Vol. 2, No. 2 (2005)
http://www.CodeBreakers-Journal.com

# Scan of the Month 33

Bilbo*

* Corresponding Author

Received: 01. Jul. 2005, Accepted: 10. Jul. 2005, Published: 15. Jul. 2005

## Abstract

The purpose of this "tutorial" is to provide an in-depth analysis for the executable provided in the present challenge [1]. This analysis will be obtained through a complete reverse engineering of the target, bypassing all "tricks" which have been adopted to make harder the job. We will document the procedures, tools and methods used for this purpose.

***Keywords:*** *Reverse Engineering, Computer Virus, Anti-debugging, Anti-disassembly, Virtual Machines.*

*Editors note: the original version of this article was published at [1].*

# Contents

1. Introduction ..... 3
2. The Approach ..... 3
3. Used Tools ..... 4
4. The Protections ..... 4
5. The Garbage Patterns ..... 5
6. Defeating The Garbage Patterns ..... 6
7. The Matrioska Layers ..... 8
8. Defeating The Matrioska Layers ..... 10
9. The Opcodes Interpreter ..... 11
10. Defeating The Opcodes Interpreter ..... 13
11. The crypto stuff ..... 21
12. The answers ..... 25
Provide a means to "quickly" analyse this uncommon feature. ..... 25
Which tools are the most suited for analysing such binaries, and why? ..... 25
References ..... 26

# 1. Introduction

Let's start with a concise presentation of the binary, just as an appetizer...

The goal of the binary (a simple console program) is to accept from the user an authentication password: if the password is missing or wrong, the string Please Authenticate! will be printf()-ed on the console; if the password is correct, the result will be the one showed at *CommandLine.jpg* (see the attachments), where you will find the 4 possible passwords.

Obviously, there are no traces of these strings inside the executable. This means that the binary has been compressed or encrypted.

Furthermore, even the PE header of the binary is heavily modified... This is the reason why some tools currently used in the cracking scene, like Ollydbg, or even Dependency Walker, by Microsoft, do not work. To be more precise, the Symbol Table and String Table entries are corrupted (with the "DarthPE" value), and so are Loader Flags and the number of Data Directory Entries (0xdfffddde).
These findings came out by a PE header dumper I made when Windows NT 3.51 was introduced. I always use that for sentimental reasons, but there are many others on the net.

On the other hand, the imported functions are just 4: printf() from MSVCRT, and GetCommandLineA(), ExitProcess() and GetTickCount() - the latter never used - from KERNEL32. Often, more and more functions are added dynamically after the executable decompression step, but this is not our case. Now we can start our trip, with the hope we will not have to go too far...

This is what you will find in the remaining text:

1. The Approach
2. Used Tools
3. The Protections
4. The Garbage Patterns
5. Defeating The Garbage Patterns
6. The Matrioska Layers
7. Defeating The Matrioska Layers
8. The Opcodes Interpreter
9. Defeating The Opcodes Interpreter
10. The crypto stuff
11. The answers

# 2. The Approach

Everytime I start some binary exploration (the binary can be run in user mode or in kernel mode, it makes no difference for the reverse engioneering purposes), I have two ways for moving my first steps:

(1) a deadlist approach: with some disassembler I create a full listing of the binary and analyze it.
This method is great only if the binary has not been compressed or encrypted; in fact it has the advantage of showing all the jumps, the subroutines and the external calls a program will perform during its execution. Furthermode a disassembler like IDA is also able to recognize the names of the subroutines inside the static libraries linked with the binary itself.

(2) a live approach: with a good debugger, we step inside the assembly code, jumping over whole blocks of instructions when they are obvious. This method is great because we can inspect the contents of the registers and of the memory in the same moment they are altered, and we can see what is the way taken by the instructions flow.

Even better, we can create a deadlist with a disassembler, then with a debugger we can step inside the code as far as we analyze the lines of assembly, writing down remarks and comments on the listing file. We use, in other words, a mixture of the two approaches.

For the current analysis I have exclusively followed the live approach, since the deadlist approach is of no use: first, because the most interesting part of the code is encrypted, and second because some garbage code has been introduced in order to fool a decompiler, as we will see.

# 3. Used Tools

Consequently, the only tool required for this analysis is a debugger, a good one. What I mean for good? It must be able of single stepping even when the code is crazy (see below), and it must be possible to ignore some kind of exceptions: in the present case, the Access Violation exceptions are heavily used and they must not interfere with our stepping through the code.

I have used Microsoft Visual Studio 6.0.
We can start it, even from a command promt, running "msdev" with the name of the executable as argument. Then we must remember to remove the processing of the Access Violation Exceptions. After a single-step, we go to the Debug menu -> Exceptions, and we remove the Access Violation processing. The only drawback of Visual Studio debugger is that it is nearly impossible to set hardware breakpoints with it (those breakpoints, in other words, which are not set by an INT 3 software opcode in place of some other code, but using directly the special debug registers of the CPU). In the current target, though, the use of hardware breakpoints is detected and nullified, as we will see.
The ability of cutting and pasting whole groups of instructions comes handy to store our findings in an ordered file.

# 4. The Protections

The program offers three primary kinds of protections against disassembling and live stepping:

(1) The Garbage Patterns:

Starting from the very first instructions of the executable, we will meet instructions which move strange values in the registers and jump in the middle of other opcodes. This is aimed exclusively to make more difficult to follow the code and to confuse the disassemblers.

(2) The Matrioska Layers:

The code evolve in layers one inside the other, just like the russian doll or Chinese boxes. In every couple of layers, the more external one contains the code to decrypt the more internal one, and so on.

(3) The Opcodes Interpreter

The most internal layer leaves away the assembly code and continues execution through special opcodes. Every opcode (which in the present case is a couple of bytes) has an unambiguous meaning (such as PUSH, POP, LOAD), interpreted by a built-in engine.
Visual Basic, Java, many installer scripts and videogames all follow this technique. This give more power and more conciseness to the code, but makes also more difficult its interpretation by humans, as far as the opcodes are not documented. Fortunately, in the present binary, all the possible opcodes are limited (only 37, and not all in use), and two or three hours of reversing are enough to catch their meanings.

# 5. The Garbage Patterns

So, let start from beginning!. There are two types of repetitive patterns in these first layers.

```
00DE2000    pushad  ; save registers
      ; following trick is to load in EBP the offset between the expected load
      ; address DE2000 and the effective one; this trick is necessary for DLLs
      ; but useless in an executable like this, because the Operating System
      ; make us sure that the base address cannot be relocated
00DE2001    call        00DE2006
00DE2006    pop         ebp  ; DE2006
00DE2007    mov         eax,ebp
00DE2009    sub         eax,6  ; DE2000
00DE200C    sub         ebp,0DE2006h  ; 0
      ; -----------garbage pattern-----------
00DE2012   pushad
.
.
.
00DE2288   popad
      ; ------------------------------------
```

All the instructions between a PUSHAD opcode (excluding the very first PUSHAD), opcode 0x60, and a POPAD (opcode 0x61) will be called garbage pattern.
They will contain MOVs, LEAs, all the possible one-byte prefixes like segment overrides and so on, and JUMPS $+3 (the displacement is fixed, the corresponding opcode is EB 01).

The problem with this pattern is that if, from one point of view, is well delimited between opcodes 0x60 and 0x61, on the other hand the ending opcode 0x61 can be found as second or following bytes of some instruction inside the garbage, and, conversely, the initial opcode 0x60 can be found inside the second or following bytes of some not-garbage instruction.

