Malware Unpacking With Hardware Breakpoints - Cobalt Strike Shellcode Loader

embee-research.ghost.io/unpacking-malware-with-hardware-breakpoints-cobalt-strike/

Matthew

November 6, 2023

Intermediate

Unpacking a simple Cobalt Strike loader using Debuggers and Hardware breakpoints.

Unpacking Malwa	re With Hard Leveraging Ghidra, X64d	Ware E	Breakpoints - Cobal	t Strike
9 DWORD local_2c; 10 11 lpAddress = VirtualAlloc(NULL,param_2,0x3 12 for (lVarl = 0x0; lVarl < param 2; lVarl		<u>)4</u>	<pre>xor dl,byte ptr ds:[r12+rax] mov byte ptr ds:[rbx+rax],dl inc rax cmp eax.edi cmov edv.eav</pre>	
<pre>13 *(lpAddress + lVarl) = *(param_3 + (lVar 14 } 15 FUN_004014f3(lpAddress); 16 VirtualProtect(lpAddress,param_2,0x20,61c 17 CreateThread(NULL,0x0,6LAB_004014f0,lpAdd 18 return; 19 }</pre>	cal_2c); ress,0x0,NUL	¥	<pre>mov edx.eax j1 99.401561 mov rcx.rbx call 99.4014F3 lea r9.qword ptr ss:[rsp+3C] mov rdx.rsi mov rcx.rbx mov r8d.20 call gword ptr ds:[//intualprotect>]</pre>	rcx:NtAllocateV ⁺ rcx:NtAllocateV ⁺ 20:
Prom Hex O II Delimiter Auto XOR O II	Ce 67 6e 72 C2 C e4 62 67 02 21 7 20 107 01 35 9 6f 2d df 33 ff e 7e 2c 41 b2 3a 6 97 e2 6e 55 d7 6 50 fc 32 66 b3 e 62 3c 56 7f 1e C fb 6e 5c d7 63 4	B	piled code. 0; IVar1 < param_2; IVar1 = IVar1 + 0×1) { .+ IVar1) = *(param_3 + (IVar1 & 0×3)) ^ *(param_1 + IVar1);	
Standard Standard	a4 f 2a ab 5f c 4 75 39 eb c 5 1 c 4 7 t 45 7 a 8 56 53 26 66 17 6 73 56 90 65 32 2 f 4 c 2 c 47 e6 5f c 4 73 95 20 90 2a 54 f 2 43 b7 ef 62 13 da ad f 2 69 c 0 80 35 ef 7a 4b 6d 4c f c a 2 ab d0 77 70 ab 55 6f 66 a3 19 c a5 65 19 44 11 6e f d 8f 81 15 34 79 64 00 4f 7d eL 7a bb 32 bb 23 a6 12 c 76 ef 95 5f 6c c 2 5b 4d 6l f 3 em 3011 r 1	The provideo each byte of `param_3` u: stored at `1p	d code snippet performs a simple form of XOR encryption or ol input data at ` param_1 ` is combined with a repeating 4-byte p sing the XOR bitwise operation. This technique creates an obfr Address` and can be easily reversed if the 4-byte pattern (key	bfuscation, where D C battern from uscated output /) is known. The
	Output	security of th	is method is low, as it relies on the obscurity of the repeating k	key rather than

In previous posts <u>here</u> and <u>here</u>, we explored methods for extracting cobalt strike shellcode from script-based malware.

In this post, we'll explore a more complex situation where Cobalt Strike shellcode is loaded by a compiled executable .exe file. This will require the use of a debugger (x64dbg) in conjunction with Static Analysis (Ghidra) in order to perform a complete analysis.

Overview

The executable is a compiled exe containing hidden and obfuscated Shellcode. The shellcode is decoded using a simple XOR routine and a 4-byte key, is then written to a simple buffer created with VirtualAlloc.

We will explore methods for obtaining the decoded shellcode using a debugger, and we will then explore methods for manually locating the Shellcode and associated decryption keys using Ghidra. We'll also look at a way to pivot between X64dbg and Ghidra, as well as a method for identifying and analysing Ghidra output using ChatGPT.

Obtaining the Sample

You can follow along by downloading the sample <u>here on Malware Bazaar</u> (pw:infected)

SHA256: 99986d438ec146bbb8b5faa63ce47264750a8fdf508a4d4250a8e1e3d58377fd

Analysis

We can begin by saving the file to an analysis machine and unzipping it with the password infected. From here we can also create a copy with a shorter file name.



