Pythons and Unicorns and Hancitor…Oh My! Decoding Binaries Through Emulation

researchcenter.paloaltonetworks.com/2016/08/unit42-pythons-and-unicorns-and-hancitoroh-my-decoding-binaries-through-emulation/

Jeff White August 30, 2016

By Jeff White

August 30, 2016 at 1:20 PM

Category: Malware, Unit 42

Tags: hancitor, LuminosityLink, Microsoft Word, Python, shellcode, Unicorn Engine, VB Dropper, WildFire, XOR key

This post is also available in: 日本語 (Japanese)

This blog post is a continuation of my previous post, VB Dropper and Shellcode for Hancitor Reveal New Techniques Behind Uptick, where we analyzed a new Visual Basic (VB) macro dropper and the accompanying shellcode. In the last post, we left off with having successfully identified where the shellcode carved out and decoded a binary from the Microsoft Word document.

Often when analysts are faced with an embedded payload for which they want to write a decoder, they simply re-write the assembly algorithm in their language of choice and process the file. The complexity of these algorithms varies when attempting to translate from machine code to a higher-level language. It can be quite frustrating at times, depending on the amount of coffee you've had and complexity of the algorithms.

In this post, I'll show how we can use an attacker's own decoding algorithm combined with CPU emulation to decode or decrypt payloads fairly easily by simply reusing the assembly in front of us. Specifically, I'll be focusing on using the Unicorn Engine module in Python to run the attacker's decoding functions within an emulated environment to extract our encoded payloads. Our end goal is to identify the command and control (C2) servers being used by the final Hancitor payload by running our Python script against the Microsoft Word document.

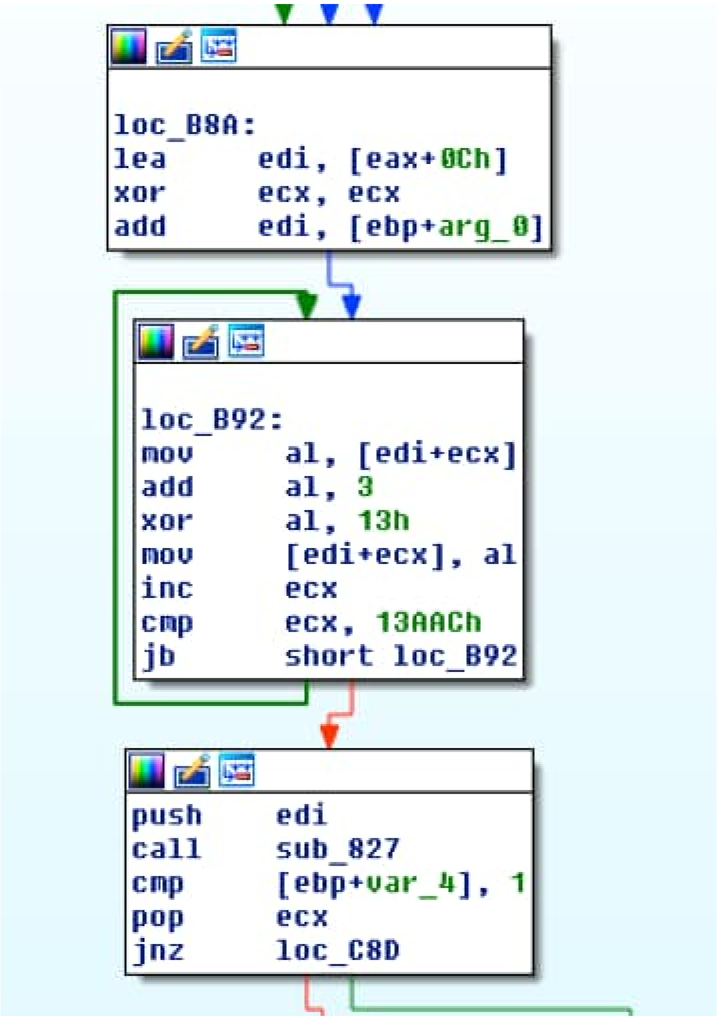
Now, you may ask, why even worry about this to begin with? In the last post we just let the program run and the payload was written to disk for easy retrieval, so why bother? The main answer to that is bulk-analysis automation. If we can write a program that we can point at a directory full of documents, then we can quickly extract embedded payloads for C2 extraction and parsing to form a more holistic view of what we're dealing with. An example of such bulk analysis was witnessed earlier this year in July when we looked at a large sample set of LuminosityLink malware samples.

Decoding Routines

As a reminder, in the last blog post we were working with the following sample:

03aef51be133425a0e5978ab2529890854ecf1b98a7cf8289c142a62de7acd1a

We'll continue where we left off after identifying the decoding routine, as seen in figure 1. The function at loc_B92 added 0x3 to each byte and uses 0x13 to XOR the result. Once every byte in the embedded binary has been processed, it pushes the location of the embedded binary to the stack and calls function sub_827.



```
loc_B8A:
lea     edi, [eax+0Ch]
xor     ecx, ecx
add     edi, [ebp+arg_0]

loc_B92:
mov     al, [edi+ecx]
add     al, 3
xor     al, 13h
mov     [edi+ecx], al
inc     ecx
cmp     ecx, 13AACh
jb      short loc_B92

push    edi
call    sub_827
cmp     [ebp+var_4], 1
pop     ecx
jnz     loc_C8D
```

Figure 1 Start of decoding routine

Without going too far into detail on the decoding routine, know that there are five parts to it, and that each one manipulates the bytes in some way before the overall function ends and our payload is decoded.

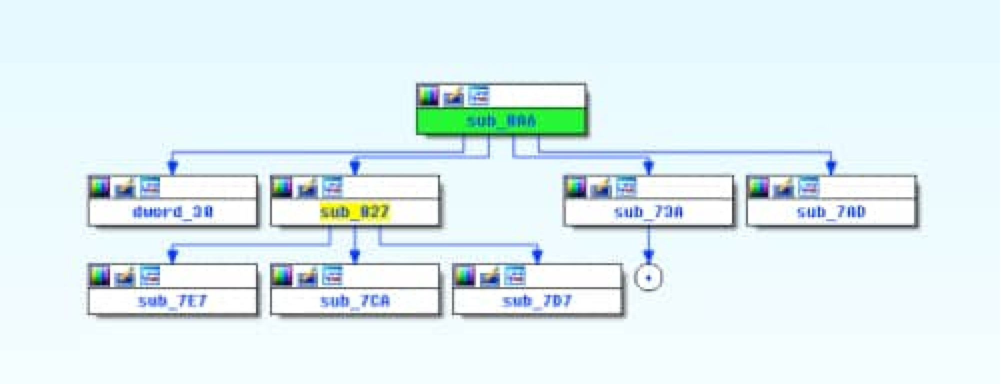
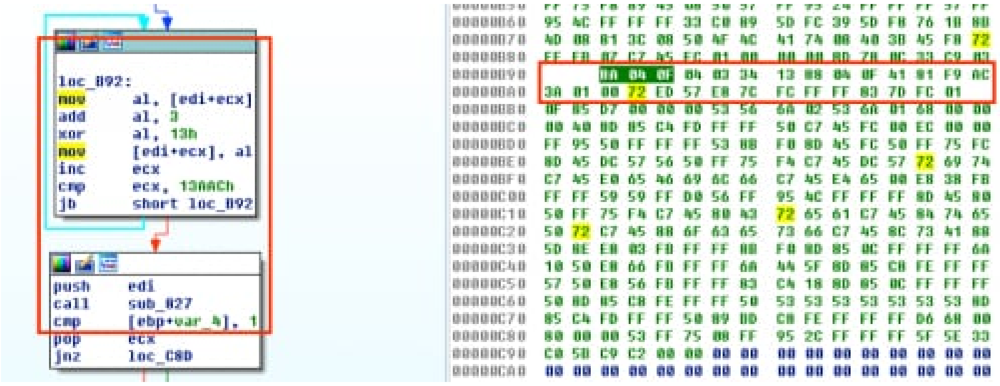


Figure 2 Proximity view of decoding functions in IDA

What we’re effectively going to do is copy the bytes from sub_8A6, sub_827, sub_7E7, sub_7CA, and sub_7D7. These are the core functions that handle all of the decoding. In addition to this, we’ll need our embedded payload, which can be located in the Word document through the magic header of “POLA” as discussed in the previous blog.

