VB Dropper and Shellcode for Hancitor Reveal New Techniques Behind Uptick

researchcenter.paloaltonetworks.com/2016/08/unit42-vb-dropper-and-shellcode-for-hancitor-reveal-new-techniquesbehind-uptick/

Jeff White

August 22, 2016

By Jeff White

August 21, 2016 at 5:00 PM

Category: Malware, Unit 42

Tags: hancitor

This post is also available in: <u>日本語 (Japanese)</u>

The Hancitor downloader has been relatively quiet since a major campaign back in June 2016. But over the past week, while performing research using Palo Alto Networks AutoFocus, we noticed a large uptick in the delivery of the Hancitor malware family as they shifted away from H1N1 to distribute Pony and Vawtrak executables. In parallel, we received reports from other firms and security researchers seeing similar activity, which pushed us to look into this further.



Figure 1 AutoFocus view of new sessions of Hancitor since July 2016

The delivery method for these documents remained consistent to other common malicious email campaigns. Lures contained subjects related to recent invoices, or other matters requiring the victim's attention, such as an overdue bill. These lures were expected, until we started digging into the actual documents attached and saw an interesting method within the Visual Basic (VB) macros in the attached documents used for dropping the malware.

This blog will review in detail the dropping technique, which isn't technically new, but this was the first time we've seen it used in this way. The end goal is to identify where the binary was embedded, but we'll cover the macro and the embedded shellcode throughout this post.

The Word Document

For this section, we'll be looking at the file with a SHA256 hash of

'03aef51be133425a0e5978ab2529890854ecf1b98a7cf8289c142a62de7acd1a', which is a typical MS Office OLE2 Word Document with your standard ploy to 'Enable Content' and run the malicious macro.



Figure 2 The ploy used by the malicious document

Opening the Visual Basic editor up, we can see two forms and a module for this particular sample.



Figure 3 VBProject components

The Malicious Macro

Visual Basic can directly execute Microsoft Windows API calls, which allows it perform a number of interesting functions -- exactly what this VB code is doing.

#If Win64 Then Private Declare PtrSafe Private Declare PtrSafe Private Declare PtrSafe Private Declare PtrSafe Private Declare PtrSafe	Function maison Lib "kernel52" Alias "VirtuslÄlloc" (ByVal lpaddr As LongFtr, ByVal dwSim Bub talepoin Lib "ntdll" Alias "RtiNoveMemory" (pDst As Any, pBro As Any, ByVal Bytelen A Function malsy Lib "kernel52" Alias "CreateEventA" (lpEventAttributes As Any,bManualReset Punction heirbrush Lib "user52" Alias "CalWindowFrocA" (lpEventMatFunc As LongFtr, hResult As Long Function viriparous Lib "user52" Alias "CalWindowFrocA" (lpEventAttributes Lib LongFtr, hResult As Long
Private Declare PtrSafe	Function functus Lib "kernel32" Alias "GetPriorityClass" (hProcess As LongPtr) As LongPtr
Private Declare PtrSafe	Function headwaters Lib "user32" Alias "GetDlgItem" (ByVal hDlg As LongPtr, nIDDlgItem As
#Else Frivate Declare Functio Frivate Declare Sub tal Frivate Declare Functio Frivate Declare Functio	n maison Lib "kernel32" Alias "VirtualAllor" (ByVal lpaddr As Long, ByVal dwSize As Long, I apoin Lib "ntdll" Alias "RtlMoveMemory" (pDst As Any, pSro As Any, ByVal ByteLen As Long) n twentythird Lib "user32" Alias "EndDialog" (ByVal hDlg As Long, nResult As Long) As Long n chopin Lib "kernel32" Alias "CreatefventA" (lpEventAttributes As Any, bManualReset As Lo
Frivate Declare Functio	n tost Lib "user32" Alias "GetDigitem" (ByVai hDig As Long, AIDDigitem As Long) As Long
Frivate Declare Functio	n hairbrush Lib "user32" Alias "CallWindowProcA" (lpPrevWndFuno As Long, hMnd As Any, Mag i
Frivate Declare Functio	n unselected Lib "kernel32" Alias "GetPriorityClass" (hFrocess As Long) As Long

Figure 4 Microsoft Windows API calls within VB code

As we can see, the macro includes logic to determine the architecture of the system it's running on and has the ability to execute correctly on either 32-bit or 64-bit platforms. The primary calls of interest for us will be VirtualAlloc(), RtlMoveMemory(), and CallWindowProcA().

When we originally started looking at this sample, we were mainly interested in where the payload was being stored, so we began debugging the macro to understand how it functions. The payload in question is base64-encoded and embedded within a form in the VBProject as a value of the 'Text' field on the 'choline' TextBox.

