Redline Stealer - Static Analysis and C2 Extraction

embee-research.ghost.io/redline-stealer-basic-static-analysis-and-c2-extraction/

Matthew

April 10, 2023



Ghidra Featured

Deep dive analysis of a redline stealer sample. I will use manual analysis to extract C2 information using a combination of Ghidra and x32dbg

Deep-dive analysis of a packed Redline Stealer sample. Utilising manual analysis and semiautomated string decryption to extract C2 information and ultimately identify the malware. In this write-up, I intentionally try to touch on as many concepts as possible in order to demonstrate practical applications and hopefully provide a better learning experience for the reader.

Quick Caveat

I realized after the initial post that this sample is actually <u>Amadey Bot</u>. The analysis and RE techniques remain equally relevant, but the sample is not actually Redline as the title suggests :)

(There is a second file in the .cab which contains <u>Redline Stealer</u>, which may explain why the initial file was semi-incorrectly marked as Redline)

I was able to determine this by researching the decrypted strings that are detailed at the end of the post.

If you're interested in how to use decrypted strings to identify or confirm a malware family. Jump to the bonus section "*Utilising Decrypted Strings To Identify the Malware Family*" of this blog.

Link Sample

The initial file can be downloaded from <u>Malware Bazaar</u> with SHA256: . 449d9e29d49dea9697c9a84bb7cc68b50343014d9e14667875a83cade9adbc60

Analysis Summary

Feel free to jump to certain sections if you are already comfortable with some of these concepts.

- Saving the file and extracting the initial .exe
- Using Entropy to identify that the initial .exe is packed
- Using a debugger to manually unpack the first payload
- Initial analysis of the unpacked payload
- Identifying interesting strings and imports
- Static Analysis to establish context of interesting strings and imports
- Utilising a debugger to analyse the String Decryption function
- Automating the String Decryption using X32dbg
- Utilising Decrypted strings to identify the malware family.

Actual Analysis

The analysis can kick off by downloading the above file and transferring it into a safe analysis machine. (I strongly recommend and personally use <u>FLARE-VM</u> for analysis)

The file can be extracted with the password infected.



Unzipping the file with the password "infected"

After successful extraction - <u>detect-it-easy</u> can be used to perform an initial analysis of the file.

This reveals that the file is a 32 bit executable. Which in this case is actually a <u>Microsoft</u> <u>Cabinet file</u>. This is essentially a .zip that can be executed as a .exe file.

Detect It Easy v3.01				_	
File name C:\Users\Milhouse\Desl	ktop\Redline2\449d9e29d49dea9697c	9a84bb7cc68b503	43014d9e14667875a83ca	de9adbc60.exe	
File type	Entry point		Base address		MIME
PE32 -	00406a60 >	Disasm	00400000	Memory map	Hash
PE	Export Import	Resources	.NET TLS	Overlay	Strings
Sections	TimeDateStamp Si	zeOfImage	Resources		Entropy
0005 >	2022-05-25 05:49:06	0010f000	Manife	st Version	Hex
Scan Detect It Fasy(DiF)	Endianness	Mode	Architecture	Туре	
			1500	-	
sfx	Microsoft Cab	oinet(11.0.0.99.0.1)[-]	S	
compiler	Microsoft Visual C/	′C++(2017 v.15.6)[msvcrt]	S	
archive	Microsoft Cabinet F	ile(1.03)[LZX,89.0	9%,2 files]	S	
linker	Microsoft Linker(14.13, V	íisual Studio 2017	′ 15.6*)[GUI32]	S ?	
					Options
Signatures			Deep scar	C	About
	100%	> Lo	og 228 msec	Scan	Exit

Initial Malware Analysis using Detect-it-easy

The file is similar enough to a .zip that 7-zip is able to extract the contents of the file just like a regular zip file.

I was able to use 7zip to extract the contents, creating two new exe's in the process. These are si684017.exe and un007241.exe in the screenshot below.

Name	Date modified	Туре	Size
🟮 449d9e29d49dea9697c9a84bb7cc68b50343014d9e14667875a83cade9adbc60.exe	4/10/2023 2:06 AM	Application	1,065 KB
22 449d9e29d49dea9697c9a84bb7cc68b50343014d9e14667875a83cade9adbc60.zip	4/10/2023 9:06 AM	7zFM.exe file	1,022 KB
si684017.exe	4/9/2023 3:46 AM	Application	236 KB
🥫 un007241.exe	4/9/2023 3:46 AM	Application	800 KB

Additional files after extracting initial .cab. For now, I'll focus on the si684017.exe file.

Initial Executable File

The initial is file recognized as a 32-bit exe file by detect-it-easy.

Interestingly - it was not a .NET as most <u>Infostealers</u> generally are. This means that the usual DnSpy won't be applicable here.

Detect It Easy v3.01 _ Х File name C:\Users\Milhouse\Desktop\Redline2\si684017.exe File type Entry point Base address MIME PE32 00405137 00400000 Disasm Memory map Hash Import Resources TLS ΡE Export Overlay Strings TimeDateStamp SizeOfImage Resources Sections Entropy 0004 2022-04-14 02:19:55 000a8000 Version Manifest Hex Endianness Scan Mode Architecture Туре GUI Detect It Easy(DiE) LE I386 Microsoft Visual C/C++(2008)[libcmt] compiler linker Microsoft Linker(9.0)[GUI32] Options Signatures Deep scan About Scan 100% Log 196 msec Exit

(Check out my analysis of dcrat for tips on using Dnspy)

Initial file analysis using Detect-it-easy

During initial analysis, I always want to determine if the file is potentially a <u>packed loader</u> rather than a final payload. If I have reason to suspect a packed payload, I typically focus on unpacking rather than strings or other static analysis.

A packed sample will typically contain areas of significantly high entropy.

To determine areas of entropy - I utilized the Entropy Graph feature within Detect-it-easy.



Malware Entropy Analysis Using Detect-it-easy

This revealed a *very* area of high entropy within the file. This is a strong indicator that the file is a packed loader and not the final payload.

In situations like this - I proceed to focus on unpacking the file.

Since this is a "regular" exe file and not a .NET-based file - I proceeded to unpack the file using X32dbg.

Unpacking Using X32dbg

When a standard-exe-based loader unpacks a file, it typically uses a combination of VirtualAlloc, VirtualProtect and CreateThread. These functions allow the malware to allocate new sections of memory that can be used to store and execute the unpacked payload. Advanced malware will heavily obfuscate these functions and/or avoid using them completely. But in 90% of cases - the previously mentioned functions are relevant. (Check out my <u>blog on API hashing</u> for how this obfuscation can be done)

In most malware - We can set breakpoints on the VirtualAlloc and VirtualProtect function calls and monitor the results using <u>Hardware Breakpoints</u>. This will alert when the newly allocated buffer is accessed, from there it is generally simple to obtain the decoded payload.

To summarise this:

- Identify a Function of Interest (In this case VirtualAlloc)
- Create a breakpoint to monitor VirtualAlloc
- Obtain the Memory Buffer created by VirtualAlloc
- Use a Hardware Breakpoint to alert when the new memory buffer is accessed
- Allow the malware to execute until the buffer is filled
- Save the buffer to a file

I've previously written a thread on how to use Hardware Breakpoints to unpack Cobalt Strike Loaders. You can check it out <u>here</u>.

Loading the File into X32dbg

To initiate this process - I dragged the file into a debugger (<u>x32dbg</u>) and allowed the file to execute until the <u>Entry Point</u>. This can be done by loading the file and *once* clicking the F9 button.



Viewing the Entrypoint using a Debugger (x32dbg)

Creating The Breakpoints

Breakpoints were then required in order to inspect the appropriate VirtualAlloc function.

Note that in this case - the primary interest is in the output (or return value) of VirtualAlloc. The relevance of this is that we care about the data at the "end" of the breakpoint, and not at the moment where the breakpoint is hit.

If that's confusing then let's just see it in action (it's always confusing the first dozen times)

Set two breakpoints using the following commands

bp VirtualAlloc, bp VirtualProtect



Setting a breakpoint on VirtualAlloc using x32dbg

Hit F9 (Continue) again, allowing the malware to execute until a breakpoint is hit.

