H1N1: Technical analysis reveals new capabilities

blogs.cisco.com/security/h1n1-technical-analysis-reveals-new-capabilities

Josh Reynolds September 13, 2016

This blog is the first in a 3 part series that will provide an in-depth technical analysis on the H1N1 malware. I'll be looking at how H1N1 has evolved, its obfuscation, analyzing its execution including new information stealing and user account control bypass capabilities, and finally exploring how we are both using and influencing security tools with this research.

Overview

Through the use of general characteristics exhibited by malware authors we are able to broadly categorize and positively identify malicious samples. These characteristics, discussed in *The General Behavior of* Ransomware are indexed in a database, which allows us to identify patterns, outliers and obtain greater visibility and insight into various threats.

H1N1's evolution: past and present

These data sets provide insight into the ever-growing attack vectors that affect our customers, which include malware delivery mechanisms. In this blog series we highlight newly added functionality to a malware variant that started out as being a 'loader' (strictly provides capabilities of loading other more complex malware variants) known as H1N1, and has now evolved into an information stealing variant.

Throughout the data mining exercises conducted by my colleagues and I on the AMP Threat Grid Research & Efficacy Team (RET) we have observed a widely distributed campaign using VBA macros to infect machines with a variant of information-stealing malware. Based on the initial characteristics observed by AMP Threat Grid we believed these malicious documents were distributing a Ransomware variant; however, we later found the dropped executables to be a variant of the H1N1 loader. H1N1 is a loader malware variant that has been known to deliver Pony DLLs and Vawtrak executables to infected machines. Upon infection, H1N1 previously only provided loading and system information reporting capabilities.[1,2]

Key findings from our analysis include:

- Unique obfuscation techniques
- A novel DLL hijacking vulnerability resulting in a User Account Control bypass
- Added information stealing capabilities
- Self-propagation/lateral movement capabilities

Background

H1N1 has added a plethora of new functionality in comparison to earlier reports. Throughout this blog series we will be analyzing the capabilities of H1N1 including: obfuscation, a User Account Control (UAC) bypass, information stealing, data exfiltration, loader/dropper, and self-propagation/lateral movement techniques used by this variant.[1,2]

Infection Vector

The use of Visual Basic macros is nothing new, however, in recent months they have become one of the most popular infection vectors for all malware types, especially for Ransomware campaigns. These macros vary in sophistication from performing the download and execution of hosted binaries, to dropping the binaries themselves. In this campaign we see the latter where the document ships an entire encoded binary within the text box of a VBA macro form. All documents throughout this campaign have used a common naming convention in the following formats:

- *[domain]_card_screenshot.doc*
- *confirmation_[random integers].doc*
- *bank_confirmation_[random integers].doc*
- *debit_request_[random integers].doc*
- *creditcard_statement_[random integers].doc*
- *insurance_[random integers].doc*
- *inventory_list_[random integers].doc*
- *debt_[random integers].doc*

The domains for the first format observed include the financial, energy, communications, military and government sectors. Unsurprisingly, these documents are delivered through spear-phishing e-mail campaigns. A number of subject headings can be observed in VirusTotal:

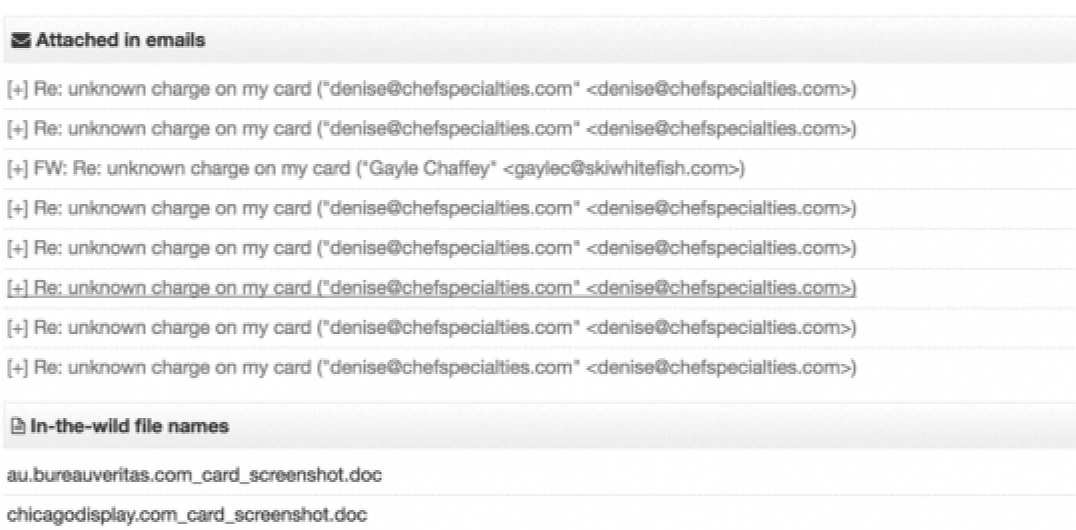


Figure 1.0: Attached e-mail subject headings in VirusTotal for identified documents

