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: <u>hancitor</u>, <u>LuminosityLink</u>, <u>Microsoft Word</u>, <u>Python</u>, <u>shellcode</u>, <u>Unicorn Engine</u>, <u>VB</u> <u>Dropper</u>, <u>WildFire</u>, <u>XOR key</u>



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

This blog post is a continuation of my previous post, <u>VB Dropper and Shellcode for Hancitor</u> <u>Reveal New Techniques Behind Uptick</u>, 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 <u>Unicorn Engine</u> 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 <u>looked at a large sample set of LuminosityLink malware samples</u>.

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.



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.

L	00000030 P		13	12	27	42	9.9	20	37		77	29	15		55	27	15
	000000000 9	2	80	11	20		5.0	AC.	10	50	78	39	50	20	10	5.0	79
	00000070	FE I	E 8	87	67	15	FC					80	78	BC	13	69	80
	BBBBBBB B	_		110	1944		150	82	3.6	12		85	BF.	41		FO	ac.
10c_892:	00000000 0 3	10 1	81	88	72	FD	57	ER	70	FC	FF	FF	83	70	FC	01	
nov al, [edi+ecx]	COODDERE O	1	15	67	80	88	88	53	56	60	82	53	60	81	68	88	88
add al, a	papager a			HD.	85	Ch	ED	EF.	FF	50	07	45	FC		FC		00
xor al, 13h	BRABBED B	EE I	20	C.B.	EE	EF	FE	59	-	EB	SD.	45	FC	58	EE	75	EC
nov [edi+ecx], al	BOBBBEE B	en i	15	DC	57	56	58	FF	75	FA	07	45	DC	57	29	40	7.6
inc ecx	BBBBBBE B P		-	EB	45	20	60	40	44	87	hE	EA	45	-	E.B.	20	ER
csp ecx, 13AACh	00000074			60	50		0.0	EA	EE	00	10	22	EF			A.C.	
jb short loc_892	agaggitag			24	24	67	200	20	10	10	45	1.	67	10		75	45
	CONTRACT IN S	240		12	1.4	67	45		44		02		11	49	-	12	0.2
	00000020 5	2.0	<u>ee</u>	L.r	45		or	0.1	85	13	00	57	45		10	22	
	00000030 5	50 1	HE.	-	83	10	**	**			#D	85	ac.	**	**	**	0.0
	000000040	18 1	50		0.0	FB	**	**	08	44	58	BD	85	CB	FE	**	11
push edi	00000050 5	57	50	E8	56	FB	FF	FF	83	CA	18	80	85	OC.	FF	FF	FF
call sub_827	00000060 5	50	6D	#5	C8	FE	FF	FF	50	53	53	53	58	53	53	53	H D
cmp [ebp+var 4], 1	00000070 8	85 (C.I4	FD	FF	FF	58	89	UD.	C	FE	FF	FF	FF	Dó	68	88
pop ecx	00000030 8	50	88	88	53	FF	75	88	FF	95	20	FF	FF	FF	5F	5E	33
inz 1oc C80	88888C98 C	CØ 1	58)	69	C2	00	88	88	88	0.0	00	00	88	88	00	00	88
	000000000		88	86	88	88	00	88	88	00	00	00	88	88	00	00	88

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 eta push eti usimi loc_t3f:	00000110 00000110 00000120 00000540 00000540 0000050 0000050 0000050	5 KG 77 77 10 10	60 C0 FC FF 80 DE 80 C0	88 84 14 15 14 15 15 15 15 15 15 15 15 15 15 15 15 15	47 C3 89 F 88 40 F 8	45 88 80 80 80 80	100 F # 104 F # 100 F # 100 F # 100 F # 100F #	相75 時間だ 町税	0 8 94 80 80 80 80 80 80 80 80 80 80 80 80 80	88 84 FC 78 EL 23	FF DE CO EN FF E OB	#557%7FF#14股	77 51 51 51 51 51 51 51 51 51 51 51 51 51	87 30 88 53 FF 88 08 15 42	#2565FFF21401	化乙酸乙酸酸	04 843 68 40 67 813 FT 08
Now eax (ebp+sar_4) (fint - fint		HE NO	5710 FF	SEMIFAI	5H 45 7B 64 59	19 88 44 59	510 32 66 64	54 08 57 68	81 67 45 59	78 45 58 68	80 80 60 65	111 NC 65	88 64 89	HU 72 50 80	40 84 84 84 84 84 84 84 84 84 84 84 84 84	110 C7 E8 FF	111 45 57 FF
now e.ax, (ebp+war_h) 00 call sub_707 00 now edi, eax 00 now edi, eax 00 new edi, eax 00	00000F0 0000900 0000910 0000910 0000920 0000920	FF 80 84 FF 84	50 34 60 FF	68 FF 66 FF	72 行前59 派	66 FF BD 6A FF	89 59 38 37 FF	8D 6A FF 66 FF	22 65 FF 89 59	646 117 800 646	89 59 FF	FF 80 68 FF 80	57 36 33 FF 40	88 FT 66 FT	征行时转行	66 FF BD 6A FF	89 59 38 2E 68

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.

