Data Talks: Deeper Down the Rabbit Hole: Second-Stage Attack and a Fileless Finale

By Dr Josh Stroschein

Published: 2018-11-05 · Archived: 2026-04-06 01:00:17 UTC

In our last blog, “Following a Trail of Confusion: PowerShell in Malicious Office Documents”, we systematically unraveled multiple layers of obfuscation initiated by a weaponized first-stage Microsoft Word document to reveal a surreptitious download script and a malicious second-stage binary file dropped onto the victim PC. For those who wish to follow the analysis through to its conclusion, the sample MD5 is 6c8e800f14f927de051a3788083635e5 and a VirusTotal report is here.

Picking Up Where Word Drops Off

Suppose the weaponized Word document was successful, bypassing all existing layered defenses, and now the next stage begins. This is the native code program that is now running in memory, and with it come additional capabilities to compromise the host computer. As with our previous analysis, we have to figure out what type of code obfuscation we’re dealing with it. With native code programs—portable executable (PE) files in the case of Microsoft Windows—the first layer is usually packing. Packing is a well-known technique that essentially takes the malicious program and wraps it inside another program. You can think of it like a zip or another archive, where if we analyze the zip file, we won’t get any information about the content it contains.

Resources 1
548E3734.doc

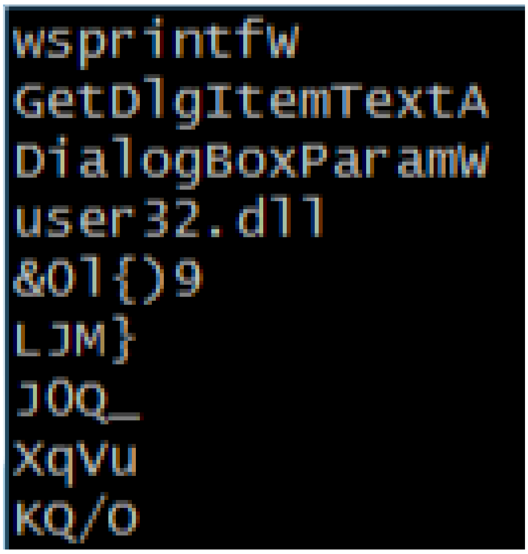
Blacklistable Files 1
70849.exe

DNS Events 2
finance-advisors-ca.bid
mobile.pipe.aria.microsoft.com

Bromium Secure Platform shows the original malicious document, the request to retrieve this sample, and the process it invoked.

Signs of Packing

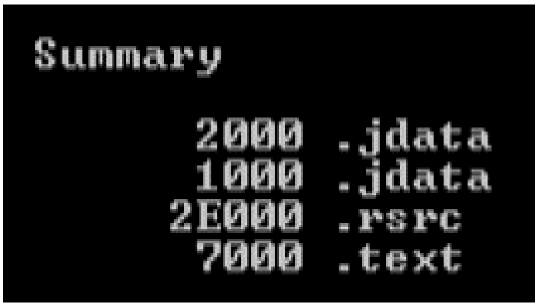
Before jumping right into IDA Pro and tackling the disassembly, it’s often worthwhile to perform initial static analysis of the PE file to get some ideas on packing and other potential code obfuscation techniques. PE parsing utilities can be valuable for getting an initial look at the characteristics of the file. Strings are a good first indicator, and the presence or lack of strings can provide critical insight into the program. Strings are an important part of any program as they are routinely needed for such functionality as making HTTP requests, writing files to disk, looking for processes, and creating files in the file system. Malware authors will often attempt to obfuscate these strings, and an added benefit of packing is that the strings are compressed and encrypted inside, obscuring their discovery. This sample presents some strings, but most of these come from the functions that it is importing. Outside of that, there are no further indicators such as command and control (C2) URLs or IPs, indications of file or process activity, or evidence of intended behavior such as a ransom note.



```
wsprintfW
GetDlgItemTextA
DialogBoxParamW
user32.dll
&Ol{)9
LJM}
JOQ_
Xqvu
KQ/O
```

Sample of strings output using strings utility

Sections of the PE file are also worth investigating. Sections provide structure to the PE file for such items as the executable code and hard-coded data. In addition, they may provide evidence of malware that is packed. There are usually two strong indicators: the name of the section and the entropy of the section. Section names are arbitrary, but some packers use consistent naming and allow for easier detection. Entropy is a measure of the randomness in a sequence of bytes, which make up the content of the sections. This is usually measured on a scale from zero to eight, with eight being the highest measure of entropy. Programs that contain sections with high entropy are more suspect for packing and other obfuscation techniques, since this garbled code tends to be more random and less deliberate.



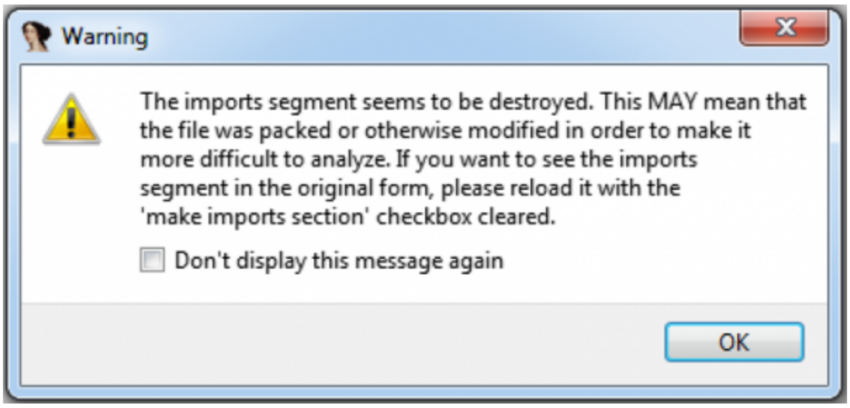
```
Summary

       2000 .jdata
       1000 .jdata
      2E000 .rsrc
       7000 .text
```

Dumpbin output of PE file sections

While there are other indicators to consider, it appears this program is packed and will require deeper investigation.

PACKING ANALYSIS AND CODE OBFUSCATION

Now we can turn to IDA Pro to start analyzing the code of this program. Upon loading the file, IDA provides further indications that the sample is packed.

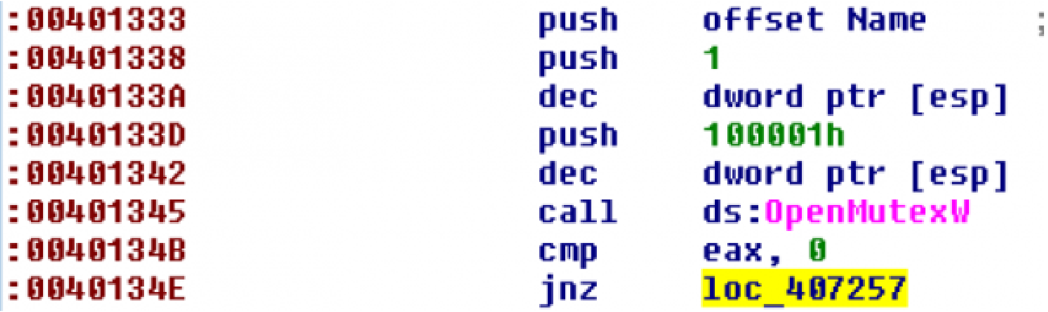


The imports segment seems to be destroyed. This MAY mean that the file was packed or otherwise modified in order to make it more difficult to analyze. If you want to see the imports segment in the original form, please reload it with the 'make imports section' checkbox cleared.

☐ Don't display this message again

IDA Pro dialog indicating potential packing

This program begins with a lot of instructions, most of them unnecessary. One way to try to filter through this code is to see how the registers, variables, and functions are being used. In this first code block, there are several function calls where the return value (in EAX) is being used in a compare/conditional jump combination. The conditional jump goes to *loc_407257*.