As a side note, what is really interesting is that the authors went through the trouble to actually write their own base64 decoder purely in VB. We'll leave that as an exercise for the reader to dig into that but it's a good overview of how base-N encoding works; the entire 'maria' module within this macro is the base64 decoder.

The macro base64 decodes the payload into a local byte-array and then we come to our first API call, VirtualAlloc().

osier = 0				
indifference	= maison(osier,	4574,	£H1000,	&H40)
heronry = "sc	imitar"			
idaho = "elog	uent"			
=]]				
ocals				
Project.ThisDocument.exmo	or			
Expression	Value			
indifference	94044160			

Figure 5 Memory page being allocated

The call commits specific pages of memory with read, write, and executable (RWX) permissions at 0x59B0000.

Base Address	Туре	Size	Protect
0x59a4000	Free (Unusable)	48 kB	
▷ 0x59b0000	Private	8 kB	RWX
0x59b2000	Free (Unusable)	56 kB	

Figure 6 New memory page with RWX permissions

Afterwards, the VB macro continues to setup the next call to RtlMoveMemory and then calls it with the location of the memory from the previous call and our base64 decoded byte array.

•	talapoin ByVal indifferenc setose = "interfering"	e, townsman(0),	UBound (townsman)	+ 1
	If Win64 Then			
-1	1			
ocals				
Project	.ThisDocument.exmoor			
Project. Expres	. ThisDocument.exmoor		Value	
Project. Expres	. ThisDocument.exmoor usion nsman		Value	
Project Expres	ThisDocument.exmoor usion nsman wwnsman(0)		Value 72	
Expres town to to	ThisDocument.exmoor usion nsman wwnsman(0) wwnsman(1)		Value 72 137	
Expres town to to to	ThisDocument.exmoor sion nsman wnsman(0) wnsman(1) wnsman(2)		Value 72 137 92	

Figure 7 Base64-decoded byte array

We can quickly validate by dumping that region of memory in our WINWORD.EXE process and comparing transferred bytes.

WI	WOF	RD.EX	E.bir	1 🗷						
Ŧ	Edit	As: H	lex 🔻		Run Script 🔻					
		Q	1	2	3	4	5			
000	0h:	48	89	5C	24	08	48			
001	Oh:	89	7C	24	20	41	54			
000		0D	DO	40	0.5	00	OF			

Figure 8 Confirming bytes match from dumped memory

Now that our code has been copied to in executable memory, the macro sets up the last API call for CallWindowProcA(). The first value supplied to this call is our memory offset +2214, which is a function pointer within this code, and the second is a string of the path to our file for a handle. These actions redirect code execution to shellcode.



Figure 9 Passing execution to the shellcode

The Shellcode

If we attach to WINWORD.EXE and break on the offset of our memory location +2214 (0x8A6), the entry point of the shellcode, we can validate program execution shifts to this code path.

- 44 X		13 53 13 4 1 4			-			• •			~ 17	~		-the state		100			
	55 REFC	PUSH EBP	•	3	-		14		*	Ű		6	è I	i	Ю	(al		P	AB
05960009	81EC 3C820000	SUB ESP, 29C PUSH EBX	WINWOR	ID.EX	E.bi	n 🐻													
05988888	56 57	PUSH ESI PUSH EDI	∓ Edit	As: 1	ies ?		Run	Scripit	-	Ru	nTer	nolati	e *						
05988882	8045 D0	LEA ERK, DWORD PTR		0	1	2	3	4	5	6	7	8		-74	B	Q.	D	E:	7
00900006	641A1 30000000	HOU ERK, DWORD PTR	0880h:	88	C3	D3	F8	8D	4E	0A	D2	E3	08	02	C6	42	01	00	80
059B06EF	8840 IC	HOU ERK, DWORD PTR	0890h:	5A	01	FF	45	08	88	45	08	8A	00	FF	45	FC	84	CO	75
05980802	3306	XOR EBK, EBX	08A0h:	9E	SF	5E	5B	C9	C3	5.5									53
059600C7	C745 D4 6F61644	HOU DWORD PTR SSE	OSBOh:	56							81								8B
05988808	661C745 D8 6C6C 885D DA	HOU BYTE PTR SSILL	OSCOh:	4.0	10	ΈF	70	08	38	DB	C7	45	DO	10	-64	72	10	C7	4.5
0598860E	E8 57FEFFFF 59	CALL 0596073A POP ECX	OSDOh:	P4	6F	61	64	44	66	C7	45	DB	60	6C	88	5D	DA	E8	57

Figure 10 Validating shellcode is executing

From here, the shellcode gets the address for LdrLoadDLL() function, which is similar to LoadLibraryEx(), by enumerating the Process Environment Block (PEB) and then begins to hunt for the functions it will use within kernel32.dll.

The values for the functions it's looking for, along with other values, are embedded into the shellcode and built on the stack for later usage.

PUSH EAX PUSH DWORD PTR SS: LEEP-AC], 61707845 HOU DWORD PTR SS: LEEP-AC], 61707845 HOU DWORD PTR SS: LEEP-AC], 61707845 HOU DWORD PTR SS: LEEP-AC], 6456506E HOU DWORD PTR SS: LEEP-AC], 6775656 HOU DWORD PTR SS: LEEP-931, 73676E69 HOU DWORD PTR SS: LEEP-941, 41 CALL 0598073A POP ECX POP ECX PUSH 104 LEA ECX, DWORD PTR SS: LEEP-901 PUSH ECX HOU DWORD PTR SS: LEEP-901, 50405425 HOU DWORD PTR SS: LEEP-901, 50405743 HOU DWORD PTR SS: LEEP-601, 61457243 HOU DWORD PTR SS: LEEP-601, 61457243 HOU DWORD PTR SS: LEEP-901, EAX LEA EAX, DWORD PTR SS: LEEP-901, EAX LEA EAX, DWORD PTR SS: LEEP-701, 74726956 HOU DWORD PTR SS: LEEP-701, 74726956 HOU DWORD PTR SS: LEEP-601, 416567 HOU DWORD PTR SS: LEEP-601, 636F6666 HOU DWORD PTR SS: LE

Figure 11 Embedded data in shellcode

Following these sets of encoded names, we can see the shellcode is interested in the following syscalls: CloseHandle(), ReadFile(), GetFileSize(), VirtualFree(), VirtualAlloc(), and CreateFileA(). For each API call, it looks up the address of the function and stores it on the stack.

Next, the shellcode calls CreateFileA() on the Word document and receives a handle back, which it passes to GetFileSize() for the file size, that is then subsequently passed to VirtualAlloc() to create a section of memory for the file contents (0x2270000). Finally, it reads in the file to that memory location and closes the handle.



Figure 12 Egg hunting by the shellcode

Once it has the copy loaded into memory, it begins a process of hunting through memory for the magic bytes 0x504F4C41, which we can see is located at 0x022836F3 in our new memory page.

1	D Dump - 02270000022A4FFF																	
	022836F3	50	4F	40	41	08	00	FF	FF	AC	3A	01	00	44	42	5F	3F	POLAD. 4:0.08_?
	02283703 02283713 02283723 02283733 02283733 02283743 02283753 02283753 02283753	844444444	++5444458P	581 44 4 4 4 200	+++++++7284	0444446322	0444447151	444442700	534444460	244444460	5444444670	04444444A	0444446612	39 3F F F E 5 6 D 4	13944446880	528F4F4F7127	544444477	0010000300099(0 0\q000000000000000 00000000000000000 000000
	02283783 02283793 02283793	46 57 4F	3D 51	4E IF	67	74 46	41 51 4F	24 43	71 63	6F	41 3F	24 20 25	71	3E 56	534F 0	27 45	44F	F=NgtA\$qoA\$q>S'D WQ * bFQCcT? UV000 000000##215d(95b

Figure 13 Egg located

Now that we've found what's likely to be our binary, the last step is to just decode it. Looking at the shellcode, we can see that it will add 0x3 to each byte starting at 0x22836FF, in our example, and then XOR it by 0x13, as shown below.



Figure 14 XOR decrypting

Once the counter reaches 0x13AAC (80556), it begins a series of sub-routines to manipulate each byte and decrypt the binary. If we set a breakpoint after the decryption routine and check our memory location, we can see that the binary is decoded and can now be dumped for further analysis. The MZ and PE headers can be seen in the following dumped memory.

D Dump - 02270000022A4FFF									
022836F3 50 4F 4C 4 02283703 03 00 00 00 02283723 00 00 00 00 02283723 00 00 00 00 02283723 00 00 00 00 02283753 72 6F 67 7 02283763 20 72 75 60 02283763 20 72 75 60 02283773 72 6F 67 7 02283773 FF AF 77 02283773 FC FF AF 77 02283773 F2 FF AF 77 02283773 F2 FF AF 77 02283773 00 00 00 00 02283773 00 00 00 00 02283753 00 00 00 00 0228383 02 00 00 00 0228383 00 00 00 00 0228383 00 00 00 00 0228383 02 00 00 00 0228383 00 00 00 00 00 00 00	08 00 FF FF AC 3A 01 00 4D 5A 9C 00 POLFE. %:0.16 04 00	ZE							

Figure 15 Decoded binary

For this particular campaign run with this dropper, it places the binary in the %TMP% directory before launching it, which then ends up writing itself to '%SYSTEMROOT%/system32/WinHost.exe'.