A breakpoint is immediately hit on the VirtualAlloc function

EIP	76BDF9F0 76BDF9F0	8BFF	mov edi,edi	VirtualAlloc	A		Hide FPU	
	7680-9F3 7680-9F3 7680-9F5 7680-9F6 7680-9F6 7680-9F0 7680-9F7 7680-9F7 7680-8F4 7680-8F4 7680-804 7680-804 7680-804 7680-804 7680-804 7680-804 7680-804	386C 50 50 57F25 6c13c476 cc cc cc cc cc cc cc cc cc cc cc cc cc	poor esp poor e	ЭМР.&VirtualAlloc		EAX 0019EA38 EBX 0000000 ECX 0019EA38 EDX 006E423D EBP 0019E7eC ESI 0019E7eC ESI 0019E7eC ESI 0019E7eC ESI 0019EA38 EDI 0000011C EFP 76BDP9F0 EFLAGS 00000300 EFLAGS 00000300 CF 0 F 0 AF 0 OF 0 SF 0 DF 0	<pre>dReadConsoleInputw> dReadConsoleInputw> "1]1" dReadConsoleInputw> L'6' decenteI32.VirtualAlloc> </pre>	Î
	768DFA0A 768DFA0B 768DFA0C 768DFA0C 768DFA0D 768DFA0E					LastError 000000 LastStatus C00001	57 (ERROR_INVALID_PARAMETER) 00 (STATUS_VARIABLE_NOT_FOUN	D)
	76BDFADF 76BDFA10 76BDFA13 76BDFA13 76BDFA13 76BDFA18 76BDFA14 76BDFA14 76BDFA14	CC 88FF 55 88EC 88555 0C 33C0 8840 08 50 64 02	<pre>int3 adi, adi moveh ebp, move bb,esp move ds.werdptr ss:[ebp+C] xone es.exa xone es.e</pre>		,	Default (stdcall) 1: [esp+4] 0000000 2: [esp+8] 000326 3: [esp+c] 0000100 4: [esp+10] 000000 5: [esp+14] 2C3CE0	0 00000000 7 0003A267 0 00001000 40 00000040 37 2c3cE037	v v 5 A Unlocked
🀇 Dump 1	🌜 Dump 2 🛛 🐇	Dump 3 🐇 Dump 4	I Dump 5 🕼 Watch 1 🔏 Locals	Struct	0019E7CC 0019E7D0	006C7C31 return t		A
Address Hex 77681000 16 00 77681010 00 00 77681010 00 00 77681020 00 00 77681030 00 00 77681040 00 00 77681040 00 07 77681060 84 00 77681060 80 84 77681080 00 00 77681080 00 00 77681080 00 00 77681080 00 00 77681080 00 00 77681080 00 00 77681080 00 00 77681080 00 00 77681080 00 00 77681080 00 00 77681080 00 00	18 00 FO 7D 68 02 00 00 5E 68 08 00 88 7F 68 08 00 80 7F 68 08 00 80 7F 68 18 00 88 81 68 60 77 10 47 78 78 77 50 44 78 68 77 10 47 10 05 00 95 71 40 78 10 00 00 57 14 01 00 00 57 14 01 01 03 59 65 D 14 01 13 35 96 50 B0	$\begin{array}{cccccccccccccccccccccccccccccccccccc$	ASCTI 77 Artw P hw 77 Artw F. hw 77 hw frw 77 hw frw 77 hw frw 77 hw P kwp] kw 77 hw P kwp] kw kw kw 77 hw P kwp] kw kw kw 77 hw P kwp] kw kw kw kw 77 hw P kwp] kw kw kw kw kw kw kw kw		00198704 00198705 00198705 00198726 00198724 00198724 00198724 00198726 00198726 00198776 00198776 00198776	0003.2667 00001000 000000040 2535603 00657768 00657778 766824010 876824010 876824010 876824010 87682400 87682400 87682400 87682400 87682400 877868950 87682400 877868950 87682400 877868950 87786850 87786850 87786850 8778680 8778600 877860000000000	o 006C7F73 from 006C78DD .766E4010 658950 .768E2420	
Command: Commans								Default 👻
Paused INT3 b	oreakpoint at <kernel3< td=""><td>32.VirtualAlloc> (76BDF9F0)!</td><td></td><td></td><td></td><td></td><td></td><td>Time Wasted Debugging: 0:02:40:49</td></kernel3<>	32.VirtualAlloc> (76BDF9F0)!						Time Wasted Debugging: 0:02:40:49

Triggering a breakpoint on VirtualAlloc

The primary purpose of VirtualAlloc is to allocate memory and return an address to the newly allocated buffer. This newly allocated memory is contained in the EAX register when the function is *completed*.

TLDR: Since I'm only interested in that buffer - I utilized the Execute Until Return or CTRL+F9 to jump straight to the end of the function and obtain the result.

Deb	ug	Tracing	Plugi	ins	Favou	rite	
⇒	Ru	n		F9			
⇒	Ru	n until seleo	tion	F4			
II	Pa	use		F12			
ΰ	Re	start		Ctrl-	+F2		
	Clo	se		Alt+	F2		How to "Execute Until Return"
÷	Ste	ep into		F7			
æ	Ste	ep over		F8			
1	Exe	ecute till ret	urn	Ctrl-	⊦F9		
→ £	Ru	n to user co	de	Alt+	F9		
۲	Ad	vanced				Þ	

using x32dbg

Allowing the malware to Execute Until Return - provides an EAX register containing the address of the memory buffer to be used by the malware.

EAX 02250000 EBX 00000000 ECX 384B0000 EDX 02250000 EBP 0019E7EC ESP 0019E7CC ESI 0019EA38 EDI 0000011C	"1]" <&ReadConsoleInputW> L'Ĝ'		Î
EIP 75AE7A5C	kernelbase.75AE7A5C		
EFLAGS 00000246 ZF 1 PF 1 AF 0 OF 0 SF 0 DF 0 CF 0 TF 0 IF 1			
LastError 0000005	7 (ERROR_INVALID_PARAMETER)		
LastStatus C000010	U (STATUS_VARIABLE_NOT_FOUND)		• •
			• •
Default (stdcall)		▼ ▼ 5	Unlocked

Viewing the memory buffer returned by VirtualAlloc

There is nothing particularly special about EAX, it is just the standard register used for returning the results of a function.

To learn more about EAX and calling conventions - there's a great video on that from <u>OALABS</u>.

To monitor the buffer returned by VirtualAlloc, Right-Click on the returned address 02250000 address and select Follow in Dump.

This will cause the bottom-left window to display the newly-allocated memory.

The buffer of memory currently contains all 00's, as nothing has used or written to the buffer yet.



Using x32dbg (Follow In Dump) to view the contents of a memory buffer

It is important to be notified when that buffer of 00's is no longer a buffer of 00's.

To achieve this - A hardware breakpoint can be applied on the first byte of the newly allocated buffer.



creation of a Hardware Breakpoint

Once the hardware breakpoint is set - the malware can continue to execute using the F9 button.

The Hardware Breakpoint will immediately be triggered.

Address	Hex		00	00	00	00	00	00	00	00	00	00	00	00	00	ASCII
02250010		00	00	00	00	00	00	00	00	00	00	00	00	00	00	<u>e</u>
02250020		00 00	00	00	00	00	00	00	00	00	00	00	00	00	00	
02250040 02250050	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	
02250060 02250070	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	
02250080	00 00		00	00	00	00	00	00	00	00	00	00	00	00	00	
022500A0			00	00	00	00	00	00	00	00	00	00	00	00	00	
022500B0 022500C0			00	00	00	00	00	00	00	00	00	00	00	00	00	
02250000		00	00	00	00	00	00	00	00	00	00	00	00	00	00	
Command:	Comman	ds a	re c	omm	a s.	epa	rat	ed	(lik	te a	.sse	mbl	y in			ions): mov eax, ebx
Paused	Hardv	vare b	reakp	point	: (by	te, r	ead/	write	e) at	<u>022</u>	<u>500(</u>	<u>)0</u> !	L			
					_				_			_				

Triggering a Hardware Breakpoint using X32dbg

Once this happens, use CTRL+F9 (Execute Until Return, aka "just finish what you're doing now, but don't do anything else") to allow the malware to continue writing to the buffer without actually executing it.

(Utilising CTRL+F9 will cause the malware to stop at the end of the current function - preventing the execution of the rest of the malware)

Once the current function is finished - the buffer will look something like this.

Address	He	×															ASCII	
02250000	EB	03	C2	0C	00	55	8B	EC	81	EC	00	10	00	00	C7	45	ë.ÂU.ì.ìÇE	
02250010		20	UE	00	00	С/	45	Ao	00	00	40	υu	ου	00	50	rr.	ÀÇE¨@Pÿ	
02250020	FF	FF	50	8D	45	D4	50	8D	45	98	50	E8	FB	08	00	00	ŷŷP.EOP.E.Peu	
02250030	83	C4		Eð	04	00	00	00	00	00	00	00	58	89	85	6C	.A.eXI	
02250040			۲۲ ۵5	ов 6С		00		74 88		C9		EU 11	E0 01		20	45	JÜÜÜ MÀ DÈ E	
02250050	F8	8R	45	FR	89	85	58	FF	FF	FF	c7	85	70	FF	FF	FF		
02250070	6B	65	72	6E	C7	85	74	FF	FF	FF	65	6C	33	32	c7	85	kernc.tvvvel32C.	
02250080	78	FF	FF	FF	2E	64	6C	6C	83	A5	7C	FF	FF	FF	00	8D	xÿÿÿ.d11.¥ ÿÿÿ	
02250090	85	70	FF	FF	FF	50	FF	55	D4	89	45	C4	C7	85	70	FF	. pÿÿÿPÿUÔ. ĖÄÇ. pÿ	
022500A0	FF	FF	56	69	72	74	<u>C</u> 7	85	74	FF	FF	FF	75	61	6C	41	ÿÿVirtÇ.tÿÿÿualA	
022500B0	C7	85	78	FF	FF	FF	6C	6C	6F	63	83	A5	7C	FF	FF	FF	Ç.xŷŷŷlloc.¥ ŷŷŷ	
022500C0		8D	85	/0	FF	FF	FF	50	FF	/5	C4	FF	55	98	89	45	pyyyPyuAyUE	
02250000	84	C/ 75	85 61	70				50	69		/4	C/ 72	85	74				
022500E0	85	$\frac{73}{7c}$	DT DT			63	05 74	00			85	70		74	03 FF	50	lyvvct nyvvp	
02250100	FF	75	c4	FF	55	98	89	45	00 80	c7	85	70	FF	FF	FF	56		
02250110	69	72	74	c7	85	74	FF	FF	FF	75	61	6Č	46	c7	85	78	irtC.tvvvalFC.x	
02250120	FF	FF	FF	72	65	65	00	8D	85	70	FF	FF	FF	50	FF	75	ÿÿÿreepÿÿÿPÿu	
02250130	C4	FF	55	98	89	45	9C	С7	85	70	FF	FF	FF	47	65	74	Âÿ̈́ŪE.Ç.pÿÿÿGet	
02250140	56	C7	85	74	FF	FF	FF	65	72	73	69	C7	85	78	FF	FF	VÇ.tÿÿÿersiÇ.xÿÿ	
02250150	FF	6F	6E	45	78	C7	85	<u>7</u> C	FF	FF	FF	41	00	00	00	8D	yonExÇ. yyyA	
02250160	85	70				50		/5	C4		55	98	89	45	BS	C/	.pyyyPyuAyUE C	
02250170	600	61	7/	C7		24 7Ω	00			65	00 50	74		гг с7		70	pate vüüüoBroc	
02230180		<u> </u>	74	67	65	73	72	22	65	80	00	ΩΠ		70	55			
Command:	Comr	land	s a	re (comn	າສ ຣ	epa	rat	ed	(⊥i}	<u>ce</u> a	ISSE	mbl	v i	nsti	ruct	ions): mov eax, ebx	