But soon after this first kind of pattern, another one comes...

```
; -----------call pattern (67h bytes)--------------
00DE2289   pushad
00DE228A   call        00DE22D7
      ; exception handler
00DE228F   mov         ecx,dword ptr [esp+0Ch]  ; CONTEXT
00DE2293   add         dword ptr [ecx+0B8h],2  ; EIP skip offending instruction
             ; clear debug registers
      xor         eax,eax
      mov         dword ptr [ecx+4],eax
      mov         dword ptr [ecx+8],eax
      mov         dword ptr [ecx+0Ch],eax
```

```
       mov         dword ptr [ecx+10h],eax
       mov         dword ptr [ecx+14h],eax
       mov         dword ptr [ecx+18h],155h

       mov         eax,dword ptr [ecx+0B0h]  ; EAX: first RDTSC read
       push        eax  ; put first read on stack
       cpuid  ; useless???
       rdtsc  ; do second read
       sub         eax,dword ptr [esp]  ; delta
       add         esp,4  ; clean stack
       cmp         eax,0E0000h  ; check delta
       ja          00DE22CD  ; modify EIP
             ; return 0
       xor         eax,eax
       ret
             ; modify EIP and return 0
00DE22CD   add         dword ptr [ecx+0B8h],63h
       xor         eax,eax
       ret

             ; set exception handler to return address
00DE22D7   xor         eax,eax
       push        dword ptr fs:[eax]
       mov         dword ptr fs:[eax],esp

       cpuid
       rdtsc  ; first RDTSC read
       xor         ebx,ebx

       pop         dword ptr [ebx]  ; generate exception

             ; clear frame
00DE22E7   pop         dword ptr fs:[0000h]
00DE22ED   add         esp,4
00DE22F0   popad
```

We will call it a "call pattern": even this is introduced by opcode 0x60 and terminated by opcode 0x61, but soon after 0x60 there is a CALL to a subroutine which will set an exception handler to the return address.

All this stuff is aimed to two purposes: cleaning possible hardware breakpoints the user has set until now; and checking the time between two RDTSC instructions. If this time is too high, there is a good chance that the whole binary is run through some emulator or single-stepping debugger.
This second pattern is a lot easier to identify because it is fixed length.

# 6. Defeating The Garbage Patterns

Initially I started attaining to my live approach: I single-stepped through the instructions of each garbage pattern up to the end of it. But soon I found out that this approach was very time and patience consuming, so I coded a simple filter to change all the patterns in NOPs (code 0x90): it is a lot easier to skip a bunch of NOPs without being obliged to single step every instruction! The filter simply jumps from instruction to instruction, taking into account each instruction length. The code of this filter is included with the name *no_garbage.c*.

In the filter there are two ways for recognizing the instruction lengths: if we are inside the garbage pattern, we use a limited set of opcodes, and - most important - we assume that the length of JMP $+3 is 3 instead of two bytes, just to take effective the jump. Out of the pattern, we use a common

instruction-counting algorithm. I found it at this link [2], but I had to correct two bugs for some instructions.

WARNING: this filter must not be used before unsealing all the layers, to avoid an incorrect decryption of them (there are some data bytes in the outer layers which make the hunting algorithm to lost the instructions synchronism).
We will apply this filter only at the inner layer, and we will find as many as 290 layers!

But not only the hardware breakpoints are checked and possibly neutralized.
Some attempt to detect software breakpoints at the start of the external API calls is also performed (an API tracer could automatically put them).
Let's go on in the code from where we leave it (address DE22F0, at the end of the call pattern), removing for better readability all the garbage and call patterns...

```
      ; E26441 <- DE2000 used as reference in the following decrypting loops
00DE2603   mov         dword ptr [ebp+0E26441h],eax

      ; first API checking
00DE2950   mov         eax,0DE100Dh
      ; EAX<-[DE100F] = E28060 pointer to resolved external API
00DE2C2C   mov         eax,dword ptr [eax+2]
00DE2EF1   mov         eax,dword ptr [eax]  ; E28060 has GetCommandLineA
      ; look for CC in first 4 bytes
00DE2F5B   mov         edi,eax
00DE2F5D   mov         ecx,4
00DE31FA   mov         eax,660h
00DE31FF   shr         eax,3
00DE3202   repne scas  byte ptr [edi]
00DE3204   test        ecx,ecx
00DE3206   je          00DE320C  ; and to DE3274
      ; a CC (sw breakpoint) was found
00DE3208   rdtsc
00DE320A   push        eax
00DE320B   ret

      ; second API checking
00DE3274   mov         eax,0DE1001h
00DE3558   mov         eax,dword ptr [eax+2]  ; E28054
00DE355B   mov         eax,dword ptr [eax]  ; E28054 has printf (MSVCRT.DLL)
00DE37F1   mov         edi,eax
00DE3A90   mov         ecx,4
00DE3A95   mov         eax,660h
00DE3A9A   shr         eax,3  ; CC
00DE3D2F   repne scas  byte ptr [edi]
00DE3D31   test        ecx,ecx
00DE3D33   je          00DE3D39  ; and to DE4048
00DE3D35   rdtsc
00DE3D37   push        eax
00DE3D38   ret

      ; third API checking
00DE4048   mov         eax,0DE1013h
00DE430B   mov         eax,dword ptr [eax+2]  ; E28064
00DE430E   mov         eax,dword ptr [eax]  ; ExitProcess
00DE45CF   mov         edi,eax
00DE45D1   mov         ecx,4
00DE45D6   mov         eax,660h
00DE45DB   shr         eax,3
00DE4884   repne scas  byte ptr [edi]
00DE4886   test        ecx,ecx
00DE4888   je          00DE488E  ; and to DE517D
00DE488A   rdtsc
```

```
00DE488C   push        eax
00DE488D   ret
```

# 7. The Matrioska Layers

At this point of the program (address DE517D) a more challenging situation awaits us. The following schema is repeated for every layer, one inside the other:

```
00DE517D   xor         edi,edi  ; loop index

       ; 4 loops for EDI from 1 to 4
00DE517F   inc         edi
       ; ESI <- bias to add to DE2000 to jump to the snippet to be executed at this
loop
       ;            E2641D has 4440B
       ;            E26421 has 443F6
       ;            E26425 has 443FE
       ;            E26429 has 4435B
00DE5180   mov         esi,dword ptr [ebp+edi*4+0E26419h]
       ; EBX <- encoded snippet return address to be pushed on stack
       ;            E26431 has FECE5A48
       ;            E26435 has FE686EDA
       ;            E26439 has FF2D68F4
       ;            E2643D has FF63FF58
00DE5187   mov         ebx,dword ptr [ebp+edi*4+0E2642Dh]
       ; FECE5A48+DE2000 = FFAC7A48
       ; FE686EDA+DE2000 = FF468EDA
       ; FF2D68F4+DE2000 = B88F4
       ; FF63FF58+DE2000 = 421F58
00DE53FE   add         ebx,dword ptr [ebp+0E26441h]
00DE5404   push        ebx

00DE5405   call        00DE5420
00DE540A   jmp         00DE540D
00DE540C

       ; complete ESI with address of the snippet to execute at current loop
       ;            4440B+DE2000 = E2640B
       ;            443F6+DE2000 = E263F6
       ;            443FE+DE2000 = E263FE
       ;            4435B+DE2000 = E2635B
00DE540D   add         esi,dword ptr [ebp+0E26441h]
00DE5413   jmp         esi

       ; common snippets return address
00DE5415   cmp         edi,4  ; check end of loops
00DE5418   jne         00DE517F  ; continue
00DE541E   jmp         00DE5423  ; jump to decrypted area

00DE5420   pop         ebx  ; throw away the return address
00DE5421   jmp         00DE540D  ; snippet selection code

       ; here the first encrypted layer starts
00DE5423
.
.
.
       ; here the first encrypted layer ends
       ; ------------------------------------------------
       ; executed at fourth loop: decode DE5423-E2635A
       ; a loop per byte
```

```
00E2635B   xor         byte ptr [eax],cl
00E2635D   inc         eax
00E2635E   dec         ecx  ; up to 1 included
00E2635F   test        ecx,ecx
00E26361   jne         00E2635B

       ; check for CC at DE541E
00E26363   lea         eax,[ebp+0DE541Eh]
00E26369   cmp         byte ptr [eax],0CCh
00E2636C   jne         00E26372  ; must jump
00E2636E   rdtsc
00E26370   push        eax
00E26371   ret

       ; this is similar to the call pattern: set an exception handler
       ; at the return address and cause the handler to be invoked
00E26372   call        00E263D3
00E26377

       ; exception handler
       ; simply clear debug registers and correct snippet return address
00E26378   mov         edi,dword ptr [esp+0Ch]  ; context
00E2637C   add         dword ptr [edi+0B8h],2  ; skip offending instruction
       ; reset debug registers
00E26383   xor         eax,eax
00E26385   lea         edi,[edi+4]
00E26388   stos        dword ptr [edi]  ; +4
00E26389   stos        dword ptr [edi]  ; +8
00E2638A   stos        dword ptr [edi]  ; +C
00E2638B   stos        dword ptr [edi]  ; +10
00E2638C   stos        dword ptr [edi]  ; +14
00E2638D   mov         ax,1AAh
00E26391   xor         al,0FFh  ; EAX <- 155
00E26393   stos        dword ptr [edi]  ; +18
00E26394   mov         eax,dword ptr [edi+0A8h]  ; +A8+1C=+C4 saved ESP
       ; saved ESP+20(after PUSHAD)+8 is encoded snippet return address 421F58
       ; 421F58+187A3F0 = 1C9C348 (yet encoded)
00E2639A   add         dword ptr [eax+28h],187A3F0h
00E263A1   mov          eax,dword ptr [edi+94h]  ; +94+1C=+B0 saved EAX (first rdtsc
read)
00E263A7   push        eax
00E263A8   cpuid  ; useless???
00E263AA   rdtsc  ; second rdtsc read
00E263AC   sub         eax,dword ptr [esp]  ; delta
00E263AF   add         esp,4
00E263B2   cmp         eax,0E0000h
00E263B7   ja          00E263C9  ; must not jump
00E263B9   mov         eax,dword ptr [edi+0A8h]  ; saved ESP
       ; common snippets return address 1C9C348-EB6F33 = DE5415
00E263BF   sub         dword ptr [eax+28h],0EB6F33h
       ; return 0
00E263C6   sub         eax,eax
00E263C8   ret
       ; ko
00E263C9   add         dword ptr [edi+9Ch],32h
       ; return 0
00E263D0   sub         eax,eax
00E263D2   ret

       ; sub_E263DE
       ; set exception handler to return address+1
00E263D3   inc         dword ptr [esp]
00E263D6   push        dword ptr fs:[0000h]
00E263DC   mov         dword ptr fs:[0000h],esp
00E263E2   pushad
00E263E3   cpuid
```

```
00E263E5   rdtsc  ; first read in EAX
00E263E7   xor         ebx,ebx
00E263E9   mov         dword ptr [ebx],ebx  ; generate exception
00E263EB   popad
       ; remove exception frame
00E263EC   pop         dword ptr fs:[0000h]
00E263F2   add         esp,4
00E263F5   ret

       ; ------------------------------------------------
       ; executed at second loop: no op
00E263F6   add         dword ptr [esp],197C53Bh
00E263FD   ret  ; go to DE5415

       ; ------------------------------------------------
       ; executed at third loop: ECX<-40F38 number of bytes to decode (layer len)
00E263FE   mov         ecx,40F38h
00E26403   add         dword ptr [esp],0D2CB21h
00E2640A   ret  ; goto DE5415

       ; ------------------------------------------------
       ; executed at first loop: EAX<-DE5423 start of encrypted layer
00E2640B   lea         eax,[ebp+0DE5423h]
00E26411   add         dword ptr [esp],131D9CDh  ; FFAC7A48+131D9CD=DE5415
00E26418   ret  ; go to DE5415
```