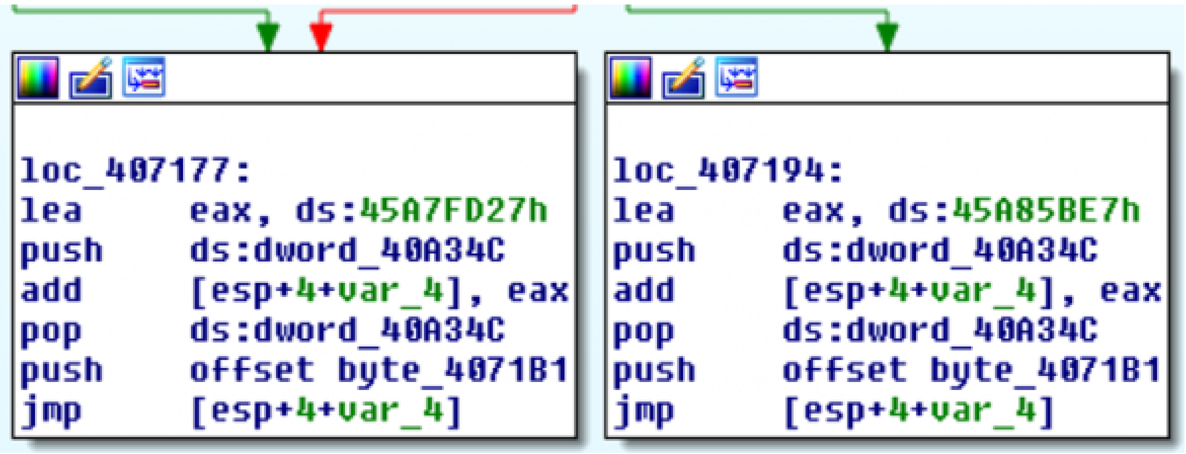


```
:00401333                    push    offset Name     ;
:00401338                    push    1
:0040133A                    dec     dword ptr [esp]
:0040133D                    push    100001h
:00401342                    dec     dword ptr [esp]
:00401345                    call    ds:OpenMutexW
:0040134B                    cmp     eax, 0
:0040134E                    jnz     loc_407257
```

If we navigate to that location, we end up in an infinite loop. This is helpful, as we can now start to visually filter out this noise and attempt to find the true purpose of this code. Since we suspect that we are looking at purely packing code, we don't want to spend a lot of time analyzing how this code works but find the point at which it's done. This will allow us to focus on whatever is unpacked. With unpacking code, I've often found that you can concentrate on the end of the functions and look for abnormal returns or control transfers. This function ends with a function call, which is far from a normal epilogue.

```
:00401466                    call    ds:OpenMutexW
:0040146C                    test    eax, eax
:0040146E                    jnz     loc_407257
:00401474                    call    sub_407027
```

Tracing into function *sub_407027*, we can investigate the code at the end. It appears there are two possible paths for it to go, both with unconventional methods of going there.



```
loc_407177:
lea     eax, ds:45A7FD27h
push    ds:dword_40A34C
add     [esp+4+var_4], eax
pop     ds:dword_40A34C
push    offset byte_4071B1
jmp     [esp+4+var_4]
```

```
loc_407194:
lea     eax, ds:45A85BE7h
push    ds:dword_40A34C
add     [esp+4+var_4], eax
pop     ds:dword_40A34C
push    offset byte_4071B1
jmp     [esp+4+var_4]
```

This function uses a technique of pushing a DWORD value onto the stack and then jumping to ESP. What is pushed onto the stack is actually an address: *0x4071B1*. This technique has actually prevented IDA from identifying the correct location and continuing with disassembly. If we go to that location manually, however, we can tell IDA to disassemble this code.

```
004071B1 byte_4071B1     db 0FFh, 15h            ; DATA XREF: sub_407027+165↑o
004071B1                                         ; sub_407027+182↑o
```

Unanalyzed JMP target

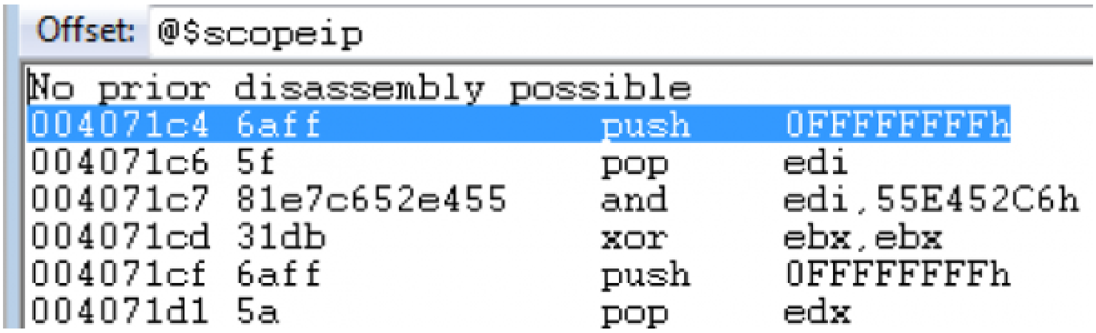
```
:004071B1 loc_4071B1:                            ; DATA XREF: sub_407027+165↑o
:004071B1                                        ; sub_407027+182↑o
:004071B1              call    ds:dword_40A34C
```

Disassembled location 0x4071B1

Once the data at this location is disassembled, we reveal a call instruction with a call target of *dword_40A34C*. The value of this DWORD is not hard-coded, which means it is populated during runtime. Instead of continuing with static-analysis, we can now turn to WinDbg for dynamic analysis to see where this call goes.

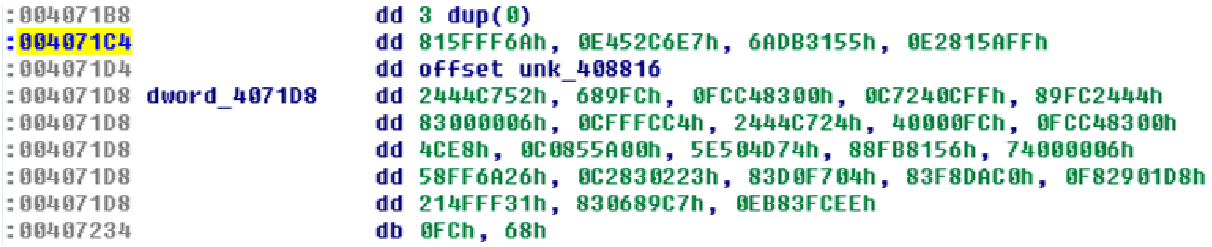
Switching to Dynamic Analysis

Setting a breakpoint on that call instruction reveals that the call target is to location *0x4071c4*.