Identifying a Memory Buffer containing Shellcode

Unfortunately - the first buffer does not contain an unpacked PE file. It does contain a large buffer of shellcode which is used to unpack the next section using another VirtualAlloc.

If the file was sucessfully unpacked - it would typically look something more akin to this

He	K															ASCII
4D	5A	90	00	03	00	00	00	04	00	00	00	FF	FF	00	00	MZÿÿ
B8	00	00	00	00	00	00	00	40	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	F8	00	00	00	ø
0E	1F	ΒA	0E	00	в4	09	CD	21	в8	01	4C	CD	21	54	68	º´.Í!、.LÍ!Th
69	73	20	70	72	6F	67	72	61	6D	20	63	61	6E	6E	6F	is program canno
71	20	62	65	20	70	75	6E	20	60	65	20	11	1 =	E 2	20	+ ha nun in Doc

Identifying an unpacked PE file in a memory buffer

In this case there is only shellcode in the buffer. You can typically determine that the buffer is shellcode by the presence of the EB (jmp) byte. You can also confirm suspected shellcode by inspecting the instruction using Right-Click -> Follow in Disassembler.

If the code disassembles without errors (No glaring red sections) - it is highly likely to be shellcode.



Using x32dbg to validate shellcode contained in a memory buffer

At this stage - the shellcode *could* be dumped into a file for further analysis.

However, It is often better to allow the shellcode to execute. Malicious actions taken by the shellcode will often trigger the same breakpoints intended for the "original" malware.

Obtaining The Unpacked Payload

Hitting F9 (Continue) to allow the malware to execute - another breakpoint is hit on VirtualAlloc

<u></u>	CPU 🔮	Log	Notes	🥼 Bre	akpoints 🛛 🐇	Memory Map	🍹 Call Stack	🏮 SEH	👼 Script	🥼 Symbols	🍹 Source
EIP			BBDF9F0 BBDF9F2 BBDF9F5 BBDF9FC BBDF9FC BBDF9FC BBDF9FC BBDF9FC BBDFA00 BBDFA00 BBDFA01 BBDFA03 BBDFA04 BBDFA06 BBDFA06 BBDFA08 BBDFA08 BBDFA08 BBDFA08 BBDFA08 BBDFA08 BBDFA08 BBDFA08 BBDFA10 BBDFA15 BBDFA18 BBDFA18 BBDFA18 BBDFA18 BBDFA18	8B FF 55 5D FF 25 6C1 CC SB FF S5 BE EC BB 55 CC CC CC CC CC CC CC	. <u>3c476</u>	<pre>mov edi,edi push ebp mov ebp,esp pop ebp jmp dword ptr int3 int3 int3 int3 int3 int3 int3 int3</pre>	dg:[<&virtua] dptr_ss:[ebp+4 dptr_ss:[ebp+4	LA11oc>] C] 8]	VirtualAl JMP.&Virt	loc ualAlloc	

Viewing VirtualAlloc function in a debugger (x32dbg)

Using the same trick of Execute Until Return, Select EAX and Right-Click -> Follow in Dump, the second allocated buffer can be obtained.



Using x32dbg to locate another memory buffer returned by VirtualAlloc Another Hardware Breakpoint will need to be set at the start of the buffer.

	_			_			
	۵	Breakpoint		•	۲	Hardware, Access	🕨 🤚 Byte
	-46	Find Pattern	Ctrl+B		-	Hardware, Write	🕨 🐗 Word
	4	Find References	Ctrl+R		\$	Hardware, Execute	👼 Dword
	4	Sync with expression	S		۲	Memory, Access	▶ d ptr ss:[ebp
	١	Allocate Memory			@	Memory, Read	r ds:[edi],0
	۲	Go to		×	0	Memory, Write	d ptr ss:[ebp
	\$	Hex		×	۲	Memory, Execute	•
🂰 Dump 1	۲	Text		►	U I	ump 4 🐡 Dum	p ၁ 🥼 Watch 1
Address He	<u> </u>	Integer		×	00	ASCI 00 00 00 00	I ·····
02290020 00 02290030 00		Float		×	00 0	00 00 00 00 00 00 00 00 00	
02290040 00 02290050 00 02290060 00	4	Address			00 00 00 00 00	$\begin{array}{cccccccccccccccccccccccccccccccccccc$	
02290070 00 02290080 00	¢	Disassembly			00 00	00 00 00 00 00 00 00 00	· · · · · · · · · · · · · · · · · · ·
02290090 00 022900A0 00 022900B0 00	00	00 00 00 00 00 00 00 00 00 00 00 00 00 0		00 00	00 00 00 00 00 00 00 00 00 00 00 00 00	$\begin{array}{cccccccccccccccccccccccccccccccccccc$	· · · · · · · · · · · · · · · · · · ·

Creating another Hardware Breakpoint on the memory address Allowing the malware to continue to execute - the hardware breakpoint is hit. This time containing a promising M. (First half on an <u>MZ header</u>)

(Side note that my debugger suddenly crashed here and had to be restarted - hence the slight change of address in future screenshots)

Hardware breakpoint (byte, read/write) at 02160000!

Memory buffer - potentially containing an unpacked pe-file payload Allowing the malware to continue to execute - A complete MZ/PE file can be found. At this point, the unpacked file has been successfully loaded into memory.

Address	Hex													ASCII	
02160000	4D 5A	90	00	03	00 00	00	04	00	00	00 FI	FF	00	00	MZÿÿ	
02160010	B8 00	00	00	00	00 00	00	40	00	00	00 00	00 00	00	00	······@·····	
02160020		00						00	00		J UU D 01	00	00		
02160040	00 00 0F 1F	BA			R4 09		21	в8	01		21	54	68	° ´f! I f! Th	
02160050	69 73	20	70	72	6F 6	72	61	6D	20	63 6	Í ĜĒ	6E	6F	is program canno	
02160060	74 20	62	65	20	72 7	6E	20	69	6E	20 44	1 4F	53	20	t be run in DOS	
02160070	6D 6F	64	65	2E	0D 00	0A	24	00	00	00 00	00 0	00	00	mode\$	
02160080	5D 0E	FC	4D	19	6F 9	1E	19	6F	92	1E 19	9 6F	92	1E].üM.ooo	
02160090	42 07	91			6F 94		42	07	97		0 6F	92		BOBO	
021600A0	42 07	90			0F 9. 6F 9.	16		02	90		5 0F	92	16	\dot{T} \dot{O} \dot{T} 50	
021600c0	42 07	93	1 _F	16	6F 9	1E	19	6F	93	1E B	4 6F	92	1E	B00°0	
021600D0	82 01	9B	1F	18	6F 92	1E	82	01	6D	1E 18	3 6F	92	1E	mo	
021600E0	82 01	90	1F	18 (6F 92	1E	52	69	63	68 19	9 6F	92	1E	Rich.o	
021600F0	00 00	00	00	00	00 00	00	00	00	00	00 00	00	00	00		
02160100	50 45	00	00	4C	01 0	00	97		3T 2T	64 0) UU	00	00	PEL1d	
02160120	00 00 00 FF	00				00	25	60	01		2ם ע 10 ו	02	00	1 %	
02160130	00 D0	02	ŏŏ	ŏŏ i	00 40	00	00	10	00	00 00	02	ŏŏ	ŏŏ		
02160140	06 00	00	00	00	00 00	00	06	00	00	00 00	00	00	00		
02160150	00 E0	03	00	00	04 00	00	00	00	00	00 02	2 00	40	81	.à@.	
02160160	00 00	10	00	00	$10^{-}0($	00	00	00	10	00 00	10^{-10}	00	00		