From the above code we can see that each layer is composed of three parts:

(1) a header: the code between addresses DE517D and DE5423. It contains the loop which will call subsequentely the four decoding snippets at the end of the layer.
(2) a body: the encrypted layer.
(3) a trailer: the code between addresses E2635B and E26418. It contains the four decoding snippets (the snippet called at second loop is always a no operation).

The encryption algorithm is rather simple. The effective byte mutations are performed inside the fourth snippet: all the bytes are xored with a decreading value; this value starts from the number of bytes in the layer and ends to 1.

# 8. Defeating The Matrioska Layers

Even in this case, I initially started stepping from a layer to another manually. I had the patience to go up to the sixth layer. Then I was tired, and furthermore I thought that, for a serious analysis, I must be able to interrupt the job and start it again on another time from the point at which I arrived the previous time. So I wrote a second filter to statically decrypt all the layers.

This is not difficult, because the final snippets are always the same; only some garbage loop may be added inside them. The program, whose sources are included (see *no_matrioska_c.txt*), starts from the bottom of the layer and looks for a known opcode; in this case only two bytes where enough to search: 0xC3, the RET from the second snippet (address E263FD in the first layer shown above), followed by 0xB9.

```
00E263FD C3                   ret
00E263FE B9 38 0F 04 00       mov         ecx,40F38h
00E26403 81 04 24 21 CB D2 00 add         dword ptr [esp],0D2CB21h
00E2640A C3                   ret
00E2640B 8D 85 23 54 DE 00    lea         eax,[ebp+0DE5423h]
```

Two bytes below (soon following C3 B9) we find the length of the layer, and 16 bytes below we find the start of the layer, biased by the base address DE0000. These two values are enough to decode the whole layer. Then we go to the bottom of this freshly decoded layer and start the hunt for a new one...
We check also that at offset +13 there is a C3, the RET from the third snippet: when this is no more true, we have finished unlayering the binary.

The filter found in this way as many as 174 layers!. The code of this filter is included with the name *no_matrioska.c*. The most internal one start at DE8653 and ends at E1BC77.
Now we can apply the second filter - the garbage remover - to it, and also the 290 garbage patterns are gone!

If we finally want to obtain an executable which has the same behaviour of the original one, we simply need to patch the unlayered binary with a jump to skip all the unlayering code. I put the jump, using a common binary editor, at address DE517D (the header of the first layer), in order to go to address DE8653 (the start of the most internal layer): we need to patch E9 (the relative jump opcode) xx xx xx xx at file offset 517D, where xx xx xx xx is a long obtained from DE8653(destination) - DE5182(address following the jump) = D1 34 00 00. Try to run this new binary, which I have renamed *_0x90.exe* and included in the files submitted for reference: it must work exactly as the original one!.

# 9. The Opcodes Interpreter

Now, we set a breakpoint at the start of the most internal layer, restart the new binary *_0x90.exe*, take something to drink, and we are ready to analyze the new evil invention of Nicolas.

After single stepping through few instructions (not including the impressive amount of NOPs, obtained from the garbage patterns, and call patterns) we can infere that we are in the presence of some interpreting engine. Let's see (as usual, I have stripped out all the garbage and call patterns).
The following lines initialize the globals area, a sort of 5 registers used by the opcodes, plus a NotZero boolean Flag:

```
00DE8653   push        0E1BA3Dh  ; start of opcodes
00DE86C0   push        0DE872Fh  ; exit address
00DE872D   jmp         00DE874E  ; skip the exit snippet

       ; exit snippet
00DE872F   push        0
00DE8731   call        00DE1013  ; jump [ExitProcess]

       ; store the encoded exit address
00DE874E   xor         dword ptr [esp],48415830h  ; DE872F -> 489FDF1F
00DE89FC   pop         dword ptr ds:[0E1BC73h]  ; <- encoded end addr

       ; initialize the globals area with a funny pattern:
       ; six longs, last will be used as Not Zero Flag
00DE8CCC   mov         dword ptr ds:[0DE8736h],6C697645h  ; Evil
00DE8F79   mov         dword ptr ds:[0DE873Ah],73614820h  ; Has
00DE920E   mov         dword ptr ds:[0DE873Eh],206F4E20h  ; No
```

```
00DE94AA   mov         dword ptr ds:[0DE8742h],6E756F42h  ; Boun
00DE977C   mov         dword ptr ds:[0DE8746h],69726164h  ; dari
00DE99E9   mov         dword ptr ds:[0DE874Ah],21207365h  ; es !

       ; load in ESI the start address of opcodes
00DE9C9A   mov         esi,dword ptr [esp]  ; 0E1BA3Dh
00DE9C9D   pop         eax  ; clear the stack
```

Now it is time to analyze the code which will fetch the next opcode and will transfer control to the specific execution snippet.