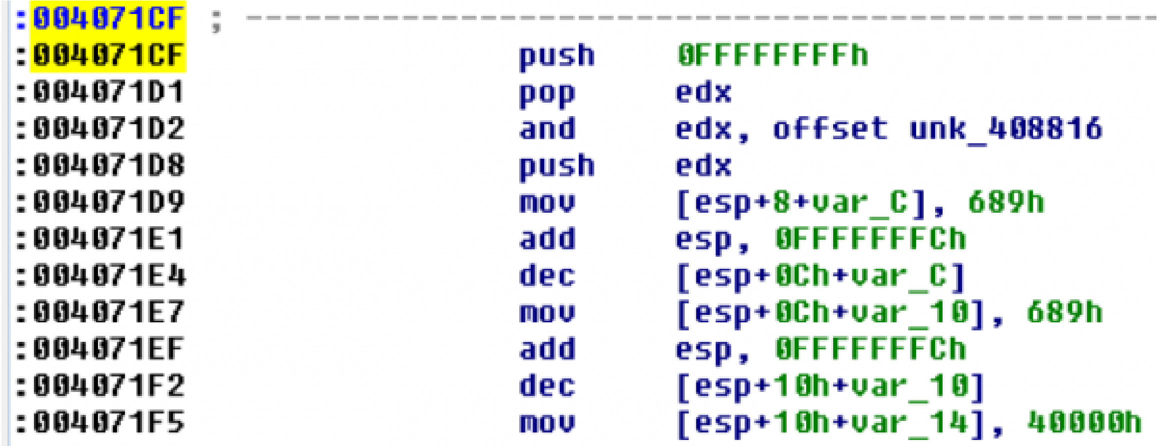
```
Offset: @$scopeip
No prior disassembly possible
004071c4 6aff              push    0FFFFFFFFh
004071c6 5f                pop     edi
004071c7 81e7c652e455      and     edi,55E452C6h
004071cd 31db              xor     ebx,ebx
004071cf 6aff              push    0FFFFFFFFh
004071d1 5a                pop     edx
```

Since IDA was unable to find this location during static analysis, it initially shows up as data instead of instructions.



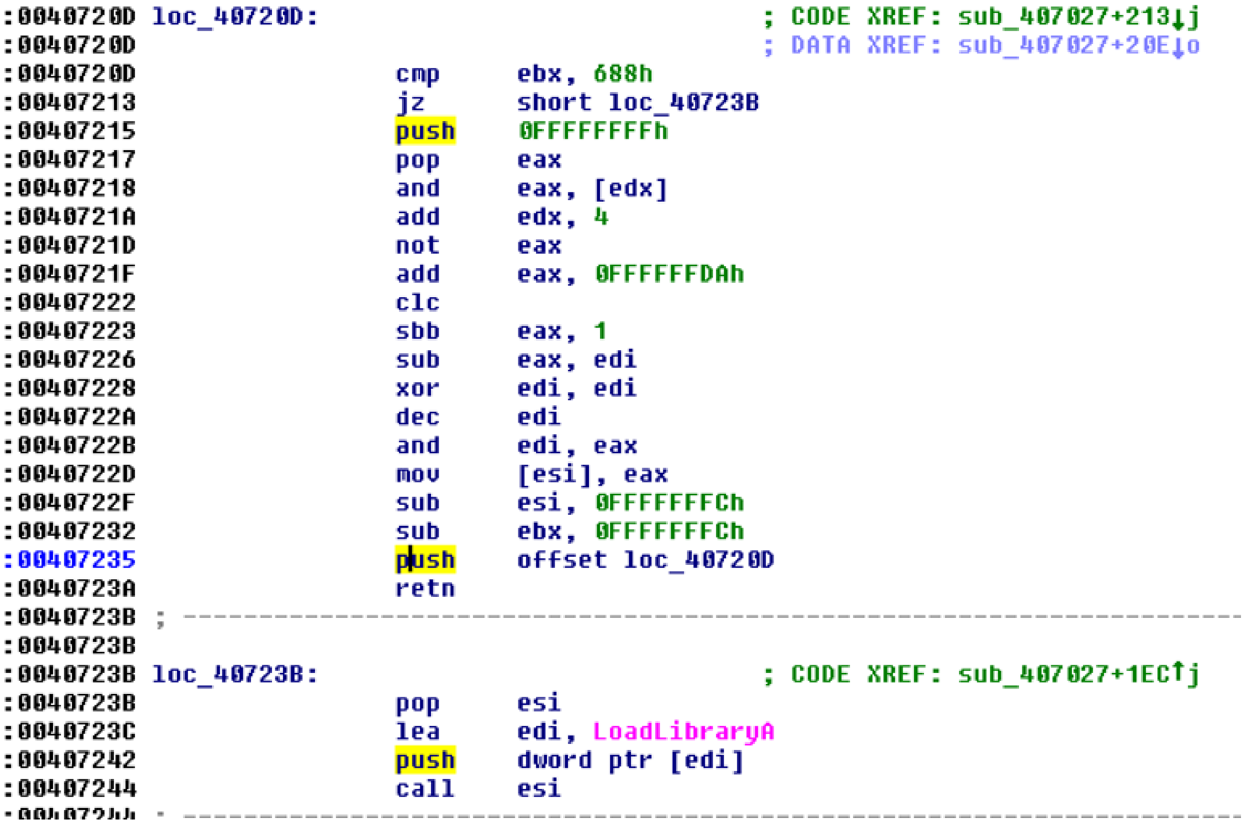
```
:004071B8                 dd 3 dup(0)
:004071C4                 dd 815FFF6Ah, 0E452C6E7h, 6ADB3155h, 0E2815AFFh
:004071D4                 dd offset unk_408816
:004071D8 dword_4071D8    dd 2444C752h, 689FCh, 0FCC48300h, 0C7240CFFh, 89FC2444h
:004071D8                 dd 83000006h, 0CFFFCC4h, 2444C724h, 40000FCh, 0FCC48300h
:004071D8                 dd 4CE8h, 0C0855A00h, 5E504D74h, 88FB8156h, 74000006h
:004071D8                 dd 58FF6A26h, 0C2830223h, 83D0F704h, 83F8DAC0h, 0F82901D8h
:004071D8                 dd 214FFF31h, 830689C7h, 0EB83FCEEh
:00407234                 db 0FCh, 68h
```

Invoking IDA's analysis reveals the disassembled instructions:



```
:004071CF ; ---------------------------------------------------------------
:004071CF                 push    0FFFFFFFFh
:004071D1                 pop     edx
:004071D2                 and     edx, offset unk_408816
:004071D8                 push    edx
:004071D9                 mov     [esp+8+var_C], 689h
:004071E1                 add     esp, 0FFFFFFFCh
:004071E4                 dec     [esp+0Ch+var_C]
:004071E7                 mov     [esp+0Ch+var_10], 689h
:004071EF                 add     esp, 0FFFFFFFCh
:004071F2                 dec     [esp+10h+var_10]
:004071F5                 mov     [esp+10h+var_14], 40000h
```

It's easy to get lost in the assembly here and important to keep the big picture in mind. This code is all likely unpacking code, so let's analyze it a little further down to see how it ends. There is a strange indirect call to ESI at *0x407244*.



```
:0040720D loc_40720D:                             ; CODE XREF: sub_407027+213↓j
:0040720D                                         ; DATA XREF: sub_407027+20E↓o
:0040720D                 cmp     ebx, 688h
:00407213                 jz      short loc_40723B
:00407215                 push    0FFFFFFFFh
:00407217                 pop     eax
:00407218                 and     eax, [edx]
:0040721A                 add     edx, 4
:0040721D                 not     eax
:0040721F                 add     eax, 0FFFFFFDAh
:00407222                 clc
:00407223                 sbb     eax, 1
:00407226                 sub     eax, edi
:00407228                 xor     edi, edi
:0040722A                 dec     edi
:0040722B                 and     edi, eax
:0040722D                 mov     [esi], eax
:0040722F                 sub     esi, 0FFFFFFFCh
:00407232                 sub     ebx, 0FFFFFFFCh
:00407235                 push    offset loc_40720D
:0040723A                 retn
:0040723B ; ---------------------------------------------------------------------------
:0040723B
:0040723B loc_40723B:                             ; CODE XREF: sub_407027+1EC↑j
:0040723B                 pop     esi
:0040723C                 lea     edi, LoadLibraryA
:00407242                 push    dword ptr [edi]
:00407244                 call    esi
:00407244 ; ---------------------------------------------------------------------------
```