\$75 r. 1011 Day 105 r	00000700	FC	ar.	87	-	36		05	87	. 83	145	-			55	-	FC
Ina err. Least-1001	REBBS70B	51	6.71	56	57	Cé	15	60	80	6.8	80	70	-		15	FC	88
con cl. 9	ROBDB7CB	-	-				-	-		-		68	CB	0.5	0.0	83	FP
in these los of h	ONDERIG TO B	87	1811	22	114		83	-	4.81		-	35	87				70
14 SHOPE 100,014	00000700	80	1.0							10	DE		2.0	10	22		
	BODDIT'S B	and the						-			Dr.					100	ar l
	0000071-0	1.1		10	1.1	22			48		100	1.4	14	"	87		BF
	account of	1.4		1.0	21	-		-			14			87		ST.	LW
anney any at	000000310	82	CO	-	-	36	28	12	84		at	58	•	**	28	15	
add air b loc atha	00000320	DR	24	58		33	CB						-		-	-	_
and car, a line and the	86666830	60	10		89	45	10		00	84	1.4	(1	04	24	20	27	1.9
Crean line should be Bar	86600340	83				88	0.9		45	FC.	0.0	70					40
Jus sume for the	80000950		ND.	14			45	F C	1.8	7.8	**	**	**				1.0
	00000940	F7	DE	нD	AE.	00	0.0	01	92	EØ	H.	CB	1.9	0.0	20	82	83
	00000170		80	70	29	HD	46	87	92	1.3	611	18	0.8	15	80	41	FE.
	000000000		•	03	FB	₩D	AE.	0.0	02	10	21	12	Cő	42	01	80	0.0
Anth Mile + Part	00000970	54	81	rr	-85	01		- 45		HA	**	FF	45	F C	54	C8	75
and the store	apphase a	91	SF.	5E	58	64		55		1.0	81	EG	30	82	08	80	53
Party and a second seco	*******	56	57	ШD	-85	DB	5.0	64	81	38			DD		14.00	HC	
rect	ODBDDDCD	48	10	FF	70	- 018	33	08	C7	45	De	AC.	64	72	ALC:	6.7	45
Jus smort for use	00000000	DA	6F.	61	64	44	66	C7	45	08	60	40	88	50	24	6.6	57
	82888998	FE	FF	FF	59	59	64	68	59	6A	65	ńó	89	80	28	FF	FF
	0000035.0	FF	59	60	72	66	89	90	32	FF	FF	FF	59	68	6E.	66	89
	85858788	#D	34	FF	FF	FF	59	64	65	66	89	HD.	36	FF	FF	FF	50
much NED - 191	00000910	68	60	66	89	80	28	FF	FF	FF	50	68	33	44		#D	38
and any loc star	88888928	55	FF	FF	59	68	32	6.5	89	HD	30	FF	FF	FF	59	68	2E
	00000938	66	89	BD	3E	FF	FF	FF	59	66	89	8D	40	FF	FF	FF	68
the same same	00000740	64	50	66	89	ND	3/2	FF	FF	FF	60	60	59	6.6	89	8D	14.14
CELO INT. anda	00000250	11	TT.	FF	0.0	.89	80	44		11	TT	33	0.0	44		SD.	14.00
Sub_rtrr enop	80000748	77	FF.	FF	BD	80	3.0	FF	**	FF.	89	5D	28	FF.	FF.	FF.	SD
	888889778	51	100	66	HH	31	83	01	82	66	2111	FB	75	15	28	0.0	Dt

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.

- 1 # sub_8A6
- 2 sc = $b^{-1}x8Ax04x0Fx04x03x34x13x88x04x0Fx41x81xF9xACx3Ax01x$
- 3 00\x72\xED\x57\xE8\x7C\xFC\xFF\xFF\x83\x7D\xFC\x01'
- 4 # sub_7CA
- 5 sc += b'\x6B\xC0\x06\x99\x83\xE2\x07\x03\xC2\xC1\xF8\x03\xC3'
- 6 # sub_7D7
- 7 sc +=
- 8 b'\x6B\xC0\x06\x25\x07\x00\x00\x80\x79\x05\x48\x83\xC8\xF8\x40\xC3'
- 9 # sub_7E7
- 10 sc +=
- 11 b'\x8D\x48\xBF\x80\xF9\x19\x77\x07\x0F\xBE\xC0\x83\xE8\x41\xC3\x8D\x
- 12 48\x9F\x80\xF9\x19\x77\x07\x0F\xBE\xC0\x83\xE8\x47\xC3\x8D\x48\xD0\x
- 13 80xF9\x09\x77\x07\x0F\xBE\xC0\x83\xC0\x04\xC3\x3C\x2B\x75\x04\x6A\x
- 14 3E\x58\xC3\x3C\x2F\x75\x04\x6A\x3F\x58\xC3\x33\xC0\xC3'
- 15 # sub_827
- 16 sc +=
- 17 b'\x55\x8B\xEC\x51\x51\x8B\x45\x08\x83\x65\xFC\x00\x89\x45\xF8\x8A\x
- 18 00\x84\xC0\x74\x68\x53\x56\x57\xE8\xA3\xFF\xFF\xFF\x8B\xD8\x8B\x45\x
- 19 FC\xE8\x7C\xFF\xFF\xFF\x8B\x4D\xF8\x8D\x14\x08\x8B\x45\xFC\xE8\x7B\x
- 20 FF\xFF\xFF\x8B\xF8\x8B\xF0\xF7\xDE\x8D\x4E\x08\xB0\x01\xD2\xE0\xFE\x
- 21 C8\xF6\xD0\x20\x02\x83\xFF\x03\x7D\x09\x8D\x4E\x02\xD2\xE3\x08\x1A\x
- 22 EB\x15\x8D\x4F\xFE\x8B\xC3\xD3\xF8\x8D\x4E\x0A\xD2\xE3\x08\x02\xC6\x
- 23 42\x01\x00\x08\x5A\x01\xFF\x45\x08\x8B\x45\x08\x8A\x00\xFF\x45\xFC\x
- 24 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.



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	Ç	D	E	E
0000h:	8A	04	OF	04	03	34	13	88	04	OF	41	81	F9	AC	ЗA	01
0010h:	00	72	ED	57	E8	61	00	00	00	83	7D	FC	01	6B	CO	06
0020h:	99	83	E2	07	03	C2	C1	F8	03	C3	6B	CO	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.



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.

- 1 # POLA 0x504F4C41
- 2 encoded_binary =
- 3 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 <u>Unicorn Engine</u>.

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.

