  - base32, 714
  - Base64, 713
  - base64, 715, 862, 953
  - base64scanner, 714
  - bash, 108
  - BASIC
    - POKE, 731
  - binary grep, 719, 787
  - Binary Ninja, 787
  - Binary tree, 586
  - BIND.EXE, 760
  - BinNavi, 787
  - binutils, 381
  - Binwalk, 947
  - Bitcoin, 658
  - Booth's multiplication algorithm, 217
  - Borland C++, 634
  - Borland C++Builder, 706
  - Borland Delphi, 706, 710, 985, 1016
  - BSoD, 745
  - BSS, 756
  - Buffer Overflow, 275, 282, 776
  - C language elements
    - C99, 110
    - bool, 304
    - restrict, 515
  - variable length arrays, 287
  - const, 9, 81
  - for, 185, 482
  - if, 125, 155
  - Pointers, 67, 73, 110, 385, 421, 626
  - Post-decrement, 441
  - Post-increment, 441
  - Pre-decrement, 441
  - Pre-increment, 441
  - return, 10, 86, 109
  - switch, 154, 155, 164
  - while, 201
  - C standard library
    - alloca(), 35, 287, 465, 767
    - assert(), 292, 716
    - atexit(), 568
    - atoi(), 499, 884
    - calloc(), 845
    - close(), 749
    - exit(), 468
    - fread(), 648
    - free(), 465, 596
    - fseek(), 844
    - ftell(), 844
    - fwrite(), 648
    - getenv(), 885
    - localtime(), 624
    - localtime\_r(), 356
    - longjmp, 653
    - longjmp(), 156
    - malloc(), 349, 465, 596
    - memchr(), 1011
    - memcmp(), 454, 513, 717, 1012
    - memcpy(), 12, 67, 511, 652, 1010
    - memmove(), 652
    - memset(), 267, 510, 905, 1011, 1012
    - open(), 749
    - pow(), 230
    - puts(), 21
    - qsort(), 386
    - rand(), 339, 707, 798, 800, 833
    - read(), 648, 749
    - realloc(), 465
    - scanf(), 66
    - setjmp, 653
    - strcat(), 514
    - strcmp(), 454, 507, 749
    - strcpy(), 12, 509, 834
    - strlen(), 201, 417, 509, 526, 1011
    - strstr(), 468
    - strtok, 212
    - time(), 624
    - tolower(), 850
    - toupper(), 533
    - va\_arg, 519
    - va\_list, 522
    - vprintf, 522
    - write(), 648
  - C++, 896
    - C++11, 579, 740
    - exceptions, 767
    - ostream, 561
    - References, 562
    - RTTI, 560