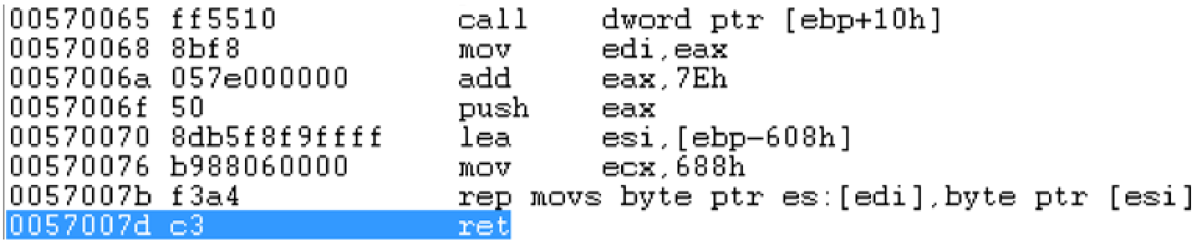
If we continue execution to this point, we can see where it intends to lead. In this case, it’s to an address not in the original image – *0x57000* for this run. This address will change, as it’s a region of read-write-execute memory that is allocated during runtime.

```
00407242 ff37          push     dword ptr [edi]
00407244 ffd6          call     esi {00570000}
00407246 0000          add      byte ptr [eax],
```

This tells us that the previous code was responsible for not only allocating this memory, but also for staging shellcode for execution. Using a tool like Process Hacker, we can extract this shellcode from memory and disassemble it.

Tracing the Shellcode

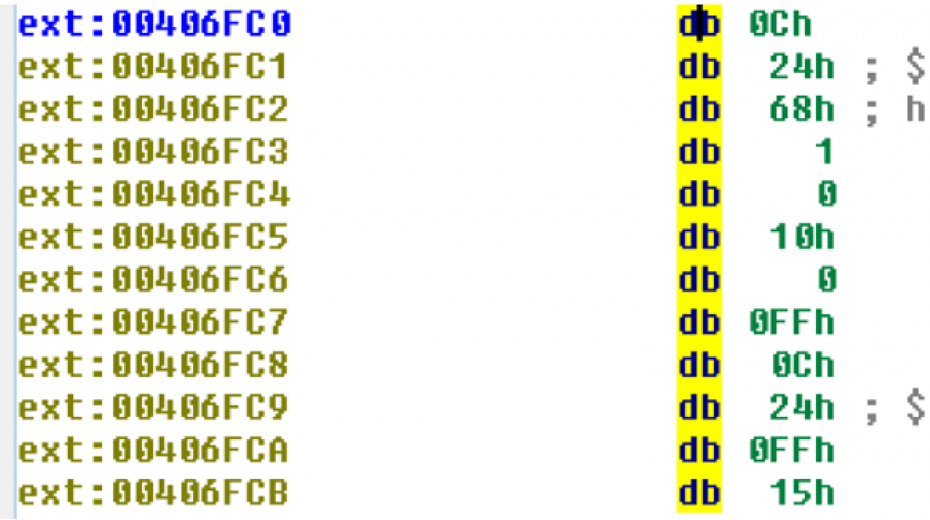
Fortunately, we know the entry point is at the beginning of the binary content from our dynamic analysis. Once this shellcode is disassembled, there will be a considerable amount of code to analyze. Let’s stick with the same approach we used to get here in the first place and analyze instructions toward the end of the shellcode. This shellcode ends with a PUSH/RET technique. The location the author wants to return to is pushed on the stack just before the return instruction.



```
00570065 ff5510          call    dword ptr [ebp+10h]
00570068 8bf8            mov     edi,eax
0057006a 057e000000      add     eax,7Eh
0057006f 50              push    eax
00570070 8db5f8f9ffff    lea     esi,[ebp-608h]
00570076 b988060000      mov     ecx,688h
0057007b f3a4            rep movs byte ptr es:[edi],byte ptr [esi]
0057007d c3              ret
```

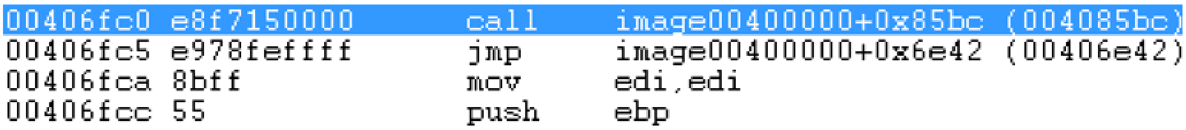
This goes further into the shellcode. However, if we trace to the end of this code, there is a *jmp esi*. ESI contains an address of *0x406FC0*. This is a good sign, as it is taking execution back to an address in the original address space of the program. But is it the same code? By comparing the original data at the location to what is now in memory, a different result means that unpacking could be complete.

Original:



```
ext:00406FC0                 db  0Ch
ext:00406FC1                 db   24h ; $
ext:00406FC2                 db   68h ; h
ext:00406FC3                 db     1
ext:00406FC4                 db     0
ext:00406FC5                 db   10h
ext:00406FC6                 db     0
ext:00406FC7                 db  0FFh
ext:00406FC8                 db   0Ch
ext:00406FC9                 db   24h ; $
ext:00406FCA                 db  0FFh
ext:00406FCB                 db   15h
```

In memory:



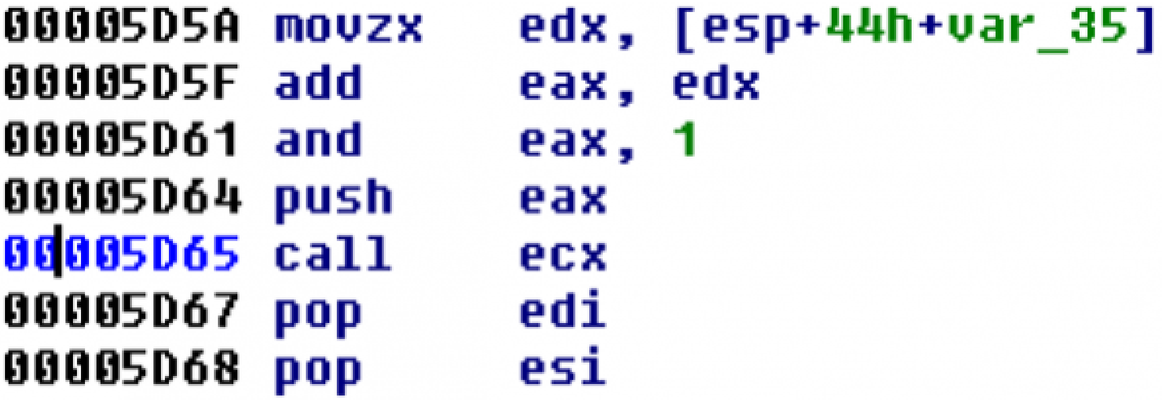
```
00406fc0 e8f7150000       call     image00400000+0x85bc (004085bc)
00406fc5 e978feffff       jmp      image00400000+0x6e42 (00406e42)
00406fca 8bff             mov      edi,edi
00406fcc 55               push     ebp
```

The Plot Thickens

Unfortunately, the malware is not yet ready to reveal what it is up to. Prior to performing a deep technical analysis, automated dynamic analysis was used to understand as much of this program's behavior as possible. This malware makes a request to *hxxps://real-estate-advisors[.win]* and starts another process. This is likely the point at which the malware receives code for its true intended purpose. However, if we let the program run from this point, the request isn't made and no additional processes are created. Not only do we now know that it's not done unpacking/deobfuscating, it is also exhibiting anti-analysis techniques not observed in our manual sandbox environment.