Although the specified domain in the filename differentiates between targets, the lure message within the phishing e-mail does not vary drastically, for example:

```
What is this $351.20 charge on my credit card?
It shows this amount charged by cchsfs.com.
Please check the screenshot i have attached and tell me what is this about?
Thank you
Denise Allen
Chef Specialties
P: 814.8206435
F: 814.8784406

From nobody Thu Jun 16 20:29:02 2016
Content-Type: application/msword;
        name="cchsfs.com_card_screenshot.doc"
```

Figure 2.0: Example phishing message within attached e-mail

The remaining formats appear to simply seem enticing enough to open being related finance, corporate or personal information.

Upon opening the document, the attacker attempts to social engineer the user into executing the malicious macro content by stating it will adjust to their version of Microsoft Word:

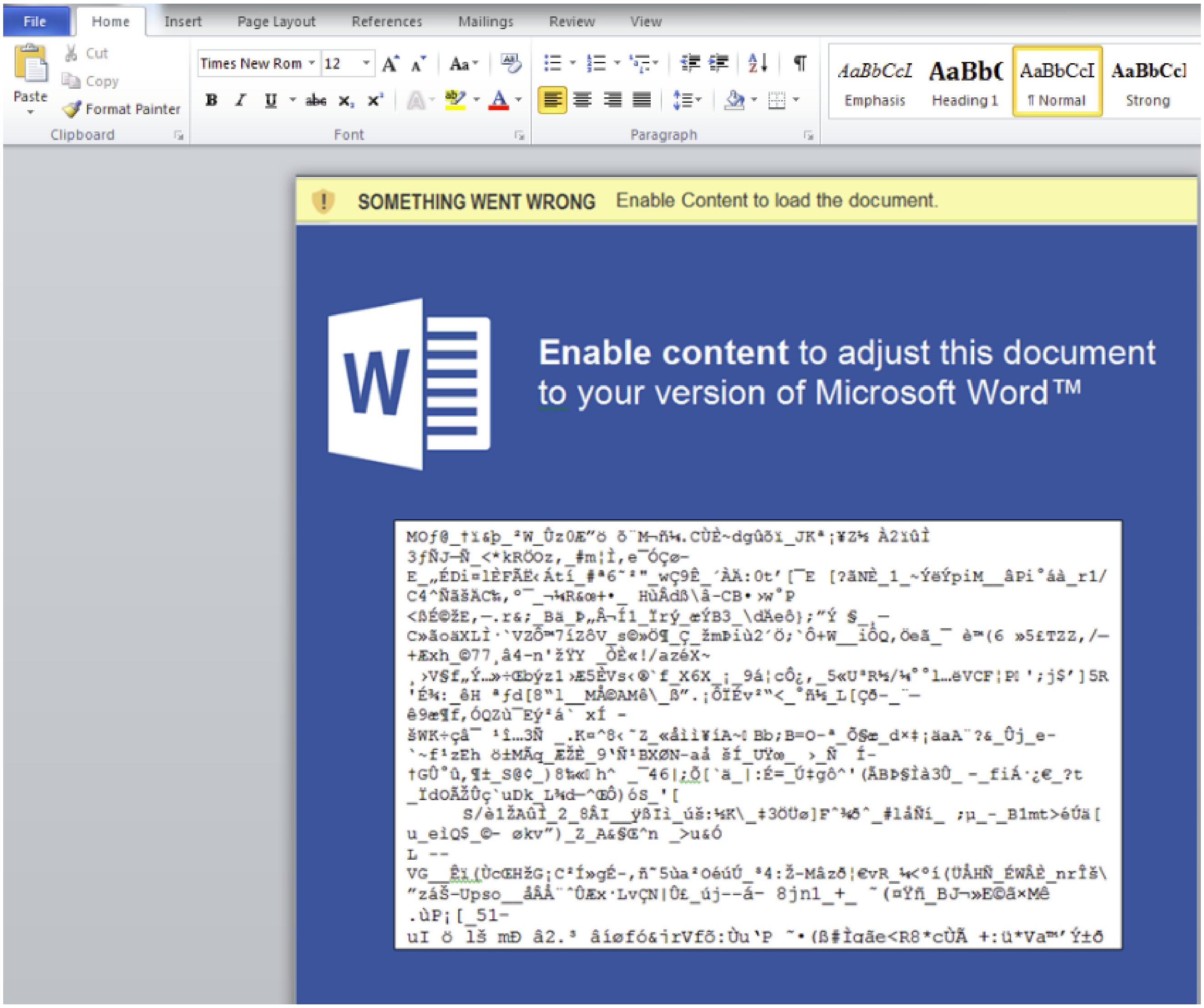


Figure 3.0: Social engineering content of document to open macros

Dropper Obfuscation

The VBA macro is highly obfuscated, making use of many VBA tricks to hide its true intent. These include the use of string functions: StrReverse, Ucase, Lcase, Right, Mid, and Left. For example, the following gets the %temp% path:

```
Function backgeared()
If tan(63) > 73 Then
foolishness = excathedra
Else
Dim notemigonus As Byte
Dim nauseating As Variant
user = Mid("vitiumScrmoider", 7, 3) & Mid("nappyiptinloam", 6, 5) & Mid("clovisg.biology", 7, 2)
End If
misanthrope = 28 + 40
Select Case misanthrope
Case 26 To 33
Dim herbivorous As Variant

Dim debatable As Long

attempered = incompetent
Case 68
cynoglossidae = Ucase("Fi") + StrReverse("tcejbOmetsySel")
south = user + cynoglossidae
Dim corner As Long

End Select
Set sung = VBA.CreateObject(south)
ies = 79 + 5 - 83
backgeared = CallByName(sung, "GetSpecialFolder", ies, 27 - 21 - 4)
End Function
```

Figure 4.0:

String obfuscation mechanisms to get %temp%

Mid is used here to produce ".*Scripting*", Ucase and StrReverse are used to produce "FIleSystemObject", which is used to create a VBA FileSystemObject, that is then used with GetSpecialFolder, and some basic arithmetic resulting in "2" to get %temp%.As mentioned, the binary to be executed is extracted from a VBA form text box:

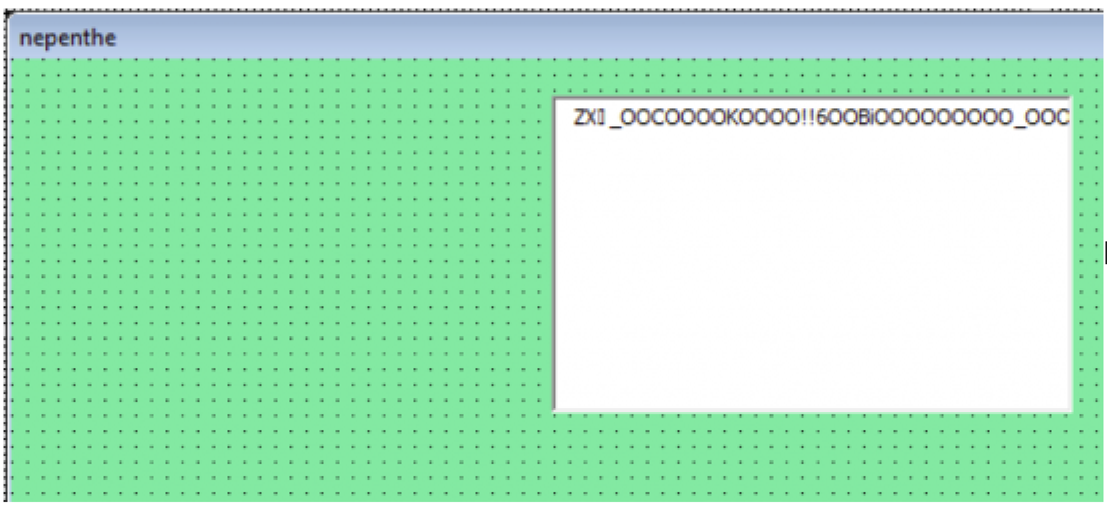


Figure 5.0:

