Write-up: solution to a RE crackme

CTFs and challenges mainly based on reverse engineering are a bit uncommon, so when I find one I am always happy to devote some time to try and solve it. This write-up will be on the crackme created by hasherezade. To make the reading more spicy I decided to explain my thought process while going through the challenge, instead of writing a plain (boring) solution.

Stage 1

Step 1

Obvious first step: run the executable. There isn’t any user input, but we are greated with a nice *“I am so sorry, you failed! :(“* message. After feeding the exe to IDA, we can directly look at the strings, hoping to find that message. The cross-reference to it leads us to the real *main* function (at 0x401910). Pretty straight forward, the return value of the function at 0x4014F0 decides whether we fail or succeed. Inside it, the pivot is the function at 0x4031C0, which receives two hardcoded buffers and does the following:

1. compute the SHA-256 hash of the second buffer
2. generate a AES-256 key from the hash (via `CryptDeriveKey`)
3. decrypt the first buffer using that key

Back to 0x4014F0, the program computes a checksum of the decrypted data and tests it against the harcoded value 0x3B47B2E6. In order to correctly solve this first step we need to get the right key, that is, the right content into the second buffer. This buffer is filled up by the 9 functions (4 bytes each) that are called before the decryption routine. Each function deals with an anti-debug or anti-emulation technique. The anomalous thing is that these functions write in the buffer only if the conditions are met (not bypassed) - the exact opposite of what a malware would do. For example, one of the functions checks for the presence of hardware breakpoints, and only if at least one is set it writes its chunk of data in the buffer.

```
push    ebp
mov     ebp, esp
sub     esp, 84h
mov     eax, dword_428600
xor     eax, ebp
mov     [ebp+var_4], eax
rdtsc
mov     ts_low, eax
mov     ts_high, edx
call    anti_isDebuggerPresent
push    1000            ; dwMilliseconds
call    ds:Sleep
call    anti_ODbgString
call    anti_HWBreak
call    anti_PEB
push    2
call    anti_Devices
call    anti_Registry
push    2
call    anti_Modules
push    2
call    anti_Processes
push    ts_high
push    ts_low
call    anti_Timing
push    0               ; char
push    400h            ; dwDataLen
push    offset encrypted_data ; int
push    100h            ; dwBufLen
lea     eax, [ebp+var_84]
push    offset unk_429E70 ; int
push    eax             ; int
mov     [ebp+var_84], 51ED0C52h
mov     [ebp+var_80], 2D72A5B3h
mov     [ebp+var_7C], 0E1DE8D78h
```

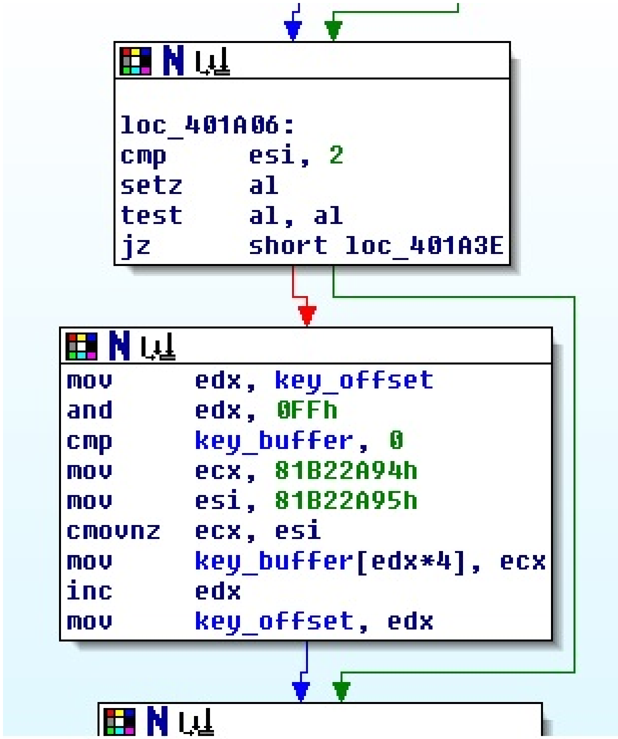
“anti-analysis” functions that write the key