A complete Pe-file written to the memory buffer

Saving the Unpacked File

To save the unpacked file - Right-Click on the start address and select Follow in Memory Map



SCPU EIP	4	ø	Follow in Memory Map	
		£	Label Current Address :	
		0	Watch DWORD	
		ø	Modify Value Space	
		0	Breakpoint	•
	-	£	Find Pattern Ctrl+B	
		ø	Find References Ctrl+R	
		£	Sync with expression S	How
		١	Allocate Memory	
		<u>@</u>	Go to	•
		\$	Hex	•
🂰 Dump	1	١	Text	•
Address 02160000 02160010	le D	¥	Integer	► C
02160020 02160030	00	۲	Float	•
02160040 02160050 02160060	0E 69 74	4	Address	é
02160070 02160080	6D 5D	£	Disassembly	
02160090 021600A0	42 42	ŏ, 07	96 1F 0B 6F 92 1E CC 02	96 1

to save a memory buffer using x32dbg

This will reveal the location where the buffer was allocated. The entire memory buffer can then be saved by using Right-Click and Dump Memory to File



button used to dump the memory to a file using x32dbg

The file can now be saved as unpacked.bin (or any other file name of choosing)

File name:	unpacked.bin
Save as type:	Binary files (*.bin)
∧ Hide Folders	
One off the state of the state	

Specifying a name for the unpacked file

Initial Analysis - Unpacked Payload

The file is a 32-bit executable with no (recognized) packers or obfuscation.

Detect It Easy v3.01	l			_								
File name												
C:\Users\Milhouse\Des	ktop\Redline2\unpacked.bin											
File type	Entry point Base address											
PE32 -	00416025 >	Disasm	00400000	Memory map	Hash							
PE	Export Import	Resources	.NET TL	S Overlay	Strings							
Sections	TimeDateStamp	SizeOfImage	Resource	s	Entropy							
0005 >	2023-04-08 14:49:43	0003e000	Manif	est Version								
Scan	Endianness	Mode	Architecture	Туре	Hex							
Detect It Easy(DiE)	▼ LE	32	I386	GUI								
compiler	Microsof	t Visual C/C++(-)[-]		S								
linker	Microsoft L	inker(14.24**)[GUI3.	2]	S ?								
					Options							
Signatures			Deep sca	in Coon	About							
	100%	> Lo	g 182 msec	Scan	Exit							

Initial analysis of suspected unpacked payload using detect-it-easy

The entropy graph does not contain any areas of significantly high or flat entropy - suggesting that the file is not packed and does not contain any additional payloads.

De Entropy						- 0	×
Type PE32 -	Total 6.29560	78% Status not packed	Off	set 00000000	Size 0003a000	R	eload
Entropy Bytes Regions							
PE Header Section(0)['.text'] Section(1)['.rdata'] Section(2)['.data'] Section(3)['.rsrc']	Name		Offset 00000000 0000400 0002b600 00035200 00036a00	Size 00000400 0002b200 00009c00 00001800 00000200	Entropy 2.76415 6.45620 4.84964 1.40114 4.72050	Statu: not packed not packed not packed not packed not packed	5 °
8 7 6 5 4 3 1 0							
Ó	50,000	100,000	150,0	00	200,000		250,000 Gave

Additional Entropy Analysis - Suggesting no hidden payloads - No significant areas of entropy

Since this was potentially a final payload - I checked the strings for any unobfuscated information.

This revealed some base64 encoded data - but I wasn't able to successfully decode it.

The base64 encoding has likely been combined with additional obfuscation.

<mark>ष</mark> ि Strings				- 🗆 X
0x0000000	00 - 0x00039fff (0x0003	3a000)		ANSI Unicode 15 🗢 Search
	Offset 💌	Size	Туре	String
79	00030a3c	00000010	А	76XoROUVDkN1OQ==
80	00030a60	00000028	А	G4FBRNIRRQssJ0FeUPqLNLICNG5DXypnDG1XL D=
81	00030aa8	00000044	А	P41JNwgsMiN IWehdhdw4SQX2J ffmGl97FflUYp5kNr3FSjdiSm4SW ZrRkX2U UU==
82	00030af0	0000005c	А	P41JNwgsMiN IWehdhdw4SQX2J ffmGl97FflUYp5kNr3FSjdiSm4SW
83	00030b68	00000018	А	T6TnAtbuADBKGEeQJxdwDTvd
84	00030b84	000003c	А	P41JNwgsMiN IWehdhdw4SQX2J ffmGl97FflUYp5kNr3FSjdiSm4SW ZrRk
85	00030bd4	00000014	А	DG1HRNAc6EMdA1KMJxN=
86	00030c00	00000058	А	P41JNwgsMiN IWehdhdw4SQX2J ffmGl97FflUYp5kNr3FSjdiSm4SW
87	00030c5c	00000014	A	EZNWJMMHMi1DHUqDKL==
88	00030d20	00000018	А	T7BoRt8b4Ep5N2qndx h4wo5
89	00030dec	00000010	А	86 o4xzqEgXh1Gp=
90	00030e00	00000010	А	66NB4dYjExArOGqq
91	00030e14	0000001c	А	M6N3LdlR3VRiJ3exeB0qKMWjgA==
92	00030e34	00000010	А	PLByRUM94SJe3Gya
93	00030e48	00000014	А	LZREMST8M01j3H7fdhZ=
94	00030e6c	00000014	А	N6xC5xYp50l2xEqfZb==
95	00030e90	00000014	А	PKxxRxH8M0Ng3XCneCp=
96	00030ea8	00000010	А	MK1m6xbpAD6iNg==
97	00030ec4	00000018	А	HDQzNxbRQUpQOWGZdhqX7G==
98	00030ee0	00000010	А	Lqd3RxYdRUXhOXB=
99	00030f18	00000010	A	Q6dxJxYdRUXhOXB=
Filter	00000/0			
Filter				Save

Base64 Encoded Strings contained within the malware file Failing to decode the "base64"

Recipe			İ Input
Fork		⊘ 11	DG1HRNAc6EMdA1KMJxN= EZNWJMMHMi1DHUqDLL==
Split delimiter \n	Merge delimiter \n	Ignore errors	MK1m6xbpAD6iNg==
			явс 58 📻 3
From Base64		⊘ Ⅱ	Output
Alphabet A-Za-z0-9+/=		Ŧ	۰ ۴·mGDD%èC% ^s ‰R∙''s
Remove non-a	alphabet chars	Strict mode	°、•V\$Ã¹、2-C [©] J•(Ø•fë [®] √é [®] >¢6

Cyberchef - Failure to decode the base64 strings - signs of additional obfuscation

Close

Import Analysis

Imported functions are an additional valuable source of information. Especially for suspected unpacked files.

The imported functions referenced capability that suggested the file can download data and make internet connections.

Since these functions need C2 information in order to work, this is a good sign that the C2 config may be contained within this file.

	iginalFirstThu	imeDateStam	orwarderChai	Name	FirstThunk	Hash	
0	00035f70	00000000	00000000	00036372	0002d024	6964c580	KERNEL32.dll
	00035f4c	00000000	00000000	00036414	0002d000	5455aa3b	ADVAPI32.dll
2	0003611c	0000000	00000000	0003645a	0002d1d0	0f35467b	SHELL32.dll
3	00036130	0000000	0000000	00036500	0002d1e4	b794cd2a	WININET.dll
	Thunk		Ordinal	Hint		_	Name
0	Thunk 00036466		Ordinal	Hint 0078	HttpOpenRequest	tA	Name
0	Thunk 00036466 000364ec		Ordinal	Hint 0078 00ce	HttpOpenRequest InternetReadFile	:A	Name
0 1 2	Thunk 00036466 000364ec 000364d8		Ordinal	Hint 0078 00ce 009b	HttpOpenRequest InternetReadFile InternetConnectA	:A	Name
0 1 2 3	Thunk 00036466 000364ec 000364d8 000364c4		Ordinal	Hint 0078 00ce 009b 007f	HttpOpenRequest InternetReadFile InternetConnectA HttpSendRequest.	A	Name
0 1 2 3 4	Thunk 00036466 000364ec 000364d8 000364c4 000364ae		Ordinal	Hint 0078 00ce 009b 007f 0095	HttpOpenRequest InternetReadFile InternetConnectA HttpSendRequest. InternetCloseHand	A dle	Name
0 1 2 3 4 5	Thunk 00036466 000364ec 000364d8 000364c4 000364ae 0003649e		Ordinal	Hint 0078 00ce 009b 007f 0095 00c6	HttpOpenRequest InternetReadFile InternetConnectA HttpSendRequest. InternetCloseHand InternetOpenA	A dle	Name
0 1 2 3 4 5 6	Thunk 00036466 000364ec 000364d8 000364c4 000364ae 000364ae 0003649e		Ordinal	Hint 0078 00ce 009b 007f 0095 00c6 00c9	HttpOpenRequest InternetReadFile InternetConnectA HttpSendRequest InternetCloseHand InternetOpenA InternetOpenW	A dle	Name
0 1 2 3 4 5 6 7	Thunk 00036466 000364ec 000364d8 000364c4 000364ae 000364ae 0003648e 0003647a		Ordinal	Hint 0078 00ce 009b 007f 0095 00c6 00c9 00c7	HttpOpenRequest InternetReadFile InternetConnectA HttpSendRequest InternetCloseHand InternetOpenA InternetOpenW InternetOpenUrIA	A dle	Name
0 1 2 3 4 5 6 7	Thunk 00036466 000364c2 000364d8 000364c4 000364ae 0003649e 0003648e 0003647a		Ordinal	Hint 0078 00ce 009b 007f 0095 00c6 00c9 00c7	HttpOpenRequest InternetReadFile InternetConnectA HttpSendRequest InternetCloseHand InternetOpenA InternetOpenW InternetOpenUrIA	A A dle	Name

Malware Import Analysis Using Detect-it-easy

Ghidra Analysis

At this point I decided to analyze the file further using Ghidra. My plan was to utilise Ghidra to gather more information on the suspicious imports related to c2 connections InternetReadFile, InternetConnectA, HttpSendRequestA etc.