Once we have the copied bytes, we’ll setup our emulation environment, adjust our assembly, and run our own shellcode to retrieve the payload. In the context of this blog, I’m just going to refer to the x86 instructions as shellcode to keep things straightforward.

Starting with offset 0xB92, we’ll copy the bytes for the two blocks, ending just after our call since the payload will be decoded by that point.

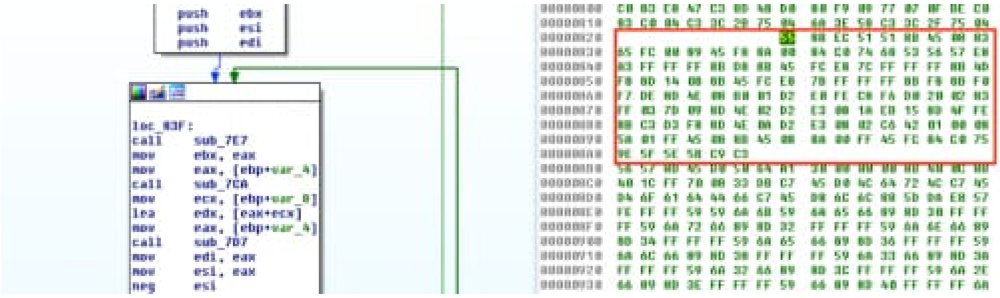


```
loc_B92:
mov     al, [edi+ecx]
add     al, 3
xor     al, 13h
mov     [edi+ecx], al
inc     ecx
cmp     ecx, 13AACh
jb      short loc_B92

push    edi
call    sub_827
cmp     [ebp+var_4], 1
pop     ecx
jnz     loc_C8D
```

Figure 3 Decoding function and associated bytes

Next we’ll copy the bytes from sub_827, which are all of the bytes from offset 0x827 to 0x8A5.



```
push    ebx
push    esi
push    edi

loc_83F:
call    sub_7E7
mov     ebx, eax
mov     eax, [ebp+var_4]
call    sub_7CA
mov     ecx, [ebp+var_8]
lea     edx, [eax+ecx]
mov     eax, [ebp+var_4]
call    sub_7D7
mov     edi, eax
mov     esi, eax
neg     esi
```

Figure 4 Additional decoding functions and associated bytes

Last, we’ll collect the bytes from the three smaller functions. If you note their location, you can see they are contiguous. Keeping the bytes in order is convenient but not necessary. If they don’t line up, you’ll simply need to adjust the operands for the calls or jumps so that they go where they should.

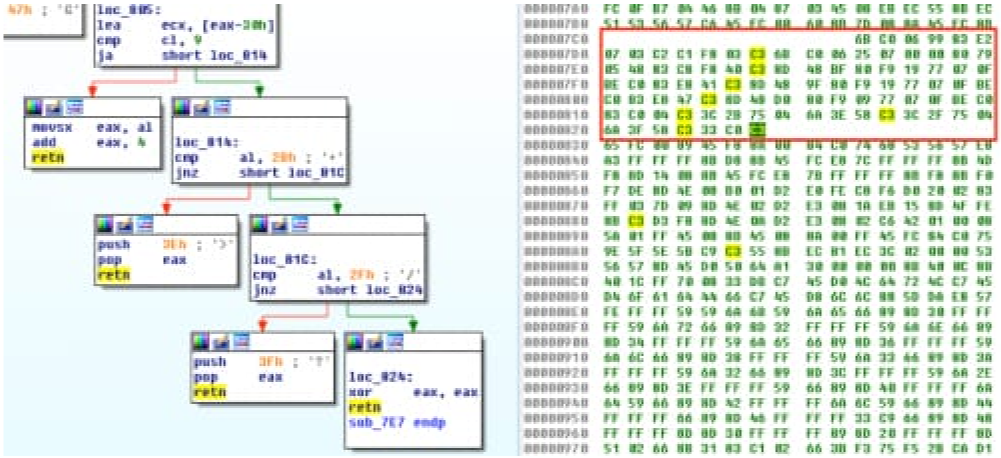
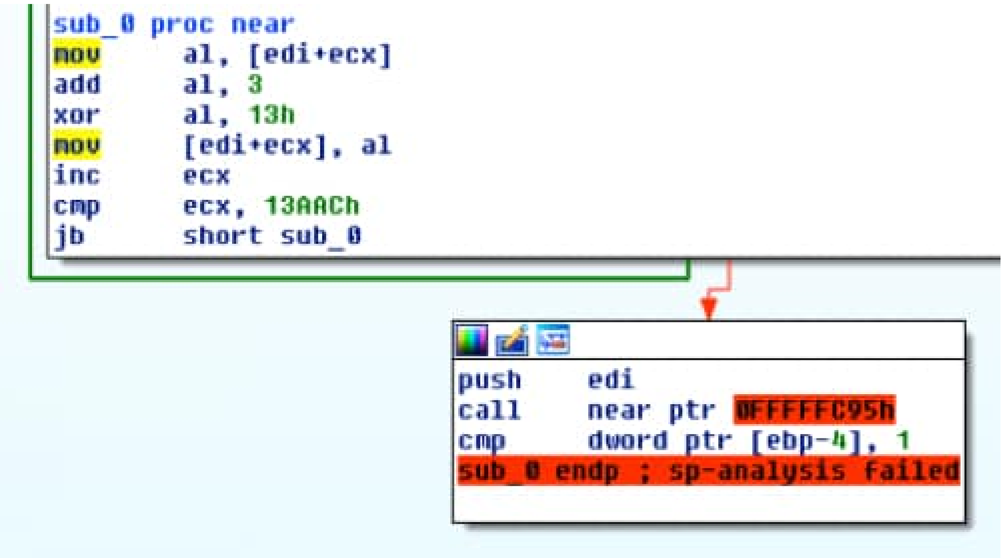