- 
- STL, 704  
   std::forward\_list, 578  
   std::list, 569  
   std::map, 586  
   std::set, 586  
   std::string, 563  
   std::vector, 579  
 C11, 740  
 Callbacks, 385  
 Canary, 283  
 cdecl, 42, 732  
 Cipher Feedback mode, 867  
 clusterization, 951  
 COFF, 820  
 column-major order, 294  
 Compiler intrinsic, 36, 456, 986  
 Compiler's anomalies, 148, 230, 302, 315, 333, 492, 531, 987  
 Cray-1, 452  
 CRC32, 465, 481  
 CRT, 751, 773  
 CryptoMiniSat, 428  
 CryptoPP, 865  
 Cygwin, 705, 708, 761, 789  
 Data general Nova, 217  
 DES, 408, 421  
 dlopen(), 749  
 dlsym(), 749  
 Donald E. Knuth, 452  
 DOSBox, 913  
 DosBox, 722  
 double, 219, 738  
 Doubly linked list, 462, 569  
 dtruss, 789  
 Duff's device, 493  
 Dynamically loaded libraries, 23  
 Edsger W. Dijkstra, 597  
 EICAR, 910  
 ELF, 79  
 Entropy, 927, 945  
 Error messages, 715  
 fastcall, 14, 34, 66, 306, 733  
 fetchmail, 450  
 FidoNet, 714  
 float, 219, 738  
 Forth, 687  
 FORTRAN, 23  
 Fortran, 294, 515, 597, 705  
 FreeBSD, 717  
 Function epilogue, 30, 54, 56, 137, 365, 721  
 Function prologue, 10, 30, 32, 55, 283, 721  
 Fused multiply-add, 104  
 Fuzzing, 506  
 Garbage collector, 688  
 GCC, 705, 1024, 1026  
 GDB, 29, 48, 51, 282, 394, 395, 788, 1026  
 GeoIP, 946  
 GHex, 787  
 Glibc, 394, 652, 745  
 Global variables, 76  
 GnuPG, 953  
 grep usage, 192, 264, 704, 719, 722, 894  
 Hash functions, 465  
 HASP, 717  
 Heartbleed, 651, 872  
 Heisenbug, 658, 664  
 Hex-Rays, 108, 304, 641  
 Hiew, 93, 134, 709, 715, 757, 758, 761, 787, 985  
 IDA, 87, 381, 514, 698, 712, 787, 788, 975, 1025  
   var\_?, 56, 73  
 IEEE 754, 219, 317, 377, 428, 1000  
 Inline code, 194, 315, 506, 552, 583  
 Integer overflow, 107  
 Intel  
   8080, 209  
   8086, 209, 314, 829  
     Memory model, 623, 990  
   8253, 912  
   80286, 829, 991  
   80386, 314, 991  
   80486, 218  
     FPU, 218  
   Intel 4004, 449  
   Intel C++, 10, 408, 987, 991, 1009  
   Intel syntax, 12, 19  
   iPod/iPhone/iPad, 19  
   Itanium, 988  
 Java, 451, 665  
 John Carmack, 524  
 JPEG, 950  
 jumptable, 169, 176  
 Keil, 19  
 kernel panic, 745  
 kernel space, 745  
 LAPACK, 23  
 LD\_PRELOAD, 749  
 Linker, 81, 546  
 Linux, 307, 746, 897  
   libc.so.6, 306, 394  
 LISP, 629  
 LLDB, 788  
 LLVM, 19  
 long double, 219  
 Loop unwinding, 187  
 LZMA, 948  
 Mac OS Classic, 812  
 Mac OS X, 789  
 Mathematica, 597, 809  
 MD5, 465, 717  
 memfrob(), 864  
 MFC, 758, 885  
 Microsoft Word, 651  
 MIDI, 717  
 MinGW, 705  
 minifloat, 443  
 MIPS, 2, 541, 725, 756, 813, 950  
   Branch delay slot, 8  
   Global Pointer, 25, 299  
   Instructions  
     ADD, 107

ADD.D, 230  
ADDIU, 26, 84, 85  
ADDU, 107  
AND, 317  
BC1F, 267  
BC1T, 267  
BEQ, 97, 139  
BLTZ, 144  
BNE, 139  
BNEZ, 178  
BREAK, 499  
C.LT.D, 267  
DIV.D, 230  
J, 6, 8, 26  
JAL, 107  
JALR, 26, 107  
JR, 167  
LB, 199  
LBU, 198  
LI, 445  
LUI, 26, 84, 85, 230, 320, 445  
LW, 26, 74, 85, 167, 446  
LWC1, 230  
MFC1, 233  
MFHI, 107, 499, 1024  
MFLO, 107, 499, 1024  
MTC1, 383  
MUL.D, 230  
MULT, 107  
NOR, 211  
OR, 29  
ORI, 317, 445  
SB, 198  
SLL, 178, 213, 336  
SLLV, 336  
SLT, 139  
SLTIU, 178  
SLTU, 139, 141, 178  
SRL, 218  
SUBU, 144  
SW, 61  
Load delay slot, 167  
O32, 61, 66, 1023  
Pseudoinstructions  
    B, 196  
    BEQZ, 141  
    L.D, 230  
    LA, 29  
    LI, 8  
    MOVE, 26, 83  
    NEGU, 144  
    NOP, 29, 83  
    NOT, 211  
Registers  
    FCCR, 266  
    HI, 499  
    LO, 499  
MS-DOS, 34, 284, 619, 634, 717, 722, 731, 755, 829, 910, 911, 955, 985, 990, 1000, 1010, 1014, 1015  
    DOS extenders, 991  
MSVC, 1024, 1025  
Name mangling, 546  
Native API, 756  
NEC V20, 913  
Non-a-numbers (NaNs), 257  
Notepad, 948  
objdump, 381, 748, 761, 787  
octet, 450  
OEP, 755, 761  
OllyDbg, 44, 69, 78, 99, 112, 128, 170, 189, 204, 221, 236, 247, 270, 277, 280, 294, 295, 325, 347, 364, 365, 370, 373, 389, 758, 788, 1025  
OOP  
    Polymorphism, 546  
opaque predicate, 543  
OpenMP, 658, 707  
OpenSSL, 651, 872  
OpenWatcom, 705, 734  
Oracle RDBMS, 10, 408, 715, 764, 897, 905, 906, 967, 977, 987, 991  
Page (memory), 419  
Pascal, 710  
PDP-11, 441  
PGP, 713  
Pin, 523  
PNG, 949  
position-independent code, 19, 746  
PowerPC, 2, 25, 812  
Propagating Cipher Block Chaining, 878  
puts() instead of printf(), 21, 71, 108, 135  
Python, 596  
Quake, 524  
Quake III Arena, 385  
rada.re, 13  
Radare, 788  
radare2, 952  
RAID4, 461  
RAM, 81  
Raspberry Pi, 19  
ReactOS, 770  
Recursion, 30, 32, 480  
    Tail recursion, 480  
Register allocation, 421  
Relocation, 23  
Reverse Polish notation, 267  
RISC pipeline, 138  
ROM, 81  
ROT13, 864  
row-major order, 294  
RSA, 5  
RVA, 755  
SAP, 704, 893  
SCO OpenServer, 820  
Scratch space, 736  
Security cookie, 283, 776  
Security through obscurity, 716  
SHA1, 465  
SHA512, 658  
Shadow space, 101, 102, 429  
Shellcode, 542, 745, 755, 911, 1018  
Signed numbers, 126, 454  
SIMD, 428, 513