A quick list of the techniques deployed by each function:

- `IsDebuggerPresent` + `CheckRemoteDebugggerPresent`
- `OutputDebugString`

- [Hardware breakpoints]
- `PEB.BeingDebugged` + `PEB.NtGlobalFlag`
- Search known devices, modules and processes: these 3 functions have the same structure, they use the Windows API to get the various names, compute their hash and check them against a list of hardcoded values
- Known VirtualBox registry key: check the existence of the key HKLM\HARDWARE\ACPI\DSDT\VBOX__
- Timing: perform the sequence `rdtsc` (**R**ea**D** **T**ime**S**tamp **C**ounter) -> `Sleep(1000)` -> `rdtsc` and check the difference between the two values

And a sample of the piece of code that writes the key chunk:

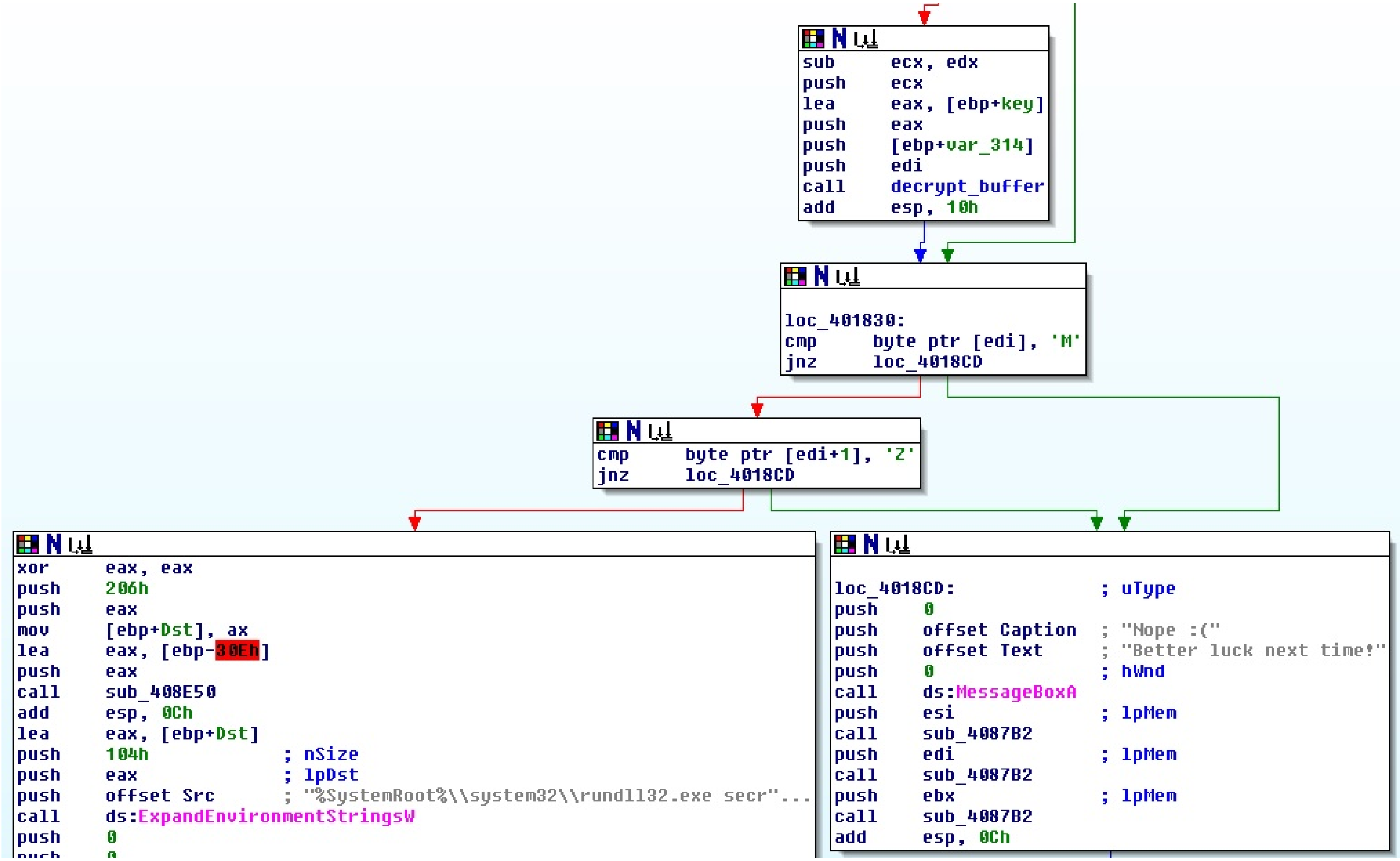


if the conditions is met, write a 4-byte key chunk

Matching all the required conditions gives us the key, and the decrypted data results in a URL: https://pastebin.com/raw/9FugFa91 . At that URL there is some Base64-encoded data.

Step 2

Confident that we have overcome the first challenge, we can let the program continue its execution, but only to be brought back to earth by a message box saying *“Better luck next time!”*. Once again we need to find its reference, which is inside the function at 0x401690; specifically, the error message is displayed if the first two bytes of a certain memory region are not “MZ”, probably meaning that the region needs to contain a PE file.



To understand what is in that region we need to go through the whole function:

1. download the data from the previous URL
2. Base64-decode it
3. decompress it via RtlDecompressBuffer
4. get the content of the clipboard
5. XOR-decrypt the decompressed buffer using the clipboard data as key
6. check the first bytes of the decrypted buffer
7. …

To get the key we can use a simple trick specific to XOR encryption. In general:

`N ^ 0 = N`

And in our case:

`key ^ 00..00 = key`

this means that if the original data contains a sufficiently long sequence of null bytes we may be able to get the whole key, or at least to guess it. This condition is easily met considering that the header of a PE file has lots of null-byte regions.

By setting a breakpoint at 0x401828 (i.e just before the decryption routine) we have access to the encrypted data, from which it is pretty clear that the key is *“malwarebytes”*.

```
0041DAC8 20 3B FC 77 62 72 65 62 7D 74 65 73 92 9E 6C 77  ;üwbreb}tes..lw
0041DAD8 D9 72 65 62 79 74 65 73 2D 61 6C 77 61 72 65 62 Ùrebytes-alwareb
0041DAE8 79 74 65 73 6D 61 6C 77 61 72 65 62 79 74 65 73 ytesmalwarebytes
0041DAF8 6D 61 6C 77 61 72 65 62 79 74 65 73 9D 61 6C 77 malwarebytes.alw
0041DB08 6F 6D DF 6C 79 C0 6C BE 4C D9 6D 3B AC 53 31 0A omßlyÀl¾LÙm;¬S1.
0041DB18 10 07 45 03 1F 0E 0B 05 00 1F 45 01 18 1A 0B 1C ..E.......E.....
0041DB28 19 41 0E 12 41 00 10 0C 59 1D 0B 53 29 2E 3F 57 .A..A...Y..S).?W
0041DB38 0C 1D 01 07 57 79 68 79 49 61 6C 77 61 72 65 62 ....WyhyIalwareb
0041DB48 C7 27 9C 54 97 53 FB 03 9B 40 F2 16 83 46 F2 07 Ç'.T.Sû..@ò..Fò.
```

```
0041DAC8 4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 MZ..........ÿÿ..
0041DAD8 B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 ........@.......
0041DAE8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0041DAF8 00 00 00 00 00 00 00 00 00 00 00 00 F0 00 00 00 ............ð...
0041DB08 0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68 ..º..´.Í!¸.LÍ!Th
0041DB18 69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F is program canno
0041DB28 74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20 t be run in DOS 
0041DB38 6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00 mode....$.......
0041DB48 BE 53 F9 27 FA 32 97 74 FA 32 97 74 FA 32 97 74 ¾Sù'ú2.tú2.tú2.t
```

PE file before and after encryption

Once the buffer is correctly decrypted, the program continues by performing a classic process hollowing. Let me summarize the steps:

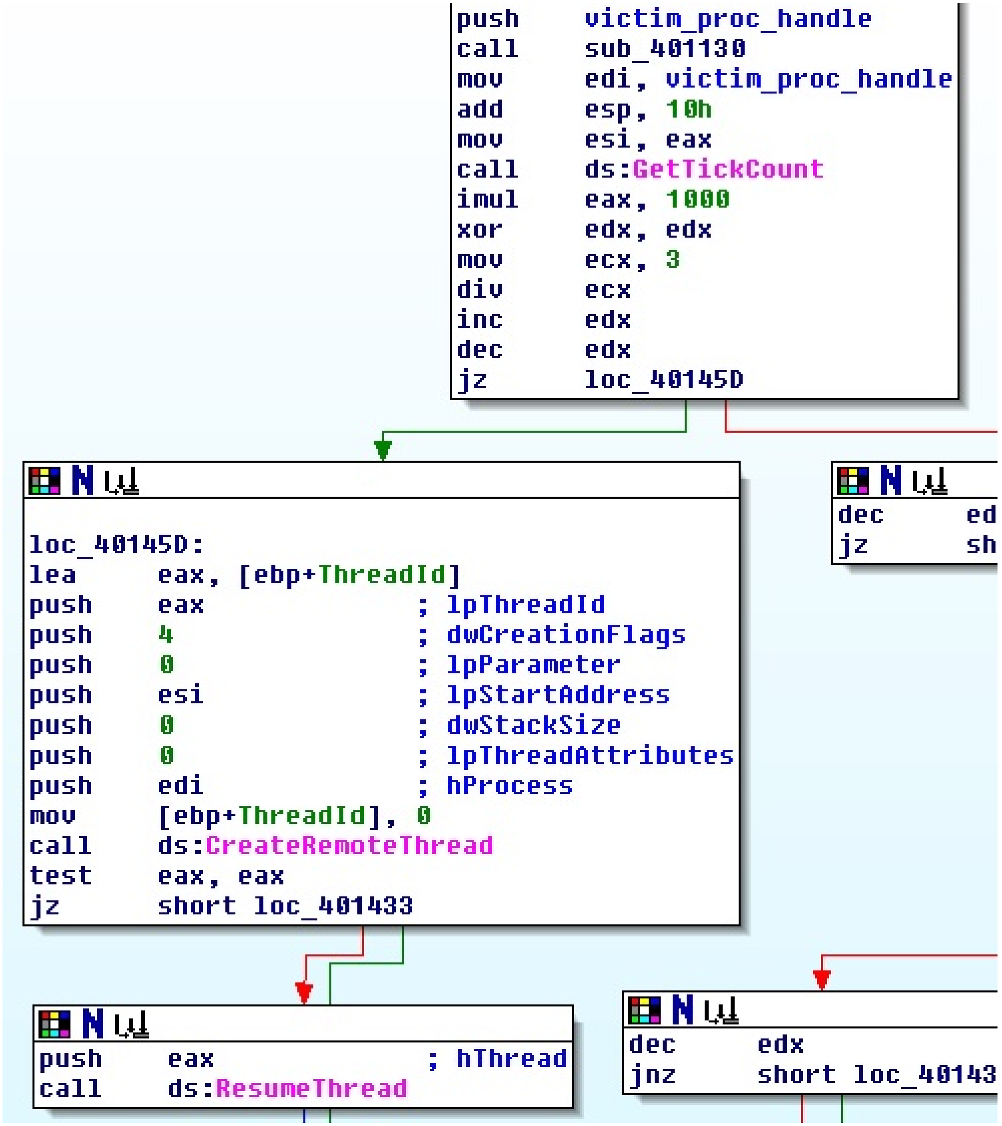
1. create a new suspended process with the command *%SystemRoot%\system32\rundll32.exe secret.dll,#1*
2. in the process memory, allocate a region with RWX permissions
3. write the PE file in the new region
4. change the base address in the PEB (the location of the PEB is stored in the EBX register)
5. change the entry point (stored in the EAX register)
6. resume the execution of the process.

PRACTICAL NOTE - how to debug the hollowed process: the cleanest way is to set a breakpoint at 0x40146F (for the crackme). At this point the new process is still suspended, so we can safely attach a debugger without interrupting anything. Moreover, in the EAX register of the crackme there is the new entry point for the hollowed process, therefore we can set a breakpoint point at it (in the debugger attached to the hollowed process of course).

Stage 2

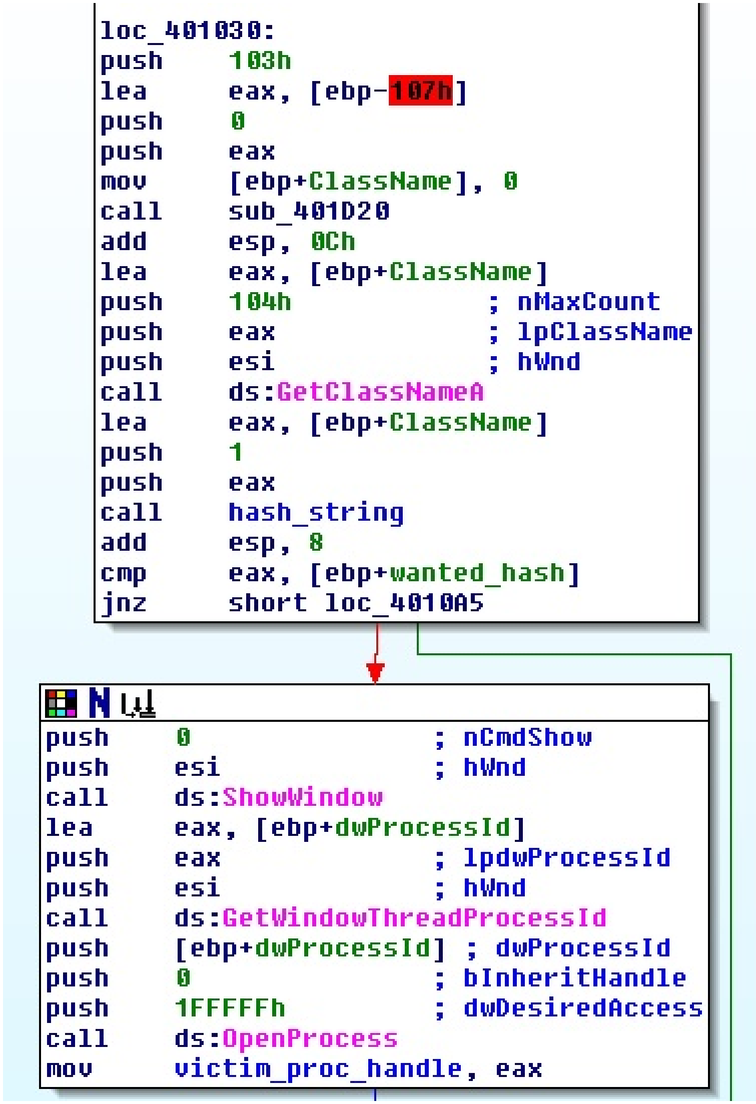
Once again letting the hollowed process run, we get the message *“You failed :(Better luck next time!”*, which is referenced in the function at 0x401260.

Looking towards the end of the function, its purpose becomes clear: it injects some code in another process and creates a new thread to execute it. Specifically, it uses one of 3 possible API functions to create the thread, namely `CreateRemoteThread`, `RtlCreateUserThread` and `ZwCreateThreadEx`. The choice is made by the randomly generate value `(GetTickCount() * 1000) % 3`. At this point we need to find which is the victim process and what code is injected.



Lets address the first question. Tracing the process handle back from the thread creation APIs we can see that it is stored in a global variable at 0x40EF50. The variable is set in a callback routine of `EnumWindows` (at 0x401000). For every window, the routine does the following:

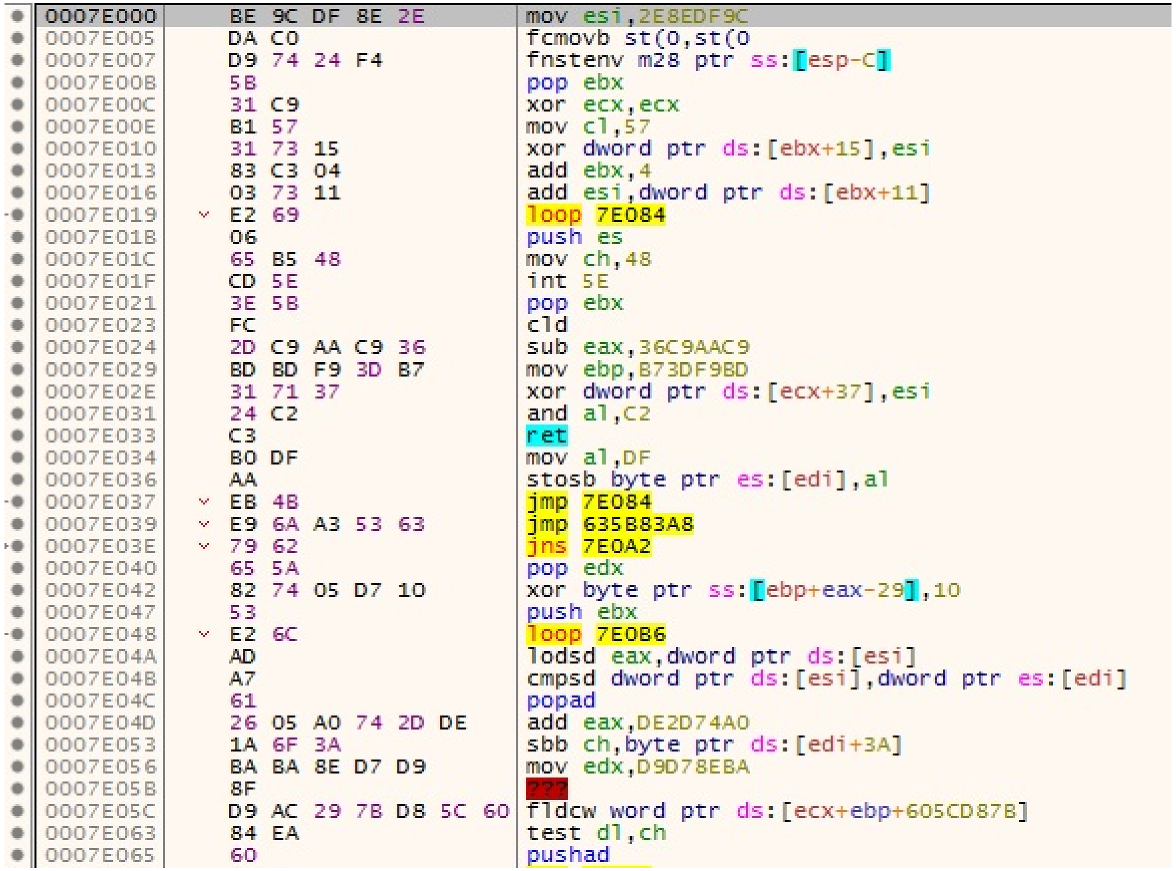
1. get the window class name (`GetClassNameA`)
2. compute a hash of the name
3. check it against the hardcoded value 0x3C5FE025, passed as a parameter by `EnumWindows`
4. if it matches, open the corresponding process and store the handle at 0x40EF50.



Since none of the windows I had in my system matched the required one, here is my personal hack: I let the callback routine run to get the hash of one of the windows I had (I chose the *"Process Hacker"* window just to be sure it was unique). I then restarted the execution and patched the code at runtime so that `EnumWindows` would pass the chosen hash, therefore injecting the *Process Hacker* process.

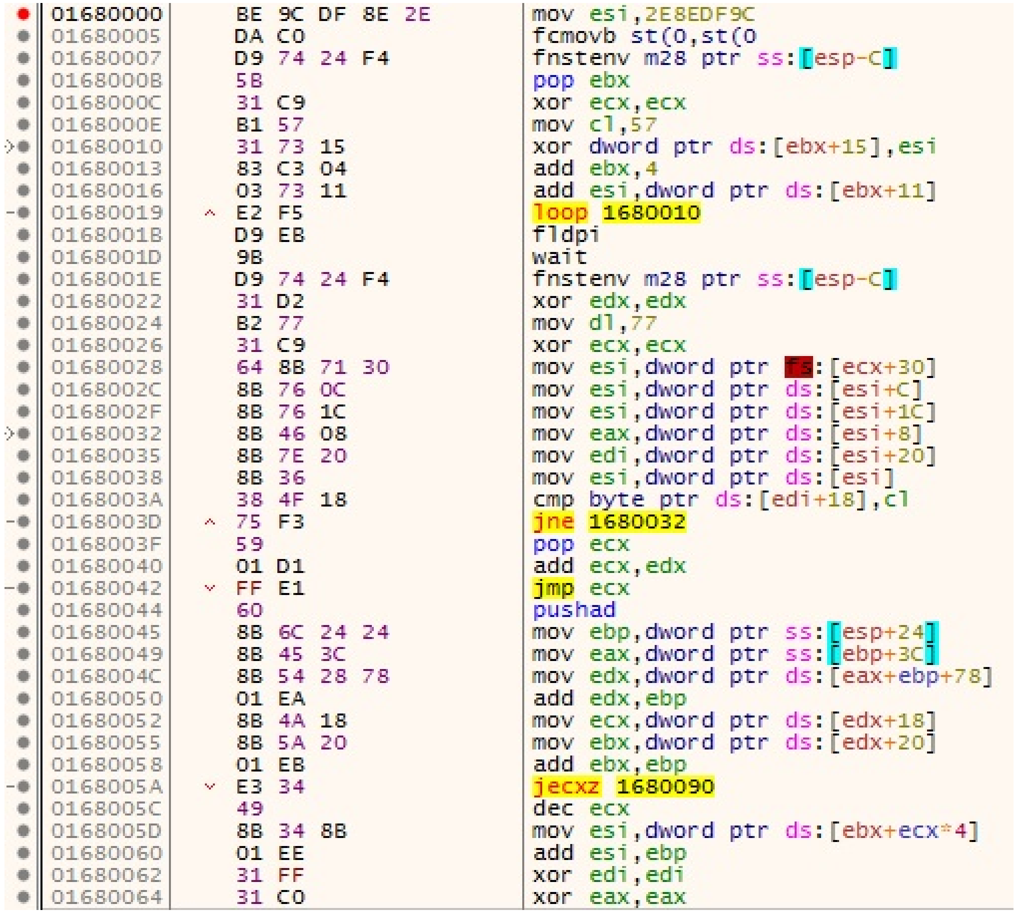
Regarding the injected code, it is pretty straight forward, since it is stored almost in clear at 0x40E000. “Almost” because the first 4 bytes are the only encrypted part and they are correctly decrypted if the `PEB.BeingDebugged` flag is set. The injection function is located at 0x401130.

NOTE: the shellcode looks like a sequence of junk instructions, which probably means it is self-modifying code - later I got the confirmation from hasherezade that she used the Metasploit polymorphic encoder *shikata ga nai*.



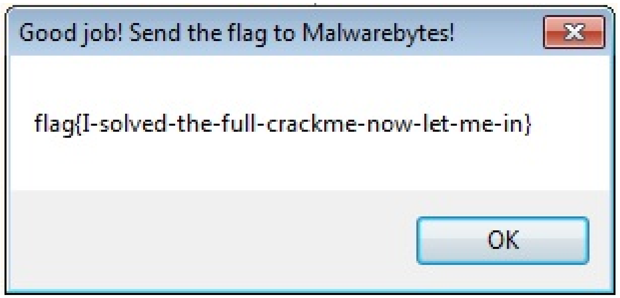
```
0007E000  BE 9C DF 8E 2E        mov esi,2E8EDF9C
0007E005  DA C0                 fcmovb st(0,st(0
0007E007  D9 74 24 F4           fnstenv m28 ptr ss:[esp-C]
0007E00B  5B                    pop ebx
0007E00C  31 C9                 xor ecx,ecx
0007E00E  B1 57                 mov cl,57
0007E010  31 73 15              xor dword ptr ds:[ebx+15],esi
0007E013  83 C3 04              add ebx,4
0007E016  03 73 11              add esi,dword ptr ds:[ebx+11]
0007E019  E2 69                 loop 7E084
0007E01B  06                    push es
0007E01C  65 B5 48              mov ch,48
0007E01F  CD 5E                 int 5E
0007E021  3E 5B                 pop ebx
0007E023  FC                    cld
0007E024  2D C9 AA C9 36        sub eax,36C9AAC9
0007E029  BD BD F9 3D B7        mov ebp,B73DF9BD
0007E02E  31 71 37              xor dword ptr ds:[ecx+37],esi
0007E031  24 C2                 and al,C2
0007E033  C3                    ret
0007E034  B0 DF                 mov al,DF
0007E036  AA                    stosb byte ptr es:[edi],al
0007E037  EB 4B                 jmp 7E084
0007E039  E9 6A A3 53 63        jmp 635B83A8
0007E03E  79 62                 jns 7E0A2
0007E040  65 5A                 pop edx
0007E042  82 74 05 D7 10        xor byte ptr ss:[ebp+eax-29],10
0007E047  53                    push ebx
0007E048  E2 6C                 loop 7E0B6
0007E04A  AD                    lodsd eax,dword ptr ds:[esi]
0007E04B  A7                    cmpsd dword ptr ds:[esi],dword ptr es:[edi]
0007E04C  61                    popad
0007E04D  26 05 A0 74 2D DE     add eax,DE2D74A0
0007E053  1A 6F 3A              sbb ch,byte ptr ds:[edi+3A]
0007E056  BA BA 8E D7 D9        mov edx,D9D78EBA
0007E05B  8F                    ???
0007E05C  D9 AC 29 7B D8 5C 60  fldcw word ptr ds:[ecx+ebp+605CD87B]
0007E063  84 EA                 test dl,ch
0007E065  60                    pushad
```

shellcode before self-modification…



… and after

At this point everything is set, so we let the rest of the code run and the final message box pops up, containing the long awaited flag.



Conclusions

Kudos to haserezade for creating a challenge that features many different techniques used by malware, from anti-analysis to process injection. Since these techniques are displayed clearly, without any obfuscation, this is a good reference to learn them, but also a good exercise to redo from time to time to keep things fresh.

Written on October 19, 2017