VBA form containing obfuscated PE within text box

The text box content is set into a variable, which is then passed off to a de-obfuscation function. The core de-obfuscation functionality is a two steps process. The first is an XOR loop with a fixed byte key of 0xE, which produces a base64 encoded portable executable (PE):

```
For winesap = 0 To UBound(aestas)
aestas(winesap) = aestas(winesap) Xor 14
Next winesap
```

Figure 6.0: XOR decoding/de-obfuscation loop

The second is a VBA implementation of base64 that decodes it to produce a final Portable Executable (PE):

```vba
kahikatea = StrConv(aestas, vbUnicode)
schoolmistress = 1
For omnipresence = 0 To 255
Select Case omnipresence
Case 65 To 90
adaptability(omnipresence) = omnipresence - 65
Case 97 To 122
adaptability(omnipresence) = omnipresence - 71
Case 48 To 57
adaptability(omnipresence) = omnipresence + 4
Case 43
adaptability(omnipresence) = 62
Case 47
adaptability(omnipresence) = 63
End Select
Next omnipresence
For omnipresence = 0 To 63
ulva(omnipresence) = omnipresence * swap
crookneck(omnipresence) = omnipresence * siaats
lepisosteus(omnipresence) = omnipresence * gallantly
Next omnipresence
crestfallen = StrConv(kahikatea, vbFromUnicode)
exact = 51 - 47
ReDim sheath((((UBound(crestfallen) + 1) \ exact) * 3) - 1)
For fief = 0 To UBound(crestfallen) Step 4
frugal = lepisosteus(adaptability(crestfallen(fief))) + crookneck(adaptability(crestfallen(fief + 1))) + _
ulva(adaptability(crestfallen(fief + 2))) + adaptability(crestfallen(fief + 3))
omnipresence = frugal And dicrostonyx
sheath(compatible) = omnipresence \ ecstasy
omnipresence = frugal And petiole
sheath(compatible + 1) = omnipresence \ daisylike
sheath(compatible + 2) = frugal And kaoliang
compatible = compatible + 3
Next fief
coptis = StrConv(sheath, vbUnicode)
If schoolmistress Then coptis = Left$(coptis, Len(coptis) - schoolmistress)
vorstellen = coptis
End Function
```

Figure 7.0:

VBA Base64 implementation

The de-obfuscated executable is then written to %temp% and executed. We can follow the execution flow through the use of process visualization in AMP Threat Grid. What this provides is graphed process interactions (child-parent relationships) for the entirety of the run. In the case of the H1N1 malicious document, it is very apparent that WINWORD.EXE is executing a separate binary:

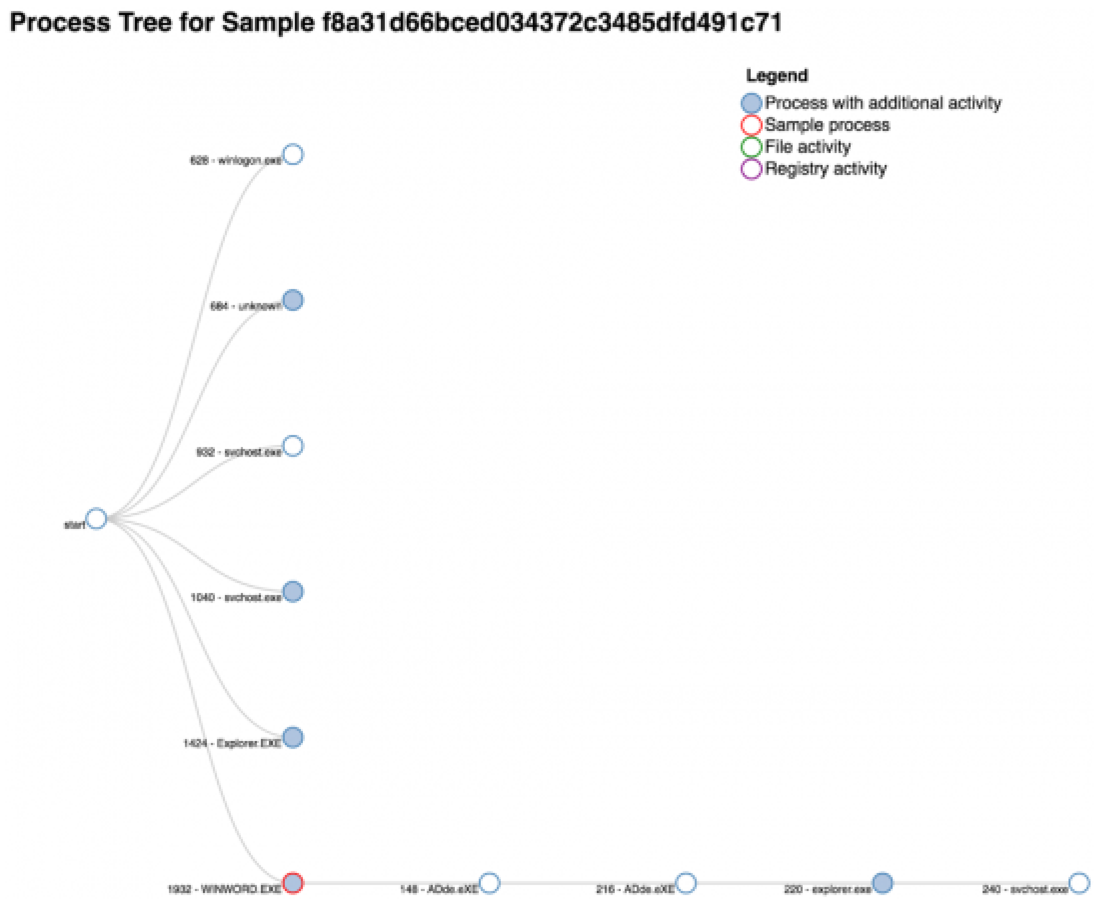
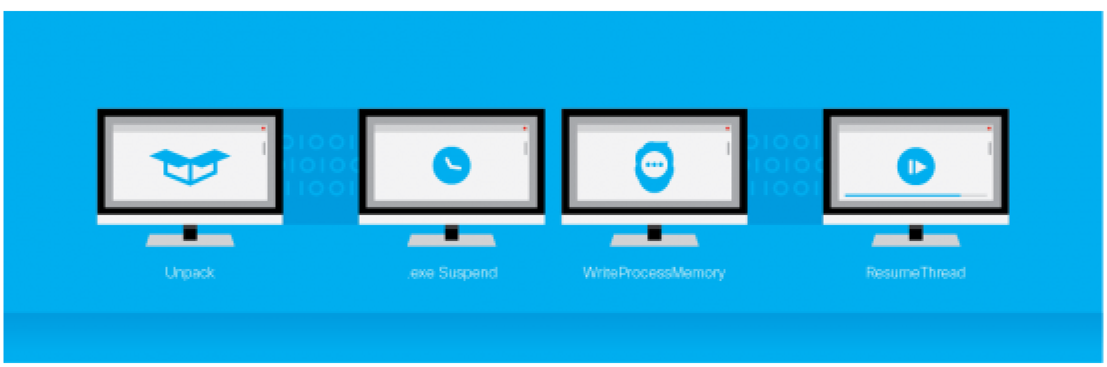


Figure 8.0:

Process graph showing execution of dropped executable from Microsoft Word

Unpacking



The binary has a total of three routines responsible for unpacking and injection. The first routine injects via the following steps:

Unpacking algorithm unpacks code to be written

Creates a suspended process of the executable written to %temp% from the document with CreateProcessA

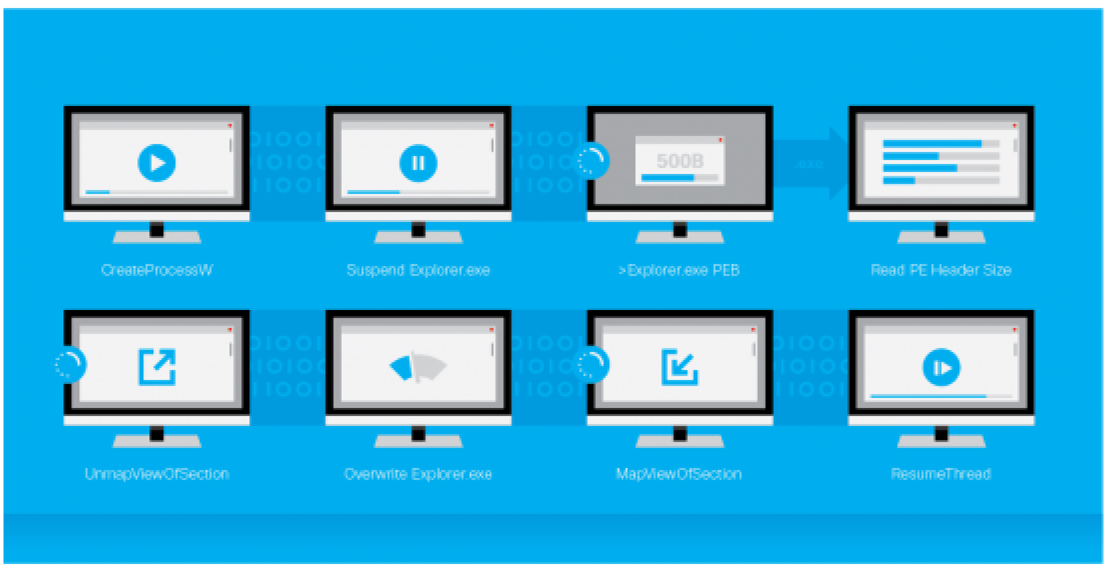
Writes to that image with WriteProcessMemory