In addition to this - I wanted to investigate the suspicious "base64" strings identified with detect-it-easy.

To investigate both - I intended to utilise cross references or X-refs to observe where the strings and imports were used throughout the code. From here I hoped to find arguments passed to the internet functions (hopefully containing a C2), or to find the logic behind the function that accesses the base64 encoded strings.

To Summarise - My plan was to Utilise Ghidra to ...

- Investigate the suspicious strings which function are they passed to? what does that function do with them? Can I trace the input and output of that function?
- Investigate Suspicious Imports Check where the imports were used, and what arguments were being passed. Can I set a breakpoint and view the decrypted C2's?

String Searching with Ghidra

I took the first approach first, using Ghidra to search for strings within the file.

🥩 Search For Strings			×
Require Null Termination Pascal Strings	Minimum Length: Alignment:	5 [1 [)ec)ec
	Word Model:	StringModel.sng	
Memory Block Types Loaded Blocks All Blocks	Selection Scope Search All Search Selection	on	
Sear	ch Cancel		

Searching for Strings Using Ghidra

By filtering on ==, I was quickly able to narrow the results down to the previously identified base64 strings. This was not all relevant strings but was a solid starting point.

Defined	Location	🖹 Label	Code Unit	String View	String Le	ingth	Is Word
Ā	004326d8	s_ErtmHH==_004326d8	ds "ErtmHH=="			9	false
Ā	004326e4	s_ErNxHH==_004326e4	ds "ErNxHH=="	"ErNxHH=="	string	9	false
Ā	004326f0	s_EqJwHH==_004326f0	ds "EqJwHH=="	"EqJwHH=="		9	false
Ā	004326fc	s_Eqx5HH==_004326fc	ds "Eqx5HH=="	"EqxSHH=="	string	9	false
Ā	00432708	s_Eqp5HH==_00432708	ds "Eqp5HH=="			9	false
Ā	00432714	s_Eq1qHH==_00432714	ds "EqlqHH=="	"Eq1qHH=="	string	9	false
Ā	00432754	s_00xs48==_00432754	ds "00xs48=="	"O0xs48=="	string	9	false
Ā	00432760	s_6LJ35urmDu==_0043	ds "6LJ35urmDu=="	"6LJ35urmDu=="	string	13	false
Ā	00432814	s_M6N3LdIR3VRiJ3exe	ds "M6N3LdIR3VRiJ3exeB0qKMWjgA=="	"M6N3LdIR3VRiJ3exeB0qKMWjgA=="	string	29	false
Ā	0043286c	s_N6xC5xYp50l2xEqfZb	ds "N6xC5xYp5012xEqfZb=="	"N6xC5xYp50l2xEqfZb=="	string	21	false
Ā	00432884	s_MZFINr==_00432884	ds "MZFINr=="	"MZFINr=="	string	9	false
Ā	004328a8	s_MK1m6xbpAD6iNg=	ds "MKlm6xbpAD6iNg=="	"MK1m6xbpAD6iNg=="	string	17	false
Ā	004328c4	s_HDQzNxbRQUpQOW	ds "HDQzNxbRQUpQOWGZdhqX7G=="	"HDQzNxbRQUpQOWGZdhqX7G=="	string		false
Ā	0043292c	s_HHwBEqTSFh51DQ=	ds "HHwBEqTSFh51DQ=="	"HHwBEqTSFh51DQ=="	string	17	false
Ā	00432a40	s_GWSQu8==_00432a40	ds "GWSQu8=="	"CWGQu0"	string	9	false
Ā	00432a58	s_GqhzRn==_00432a58	ds "GqhzRn=="	"GqhzRn=="	string	9	false
Ā	00432ab0	s_P5dWNvYEPCFY2nCjc	ds "P5dWNvYEPCFY2nCjciWA4SWXhq5iY2KQS	"P5dWNvYEPCFY2nCjciWA4SWXhq5iY2KQSIFy4eUp40p F22rdC0XRNAL30Xb1ECl71t46xYpLkxqOQ=="	string	81	false
Ā	00432bfc	s_QqdnRNbAJy==_004	ds "QqdnRNbAJy=="	QqunRNbAy=="	string	13	false
Ā	00432cbc	s_HnsAGH==_00432cbc	ds "HnsAGH=="	"HnsAGH=="	string	9	false
Ā	00432cc8	s_HnsBE8==_00432cc8	ds "HnsBE8=="	"HnsBE8=="	string	9	false
Ā	00432cd4	s_HnsAF8==_00432cd4	ds "HnsAF8=="	"HnsAF8=="	string	9	false
Ā	00432d28	s_L4xGLwP8Ae==_004	ds "L4xGLwP8Ae=="	"L4xGLwP8Ae=="	string	13	false
Ā	00432dac	s_DmspB_ER3UTi13OYJ	ds "DmspB ER3UTi130YJyJdB9Pd4KRiMA=="	"DmspB ER3UTi13OYJyJdB9Pd4KRiMA=="	string		false
Ā	00432df0	s_DGQpAr==_00432df0	ds "DGQpAr=="	"DGQpAr=="	string	9	false
Ä	00432e14	s_G0N7RNQS6Eds1nutc	ds "GON7RNQS6Eds1nutcBqg7Isv40X1hGKp6	"G0N7RNQS6Eds1nutcBqg7lsv40XlhGKp606xRNT8DSRm1GNeJb=="	string	53	false
Filter:	and the second					× 💿	章 🕇
_							

Locating base64 strings using Ghidra

I double clicked on one of the larger strings, taking me to it's reference within the file.



From here I could hit CTRL+SHIFT+F to find references to this string. Alternatively you could Right Click -> References -> Show References to Address

00432aad 00	??	00h		🥩 Edit Help	References to s_P5dW	'NvYE — 🗖	x נ
00432aae 00							
00432aaf 00				References to s_P5	dWNvYEPCFY2nCjciW.	- <u>></u> 🖻 🏵 🗏 I	<u> </u>
				Locati 🕒 Label	Code Unit	Context	
	s_P5dWNvYB	EPCFY2nCjciWA4SWXhq5iY	2KQ_0 XREF[1]:	00401b82	PUSH s_P5dWNv	Y DATA	
00432ab0 50 35 6							
57 4e 7							
59 45 5							
00432b01 00							
00432b02 00							
00432b03 00							🔊 🚍 - 🛛

Using Ghidra to locate Cross-references from strings

Clicking on the one available reference - reveals an undefined function acting upon the string.



Encountering an Undefined Function in Ghidra

By clicking on the first address of the function and hitting F, we can define a function at the current address.



Defining a Function in Ghidra

After defining a function - the decompiler output now looks much cleaner.



Viewing a new function in Ghidra - an obfuscated string can be seen From here we can enter the function at FUN_00414550 and investigate.

The function contains a bunch of c++ looking junk which was difficult to analyse - so I decided to take a slightly different approach.

Viewing a suspicious function using Ghidra

I checked the number of cross references on the FUN_00414550 function. A high number of cross references would indicate that the function is responsible for decoding more than just this encoded string.

If the same function is used for all string related decryption, then perhaps a debugger and a breakpoint is the better approach.

At minimum - a debugger will at least confirm the theory that this function is related to string decryption.

String Decryption Via X32dbg

I decided to investigate the string decryption using X32dbg.

To do this - I would need to set a breakpoint on the function that I suspected was responsible for string decryption.

Attempting to copy-and-paste the address directly from Ghidra will likely result in an error as the addresses may not align.

Syncing Addresses with Ghidra and X32dbg

To Sync the Addresses between Ghidra and X32dbg. We need to find the base of our current file. This can be found in the memory map and in this case is 003e0000. Although it may be different for you.

۹	CPU	۵	Log	٥	Notes	0	Breakpoints	🔮 Me	emory M	ap 🏅	Call Stack	-A	SEH	۹	Script	4	Symbols	۲	Source
Addr	<u>ess</u>	Size	P	arty	Info					Content				туре	Protec	tion	Initial		
003E	0000	00001	000 🔛	User	unpac	ked.	bin							IMG			ERWC-		
UUDE	1000	0002C	000 🔮	User	".te	xt"				Executab	le_code			IMG	ER		ERWC-		
0040	D000	0000A	000 📑	User	rc	lata"				Read-on]	y initializ	ed da	ita	IMG	-R		ERWC-		
0041	7000	00003	000 📑	User	". da	ta"				Initiali	zed data			IMG	-RW		ERWC-		
0041	A000	00001	000 🔮	User	".rs	rc"				Resource	s j			IMG	-R		ERWC-		
0041	B000	00003	000 📑	User	".re	loc"				Base rel	ocations			IMG	-R		ERWC-		
0066	0000	00001	000 📑	User										MAP	-R		-R		
0067	0000	00001	000 📑	User										MAP	-R		-R		

How to identify a base address in a debugger (x32dbg)

From here we can select the memory map within Ghidra.