Just to make things more amusing, the whole operation is mode under Access Violation exception!. Every opcode is made of two bytes. The first byte select an appropriate jump table, and the second one is the index into the selected jump table. Fortunately, the used opcodes are not too many!.

```
       ; get the first byte of the opcode
00DE9C9E   movzx       eax,byte ptr [esi]
       ; switch to the correct address table
00DE9F5D   mov         edi,dword ptr [eax*4+0E1B991h]
       ; get the second byte of the opcode
00DEA213   movzx       eax,byte ptr [esi+1]  ; 2 get second byte

       ; this recall us the 'call pattern', isn't it?
00DEA4B0   pushad
00DEA4B1   call        00DEA4EA  ; set exception handler to return addr

       ; exception handler
00DEA4B6   mov         ecx,dword ptr [esp+0Ch]  ; context
       ; reset debug registers
00DEA4BA   xor         eax,eax
00DEA4BC   mov         dword ptr [ecx+4],eax
00DEA4BF   mov         dword ptr [ecx+8],eax
00DEA4C2   mov         dword ptr [ecx+0Ch],eax
00DEA4C5   mov         dword ptr [ecx+10h],eax
00DEA4C8   mov         dword ptr [ecx+14h],eax
00DEA4CB   mov         dword ptr [ecx+18h],155h

00DEA4D2   mov         eax,dword ptr [ecx+0B0h]  ; saved EAX: second byte
00DEA4D8   mov         edi,dword ptr [ecx+9Ch]  ; saved EDI: address table
00DEA4DE   mov         eax,dword ptr [edi+eax*4]  ; snippet address
00DEA4E1   mov         dword ptr [ecx+0B8h],eax  ; set it to saved EIP
       ; return 0
00DEA4E7   xor         eax,eax
00DEA4E9   ret

       ; sub DEA4EA
       ; set exception handler
00DEA4EA   push        dword ptr fs:[0000h]
00DEA4F0   mov         dword ptr fs:[0000h],esp
00DEA4F6   xor         ebx,ebx

       ; generate exception
00DEA4F8   pop         dword ptr [ebx]

       ; remove the exception frame
00DEA4FA   pop         dword ptr fs:[0000h]
00DEA500   add         esp,4
00DEA503   popad
```

So we now know where the opcodes start (address E1BA3D) and where are the pointers to the address tables (address E1B991).
We don't yet know the length of each instruction (two byte for the opcode, but without taking into account the arguments, which can be one, or more, or none), and what each opcode means. The only way to know all that is to go on in our reversing!.

# 10. Defeating The Opcodes Interpreter

Unfortunately, I don't know a fast solution for this kind of stuff. I have adopted the live approach also this time.
I have put a breakpoint at all the entry points in the jump tables and I have started watching what happened. This can be time consuming when there are loops and many repeated opcodes, so some breakpoints may need to be disabled.

Nicolas built in another nice opportunity: the code <02 07>!
Let's see first the address tables, whose pointers we find at address E1B991.

```
00 - 00E1B9A9
       00 - 00DEA4FA PUSH ENCODED CONSTANT (^37195411h): 2+4 bytes
       01 - 00DEC36A PUSH ENCODED CONSTANT (+=ADD01337h): 2+4 bytes
       02 - 00E173B7 PUSH GLOBAL WITH ENCODED-INDEX (^47): 2+1 bytes
       03 - 00E1B0C1 POP TO GLOBAL WITH ENCODED-INDEX (^66): 2+1 bytes
       04 - 00E14CC1 STACK ADJUST +BYTE^45: 2+1 bytes

01 - 00E1B9BD
       00 - 00DF1A7B unused: 2 bytes
       01 - 00DF442F garbled
       02 - 00DF59D2 unused: 2 bytes
       03    -    00DF74EF    XOR:    <01    03>    srcidx    xortype    dstidx    =>
global[dstidx]^=global[srcidx]
              where xortype may be 00(byte XOR), 01(word XOR) or 02(long XOR): 2+3
bytes
       04 - 00E181A6 global[idx-3]+=constant: 2+1+4 bytes
       05 - 00E1A05B global[idx-2]-=constant: 2+1+4 bytes
       06 - 00E1226D global[idx-5]&=constant: 2+1+4 bytes
       07 - 00E14128 global[idx-4]|=constant: 2+1+4 bytes
       08 - 00E016ED global[idx]^=constant: 2+1+4 bytes
       09 - 00E05353 global[secondidx-3] += global[firstidx-1]: 2+1+1 bytes
       0A - 00DFB948 global[secondidx-1] CMP global[firstidx-2]: 2+1+1 bytes

02 - 00E1B9F5
       00 - 00DEE16F EXIT: 2 bytes
       01 - 00DEE947 CALL EXTERNAL, result in global[0]: 2+4 bytes
       02 - 00DFDCA3 ASSIGN ENCODED LONG TO GLOBAL[idx]: 2+4+1 bytes
       03 - 00E09AB5 DEC global[idx]: 2+1 bytes
       04 - 00E0D4BC INC global[idx]: 2+1 bytes
       05 - 00E11477 CLEAR global[idx]: 2+1 bytes
       06 - 00E15B11 SCAN idx n_word: 2+1+2 bytes
              SCAN  MEM  POINTED  BY  GLOBAL[idx]  FOR  VAL  IN  GLOBAL[0],  FOR  AT  MOST
n_word BYTES
       07 - 00E1B92B SET BREAKPOINT: 2 bytes
       08 - 00E0E816 LONG-DEREFERENCE
              GLOBAL[dstidx] <- *GLOBAL[srcidx]: 2+1+1 bytes
       09 - 00E0C6C9 BSWAP GLOBAL[idx]: 2+1 bytes
       0A - 00E0B5E7 BYTE-DEREFERENCE
              GLOBAL[dstidx] <- (BYTE)*GLOBAL[srcidx]: 2+1+1 bytes
       0B - 00E03D6A WORD-DEREFERENCE
              GLOBAL[dstidx] <- (WORD)*GLOBAL[srcidx]: 2+1+1 bytes
```

```
03 - 00E1BA25
      00 - 00DFEFDC CMP 2 VALUES ON STACK AND CLEAN IT:
             JUMP AT ADDRESS-31337 IF EQUAL: 2+4 bytes

04 - 00E1BA29
      00 - 00E034FD JMP ADDRESS-DEADEAD: 2+4 bytes
      01 - 00E07989 JNZ ADDRESS+E1BA3E: 2+4 bytes
      02 - 00E0F8A7 JZ ADDRESS+E1BA41: 2+4 bytes

05 - 00E1BA35
      00 - 00E13604 CALL ADDRESS: 2+4 bytes
      01 - 00E15554 RET: 2 bytes
```