## INDEX

---

SQLite, 639  
SSE, 428  
SSE2, 428  
Stack, 30, 98, 156  
    Stack frame, 68  
    Stack overflow, 32  
stdcall, 732, 985  
strace, 749, 789  
strtoll(), 876  
Stuxnet, 717  
Syntactic Sugar, 155  
syscall, 306, 745, 789  
Sysinternals, 715, 789  
  
Tagged pointers, 629  
TCP/IP, 464  
thiscall, 546, 547, 734  
Thumb-2 mode, 22  
thunk-functions, 23, 760, 813, 822  
TLS, 284, 740, 756, 761, 1003  
    Callbacks, 743, 761  
Tor, 714  
tracer, 190, 391, 393, 708, 719, 722, 772, 781, 788, 865, 895, 901, 905, 906, 908, 985  
Turbo C++, 634  
  
uClibc, 652  
UCS-2, 450  
UFS2, 717  
Unicode, 710  
UNIX  
    chmod, 4  
    fork, 653  
     getopt, 876  
    grep, 715, 986  
    mmap(), 634  
    od, 787  
    strings, 714, 787  
    xxd, 787, 933  
Unrolled loop, 194, 286, 493, 496, 510  
uptime, 749  
UPX, 952  
USB, 814  
UseNet, 714  
user space, 745  
UTF-16, 450  
UTF-16LE, 710, 711  
UTF-8, 710, 954  
Uuencode, 953  
Uuencoding, 714  
  
VA, 755  
Valgrind, 664  
Variance, 862  
  
Watcom, 705  
win32  
    FindResource(), 629  
    GetOpenFileName, 212  
    GetProcAddress(), 639  
    HINSTANCE, 639  
    HMODULE, 639  
    LoadLibrary(), 639  
    MAKEINTRESOURCE(), 629  
WinDbg, 788  
  
Windows, 785  
    API, 1000  
    IAT, 755  
    INT, 755  
    KERNEL32.DLL, 305  
    MSVCR80.DLL, 387  
    NTAPI, 792  
    ntoskrnl.exe, 897  
    PDB, 704, 757, 792, 800, 893  
    Structured Exception Handling, 37, 762  
    TIB, 284, 762, 1003  
    Win32, 304, 711, 749, 755, 991  
        GetProcAddress, 761  
        LoadLibrary, 761  
        MulDiv(), 456, 808  
        Ordinal, 758  
        RaiseException(), 762  
        SetUnhandledExceptionFilter(), 764  
    Windows 2000, 756  
    Windows 3.x, 615, 991  
    Windows NT4, 756  
    Windows Vista, 755, 792  
    Windows XP, 756, 761, 800  
Wine, 770  
Wolfram Mathematica, 927  
  