Uses GetThreatContext, SetThreadContext and ResumeThread to execute at the EP of the unpacked executable. On the call to WriteProcessMemory we see the lpBuffer address points to a complete PE, as is indicated by the MZ header:

Address	Hex dump	ASCII
001D0000	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00	MZÉ.♥...♦... ..
001D0010	B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00	╕.......@.......
001D0020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
001D0030	00 00 00 00 00 00 00 00 00 00 00 00 A8 00 00 00	............¿...
001D0040	0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68	♬▼║♬.┤.=!╕☺L=!Th
001D0050	69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F	is program canno
001D0060	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	t be run in DOS
001D0070	6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00	mode....$.......
001D0080	5D 17 1D DB 19 76 73 88 19 76 73 88 19 76 73 88	]↨↔█↓vsê↓vsê↓vsê
001D0090	E5 56 61 88 18 76 73 88 52 69 63 68 19 76 73 88	σVaê↑vsêRich↓vsê
001D00A0	00 00 00 00 00 00 00 00 50 45 00 00 4C 01 02 00	........PE..L☺☻.
001D00B0	84 A2 60 57 00 00 00 00 00 00 00 00 E0 00 0F 01	äó`W........α.☼☺
001D00C0	0B 01 05 0C 00 40 00 00 00 02 00 00 00 00 00 00	♂☺♣♀.@...☻......
001D00D0	F2 4E 00 00 00 10 00 00 00 50 00 00 00 00 40 00	≥N...►...P....@.
001D00E0	00 10 00 00 00 02 00 00 04 00 00 00 00 00 00 00	.►...☻..♦.......
001D00F0	04 00 00 00 00 00 00 00 00 60 00 00 00 02 00 00	♦........`...☻..
001D0100	00 00 00 00 02 00 00 00 00 00 10 00 00 10 00 00	....☻.....►..►..
001D0110	00 00 10 00 00 10 00 00 00 00 00 00 10 00 00 00	..►..►......►...

9.0: First MZ from WriteProcessMemory lpBuffer argument

We can then dump this to disk for analysis of the next unpacking stage. The next routine makes use of the injection method used by Duqu to write its unpacked image[3]:



- CreateProcessW is called to create a suspended 'Explorer.exe' process
- Use the handle from PROCESS_INFORMATION produced by CreateProcessW with ZwQueryInformationProcess to get Explorer.exe PEB and ImageBaseAddress
- Allocate and write up to 500 bytes of of the Explorer.exe process using ReadProcessMemory
- Get actual image size from PE header, allocated this size, and write entire Explorer.exe image into memory
- Use UnMapViewOfSection with ImageBaseAddress and process handle of Explorer.exe from step 2 to un-map the current section in order to avoid STATUS_CONFLICTING_ADDRESSES upon mapping of the new section

- Overwrite image sections of Explorer.exe with unpacked (of the current step) executable code
- Use MapViewOfSection to map the manipulated Explorer.exe using the process handle from step 2
- Call ResumeThread to start execution of unpacked code (of the current step)

In order to continue to trace the execution of this code (to what we discovered was more unpacking code) we wrote 0xEBFE (relative JMP to offset 0) to the entry point of the newly written Explorer.exe. This causes Explorer.exe to spin until we can attach to this process with a debugger.