- 1 ADDRESS = 0x1000000
- 2 mu = Uc(UC_ARCH_X86, UC_MODE_32)
- 3 mu.mem_map(ADDRESS, 4 * 1024 * 1024)
- 4
- 5 # Write code to memory
- 6 mu.mem_write(ADDRESS, X86_CODE32)
- 7 # Start of encoded data + offset to binary, pushed to Stack at
- 8 start
- 9 mu.reg_write(UC_X86_REG_EDI, 0x10000F9 + 0x0C)
- 10 # Initialize ECX counter to 0
- 11 mu.reg_write(UC_X86_REG_ECX, 0x0)
- 12 # Initialize Stack for functions
- 13 mu.reg_write(UC_X86_REG_ESP, 0x1300000)
- 14
- 15 print "Encrypt: %s" % mu.mem_read(0x10000F9,150)
- 16 mu.emu_start(ADDRESS, ADDRESS + len(X86_CODE32))
- 17 print "Decrypt: %s" % mu.mem_read(0x10000F9,150)



Figure 9 Successful decoding

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

- 1 f = open("demo.exe", "w") 2 f.write(mu.mem_read(0x10000F9 + 0x0C, 0x13AAC))
- 3 f.close()

neral Compatibi	lity Details Previous Versions
Property	Value
Description -	
ile description	MmaeZo
ype	Application
ile version	15.6.6.22
Product name	ngqlgdA
Product version	15,6,6,22
Copyright	Copyright 1990 - 2013
Size	78.6 KB
ate modified	8/24/2016 5:13 PM
anguage	English (United States)
Driginal 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, RtIMoveMemory(), and then copy what we presume is our encoded payload.