x86  
    AVX, 408  
    Flags  
        CF, 34, 1008, 1011, 1012, 1015, 1016  
        DF, 1012, 1016  
        IF, 1012, 1016  
    FPU, 1004  
    Instructions  
        AAA, 1019  
        AAS, 1019  
        ADC, 399, 619, 1008  
        ADD, 9, 42, 98, 501, 619, 1008  
        ADDSD, 429  
        ADDSS, 441  
        ADRcc, 145  
        AESDEC, 865  
        AESENC, 865  
        AESKEYGENASSIST, 868  
        AND, 10, 305, 309, 323, 338, 372, 1008, 1012  
        BSF, 420, 1012  
        BSR, 1012  
        BSWAP, 464, 1012  
        BT, 1012  
        BTC, 319, 1012  
        BTR, 319, 786, 1012  
        BTS, 319, 1012  
        CALL, 9, 31, 538, 760, 878, 945, 1008  
        CBW, 455, 1012  
        CDQ, 407, 455, 1012  
        CDQE, 455, 1012  
        CLD, 1012  
        CLI, 1012  
        CMC, 1012  
        CMOVcc, 138, 145, 147, 149, 153, 465, 1012  
        CMP, 86, 1008, 1019  
        CMPSB, 717, 1012  
        CMPSD, 1012  
        CMPSQ, 1012

---

CMPSW, 1012  
 COMISD, 437  
 COMISS, 441  
 CPUID, 370, 1014  
 CWD, 455, 620, 922, 1012  
 CWDE, 455, 1012  
 DEC, 203, 1008, 1019  
 DIV, 455, 1014  
 DIVSD, 429, 720  
 FABS, 1016  
 FADD, 1016  
 FADDP, 220, 226, 1016  
 FATRET, 332, 333  
 FCHS, 1016  
 FCMOVcc, 259  
 FCOM, 246, 257, 1016  
 FCOMP, 234, 1016  
 FCOMPP, 1016  
 FDIV, 220, 719, 1017  
 FDIVP, 220, 1017  
 FDIVR, 226, 1017  
 FDIVRP, 1017  
 FDUP, 687  
 FILD, 1017  
 FIST, 1017  
 FISTP, 1017  
 FLD, 231, 234, 1017  
 FLD1, 1017  
 FLDCW, 1017  
 FLDZ, 1017  
 FMUL, 220, 1017  
 FMULP, 1017  
 FNSTCW, 1017  
 FNSTSW, 234, 257, 1017  
 FSCALE, 383  
 FSINCOS, 1017  
 FSQRT, 1017  
 FST, 1017  
 FSTCW, 1017  
 FSTP, 231, 1017  
 FSTSW, 1017  
 FSUB, 1017  
 FSUBP, 1017  
 FSUBR, 1017  
 FSUBRP, 1017  
 FUCOM, 257, 1017  
 FUCOMI, 259  
 FUCOMP, 1017  
 FUCOMPP, 257, 1017  
 FWAIT, 218  
 FXCH, 988, 1017  
 IDIV, 455, 496, 1014  
 IMUL, 98, 302, 455, 456, 629, 1008, 1019  
 IN, 538, 829, 912, 1015  
 INC, 203, 985, 1008, 1019  
 INT, 34, 910, 1014  
 INT3, 708  
 IRET, 1014, 1015  
 JA, 126, 258, 455, 1008, 1019  
 JAE, 126, 1008, 1019  
 JB, 126, 455, 1008, 1019  
 JBE, 126, 1008, 1019  
 JC, 1008  
 Jcc, 97, 148  
 JCXZ, 1008  
 JE, 155, 1008, 1019  
 JECXZ, 1008  
 JG, 126, 455, 1008  
 JGE, 126, 1008  
 JL, 126, 455, 1008  
 JLE, 126, 1008  
 JMP, 31, 54, 760, 985, 1008  
 JNA, 1008  
 JNAE, 1008  
 JNB, 1008  
 JNBE, 258, 1008  
 JNC, 1008  
 JNE, 86, 126, 1008, 1019  
 JNG, 1008  
 JNGE, 1008  
 JNL, 1008  
 JNLE, 1008  
 JNO, 1008, 1019  
 JNS, 1008, 1019  
 JNZ, 1008  
 JO, 1008, 1019  
 JP, 235, 913, 1008, 1019  
 JPO, 1008  
 JRCXZ, 1008  
 JS, 1008, 1019  
 JZ, 96, 155, 987, 1008  
 LAHF, 1009  
 LEA, 68, 101, 352, 470, 483, 501, 737, 796, 878, 1009  
 LEAVE, 11, 1009  
 LES, 834, 921  
 LOCK, 785  
 LODSB, 912  
 LOOP, 185, 200, 721, 921, 1015  
 MAXSD, 437  
 MOV, 8, 10, 12, 510, 511, 538, 758, 878, 945, 985, 1010  
 MOVDQA, 411  
 MOVDQU, 411  
 MOVSB, 1010  
 MOVSD, 436, 512, 849, 1010  
 MOVSDX, 436  
 MOVSQ, 1010  
 MOVSS, 441  
 MOVSW, 1010  
 MOVSX, 201, 209, 364–366, 455, 1010  
 MOVSXD, 288  
 MOVZX, 202, 349, 813, 1010  
 MUL, 455, 456, 629, 1010  
 MULSD, 429  
 NEG, 504, 1010  
 NOP, 483, 985, 1010, 1019  
 NOT, 208, 210, 854, 1010  
 OR, 309, 526, 1010  
 OUT, 538, 829, 1015  
 PADD, 411  
 PCMPEQB, 420  
 PLMULHW, 408  
 PLMULLD, 408  
 PMOVMSKB, 420  
 POP, 10, 30, 32, 1010, 1019  
 POPA, 1015, 1019  
 POPCNT, 1015