At this point, the Hancitor downloader has been fully loaded on the victim's machine, where it will proceed to perform additional malicious activities.

Conclusion

Macro-based techniques are quite common, but the technique being used here with the macro dropper is an interesting variation. From the encoded shellcode within the macro and using native API calls within VB code to pass execution to carving out and decrypting the embedded malware from the Word document, it's a new use of Hancitor that we'll be following closely.

Palo Alto Networks customers are protected from the dropper detailed throughout this blog and its contained Hancitor payload. You can continue to track this threat through the <u>AutoFocus Hancitor tag</u>. Additionally, all Hancitor downloader samples are identified as malicious in WildFire. Domains used by Hancitor are also categorized as malicious.

Acknowledgements

For more analysis of the Hancitor payload, please see this write-up by Minerva Labs.

Indicators of Compromise

Below are some of the most common observed e-mail subjects and file names seen in the latest campaign this week from over 380,000 sessions. Patterns substituted with regex or representation.

Email Subjects

<domain> invoice for <month>

levi.com invoice for august

<domain> bill <domain> deal <domain> receipt <domain> contract <domain> invoice

metlife.com bill metlife.com deal metlife.com receipt metlife.com contract metlife.com invoice

File Names

artifact[0-9]{9}.doc bcbsde.com_contract.doc contract_[0-9]{6}.doc generic.doc price_list.doc_[0-9]{6}.doc report_[0-9]{6}.doc

In addition, we observed these C2 calls out during analysis, which can be detected at your perimeter by the use of '/(sl|zaopy)/gate.php'.

hxxp://betsuriin[.]com/sl/gate.php hxxp://callereb[.]com/zapoy/gate.php hxxp://evengsosandpa[.]ru/ls/gate.php hxxp://felingdoar[.]ru/sl/gate.php hxxp://gmailsign[.]info/plasma/gate.php hxxp://hecksafaor[.]com/zapoy/gate.php hxxp://heheckbitont[.]ru/sl/gate.php hxxp://hianingherla[.]com/sl/gate.php hxxp://hihimbety[.]ru/sl/gate.php hxxp://meketusebet[.]ru/sl/gate.php hxxp://moatleftbet[.]com/sl/gate.php hxxp://moatleftbet[.]com/sl/gate.php hxxp://mopejusron[.]ru/sl/gate.php hxxp://ningtoparec[.]ru/sl/gate.php hxxp://nodosandar[.]com/ls/gate.php hxxp://nodosandar[.]com/zapoy/gate.php hxxp://ritbeugin[.]ru/ls/gate.php hxxp://rutithegde[.]ru/sl/gate.php hxxp://surofonot[.]ru/sl/gate.php hxxp://uldintoldhin[.]com/sl/gate.php hxxp://unjustotor[.]com/sl/gate.php hxxp://wassuseidund[.]ru/sl/gate.php

The below Yara rule can be used to detect this particular dropper and technique described throughout this blog.

```
1
    rule hancitor dropper : vb win32api
2
    {
3
     meta:
4
      author = "Jeff White - jwhite@paloaltonetworks @noottrak"
5
      date = "18AUG2016"
6
      hash1 =
7
    "03aef51be133425a0e5978ab2529890854ecf1b98a7cf8289c142a62de7acd1a"
8
      hash2 =
9
    "4b3912077ef47515b2b74bc1f39de44ddd683a3a79f45c93777e49245f0e9848"
10
      hash3 =
11
    "a78972ac6dee8c7292ae06783cfa1f918bacfe956595d30a0a8d99858ce94b5a"
12
13
     strings:
14
      $api 01 = { 00 56 69 72 74 75 61 6C 41 6C 6C 6F 63 00 } // VirtualAlloc
15
      $api 02 = { 00 52 74 6C 4D 6F 76 65 4D 65 6D 6F 72 79 00 } // RtlMoveMemory
      $api 04 = { 00 43 61 6C 6C 57 69 6E 64 6F 77 50 72 6F 63 41 00 } //
16
17 CallWindowProcAi
18
      $magic = { 50 4F 4C 41 } // POLA
     condition:
      uint32be(0) == 0xD0CF11E0 and all of ($api *) and $magic
    }
```

Get updates from Palo Alto Networks!

Sign up to receive the latest news, cyber threat intelligence and research from us

By submitting this form, you agree to our <u>Terms of Use</u> and acknowledge our <u>Privacy</u> <u>Statement</u>.