Since the file is a compiled executable, we can attempt to analyse it using a debugger. In this case x64dbg.

We can go ahead and open the file with x64dbg, clicking through until we reach the entry point.

I	• *	?	>> 4	🕇	→ ≞	8	/ =	49	11	fx	#	Az	₽,		9					
1	Notes	• E	Breakpoints	EEE Me	emory Map		Call Stack	2		D	Script	2	Symbols	\diamond	Source	 References	 Threads	-	Handles	
		0401480 140148E 140148E 14014C3 14014C3 14014C9 14014C9 14014CF 14014CF 14014CF 14014CE 14014E3 14014E8 14014E8 14014E9 14014E9 14014E9 14014E8 14014E8 14014E9 14014E8 140	48:8 C705 E8 E 90 90 48:8 C3 8 28 90 48:8 C705 E8 9 90 48:8 C3 8 28 90 48:8 C3 28 28 28 28 28 28 20 20 20 20 20 20 20 20 20 20	326C 28 526B0000 19150000 18FCFFFF 33C4 28 326C 28 5926B0000 19150000 18FCFFFF 33C4 28	010000 mc ca ca ca ca ca ca ca ca ca ca ca ca ca	b rsp,2 v dword 1 99.40 1 9	8 ptr ds: [4 22A80 201180 8 8 8 ptr ds: [4 22A80 201180 8	08070],1)		Optiona	l Heade	r.Address(OfEntry	Point					
	0000000000 000000000 000000000 00000000	04014EF 04014F3 04014F3 04014F7 04014FD 0401501 0401508 0401508 0401508 0401513 0401513 0401517	90 48:E 48:E 8800 85CC 7E 2 833D 7E 1 48:E 48:E 48:E 48:E	FE1 3EC 18 0F2B0000 8 0082B0000 F 8 8 19 1401 305 F22AC	0 00 cm 0 000 mc 0 000 mc	pp rcx hb rsp,1 by eax,d est eax, e 99.40 pp dword e 99.40 by rdx,q lge by rdx,q lge by rdx,q	8 word ptr c eax L529 ptr ds:[4 L529 word ptr c ptr ds:[r <,dword pt	s:[40400 04010],(s:[<getm cx+rax], r ds:[40</getm)с]) toduleHa ,rdx)4010]	ndleA>	eax:Entr eax:Entr rdx:Entr rdx:Entr rax:Entr	∙yPoin •yPoin •yPoin •yPoin	t t t t						-	•

We can now go ahead and create some breakpoints on API's that are commonly (but not always) used when malware is unpacking.

We can go ahead and create 2 breakpoints by running bp VirtualAlloc and bp VirtualProtect

00007FF 00007FF 00007FF	BADD310B0 BADD310C0 BADD310D0	42 48 49	C6 33 8B	04 сс 5в	3E E8 28	00 B8 49	EB B2 8B	02 08 73	EB 00 30	02 4C 49	33 8D 8B	C0 9C E3	48 24 41	8B 80 5F
Command:	bp Virtual	LAll	oc											
Paused	Breakpoir	nt at	000	<u>07FF</u>	BAC	OCBI	<u>D30</u>	set!						

After creating the breakpoints, we can go ahead and allow the malware to continue (F9)

The malware will continue to run and trigger a breakpoint on VirtualAlloc.

Our primary purpose here is to obtain the buffer being created by VirtualAlloc, we can do this by using Execute Until Return.

"Execute Until Return" will allow the VirtualAlloc function to complete, but won't allow any further actions to occur. This means we can easily obtain the address of the buffer that was created.