Figure 5 Additional decoding functions and associated bytes

Once all of the bytes have been saved, we can write them to a file and open it up in a disassembler to see what issues we need to correct, if any.

```
# sub_8A6
sc = b'\x8A\x04\x0F\x04\x03\x34\x13\x88\x04\x0F\x41\x81\xF9\xAC\x3A\x01\x
00\x72\xED\x57\xE8\x7C\xFC\xFF\xFF\x83\x7D\xFC\x01'
# sub_7CA
sc += b'\x6B\xC0\x06\x99\x83\xE2\x07\x03\xC2\xC1\xF8\x03\xC3'
# sub_7D7
sc +=
b'\x6B\xC0\x06\x25\x07\x00\x00\x80\x79\x05\x48\x83\xC8\xF8\x40\xC3'
<em># sub_7E7</em>
sc +=
b'\x8D\x48\xBF\x80\xF9\x19\x77\x07\x0F\xBE\xC0\x83\xE8\x41\xC3\x8D\x
48\x9F\x80\xF9\x19\x77\x07\x0F\xBE\xC0\x83\xE8\x47\xC3\x8D\x48\xD0\x
80xF9\x09\x77\x07\x0F\xBE\xC0\x83\xC0\x04\xC3\x3C\x2B\x75\x04\x6A\x
3E\x58\xC3\x3C\x2F\x75\x04\x6A\x3F\x58\xC3\x33\xC0\xC3'
# sub_827
sc +=
b'\x55\x8B\xEC\x51\x51\x8B\x45\x08\x83\x65\xFC\x00\x89\x45\xF8\x8A\x
00\x84\xC0\x74\x68\x53\x56\x57\xE8\xA3\xFF\xFF\xFF\x8B\xD8\x8B\x45\x
FC\xE8\x7C\xFF\xFF\xFF\x8B\x4D\xF8\x8D\x14\x08\x8B\x45\xFC\xE8\x7B\x
FF\xFF\xFF\x8B\xF8\x8B\xF0\xF7\xDE\x8D\x4E\x08\xB0\x01\xD2\xE0\xFE\x
C8\xF6\xD0\x20\x02\x83\xFF\x03\x7D\x09\x8D\x4E\x02\xD2\xE3\x08\x1A\x
EB\x15\x8D\x4F\xFE\x8B\xC3\xD3\xF8\x8D\x4E\x0A\xD2\xE3\x08\x02\xC6\x
42\x01\x00\x08\x5A\x01\xFF\x45\x08\x8B\x45\x08\x8A\x00\xFF\x45\xFC\x
84\xC0\x75\x9E\x5F\x5E\x5B\xC9\xC3'
```

Looking at our shellcode, only one major issue appears, which is the initial call to the decoding function being at a different address.

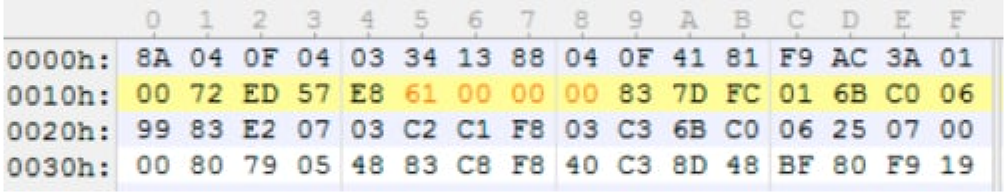


```
sub_0 proc near
mov     al, [edi+ecx]
add     al, 3
xor     al, 13h
mov     [edi+ecx], al
inc     ecx
cmp     ecx, 13AACh
jb      short sub_0

push    edi
call    near ptr 0FFFFFC95h
cmp     dword ptr [ebp-4], 1
sub_0 endp ; sp-analysis failed
```

Figure 6 Broken call within shellcode

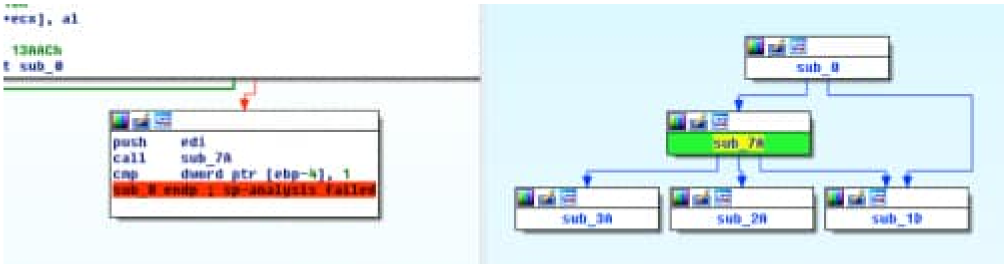
As we want to call to our previous function sub_827, which is at the end of our shellcode, we can adjust this call to point to the start of that function. Looking at our code in a hex editor, the start of the function is exactly 97 bytes (0x61) into our shellcode, so we can change the instruction 0xE87CFCFFFF to 0xE861000000.



```
        0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
0000h: 8A 04 0F 04 03 34 13 88 04 0F 41 81 F9 AC 3A 01
0010h: 00 72 ED 57 E8 61 00 00 00 83 7D FC 01 6B C0 06
0020h: 99 83 E2 07 03 C2 C1 F8 03 C3 6B C0 06 25 07 00
0030h: 00 80 79 05 48 83 C8 F8 40 C3 8D 48 BF 80 F9 19
```

Figure 7 Correcting the previously broken call

Next, we can validate our change worked as expected within the disassembler and that our functions are now all correctly linked.



```
+ecx], al

 13AACh
t sub_0

push    edi
call    sub_7A
cmp     dword ptr [ebp-4], 1
sub_0 endp ; sp-analysis failed
```

Figure 8 Validating correction of call

Embedded Payload

We know that our embedded payload address is located on the EDI register that gets pushed onto the stack through our previous dynamic analysis. For the initial validation of this method, we'll go ahead and manually copy the bytes, starting with the magic header of "POLA" and a size of 0x13AAAC bytes, to our Python script. At the end of the blog, I'll include a full script that will automatically extract this binary from the Word Document.

```
# POLA 0x504F4C41
encoded_binary =
b'\x50\x4F\x4C\x41\x08\x00\xFF\xFF\xAC\x3A\x01[truncated]'
```

Enter the Unicorn

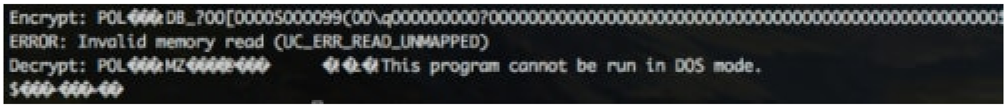
As we now have all of the data we need to decode the binary, the last step for this part is to build the emulation environment for our code to run on. To accomplish this, I'll use the open-source Unicorn Engine.

The first thing we'll want to do is assign the address space we'll be working within, along with initializing Unicorn for the architecture we want to emulate (x86), and map some memory to use. Next we'll write our shellcode and encoded binary to our memory space and initialize some values. Finally, we'll output the decrypted data to STDOUT.

```
ADDRESS = 0x1000000
mu = Uc(UC_ARCH_X86, UC_MODE_32)
mu.mem_map(ADDRESS, 4 * 1024 * 1024)

# Write code to memory
mu.mem_write(ADDRESS, X86_CODE32)
# Start of encoded data + offset to binary, pushed to Stack at
start
mu.reg_write(UC_X86_REG_EDI, 0x10000F9 + 0x0C)
# Initialize ECX counter to 0
mu.reg_write(UC_X86_REG_ECX, 0x0)
# Initialize Stack for functions
mu.reg_write(UC_X86_REG_ESP, 0x1300000)

print "Encrypt: %s" % mu.mem_read(0x10000F9,150)
mu.emu_start(ADDRESS, ADDRESS + len(X86_CODE32))
print "Decrypt: %s" % mu.mem_read(0x10000F9,150)
```