Let see now the (depurated and commented) snippets for each of these opcodes.
The first three instructions are common to every opcode and are intended to remove the last exception frame from the stack and balance the initial PUSHAD.

```
      ; OPCODE <00 00>
00DEA4FA   pop         dword ptr fs:[0000h]
00DEA500   add         esp,4
00DEA503   popad
00DEA7BF   mov         eax,dword ptr [esi+2]  ; get a long
00DEAA79   xor         eax,37195411h  ; decrypt it
00DEAD23   push        eax  ; push on stack
00DEAFEA   mov         eax,0FFFFFF3Fh
00DEB299   not         eax  ; C0
00DEB553   shr         eax,5  ; 6: instruction length
00DEB829   lea         esi,[esi+eax]  ; update pointer
...

      ; OPCODE <00 01>
00DEC36A   pop         dword ptr fs:[0000h]
00DEC370   add         esp,4
00DEC373   popad
00DEC633   mov         eax,dword ptr [esi+2]  ; get a long
00DEC8CD   add         eax,0ADD01337h  ; decrypt it
00DECB8A   push        eax  ; push on stack
00DECE37   mov         eax,0FFFFF5D1h
00DED113   not         eax  ; 0A2E
00DED3BB   xor         eax,0A28h  ; 6: instruction length
00DED680   lea         esi,[esi+eax]  ; update pointer
...

      ; OPCODE <00 02>
00E173B7   pop         dword ptr fs:[0000h]
00E173BD   add         esp,4
00E173C0   popad
00E1766E   movzx       eax,byte ptr [esi+2]  ; get a byte
00E17920   xor         eax,47h  ; decode the global index
00E17BF4   mov         eax,dword ptr [eax*4+0DE8736h]  ; global[index]
00E17EB7   push        eax  ; push on stack
00E18144   mov         ebx,0Ah
00E18149   xor         ebx,9  ; 3: instruction length
00E1814C   add         esi,ebx  ; update pointer
...

      ; OPCODE <00 03>
00E1B0C1   pop         dword ptr fs:[0000h]
00E1B0C7   add         esp,4
00E1B0CA   popad
00E1B372   movzx       eax,byte ptr [esi+2]  ; get a byte
00E1B376   xor         eax,66h  ; decode the global index
00E1B64A   pop         dword ptr [eax*4+0DE8736h]  ; pop to global[index]
00E1B8CF   add         esi,4  ; update pointer
```

```
00E1B8D2   dec         esi  ; += 3: instruction length
...

      ; OPCODE<00 04>
00E14CC1   pop         dword ptr fs:[0000h]
00E14CC7   add         esp,4
00E14CCA   popad
00E14F95   movzx       eax,byte ptr [esi+2]  ; get a byte
00E14F99   xor         eax,45h  ; decrypt it
00E1524C   add         esp,eax  ; inc ESP by those bytes
00E1524E   add         esi,3  ; update pointer
...

      ; OPCODE <01 03>
00DF74EF   pop         dword ptr fs:[0000h]
00DF74F5   add         esp,4
00DF74F8   popad
00DF7787   movzx       eax,byte ptr [esi+2]  ; get srcidx
00DF7A49   mov         eax,dword ptr [eax*4+0DE8736h]  ; global[srcidx]
00DF7CF9   movzx       ecx,byte ptr [esi+3]  ; get xortype
00DF7FC4   jmp         dword ptr [ecx*4+0E1B9E9h]

      ; xortype == 0
00DF8266   movzx       ecx,byte ptr [esi+4]  ; get dstidx
00DF8532   mov         ecx,dword ptr [ecx*4+0DE8736h]  ; global[dstidx]
00DF87CF   xor         byte ptr [ecx],al  ; byte xor
00DF8A6F   add         esi,5  ; += instruction length
...

      ; xortype == 1
00DF92E8   movzx       ecx,byte ptr [esi+4]  ; get dstidx
00DF9593   mov         ecx,dword ptr [ecx*4+0DE8736h]  ; global[dstidx]
00DF9848   xor         word ptr [ecx],ax  ; word xor
00DF9AE6   add         esi,5  ; += instruction length
...

      ; xortype == 2
00DFA5F7   movzx       ecx,byte ptr [esi+4]  ; get dstidx
00DFA8C1   mov         ecx,dword ptr [ecx*4+0DE8736h]  ; global[dstidx]
00DFAE0F   xor         dword ptr [ecx],eax  ; long xor
00DFB0EE   add         esi,5  ; += instruction length
...

      ; OPCODE <01 04>
00E181A6   pop         dword ptr fs:[0000h]
00E181AC   add         esp,4
00E181AF   popad
00E18478   movzx       eax,byte ptr [esi+2]  ; idx
00E18721   sub         eax,3  ; idx -= 3
00E189C0   mov         ebx,dword ptr [eax*4+0DE8736h]  ; get global[idx]
00E18C5F   mov         edi,dword ptr [esi+3]  ; get constant
00E18F2C   add         ebx,edi  ; global[idx]+constant
      ; update NotZero Flag
00E18F2E   jne         00E1921A
00E19211   and         dword ptr ds:[0DE874Ah],0
00E19218   jmp         00E19224  ; E19502
00E1921A   mov         dword ptr ds:[0DE874Ah],1

00E19502   mov         dword ptr [eax*4+0DE8736h],ebx  ; update global[idx]
00E197AA   add         esi,7  ; += instruction length
...

      ; OPCODE <01 05>
00E1A05B   pop         dword ptr fs:[0000h]
00E1A061   add         esp,4
00E1A064   popad
```

```
00E1A2DA   movzx       eax,byte ptr [esi+2]  ; idx
00E1A582   sub         eax,2  ; idx -= 2
00E1A852   mov         ebx,dword ptr [eax*4+0DE8736h]  ; get global[idx]
00E1AB03   mov         edi,dword ptr [esi+3]  ; get constant
00E1ADD3   sub         ebx,edi  ; global[idx]-constant
       ; update NotZero Flag
00E1ADD5   jne         00E1ADE0
00E1ADD7   and         dword ptr ds:[0DE874Ah],0
00E1ADDE   jmp         00E1ADEA  ; E1B05F
00E1ADE0   mov         dword ptr ds:[0DE874Ah],1

00E1B05F   mov         dword ptr [eax*4+0DE8736h],ebx  ; update global[idx]
00E1B066   add         esi,7  ; update pointer
...

       ; OPCODE <01 06>
00E1226D   pop         dword ptr fs:[0000h]
00E12273   add         esp,4
00E12276   popad
00E12526   movzx       eax,byte ptr [esi+2]  ; idx
00E1252A   sub         eax,5  ; idx -= 5
00E127B4   mov         ebx,dword ptr [eax*4+0DE8736h]  ; get global[idx]
00E12A80   mov         edi,dword ptr [esi+3]  ; get constant
00E12D2E   and         ebx,edi  ; global[idx]-constant
       ; update NotZero Flag
00E12D30   jne         00E12D3E
00E12D32   and         dword ptr ds:[0DE874Ah],0
00E12D39   jmp         00E12FF9  ; E132EE
00E12FEF   mov         dword ptr ds:[0DE874Ah],1

00E132EE   mov         dword ptr [eax*4+0DE8736h],ebx  ; update global[idx]
00E132F5   add         esi,7  ; update pointer
...

       ; OPCODE <01 07>
00E14128   pop         dword ptr fs:[0000h]
00E1412E   add         esp,4
00E14131   popad
00E143F8   movzx       eax,byte ptr [esi+2]  ; idx
00E143FC   sub         eax,4  ; idx -= 5
00E146AE   mov         ebx,dword ptr [eax*4+0DE8736h]  ; get global[idx]
00E146B5   mov         edi,dword ptr [esi+3]  ; get constant
00E14974   or          ebx,edi  ; global[idx]|constant
       ; update NotZero Flag
00E14976   jne         00E14981
00E14978   and         dword ptr ds:[0DE874Ah],0
00E1497F   jmp         00E1498B
00E14981   mov         dword ptr ds:[0DE874Ah],1

00E1498B   mov         dword ptr [eax*4+0DE8736h],ebx  ; update global[idx]
00E14C66   add         esi,7  ; update pointer
...

       ; OPCODE <01 08>
00E016ED   pop         dword ptr fs:[0000h]
00E016F3   add         esp,4
00E016F6   popad
00E019C1   movzx       eax,byte ptr [esi+2]  ; idx
00E01C73   mov         ebx,dword ptr [eax*4+0DE8736h]  ; get global[idx]
00E01F36   mov         edi,dword ptr [esi+3]  ; get constant
00E021B7   xor         ebx,edi  ; global[idx]^constant
       ; update NotZero Flag
00E021B9   jne         00E02732
00E0247E   and         dword ptr ds:[0DE874Ah],0
00E0272D   jmp         00E02C74
00E029B3   mov         dword ptr ds:[0DE874Ah],1
```

```
00E02C74   mov         dword ptr [eax*4+0DE8736h],ebx  ; update global[idx]
00E02F55   mov         edi,0E00h
00E02F5A   shr         edi,9  ; 7: instruction length
00E031FB   add         esi,edi  ; update pointer
...

       ; OPCODE <01 09>
00E05353   pop         dword ptr fs:[0000h]
00E05359   add         esp,4
00E0535C   popad
00E055EA   movzx       eax,byte ptr [esi+2]  ; firstidx
00E058B1   dec         eax  ; firstidx-1
00E05B4B   mov         ebx,dword ptr [eax*4+0DE8736h]  ; get global[firstidx-1]
00E05DFF   movzx       eax,byte ptr [esi+3]  ; secondidx
00E060DF   sub         eax,3  ; secondidx-3
00E0636D   mov         edi,dword ptr [eax*4+0DE8736h]  ; global[secondidx-3]
00E06610   add         edi,ebx  ; sum both globals
       ; update NotZero Flag
00E06612   jne         00E06B54  ; E06E08
00E068AE   and         dword ptr ds:[0DE874Ah],0
00E06B4F   jmp         00E06E12  ; E070E0
00E06E08   mov         dword ptr ds:[0DE874Ah],1

00E070E0   mov         dword ptr [eax*4+0DE8736h],edi  ; update global[secondidx-3]
00E070E7   add         esi,4  ; update pointer
...

       ; OPCODE <01 0A>
00DFB948   pop         dword ptr fs:[0000h]
00DFB94E   add         esp,4
00DFB951   popad
00DFBBE6   movzx       eax,byte ptr [esi+2]  ; firstidx
00DFBEAA   sub         eax,2  ; firstidx-2
00DFBEAD   mov         ebx,dword ptr [eax*4+0DE8736h]  ; get global[firstidx-2]
00DFC166   movzx       eax,byte ptr [esi+3]  ; secondidx
00DFC41D   dec         eax  ; secondidx-1
00DFC6C7   mov         edi,dword ptr [eax*4+0DE8736h]  ; get global[secondidx-1]
00DFC98A   cmp         edi,ebx  ; second CMP first
       ; update NotZero Flag
