

# Smoke Loader Malware – An Easy Guide 101

By SentinelOne

Published: 2019-11-21 · Archived: 2026-04-05 22:33:11 UTC

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.

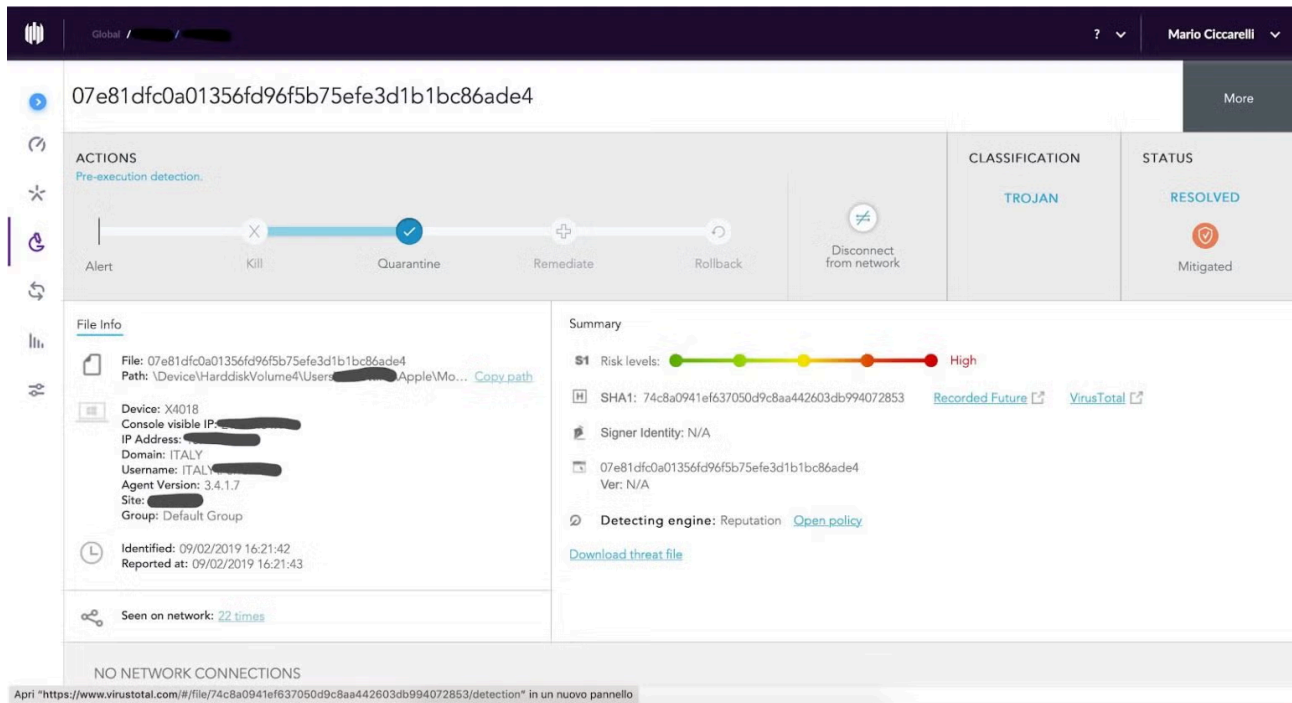


This alert was raised against a suspicious file, classified as a trojan, that was killed and quarantined. What raised my curiosity was the number of detections over only a few hours, always from the same workstation, and only from the same user.

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 :

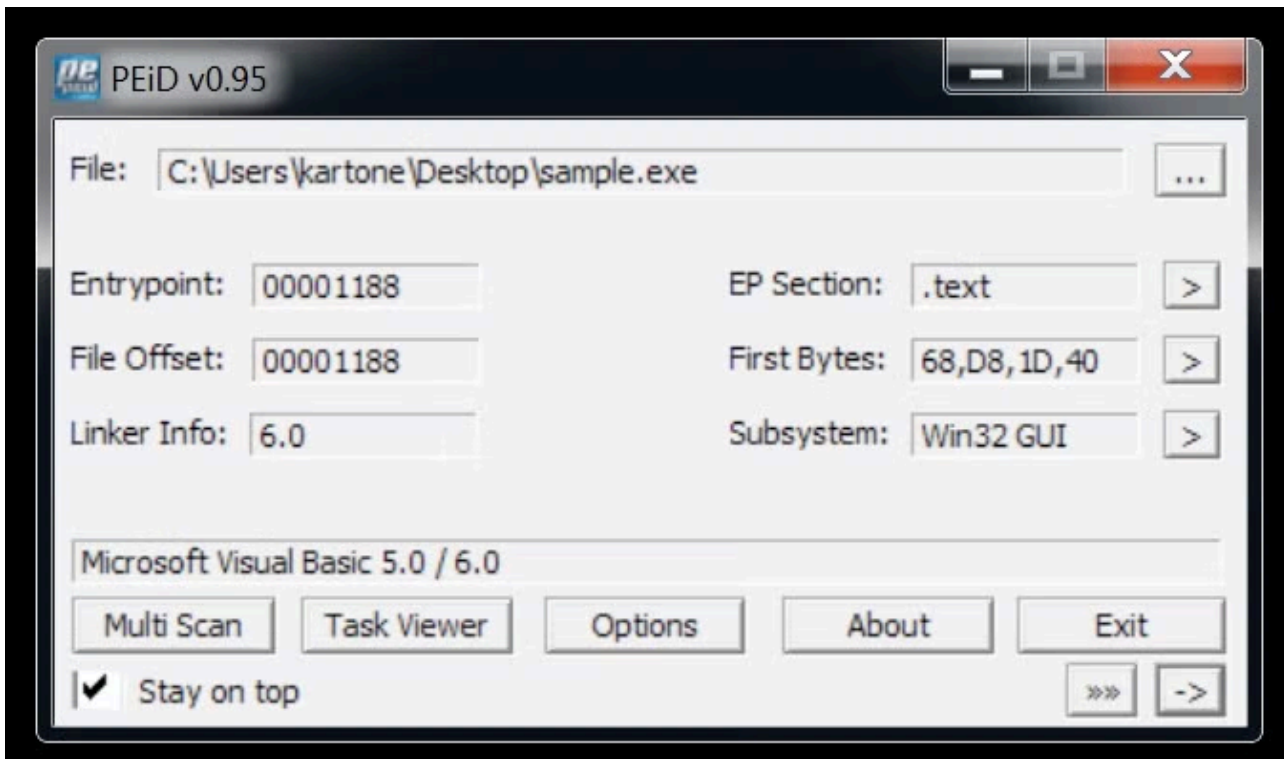
- The full path where the detection was made.
- The associated risk level is High: this implies that it's a positive detection.
- File unique hash that can be tested for any public Indicators of Compromise (IoC).



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

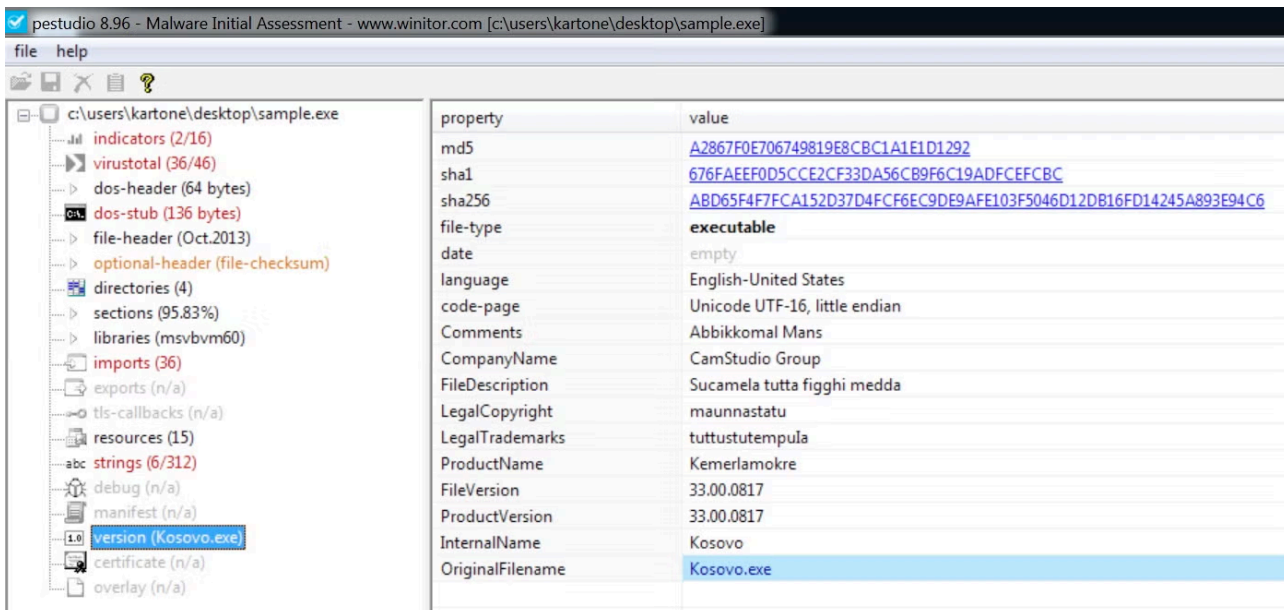
With the downloaded file in my pocket, I quickly fired up an isolated analysis machine equipped with the [Flare](#) tools and started to investigate. At first glance, the sample appears to be a Visual Basic program leveraging Win32 APIs.



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.



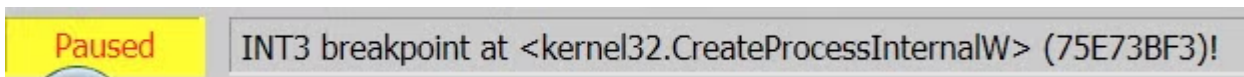
With a bit of experience, we can safely assume that the file is packed with an external layer of Visual Basic that tries to stop, or at least slow down, static analysis. But what about its runtime behavior?

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.

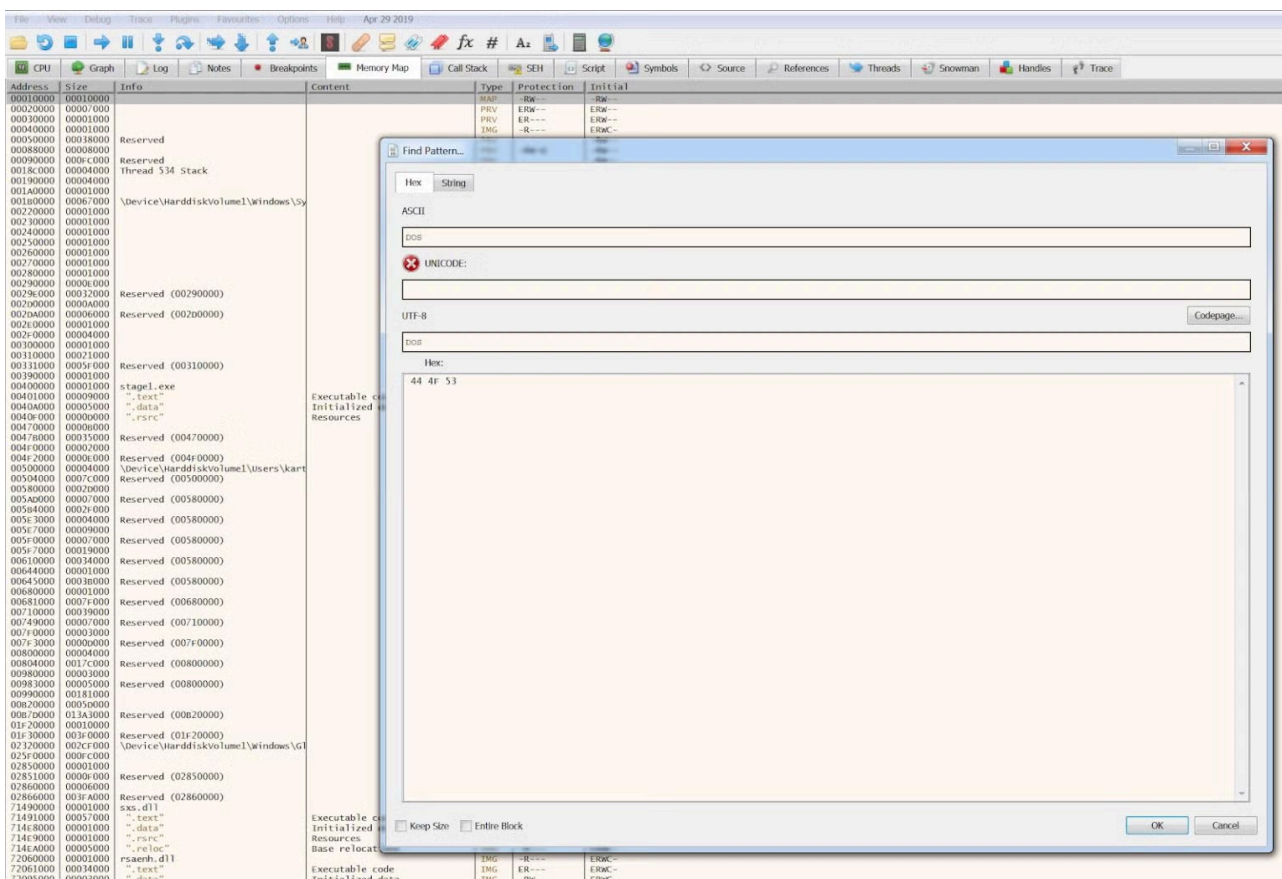
## Defeating Visual Basic Packer

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.

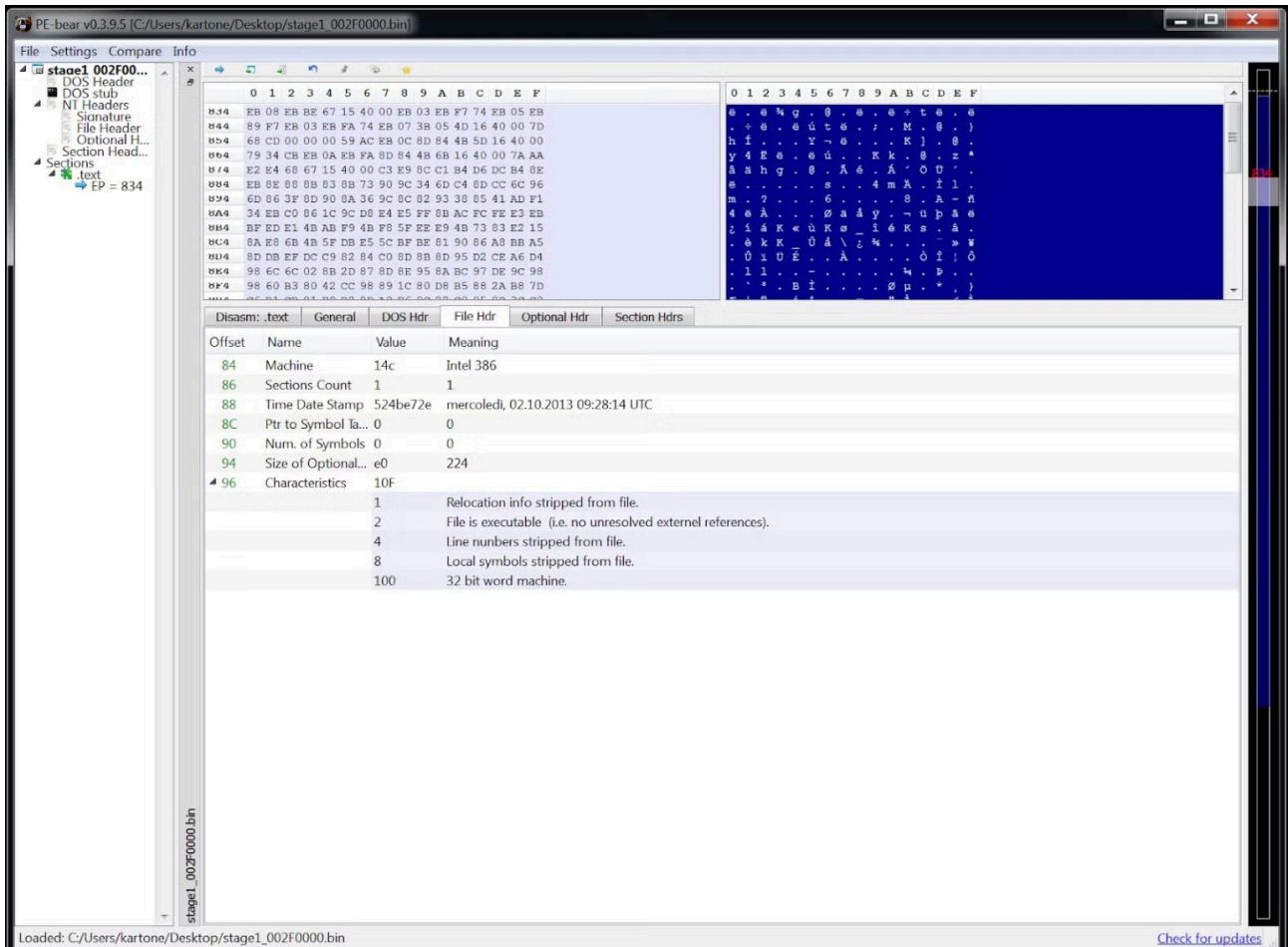
Things become particularly interesting at `0x002F0094`, which we can follow in the memory dump.

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	.....°...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--

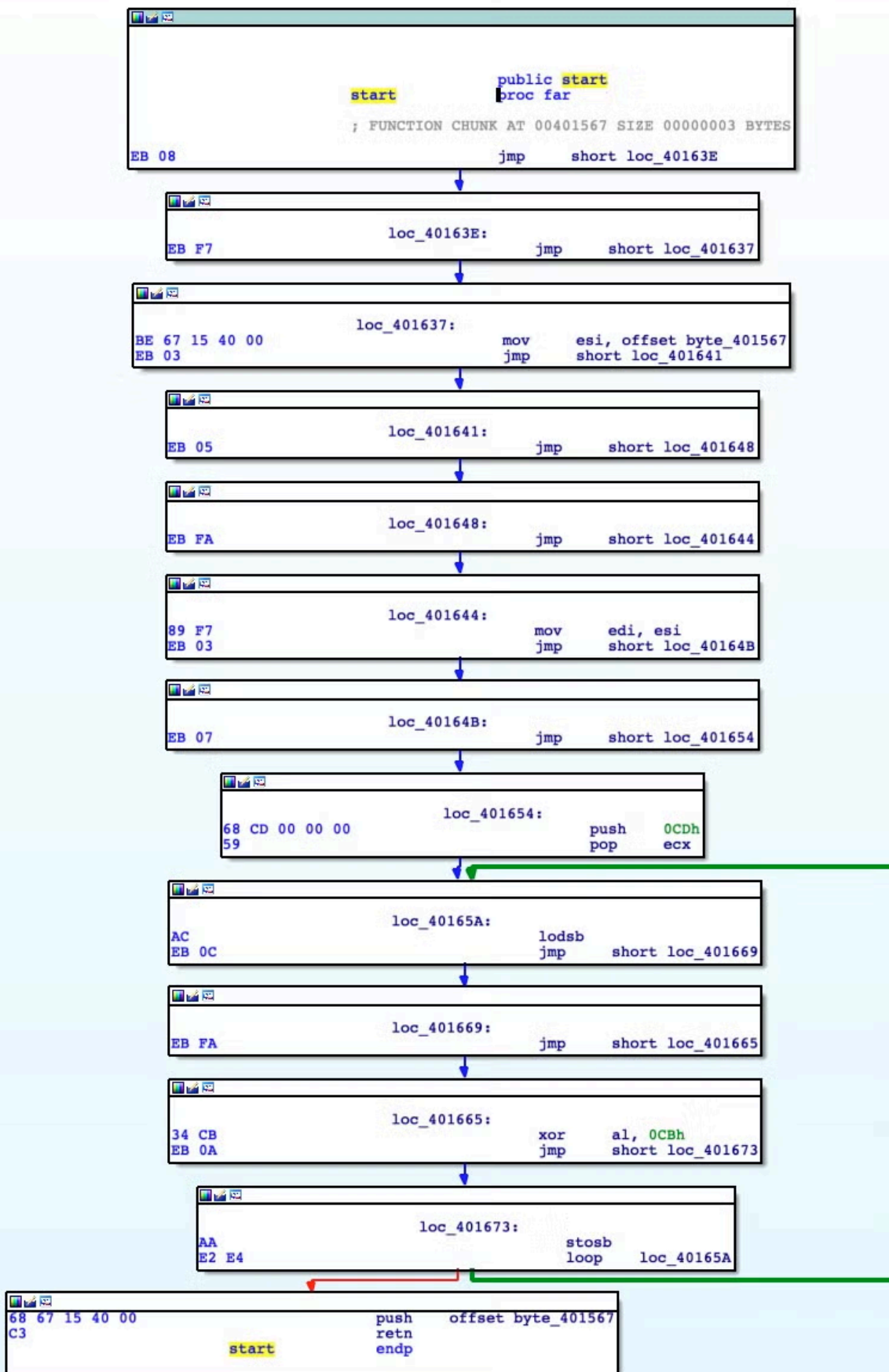
We can simply dump out this memory region to file, clean the junk before the MZ header and analyze its headers.



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.

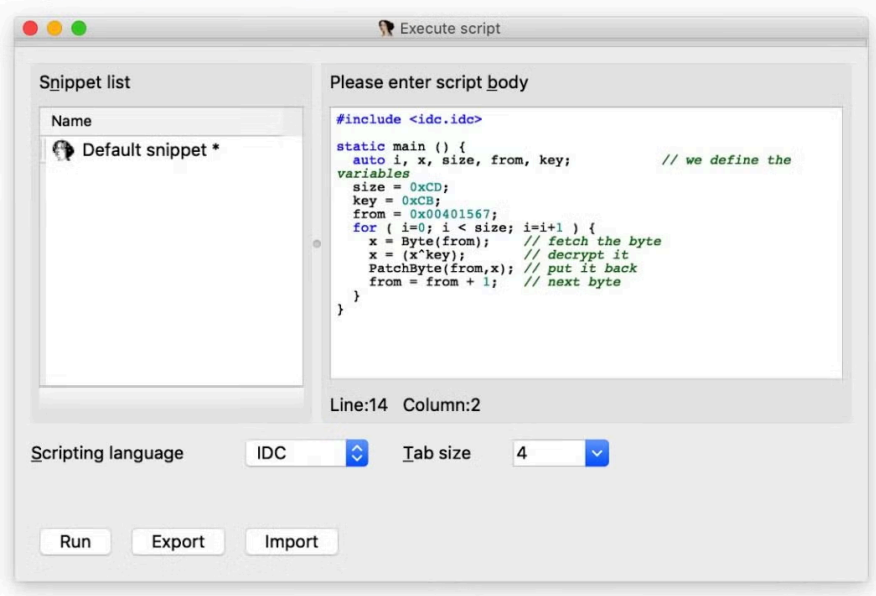


Here we recognize that it's a **XOR** loop that will decode, from address `0x00401567`, a blob of code with the size of `0xCD` bytes with a **XOR** key equal to `0xCB`. At the end of the loop, the same starting address `0x00401567` is pushed onto the stack and with the **RET** instruction the program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

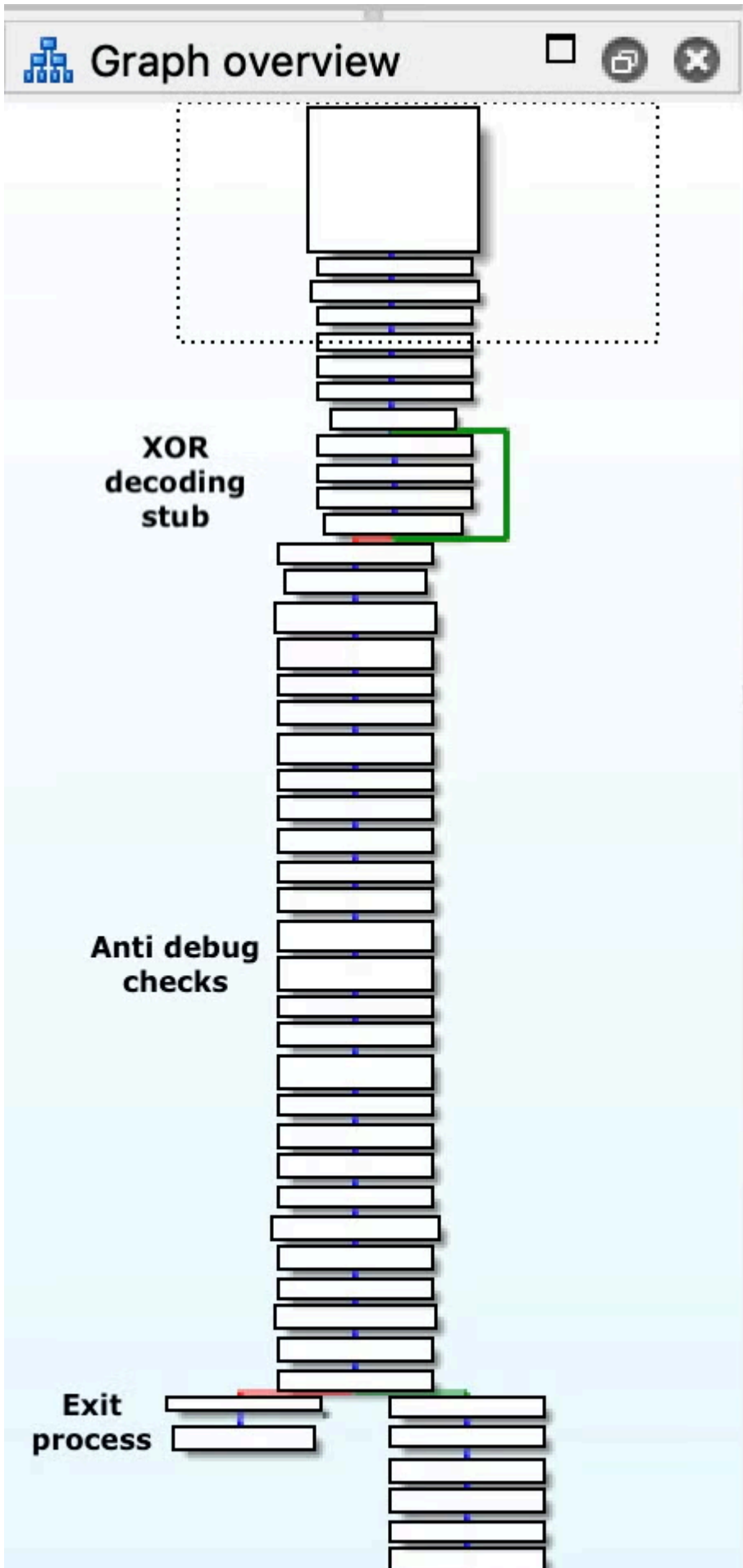
## Decoding the Buffer

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

```
.text:00401010 dd 35429DCAh, 6F380DE2h, 0CB452095h, 41CEBE19h, 19DB8DDh
.text:00401010 dd 8A02FA08h, 34342523h, 2302DA34h, 3434342Ch, 80839B9h
.text:00401010 dd 4020C220h, 8BF7F3F6h, 20C820CBh, 0CC20BF3Dh, 0DF33CF0h
.text:00401010 dd 0A1B6CB8Bh, 0CC2092BBh, 84F24FC4h, 20C820A3h, 0CA20B132h
.text:00401010 dd 20C0BEB1h, 0DECF0CCh, 0B6CB8BDEh, 0C320089Ch, 0DBC7520h
.text:00401010 dd 0C820CB8Bh, 20BF3C20h, 3C4220CEh, 3120C820h, 0F0CC20BFh
.text:00401010 dd 8BDEFCCCh, 23A3B6CBh, 92CBCBCFh, 46C72067h, 0DE8C804Fh
.text:00401010 dd 0FFB2CB8Bh, 20C12000h, 804F4631h, 0CB8BDE9Eh, 2F2961B1h
.text:00401010 dd 8BDBC8A3h
.text:00401564 ; -----
.text:00401564 retf
.text:00401565 ; -----
.text:00401565 or [edx], ah
.text:00401565 ; -----
.text:00401565 ; START OF FUNCTION CHUNK FOR start
.text:00401567 byte_401567 db 23h ; CODE XREF: start+474j
.text:00401567 ; DATA XREF: start:loc_4016374o ...
.text:00401568 db 0CFh ; I
.text:00401569 db 0CBh
.text:00401569 ; END OF FUNCTION CHUNK FOR start
.text:0040156A db 0CBh
.text:0040156B db 0CBh
.text:0040156C db 71h
.text:0040156D db 17h
.text:0040156E db 35h ; 5
.text:0040156F db 0CFh ; E
.text:00401570 db 0A3h ; E
.text:00401571 db 0BCh ; 4
.text:00401572 db 0DEh ; D
.text:00401573 db 8Bh ; 8
.text:00401574 db 0CBh ; E
.text:00401575 db 20h
.text:00401576 db 0CFh ; I
.text:00401577 db 71h ; 7
.text:00401578 db 17h ; 7
.text:00401579 db 25h ; 5
.text:0040157A db 0CBh ; E
.text:0040157B db 92h ; '
.text:0040157C db 20h
.text:0040157D db 0CEh ; I
.text:0040157E db 20h
.text:0040157F db 40h ; @
.text:00401580 db 0C2h ; A
.text:00401581 db 20h
.text:00401582 db 0CBh ; E
.text:00401583 db 20h
.text:00401584 db 31h ; 1
.text:00401585 db 0BFh ; z
.text:00401586 db 93h ; "
.text:00401587 db 20h
.text:00401588 db 0CEh ; I
.text:00401589 db 20h
.text:0040158A db 40h ; @
.text:0040158B db 0CBh ; E
.text:0040158C db 20h
.text:0040158D db 0CBh ; E
.text:0040158E db 20h
.text:0040158F db 31h ; 1
.text:00401590 db 0BFh ; z
.text:00401591 db 20h
.text:00401592 db 0CEh ; I
.text:00401593 db 0BEh ; %
```



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.





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
```

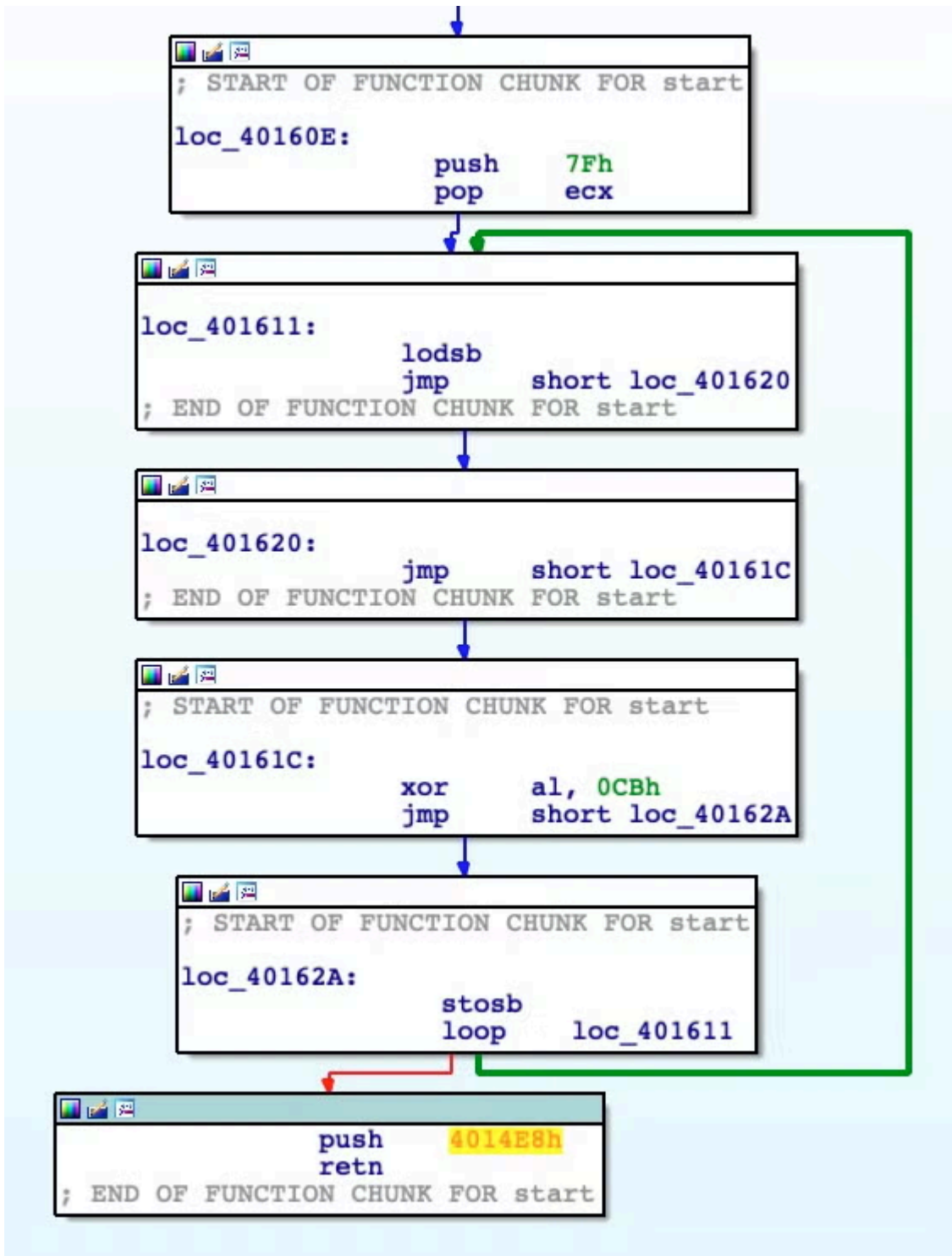
```
; START OF FUNCTION CHUNK FOR start  
loc_4015B6:  
        pop     eax  
        mul    ecx  
        jmp     short loc_4015C1  
; END OF FUNCTION CHUNK FOR start
```

```
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`.

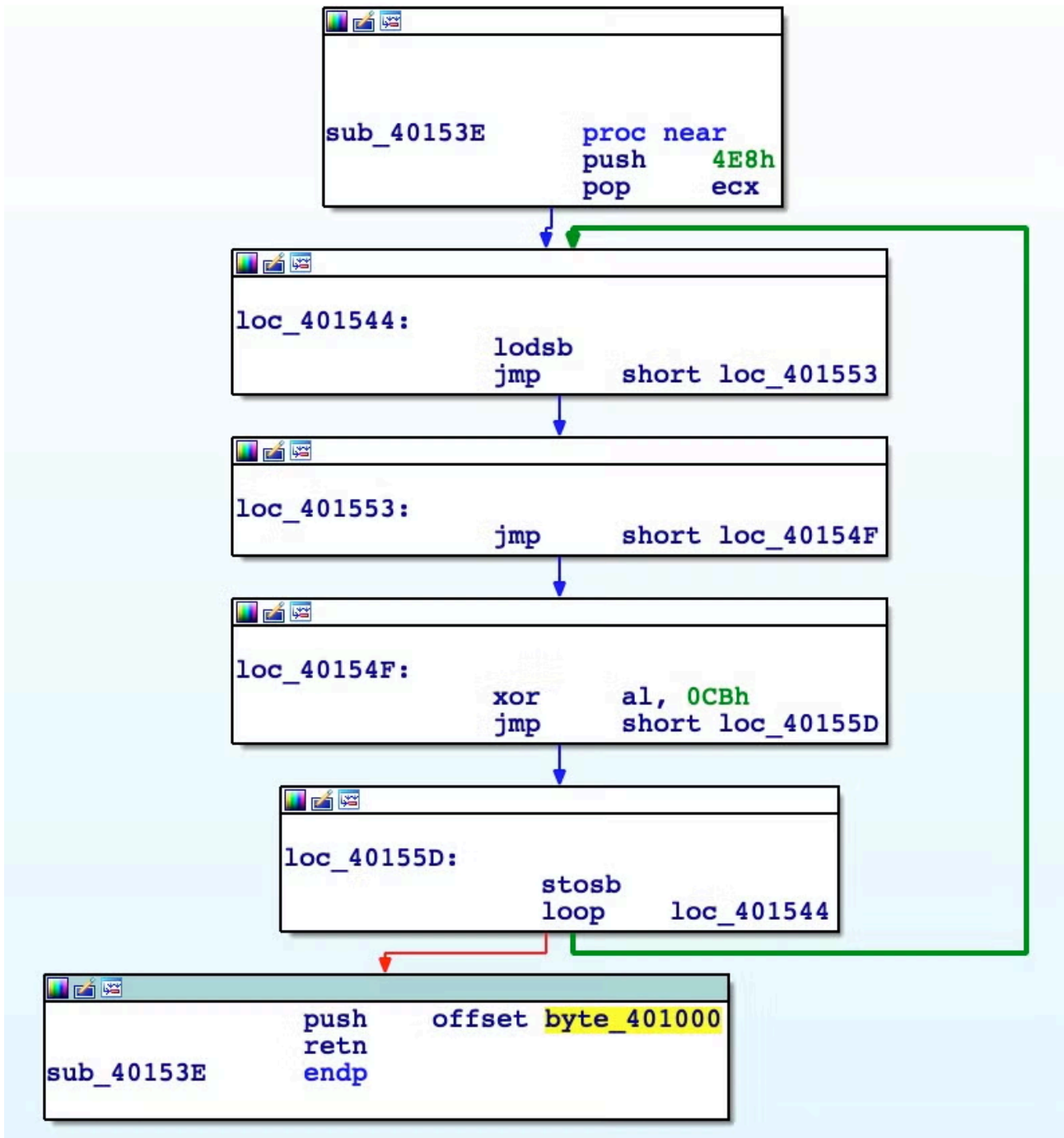


As before, we can proceed in the static analysis, manually decoding this buffer with the same script.

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

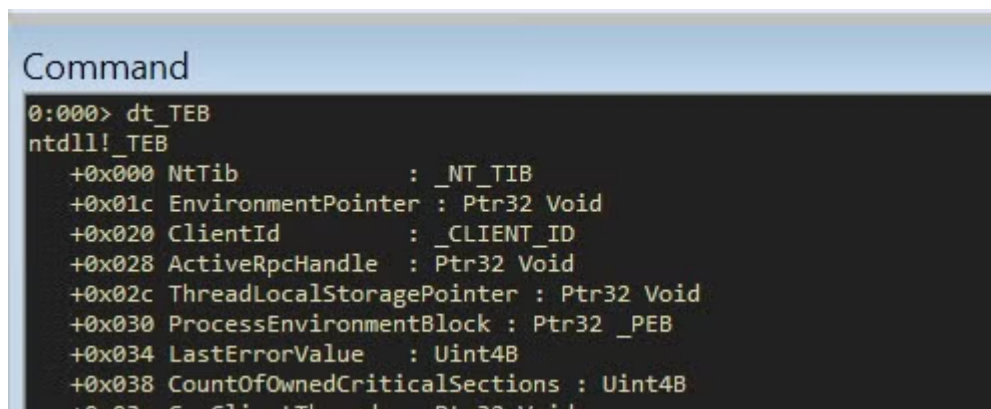
At address `0x00401000` there's a simple call to another address `0x00401049`, where it starts to become interesting as the malware appears to dynamically resolve its imports. As we noted before, the binary header analysis showed no imports at all. With this code, from the PEB location found earlier,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



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 IFE0Key : Ptr32 Void
+0x028 CrossProcessFlags : Uint4B
```

With the third instruction, we are following the offset `0x00c`, 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:

The screenshot shows the Immunity Debugger interface. The CPU window displays assembly code for the `stage2_00270000.exe` process. The code includes instructions like `mov edi, edx` and `push ecx`. The PE location window shows the `NTDDLL` module loaded at `00401000`. The hex dump window shows the module's header, including the `NTDDLL` signature and other fields. A red arrow points from the hex dump to the assembly code.

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.

Hide FPU		
EAX	75B61856	<kernel32.VirtualAlloc>
EBX	7EFDE000	
ECX	00000009	
EDX	75B50000	kernel32.75B50000
EBP	000CFF88	
ESP	000CFF88	
ESI	00401011	stage2_002f0000.00401011
EDI	00401011	stage2_002f0000.00401011
EIP	0040109A	stage2_002f0000.0040109A

## 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
; CloseHandle
dword_40100D dd 1E360E7Fh ; DATA XREF: AllocateMemory+A↓r
dd 0A7E6B43h ; VirtualAlloc
; lstrlen
dword_401015 dd 5581963h ; DATA XREF: sub 4010D1+20↓r
dd 65233F5Ah ; sub 40118E+12E↓r
; Sleep
; GetModuleHandleA
dword_40101D dd 18732107h ; DATA XREF: .text:00401156↓r
dd 314F7A04h ; 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
; LdrGetDLLHandle
dword_401039 dd 130D013Ah ; DATA XREF: .text:00401359↓r
dd 7D1C4B5Fh ; RegOpenKeyExA
; RegQueryValueExA
dword_401041 dd 96B111Fh ; DATA XREF: .text:loc 401399↓o
dd 102F2046h ; 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'
    dd '23ip'
    db 0
    advapi32

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

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

loc_4010BC:
    lodsd
    call   DecryptionFunction
    stosd
    loop  loc_4010BC
    call  User32_Caller

; -----
Advapi32_Caller endp
    dd 'resu'
    dw '23'
    db 0
    user32

; -----

User32_Caller:
    call   dword_401025 ; CODE XREF: Advapi32_Caller+1B↑p ; Call LoadLibrayA
    mov   edx, eax
    push  1
    pop   ecx
    mov   esi, offset dword_401045 ; CharLowerA
    mov   edi, esi

loc_4010E3:
    lodsd
    call   DecryptionFunction

```

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



```

loc_40113C:
    mov     dword_403E00+163h, '\:C' ; CODE XREF: .text:00401135↑j
    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:
    jmp    sub_4012C2 ; CODE XREF: .text:00401163↑j
                ; 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:
    call   loc_401349 ; CODE XREF: .text:0040119A↑p
    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:
    pop    eax ; CODE XREF: .text:loc_401318↑p
                ; 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.

The screenshot displays a debugger interface with three main panes: Assembly, Registers, and Stack. The Assembly pane shows instructions such as `push eax`, `push stage2_002f0000_403f67`, and `mov dword ptr ds:[546c2b34f6060000], 30:0`. The Registers pane shows values for EAX (00403E60), ECX (688A3121), EDI (00000000), and EIP (0040118C). The Stack pane shows memory addresses and values, including `stage2_002f0000_401180`. The bottom pane shows a list of memory addresses and values, including `00000000`, `7F 69`, `72 74`, `75 61`, `00E8`, `007E 60`, `77 61`, `72 65`, `00E8`, `84 00`, `0000`, `78 65`, and `00`.

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     'umeg'
db     0
; -----

loc_4013B6:
call    loc_4013C3 ; CODE XREF: .text:loc 4013AC↑p
dd     'triv'
dd     'lau'
; -----

loc_4013C3:
call    loc_4013CF ; CODE XREF: .text:loc 4013B6↑p
word_4013CC
dw     'er'
db     0
; -----

loc_4013CF:
call    loc_4013D8 ; CODE XREF: .text:loc 4013C3↑p
dd     'nex'
; -----

loc_4013D8:
; CODE XREF: .text:loc 4013CF↑p
; .text:004013F3↓j
; Pointer to the retrieved registry value
push    403E60h
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:
jmp     short loc_4013FB ; CODE XREF: .text:004013F1↑j
; -----

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

loc_4013FB:
xor     eax, eax ; CODE XREF: .text:loc 4013F5↑j
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:      call     loc_4011B6      ; CODE XREF: .text:004011A2↑j
; -----
                dd     'eibs'
                dd     'lld'
; -----
loc_4011B6:      call     loc_4011C3      ; CODE XREF: .text:loc_4011A9↑p
                dd     'hgbd'
                dd     'ple'
; -----
loc_4011C3:      push    2                ; CODE XREF: .text:loc_4011B6↑p
                pop     ebx
loc_4011C6:      call     dword_401019    ; CODE XREF: .text:004011DC↓j
                cmp     eax, 0      ; Call GetModuleHandleA
                jz      short loc_4011D6
                jmp     ExitProcess
; -----
loc_4011D6:      dec     ebx              ; CODE XREF: .text:004011CF↑j
                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.

Dump 1										Dump 2										Dump 3										Dump 4										Dump 5										Watch 1									
Address	Hex															ASCII																																											
0040167D	8C	4D	38	5A	50	38	02	67	02	04	07	0F	07	FF	1C	10	.M8ZP8.g.....ÿ..																																										
0040168D	B8	E1	48	01	40	E0	1A	E1	0A	B3	01	1C	06	BA	10	00	.áH.@à.á.³...°..																																										
0040169D	0E	1F	B4	09	CD	21	7D	B8	67	4C	0A	90	10	54	68	69	..'.í!}gL...Thi																																										
004016AD	73	07	20	70	72	6F	67	33	61	6D	C7	27	75	C7	74	D3	s. prog3amÇ'uçtÓ																																										
004016BD	62	65	C7	FF	0F	6E	99	06	64	E7	C7	D3	57	69	D0	33	beÇÿ.n..dçÇÓWiÐ3																																										
004016CD	32	0D	1C	0A	24	37	29	01	57	63	50	45	0E	08	4C	01	2...\$7).wcPE..L.																																										
004016DD	07	01	19	5E	42	2A	58	14	E0	E0	8E	07	A1	0B	01	02	...^B*X.àà..j...																																										
004016ED	19	06	30	1B	52	10	14	14	EC	3F	0C	CE	40	14	05	90	..0.R...ì?.í@...																																										
004016FD	0C	54	39	04	A6	34	F1	4A	3D	47	0D	65	3E	01	2C	6A	.T9.¡4ñj=G.e>.,j																																										
0040170D	10	AE	4C	89	07	B0	4C	4B	A0	81	14	17	05	D0	24	50	.°L..°LK....Ð\$P																																										
0040171D	7C	01	C0	C5	06	28	05	3A	F8	E0	43	4F	44	50	45	08	.À.À.(.:øàCODPE.																																										
0040172D	FC	57	2F	91	2B	30	C5	48	01	46	20	06	0E	60	44	41	üw/.+0ÄH.F..`DA																																										
0040173D	54	54	0C	15	E4	03	FC	D0	24	2B	34	28	4C	2F	C0	38	TT..ä.üÐ\$+4(L/A8																																										
0040174D	42	53	D4	0C	19	A1	24	50	4C	19	38	A5	CE	C0	2E	69	BSÔ..j\$PL.8¥ÏA.i																																										
0040175D	64	61	72	74	AE	F4	47	A0	29	38	50	50	2E	65	A4	28	dart®ôG )8PP.eπ(																																										
0040176D	F9	27	2C	44	28	3A	85	60	50	2E	72	65	12	6C	6F	63	ù',D(:.`P.re.loc																																										
0040177D	72	24	C0	24	06	4A	3C	28	5C	72	73	D0	4E	48	D4	6C	r\$Ä\$.J<(\rsÐNHÔl																																										
0040178D	A2	42	95	28	21	11	E5	E0	32	08	44	BD	28	A1	01	FD	CB.(!.ää2.D%(j.ý																																										
0040179D	60	83	EC	0C	D9	39	3C	24	B9	7C	83	02	9B	66	81	4C	`..î.Ü9<\$` ...f.L																																										
004017AD	58	0C	07	7F	66	6C	0C	DF	20	3C	04	9B	BF	2C	80	59	x...f1.ß<..z..Y																																										
004017BD	58	5A	C3	8B	C0	25	E0	73	E0	5C	E1	47	06	08	68	0A	XZÄ.Ä%asà\äg..h.																																										
004017CD	4C	02	72	E0	65	E0	67	CC	19	76	15	33	E0	32	9A	30	L.ràèàgÏ.v.3à2.0																																										
004017DD	6E	2F	1D	D5	20	D0	38	A3	2E	83	6C	83	6E	82	6B	A7	n/.Õ Ð8f..l.n.k§																																										
004017ED	14	41	06	50	05	66	44	11	54	09	55	25	14	A4	14	8D	.A.P.fD.T.U%.π..																																										
004017FD	1A	4D	2C	2E	0E	1B	5D	34	B0	78	08	33	0C	64	72	79	.M,...]4°x.3.dry																																										
0040180D	AA	0C	C2	4E	C1	42	33	B2	CE	A0	68	43	F4	8E	13	1C	ª.ÄNÁB3²Ï hcô...																																										
0040181D	61	64	76	C8	70	69	B0	03	63	72	79	F6	74	C7	17	77	advËpi°.cryötÇ.w																																										
0040182D	73	F5	5F	00	12	75	72	6C	6D	6F	DC	E6	3A	77	54	65	sõ...ur1moÜæ:wTe																																										
0040183D	21	3A	6F	5A	1C	45	2E	49	C7	84	61	74	69	66	CB	40	!:oz.E.IÇ.atifË@																																										
0040184D	90	34	4D	3E	6F	7A	60	92	61	2F	34	2E	60	30	80	6D	.4M>oz`.a/4.`0.m																																										
0040185D	64	3D	67	26	65	74	4C	33	AA	26	7A	0A	B8	A7	FD	9F	d=g&etL3ª&z..şý.																																										
0040186D	40	68	F8	8E	70	3A	2F	2C	77	02	2E	1D	6D	73	6E	0E	@hø.p:/,w...msn.																																										
0040187D	63	6F	D4	E0	07	47	45	54	20	67	25	EE	8E	48	FE	1F	coÔà.GET g%î.Hþ.																																										
0040188D	50	8E	31	2E	57	0A	CC	55	96	86	2D	41	67	B3	36	3A	P.1.w.ÏU..-Ag³6:																																										
0040189D	20	32	20	3A	48	6F	D3	0A	1C	41	63	4D	65	70	19	FF	2 :HoÓ..AcMep.ÿ																																										
004018AD	B3	78	B3	2F	0F	9A	6D	6C	2C	74	B0	71	EC	69	76	63	³x³/..m1,t°qìivc																																										
004018BD	FF	D2	83	8E	2F	78	01	22	3B	71	3D	30	2E	39	9F	2E	ÿò../x.";q=0.9..																																										
004018CD	2A	79	54	0B	31	AC	35	03	2D	4C	61	6E	67	75	A6	AC	*yT.1-5.-Langu ~																																										
004018DD	D0	7C	E2	2C	85	06	2D	55	53	36	2C	18	80	00	38	0A	ù1z`us6`8																																										

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 complex routine.

```

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

loc_4011EF:      lodsb
                xor     al, dl ; CODE XREF: .text:004011F3↑j
                stosb ; Dexoring buffer at 0x0040167d with size 0x25b5 and xor key 0x8c
                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:      push   ecx ; CODE XREF: .text:00401258↑j
                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](#)}

Source: <https://www.sentinelone.com/blog/going-deep-a-guide-to-reversing-smoke-loader-malware/>