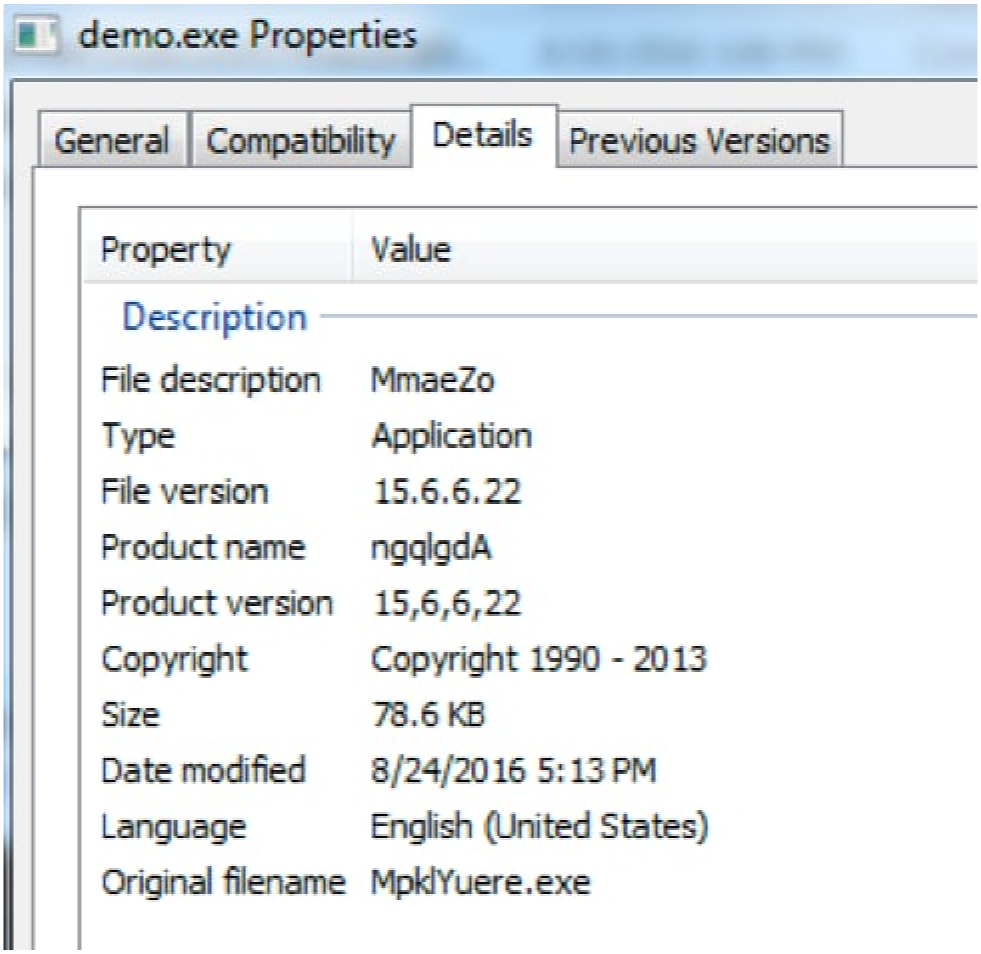


```
Encrypt: POL���DB_?00[00005000099(00\q000000000?000000000000000000000000000000000000000000000000000000
ERROR: Invalid memory read (UC_ERR_READ_UNMAPPED)
Decrypt: POL���MZ�������      ���This program cannot be run in DOS mode.
$������
```

Figure 9 Successful decoding

Success! We can write that section of memory to a file and see what we have.

```
f = open("demo.exe", "w")
f.write(mu.mem_read(0x10000F9 + 0x0C, 0x13AAC))
f.close()
```



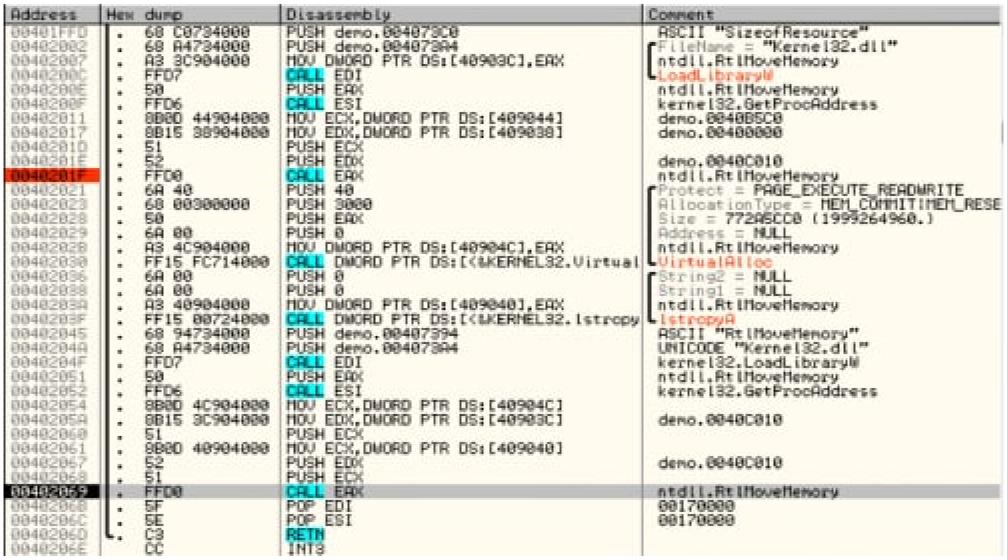
Property	Value
File description	MmaeZo
Type	Application
File version	15.6.6.22
Product name	ngqlgdA
Product version	15,6,6,22
Copyright	Copyright 1990 - 2013
Size	78.6 KB
Date modified	8/24/2016 5:13 PM
Language	English (United States)
Original filename	MpklYuere.exe

Figure 10 Decoded binary properties

Unfortunately we find ourselves with a packed binary that may have our actual Hancitor sample, so we'll need to try and decode yet another payload.