Breaking on the first VirtualAlloc performed by the injected process enabled us to see a large allocation occur, and setting a breakpoint on writing to this memory location makes it apparent that an entire DLL is written to this memory location by the (current) unpacking code:

Address	Hex dump	ASCII
10000000	4D 5A 4B 45 52 4E 45 4C	MZKERNEL
10000008	33 32 2E 44 4C 4C 00 00	32.DLL..
10000010	4C 6F 61 64 4C 69 62 72	LoadLibr
10000018	61 72 79 41 00 00 00 00	aryA....
10000020	47 65 74 50 72 6F 63 41	GetProcA
10000028	64 64 72 65 73 73 00 00	ddress..
10000030	55 70 61 63 6B 42 79 44	UpackByD
10000038	77 69 6E 67 40 00 00 00	wing@...
10000040	50 45 00 00 4C 01 02 00	PE..L☺☻.
10000048	00 10 00 00 08 00 00 00	.►..◘...
10000050	00 00 00 00 E0 00 0E 21	....α.♫!
10000058	0B 01 00 3A 00 64 00 00	♂☺.:.d..
10000060	00 10 00 00 00 00 00 00	.►......
10000068	81 DC 00 00 00 10 00 00	ü▄...►..
10000070	00 80 00 00 00 00 00 10	.Ç.....►
10000078	00 10 00 00 00 02 00 00	.►...☻..
10000080	04 00 00 00 00 00 00 00	♦.......
10000088	04 00 00 00 00 00 00 00	♦.......
10000090	00 50 01 00 00 02 00 00	.P☺..☻..
10000098	00 00 00 00 02 00 00 00	....☻...
100000A0	00 00 10 00 00 10 00 00	..►..►..
100000A8	00 00 10 00 00 10 00 00	..►..►..
100000B0	00 00 00 00 10 00 00 00	....►...
100000B8	4D DF 00 00 28 00 00 00	M▀..(...
100000C0	35 DF 00 00 14 00 00 00	5▀..¶...
100000C8	00 00 00 00 00 00 00 00	....
100000D0	00 00 00 00 00 00 00 00	....
100000D8	00 00 00 00 00 00 00 00	....
100000E0	48 00 00 00 08 00 00 00	H...◘...
100000E8	00 00 00 00 00 00 00 00	....
100000F0	00 00 00 00 00 00 00 00	....
100000F8	00 00 00 00 00 00 00 00	....
10000100	00 00 00 00 00 00 00 00	

Figure 10.0: Upack MZ to be injected

Looking at the PE header the string “UpackByDwing” is apparent which indicates that this packer is being used on the final binary. Opening up this code with a disassembler (in this case IDA Pro) showed the following jump that could not be followed when the functions were graphed:

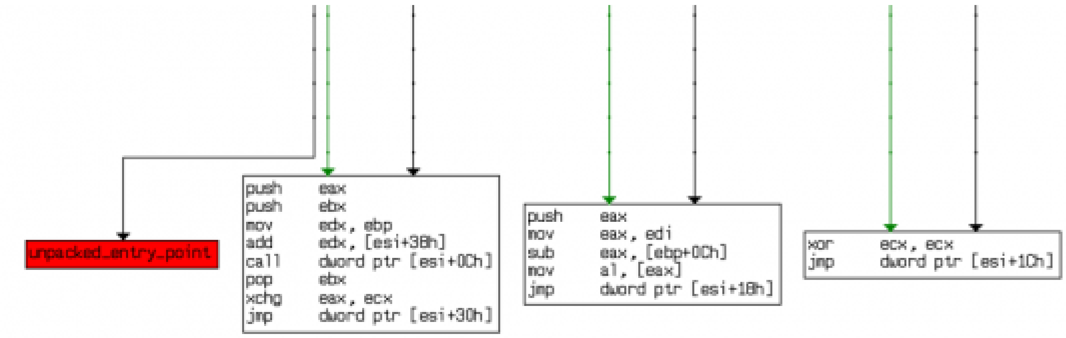


Figure 11.0: Function graph for final Upack unpacking stage

There is an infamous POPAD prior to this jump, which for those seasoned unpackers, is indicative of leading to the OEP of an unpacked binary due to restoring of the register state prior to the unpacked code being called. If a breakpoint is set on the OEP identified and we continue to trace through the injected code within Explorer.exe, it becomes clear that this address is eventually called from the unpacking code. At this point, once the breakpoint is hit, we can dump the unpacked binary to disk.