Viewing Memory Created by VirtualAlloc

Debug Tracing Plugins Favourites Options Help Jun 20 2023 (Titaribingine)	
🔳 🔿 II 🝷 🐢 🦛 🍹 📫 🛐 🥒 🚍 🛷 🥒 fx # Az 🖳 🗐 🌻	
関 Log 😰 Notes 🔹 Breakpoints 🚥 Memory Map 🗊 Call Stack 🤷 SEH 📓 Script 🎴 Symbols 🗘 Source 🔎 References 🌱	🛎 Threads 🙀 Handles 👔 Trace
[00007FFBAC0C8C9014]	Hide FPU
00007FFBAC063C97 CC inc3 00007FFBAC063C99 CC inc3 00007FFBAC08439 CC inc3 00007FFBAC08439 CC inc3 00007FFBAC08439 CC inc3 00007FFBAC08430 CC inc3 00007FFBAC08430 CC inc3 00007FFBAC08431 CC inc3 00007FFBAC08432 CC inc3 00007FFBAC08432 CC inc3 00007FFBAC08433 CC inc3 00007FFBAC08432 CC inc3 00007FFBAC08432 CC inc3 00007FFBAC08432 CC inc3 00007FFBAC08432 CC inc3 00007FFBAC08434 CC inc3 00007FFBAC08434 CC inc3	КАХ ООВООДОВОДОВОДОВОДО КСХ ООВООДОВОДОВОДОВОДО КСХ ООВООДОВОДОВОДО КСХ ООВООДОВОДОВОДО КСХ ООВООДОВОДОВОДО КСХ ООВООДОВОДОВОДО КСХ ООВООДОВОДОВОДОВОДО КСХ ООВООДОВОДОВОДОВОДОВОДОВОДОВОДОВОДОВОДО
00007FF8AC08C8B 00007FF8AC08cc2 00007FF8AC08cc8	LastError 0000000 (ERR08_SUCCESS) LastEstatus CO000139 (STATUS_ENTRYPOINT_NOT_FOUND)
	Default (x64 fastcall) 🗸 🗸 🗍 Unlocked
(0000FFBAC132828 «kernel32.VirtualAllocs)«kernelbase.VirtualAllocs BAC0C8C90 kernel32.d11:518C90 #18990 «virtualAllocs	1: res 00000000000000000000000000000000000

After hitting execute until return. We can observe the address of the newly created buffer inside of RAX.

We want to go ahead and monitor this buffer for suspicious content and unpacked malware.



We can begin the monitoring process by right-clicking on the address contained inside of RAX.

From here we can select Follow in Dump. This will open the content of the buffer in the bottom-left window.

Jes -	miedus	Tianules	r nace		
			Show	FPU	
	RAX 00000000 KDA 00000000 RCX 00007FFB RCX 000007FFB	001600^^ 001010 ADDCD3	Modify value	Enter	Î
	RBP 00000000 BSP 00000000 RST 00000000	000000 0004040 00060FD 1000003	Increment value	+	I
	RDI 0000000	000003	Decrement value		
		000	Zero value	0	► ►
Ļ	Default (x64 fastcal		Follow in Dump		Unlocked
	2: rdx 00000000 3: r8 00000000	0000000 0060FD1	Follow in Dump	+	i
	4: r9 00000000 5: [rsp+28] 000	0000000	Follow in Disassemb	ler	
		•⊙	Follow in Memory Ma	ар	Ŧ
060FD58	000000000040155/	A reti	Copy value	Ctrl+C	
060FD60 060FD68	0000000000160000 0000000000000000000000		Copy all registers		Î.
060FD70 060FD78 060FD80	00000000000000000000000000000000000000		Highlight	н	
060FD88 060FD90 060FD98	00000000000000000000000000000000000000		Undo		
060FDA0 060FDA8 060FDB0	00000000001D1550 00007FFBAC0CB530 00000000000000000	0 0 kerr 🗖 1	Copy old value: 000	000000000000000000000000000000000000000	
060FDB8	00000000001D14B	0 &"C:\\User	s\\Lenny\\Deskto	op\\malware\\cob_99\\99	.exe"

By clicking "Follow In Dump", we can observe the contents of the dump in the bottom-left window.

We can note here that the buffer is empty and contains only 00.

🛄 Dump 1	🛄 Dump 2	🛄 Dump 3 🛛 🛄	📕 Dump 4 🛛 💭 Dump 5 🛛 🦃 🕈	Watch 1 Locals 🎾 Struct
Address	Нех		ASCII	
0000000000160000 000000000160010 00000000	$\begin{array}{c ccccccccccccccccccccccccccccccccccc$	$\begin{array}{cccccccccccccccccccccccccccccccccccc$	000 000 <td>Empty Buffer Created by VirtualAlloc</td>	Empty Buffer Created by VirtualAlloc
0000000001600A0 0000000001600B0 0000000001600C0 0000000001600D0 000000001600E0 0000000001600E0	$\left \begin{array}{cccccccccccccccccccccccccccccccccccc$	$\begin{array}{cccccccccccccccccccccccccccccccccccc$	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	

Monitoring Memory With Hardware Breakpoints

VirtualAlloc has finished creating an empty buffer and we have successfully found it.