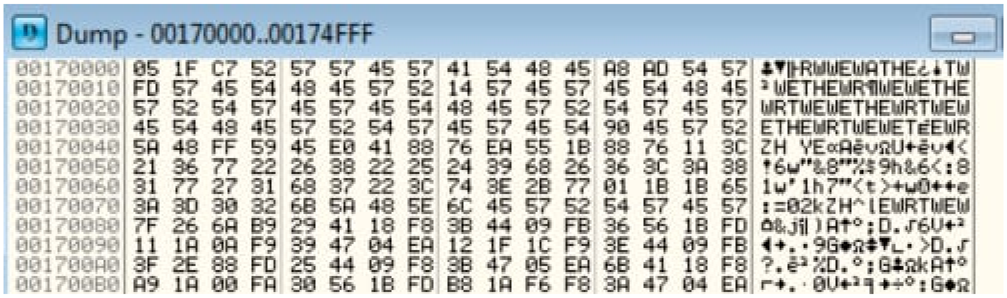
Attack of the Binaries

This binary has a fair amount of functions and code, but very early on we see the binary lookup the address for the same API we discussed in our earlier blog post, RtlMoveMemory(), and then copy what we presume is our encoded payload.



```
Address   Hex dump           Disassembly                              Comment
00401FFD  .  68 C0734000     PUSH demo.004073C0                       ASCII "SizeofResource"
00402002  .  68 A4734000     PUSH demo.004073A4                       FileName = "Kernel32.dll"
00402007  .  A3 3C904000     MOV DWORD PTR DS:[40903C],EAX            ntdll.RtlMoveMemory
0040200C  .  FFD7            CALL EDI                                 LoadLibraryW
0040200E  .  50              PUSH EAX                                 ntdll.RtlMoveMemory
0040200F  .  FFD6            CALL ESI                                 kernel32.GetProcAddress
00402011  .  8B0D 44904000   MOV ECX,DWORD PTR DS:[409044]            demo.0040B5C0
00402017  .  8B15 38904000   MOV EDX,DWORD PTR DS:[409038]            demo.00400000
0040201D  .  51              PUSH ECX
0040201E  .  52              PUSH EDX                                 demo.0040C010
0040201F  .  FFD0            CALL EAX                                 ntdll.RtlMoveMemory
00402021  .  6A 40           PUSH 40                                  Protect = PAGE_EXECUTE_READWRITE
00402023  .  68 00300000     PUSH 3000                                AllocationType = MEM_COMMIT|MEM_RESE
00402028  .  50              PUSH EAX                                 Size = 772A5CC0 (1999264960.)
00402029  .  6A 00           PUSH 0                                   Address = NULL
0040202B  .  A3 4C904000     MOV DWORD PTR DS:[40904C],EAX            ntdll.RtlMoveMemory
00402030  .  FF15 FC714000   CALL DWORD PTR DS:[<&KERNEL32.Virtual    VirtualAlloc
00402036  .  6A 00           PUSH 0                                   String2 = NULL
00402038  .  6A 00           PUSH 0                                   String1 = NULL
0040203A  .  A3 40904000     MOV DWORD PTR DS:[409040],EAX            ntdll.RtlMoveMemory
0040203F  .  FF15 00724000   CALL DWORD PTR DS:[<&KERNEL32.lstrcpy    lstrcpyA
00402045  .  68 94734000     PUSH demo.00407394                       ASCII "RtlMoveMemory"
0040204A  .  68 A4734000     PUSH demo.004073A4                       UNICODE "Kernel32.dll"
0040204F  .  FFD7            CALL EDI                                 kernel32.LoadLibraryW
00402051  .  50              PUSH EAX                                 ntdll.RtlMoveMemory
00402052  .  FFD6            CALL ESI                                 kernel32.GetProcAddress
00402054  .  8B0D 4C904000   MOV ECX,DWORD PTR DS:[40904C]
0040205A  .  8B15 3C904000   MOV EDX,DWORD PTR DS:[40903C]            demo.0040C010
00402060  .  51              PUSH ECX
00402061  .  8B0D 40904000   MOV ECX,DWORD PTR DS:[409040]
00402067  .  52              PUSH EDX                                 demo.0040C010
00402068  .  51              PUSH ECX
00402069  .  FFD0            CALL EAX                                 ntdll.RtlMoveMemory
0040206B  .  5F              POP EDI                                  00170000
0040206C  .  5E              POP ESI                                  00170000
0040206D  .  C3              RETN
0040206E     CC              INT3
```

Figure 11 RtlMoveMemory() being called

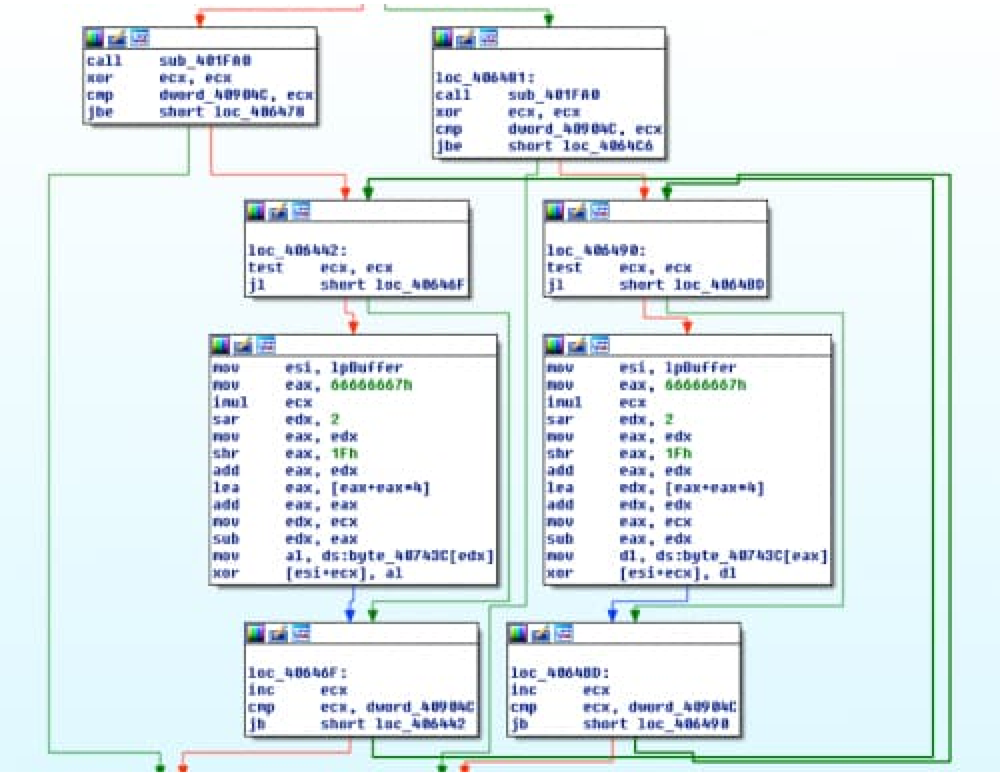


Dump - 00170000..00174FFF

```
00170000  05 1F C7 52 57 57 45 57 41 54 48 45 A8 AD 54 57  ♣▼╟RWWEWATHE¿¡TW
00170010  FD 57 45 54 48 45 57 52 14 57 45 57 45 54 48 45  ²WETHEWR¶WEWETHE
00170020  57 52 54 57 45 57 45 54 48 45 57 52 54 57 45 57  WRTWEWETHEWRTWEW
00170030  45 54 48 45 57 52 54 57 45 57 45 54 90 45 57 52  ETHEWRTWEWETÉEWR
00170040  5A 48 FF 59 45 E0 41 88 76 EA 55 1B 88 76 11 3C  ZH YEαAêvΩU←êv◄<
00170050  21 36 77 22 26 38 22 25 24 39 68 26 36 3C 3A 38  !6w"&8"%$9h&6<:8
00170060  31 77 27 31 68 37 22 3C 74 3E 2B 77 01 1B 1B 65  1w'1h7"<t>+w☺←←e
00170070  3A 3D 30 32 6B 5A 48 5E 6C 45 57 52 54 57 45 57  :=02kZH^lEWRTWEW
00170080  7F 26 6A B9 29 41 18 F8 3B 44 09 FB 36 56 1B FD  ⌂&j╣)A↑°;D.√6V←²
00170090  11 1A 0A F9 39 47 04 EA 12 1F 1C F9 3E 44 09 FB  ◄→.·9G♦Ω↕▼∟·>D.√
001700A0  3F 2E 88 FD 25 44 09 F8 3B 47 05 EA 6B 41 18 F8  ?.ê²%D.°;G♣ΩkA↑°
001700B0  A9 1A 00 FA 30 56 1B FD B8 1A F6 F8 3A 47 04 EA  ⌐→.·0V←²╕→÷°:G♦Ω
```

Figure 12 Encoded payload

Continuing to debug the program, just three instructions later it returns to what looks like our next decoding routine.



```
call    sub_401FA0
xor     ecx, ecx
cmp     dword_40904C, ecx
jbe     short loc_406478
```

```
loc_406481:
call    sub_401FA0
xor     ecx, ecx
cmp     dword_40904C, ecx
jbe     short loc_4064C6
```

```
loc_406442:
test    ecx, ecx
jl      short loc_40646F
```

```
loc_406490:
test    ecx, ecx
jl      short loc_4064BD
```

```
mov     esi, lpBuffer
mov     eax, 66666667h
imul    ecx
sar     edx, 2
mov     eax, edx
shr     eax, 1Fh
add     eax, edx
lea     eax, [eax+eax*4]
add     eax, eax
mov     edx, ecx
sub     edx, eax
mov     al, ds:byte_40743C[edx]
xor     [esi+ecx], al
```