09441FFD 68 C0724000 PUSH demo.98407304 ASCII "SizeofResource" 09442020 68 A4724000 PUSH demo.98407304 ASCII "SizeofResource" 09442000 FD7 PUSH demo.98407304 ASCII "SizeofResource" 09442010 SD5 SD904080 HOU DUDRD PTR DS:14099041, ERX Itel I.Rt HowefResory 09442201 SD FFD6 PUSH 40 PUSH 40 09442201 SD FFD6 PUSH 40 PUSH 40 09442202 66 03390000 PUSH 40 PUSH 40 PUSH 40 09442202 FD 6 PUSH 40 PUSH 40 PUSH 40 09442202 FD 6 PUSH 40 PUSH 40 PUSH 40 09442202 FD FC7 PUSH 40 PUSH 40 PUSH 40 09442205 FD FC7 PUSH 40 PUSH 40 PUSH 40 </th <th>Address</th> <th>Hex dunp</th> <th>Disassembly</th> <th>Connent</th>	Address	Hex dunp	Disassembly	Connent
09442002 - 68 A4734008 PUSH demo: 09442384 09442002 - 68 A4734008 HOU DUGRD PTR D5: 140903C1, ERX 09442002 - FFD6 HOU DUGRD PTR D5: 140903C1, ERX 09442002 - FFD6 CALL EDI 09442002 - FFD6 CALL EDI 09442011 - 8015 39904808 HOU ECX, DU0RD PTR D5: 14090301, ERX 09442011 - 8015 39904808 HOU ECX, DU0RD PTR D5: 14090301, ERX 09442011 - 522 PUSH 4EX 09442021 - 64 40 PUSH 460 09442022 - 64 00 PUSH 460 09442205 - 64 00 PUSH 460 09442205 - 64 00 PUSH 40 09442205 - 64 00 PUSH 0 09442	00401FFD	1. 68 C8734008	PUSH deno, 88487308	RSCII "SizeofResource"
00402007 00402005 A3 3C904000 FFD7 DiV DU0RD PTR DS: [40903C], ERX CHL EDI PUSH EDX D0402015 ntdll.ftt Howeftemory utdll.ftt Howeftemory biologi PTR DS: [409030] 00402005 00402015 SD0 FFD6 FFD6 FFD6 00402011 ORL EDI PUSH EDX DVD LSX, DU0RD PTR DS: [409030] ntdll.ftt Howeftemory utdll.ftt Howeftemory biologi PTR DS: [409030] 00402015 00402011 S1 90402011 ORL EDX FFD6 FFD6 FFD8 FFD8 FFD8 FFD8 FFD8 FFD8	88482882	68 R4734008	PUSH demo, 00407394	FileName = "Kernel32.dll"
0044200C FF07 CPLL EDI Codd.braryW 0044200F F50 OPLL EDI CLOADLIFT 0044200F FF06 OPLL EDI CLOADLIFT 0044200F S00 44994080 MOV ECX, DUORD PTR DS: L4090441 The codd braryW 0044200F S00 44994080 MOV ECX, DUORD PTR DS: L40904301 demo.00400000 0044201F S1 PUSH ECX PUSH EDX demo.00400000 00442025 G8 00300000 PUSH 40 ntdll.RtlNowHemory 00442025 G8 00300000 PUSH 80 PUSH 60 00442025 G8 04734000 PUSH 80 PUSH 80 00442025 FF15 80724000 PUSH 40 PUSH 40 00442025 FF77 OPL 1000RD PTR D5: L4090401, ERX NULL 00442025 FF77 OPL 1000RD PTR D5: L4090401, ERX NULL 00442035 FF77 OPL 1000RD PTR D5: L4090401, ERX NULL	00402007	. A3 3C904000	HOU DWORD PTR DS: [40903C].ERX	ntdil.RtiMoveMemory
0040200E - 50° PUSH ED0 0040200E - 50° PUSH ED0 0040200F - 50° PUSH ED0 0040200F - 51° BD00 4490408 0040201F - 51° BD00 4490408 0040201F - 51° BD00 4490408 0040201F - 51° PUSH ED0 0040202F - 64° PUSH 40° 0040202F - 66° PUSH 40° 004020F - 77285CC0° 1999264960. 004020F - 66° PUSH 40° 004020F - 66° PUSH 40°	0040200C	. FFD7	CALL EDI	Load ibraryl
0040200F . FFD6 CPLL ESI MOV ECX, DUORD PTR DS: L4090441 HOV ECX, DUORD PTR DS: L4090441 HOV ECX, DUORD PTR DS: L4090401 S12 demo.00400000 demo.00400000 0040201F . S12 PUSH ECX PLSH ECX P	8848288E	. 58	PUSH ERK	ntdll.RtlHoveHenory
00442211 8060 44904000 000 ECX,DUORD PTR DS:L4090441 demo.00440500 00442211 51 9000 4490400 000 ECX,DUORD PTR DS:L4090301 demo.00440500 00442211 51 PUSH ECX PUSH ECX demo.00440500 00442201 52 PUSH ECX PUSH 40 demo.004405010 00442202 64 08 PUSH 40 ntdll.ftlibuettemory demo.004405010 00442202 64 08 PUSH 40 ntdll.ftlibuettemory demo.004405010 00442202 64 08 PUSH 3080 PUSH 3080 ntdll.ftlibuettemory demo.00440510 00442202 64 08 PUSH 3080 PUSH 3080 NUL ntdll.ftlibuettemory 00442202 64 08 PUSH 3080 PUSH 40 NULL Ntdlibuettemory 00442202 64 08 PUSH 6 PUSH 60 NULL Ntdlibuettemory 00442202 68 49294000 PUSH 60 PUSH 60 NULL Ntdlibuettemory 00442202 68 4734000 PUSH 60 PUSH 60 Ntdlibuettemory Ntdlibuettemory 004422051 68 4734000 PUSH 60 PUSH 60 Ntdlibuettem	8848288F	FFD6	CALL ESI	kernel32.GetProcAddress
09442201 09442201 8815 39964000 HOV EDX,DUORD PTR DS:[409838] demo.0040000 09442201 51 PUSH EDX PUSH EDX 09442201 64 48 PUSH 40 ntdll.RtlHoweHemory 09442202 66 00300000 PUSH 5000 PUSH 5000 09442202 66 00300000 PUSH 5000 PUSH 5000 09442202 66 00300000 PUSH 5000 PUSH 5000 09442202 66 00 PUSH 5000 PUSH 6000 09442202 66 00 PUSH 8000 PUSH 6000 09442202 66 00 PUSH 8000 PUSH 8000 09442202 66 00 PUSH 8000 PUSH 8000 09442203 64 00 PUSH 8000 PUSH 8000 09442203 64 00 PUSH 8000 PUSH 8000 09442203 64 00 PUSH 8000 PUSH 8000 09442203 68 447340000 PUSH 8000 PUSH 8000 094422051 68 447340000 PUSH 4000 OPTR DS:[409940] Rene132.lstring1 = NULL 094422051 50 PUSH 4000 OPTR DS:[409940] Rene132.lstring2 = NULL 094422051 50 PUSH 4000 OP	00402011	. 8B80 44904000	MOU ECX, DWORD PTR DS: [409044]	deno.004005C0
0044281D 51 PUSH ECK 0044281D . 51 PFD0 0044281D . 52 PUSH EDK 0044281D . 68 00300000 PUSH 40 0044281D . 68 00300000 PUSH 3060 00442821 . 68 00300000 PUSH 40 00442822 . 68 00 PUSH 3060 00442823 . 64 00 PUSH 3060 00442828 . 64 00 PUSH 0 00442845 . 66 94734000 PUS 10 CFL EDI 00442845 . 68 94734000 PUSH 0 09442845 <	00402017	. 8B15 38904000	HOU EDX, DWORD PTR DS: [409038]	deno. 00400000
00440201E - 52 PUSH EDX 00440201 - 68 40 PUSH 40 004402021 - 68 00300000 PUSH 40 004402023 - 68 00300000 PUSH 40 004402024 - 66 003000000 PUSH 400 004402025 - 50 PUSH 2000 004402030 - 68 00300000 PUSH 400 004402030 - 68 00 PUSH 800 004402030 - 67 00 PUSH 800 004402030 - 67 00 PUSH 80 004402030 - 68 00 PUSH 80 004402031 - 68 040 PUSH 80 004402031 - 68 04734000 PUSH 80 004402031 - 68 04734000 PUSH 40 004402031 - 68 04734000 PUSH 40 004402031 - 50 PUSH 40 004402031 - 68 04734000 PUSH 40007394 004402041 - FFD7 PUSH 4000 PUSH 4000 004402041 - FFD6	00402010	. 51	PUSH ECX	
004402011 004402021 004402021 004402021 004402021 004402022 004402022 004402022 004402025 004402025 004402025 004402025 004402025 004402025 004402025 004402052 00502052 00502000 00500000000	8848281E	. 52	PUSH EDX	demo.0040C010
004402021 004402021 004402025 004402025 004402025 004402025 004402025 004402025 004402052 00522 0052 00522 0052 00522 0052 00522 0052	0040201F	. FFD0	CALL ERK	ntdil.RtiHoveHemory
004402023 - 68 00300000 PUSH 3060 PUSH 3060 004402025 - 60 00 PUSH 4000 PUSH 4000 004402050 - 60 00 PUSH 4000 PUSH 4000 004402050 - 60 00 PUSH 00 PUSH 00 004402051 - 60 00 PUSH 00 PUSH 00 004402052 - 68 94734000 PUSH 00 PUSH 00 004402051 - 68 94734000 PUSH 00 PUSH 00 004402051 - 68 94734000 PUSH 00000 PTR D5: (1409040], ERX Istring1 = NULL 004402051 - 68 94734000 PUSH 00000 PTR D5: (1409040] NUMICOC "Kernel32.011" Rt HoveHemory" 004402051 - 68 94734000 PUSH 00000 PTR D5: (1409040] NUMICOC "Kernel32.011" Rt HoveHemory" 004402051 - 68 94734000 PUSH EDX NUMICOC "Kernel32.06 throeAddress 004402051 - 570 PUSH EDX NUMICOC "Kernel32.06 throeAddress	00402021	. 6A 48	PUSH 40	Protect = PAGE_EXECUTE_READMRITE
004402252 . 50 PUSH EDX 004402252 . 6A 00 PUSH # D 004402252 . 6A 00 PUSH 0 004402252 . 6A 00 PUSH 0 004402253 . 6A 00 PUSH 0 004402051 . 6A 00 PUSH 0 004402051 . 68 44734000 PUSH 0 004402051 . 68 44734000 PUSH 0 004402051 . 68 44734000 PUSH 0 004402051 . 50 PUSH EDX 004402051 . 51 30594000 PUSH EDX 004402051 . 8050 40904000 PUSH EDX 004402051 . 51 2 PUSH EDX 004402051 . 51 2 PUSH EDX 004402051 . 51 2 PUSH EDX 004402051 <td< td=""><td>00402023</td><td>. 68 88388888</td><td>PUSH 3000</td><td>AllocationType = MEM_CONMITIMEM_RESE</td></td<>	00402023	. 68 88388888	PUSH 3000	AllocationType = MEM_CONMITIMEM_RESE
004402029 . 6A 00 PUSH 0 Push 0 Push 0 Push 0 004402029 . 6A 00 PUSH 0 Push 0 Push 0 Push 0 00402029 . 6A 00 Push 0 Push 0 Push 0 Push 0 00402029 . 6A 00 Push 0 Push 0 Push 0 Push 0 00402035 . 6A 00 Push 0 Push 0 Push 0 Push 0 00402035 . 6A 00 Push 0 Push 0 Push 0 Push 0 00402035 . 63 04734000 Push 0 Push 0 Push 0 Push 0 00402035 . 68 04734000 Push 0 Push 0 Push 0 Push 0 00402035 . 68 04734000 Push 0 Push 0 Push 0 Push 0 00402035 . 68 04734000 Push 0 Push 0 Push 0 Push 0 00402035 . 68 04734000 Push 0 Push 0 Push 0 Push 0 00402055 . 68 04734000 Push 0 Push 0 Push 0 Push 0 Push 0 00402055 . 570 Push 0 Push 0 Push 0	00402828	. 50	PUSH ERK	Size = 772ASCC8 (1999264960.)
004020250 . A3 4C904000 DOU DUORD PTR DS:L40904C)_EAX ntdll.ttlhowfeenory 004020350 . 6A 00 DOULDOOD PTR DS:L(40904C)_EAX UirtualAlloc 004020350 . 6A 00 PUSH 0 DS:L(4090402)_EAX UirtualAlloc 004020351 . 63 04 PUSH 0 DUDORD PTR DS:L409040]_EAX String2 = NULL 004020351 . 63 94734000 PUSH demo.00407394 UIRLD00CF Kernel32.lstropy 00402051 . 68 94734000 PUSH demo.00407394 URIDODE Kernel32.dll" 00402051 . 50 PUSH demo.00407384 URIDODE Kernel32.dll" 00402051 . 50 PUSH EDX Ntdll.ftllowflemory 00402051 . 51 S004000 PUSH EDX 00402052 . 51 S0040000 PUSH EDX 00402051 . 51 PUSH EDX PUSH EDX 00402052 . 51 PUSH EDX PUSH EDX 00402053 . 51 PUSH EDX 0eno.0040C010 00402054 . 552 PUSH EDX 0eno.0040C010 00402055 . 51 PUSH EDX 0eno.0040C010 004020567 . 552 PUSH EDX 0	00402029	. 6A 88	PUSH 0	Address = NULL
00402030 . FF15 FC714000 CPLL DWORD PTR DS:[(4KERNEL32.Uirtual WirtualAlloc 00402035 . 6A 00 PUSH 0 PUSH 0 00402035 . 6A 409 PUSH 0 String1 = NULL 00402035 . 6A 409 PUSH 0 String2 = NULL 00402035 . FF15 80724000 PUSH 0 NU MORD PTR DS:[(4KERNEL32.Istropy 00402035 . 68 44734000 PUSH demo.08407394 NU MORD PTR DS:[1000FD PTR DS:[100	00402020	. A3 4C904000	HOU DWORD PTR DS: [40904C], EAX	ntdll.RtlHoveHemory
004022055 . 6A 00 PUSH 0 004022055 . 6A 00 PUSH 0 004022051 . 6A 00 PUSH 0 004022052 . A3 40904000 PUSH 0 004022054 . 68 94734000 OUCRD PTR D5:1(409040],ERX 004022051 . 68 94734000 PUSH demo.00407394 004022051 . 68 94734000 PUSH demo.00407394 004022051 . 68 94734000 PUSH demo.00407394 004022051 . 50 PUSH demo.00407394 004022051 . 50 PUSH bEX 004022052 . FFD6 PUSH bEX 004022052 . 51 PUSH bEX 004022054 . 8080 40904000 PTR D5:140904C1 004022057 . 51 PUSH EX 004422062 . 55 PUSH E	00482830	. FF15 FC714000	CALL DWORD PTR DS: [{&KERNEL32.Uirtual	VirtualAlloc
004022053 . 64 00 905H 0 <td>00402036</td> <td>. 6A 88</td> <td>PUSH 0</td> <td>String2 = NULL</td>	00402036	. 6A 88	PUSH 0	String2 = NULL
00402033 00402035 . A3 40904000 	00402038	. 68 66	PUSH 0	String1 = MULL
00440205 FF15 00724000 CHL DMORD PTR DS: ICAKERMEL32.1stropy Istropy 004402051 -68 44734000 PUSH demo.004407394 ASCI1 "Rt HoveHemory" 004402051 -68 44734000 PUSH demo.004407394 UNICODE "Kernel32.dll" 004402051 -50 PUSH demo.004407394 UNICODE "Kernel32.dll" 004402051 -50 PUSH EDI ntdll.Rt HoveHemory 004402051 -50 PUSH EDI ntdll.Rt HoveHemory 004402054 -8880 4C904000 HOV ECX.DUORD PTR DS: L40904C1 demo.00440C010 004402054 -8880 4C904000 HOV ECX.DUORD PTR DS: L40904C1 demo.00440C010 004402055 -51 PUSH ECX PUSH ECX demo.00440C010 004402055 -51 PUSH ECX demo.00440C010 demo.00440C010 004402055 -51 PUSH ECX demo.00440C010 demo.00440C010 004402056 -55E POP EDI 00170000 00170000 004402056 -	00482839	· A3 40904000	HOU DWORD FTR DS: [409040], EAX	ntdil.Rtinovenemory
00342095 - 68 94734008 PUSH demo.08407394 HSL11 "Rt (hoverhenory" 00342045 - 68 4734008 PUSH demo.08407394 UNICODE "Kernel32.cli" 00342047 - FFD7 - 68 4734008 PUSH demo.08407394 UNICODE "Kernel32.cli" 00342047 - FFD7 - FFD7 - GRL EDI vernel32.cli" vernel32.cli" 00442055 - SB0 4C904808 HOV ECX, DUORD PTR DS: L40984C1 ntdll.Rt Hoverhemory" vernel32.GetProcAddress 00442055 - SB0 4C904808 HOV ECX, DUORD PTR DS: L40983C1 demo.084802010 demo.084802010 00442056 - S1 - PUSH ECX PUSH ECX demo.084802010 00442056 - S1 PUSH ECX demo.084802010 demo.084802010 00442056 - S1 PUSH ECX demo.084802010 demo.084802010 00442065 - S1 PUSH EDX ntdll.Rt Hoverhemory demo.084802010 00442065 - S1 PUSH EDX demo.084802010 demo.084802010 00442065 - S1 PUSH EDX e0170600 e0170600 00442065 - S1 <td>0040203F</td> <td>. FF15 88724888</td> <td>CALL DWORD PTR DS: [(LKERNEL32. 1stropy</td> <td>IstropyA</td>	0040203F	. FF15 88724888	CALL DWORD PTR DS: [(LKERNEL32. 1stropy	IstropyA
09482044 - 58 H4734088 CRL EDI ORICOLE Kernel32.cdl" 09482051 - 57D7 PUSH EDG Number of the construction	00402045	. 68 94734000	PUSH demo, 00407394	HSCII "Ht Inovenenory"
09482294 FPU7 0412 EDI Kernel32.Load.iDraryW 094822952 FPD6 OLL EDI ntdll.Rtilovellemory 09482955 FPD6 OLL ESI ntdll.Rtilovellemory 09482955 8880 40994898 HOV ECX,DWORD PTR DS: L40983C1 demo.804802016 09482956 51 PUSH ECX PUSH ECX 09482956 51 PUSH ECX demo.804802016 09482956 51 PUSH ECX demo.804802016 09482956 S1 PUSH EXX demo.804802016 09482956 S1 PUSH EXX demo.804802016 0948	0040204H	· 68 H4/34000	PUSH demo, 004073H4	UNICODE "Kernel32.dil"
00402051 .50 PDSH_EAX ntdll.ntlNovRemory 00402052 .FFD6 CALESI ntdll.ntlNovRemory 00402054 .8880 4094808 HOV ECX,DWORD PTR DS:[409840] demo.00400010 00402057 .8880 4094808 HOV ECX,DWORD PTR DS:[409830] demo.00400010 00402057 .52 PUSH EDX demo.00400010 00402055 .51 PUSH EDX demo.00411.ntlNoveHemory 00402050 .52 POP EDI 00170000 00402050 .53 RETN 00170000 00402050 .63 RETN 00170000	0040204F	· FFU/	EDI EDI	kernel32.LoadLibraryW
09482052 • FFD0 4C2 652 Kernel32.setFrocHodress 09482054 • BB15 3C594888 HOV ECX,DWORD PTR DS: L40984C1 demo.80480218 09482055 • BB15 3C594888 HOV ECX,DWORD PTR DS: L40983C1 demo.80480218 09482055 • S1 409048205 PUSH ECX DWORD PTR DS: L4098401 demo.80480218 09482055 • S1 909482055 • S1 PUSH ECX DWORD PTR DS: L4098401 demo.80480218 09482055 • S1 PUSH ECX PUSH ECX ntdl11.RtilNoveltemory 09470000 09482055 • SE POP EDI 09170000 09170000 09170000 09482050 • C3 RETN 09170000 09170000 09170000	00402051	. 50	PUSH ENK	ntdil. Ht inovenemory
00402057 . 8000 4C90488 HOV ECX.000RD PTR D5:140993C1 demo.8048C010 00402056 .51 PUSH ECX. PUSH ECX.000RD PTR D5:14093401 demo.8040C010 00402057 .52 PUSH ECX. DUORD PTR D5:14093401 demo.8040C010 00402058 .51 PUSH ECX. DUORD PTR D5:14093401 demo.8040C010 00402057 .52 PUSH ECX. DUORD PTR D5:14093401 demo.8040C010 00402058 .51 PUSH ECX. DUORD PTR D5:14093401 demo.8040C010 00402050 .55 POP EDI 00170000 00170000 00402050 .C3 RETN 00170000 00170000	00402052	· FFUB	LALL EDI	Kernelsz.GetProcHddress
09482054 . 51 3015 3C.904808 PUSH ECX 00482051 0480.0010 </td <td>00402054</td> <td>. ODUL 20004000</td> <td>HOU EDV, DUORD PTR DS: 14090401</td> <td>dama 00405010</td>	00402054	. ODUL 20004000	HOU EDV, DUORD PTR DS: 14090401	dama 00405010
09462061 .9800 40964080 HOV ECX, DUORD PTR DS: [409040] demo.00400010 09462067 .52 PUSH EDX demo.0040000 demo.0040000 09462067 .51 PUSH EDX ntdll.RtlHoveHemory 09462067 .51 PUSH EDX ntdll.RtlHoveHemory 09462067 .51 POP EDI 00170000 09462067 .55 POP EDI 00170000 09462067 .55 POP EDI 00170000 09462060 .63 RETN 00170000	0040205H	. 0015 30904000	DUCH FOR	0eno.00400010
Objectoria Sector 46/56/86 INCREMENTING demo.00402010 00402057 • 52 PUSH EDX demo.00402010 00402058 • 51 PUSH EDX ntdll.RtlMoveHemory 00402056 • 5F POP EDI 00170000 00402056 • 5F POP EDI 00170000 00402050 • C3 RETN 00170000	00402060	· 0000 40004000	HOLL ECV DUODD DTD DC. [4000401	
00402062 • 52 PUSH ECK 00402062 • 51 PUSH ECK 00402062 • 5F POP EDI 00170000 00402062 • 5F POP EDI 00170000 00402062 • 5E POP ESI 00170000 00402062 • C3 RETN 00170000	00402061	· 6560 40304000	DICH FRY	days 00400010
OCTOPERSY From CFL ERG ntdll.RtlNovelemory 00442062 .5F POP EDI 00170000 00442062 .5E POP EDI 00170000 00442062 .5E POP EDI 00170000 00442062 .C3 RETN 00170000 00442062 .C3 RETN 00170000	00402057	· 25	PUCH ECV	0640.00400010
00442868 00442862 00442862 00442862 00442265 C2 RETN 00478088 0017808 0017808 0017888 0017888 0017888 001788 001788 0017888 0017888 0	DEMONICIPAL	FEDO	FOR ECA	ARALL DELHAMANAMANAN
00402060 C SE POP ESI 001700000 00402060 C C3 RETO	08492969	SE	POP EDI	00170000
00402060 L. C3 RETR	88482860	·	POP EST	00170000
0040202E NC INTR	00402060		DETN	00110000
	00402025	00	INTS	

Figure 11 RtlMoveMemory() being called

Dump - 001	700000	0174FFF			
00170000 05 1 00170010 FD 57 00170020 57 52 00170020 57 52 00170020 57 52 00170020 57 52 00170020 57 52 00170020 21 36 00170020 31 72 00170070 31 32 00170070 31 22 00170070 32 00170070 32 00170070 32 00170070 32 00170070 32 00170070 32 00170070 32 00170070 32 00070070 32 00070070 32 0007000000000000000000000000000000000	C7 52 45 54 55 45 55 45 57 22 30 30 56 4 8 45 59 27 31 30 89 56 4 8 99 56 4 59 57 20 30 89 56 4 59 57 50 50 50 50 50 50 50 50 50 50 50 50 50	57 57 45 57 48 45 57 55 57 52 54 5 45 57 45 2 45 50 41 8 26 38 22 2 68 5A 48 5 29 41 18 F 29 41 18 F 29 44 0B F 29 44 0B F	57 41 54 48 45 A 52 14 57 45 57 52 5 54 48 45 57 52 5 57 45 57 45 57 54 9 88 76 EA 55 18 8 25 24 39 68 26 3 35E 6C 45 57 52 5 78 38 44 09 FB 3 36E 64 55 752 5 F8 38 44 09 FB 3 F8 38 14 F6 FA 8 FD B8 1A F6 FA 8	8 AD 54 57 4 5 54 48 45 4 57 45 57 52 E 8 76 11 3C 1 8 76 11 3C 1 1 15 18 18 57 1 15 18 18 57 4 57 45 7 8 76 11 3C 1 1 15 18 18 57 4 57 45 7 4 57 45 7 5 7 5 7 5 7 5 7 5 7 5 7 5 7	THRWWEWATHE2.1TW WETHEWRTWEWETHE RTWEWETHEWRTWEW THEWRTWEWETEEWR H YExAFévΩU+év4 6w"&8"%59h&6<:8

Figure 12 Encoded payload

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



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 - 001700	0000174FFF				
00170000 4D 5A 90 00170010 B8 00 00 00170020 00 00 00	00 03 00 00 00 00 00 00 00	00 00 04 00 00 00 40 00 00 00 00 00	00 00 FF FF 0 00 00 00 00 0 57 52 54 57 4	00 00 MZÉ.♥♦ 00 00 ₹@ 45 57WRT	 WEW

Figure 14 Validation of decoding

We've now found the location of the encoded binary, due to RtIMoveMemory(), 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	He	K d	IMP														ASCII
0040740E 0040741E 0040742E 0040743E 0040743E 0040744E 0040745E 0040746E	6F 677 564 73	6ECF 2300	740664754	65 05 57 20 00	78 00 65 47 66 61	74 00 63 57 42 00 00	004E44544400	00 74 69 47 00 00	0055F80680	00 6EE 40 00 00	6E 00 3 26 3 26 3 50 50 50 50 50 50 50 50 50 50 50 50 50	74 61 00 45 00 90	64 70 052 20 64 052 64 0	6C 56 80 80 80 80 80 80 80 80	609 48 49 68 68 68	2E55 457 000 90	ontextntdll. dllNtUnmapVie wOfSectionHE WRTWEWETHGSER.*g dsG2GBDG&.*y g.u.f.d.k.s.j.f. s.d.aPE0.će

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.



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

- 1 # loc 406442
- 2 sc =
- 3 b'\x85\xC9\x7C\x29\x8B\x35\x40\x90\x40\x00\xB8\x67\x66\x66\x66\xF
- 4 7\xE9\xC1\xFA\x02\x8B\xC2\xC1\xE8\x1F\x03\xC2\x8D\x04\x80\x03\xC0
- 5 \x8B\xD1\x2B\xD0\x8A\x82\x3C\x74\x40\x00\x30\x04\x0E\x41\x3B\x0D\
- 6 x4C\x90\x40\x00\x72\xCA'
- 7 # XOR Key
- 8 sc += b'\x48\x45\x57\x52\x54\x57\x45\x57\x45\x54\x48\x47'
- 9 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 IpBuffer 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	в	ç	D	E	F	0123456789ABCDEF
0000h:	85	C9	7C	29	BE	42	00	00	01	90	B 8	67	66	66	66	F7	_É)%8,gfff÷
0010h:	E9	C1	FA	02	8B	C2	C1	E8	1F	03	C2	8D	04	80	03	CO	éÁú.∢ÂÁè€.À
0020h:	8B	D1	2B	DO	8A	82	3C	74	40	00	30	04	OE	41	3B	OD	<Ñ+ĐŠ, <t@.0a;.< td=""></t@.0a;.<>
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	HGÇRWWEWATHE -
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	В	ç	D	Ε	Ŧ	0123456789ABCDEF
0000h:	85	C9	7C	29	BE	42	00	00	01	90	B 8	67	66	66	66	F7	É)%8,gfff÷
0010h:	E9	C1	FA	02	8B	C2	C1	E8	1F	03	C2	8D	04	80	03	CO	éÁú.∢ÂÁè€.À
0020h:	8B	D1	2B	DO	8A	82	36	00	00	01	30	04	0E	41	3B	OD	<Ñ+ĐŜ,60A;.
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	в	ç	D	E	F	0123456789ABCDEF
0000h:	85	C9	7C	29	BE	42	00	00	01	90	B 8	67	66	66	66	F7	_É)\8,gfff÷
0010h:	E9	C1	FA	02	8B	C2	C1	E8	1F	03	C2	8D	04	80	03	CO	éÂú.∢ÂÁè€.À
0020h:	8B	D1	2B	DO	A8	82	36	00	00	01	30	04	0E	41	81	79	<n+đš, 60a.ù<="" td=""></n+đš,>
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	AB	AD	HGÇRWWEWATHE"-
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.

- 1 ADDRESS = 0x10000002 mu = Uc(UC ARCH X86, UC MODE 32) 3 mu.mem map(ADDRESS, 4 * 1024 * 1024) 4 5 # Write code to memory mu.mem write(ADDRESS, X86 CODE32) 6 7 # Start of encoded data 8 mu.reg write(UC X86 REG EDX, 0x1000042) 9 # Initialize ECX counter to 0 10 mu.reg write(UC X86 REG ECX, 0x0) # Initialize Stack for functions 11 mu.reg write(UC_X86_REG_ESP, 0x1300000) 12 13 14 print "Encrypt: %s" % mu.mem read(0x1000042,250) mu.emu start(ADDRESS, ADDRESS + len(X86 CODE32)) 15
- 16 print "Decrypt: %s" % mu.mem read(0x1000042,250)

This yields the following result:



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	.text0x31CD	*	http://api.ipify.org
ascii	109	.text0x3E4A	*	http://bettitotuld.com/ls3/gate.phplhttp://tefaverrol.ru/ls3/gate.phplhttp://eventtorshendint.ru/ls3/gate.php
ascii	17	.text0x31F8	*	http://geogle.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 <u>Hancitor decoder script</u> 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.

- 1 [+] FILE: fe23150ffec79eb11a0fed5e3726ca6738653c4f3b0f24dd9306f6460131b34c
- 2 #### PHASE 1 ####
- 3 [-] ADD: 0x3
- 4 [-] XOR: 0x13
- 5 [-] SIZE: 80556
- 6 [!] Success! Written to disk as
- 7 fe23150ffec79eb11a0fed5e3726ca6738653c4f3b0f24dd9306f6460131b34c_S1.exe
- 8 #### PHASE 2 ####
- 9 [-] XOR: HEWRTWEWETHG
- 10 [!] Success! Written to disk as
- 11 fe23150ffec79eb11a0fed5e3726ca6738653c4f3b0f24dd9306f6460131b34c_S2.exe
- 12 ### PHASE 3 ###
- 13 [-] http://api.ipify.org
- 14 [-] http://google.com
- 15 [-] http://bettitotuld.com/ls3/gate.php
- 16 [-] http://tefaverrol.ru/ls3/gate.php
- 17 [-] http://eventtorshendint.ru/ls3/gate.php
- 18 [+] FILE:
- 19 fe7d4a583c1ae380eff25a11bda4f6d53b92d49a7a4d72c775b21488453bbc96
- 20 #### PHASE 1 ####
- 21 [-] ADD: 0x3
- 22 [-] XOR: 0x13
- 23 [-] SIZE: 80556
- 24 [!] Success! Written to disk as
- 25 fe7d4a583c1ae380eff25a11bda4f6d53b92d49a7a4d72c775b21488453bbc96_S1.exe
- 26 #### PHASE 2 ####
- 27 [-] XOR: HEWRTWEWETHG
- 28 [!] Success! Written to disk as
- 29 fe7d4a583c1ae380eff25a11bda4f6d53b92d49a7a4d72c775b21488453bbc96_S2.exe
- 30 ### PHASE 3 ###
- 31 [-] http://api.ipify.org
- 32 [-] http://google.com
- 33 [-] http://bettitotuld.com/ls3/gate.php
- 34 [-] http://tefaverrol.ru/ls3/gate.php
- 35 [-] http://eventtorshendint.ru/ls3/gate.php
- 36 [+] FILE: fea98cc92b142d8ec98be6134967eacf3f24d5e089b920d9abf37f372f85530d
- 37 #### PHASE 1 ####
- 38 [-] ADD: 0x3
- 39 [-] XOR: 0x14
- 40 [-] SIZE: 162992
- 41 [!] Success! Written to disk as
- 42 fea98cc92b142d8ec98be6134967eacf3f24d5e089b920d9abf37f372f85530d_S1.exe
- 43 #### PHASE 2 ####
- 44 [-] XOR: ð~ð~ð~
- 45 [!] Detected Nullsoft Installer! Shutting down.

- 46 [+] FILE:
- 47 feb58e18dd320229d41d5b5932c14d7f2a26465e3d1eec9f77de211dc629f973
- 48 #### PHASE 1 ####
- 49 [-] ADD: 0x3
- 50 [-] XOR: 0x13
- 51 [-] SIZE: 80556
- 52 [!] Success! Written to disk as
- 53 feb58e18dd320229d41d5b5932c14d7f2a26465e3d1eec9f77de211dc629f973_S1.exe
- 54 #### 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:

- 1 hxxp://bettitotuld[.]com/ls3/gate.php
- 2 hxxp://tefaverrol[.]ru/ls3/gate.php
- 3 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:

1 13 [-] XOR: 0xe 2 1103 [-] XOR: 0x14

3 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.

- 1 3 [-] SIZE: 114688
- 2 10 [-] SIZE: 109912
- 3 1103 [-] SIZE: 162992
- 4 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 <u>GitHub repository</u>, 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

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>.