We can now go ahead and monitor for changes to this buffer by creating a Hardware Breakpoint.

A hardware breakpoint can be created by selecting the first byte in the memory dump and Right Click -> Breakpoint -> Hardware, Access -> Byte



From here we can allow the malware to continue to execute.

We should soon see our hardware breakpoint triggered. With an FC byte contained in the first part of the buffer.

We can recall from previous blogs that FC is a very common first byte in shellcode.



At this point we want the malware to continue to fill up the buffer, but we don't want it to do anything after that.

We can go ahead and use another Execute Until Return. Which will allow the buffer to fill up. At which point we can monitor it's contents.

Below we can see the buffer after it has filled. We can see the first byte is $0 \times FC$ and there is a wininet string present in the initial bytes. From previous blogs (<u>1</u>, <u>2</u>)we know that this could indicate shellcode.



Validating Shellcode Using a Disassembler

Now that we have a reasonable assumption that the buffer contains shellcode, we can go ahead and try to disassemble it using X64dbg.

If we disassemble the code and there are no glaring errors, then there is a very high chance that we are looking at shellcode.

We can achieve this by selecting the first FC byte and Follow in Disassembler.



X64dbg will now attempt to disassemble the bytes from our buffer.

Below, we can observe the buffer disassembled in the top disassembly window. There appear to be no glaring errors, and there are valid function calls, loops and overall "normal" looking instructions.



Final Validation Using SpeakEasy Emulator

We now have a very high suspicion that the buffer contains shellcode. So we can go ahead and emulate it using Speakeasy.

We could also achieve the same thing with X64dbg, but for shellcode, this is a much more involved process that will be covered in a later blog.

To emulate the shellcode using speakeasy, we first need to save it.

We can select our first FC byte, right-click and go to Follow in Memory Map

0000000000070000 000000000140000	00000000000000000000000000000000000000	ļ	Jser Jser	\Device\HarddiskVolume3\Windows\Sy	
0000000000160000	0000000000001000		Jser Jser		
00000000000000000000000	00000000000000000	ι.	Jser	Reserved	
00000000023c000	0000000000005000	īι	Jser	PEB, TEB (3060), TEB (264)	
000000000241000	0000000001BF000	Īι	Jser	Reserved (0000000000000000)	
0000000000400000	000000000000000000000000000000000000000	1ι	Jser	99.exe	
000000000401000	000000000003000	Īι	Jser	".text"	Execut
000000000000000000000000000000000000000	0000000000001000		Icon	" data"	Traitia

From here we can save the memory buffer to a file.

I will go ahead and save my file as memdump.bin.



Emulating the Unpacked Shellcode with Speakeasy

With the shellcode buffer now saved to a file memdump.bin. We can go ahead and emulate the shellcode using Speakeasy.

We can do this with the command speakeasy -t memdump.bin -r -a x64

- speakeasy Runs the speakeasy tool
- -t Which file we want to use
- -r (Raw) Indicates that we are using shellcode
- -a x64 Indicates that our file contains 64-bit instructions. (we know this as we're using x64dbg and not x32dbg)

Upon running this command, the shellcode is emulated successfully and we are given a lot of information about it's functionality.



The Speakeasy output shows a C2 address of <u>116.62[.]138.47</u>, as well as a partial url of /8yHd.

We can also see references to a user agent of User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0; InfoPath.2; .NET CLR 2.0.50727)\r\n

(This user agent would be a great place to go hunting in proxy logs if you had them available)



Locating the Shellcode Decryption Function In Ghidra

At the point where the hardware breakpoint was first triggered, the primary executable was likely in the middle of the decryption function. We can use this information to locate the same decryption function within Ghidra.

From here, we can do some interesting things which are covered in the next 7 sections.

- Locating the Shellcode Decryption Function In Ghidra
- Identifying Decryption Routine Logic With ChatGPT
- Identifying the Decryption Key Using Ghidra
- Locating the Encrypted Shellcode Using Entropy
- Performing Manual Decoding Using Cyberchef
- Hunting For Additional Samples Using Decryption Bytes
- Creating a Yara Rule Using Decryption Code

These remaining sections are available for paid members of the site.

You can subscribe using the button below.

Paid members will receive priority access to posts about Ghidra, Static Analysis and Advanced Debugging techniques.

This post is for paying subscribers only

<u>Subscribe now</u> Already have an account? <u>Sign in</u>