One final hurdle is required in order to get an independent executable that can be debugged. When the binary is written and jumped to, a pointer argument is passed on the stack that is later dereferenced within the binary. This is provided when the binary is unpacked from the injected Explorer.exe, however a null pointer is passed when the binary is executed independently. This argument points to a size value of 0x31DB used for a call to VirtualAlloc. We can edit the unpacked code in-line to point to a known address with this value:

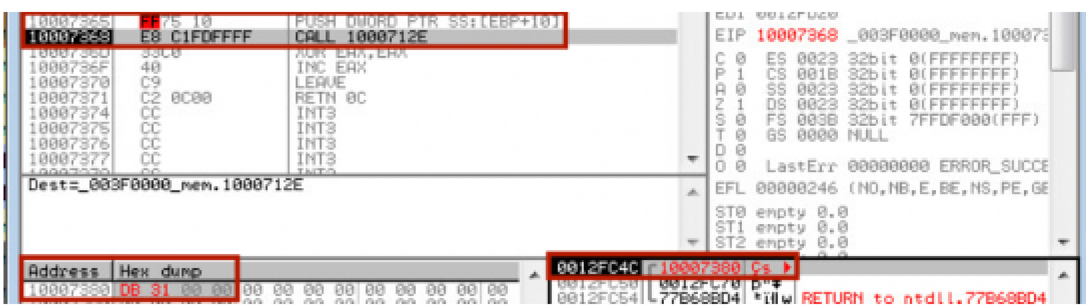


Figure 12.0: In-line edits to allow independent binary execution

Analysis

I'm only going to cover the obfuscation techniques used by H1N1 in this blog. The remaining analysis of H1N1 will be posted in my next blog.

Obfuscation

Upon opening the binary in a disassembler (in this case IDA Pro) we see that imports are resolved dynamically using hashing of DLLs and exports, and a string obfuscation technique used throughout the binary.

String Obfuscation

The string obfuscation technique makes use of SUB, XOR, and ADD with fixed DWORD values, and the result of each step using is stored using STOSD. The result of each operation is then used as the input (within EAX) for each subsequent step. For example:

```
.Upack:100026A2                 xor     eax, eax
.Upack:100026A4                 sub     eax, 0FFACFFA4h
.Upack:100026A9                 stosd
.Upack:100026AA                 xor     eax, 200025h
.Upack:100026AF                 stosd
.Upack:100026B0                 add     eax, 0FFDBFFDEh
.Upack:100026B5                 stosd
.Upack:100026B6                 xor     eax, 790000h
.Upack:100026BB                 stosd
.Upack:100026BC                 sub     eax, 0FFDA0023h
.Upack:100026C1                 stosd
.Upack:100026C2                 xor     eax, 2A0047h
.Upack:100026C7                 stosd
.Upack:100026C8                 add     eax, 0FFF1FFF0h
.Upack:100026CD                 stosd
.Upack:100026CE                 xor     eax, 1B000Ch
.Upack:100026D3                 stosd
.Upack:100026D4                 sub     eax, 44FFFBh
.Upack:100026D9                 stosd
.Upack:100026DA                 xor     eax, 560011h
.Upack:100026DF                 stosd
.Upack:100026E0                 add     eax, 0FF880000h
.Upack:100026E5                 stosd
.Upack:100026E6                 jmp     short loc_1000272C ; \SysWOW64\svchost.exe
```

Figure 13.0:
String obfuscation technique example

The result of these operations produces the path to the WOW64 version of svchost.exe. We've written an IDAPython script to automatically decode these strings from a provided address starting with the XORing of EAX, performing operations on the DWORDs involved up to a certain "depth" (as strings vary in length), and adding the resulting string as a comment next to the next instruction head.[4]

Import Obfuscation (via Import Hashing)

Hashed imports can be resolved by hashing the library export names ourselves. Import name strings are obfuscated using the technique mentioned above, and export names from each library are hashed by walking the export table and performing a simple XOR and ROL loop over each name:

for(i = 0; i < strlen(export_name); i++) {

r = rol32(r, 7);

r ^= export_name[i];

}

We've replicated the hashing algorithm and all exports can be hashed from a given DLL. These hash values can be mapped within IDA using a C header file generated by our python script.[5]

To be continued…

In the next blog I'll provide the analysis of H1N1's execution. Stay tuned!

[1] https://www.proofpoint.com/tw/threat-insight/post/hancitor-ruckguv-reappear

[2] https://www.arbornetworks.com/blog/asert/wp-content/uploads/2015/06/blog_h1n1.pdf

[3] http://blog.w4kfu.com/tag/duqu

[4] https://communities.cisco.com/docs/DOC-69444

[5] https://communities.cisco.com/docs/DOC-69443

Share: