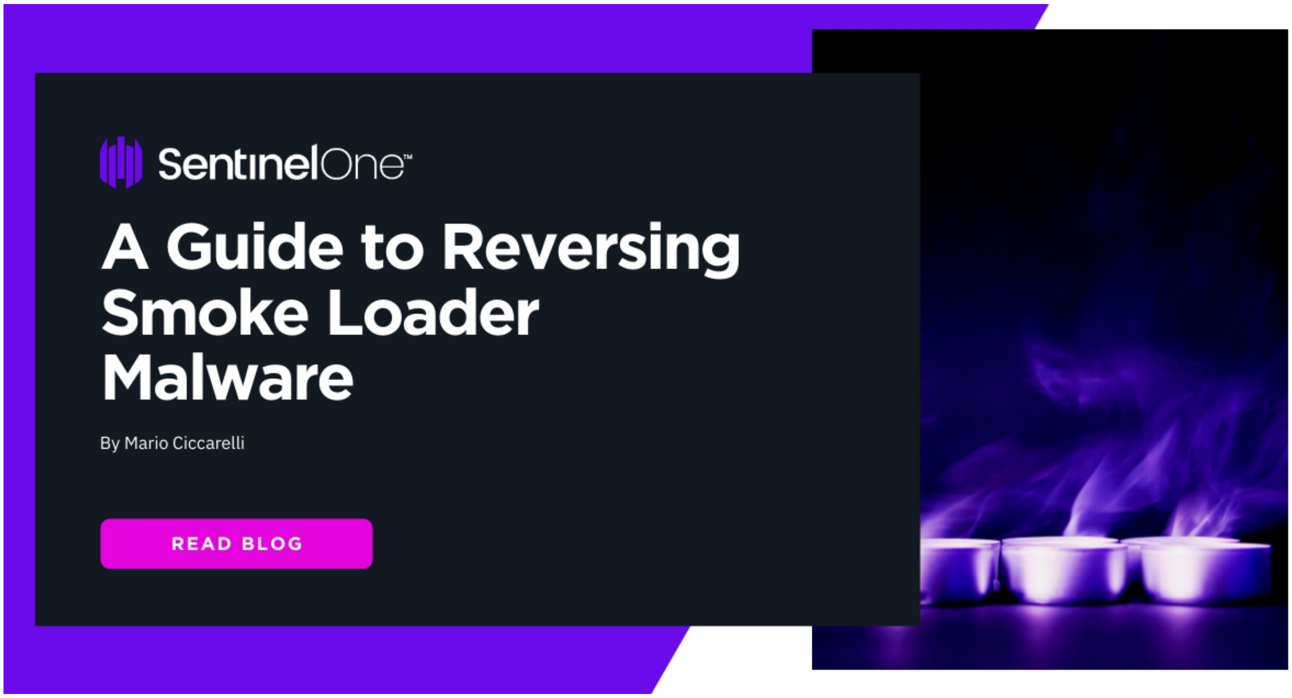
Going Deep | A Guide to Reversing Smoke Loader Malware

sentinelone.com/blog/going-deep-a-guide-to-reversing-smoke-loader-malware/

November 21, 2019



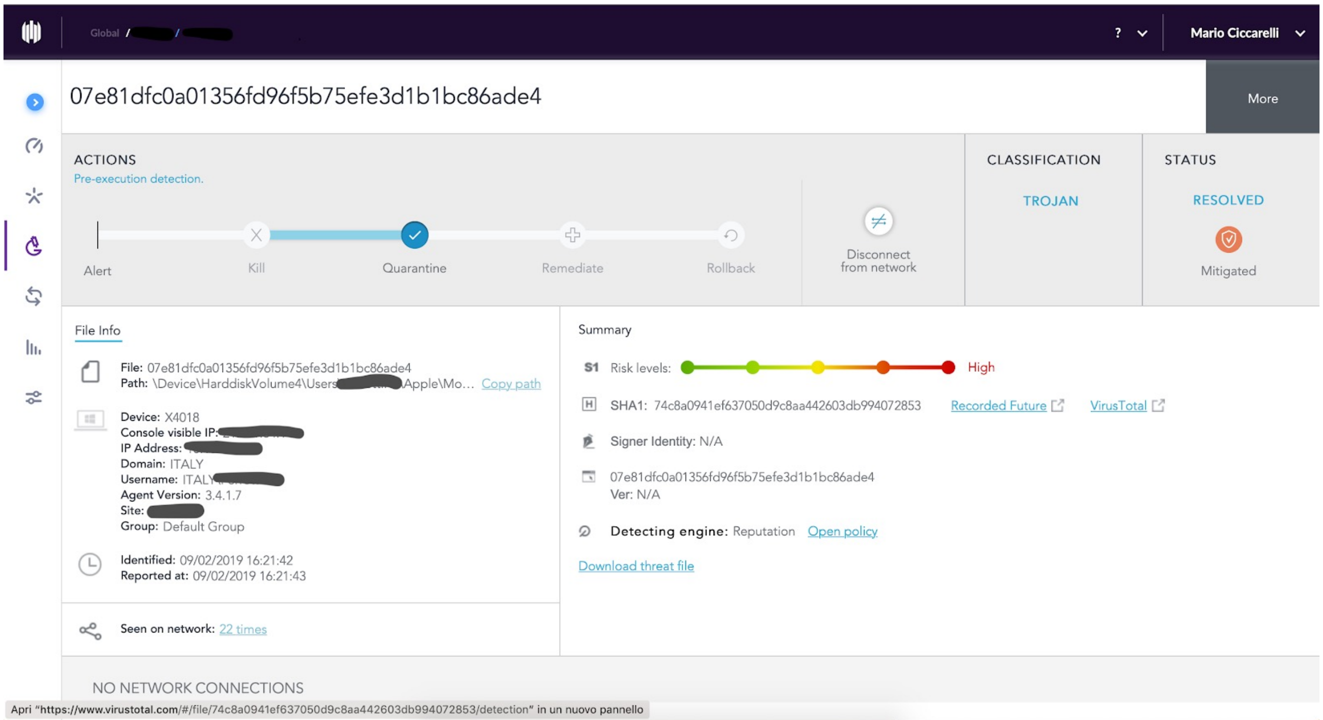
Working in infosec and supporting clients and SOCs has always exposed me to a huge number of alerts and incidents. Some of these are more interesting than others. Recently we stumbled across a particular sample of Smoke Loader malware. Smoke Loader has been in-the-wild since circa 2013 and is often used to distribute additional malicious components or artifacts. While the sample is not new, it did prove to be a good opportunity to revisit this threat and walk through some of the internals.

This alert was raised against a suspicious file, classified as a trojan, that was killed and quarantined. What raised my curiosity was the number of detections over only a few hours, always from the same workstation, and only from the same user.

Status	File Details	Endpoints	Reported Time	Sites	Classification	Actions Done
	07e81dfc0a01356fd96f5b75efe3d1b1bc86ade4	X4018	Sep 2nd 2019 • 16:21:43		Trojan	Killed, Quarantined
	07e81dfc0a01356fd96f5b75efe3d1b1bc86ade4.up...	X4018	Sep 2nd 2019 • 16:21:43		Trojan	Killed, Quarantined
	07e81dfc0a01356fd96f5b75efe3d1b1bc86ade4	X4018	Sep 2nd 2019 • 11:31:16		Trojan	Killed, Quarantined
	07e81dfc0a01356fd96f5b75efe3d1b1bc86ade4	X4018	Sep 2nd 2019 • 11:31:16		Trojan	Killed, Quarantined
	07e81dfc0a01356fd96f5b75efe3d1b1bc86ade4.up...	X4018	Sep 2nd 2019 • 11:31:16		Trojan	Killed, Quarantined
	07e81dfc0a01356fd96f5b75efe3d1b1bc86ade4	X4018	Sep 2nd 2019 • 11:09:14		Trojan	Killed, Quarantined
	07e81dfc0a01356fd96f5b75efe3d1b1bc86ade4	X4018	Sep 2nd 2019 • 11:09:14		Trojan	Killed, Quarantined
	07e81dfc0a01356fd96f5b75efe3d1b1bc86ade4.up...	X4018	Sep 2nd 2019 • 11:09:14		Trojan	Killed, Quarantined
	07e81dfc0a01356fd96f5b75efe3d1b1bc86ade4	X4018	Sep 2nd 2019 • 10:29:02		Trojan	Killed, Quarantined

Knowing that the threat was killed without doing any harm, I decided to dig into it a bit more. Just looking at the SentinelOne console, I was able to see :

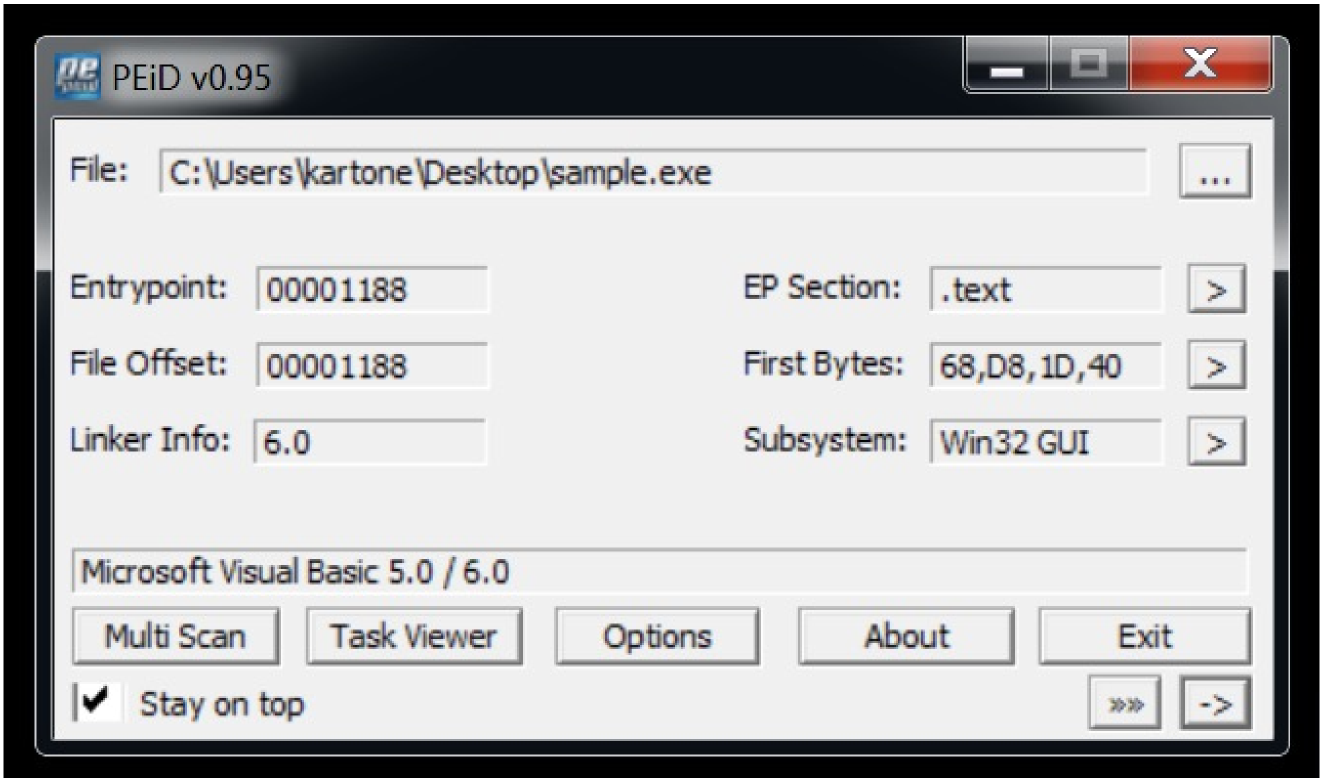
- The full path where the detection was made.
- The associated risk level is High: this implies that it’s a positive detection.
- File unique hash that can be tested for any public Indicators of Compromise (IoC).