00DFC98C   jne         00DFCEFD
00DFCC35   and         dword ptr ds:[0DE874Ah],0
00DFCEF8   jmp         00DFD18C
00DFCEFD   mov         dword ptr ds:[0DE874Ah],1
       ; assign global[firstidx-2] to global[secondidx-1]
00DFD18C   mov         dword ptr [eax*4+0DE8736h],ebx
00DFD43F   add         esi,4  ; update pointer
...

       ; OPCODE <02 00>
00DEE16F   pop         dword ptr fs:[0000h]
00DEE175   add         esp,4
00DEE401   popad
00DEE402   push        dword ptr ds:[0E1BC73h]  ; encoded exit address
00DEE684   xor         dword ptr [esp],48415830h  ; DE872F
00DEE946   ret  ; go to exit snippet

       ; OPCODE <02 01>
00DEE947   pop         dword ptr fs:[0000h]
00DEE94D   add         esp,4
00DEE950   popad
00DEEC12   mov         eax,dword ptr [esi+2]  ; get a long
00DEEEC7   add         eax,0FEA731DEh  ; decode it
00DEF171   mov         eax,dword ptr [eax+2]
00DEF411   mov         eax,dword ptr [eax]  ; external address
       ; make the address relative, to be put in the following call
```

```
00DEF413   mov         edi,0DEF94Eh
00DEF418   sub         eax,edi
00DEF69D   sub         eax,4
00DEF94C   stos        dword ptr [edi]  ; put address to DEF94E
00DEF94D   call        ...  ; call the API
       ; store the result in global[0]
00DEFC14   mov         [00DE8736],eax
       ; update NotZero Flag
00DEFEBA   test        eax,eax
00DEFEBC   jne         00DF09CA  ; DF0C4B
00DF0150   and         dword ptr ds:[0DE874Ah],0
       ; go to DF0C55 -> DF0F24
00DF0443   mov         edi,0DEE6B3h
00DF0448   push        edi
00DF0715   add         dword ptr [esp],25A2h  ; DF0C55
00DF09C9   ret  ; go to DF0C55
00DF0C4B   mov         dword ptr ds:[0DE874Ah],1
00DF0F24   add         esi,6  ; instruction length
...

       ; OPCODE <02 02>
00DFDCA3   pop         dword ptr fs:[0000h]
00DFDCA9   add         esp,4
00DFDCAC   popad
00DFDCAD   mov         eax,dword ptr [esi+2]  ; get encoded long
00DFDF5E   add         eax,0AEFDED04h
00DFE20E   movzx       edi,byte ptr [esi+6]  ; idx
00DFE4DC   mov         dword ptr [edi*4+0DE8736h],eax
00DFE4E3   add         esi,7  ; update pointer
...

       ; OPCODE <02 03>
00E09AB5   pop         dword ptr fs:[0000h]
00E09ABB   add         esp,4
00E09ABE   popad
00E09D76   movzx       eax,byte ptr [esi+2]  ; idx
00E0A03E   dec         dword ptr [eax*4+0DE8736h]
       ; update NotZero Flag
00E0A045   jne         00E0A811
00E0A2E4   and         dword ptr ds:[0DE874Ah],0
00E0A587   jmp         00E0AD8F
00E0AACC   mov         dword ptr ds:[0DE874Ah],1

00E0AD8F   add         esi,3  ; update pointer
...

       ; OPCODE <02 04>
00E0D4BC   pop         dword ptr fs:[0000h]
00E0D4C2   add         esp,4
00E0D4C5   popad
00E0D4C6   movzx       eax,byte ptr [esi+2]  ; idx
00E0D76F   inc         dword ptr [eax*4+0DE8736h]
       ; update NotZero Flag
00E0D776   jne         00E0DCF2  ; E0DF7A
00E0DA49   and         dword ptr ds:[0DE874Ah],0
00E0DCED   jmp         00E0E229
00E0DF7A   mov         dword ptr ds:[0DE874Ah],1  ; set NZflag

00E0E229   add         esi,3  ; update pointer
...

       ; OPCODE <02 05>
00E11477   pop         dword ptr fs:[0000h]
00E1147D   add         esp,4
00E11480   popad
00E11726   movzx       eax,byte ptr [esi+2]  ; idx
```

```
00E119CF   and         dword ptr [eax*4+0DE8736h],0
00E11C70   and         dword ptr ds:[0DE874Ah],0  ; clear NotZero Flag
00E11C77   add         esi,3  ; update pointer
...

       ; OPCODE <02 06>
00E15B11   pop         dword ptr fs:[0000h]
00E15B17   add         esp,4
00E15B1A   popad
00E15DBC   movzx       eax,byte ptr [esi+2]  ; idx
00E16049   mov         edi,dword ptr ds:[0DE8736h]  ; get value from global[0]
00E162EE   movzx       ecx,word ptr [esi+3]  ; get count
00E1658B   repne scas  byte ptr [edi]
00E16843   test        ecx,ecx
00E16845   je          00E17352

       ; found
00E16AF2   mov         dword ptr ds:[0DE8736h],edi  ; store new ptr in result[0]
       ; go to E1735C
00E16DCB   push        0E0DD08h
00E1707F   add         dword ptr [esp],9654h  ; E1735C
00E17351   ret

       ; not found
00E17352   mov         dword ptr ds:[0DE874Ah],0  ; clear NZflag

00E1735C   add         esi,5  ; update pointer
...

       ; OPCODE <02 07>
00E1B92B   pop         dword ptr fs:[0000h]
00E1B931   add         esp,4
00E1B934   popad
00E1B935   int         3  ; break
00E1B936   add         esi,2  ; update pointer (only 2 bytes)
...

       ; OPCODE <02 08>
00E0E816   pop         dword ptr fs:[0000h]
00E0E81C   add         esp,4
00E0E81F   popad
00E0EAE9   movzx       eax,byte ptr [esi+2]  ; srcidx
00E0ED84   mov         eax,dword ptr [eax*4+0DE8736h]
00E0F030   mov         eax,dword ptr [eax]  ; dereference
00E0F032   movzx       edi,byte ptr [esi+3]  ; dstidx
00E0F2E0   mov         dword ptr [edi*4+0DE8736h],eax  ; assign
00E0F2E7   add         esi,4  ; update pointer
...

       ; OPCODE <02 09>
00E0C6C9   pop         dword ptr fs:[0000h]
00E0C6CF   add         esp,4
00E0C6D2   popad
00E0C6D3   movzx       eax,byte ptr [esi+2]  ; idx
00E0C99A   mov         eax,dword ptr [eax*4+0DE8736h]  ; get global[idx]
00E0CC35   bswap       eax  ; BUG: do not save back
00E0CF0A   add         esi,3  ; update pointer
...

       ; OPCODE <02 0A>
00E0B5E7   pop         dword ptr fs:[0000h]
00E0B5ED   add         esp,4
00E0B5F0   popad
00E0B8C1   movzx       eax,byte ptr [esi+2]  ; srcidx
00E0B8C5   mov         eax,dword ptr [eax*4+0DE8736h]
00E0BB76   movzx       eax,byte ptr [eax]  ; dereference
```

```
00E0BE33   movzx       edi,byte ptr [esi+3]  ; dstidx
00E0C0C0   mov         dword ptr [edi*4+0DE8736h],eax  ; assign
00E0C3C4   add         esi,4  ; update pointer
...

       ; OPCODE <02 0B>
00E03D6A   pop         dword ptr fs:[0000h]
00E03D70   add         esp,4
00E03D73   popad
00E0402D   movzx       eax,byte ptr [esi+2]  ; srcidx
00E042E6   mov         eax,dword ptr [eax*4+0DE8736h]
00E04588   movzx       eax,word ptr [eax]  ; dereference
00E04823   movzx       edi,byte ptr [esi+3]  ; dstidx
00E04ACD   mov         dword ptr [edi*4+0DE8736h],eax  ; assign
00E04D5F   add         esi,4  ; update pointer
...

       ; OPCODE <03 00>
00DFEFDC   pop         dword ptr fs:[0000h]
00DFEFE2   add         esp,4
00DFEFE5   popad
00DFF2A3   mov         eax,dword ptr [esp]  ; first value
00DFF51B   cmp         eax,dword ptr [esp+4]  ; cmp second value
00DFF51F   jne         00E005E5  ; E008A4

       ; equal
00DFF80F   add         esp,8  ; clean stack
00DFFAC1   mov         esi,dword ptr [esi+2]  ; address
00DFFD4A   sub         esi,31337h  ; address -= 31337
00DFFFF6   movzx       eax,byte ptr [esi]  ; first byte from new flow
...

       ; not equal
00E008A4   mov         eax,0FFFFFFF9h
00E00B6A   not         eax  ; 6
00E00B6C   add         esi,eax  ; update pointer
00E00E27   add         esp,8  ; clear the stack
...

       ; OPCODE <04 00>
00E034FD   pop         dword ptr fs:[0000h]
00E03503   add         esp,4
00E03506   popad
       ; modify the pointer
00E037D6   mov         esi,dword ptr [esi+2]  ; address
00E03A61   add         esi,0DEADEADh  ; address+DEADEAD
...

       ; OPCODE <04 01>
00E07989   pop         dword ptr fs:[0000h]
00E0798F   add         esp,4
00E07992   popad
00E07C49   mov         eax,[0DE874Ah]  ; NotZero Flag
00E07EF4   test        eax,eax
00E07EF6   jne         00E08A51  ; jump to given address

       ; continue in sequence
00E081AF   add         esi,6  ; update pointer
...

       ; jump to given address
00E08CFE   mov         esi,dword ptr [esi+2]  ; address
00E08FC2   lea         esi,[esi+0E1BA3Eh]  ; address+E1BA3E
...

       ; OPCODE <04 02>
```

```
00E0F8A7   pop         dword ptr fs:[0000h]
00E0F8AD   add         esp,4
00E0F8B0   popad
00E0FB4D   mov         eax,[00DE874A]  ; get NotZero Flag
00E0FE17   test        eax,eax
00E0FE19   je          00E1096C  ; jump to given address

       ; continue in sequence
00E0FE1F   add         esi,6  ; update pointer
...

       ; jump to given address
00E1096C   mov         esi,dword ptr [esi+2]  ; address
00E10BE5   lea         esi,[esi+0E1BA41h]  ; address+E1BA41
...

       ; OPCODE <05 00>
00E13604   pop         dword ptr fs:[0000h]
00E1360A   add         esp,4
00E1360D   popad
00E1389E   mov         eax,dword ptr [esi+2]  ; address
00E13B63   lea         edi,[esi+6]  ; return address
00E13B66   push        edi  ; push the return address
00E13E36   mov         esi,eax  ; set the pointer to the new address
...

       ; OPCODE <05 01>
00E15554   pop         dword ptr fs:[0000h]
00E1555A   add         esp,4
00E1555D   popad
       ; retrieve the return address from the stack
00E1555E   pop         esi
...
```

At this point, the biggest part of our job is over.

We have mentioned before the opcode <02 07>: as we have seen above, it simply perform an Interrupt 3. This means that we can patch in any point the opcodes flow with the couple of bytes 02 07, and our debugger will break at that point: we are now free to examine the globals, or to restart in single stepping mode.
The only caution we must take is to replace back the two opcodes overwritten by 02 07, before restarting.

# 11. The crypto stuff

We can finally analyze the opcodes. with the opcodes table in front of us! Let's start, founding the first 4 characters of the password!

```
       ;
       ; E1BA3D: coded area starts here
       ; globals[0-5] at DE8736, NotZero Flag at DE874A
       ;
             ; decrypt first the opcodes from E1BA65 to E1BC73
E1BA3D  02 02 5102150A 03  ; global[3] <- encoded 20E (opcodes length)
E1BA44  00 01 5311A72E  ; push constant (ptr E1BA65)
E1BA4A  00 03 66  ; pop at global[66^66]
E1BA4D  02 02 5102134F 02  ; put 00000053 in global[2]: xor value

             ; decryption loop
E1BA54  01 03 02 00 00  ; *global[0] ^= (BYTE)global[2]
E1BA59  02 04 00  ; bump global[0]
```

```
E1BA5C  02 03 03  ; dec global[3] (length)
E1BA5F  04 01 00000016  ; jnz E1BA3E+16h (E1BA54)

             ; here start the decrypted code
             ; (shown already decrypted)
E1BA65  02 02 ADCA212D 01  ; 5CC80E31 to global[1]

E1BA6C  02 01 0236DE2F  ; call GetCommandLineA, store the result in global[0]

             ; after the scan, global[0] has the pointer to the password:
             ; c0 c1 c2 c3 c4 c5 c6 c7 c8
E1BA72  02 06 20 0255  ; scan for <20> in *(result[0]) for at most 0255h bytes
E1BA77  04 02 0000012E  ; jz (not found) E1BA41+12E=E1BB6F (ko)
E1BA7D  02 08 00 02  ; global[2] <- *global[0] = c3 c2 c1 c0

             ; check c0 c1 c2 c3
E1BA81  01 04 05 1D9BDC45  ; global[5-3] += 1D9BDC45 = c3c2c1c0 + 1D9BDC45
E1BA88  01 04 04 74519745  ; global[4-3] += 74519745 = D119A576
E1BA8F  01 05 04 AD45DFE2  ; global[4-2] -= AD45DFE2 = c3c2c1c0 - 8FAA039D
E1BA96  01 04 04 DEADBEEF  ; global[4-3] += DEADBEEF = AFC76465
        01 04 05 68656C6C  ; global[5-3] += 68656C6C = c3c2c1c0 - 27449731
        01 05 03 17854165  ; global[3-2] -= 17854165 = 98422300
        01 05 04 41776169  ; global[4-2] -= 41776169 = c3c2c1c0 - 68BBF89A
        01 04 04 73686F77  ; global[4-3] += 73686F77 = 0BAA9277
        01 04 05 69747320  ; global[5-3] += 69747320 = c3c2c1c0 + 00B87A86
        01 05 03 206E6F20  ; global[3-2] -= 206E6F20 = EB3C2357
        01 04 05 64726976  ; global[5-3] += 64726976 = c3c2c1c0 + 652AE3FC
        01 04 04 6D657263  ; global[4-3] += 6D657263 = 58A195BA
        01 05 04 6E757473  ; global[4-2] -= 6E757473 = c3c2c1c0 - 094A9077
        01 05 03 79212121  ; global[3-2] -= 79212121 = DF807499
        01 05 04 65683F21  ; global[4-2] -= 65683F21 = c3c2c1c0 - 6EB2CF98
        01 06 07 DFFFFFFF  ; global[7-5] &= DFFFFFFF

             ; c3c2c1c0-6EB2CF98(global[2]) must equal DF807499(global[1]
             ; this means c3c2c1c0 = DF807499+6EB2CF98 = 4E334431 ("n3D1")
        00 02 45  ; push global[45^47(2)]
        00 02 46  ; push global[1]
        03 00 00E4CE3D  ; cmp and jmp to E1BB06 if same
             ; not the same
        02 05 03  ; clear global[3] and NZflag
        04 02 0000012E  ; jz E1BB6F ko

             ; skip c0 c1 c2 c3, already checked
             ; password ptr += 4
E1BB06  02 04 00  ; bump global[0], ptr to password
        01 04 03 00000002  ; global[3-3] += 2
        02 04 00  ; bump global[0] long
```

The following instructions will check two other characters of the password, c4 and c5. To be more exact, the two bytes c4 and c5 are xored with two bytes inside the flow of the opcodes: if the result is not correct, it is very easy that the program will crash. We note here that the solution can be more than one.

The original bytes, 47 42 (03), xored with c4 c5, must give a valid instruction having 3 bytes as length. From all the possible 3-bytes solutions, we must exclude the push/pop instructions 00 02 03 and 00 03 03) to avoid stack unpairing. The other possible solutions are 02 03 03 (DEC global[3]), 02 04 03 (INC global[3]), 02 05 03 (CLEAR global[3]) and 02 09 03 (BSWAP global[3]). They are all valid, since the content of global[3], which will be pushed as second argument of printf, is meaningless. In fact, printf() requires only one argument when the format string has no variables inside it ;-)
I have adopted 02 05 03 (CLEAR) for analogy with the other printf() call, so we have:

```
    c4 = 47 ^ 02 = 45 'E' (MANDATORY)
    c5 = 42 ^ 05 = 47 'G' (FOR EXAMPLE; see other solutions at the end of this
chapter)



        02 0B 00 01  ; get word pointed by global[0] and put in global[1]
        00 02 46  ; push global[1]
        00 03 64  ; pop at global[2] <- c5c4
        00 02 47  ; save global[0] on stack

             ; one word in the opcodes flow will be patched using a value derived
             ; from the password: put in local[0] the pointer to this word
        00 01 530822D6  ; push constant D8360D
        00 03 66  ; pop at global[0]
        01 04 03 00098548  ; global[3-3] += 98548 <- E1BB55
        02 03 00  ; dec global[0] <- E1BB54 pointer to word to patch

             ; modify that word xoring it with c5c4
        01 03 02 01 00     ; *global[0] .wordXOR. global[2]
```

The remaining characters c6 and c7 will be determined analyzing the remaining opcodes:

```
        00 03 66  ; restore global[0] from stack

             ; global[2] <- c4+c6
E0B5E7  02 0A 00 02  ; global[2] <- BYTE pointed by global[0]: c4
        01 04 03 00000002  ; global[0] += 2
        02 0A 00 01  ; global[1] <- BYTE pointed by global[0] c6
        01 09 02 05  ; global[5-3] += global[2-1] c4+c6

E1BB4E  05 00 00E1BB86  ; call E1BB86

             ; three bytes instruction, modified in the previous snippet from 47 42
03 to 02 05 03
E1BB54  47 42 03->02 05 03  ; clear global[3] and NZflag
E1BB57 00 02 44  ; push global[44^47=3] USELESS!
        00 01 5311A8F3  ; push encoded constant E1BC2A (congrats)
        02 01 0236DE23  ; call [E28052+2] MSVCRT!printf
        00 04 4D  ; adjust the stack += 8
        04 00 F2F6DCD7  ; jmp E1BB84 (exit)

             ; ko
E1BB6F  00 00 37195411  ; push encoded constant 0 USELESS!
        00 01 5311A8DC  ; push encoded long E1BC13 "Please Authenticate!"
        02 01 0236DE23  ; call [E28054] MSVCRT!printf
        00 04 4D  ; adjust the stack += 4D^45(8)
E1BB84  02 00  ; exit at retaddr

      ;
      ; sub E1BB86
      ;
             ; put 4E into global[1]
E1BB86  02 02 51021348 01  ; global[1] <- 4C
        02 04 01  ; bump global[1] 4D
        02 04 01  ; bump global[1] 4E
        01 04 04 00000005  ; global[1] += 5 53
        02 03 01  ; dec global[1] 52
        01 05 03 00000004  ; global[1] -= 4 4E

             ; condition: c4+c6-5A == 4E, c4+c6 = A8 => 45+c6 = A8 => c6 = 63
        01 05 04 0000005A  ; global[2] c4 + c6 - 5A
```

```
        01 0A 04 02  ; global[4-2] CMP global[2-1] and put there
E1BBEF  04 01 00000131  ; jnz E1BA3E+131 = E1BB6F ko

        02 03 00  ; dec global[0]: pointer to c5
        02 0A 00 02  ; global[2] <- BYTE pointed by global[0] c5
        01 04 03 00000002  ; global[3-3] += 2 pointer to c7
        02 0A 00 01  ; global[1] = c7
        01 09 02 05  ; global[5-3] += global[2-1] c5+c7
        02 04 02  ; bump global[2] c5+c7+1
        01 05 04 0000004E  ; global[2] -= c5+c7-4D

             ; global[0] <- pointer to return opcode to decrypt (E1BC11)
        00 01 5310CA2D  ; push encoded E0DD64
        00 03 66  ; pop at global[0]
        01 04 03 AC DE 00 00  ; global[0] += DEAC E1BC10
        02 04 00  ; bump global[0] E1BC11

             ; decrypt 2-bytes return opcode 47 01 to 05 01
             ; MUST BE 47^42 05
             ; this means c5+c7-4D == 42 => c7 = 8F-c5
        01 03 02 00 00  ; *global[0] byteXOR global[2] E1BC11

             ; as a further confirmation, the congrats string is decrypted
        02 02 45 13 02 51 03  ; global[3] <- 0x49 counter
        00 01 5311A8F3  ; push encoded
        00 03 66  ; pop at global[0] E1BC2A string to decrypt
        02 04 02  ; bump global[2]: XOR value 42+1=43

             ; decryption loop (XOR with constant byte)
E1BC00:
        01 03 02 00 00  ; *global[0] byteXOR global[2]
        02 04 00  ; bump global[0] ptr to string
        02 03 03  ; dec global[3] counter
        04 01 000001C2  ; jnz (E1BA3E+1C2)E1BC00

E1BC11  47 01 -> 05 01  ; return

E1BC13 "Please Authenticate!\00A\00D"

             ; string to decrypt
E1BC2A-E1BC73
       14 26 2F 20 2C 2E 26 6D 6D 6D 49 4E 06 3B 33 2F
       2C 2A 37 63 25 2C 31 63 2A 37 63 27 2C 26 30 2D
       64 37 63 2E 22 37 37 26 31 63 72 6D 3B 63 00 2C
       36 31 37 26 30 3A 63 2C 25 63 0D 2A 20 2C 2F 22
       30 63 01 31 36 2F 26 39 43
```

We can finally discover the four possible solutions: The mandatory values are `1D3nE_c_`

**I**
c5 = 42^05 = 47 'G'
c7 = 8F-c5 = 8F-47 = 48 'H'
"1D3nEGcH"

**II**
c5 = 42^03 = 41 'A'
c7 = 8F-c5 = 8F-41 = 4E 'N'
"1D3nEAcN"

**III**
c5 = 42^04 = 46 'F'
c7 = 8F-c5 = 8F-46 = 49 'I'
"1D3nEFcI"

```
IV
c5 = 42^09 = 4B 'K'
c7 = 8F-c5 = 8F-4B = 44 = 'D'
"1D3nEKcD"
```

As an alternate method, specially if we are not yet very confident with the Opcodes meaning, and we do not know what to patch at addresses E1BB54 and E1BC11, we can start from the bottom, brute-forcing the byte which will correctly decrypt the congratulations string. A simple program which will do the job has been included with the name *getcongrats.c*.

# 12. The answers

As a summary for the above analysis, I will try to answer the questions required by the challenge.

**Identify and explain any techniques in the binary that protect it from being analyzed or reverse engineered.**
We have seen a lot of big and little tricks, from the instruction patterns aimed to make reverser's life harder and confuse the disassemblers (The Garbage Patterns), to the patterns aimed at confusing the debuggers (the "call patterns" which generate exceptions), to the Matrioska Layers, to the codification through proprietary Opcodes, to the decryption on the fly of the Opcodes themselves using Opcoded code.
There are also many other little tricks, such as avoiding to directly expose constants and strings, nullifying hardware breakpoints, checking software breakpoints on the used external APIs, the PE header garbling, the use of the RDTSC instruction, and so on.

**Something uncommon has been used to protect the code from beeing reverse engineered, can you identificate what it is and how it works?**
Have a look at The Opcodes Interpreter.

**Provide a means to "quickly" analyse this uncommon feature.**
Have a look at Defeating The Opcodes Interpreter.

**Which tools are the most suited for analysing such binaries, and why?**
We have seen that since we cannot trust on disassemblers, the most powerful tool is a good debugger, which will permit a live approach to the reverse engineering problem.

**Identify the purpose (fictitious or not) of the binary.**
The real purpose of the binary is to hide something. In this case a simple crypto algorithm used to check a password is hidden inside the Opcodes.

**What is the binary waiting from the user? Please detail how you found it.**
The password itself must be provided as argument to the executable. The only way to understand this is to closely analyze the Opcodes which follows the call to the Windows API GetCommandLineA().

**BONUS: What techniques or methods can you think of that would make the binary harder to reverse engineer?**

Well, since the only tool I could successfully use was a debugger, we should try to make harder its use. For example we could use, as exceptions, not just Access Violations but Debugger Exceptions (INT 3) which would have interferences with the debugger operations such as single-stepping and breakpointing.

If simple or more complex CRC checks would have been performed in more places of the binary, setting breakpoints would be even harder.

Furthermore, the unveiling of the Matrioska layers has been rather simple, due to the tedious repetition of the patterns and to the same encryption algorithm for all the 175 layers.

Also the garbage patterns are rather repetitive, and, what is worst, they can always surely be identified by a leading PUSHAD and a trailing POPAD instruction...

# References

1. HoneyPot project at www.honepot.net.
2. http://board.win32asmcommunity.net/viewtopic.php?t=19965.