POPF, 912, 1015  
PUSH, 9, 11, 30, 31, 68, 538, 878, 945, 1010,  
    1019  
PUSHA, 1015, 1019  
PUSHF, 1015  
PXOR, 419  
RCL, 721, 1015  
RCR, 1015  
RET, 5, 7, 10, 32, 283, 547, 616, 985, 1010  
ROL, 333, 986, 1015  
ROR, 986, 1015  
SAHF, 257, 1010  
SAL, 1016  
SALC, 913  
SAR, 337, 455, 517, 921, 1016  
SBB, 399, 1011  
SCASB, 912, 913, 1011  
SCASD, 1011  
SCASQ, 1011  
SCASW, 1011  
SET, 467  
SETALC, 913  
SETcc, 139, 202, 258, 1016  
SHL, 213, 269, 337, 1011  
SHR, 217, 337, 372, 1011  
SHRD, 406, 1011  
STC, 1016  
STD, 1016  
STI, 1016  
STOSB, 495, 1011  
STOSD, 1011  
STOSQ, 510, 1011  
STOSW, 1011  
SUB, 10, 11, 86, 155, 501, 1008, 1012  
SYSCALL, 1014, 1016  
SYSENTER, 746, 1014, 1016  
TEST, 201, 305, 308, 338, 1012  
UD2, 1016  
XADD, 786  
XCHG, 1010, 1016  
XOR, 10, 86, 208, 518, 720, 825, 985, 1012,  
    1019  
MMX, 407  
Prefixes  
    LOCK, 786, 1007  
    REP, 1007, 1010, 1011  
    REPE/REPNE, 1007  
    REPNE, 1011  
Registers  
    AF, 449  
    AH, 1009, 1010  
    CS, 990  
    DF, 652  
    DR6, 1006  
    DR7, 1006  
    DS, 990  
    EAX, 86, 107  
    EBP, 68, 98  
    ECX, 546  
    ES, 921, 990  
    ESP, 42, 68  
Flags, 86, 128, 1003  
FS, 742  
GS, 284, 742, 745  
JMP, 174  
RIP, 748  
SS, 990  
ZF, 86, 305  
SSE, 407  
SSE2, 407  
x86-64, 14, 15, 50, 67, 72, 94, 100, 421, 428, 539,  
    735, 748, 1000, 1006  
Xcode, 19  
XML, 713, 862  
XOR, 867  
Z80, 450  
zlib, 653, 864