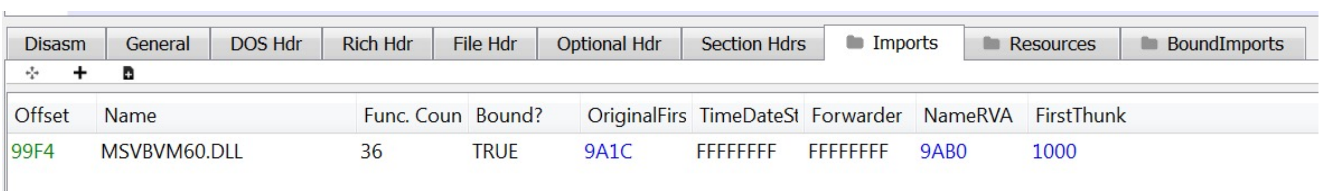
In order to do a walk-through of malware reverse engineering steps, I downloaded the threat file and started the analysis.

First Layer: A Packed VB Win32 Program

With the downloaded file in my pocket, I quickly fired up an isolated analysis machine equipped with the Flare tools and started to investigate. At first glance, the sample appears to be a Visual Basic program leveraging Win32 APIs.

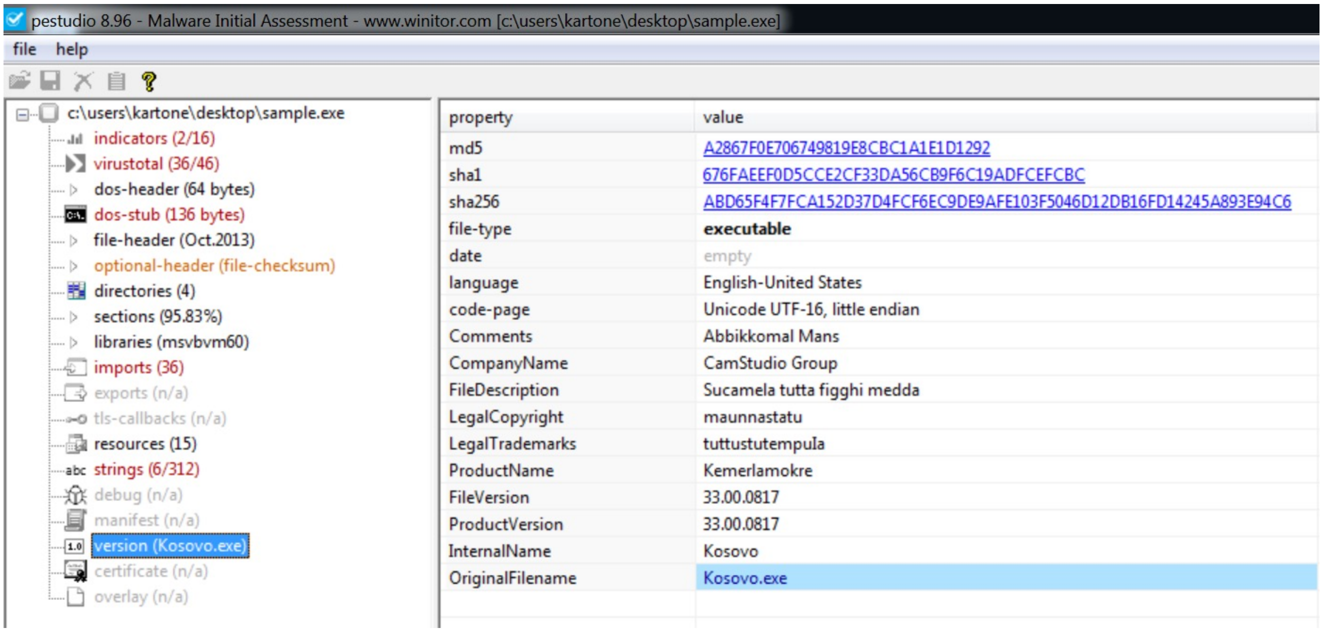


Let’s see what else we can get from its headers. Looks like pretty standard information, confirming a Visual Basic program due to its import table.



Offset	Name	Func. Coun	Bound?	OriginalFirs	TimeDateSt	Forwarder	NameRVA	FirstThunk
99F4	MSVBVM60.DLL	36	TRUE	9A1C	FFFFFFFF	FFFFFFFF	9AB0	1000

We can spot some rude and folkloristic words inside the binary strings, some of which make me think of a regional dialect of southern Italy.

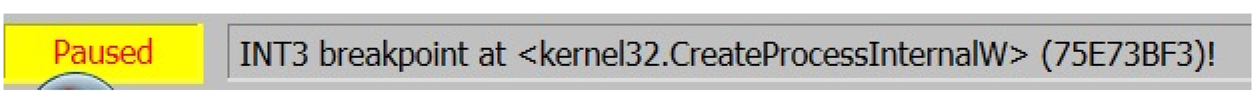


With a bit of experience, we can safely assume that the file is packed with an external layer of Visual Basic that tries to stop, or at least slow down, static analysis. But what about its runtime behavior?

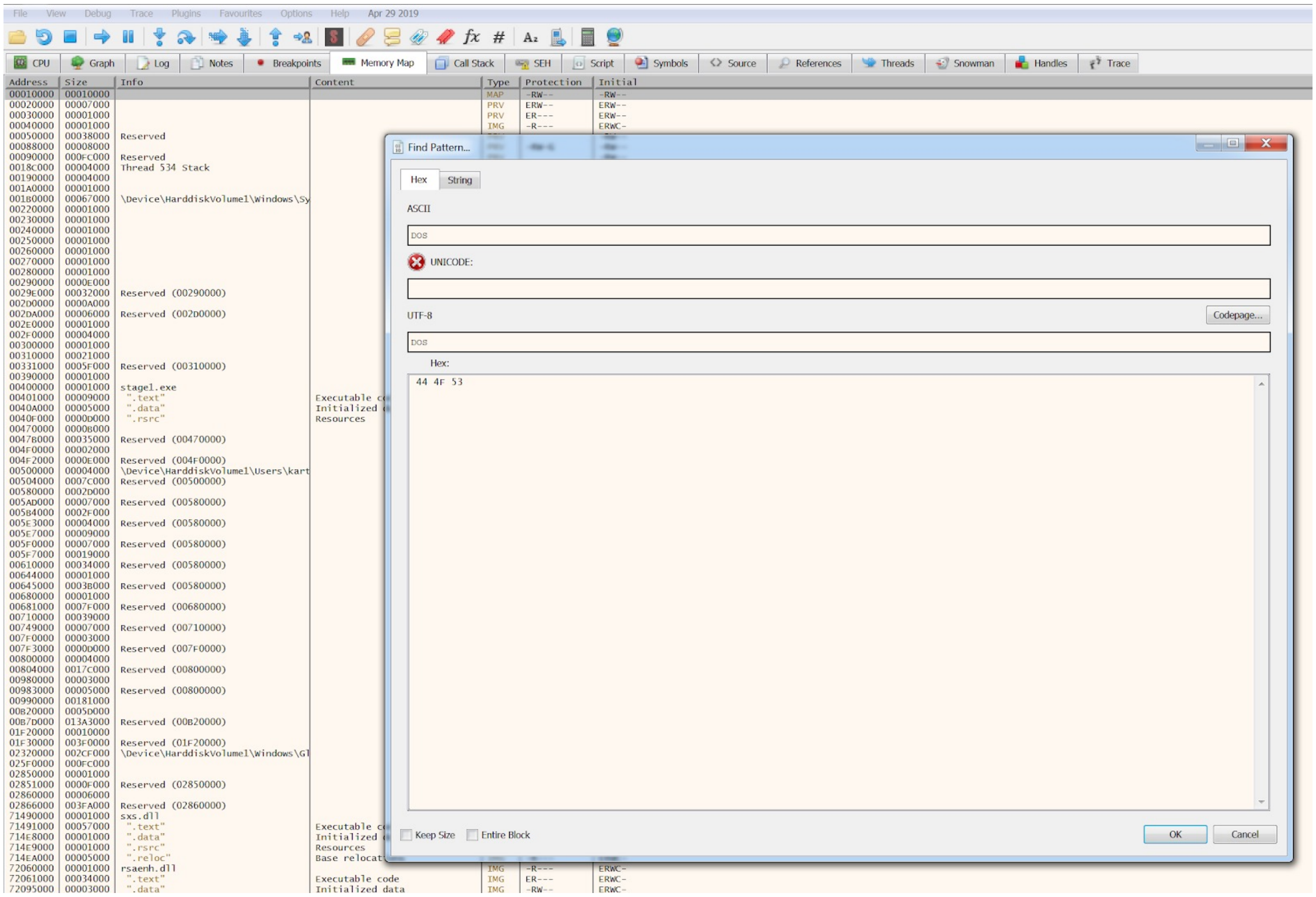
Observing the sample during runtime, we can observe the process injection: this behaviour is common for VB packers and luckily for us, is often trivial to defeat.

Defeating Visual Basic Packer

We won't spend too much time on this: there are plenty of resources on how to unpack such packers and I highly recommend the OALabs Youtube video tutorials. It's necessary and enough to put a breakpoint at `CreateProcessInternalW` API inside the debugger to stop the execution at the right time.



At this point, somewhere in memory, there is a PE file ready to be run. We only need to find it. To do so, we can search the entire memory map for a clue: I decided to search for the "*DOS*" substring that can usually be found as part of the "*This program cannot be run in DOS mode*" string within the PE.



We got plenty of results for the string, whose hex is `44 4F 53`.

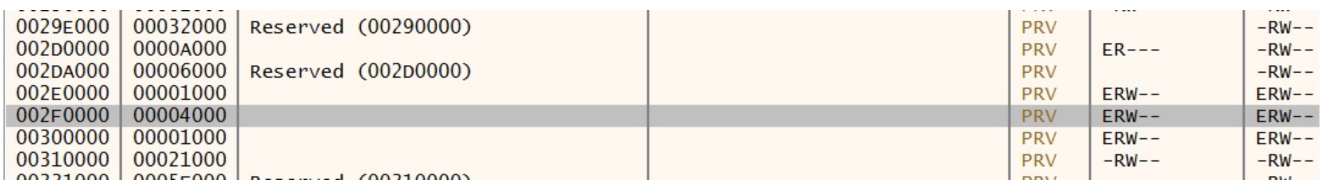
Address	Data
002F0094	44 4F 53
0040006C	44 4F 53
7149006C	44 4F 53
7206006C	44 4F 53
721B006C	44 4F 53
7294006C	44 4F 53
73F1006C	44 4F 53
73F3006C	44 4F 53
7508006C	44 4F 53
7509006C	44 4F 53
750F006C	44 4F 53
750F3F61	44 4F 53
758B006C	44 4F 53
758C006C	44 4F 53
75C5006C	44 4F 53
75CB006C	44 4F 53
75E5006C	44 4F 53
75F135FF	44 4F 53
75F13617	44 4F 53
75F1C59C	44 4F 53
75F6006C	44 4F 53
7619006C	44 4F 53
764F006C	44 4F 53
7650006C	44 4F 53
765B006C	44 4F 53
772D006C	44 4F 53
7746006C	44 4F 53
7756006C	44 4F 53
7758006C	44 4F 53
775FB7BA	44 4F 53
77613AE0	44 4F 53
77630A80	44 4F 53
7768006C	44 4F 53
7787006C	44 4F 53
77B9006C	44 4F 53
77C1699A	44 4F 53
77CA074D	44 4F 53
77D7006C	44 4F 53
77D8A9E0	44 4F 53
77D97F5A	44 4F 53

However, we are particularly interested in just a few locations. Usually the executable is loaded at `0x00400000` address, so the result we had at `0x0040006C` looks like our executable itself.

Things become particularly interesting at `0x002F0094`, which we can follow in the memory dump.

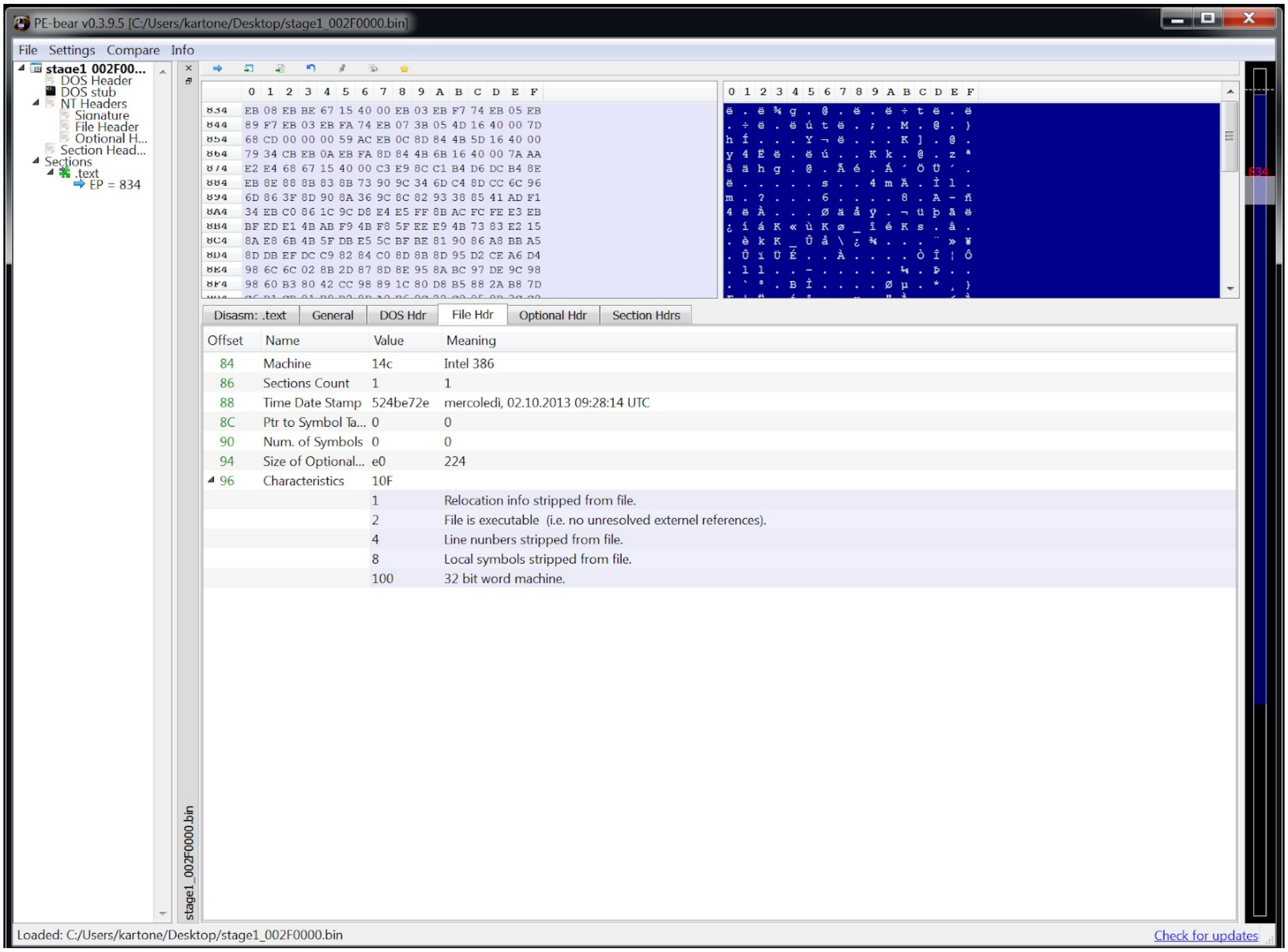
Address	Hex	ASCII
002F0000	28 00 00 00 50 01 00 00 10 00 00 00 01 00 18 00	(...P...........
002F0010	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
002F0020	00 00 00 00 00 00 00 00 4D 5A 80 00 01 00 00 00	........MZ......
002F0030	04 00 10 00 FF FF 00 00 40 01 00 00 00 00 00 00	....ÿÿ..@.......
002F0040	40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	@...............
002F0050	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
002F0060	00 00 00 00 80 00 00 00 0E 1F BA 0E 00 B4 09 CD	..........º..´.Í
002F0070	21 B8 01 4C CD 21 54 68 69 73 20 70 72 6F 67 72	!¸.LÍ!This progr
002F0080	61 6D 20 63 61 6E 6E 6F 74 20 62 65 20 72 75 6E	am cannot be run
002F0090	20 69 6E 20 44 4F 53 20 6D 6F 64 65 2E 0D 0A 24	in DOS mode...$
002F00A0	00 00 00 00 00 00 00 00 50 45 00 00 4C 01 01 00	........PE..L...
002F00B0	2E E7 4B 52 00 00 00 00 00 00 00 00 E0 00 0F 01	.çKR........à...
002F00C0	0B 01 01 46 00 2E 00 00 00 00 00 00 00 00 00 00	...F............
002F00D0	34 16 00 00 00 10 00 00 00 00 00 00 00 00 40 00	4.............@.
002F00E0	00 10 00 00 00 02 00 00 01 00 00 00 00 00 00 00	....
002F00F0	04 00 00 00 00 00 00 00 00 40 00 00 00 02 00 00	.........@......
002F0100	5E 85 00 00 02 00 00 00 00 10 00 00 00 10 00 00	^...............
002F0110	00 00 01 00 00 00 00 00 00 00 00 00 10 00 00 00	....
002F0120	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
002F0130	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
002F0140	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....

A memory region with a PE file inside, mapped as Executable, Read, Write. This is definitely our injected file.



0029E000	00032000	Reserved (00290000)		PRV		-RW--
002D0000	0000A000			PRV	ER---	-RW--
002DA000	00006000	Reserved (002D0000)		PRV		-RW--
002E0000	00001000			PRV	ERW--	ERW--
002F0000	00004000			PRV	ERW--	ERW--
00300000	00001000			PRV	ERW--	ERW--
00310000	00021000			PRV	-RW--	-RW--

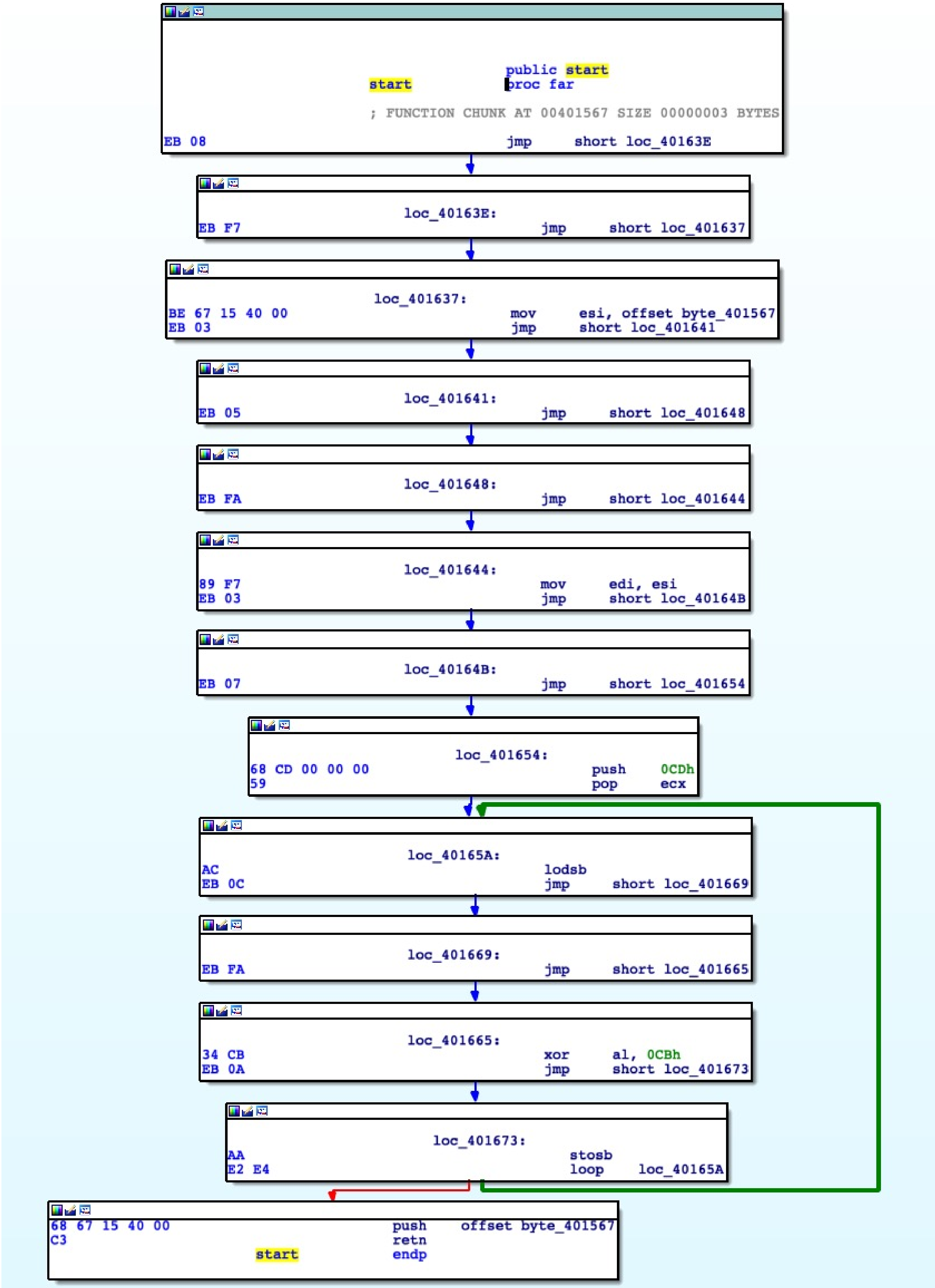
We can simply dump out this memory region to file, clean the junk before the MZ header and analyze its headers.



It seems like a legitimate executable, but something is going on: no imports at all. This is interesting!

Second layer: Static Analysis

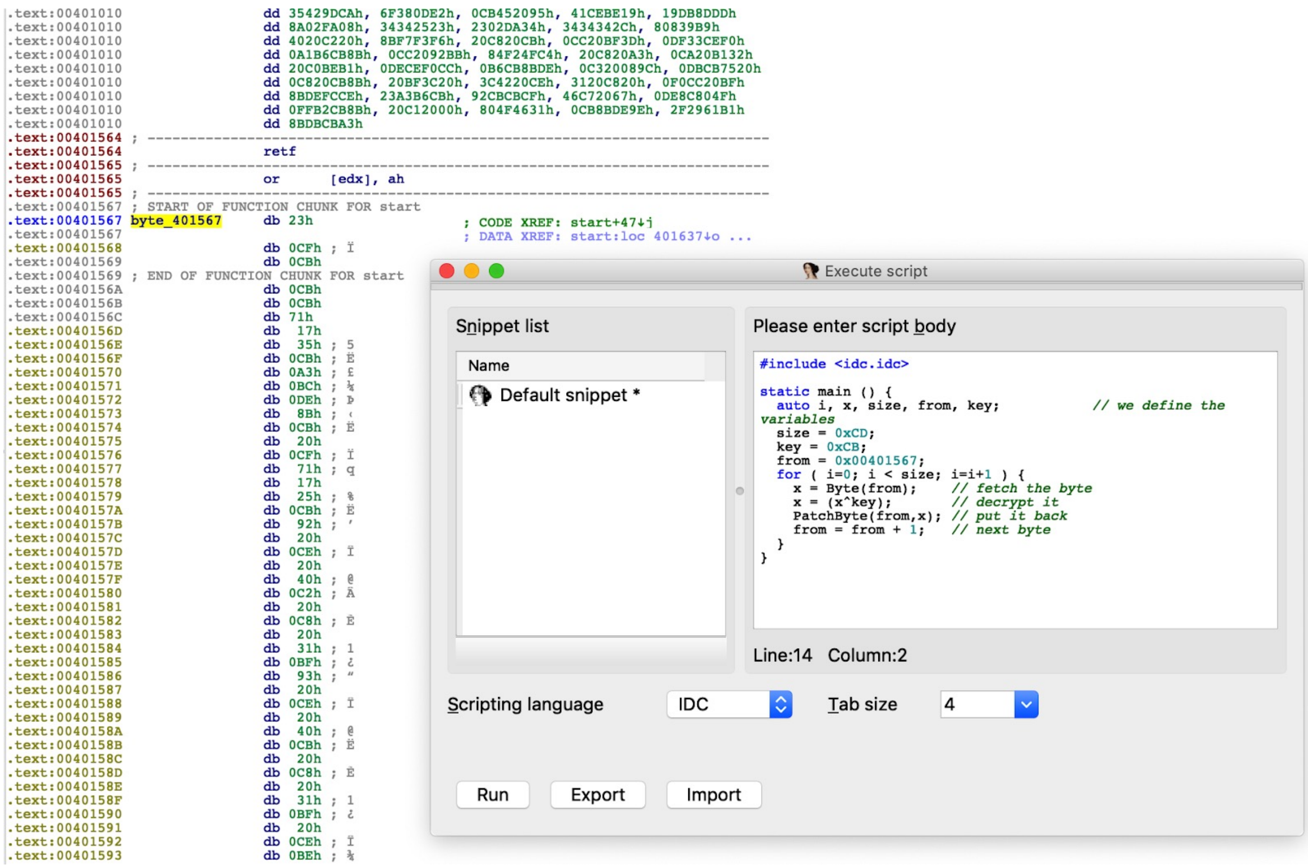
When we load this new executable in IDA Pro, this was the only chunk of code that was disassembled.


public start
start proc far
; FUNCTION CHUNK AT 00401567 SIZE 00000003 BYTES
EB 08 jmp short loc_40163E
loc_40163E:
EB F7 jmp short loc_401637
loc_401637:
BE 67 15 40 00 mov esi, offset byte_401567
EB 03 jmp short loc_401641
loc_401641:
EB 05 jmp short loc_401648
loc_401648:
EB FA jmp short loc_401644
loc_401644:
89 F7 mov edi, esi
EB 03 jmp short loc_40164B
loc_40164B:
EB 07 jmp short loc_401654
loc_401654:
68 CD 00 00 00 push 0CDh
59 pop ecx
loc_40165A:
AC lodsb
EB 0C jmp short loc_401669
loc_401669:
EB FA jmp short loc_401665
loc_401665:
34 CB xor al, 0CBh
EB 0A jmp short loc_401673
loc_401673:
AA stosb
E2 E4 loop loc_40165A
68 67 15 40 00 push offset byte_401567
C3 retn
start endp

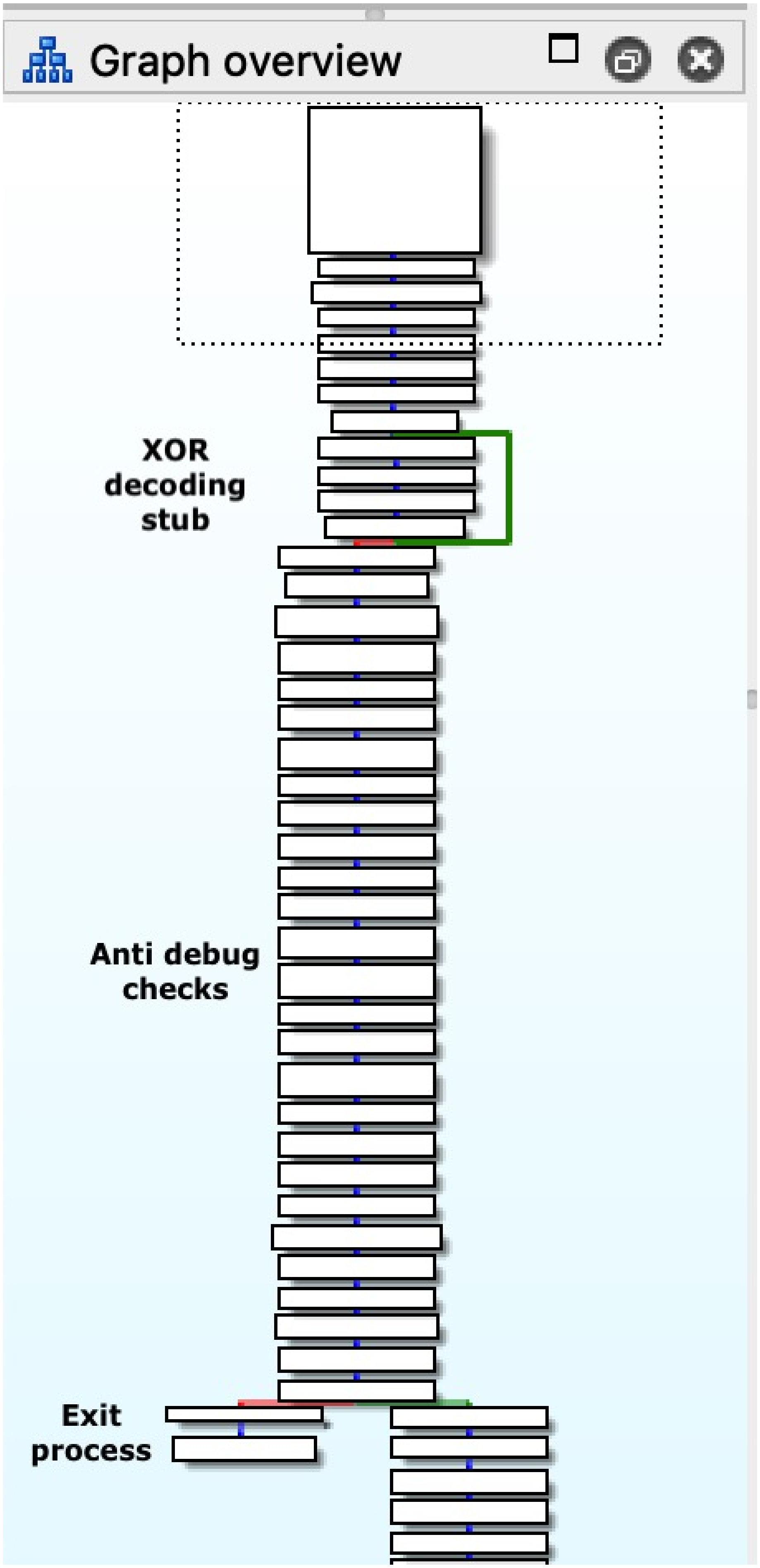
Here we recognize that it's a **XOR** loop that will decode, from address `0x00401567`, a blob of code with the size of `0xCD` bytes with a **XOR** key equal to `0xCB`. At the end of the loop, the same starting address `0x00401567` is pushed onto the stack and with the **RET** instruction the program flow will be branched over there.

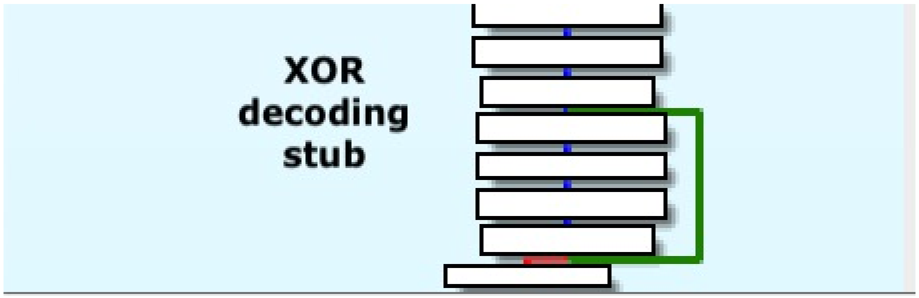
Decoding the Buffer

With a little bit of IDA scripting, we can **XOR** the encrypted buffer and move forward in the analysis.



After de-xoring the buffer we are met with a mixture of anti-disassembly and anti-debug techniques. It is now possible to map the purpose of the code blocks.


Graph overview
XOR decoding stub
Anti debug checks
Exit process



Inside this code, we can observe plenty of tricks that try to fool the disassembly flow. A few examples:

- Abusing **CALL** and **RET** instruction to mess up function boundaries. The **CALL** instruction will push the return address onto the stack. The **RET** instruction will then pop off this address into the EIP register, which effectively makes these two instructions useless. However, these few opcodes make IDA think that the function ends there and that the next instruction is the end of another function.
- Abusing branch instructions that do nothing: **CALL** `<address>` and at `<address>`: **POP** `<reg>`. It's the easiest way to get an address inside the EIP register and so to control the program's flow.
- Abusing **JMP** instructions: simply putting a lot of **JMP** instructions that will jump back and forth only to make the life of the analyst miserable.

Obfuscated with these techniques, the malware checks if it's being debugged. The code that implements this check is nothing complicated: it queries certain flags of the PEB in order to spot the debugger, `IsDebuggerPresent`.

```
mov eax, fs:[30h] ; Process Environment Block

cmp b [eax+2], 0 ; check BeingDebugged
jne being_debugged
```

As said, this code is heavily obfuscated with junk jumps and a lot of instructions with the only purpose of increasing complexity of analysis. As an example, this little chunk of code is the final part of a dozen lines of code used to put value `0x30` inside the EAX register with the purpose of locating the PEB.

```
; START OF FUNCTION CHUNK FOR start

loc_4015A8:
                push    3
                bswap   eax
                jmp     short loc_4015B3
; END OF FUNCTION CHUNK FOR start
```

```
loc_4015B3:
                jmp     short loc_4015AF
; END OF FUNCTION CHUNK FOR start
```

```
; START OF FUNCTION CHUNK FOR start

loc_4015AF:
                mov     ecx, eax
                jmp     short loc_4015B6
```

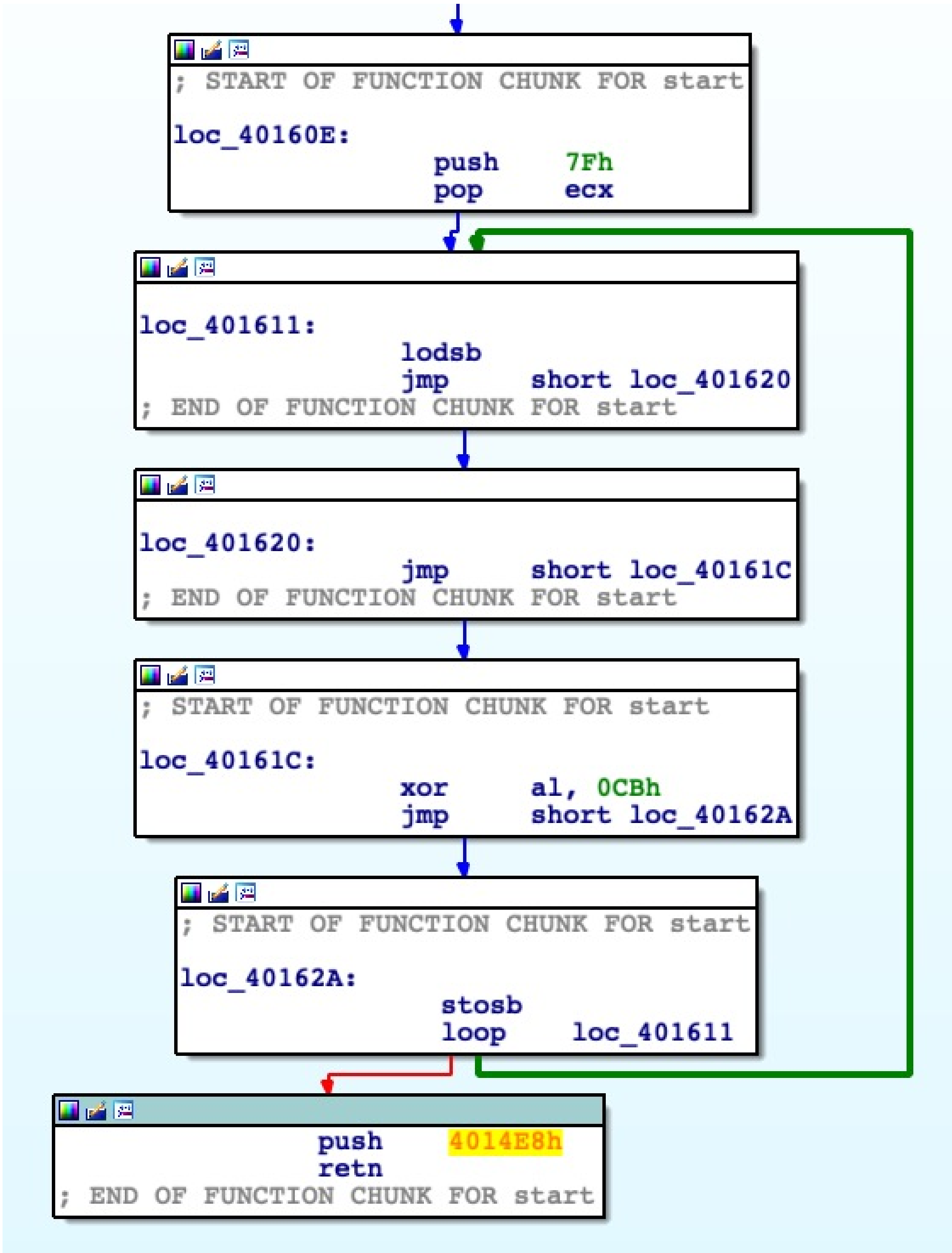
```
; START OF FUNCTION CHUNK FOR start

loc_4015B6:
                pop     eax
                mul     ecx
                jmp     short loc_4015C1
; END OF FUNCTION CHUNK FOR start
```

```
loc_4015C1:
                jmp     short loc_4015BC
; END OF FUNCTION CHUNK FOR start
```

```
; START OF FUNCTION CHUNK FOR start

loc_4015BC:
                mov     eax, fs:[eax]
                jmp     short loc_4015C4
```

```
; START OF FUNCTION CHUNK FOR start

loc_4015C4:
                jmp     short loc_4015CE
; END OF FUNCTION CHUNK FOR start
```

At the end of this function, we spot another **XOR** stub decoding routine that will decode another blob of code and, after that, redirect the execution flow. Decoding will start at address `0x004014E8`, with a buffer size of `0x7F` and the same **XOR** key `0xCB`.

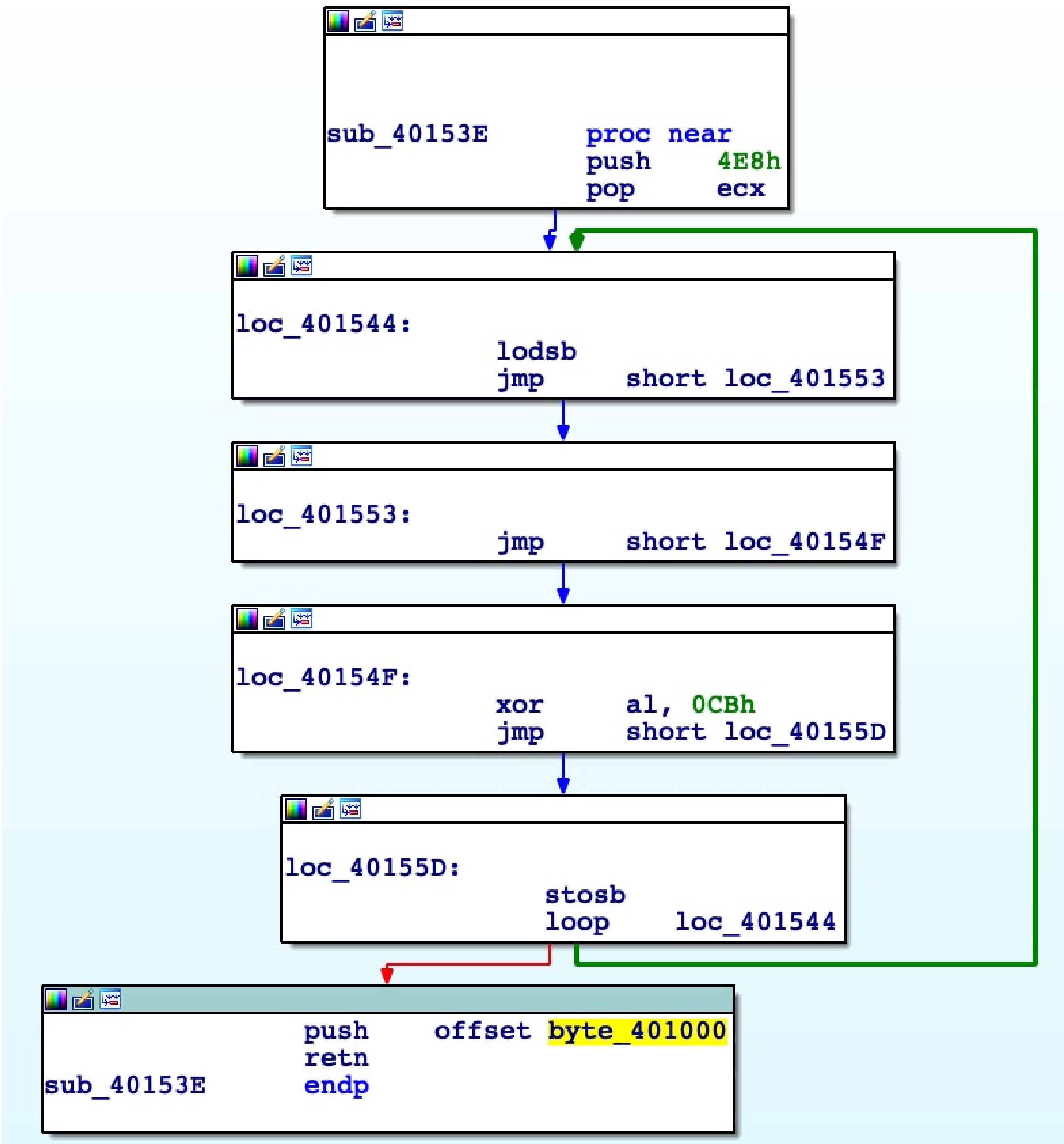


As before, we can proceed in the static analysis, manually decoding this buffer with the same script.

But wait! Here we go again, another anti-debugging trick, `NtGlobalFlag` check:

```
mov eax, fs:[30h] ; Process Environment Block
mov al, [eax+68h] ; NtGlobalFlag
and al, 70h
cmp al, 70h
je  being_debugged
```

This chunk of code checks if the process is attached to a debugger and, if it goes well, another **XOR** decoding stub starts from address `0x00401000`, with buffer size `0x4E8` and **XOR** key `0xCB`.



After decoding the new buffer, we need to face another anti-disassembly trick; namely, **JMP** instructions with a constant value. This is the most common trick used by malware to fool static analysis. Basically, it creates jumps into a new location plus one or a few bytes. It results in an erroneous interpretation of the opcode by the disassembler. It's trivial to defeat but time intensive.

IAT Resolution at Runtime

At address `0x00401000` there's a simple call to another address `0x00401049`, where it starts to become interesting as the malware appears to dynamically resolve its imports. As we noted before, the binary header analysis showed no imports at all. With this code, from the PEB location found earlier, the malware finds the base address of `ntdll.dll`.

```
loc_401049:                              ; CODE XREF: .text:loc 401000↑p
                mov     ebp, esp
                mov     edi, PEB_location
                mov     esi, [edi+0Ch]
                mov     esi, [esi+1Ch]
                mov     edx, [esi+8]
```

But how is this happening? In all recent Windows versions, the GS register points to a data structure called the Thread Environment Block (TEB). At offset `0x30` of the TEB, there's another data structure, namely the Process Environment Block (PEB) we saw earlier.

77EE5000	0000B000	Reserved (77D70000)		IMG		ERWC-
7EFB0000	00023000			MAP	-R---	-R---
7EFDB000	00003000	Thread 3F8 WoW64 TEB		PRV	-RW--	-RW--
7EFDE000	00001000	PEB		PRV	-RW--	-RW--
7EFDF000	00001000			PRV	-RW--	-RW--
7EFE0000	00005000			MAP	-R---	-R---

We can inspect these data structures with the help of Microsoft public symbols and WinDBG.

Command

```
0:000> dt_TEB
ntdll!_TEB
   +0x000 NtTib            : _NT_TIB
   +0x01c EnvironmentPointer : Ptr32 Void
   +0x020 ClientId         : _CLIENT_ID
   +0x028 ActiveRpcHandle  : Ptr32 Void
   +0x02c ThreadLocalStoragePointer : Ptr32 Void
   +0x030 ProcessEnvironmentBlock : Ptr32 _PEB
   +0x034 LastErrorValue   : Uint4B
   +0x038 CountOfOwnedCriticalSections : Uint4B
```

With the same tools we can inspect the PEB too:

```
0:000> dt_PEB
ntdll!_PEB
   +0x000 InheritedAddressSpace : UChar
   +0x001 ReadImageFileExecOptions : UChar
   +0x002 BeingDebugged    : UChar
   +0x003 BitField         : UChar
   +0x003 ImageUsesLargePages : Pos 0, 1 Bit
   +0x003 IsProtectedProcess : Pos 1, 1 Bit
   +0x003 IsLegacyProcess  : Pos 2, 1 Bit
   +0x003 IsImageDynamicallyRelocated : Pos 3, 1 Bit
   +0x003 SkipPatchingUser32Forwarders : Pos 4, 1 Bit
   +0x003 SpareBits        : Pos 5, 3 Bits
   +0x004 Mutant           : Ptr32 Void
   +0x008 ImageBaseAddress : Ptr32 Void
   +0x00c Ldr              : Ptr32 _PEB_LDR_DATA
   +0x010 ProcessParameters : Ptr32 _RTL_USER_PROCESS_PARAMETERS
   +0x014 SubSystemData    : Ptr32 Void
   +0x018 ProcessHeap      : Ptr32 Void
   +0x01c FastPebLock      : Ptr32 _RTL_CRITICAL_SECTION
   +0x020 AtlThunkSListPtr : Ptr32 Void
   +0x024 IFEOKey          : Ptr32 Void
   +0x028 CrossProcessFlags : Uint4B
```

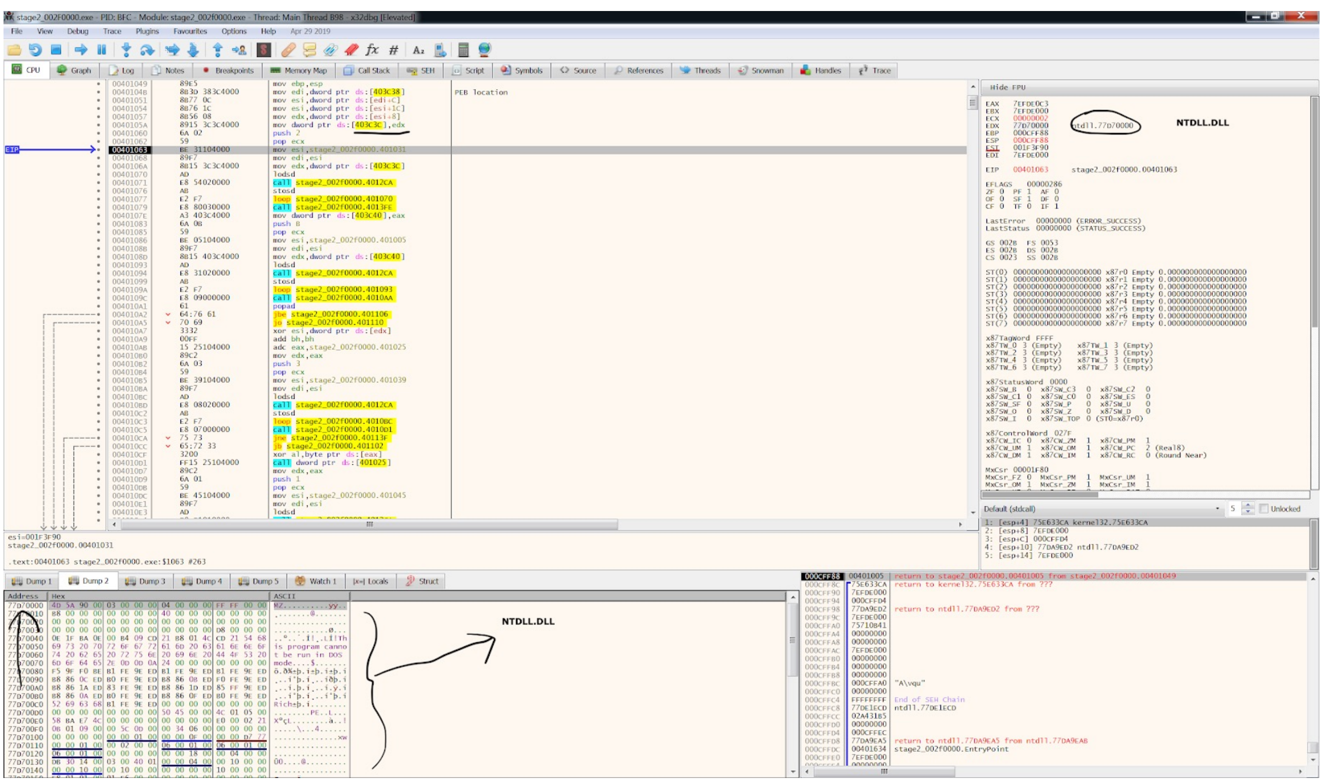
With the third instruction, we are following the offset `0x0C`, the `_PEB_LDR_DATA` structure. This structure is fairly important because it contains a pointer, `InInitializationOrderModuleList`, to the head of a double-linked list that contains the NTDLL loader data structures for the loaded modules.

```
0:000> dt _PEB_LDR_DATA
ntdll!_PEB_LDR_DATA
   +0x000 Length           : Uint4B
   +0x004 Initialized      : UChar
   +0x008 SsHandle         : Ptr32 Void
   +0x00c InLoadOrderModuleList : _LIST_ENTRY
   +0x014 InMemoryOrderModuleList : _LIST_ENTRY
   +0x01c InInitializationOrderModuleList : _LIST_ENTRY
   +0x024 EntryInProgress  : Ptr32 Void
   +0x028 ShutdownInProgress : UChar
   +0x02c ShutdownThreadId : Ptr32 Void
```

Each item in the list is a pointer to an LDR_DATA_TABLE_ENTRY structure. If we inspect this structure, we get the `DLLBase`.

```
0:000> dt _LDR_DATA_TABLE_ENTRY
ntdll!_LDR_DATA_TABLE_ENTRY
   +0x000 InLoadOrderLinks : _LIST_ENTRY
   +0x008 InMemoryOrderLinks : _LIST_ENTRY
   +0x010 InInitializationOrderLinks : _LIST_ENTRY
   +0x018 DllBase          : Ptr32 Void
   +0x01c EntryPoint       : Ptr32 Void
   +0x020 SizeOfImage      : Uint4B
   +0x024 FullDllName      : _UNICODE_STRING
   +0x02c BaseDllName      : _UNICODE_STRING
   +0x034 Flags            : Uint4B
   +0x038 LoadCount        : Uint2B
   +0x03a TlsIndex         : Uint2B
   +0x03c HashLinks        : _LIST_ENTRY
   +0x03c SectionPointer   : Ptr32 Void
   +0x040 CheckSum         : Uint4B
   +0x044 TimeDateStamp    : Uint4B
   +0x044 LoadedImports    : Ptr32 Void
   +0x048 EntryPointActivationContext : Ptr32 _ACTIVATION_CONTEXT
   +0x04c PatchInformation : Ptr32 Void
   +0x050 ForwarderLinks   : _LIST_ENTRY
   +0x058 ServiceTagLinks  : _LIST_ENTRY
   +0x060 StaticLinks      : _LIST_ENTRY
   +0x068 ContextInformation : Ptr32 Void
   +0x06c OriginalBase     : Uint4B
   +0x070 LoadTime         : _LARGE_INTEGER
```

Looking at this inside the debugger helps to shed some light:

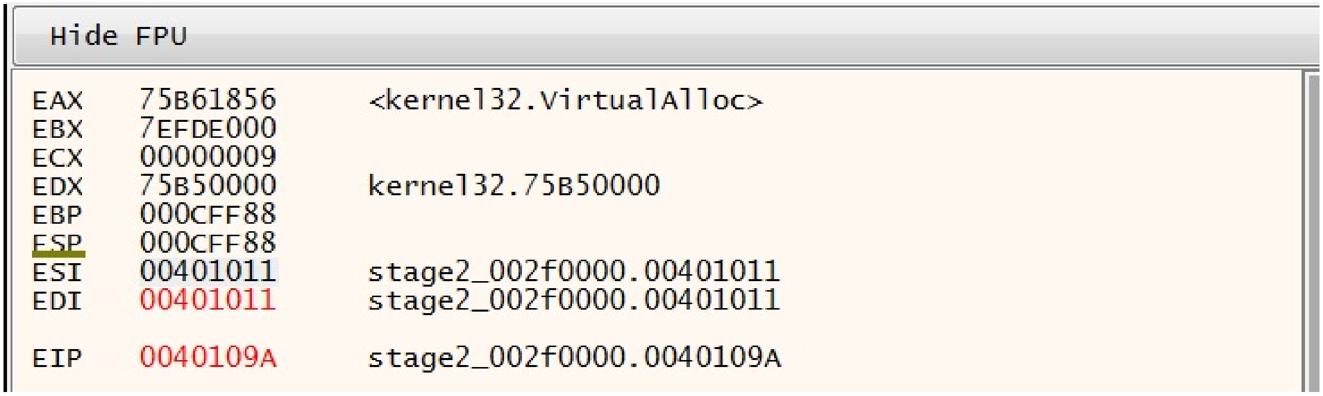


We got the base address of module `ntdll.dll` into EDX register, because this is the first module loaded into every process in a Windows environment. We have added comments and renamed select functions to clear up some of the observables.

```
ResolveImports  proc near              ; CODE XREF: sub 4014E8:loc 401000↑p

; FUNCTION CHUNK AT .text:00401106 SIZE 00000006 BYTES
; FUNCTION CHUNK AT .text:00401110 SIZE 0000002C BYTES
; FUNCTION CHUNK AT .text:00401180 SIZE 0000000E BYTES

                mov     ebp, esp
                mov     edi, PEB_location
                mov     esi, [edi+0Ch]
                mov     esi, [esi+1Ch]
                mov     edx, [esi+8]

loc_40105A:
                mov     ntdll_location, edx
                push    2
                pop     ecx
                mov     esi, offset dword_401031 ; Some hash of strstr function
                mov     edi, esi
                mov     edx, ntdll_location

DecryptionLoop:                        ; CODE XREF: ResolveImports+2E↓j
                lodsd
                call    DecryptionFunction
                stosd
                loop    DecryptionLoop
                call    Wrapper_LdrGetDLLHandle
                mov     dword_403C40, eax
                push    0Bh
                pop     ecx
                mov     esi, offset dword_401005
                mov     edi, esi
                mov     edx, dword_403C40

DecryptionLoop_1:                      ; CODE XREF: ResolveImports+51↓j
                lodsd
                call    DecryptionFunction
                stosd
                loop    DecryptionLoop_1
                call    sub_4010AA
                popa
                db      64h
                jbe     short loc_401106
                jo      short loc_401110
                xor     esi, [edx]
ResolveImports  endp
```

After the malware gets the `ntdll.dll` base address, it loops twice calling a function named `DecryptionFunction`. This function receives as input a **dword**, which here is a hash. As we're going to see, it will walk the Export Address Table of the module searching for a particular function with the name matching to the passed hash. With this first loop, the malware finds two functions: `strstr` and `LdrGetDllHandle`.

As an example, in this particular case, the `DecryptionFunction` is walking, as we explained before for `ntdll.dll`, the module `kernel32.dll`, retrieving the address of `VirtualAlloc` put inside the EAX register as return value.



DecryptionFunction

After fully disassembling the function(s) we have the following:

```
DecryptionFunction proc near

var_4           = dword ptr -4

                pusha
                mov     ebp, eax ; EAX has the HASH
                mov     ebx, edx ; EDX has the base address of the module
                mov     edi, [ebx+3Ch]
                mov     edi, [edi+ebx+78h]
                add     edi, ebx
                push    edi
                mov     ecx, [edi+18h]
                mov     edx, [edi+20h]
                add     edx, ebx

loc_4012E1:
                dec     ecx
                push    ecx
                mov     esi, [edx+ecx*4]
                add     esi, ebx
                mov     eax, esi
                xor     ecx, ecx

loc_4012EC:
                xor     ch, [eax]
                rol     ecx, 8
                xor     cl, ch
                inc     eax
                cmp     byte ptr [eax], 0
                jnz     short loc_4012EC

                cmp     ecx, ebp
                pop     ecx
                jnz     short loc_4012E1

                pop     edi
                mov     eax, [edi+24h]
                add     eax, ebx
                movzx   ecx, word ptr [eax+ecx*2]
                mov     eax, [edi+1Ch]
                add     eax, ebx
                mov     eax, [eax+ecx*4]
                add     eax, ebx
                mov     [esp+20h+var_4], eax
                popa
                retn
DecryptionFunction endp
```

The hashes of the resolved and imported functions appear as follows:

```
dword_401005    dd 416F346Fh            ; DATA XREF: ResolveImports+3D↓o
                                        ; ResolveImports:loc 401106↓o
                                        ; GetModuleFileNameA
                dd 1E360E7Fh            ; CloseHandle
dword_40100D    dd 0A7E6B43h            ; DATA XREF: AllocateMemory+A↓r
                                        ; VirtualAlloc
                dd 5581963h             ; lstrlen
dword_401015    dd 65233F5Ah            ; DATA XREF: sub 4010D1+20↓r
                                        ; sub 40118E+12E↓r
                                        ; Sleep
                dd 18732107h            ; GetModuleHandleA
dword_40101D    dd 314F7A04h            ; DATA XREF: .text:00401156↓r
                                        ; GetModuleInformationA
dword_401021    dd 4179346Fh            ; DATA XREF: ResolveImports+CA↓o
                                        ; GetModuleFileNameW
dword_401025    dd 35595F41h            ; DATA XREF: sub 4010AA↓r
                                        ; sub 4010D1↓r
                                        ; LoadLibraryA
dword_401029    dd 9320252h             ; DATA XREF: sub 4012C2+2↓r
                                        ; ExitProcess
dword_40102D    dd 6E7A0640h            ; DATA XREF: sub 40118E↓r
                                        ; EnumSystemLocalesA
dword_401031    dd 7757506h             ; DATA XREF: ResolveImports+1A↓o
                                        ; ResolveImports+E0↓r
                                        ; strstr
                dd 130D013Ah            ; LdrGetDLLHandle
dword_401039    dd 7D1C4B5Fh            ; DATA XREF: .text:00401359↓r
                                        ; RegOpenKeyExA
                dd 96B111Fh             ; RegQueryValueExA
dword_401041    dd 102F2046h            ; DATA XREF: .text:loc 401399↓o
                                        ; RegCloseKey
dword_401045    dd 45324E13h            ; DATA XREF: .text:004013A4↓o
                                        ; CharLowerA
```

After using the debugger to step into the loops of the `DecryptionFunction`, we were able to find what functions the malware uses next.

This part of the executable almost works the same way through libraries and functions. I highly suggest looking at the disassembly line by line to understand the inner working of the Windows Internal Subsystem and API calls.

Another interesting trick to be even more stealthy is the use of stack strings to build calls to `LoadLibraryA`. The secret here is that, by definition, the **CALL** instruction pushes the next address onto the stack as the return address. But this address is an ASCII null terminated string that will be an argument for the next `LoadLibraryA` call. Here you can see how it loads two libraries: `advapi32` and `user32`.

```
DecryptionLoop_1:                           ; CODE XREF: ResolveImports+51↓j
                lodsd
                call    DecryptionFunction
                stosd
                loop    DecryptionLoop_1
                call    Advapi32_Caller
ResolveImports  endp ; sp-analysis failed

; ---------------------------------------------------------------------------
                dd 'avda'                                advapi32
                dd '23ip'
                db 0

; =============== S U B R O U T I N E =======================================


Advapi32_Caller proc near                   ; CODE XREF: ResolveImports+53↑p
                call    dword_401025        ; Call LoadLibraryA
                mov     edx, eax
                push    3
                pop     ecx
                mov     esi, offset dword_401039 ; RegOpenKeyExA
                mov     edi, esi

loc_4010BC:                                 ; CODE XREF: Advapi32 Caller+19↓j
                lodsd
                call    DecryptionFunction
                stosd
                loop    loc_4010BC
                call    User32_Caller
; ---------------------------------------------------------------------------
                dd 'resu'
Advapi32_Caller endp
                                                          user32
                dw '23'
                db    0
; ---------------------------------------------------------------------------

User32_Caller:                              ; CODE XREF: Advapi32 Caller+1B↑p
                call    dword_401025        ; Call LoadLibrayA
                mov     edx, eax
                push    1
                pop     ecx
                mov     esi, offset dword_401045 ; CharLowerA
                mov     edi, esi

loc_4010E3:                                 ; CODE XREF: .text:004010EA↓j
                lodsd
                call    DecryptionFunction
```

Immediately after resolving the imports, the malware sleeps for 10 seconds and then retrieves a filename via `GetModuleFileNameA`.

```
loc_4010E3:                             ; CODE XREF: .text:004010EA↓j
                lodsd
                call    DecryptionFunction
                stosd
                loop    loc_4010E3
                push    10000
                call    dword_401015    ; Sleep for 10 seconds
                xor     esi, esi
                mov     edi, 104h
                push    edi

loc_4010FF:                             ; CODE XREF: sub 4010AA+22↑j
                push    offset dword_403C58
                push    esi

loc_401105:                             ; CODE XREF: ResolveImports+59↑j
                call    dword_401005    ; Call GetModuleFileNameA
                push    edi

loc_40110C:                             ; CODE XREF: ResolveImports+5C↑j
                push    offset dword_403D5C
                push    esi
                call    dword_401021    ; Call GetModuleFileNameW
                call    loc_401124
; ---------------------------------------------------------------------------
                dw 'as'
                dw 'pm'
                dw 'el'
                db    0
; ---------------------------------------------------------------------------

loc_401124:                             ; CODE XREF: .text:00401118↑p
                push    offset dword_403C58
                call    dword_401031    ; Call strstr
                add     esp, 8
                cmp     eax, 0
                jz      short loc_40113C
                jmp     sub_4012C2
; ---------------------------------------------------------------------------
```

Interestingly, the image above also shows how the code checks if its own name contains the string "*sample*" and if so consequently terminates itself. You can see how the call to the `strstr` function is built and how the previous push is given to check for the "*sample*" string.

It's a simple anti-analysis technique that might easily catch you out. **Protip**: do not call your sample *"sample"*. 🙂

```
00403C48 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00 | ................
00403C58 | 43 3A 5C 55 | 73 65 72 73 | 5C 6B 61 72 | 74 6F 6E 65 | C:\Users\kartone
00403C68 | 5C 44 65 73 | 6B 74 6F 70 | 5C 73 74 61 | 67 65 32 5F | \Desktop\stage2_
00403C78 | 30 30 32 46 | 30 30 30 30 | 2E 65 78 65 | 00 00 00 00 | 002F0000.exe....
00403C88 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00 | ................
00403C98 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00 | ................
00403CA8 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00 | ................
```

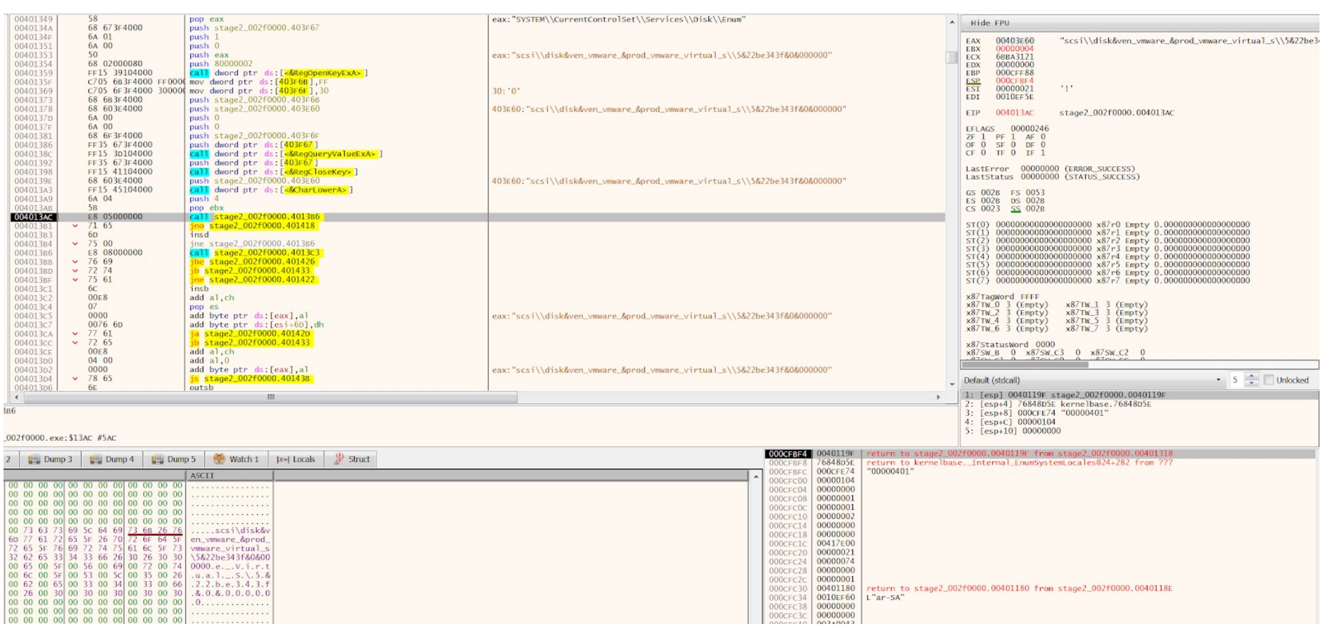
Next, the malware performs another check via `GetVolumeInformationA`, which is thoroughly documented in MSDN. Let's look inside this call to understand its purpose.

```
loc_40113C:                             ; CODE XREF: .text:00401135↑j
                mov     dword_403E00+163h, '\:C'
                push    esi
                push    esi
                push    esi
                push    esi
                push    ebp
                push    80h
                push    esi
                push    403F63h
                call    dword_40101D    ; GetVolumeInformationA
                cmp     dword ptr [ebp+0], 0CD1A40h
                jz      short loc_40116E
                cmp     dword ptr [ebp+0], 70144646h
                jnz     short loc_401173

loc_40116E:                             ; CODE XREF: .text:00401163↑j
                jmp     sub_4012C2      ; Exit process
; ---------------------------------------------------------------------------
```

From the above disassembly, we can see that it retrieves the volume serial number and checks if it’s equal to some two serials. It then opens a registry key with `RegOpenKeyExA`, pushing one of the arguments with the same **CALL** technique. It then obtains the value of the registry key, closes the handle, and converts the value to lowercase before proceeding.

```
                retn
DecryptionFunction endp

; ---------------------------------------------------------------------------

loc_401318:                             ; CODE XREF: .text:0040119A↑p
                call    loc_401349
                dd 'TSYS'
                dd 'C\ME'
                dd 'erru'
                dd 'oCtn'
                dd 'ortn'
                dd 'teSl'
                dd 'reS\'
                dd 'eciv'
                dd 'iD\s'
                dd 'E\ks'
                dd 'mun'
; ---------------------------------------------------------------------------

loc_401349:                             ; CODE XREF: .text:loc_401318↑p
                pop     eax             ; SYSTEM\CurrentControlSet\Services\Disk\Enum
                push    403F67h
                push    1
                push    0
                push    eax
                push    HKEY_LOCAL_MACHINE
                call    dword_401039    ; Call RegOpenKeyExA
                mov     dword_403E00+16Bh, 0FFh
                mov     dword_403E00+16Fh, 30h
                push    403F6Bh
                push    403E60h
                push    0
                push    0
                push    403F6Fh
                push    dword_403E00+167h
                call    dword_40103D    ; Call RegQueryValueExA
                push    dword_403E00+167h
                call    dword_401041    ; Call RegCloseKey
                push    403E60h
                call    dword_401045    ; Call CharLowerA
                push    4
                pop     ebx
                call    loc_4013B6
```

It looks clear when you see it in the debugger.



With this string saved somewhere in memory, the code goes on to perform some other checks trying to find any sign of running inside a virtual environment.

```
                push    403E60h
                call    dword_401045     ; Call CharLowerA
                push    4
                pop     ebx

loc_4013AC:
                call    loc_4013B6
; ---------------------------------------------------------------------------
                dd 'umeq'
                db    0
; ---------------------------------------------------------------------------

loc_4013B6:                              ; CODE XREF: .text:loc_4013AC↑p
                call    loc_4013C3
                dd 'triv'
                dd 'lau'
; ---------------------------------------------------------------------------

loc_4013C3:                              ; CODE XREF: .text:loc_4013B6↑p
                call    loc_4013CF
                dd 'awmv'
word_4013CC     dw 'er'
                db    0
; ---------------------------------------------------------------------------

loc_4013CF:                              ; CODE XREF: .text:loc_4013C3↑p
                call    loc_4013D8
                dd 'nex'
; ---------------------------------------------------------------------------

loc_4013D8:                              ; CODE XREF: .text:loc_4013CF↑p
                                         ; .text:004013F3↓j
                push    403E60h          ; Pointer to the retrieved registry value
                call    dword_401031     ; Call strstr
                add     esp, 8
                cmp     eax, 0
                jz      short loc_4013ED
                jmp     short loc_4013F7
; ---------------------------------------------------------------------------

loc_4013ED:                              ; CODE XREF: .text:004013E9↑j
                dec     ebx
                cmp     ebx, 0
                jz      short loc_4013F5
                jmp     short loc_4013D8
; ---------------------------------------------------------------------------

loc_4013F5:                              ; CODE XREF: .text:004013F1↑j
                jmp     short loc_4013FB
; ---------------------------------------------------------------------------

loc_4013F7:                              ; CODE XREF: .text:004013EB↑j
                push    1
                pop     eax
                retn
; ---------------------------------------------------------------------------

loc_4013FB:                              ; CODE XREF: .text:loc_4013F5↑j
                xor     eax, eax
                retn
```

As part of the anti-virtual machine checks, it initializes a 4 cycle loop; during this loop it performs a call to the `strstr` function to search inside the retrieved registry value for any sign of the strings: *"qemu"*, *"virtual"*, *"vmware"*, *"xen"*. If you notice in the previous debugger screenshot, I'm running the sample inside a VMWare machine, so to continue I will need to patch the return value of `strstr` function calls to return zero.

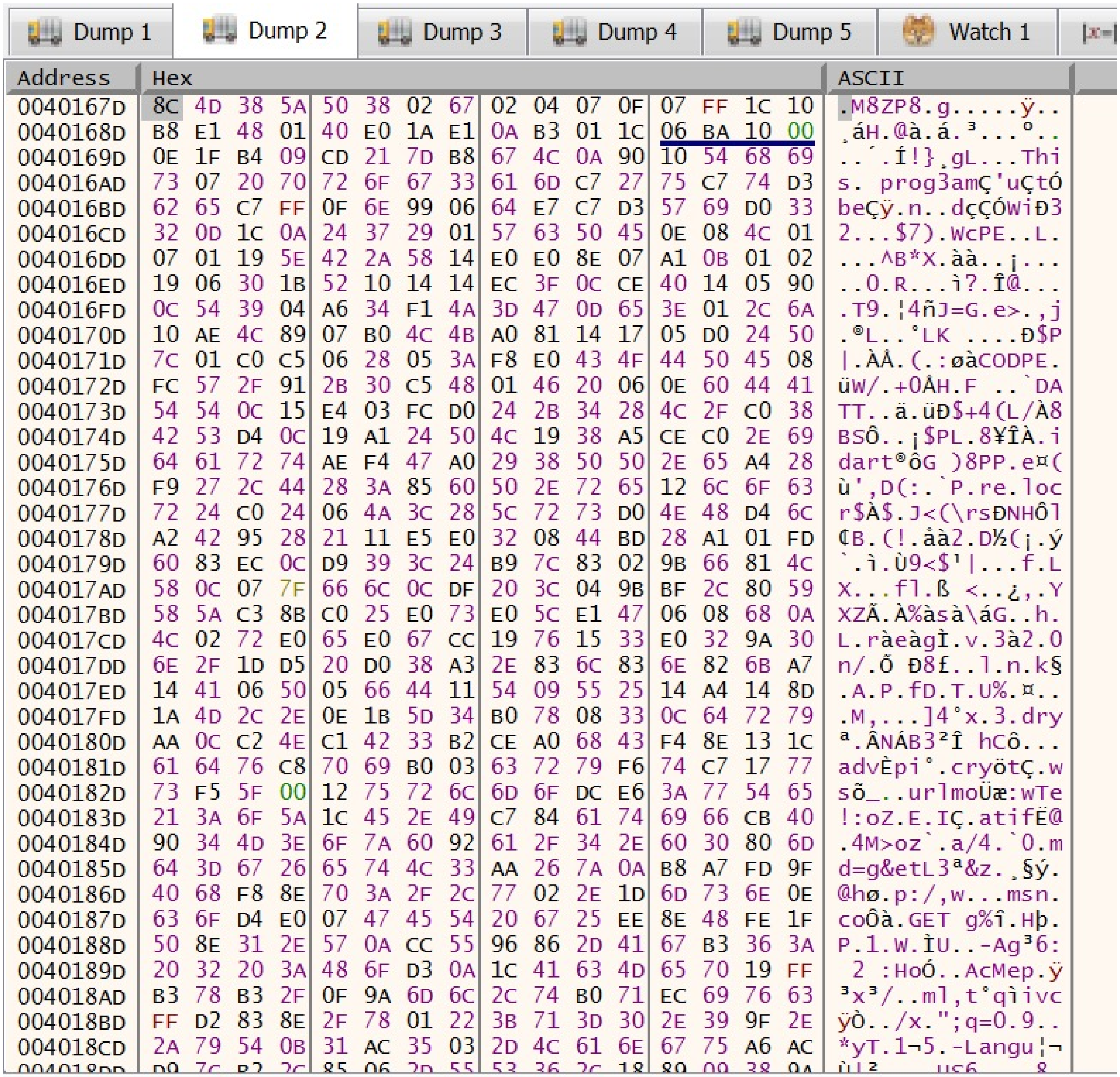
Other checks are waiting:

```
loc_4011A9:                                  ; CODE XREF: .text:004011A2↑j
                call    loc_4011B6
; ---------------------------------------------------------------------------
                dd 'eibs'
                dd 'lld'
; ---------------------------------------------------------------------------

loc_4011B6:                                  ; CODE XREF: .text:loc 4011A9↑p
                call    loc_4011C3
                dd 'hgbd'
                dd 'ple'
; ---------------------------------------------------------------------------

loc_4011C3:                                  ; CODE XREF: .text:loc 4011B6↑p
                push    2
                pop     ebx

loc_4011C6:                                  ; CODE XREF: .text:004011DC↓j
                call    dword_401019         ; Call GetModuleHandleA
                cmp     eax, 0
                jz      short loc_4011D6
                jmp     ExitProcess
; ---------------------------------------------------------------------------

loc_4011D6:                                  ; CODE XREF: .text:004011CF↑j
                dec     ebx
                cmp     ebx, 0
                jz      short loc_4011DE
                jmp     short loc_4011C6
; ---------------------------------------------------------------------------
```

As you can see, the malware tries to understand if it's being debugged or executed inside a sandbox by trying to get a handle to modules sbiedll and dbghelp. If it's able to detect one of these two libraries, it terminates the process and exits.

Finally, The Payload!

Having passed all sorts of anti-analysis and anti-debugging checks, we finally reach the payload! Now, the malware begins to reveal its secrets in memory.



We can clearly see it's a PE file, but it's scrambled somehow. This code will be decoded and managed in memory with a complete routine.

```
; ---------------------------------------------------------------------------

loc_4011DE:                             ; CODE XREF: .text:004011DA↑j
                mov     esi, offset unk_40167D
                lodsb
                mov     dl, al
                mov     edi, esi
                mov     ecx, 25B5h
                push    ecx
                push    esi

loc_4011EF:                             ; CODE XREF: .text:004011F3↓j
                lodsb                   ; Dexoring buffer at 0x0040167d with size 0x25b5 and xor key 0x8c
                xor     al, dl
                stosb
                loop    loc_4011EF
                mov     eax, 10000h
                call    AllocateMemory
                mov     ecx, eax
                pop     eax
                push    eax
                push    ecx
                call    loc_40144D
                xor     al, al
                pop     ebp
                pop     edi
                dec     edi
                pop     ecx
                rep stosb
                mov     eax, [ebp+3Ch]
                add     eax, ebp
                push    eax
                mov     eax, [eax+50h]
                call    AllocateMemory
                mov     ebx, eax
                pop     eax
                push    eax
                mov     ecx, [eax+54h]
                mov     esi, ebp
                mov     edi, ebx
                rep movsb
                mov     [ebx+134h], ebx
                movzx   ecx, byte ptr [eax+6]
                pop     esi
                push    esi
                add     esi, 0F8h

loc_40123E:                             ; CODE XREF: .text:00401258↓j
                push    ecx
                mov     ecx, [esi+10h]
                test    ecx, ecx
                jz      short loc_401254
                push    esi
                mov     edi, [esi+0Ch]
                add     edi, ebx
                mov     esi, [esi+14h]
                add     esi, ebp
                rep movsb
                pop     esi
```

Digging into this code will require more time and effort than the analyst will normally want to expend. Instead, we can detonate the malware in our isolated environment and observe its execution. As we will see in the next post, this will reveal that a new instance of `svchost.exe` is loaded into memory, which suggests some sort of process injection. If you enjoyed this deep dive and would like to know when the next **Going Deep** post is available, just subscribe to the SentinelOne blog newsletter!

IOCs

Sample Hash `07e81dfc0a01356fd96f5b75efe3d1b1bc86ade4`

MITRE ATT&CK

Smoke Loader {S0226}
Virtualization/Sandbox Evasion {T1497}