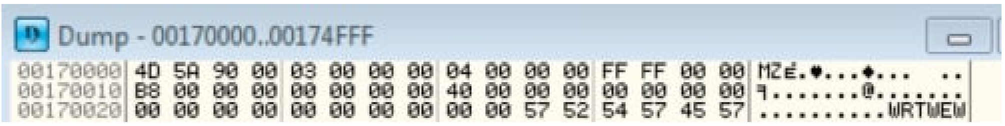
```
mov     esi, lpBuffer
mov     eax, 66666667h
imul    ecx
sar     edx, 2
mov     eax, edx
shr     eax, 1Fh
add     eax, edx
lea     edx, [eax+eax*4]
add     edx, edx
mov     eax, ecx
sub     eax, edx
mov     dl, ds:byte_40743C[eax]
xor     [esi+ecx], dl
```

```
loc_40646F:
inc     ecx
cmp     ecx, dword_40904C
jb      short loc_406442
```

```
loc_4064BD:
inc     ecx
cmp     ecx, dword_40904C
jb      short loc_406490
```

Figure 13 Decoding function

Letting these blocks complete a few times validates we're in the right spot, as we quickly identify the MZ executable header.



Dump - 00170000..00174FFF

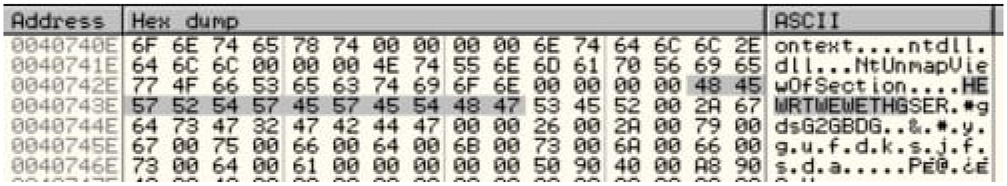
```
00170000 4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 MZÉ.♥...♦...  ..
00170010 B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 ╕.......@.......
00170020 00 00 00 00 00 00 00 00 00 00 57 52 54 57 45 57 ..........WRTWEW
```

Figure 14 Validation of decoding

We've now found the location of the encoded binary, due to RtlMoveMemory(), and the location of our function that we need to emulate.

Function Copying

Analyzing this function, it's much less complex than the last one, but takes a different approach of iterating over a 12-byte key, located at 0x40743C in our example, and using it to XOR the encoded payload.

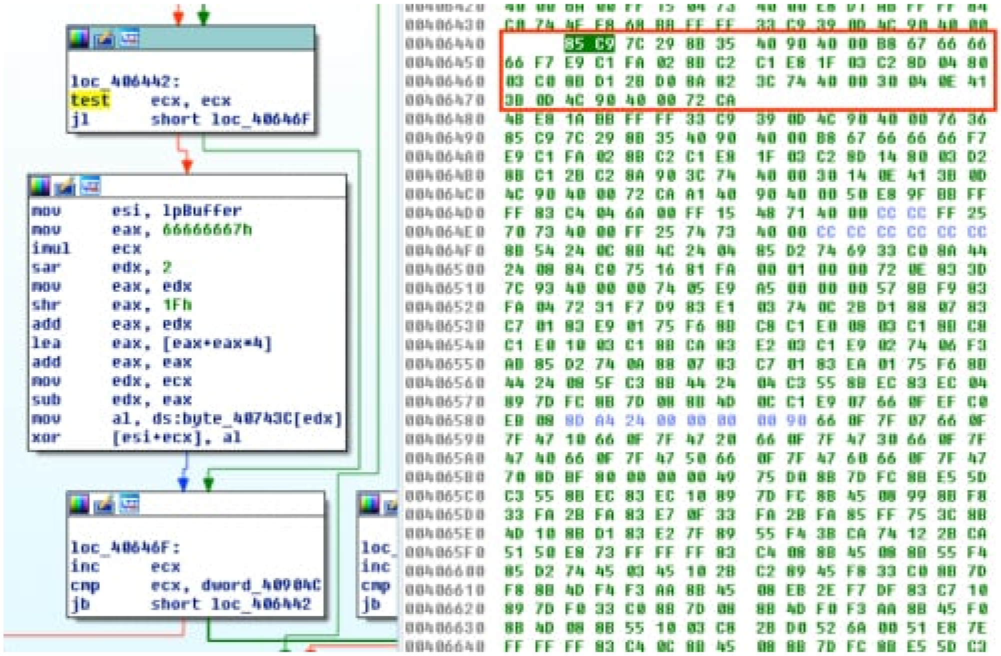


```
Address  Hex dump                                         ASCII
0040740E 6F 6E 74 65 78 74 00 00 00 00 6E 74 64 6C 6C 2E ontext....ntdll.
0040741E 64 6C 6C 00 00 00 4E 74 55 6E 6D 61 70 56 69 65 dll...NtUnmapVie
0040742E 77 4F 66 53 65 63 74 69 6F 6E 00 00 00 00 48 45 wOfSection....HE
0040743E 57 52 54 57 45 57 45 54 48 47 53 45 52 00 2A 67 WRTWEWETHGSER.*g
0040744E 64 73 47 32 47 42 44 47 00 00 26 00 2A 00 79 00 dsG2GBDG..&.*.y.
0040745E 67 00 75 00 66 00 64 00 6B 00 73 00 6A 00 66 00 g.u.f.d.k.s.j.f.
0040746E 73 00 64 00 61 00 00 00 00 00 50 90 40 00 A8 90 s.d.a.....PÉ@.¿É
```

Figure 15 12-byte XOR key

We'll follow the same methodology as previous to add it into our program.

Starting at loc_406442, we'll copy all of the bytes for the three blocks in the picture below, which is the decoding loop.



```
loc_406442:
test    ecx, ecx
jl      short loc_40646F
```

```
mov     esi, lpBuffer
mov     eax, 66666667h
imul    ecx
sar     edx, 2
mov     eax, edx
shr     eax, 1Fh
add     eax, edx
lea     eax, [eax+eax*4]
add     eax, eax
mov     edx, ecx
sub     edx, eax
mov     al, ds:byte_40743C[edx]
xor     [esi+ecx], al
```

```
loc_40646F:
inc     ecx
cmp     ecx, dword_40904C
jb      short loc_406442
```

Figure 16 Decoding loop and associated bytes

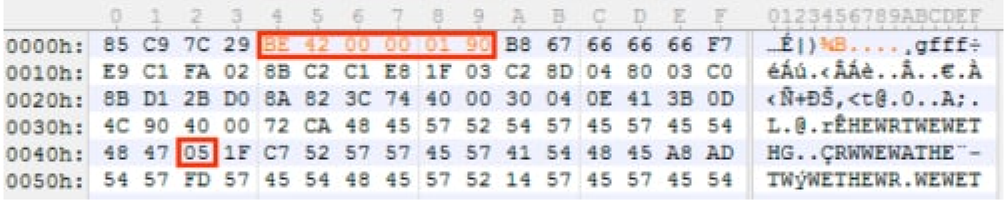
Next we'll copy the XOR key and encoded payload into our script and build a test file so that it follows the following order of operation:

shellcode -> key -> payload

```
# loc_406442
sc =
b'\x85\xC9\x7C\x29\x8B\x35\x40\x90\x40\x00\xB8\x67\x66\x66\x66\xF
7\xE9\xC1\xFA\x02\x8B\xC2\xC1\xE8\x1F\x03\xC2\x8D\x04\x80\x03\xC0
\x8B\xD1\x2B\xD0\x8A\x82\x3C\x74\x40\x00\x30\x04\x0E\x41\x3B\x0D\
x4C\x90\x40\x00\x72\xCA'
# XOR Key
sc += b'\x48\x45\x57\x52\x54\x57\x45\x57\x45\x54\x48\x47'
encoded_binary = b'\x05\x1F\xC7\x52\x57\x57\x45\x57[truncated]'
```

Looking at the code in the disassembler, we can tell there are a few values we'll have to prep before we can make this code run in our emulated environment. Specifically, we'll need to edit two MOV instructions and a CMP instruction that reference locations that don't exist in our code.

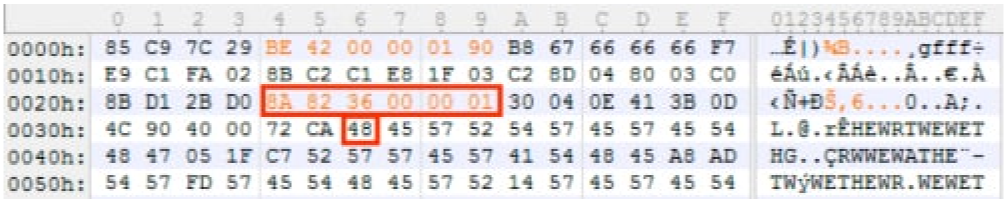
Based on our dynamic analysis, we know that the lpBuffer is a pointer to the address of the encoded payload, so we can change this instruction to move the starting location, where our payload will reside, into the ESI register. The current instruction is referencing an address in the data segment that holds the address to the payload. We'll replace it with an immediate MOV instruction by changing 0x8B3540904000 to 0xBE42000190, where 0x100042 is the start of our buffer. Since we changed the opcode, the length of our new instruction was one byte short and I padded it with a 0x90 – NOP to keep everything aligned.



```
        0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F   0123456789ABCDEF
0000h: 85 C9 7C 29 BE 42 00 00 01 90 B8 67 66 66 66 F7   …É|)¾B....¸gfff÷
0010h: E9 C1 FA 02 8B C2 C1 E8 1F 03 C2 8D 04 80 03 C0   éÁú.‹ÂÁè..Â..€.À
0020h: 8B D1 2B D0 8A 82 3C 74 40 00 30 04 0E 41 3B 0D   ‹Ñ+ÐŠ‚<t@.0..A;.
0030h: 4C 90 40 00 72 CA 48 45 57 52 54 57 45 57 45 54   L.@.rÊHEWRTWEWET
0040h: 48 47 05 1F C7 52 57 57 45 57 41 54 48 45 A8 AD   HG..ÇRWWEWATHE¨-
0050h: 54 57 FD 57 45 54 48 45 57 52 14 57 45 57 45 54   TWýWETHEWR.WEWET
```

Figure 17 Change location of payload

The first MOV is for our encoded payload, the second MOV is for our XOR key. The second MOV uses a different opcode that plays more favorably to our needs, so we'll simply change the existing address to the location of the key by modifying 0x8A823C744000 to a value of 0x8A8236000001.



```
        0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F   0123456789ABCDEF
0000h: 85 C9 7C 29 BE 42 00 00 01 90 B8 67 66 66 66 F7   …É|)¾B....¸gfff÷
0010h: E9 C1 FA 02 8B C2 C1 E8 1F 03 C2 8D 04 80 03 C0   éÁú.‹ÂÁè..Â..€.À
0020h: 8B D1 2B D0 8A 82 36 00 00 01 30 04 0E 41 3B 0D   ‹Ñ+ÐŠ‚6...0..A;.
0030h: 4C 90 40 00 72 CA 48 45 57 52 54 57 45 57 45 54   L.@.rÊHEWRTWEWET
0040h: 48 47 05 1F C7 52 57 57 45 57 41 54 48 45 A8 AD   HG..ÇRWWEWATHE¨-
0050h: 54 57 FD 57 45 54 48 45 57 52 14 57 45 57 45 54   TWýWETHEWR.WEWET
```

Figure 18 Change location of the XOR key

The final item to change is the compare instruction. Based off dynamic analysis, we know it's looking for the value 0x5000, so we'll change the opcode to support an immediate operand and modify 0x3B0D4C904000 to a value of 0x81F900500000.

```
        0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F   0123456789ABCDEF
0000h: 85 C9 7C 29 BE 42 00 00 01 90 B8 67 66 66 66 F7   …É|)¾B....¸gfff÷
0010h: E9 C1 FA 02 8B C2 C1 E8 1F 03 C2 8D 04 80 03 C0   éÁú.‹ÂÁè..Â..€.À
0020h: 8B D1 2B D0 8A 82 36 00 00 01 30 04 0E 41 81 F9   ‹Ñ+ÐŠ‚6...0..A.ù
0030h: 00 50 00 00 72 CA 48 45 57 52 54 57 45 57 45 54   .P..rÊHEWRTWEWET
0040h: 48 47 05 1F C7 52 57 57 45 57 41 54 48 45 A8 AD   HG..ÇRWWEWATHE¨-
0050h: 54 57 FD 57 45 54 48 45 57 52 14 57 45 57 45 54   TWýWETHEWR.WEWET
```

Figure 19 Hard-set compare value

Emulation

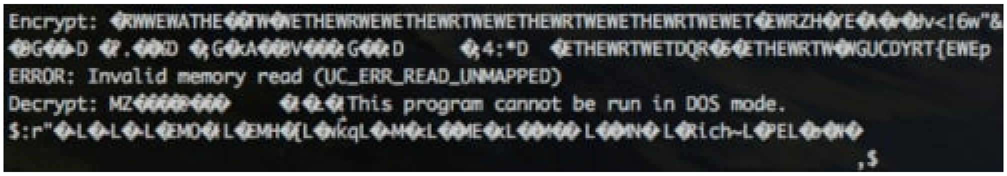
To set up our environment for this sample, the only value we need to worry about is EDX, which needs to be a pointer to our encoded payload, and gets moved into the EAX register during the loop. Similar to before, we'll setup our address space, define the architecture, map memory, and configure some initial register values.

```
ADDRESS = 0x1000000
mu = Uc(UC_ARCH_X86, UC_MODE_32)
mu.mem_map(ADDRESS, 4 * 1024 * 1024)

# Write code to memory
mu.mem_write(ADDRESS, X86_CODE32)
# Start of encoded data
mu.reg_write(UC_X86_REG_EDX, 0x1000042)
# Initialize ECX counter to 0
mu.reg_write(UC_X86_REG_ECX, 0x0)
# Initialize Stack for functions
mu.reg_write(UC_X86_REG_ESP, 0x1300000)

print "Encrypt: %s" % mu.mem_read(0x1000042,250)
mu.emu_start(ADDRESS, ADDRESS + len(X86_CODE32))
print "Decrypt: %s" % mu.mem_read(0x1000042,250)
```

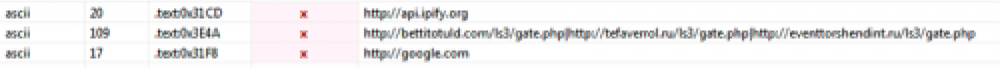
This yields the following result:



```
Encrypt: �RWWEWATHE��TW�WETHEWRWEWETHEWRTWEWETHEWRTWEWETHEWRTWEWET�EWRZH�YE�A�r�bv<!6w"&
�G��D �7.��D �G�A��V���G��D     �4:*D  �ETHEWRTWETDQR�6�ETHEWRTW�GUCDYRT{EWEp
ERROR: Invalid memory read (UC_ERR_READ_UNMAPPED)
Decrypt: MZ������     ���This program cannot be run in DOS mode.
$:r"�L�L�L�EMD�L�EMH�{L�kqL�M�L��E�L���L��N�L�Rich~L�PEL���
                                                        ,$
```

Figure 20 Decrypted payload after running Python script

If we take a look at this binary and peer at the strings, we can see that we're finally at the end of the road.



ascii	20	.text:0x31CD	×	http://api.ipify.org
ascii	109	.text:0x3E4A	×	http://bettitotuld.com/ls3/gate.php|http://tefaverrol.ru/ls3/gate.php|http://eventtorshendint.ru/ls3/gate.php
ascii	17	.text:0x31F8	×	http://google.com

Figure 21 Hancitor C2 URLs, external IP check, and Google remote check

To recap the process:

- Started with a Microsoft Word document
- Extracted base64 encoded shellcode
- Extracted encoded payload
- Emulated decoding function from shellcode to decode payload (binary)
- Extracted XOR key from new binary
- Extracted next encoded payload from new binary
- Emulated decoding function from new binary to decode Hancitor (binary)

Our last step is to put everything together into a nice package that we can use to scan thousands of Microsoft Word documents containing Hancitor and identify all of the C2 communications. Here’s a link to the Hancitor decoder script we created.

For the purpose of this test, I took a small sample set of 10,000 unique Microsoft Word documents that were first seen on August 15, 2016 and observed by Palo Alto Networks WildFire as creating a process with a name of “WinHost32.exe”. This, coupled with a few other criteria, gives me a corpus of testing samples that we know will be Hancitor and that I can run this script against.

```
[+] FILE: fe23150ffec79eb11a0fed5e3726ca6738653c4f3b0f24dd9306f6460131b34c
  #### PHASE 1 ####
  [-] ADD:  0x3
  [-] XOR:  0x13
  [-] SIZE: 80556
  [!] Success! Written to disk as
fe23150ffec79eb11a0fed5e3726ca6738653c4f3b0f24dd9306f6460131b34c_S1.exe
  #### PHASE 2 ####
  [-] XOR:  HEWRTWEWETHG
  [!] Success! Written to disk as
fe23150ffec79eb11a0fed5e3726ca6738653c4f3b0f24dd9306f6460131b34c_S2.exe
  ### PHASE 3 ###
  [-] http://api.ipify.org
  [-] http://google.com
  [-] http://bettitotuld.com/ls3/gate.php
  [-] http://tefaverrol.ru/ls3/gate.php
  [-] http://eventtorshendint.ru/ls3/gate.php
[+] FILE:
fe7d4a583c1ae380eff25a11bda4f6d53b92d49a7a4d72c775b21488453bbc96
  #### PHASE 1 ####
  [-] ADD:  0x3
  [-] XOR:  0x13
  [-] SIZE: 80556
  [!] Success! Written to disk as
fe7d4a583c1ae380eff25a11bda4f6d53b92d49a7a4d72c775b21488453bbc96_S1.exe
  #### PHASE 2 ####
  [-] XOR:  HEWRTWEWETHG
  [!] Success! Written to disk as
fe7d4a583c1ae380eff25a11bda4f6d53b92d49a7a4d72c775b21488453bbc96_S2.exe
  ### PHASE 3 ###
  [-] http://api.ipify.org
  [-] http://google.com
  [-] http://bettitotuld.com/ls3/gate.php
  [-] http://tefaverrol.ru/ls3/gate.php
  [-] http://eventtorshendint.ru/ls3/gate.php
[+] FILE: fea98cc92b142d8ec98be6134967eacf3f24d5e089b920d9abf37f372f85530d
  #### PHASE 1 ####
  [-] ADD:  0x3
  [-] XOR:  0x14
  [-] SIZE: 162992
  [!] Success! Written to disk as
fea98cc92b142d8ec98be6134967eacf3f24d5e089b920d9abf37f372f85530d_S1.exe
  #### PHASE 2 ####
  [-] XOR:  ð~ð~ð~
  [!] Detected Nullsoft Installer! Shutting down.
```

```
[+] FILE:
feb58e18dd320229d41d5b5932c14d7f2a26465e3d1eec9f77de211dc629f973
  #### PHASE 1 ####
  [-] ADD:  0x3
  [-] XOR:  0x13
  [-] SIZE: 80556
  [!] Success! Written to disk as
feb58e18dd320229d41d5b5932c14d7f2a26465e3d1eec9f77de211dc629f973_S1.exe
  #### PHASE 2 ####
  [-] XOR:  HEWRTWEWETHG
  [!] Success! Written to disk as
feb58e18dd320229d41d5b5932c14d7f2a26465e3d1eec9f77de211dc629f973_S2.exe
  ### PHASE 3 ###
  [-] http://api.ipify.org
  [-] http://google.com
  [-] http://bettitotuld.com/ls3/gate.php
  [-] http://tefaverrol.ru/ls3/gate.php
  [-] http://eventtorshendint.ru/ls3/gate.php
```

Analysis

The results were fairly unimpressive, however you win some and you lose some. It still provides some interesting observations.

For our sample set, there were only 3 C2 URLs across all 8,851 Hancitor payloads we successfully decoded:

```
hxxp://bettitotuld[.]com/ls3/gate.php
hxxp://tefaverrol[.]ru/ls3/gate.php
hxxp://eventtorshendint[.]ru/ls3/gate.php
```

Looking at the stage 1 payloads, we decoded 9,967, which is almost the entire set. Reviewing the metadata for the PE files, 8,851 exhibited the following characteristics, which are included in a YARA rule at the end of this document.

CompanyName: 'SynapticosSoft, Corporation.'
OriginalFilename: 'MpklYuere.exe'
ProductName: 'ngqlgdA'

Additionally, we identified three XOR keys being used in stage 1:

```
13   [-] XOR:  0xe
1103   [-] XOR:  0x14
8851   [-] XOR:  0x13
```

After correlating the data, each of the keys corresponded to a different stage 2 dropper and our script was designed to target and decoded the most heavily used. General observations for the other two decoders are that the one with key 0xE uses the same XOR key for the second stage Hancitor payload “HEWRTWEWETHG” and would likely be straightforward to add to the decoding script. The 1,103 other files with key 0x14 were identified as Nullsoft Installers.

For the 8,851 that successfully decoded their stage 2 payload, I did not note any PE’s with any file information; however, a YARA rule is included which matches them all. The last thing I’ll mention regarding the stage 2 files is the different file sizes.

```
3   [-] SIZE: 114688
10   [-] SIZE: 109912
1103   [-] SIZE: 162992
8851   [-] SIZE: 80556
```

This data is pulled from the variable in our shellcode and we can see that there is a slight file size variation in the 13 that used the XOR key 0xE, which might imply slightly modified payloads.

Conclusion

Hopefully this was an educational demonstration using the extremely powerful Unicorn Engine to build a practical malware decoder. These techniques can be applied to many different samples of malware and can free you up from the more tedious process of figuring out how to program a slew of bitwise interactions and focus more on analysis and countermeasures.

Indicators

At the following GitHub repository, you will find 3 YARA rules, listed below, which can be used to detect the various pieces described throughout these two blogs, and the script that was built throughout this blog for decoding Hancitor.

hancitor_dropper.yara – Detect Microsoft Word document dropper
hancitor_stage1.yara – Detect first PE dropper
hancitor_payload.yara – Detect Hancitor malware payload