Looking at the cross-reference graph from *sub_406FC0*, there is a considerable amount of code. How do we overcome this mess? One method is to start by setting breakpoints on expected. For example, *CreateProcessA* or *InternetOpenURLA*. Letting this code run ends in a call to *TerminateProcess*, and in this case none of these breakpoints were hit. This could indicate a few things, including anti-analysis techniques. Instead of trying to analyze this function from the top-down, focusing on the call instructions towards the end of the function may speed up analysis. Especially if this involves more unpacking, then the earlier function calls will likely be for memory allocation and more unpacking, and the later function calls for executing the unpacked code. This function ends with three function calls and after inspecting them, the call to *sub_5200* appears to be the most promising.

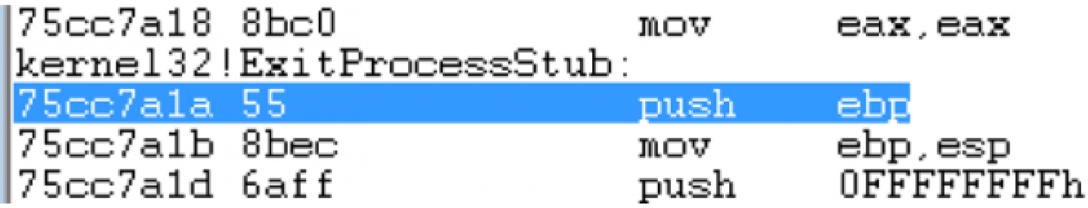
Again, we're faced with a significant amount of code and a limited amount of time for analysis, so let's focus on the end of the function. Toward the end of this function is another indirect function call. These are usually interesting as they may indicate a dynamically-generated address.



```
00005D5A movzx   edx, [esp+44h+var_35]
00005D5F add     eax, edx
00005D61 and     eax, 1
00005D64 push    eax
00005D65 call    ecx
00005D67 pop     edi
00005D68 pop     esi
```

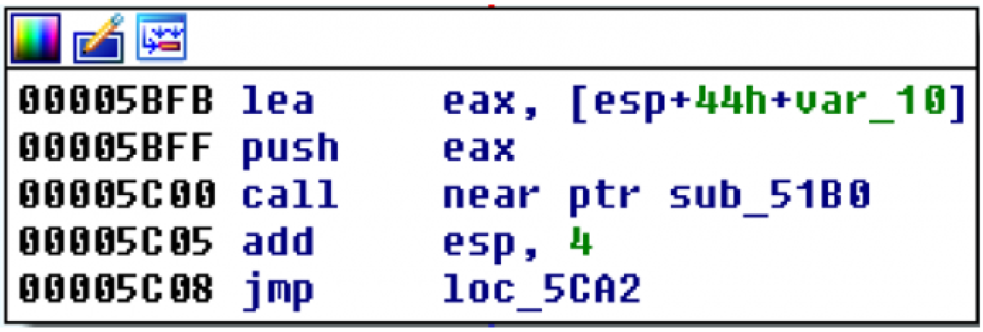
Indirect function call at offset 0x5D65

As it turns out, this is the call to *ExitProcess*, so somewhere before this call is not only any anti-analysis, but also the next stage of functionality.



```
75cc7a18 8bc0            mov     eax,eax
kernel32!ExitProcessStub:
75cc7a1a 55              push    ebp
75cc7a1b 8bec            mov     ebp,esp
75cc7a1d 6aff            push    0FFFFFFFFh
```

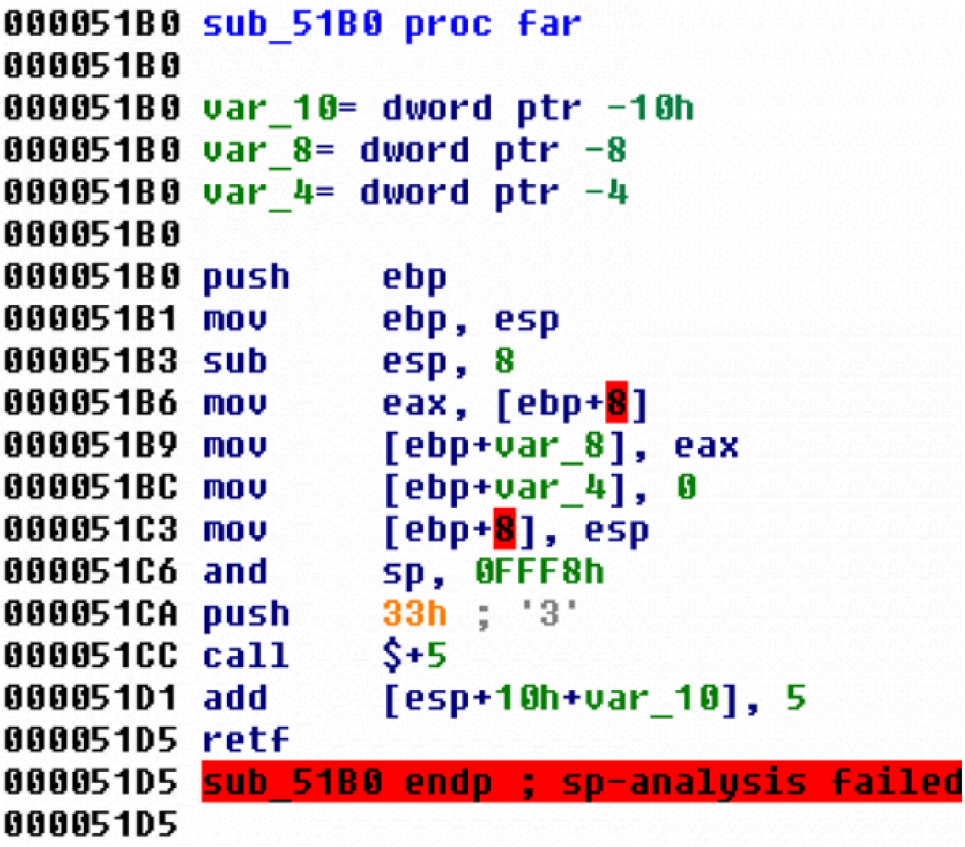
After spending some time analyzing this function, another promising location presents itself:



```
00005BFB lea     eax, [esp+44h+var_10]
00005BFF push    eax
00005C00 call    near ptr sub_51B0
00005C05 add     esp, 4
00005C08 jmp     loc_5CA2
```

Call instruction to offset 0x51B0 at location 0x5C00

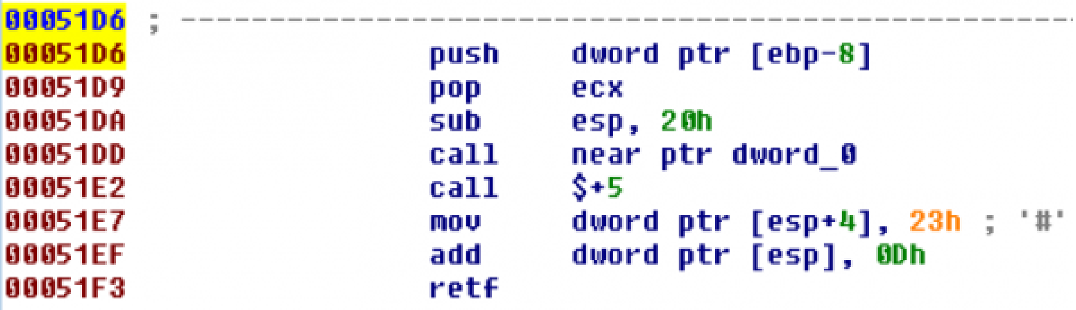
This function is limited in functionality, but it ultimately proves to be the location responsible for the next stage of this malware.



```
000051B0 sub_51B0 proc far
000051B0
000051B0 var_10= dword ptr -10h
000051B0 var_8= dword ptr -8
000051B0 var_4= dword ptr -4
000051B0
000051B0 push    ebp
000051B1 mov     ebp, esp
000051B3 sub     esp, 8
000051B6 mov     eax, [ebp+8]
000051B9 mov     [ebp+var_8], eax
000051BC mov     [ebp+var_4], 0
000051C3 mov     [ebp+8], esp
000051C6 and     sp, 0FFF8h
000051CA push    33h ; '3'
000051CC call    $+5
000051D1 add     [esp+10h+var_10], 5
000051D5 retf
000051D5 sub_51B0 endp ; sp-analysis failed
000051D5
```

The *call $+5* is a common shell code technique to get the address of the stack, as the call instruction will push the address of the next instruction (*add [esp+10h+var_10], 5)* onto the stack and then add 5 to it. The push instruction will push the address 0x51D5 onto the stack, once 5 is added to it the address that this function will return to is

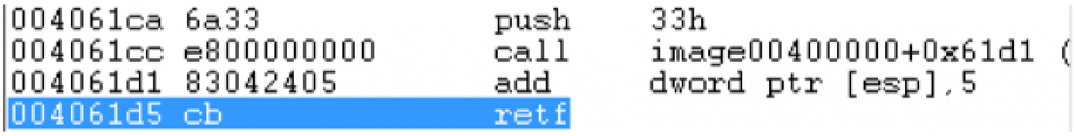
0x51D6. This takes execution to the first instruction after the return. Since IDA was not able to follow this logic, we need to disassemble the code at this location.



```
000051D6 ; ---------------------------------------------------
000051D6                 push    dword ptr [ebp-8]
000051D9                 pop     ecx
000051DA                 sub     esp, 20h
000051DD                 call    near ptr dword_0
000051E2                 call    $+5
000051E7                 mov     dword ptr [esp+4], 23h ; '#'
000051EF                 add     dword ptr [esp], 0Dh
000051F3                 retf
```

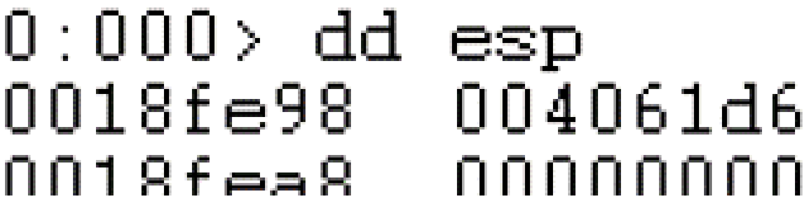
There’s a call to DWORD_0, which actually represents the beginning of this section of code (.TEXT section). We can resume our dynamic analysis to continue to trace this code.

Setting the appropriate breakpoints, I stopped at the *RETF* to ensure that my analysis of where this code was going to return to was correct.



```
004061ca 6a33            push    33h
004061cc e800000000      call    image00400000+0x61d1 (
004061d1 83042405        add     dword ptr [esp],5
004061d5 cb              retf
```

And the value on top of the stack is:

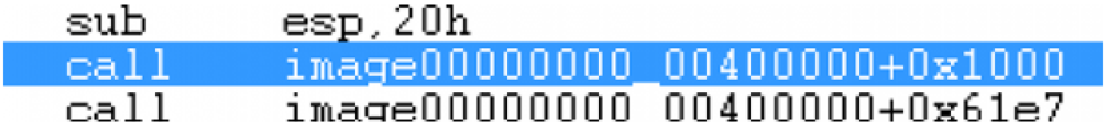


```
0:000> dd esp
0018fe98  004061d6
0018fea8  00000000
```

However, if you trace into this RETF the program doesn’t go to the address we expect:

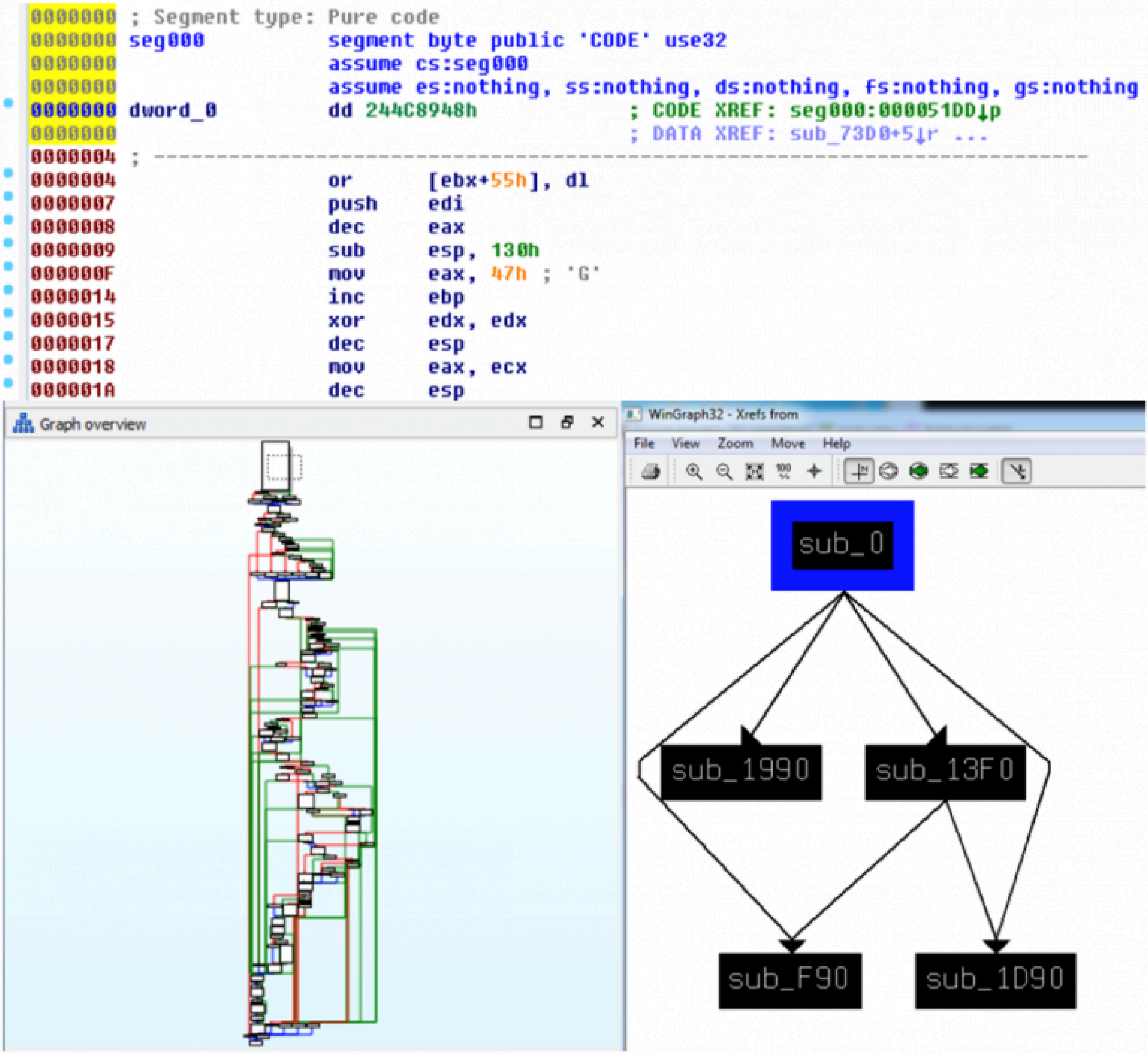
```
77a1fafb 64ff15c0000000  call    dword ptr fs:[0C0h]
77a1fb02 83c404          add     esp,4
77a1fb05 c21800          ret     18h
```

What happened? Turns out, RETF takes two values off of the stack: one value for the segment and a second value for the return address. Notice the *PUSH 33h* before the RETF, this will force the CPU into 64-bit mode instead of 32-bit! Since I was using a 32-bit instance of WinDbg, I was getting unexpected results. Switching to a 64-bit instance of WinDbg allows us to trace into this RETF.



```
    sub     esp,20h
    call    image00000000_00400000+0x1000
    call    image00000000_00400000+0x61e7
```

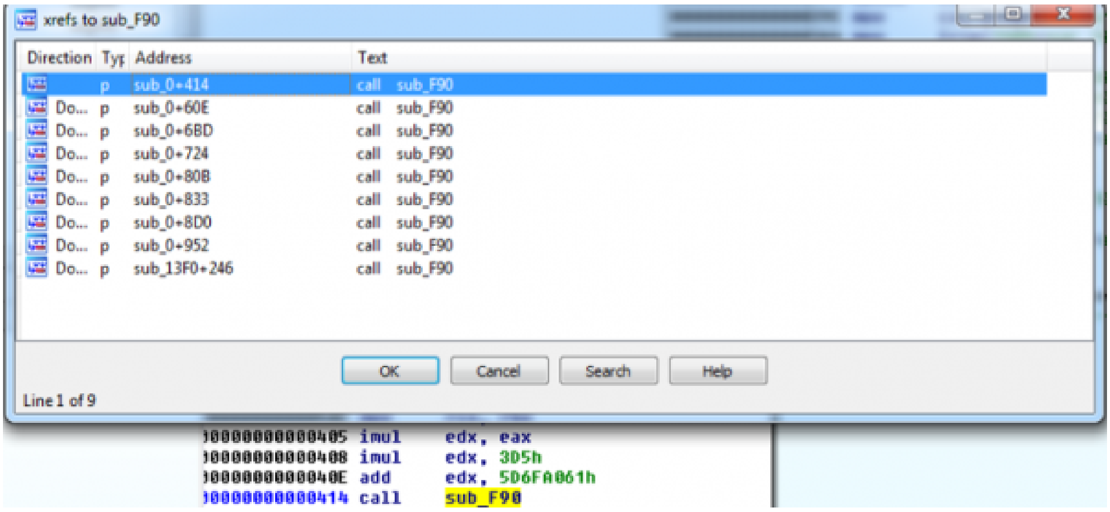
It’s a call to *0x401000*. We have to go back to our original shellcode. IDA wasn’t able to find a reference to this location, so the code was never disassembled. Keep in mind that I extracted this as shellcode from the .text section, so an offset of 0 is equivalent to a virtual address of *0x401000*. We also know something else that is very important–this is 64-bit code. Opening this code with the 64-bit version of IDA gives us an accurate disassembly listing.



```
0000000 ; Segment type: Pure code
0000000 seg000          segment byte public 'CODE' use32
0000000                 assume cs:seg000
0000000                 assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing
0000000 dword_0         dd 244C8948h            ; CODE XREF: seg000:000051DD↓p
0000000                                         ; DATA XREF: sub_73D0+5↓r ...
0000004 ; ---------------------------------------------------------------------------
0000004                 or      [ebx+55h], dl
0000007                 push    edi
0000008                 dec     eax
0000009                 sub     esp, 130h
000000F                 mov     eax, 47h ; 'G'
0000014                 inc     ebp
0000015                 xor     edx, edx
0000017                 dec     esp
0000018                 mov     eax, ecx
000001A                 dec     esp
```

Disassembled 64-bit shellcode, function graph, and call graph

One of the first things to determine is if we can find any API calls. This code doesn't have an extensive call graph, but one function, *Sub_F90*, stands out simply due to the number of times it is called.



Direction	Typ	Address	Text
	p	sub_0+414	call sub_F90
Do...	p	sub_0+60E	call sub_F90
Do...	p	sub_0+6BD	call sub_F90
Do...	p	sub_0+724	call sub_F90
Do...	p	sub_0+80B	call sub_F90
Do...	p	sub_0+833	call sub_F90
Do...	p	sub_0+8D0	call sub_F90
Do...	p	sub_0+952	call sub_F90
Do...	p	sub_13F0+246	call sub_F90

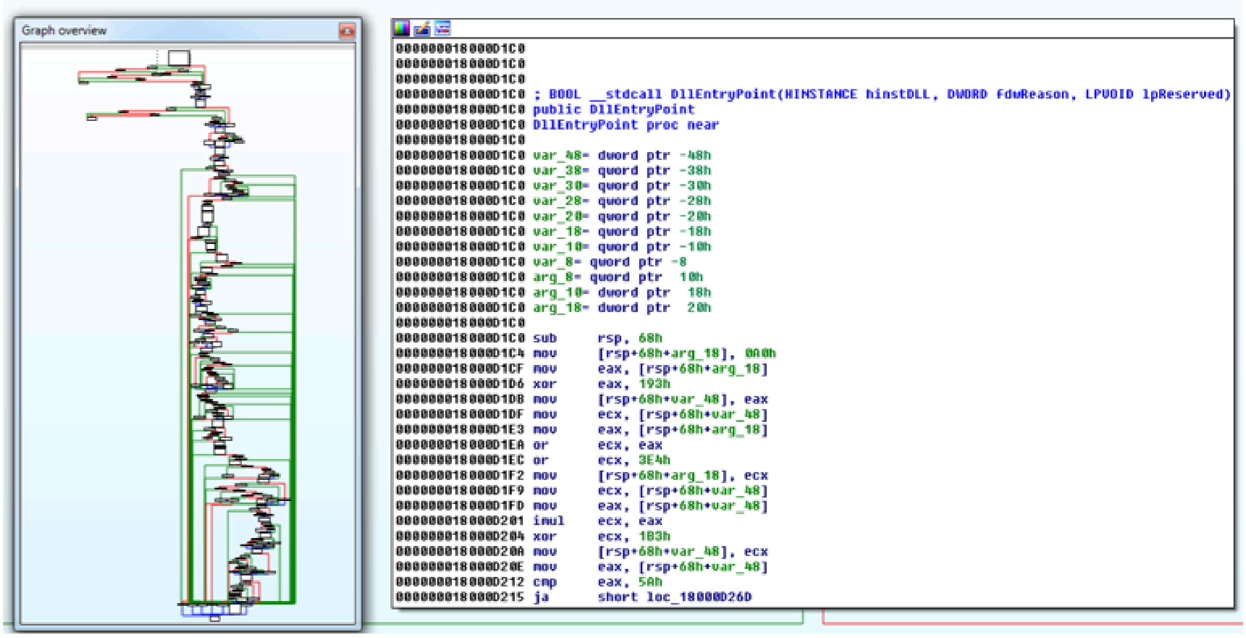
Line 1 of 9

```
000000000000405 imul    edx, eax
000000000000408 imul    edx, 3D5h
00000000000040E add     edx, 5D6FA061h
000000000000414 call    sub_F90
```

Sub_F90 may be responsible for resolving APIs. Setting a breakpoint on this function allows us to investigate the return value in the EAX register. Sure enough, they’re function pointers! Some of the more relevant ones are: *NtAllocateVirtualMemory, NtWirteVirtualMemory, and RtlCreateUserThread.* Following these API calls, it eventually becomes clear that the code is attempting to load a DLL via the *CreateUserThread* method. During execution, the DLL is copied directly into memory and never touches disk! It’s unpacked purely in memory and then loaded into the current process by the createthread call. As this is a “fileless” stage of the attack, extracting this DLL from memory provides the opportunity to continue our analysis.

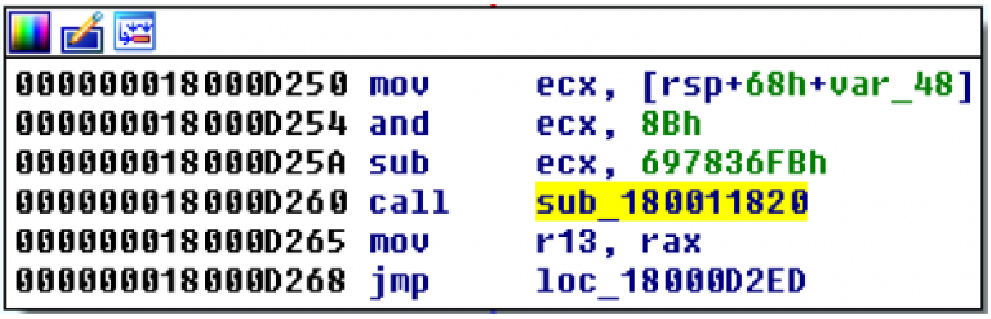
Closer to the End

This DLL has only one export, which is DllEntryPoint (or DLLMain). This is called by the thread created in the previous stage, and it reveals yet another round of complicated code.



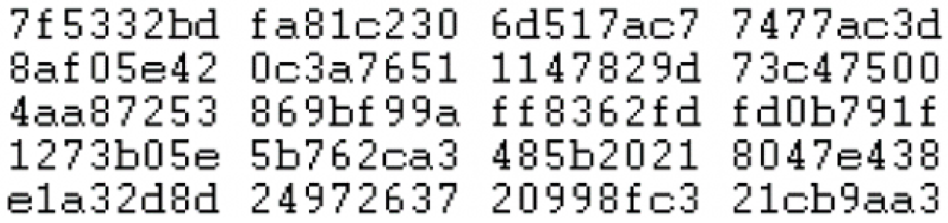
```
000000018000D1C0
000000018000D1C0
000000018000D1C0
000000018000D1C0 ; BOOL __stdcall DllEntryPoint(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpReserved)
000000018000D1C0 public DllEntryPoint
000000018000D1C0 DllEntryPoint proc near
000000018000D1C0
000000018000D1C0 var_48= dword ptr -48h
000000018000D1C0 var_38= qword ptr -38h
000000018000D1C0 var_30= qword ptr -30h
000000018000D1C0 var_28= qword ptr -28h
000000018000D1C0 var_20= qword ptr -20h
000000018000D1C0 var_18= qword ptr -18h
000000018000D1C0 var_10= qword ptr -10h
000000018000D1C0 var_8= qword ptr -8
000000018000D1C0 arg_8= qword ptr  10h
000000018000D1C0 arg_10= dword ptr  18h
000000018000D1C0 arg_18= dword ptr  20h
000000018000D1C0
000000018000D1C0 sub     rsp, 68h
000000018000D1C4 mov     [rsp+68h+arg_18], 0A0h
000000018000D1CF mov     eax, [rsp+68h+arg_18]
000000018000D1D6 xor     eax, 193h
000000018000D1DB mov     [rsp+68h+var_48], eax
000000018000D1DF mov     ecx, [rsp+68h+var_48]
000000018000D1E3 mov     eax, [rsp+68h+arg_18]
000000018000D1EA or      ecx, eax
000000018000D1EC or      ecx, 3E4h
000000018000D1F2 mov     [rsp+68h+arg_18], ecx
000000018000D1F9 mov     ecx, [rsp+68h+var_48]
000000018000D1FD mov     eax, [rsp+68h+var_48]
000000018000D201 imul    ecx, eax
000000018000D204 xor     ecx, 183h
000000018000D20A mov     [rsp+68h+var_48], ecx
000000018000D20E mov     eax, [rsp+68h+var_48]
000000018000D212 cmp     eax, 5Ah
000000018000D215 ja      short loc_18000D26D
```

Similar to the last stage, I was able to identify the function responsible for resolving APIs. In this code, *sub_180011820* returns a function pointer in the RAX register.



```
000000018000D250 mov     ecx, [rsp+68h+var_48]
000000018000D254 and     ecx, 8Bh
000000018000D25A sub     ecx, 697836FBh
000000018000D260 call    sub_180011820
000000018000D265 mov     r13, rax
000000018000D268 jmp     loc_18000D2ED
```

Tracing this allows visibility into the different APIs being called, and that is where the majority of the anti-analysis is employed. For example, there is a call to *CreateToolhelpSnapshot32*, which is then used to look for evidence of sandbox/analysis processes. Each process name is converted to a multibyte string, changed to upper-case, and then used to create a CRC32 checksum. The checksum value is compared to a list of pre-computed values to avoid using any strings in the sample, a deliberate obfuscation technique used to avoid clear-text strings which are easily discovered.



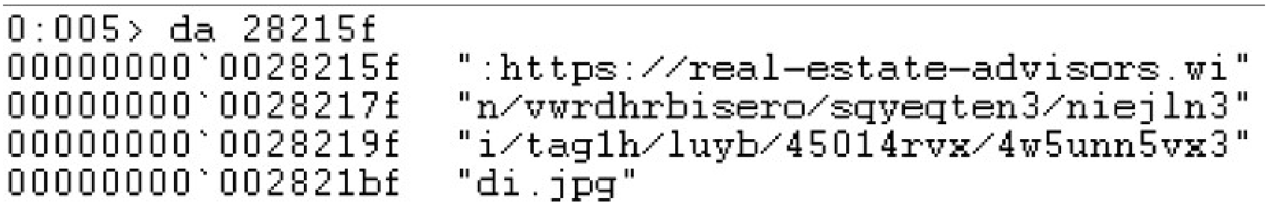
```
7f5332bd fa81c230 6d517ac7 7477ac3d
8af05e42 0c3a7651 1147829d 73c47500
4aa87253 869bf99a ff8362fd fd0b791f
1273b05e 5b762ca3 485b2021 8047e438
e1a32d8d 24972637 20998fc3 21cb9aa3
```

Array of DWORD pre-computed checksum values

```
lea     r8,[rax+r8*4]
cmp     dword ptr [r8],ebx ds:000007fe`f6bb6e08=7f5332bd
```

R8 contains a pointer to pre-computed checksum and compared with dynamically computed checksum from process name in EBX

This code also looks for manufacturer information through a call to *GetSystemFirmwareTable*. Bypassing these checks allows the program to finally deliver its intended payload—to make a request for another stage to ***hxxps://real-estate-advisors[.win]/vwrdhrbisero/sqyeqten3/niejln3i/tag1h/luyb/45014rvw/4w5unn5vx4di.jpg***!



```
0:005> da 28215f
00000000`0028215f  ":https://real-estate-advisors.wi"
00000000`0028217f  "n/vwrdhrbisero/sqyeqten3/niejln3"
00000000`0028219f  "i/tag1h/luyb/45014rvx/4w5unn5vx3"
00000000`002821bf  "di.jpg"
```

This resource is retrieved from a command and control node and then is used to create yet another process. However, this server has now gone offline, but not before its ultimate payload was categorized as a malicious banking trojan by the anti-malware community.

https://www.virustotal.com/#/file/ee32c4e0a4b345029d8b0f5c6534fa9fc41e795cc937d3f3fd743dcb0a1cea35/detection

Despite all of the obfuscation and anti-analysis we have examined together—and the fact that we utilized multiple tools to reveal the complete picture—every stage of this malware would have been safely contained within the Bromium Secure Platform in an isolated micro-virtual machine. Detection failed to stop the initial stages of the attack, which gave the attacker complete freedom to place secondary payloads onto the victim's PC. This one was a banking trojan, but next time it could be something entirely different or completely new. Attackers never stop innovating—and they are always a step or two ahead of detection-focused defenders—so consider application isolation and control using virtualization-based security to protect your endpoints against whatever they come up with next.

Source: https://www.bromium.com/second-stage-attack-analysis/