How to use Ghidra to Sync a Memory Address Then select the Home button

🥩 File	Edit Help													
🔡 Ko e														
🛄 Memory	Map - Image Ba	ase: 0040000)							++	∎∓±	� ×	<u></u>	
Memory Blocks														
Name	Start 🔁	End	Length	R	W X	Volatile	Overlay	Туре	Initialized	Byte Source	Source	Com	iment	
Headers	00400000	004003ff	0x400	\checkmark				Default	\checkmark	File: unpac				
.text	00401000	0042c1ff	0x2b200	\checkmark		/		Default	\checkmark	File: unpac				
.rdata	0042d000	00436bff	0x9c00	\checkmark				Default	\checkmark	File: urpac				
.data	00437000	004387ff	0x1800	\checkmark	I			Default	\checkmark	File: unpac				
.data	00438800	00439747	0xf48	\checkmark	\checkmark			Default						
.rsrc	0043a000	0043a1ff	0x200	\checkmark				Default	\checkmark	File: unpac				
.reloc	0043b000	0043d7ff	0x2800	\checkmark				Default	\checkmark	File: unpac				
tdb	ffdff000	ffdfffff	0x1000	\checkmark	I			Default	\checkmark					

Using Ghidra to Sync memory address with x32dbg and set the base address according to what was obtained with x32dbg.

🥩 File Edit Help			Memory Map [CodeBrowser: TestProject:/unpacked.bin]					—		\times	
📋 🗠 d											
Memory	Map - Image B	ase: 0040000)					+ +	≣⊼±	🔶 🗙 🛃	$\land \times$
				Me	emory Blocks						
Name	Start 🖹	End	Length	R Baso Imago	Addross	$\mathbf{\vee}$	Initialized	Byte Source	Source	Comm	nent
Headers	00400000	004003ff	0x400		Address		\checkmark	File: unpac			
.text	00401000	0042c1ff	0x2b200				\checkmark	File: unpac			
.rdata	0042d000	00436bff	0x9c00	✓ 003e0000			\checkmark	File: unpac			
.data	00437000	004387ff	0x1800	✓			\checkmark	File: unpac			
.data	00438800	00439747	0xf48								
.rsrc	0043a000	0043a1ff	0x200		Cancel	1	\checkmark	File: unpac			
.reloc	0043b000	0043d7ff	0x2800			J	\checkmark	File: unpac			
tdb	ffdff000	ffdffff	0x1000	\checkmark	Defa	ult	\checkmark				

Using Ghidra to sync a memory address with x32dbg

From here, the address of the suspected-string-decryption function will be updated accordingly and be in-sync with x32dbg.



String Decryption Function in Ghidra with Updated Memory Address

The new function address is 003f4550. This value can be used to create a breakpoint inside of x32dbg.



Updated Memory Address in Ghidra



creating a breakpoint on a known suspicious function The breakpoint is then hit with an argument of j h1#A

۹	CPU	l og	Notes A Breakpoints	: 🤹 Memory Map	🍹 Call Stack	🥼 SEł	н 🔎	Script	🥼 Symbols	🔮 Source	🐇 References	🁼 Threads
EIP		003F4550 003F4551	55 8BEC	push ebp mov ebp.esp						_		Hide FPU
		00374536 00374536 00374530 00374530 00374530 00374530 00374530 00374530 003745367 003745367 003745367 00374537 00374537 00374537 00374537 00374537 00374537 00374535 00374535 00374538 00374558 00374558 00374558 00374558 00374558 00374558 00374558 00374558 0037458	33EC 0C 88B5 08 536 88 8945 F8 57 50 88F1 8945 8945 F8 57 70 8844 14 38F9 10 77 26 83F9 10 72 02 81E 57 50 53 8376 10 8376 10 8376 10 8377 50 53 53 54 55 55 55 50 50 50 50 50 50 50 50 50 50 50 50 50 50 50 50 50 50 50 50 50 50 50 50	sing aspirate mov easi, dword ptr push ebs; mov easi, ecx mov devi, dword ptr s; push edi mov edi, dword ptr mov edi, dword ptr mov edv, dword ptr s; cmp edi, ecx ja unpacked. 3F4577 mov ebx, esi cmp ecx, 10 jb unpacked. 3F4577 mov ebx, dword ptr push edx push edx push edx push edx push edx pop esi pop esi pop esi pop ebx mov esp, ebp ret 8 cmp etd, 2FFFFFFFF is unpacked 2FFFFFFFF	<pre>ss:[ebp+8] [ebp-8],eax ss:[ebp+c] ds:[esi+14] [ebp-C],ecx 7 ds:[esi] [esi+10],edi 200 [eci+ebx],0</pre>		EAX ECX EDX EDX EDX EDX EDX EDX EDX EDX EDX EST EDX EST EDX EST EDX EST EDX EST EDX EST EDX EST EDX EST EDX EDX EDX EDX EDX EDX EDX EDX EDX EDX	003E1020 00437E74 00864000 007FE7A4 0040021C 0040021C 0040021C 0040021C 0040021C 0040021C 0040021C 0040021C 0040021C (stdcall) p+4] 0041 p+3] 0000 p+C] 003F p+10] 0037 p+14] 003	"j hl#A" unpacked. &"j hl#A' unpacked. 300 0 1 00000 (ERROR_3 00000 (ERROR_3 00000 (STATUS, 236C unpacked. 00020 000020 B164 unpacked. F6025 <unpacke< td=""><td>00417874 , , , , , , , , , , , , , , , , , , ,</td><td>3d32d239380a49b6 > (003F6025) > (003F6025)</td><td>f83128fe71ea01"</td></unpacke<>	00417874 , , , , , , , , , , , , , , , , , , ,	3d32d239380a49b6 > (003F6025) > (003F6025)	f83128fe71ea01"
-5	Dump	1 🐇 Dun	np 2 🐇 Dump 3 🐇 [Dump 4 🍏 Dump 5	🔮 Watch 1	e Lo	cals	Struct		00	7EFAA4 003E1031 7EFAA8 0041236C	return to unpa unpacked."6e3d

Beginning of a suspicious function in x32dbg

Allowing the malware to Execute Until Return will retrieve the result of the function. In this case it was a large hex string that was pretty uninteresting.

003F4676	C2 0800	ret 8				Hide FPU
003F4679	E8 8E5D0000	call unpacked.3FA40C				
003F467E	E8 FDUAUUUU	call unpacked. 3F5180		EAX 00)417в74	&"6e3d32d239380a49b6f83128fe71ea01"
003F4683	E8 88DAFEFF	Call unpacked.3E2110		FBX 00	000086	
003F4688	cc			ECX 00	000000	
003F4689	cc				0000000	
003F468A	cc					
003F468B						
003F468C				ESP 00	J96FB64	0112 112 114 11
003F468D				ESI OU	040D21C	& jni#A
003F468E				EDI 00	0000004	
003F468F	CC					
003F4690	68 <u>2C2F4100</u>	push unpacked.412F2C		EIP 00	03F4676	unpacked.003F4676
003F4695	E8 78100000	Call unpacked. SFS/12				
003F469A				EFLAGS	00000287	
003F469B				ZF 0 PF	= 1 AF 0	
003F469C				OF 0 SF	= 1 DF 0	
003F469D				CF 1 TF		
003F469E						
00354640		nuch ehn		LastErro	r 000000	ON (EPROP SUCCESS)
00354641	88EC	mov ebp esp		LactStat	uc c00000	C (STATUS NO TOKEN)
003F46A3	83EC 08	sub esp,esp			us coooo	C (STATUS_NO_TOKEN)
003F4646	53	nuch ehv				
003F46A7	56	nush esi				
003F46A8	57	push edi		Default (sto	lcall)	
003F46A9	8BFA	mov edi.edx		Deradic (St	accuny	
003F46AB	8BD9	mov ebx.ecx		1: [esp+4	4] 00412360	C unpacked.0041236C "6e3d32d239380a49b6f83128fe71ea01"
003F46AD	8B55 08	mov edx.dword ptr ss:[ebp+8]		2: [esp+	8] 00000020	00000020
003F46B0	895D F8	mov dword ptr ss:[ebp-8],ebx		3: [esp+	C] 003FB164	unpacked.003FB164
003F46B3	8B47 14	mov eax, dword ptr ds:[edi+14]		4: [esp+	10] 003F602	25 <unpacked.entrypoint> (003F6025)</unpacked.entrypoint>
003F46B6	8B77 10	mov esi, dword ptr ds:[edi+10]		5: [esp+	14] 003F602	25 <unpacked.entrypoint> (003F6025)</unpacked.entrypoint>
003F46B9	2BC6	sub eax,esi	-			
003=1600	<u><u><u></u></u><u></u><u></u><u></u><u></u><u></u><u></u><u></u><u></u><u></u><u></u><u></u><u></u><u></u><u></u><u></u><u></u><u></u></u>	mov ocy dword ntr. dc+Fody 101				

End of a suspicious function - viewing the returned value - possible decoded string However, Clicking F9 or Continue will cause the Decryption code to be hit again.

Sadly, this again revealed some largely uninteresting strings



I eventually realised that this function was not used to decode the final strings. But was rather to obtain copies of the same base64 obfuscated strings that were previously found.

At this point I experimented with the Suspicious imports, but could not reliably trace them back to a function that would obtain the decrypted c2's .

However - I did get lucky and was able to locate an interesting function towards the main malware function of the code.

This function was located at 003d29b0.

Locating Main

I was able to locate main by browsing to the EntryPoint.



Attempting to locate the main function in Ghidra - Starting from Entry Point

Cf	Decom	pile: FUN_003f5ea3 - (unpacked.bin)
52		register_thread_local_exe_atexit_callback(*puVar7);
53		
54		FUN_003fabfb();
55		FUN_003fb2f7();
56	_	FUN 003fb2f1();
57		$unaff_ESI = FUN_003f4040();$
58	•	<pre>bVar3 = is_managed_app();</pre>
59		if (bVar3) {
60		if (!bVar2) {

Attempting to locate the main function using Ghidra

C	Decompile: FUN_003f4040 - (unpacked.bin)
1	
2	void FUN_003f4040(void)
3	
4	{
5	FUN_003e9870();
6	FUN_003e7b70();
7	FUN_003e93a0(1);
8	FUN_003e8d50(1);
9	FUN_003f14b0();
10	FUN_003f4000();
11	return;
12	}
13	

Successfully finding the main function within Ghidra



Identifying a possible string decryption function in Ghidra

When this function is executed - a base64 encoded value is passed as an argument.

EIP 00322001 55 00322083 66 FF 00322083 66 FF 00322084 66 45604000 00322400 65141 000000 00322400 1032240 10000 0032240 1000000 0032240 1000000 00322400 8045 F0 00322400 8045 F0 00322400 8045 F4 00322400 8045 F4 0032240 8045 F4 0045	<pre>push ebp push FFFFFFF push unpacted.406048 push seatmord ptr [6][0] push seatmord ptr [6][0] push seatmord ptr [6][0] mov dest, dword ptr ss[[dbp-10].eax push esi bas eax dword ptr ss:[ebp-4].esi mov dword ptr [6][0].eax mov dword ptr [6][0].eax mov dword ptr [6][0].eax mov dword ptr ss:[ebp-4].esi mov dword ptr ss:[ebp-4].esi mov dword ptr ss:[ebp-4].esi mov dword ptr ss:[ebp-4].esi push esi, acx mov dword ptr ss:[ebp-4].esi push esi, acx mov dword ptr ss:[ebp-4].esi push esi esi esi esi esi esi esi esi esi esi</pre>	eax:&"Ixc4aDpERIOD==" eax:&"Ixc4aDpERIOD==" eax:&"Ixc4aDpERIOD==" esi:&"c:\\Users\\Wilhouse\\Deskt eax:&"Ixc4aDpERIOD==" eax:&"Ixc4aDpERIOD==" eax:&"Ixc4aDpERIOD=="	Hide FPU EXX 00CFFCA0 EXX 00CFFCA EXX 00C
00322365 C745 PC 0000 0032236C 840 C747 PC 0000 0032236C 840 C747 PF 00322377 8845 C0 00322377 8845 C0 00322377 8845 C0 00322378 C645 PC 01 00322378 8845 C8 00322402 8845 D8 00322403 8845 D8 00322403 8645 PF 00 00322401 8845 C8 00322411 8855 08	<pre>000 met dword ptr ss:[cbp-4]] o' lea ecx.dword ptr ss:[cbp-40] cal unpacked.sE2780 dea ext.dword ptr ss:[cbp-40] mov byte ptr ss:[cbp-4], push ext.dword ptr ss:[cbp-4] lea ecx.dword ptr ss:[cbp-4], cal unpacked.sE2800 add bype fr ss:[cbp-28] mov eax.dword ptr ss:[cbp-28] mov eax.dword ptr ss:[cbp-3].</pre>	eax:&"IXc4RaDpERIODA=="	LastError 0000000 (ERROR_SUCCESS) LastError 0000000 (ERROR_SUCCESS) LastErtatus 000007C (STATUS_NO_TOKEN) Default (stdcall) 1: [sep+4] 0000052 4000055 (STATUS_NO_TOKEN) 2: [sep+4] 0000052 4000055 (STATUS_NO_TOKEN) 2: [sep+10] 0000054 4000052 3: [sep+10] 00000054 (STATUS_NO_TOKEN) 3: [sep+10] 00000054 (STATUS_NO_TOKEN)

Base64 Function Arguments viewed in a debugger.

Executing until the end of the function - A value is obtained which the malware used to create a folder in the users temp directory.

003E2ADF	C3	ret		▲			Hide FPU
003E2ADF 003E2AE0 003E2AE7 003E2AE7 003E2AE7 003E2AE7 003E2AE9 003E2AE9 003E2AE9 003E2AE9 003E2AE9 003E2AE9 003E2AE9 003E2AE9 003E2AE9 003E2AF1 003E2AF1 003E2AF3	C3 E8 27790100 CC CC CC CC CC CC CC CC CC CC CC CC C	Pet Call unpacked.3FA40C int3 int3 int3 int3 int3 int3 int3 int3		EAX EBX ECX EBP ESI EDI EIP EFL ZF	00CFFCF4 0089D000 7CCED833 00EC0000 00CFFE78 00CFFC9C 00EC74F0 00E	"595f021478" &"C:\\Users\\Mil &"ALLUSERSPROFIL unpacked.003E2AD	Hide FPU house\\Desktop\\Redline2\\u E=C:\\ProgramData"
003E2AFA 003E2B00 003E2B01 003E2B07 003E2B07 003E2B0C 003E2B12 003E2B12	64:A1 00000000 50 81EC 0C040000 A1 0C704100 33C5 8945 F0 56 57 57 57	<pre>mov eax; dword ptr file:[0] push eax sub esp,40C mov eax; dword ptr ds:[41700C] xor eax,ebp mov dword ptr ss:[ebp-10],eax push esi push edi</pre>	eax:"595f021478" eax:"595f021478" eax:"595f021478" esi:&"c:\\users\\Milhouse\\Deskt edi:&"ALLUSERSPROFILE=C:\\Progra	OF CF Las Las	0 SF 0 DF 0 0 TF 0 IF 1 tError 000000 tStatus C00000	000 (ERROR_SUCCESS) 07C (STATUS_NO_TOKE	
003E2814 003E2817 003E281D 003E281F 003E2821 003E2827 003E2820 003E2820	30 8045 F4 64:A3 00000000 88F2 89BD ECFBFFFF 89BD ECFBFFFF 68 00040000 90040000	<pre>public ex. dword ptr ss:[ebp-C] mov dword ptr is:[0].eax mov edsi.edx mov edsi.edx mov dword ptr ss:[ebp-414].edi public 400 las.oax dword ptr ss:[ebp-414].edi public 400</pre>	eax: 5551621478 es; 257621478 es; 267C:\Users\Wilhouse\\Deskt ed; 267C:\UsersPROFILE=C:\\Progra	1: 2: 3: 4: 5:	GEC (SUCCAII) [esp+4] 00ED5EF [esp+5] 4000006 [esp+c] 00CFFBD [esp+10] 007000 [esp+14] 000000	78 00ED5EF8 52 40000062 58 00CFFBD8 96F 0070006F 910 00000010	

Obtaining a decoded value using x32dbg

The next call to this function - took a base64 encoded argument and returned a file name that the malware was copied into.

		Hide FPU
EAX EBX ECX EDX EBP ESP	00CFFCA0 00B9D000 00CFFCC4 3D3D514F 00CFFE78 00CFFC9C	&"76xoROUVDkN10Q=="

A second encoded value in eax- viewed in x32dbg

			Hide F	PU		
EAX EBX ECX EDX EBP ESP	00CFFCC4 00B9D000 7CCEDB33 00EC0000 00CFFE78 00CFFC9C	"oneetx.exe"				
Default	: (stdcall)				. . 5	▲ Unlocked
1: [e 2: [e 3: [e 4: [e 5: [e	sp+4] 00ED50 sp+8] 00CFFD sp+C] 000000 sp+10] 00CFF sp+14] 00000	00 00ED5000 30 00CFFD30 &"C:\ 0A 0000000A E78 00CFFE78 010 00000010	Users\\Milhouse\\Ap	Data\\Local\\	Temp\\595f021478"	Î

A decoded filename - located using return addresses in x32dbg

At a location of 003e9870 - was a function responsible for checking the location of the current running file.



If the location did not match C:\\users\\

<user>\\appdata\\local\\temp\\595f021478\\oneetx.exe - then the malware would
terminate.

Here we can see the return value from the function.

		Hide FPU			
EAX 00CFFCC4 EBX 00B9D000 ECX 7CCEDB33 EDX 00EC0000 EBP 00CFFE78 ESP 00CFFC9C	"oneetx.exe"				
Default (stdcall)				🔺 🗌 Unlocked
1: [esp+4] 00ED500 2: [esp+8] 00CFFD 3: [esp+C] 0000000 4: [esp+10] 00CFFE 5: [esp+14] 000000	00 00ED5000 80 00CFFD30 &"C:\\U 0A 0000000A 278 00CFFE78 010 00000010	sers\\Milhouse\\AppD	ata\\Local\\	Temp\\595f021478"	Î

As well as the outgoing function calls in the Ghidra Function Tree.



Viewing the Function Tree Using Ghidra

After the directory check is performed - the malware enters FUN_003e7b70 attempts to creates a mutex with a value of 006700e5a2ab05704bbb0c589b88924d



By breaking on CreateMutexA - The value of <u>006700e5a2ab05704bbb0c589b88924d</u> can be seen as an argument.

00327885 FF15 4C03 00327894 3D 87003 00327899 74 01 00327895 C3 00327895 C3 00327858 C3 00327858 C3 00327858 C3 00327858 C3 00327858 C3 00327858 C3 0032784 CC 0032784 CC	00 Icali dword ptr ds: [-&detLasterrors 1] 1 deax: "006700e5a2ab05704bbb 1 deax: 0035rbac 1 deax: 0005rbac 1 deax: 0	Ļ
EAX 00AB4B98 EBX 0063B000 ECX 00000000 EDX 0063E000 EBP 0055FDB4 ESP 0055FD60	"006700e5a2ab05704bbb0c589b88924d"	

If the mutex creation returned a value of 0xb7 (Already Exists) - then the malware would terminate itself.



Bypassing Anti-Something Checks

These two checks on the file path and Mutex can function as pseudo anti-debug checks. In order to continue analysis, they needed to either be patched or bypassed.

In order to bypass the file path check - I allowed the malware to execute inside the analysis VM and copy itself to the correct folder. I then opened the new file inside the debugger.

Alternatively - You could have patched or nop'd the function. but I found that just moving it to the expected folder worked fine.

🕨 📙 🔺 This PC 🔺	Local Disk (C:) > Users > Milhouse > AppData > Local	> Temp > 595f021478	ٽ ~
Microsoft ^	Name	Date modified	Туре
Mozilla	oneetx.exe	4/10/2023 9:27 AM	Application
NuGet			

Package Cache

Once the new file was loaded - I updated the base address in Ghidra to match the new address in x32dbg.

00400000	00004000 👔 User		
000AA0000	00001000 ¥ User	oneetx.exe	
00AA1000	0002C000 🕆 User	text"	Executable code
00ACD000	0000A000 🍸 User	".rdata"	Read-only initialized dat
00AD7000	00003000 🝸 User	".data"	Initialized data
00ADA000	00001000 🝸 User	".rsrc"	Resources
00ADB000	00003000 🝸 User	".reloc"	Base relocations
00AE0000	000FC000 Vser	Reserved	

Once I updated the base address - I set a breakpoint on CreateMutexA and the suspected decryption function FUN_XXX29b0



Once I hit the breakpoint on CreateMutexA - I stepped out of the function using Execute Until Return and then Step Over twice.

This allowed me to see the return value of b7 from the GetLastError function. When I allowed the malware to continue to run - it quickly terminated itself without hitting the decryption breakpoint.

	00AA7B84 00AA7B86	6A 00 6A 00	push 0 push 0		Hide FF
	00AA7B88 00AA7B8E	FF15 90D0AC00 FF15 4CD0AC00	call dword ptr ds:[<&CreateMutexA>]	EAX 000000B7 '.'	
e	00AA7B94	3D B7000000 74 01	cmp eax, B7	ECX 00770000	
	00AA7B9B	C3	ret nush 0	EDX 00770000 EBP 004FFBC8	
	00AA7B9E	E8 2C160100	call oneetx.AB91CF	ESP 004FFBC4	
	00AA7BA4	cc		1	
	00AA7BA5 00AA7BA6	cc	int3	Default (stdcall)	

To fix this - I used Edit to patch the return value to be B6 instead.

Edit			×	
Expression:	000000в6			<u>ЕАХ</u> 00000в7 '.'
Bytes:	в6000000			EBX 003AE000 ECX 00770000
Signed:	182			EDX 00770000 EBP 004FFBC8
Unsigned:	182			ESP 004FFBC4
ASCII:	· · · ¶			Default (stdcall)
		ОК	Cancel	1: [esp+4] 004FFC10 004FFC10 2: [esp+8] 00AB5F9D oneetx.0

Patching a return value using X32dbg

Upon running the malware - The decryption function was hit again.

Following the return of the decryption function using Exeute Until Return revealed a pretty boring \\ character.



But allowing it to hit a few more times - it eventually returned a value of **Startup** which was pretty interesting.

			Ніс
EAX EBX ECX	004FFAF0 003AE000 C351E3AF	"Startup"	

Hitting again revealed a registry path of

SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Explorer\\User Shell Folders



Eventually some more interesting values were returned. Including a partial command likely used to create persistence.

As well as some possible signs of enumeration

004FFA6C

003AE000

EAX

EBX

		нае ни	
EAX EBX ECX	004FF2FC 004FF508 c351e3AF	&"SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion"	
EAX EBX	< 004F < 004F	374 &"GetNativeSystemInfo" 508	
EAX EBX ECX	004FF4C8 8 003AE000 C351E34E	SYSTEM\\CurrentControlSet\\Control\\ComputerName\\ComputerName"	

Eventually - The names of some security products was also observed. Likely the malware was scanning for the presence of these tools.

EAX EBX	004FF4B0 004FF990	"AVAST Software"
EAX EBX	004FF4B0 FFFFFF00 C351E34E	"Kaspersky Lab"

C2 Information

Allowing the decryption function to continue to execute and hit our breakpoint. We can eventually observe C2 information.

			Hid
EAX EBX ECX	02A3FDC4 00000000 C351E3AF	&"/plays/chapter/index.php"	
EAX EBX	02A3FCE4 00000000	"KKE+"	
ECX EDX	02A3FCFC 00770000	&"77.91.124.207"	

Automating the Decryption - Kinda

Eventually the constant breakpoint + execute until return combination got tiring. So I decided to try and automate it using a Conditional Breakpoint and Log.

To do this - I allowed the malware to execute until the end of a decryption function.



And then created a Conditional Breakpoint that would log any string contained at eax, then continue execution.



Settiing a Conditional Breakpoint (and logging a value) using X32dbg

Allowing the malware to continue to execute. I could observe the decoded values printed to the log menu of x32dbg.

🍹 CPU	4	Log	Note	es 🔮	Breakpoints	ø	Memory Map	4	Call Stack	۲	SEH	١	Script	4	Symbols
DLL Unic DLL Unic DLC Unic DLC Unic DLC Unic DLC Unic DLC Unic DLC Unic DCC 04 Decoded	<pre>paded: pade</pre>	Log 6A5A0000 6A590000 6A5F0000 6A610000 6A610000 6A610000 6A620000 it .dll cli ativeSys 132.dll" //" //" ativeSys 132.dll" ys/chapt .124.207 ns/"	<pre> Note bcp471a sppc.dl slc.dll userenv appreso windows p.dll " temInfo" er/index " temInfo" </pre>	ss ngs.dll l .dll lver.dll .statere .php"	Breakpoints		Memory Map	4	Call Stack		SEH	0	Script	4	Symbols
Decoded Decoded	"Plugi ???	ns/"													
Decoded Decoded	??? "dll"														
Decoded Decoded	??? ??? "http:	//"													
Decoded Decoded	"https "http:	://" //"													

Successfully using conditional breakpoints to decode a malware sample.

This revealed some c2 information - referencing an IP with 1/87 detections as of 2023/04/10

	() 1 security vendor flagged this IP address as malicious					
/ 87	77.91.124.207 (77.91.124.0/24) AS 203727 (Daniil Yevchenko)					
Community Score C	AILS RELATIONS COMMUNITY					

The full list of decoded strings can be found here.

```
&"SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Explorer\\User Shell Folders"
&"SYSTEM\\CurrentControlSet\\Control\\ComputerName\\ComputerName"
&"SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion"
&"abcdefghijklmnopqrstuvwxyz0123456789-_"
&"/Create /SC MINUTE /MO 1 /TN "
&"/plays/chapter/index.php"
&"GetNativeSystemInfo"
&"cred.dll|clip.dll|"
"77[.]91[.]124[.]207"
"Panda Security"
"AVAST Software"
"Kaspersky Lab"
"ProgramData\\"
"ComputerName"
"CurrentBuild"
"kernel32.dll"
"Bitdefender"
"Doctor Web"
"https://"
"Plugins/"
"SCHTASKS"
"http://"
" /TR \""
"Startup"
"Comodo"
"Sophos"
"Norton"
"Avira"
"\" /F"
L"\\¬="
"POST"
"&vs="
"3.70"
"&sd="
"&os="
"&bi="
"&ar="
"&pc="
"&un="
"&dm="
"&av="
"&lv="
"&og="
"ESET"
"dll"
"<c>"
"id="
"AVG"
???
```

Bonus: Utilising Decrypted Strings To Identify the Malware Family

This section was not in the original blog, but was later added when I was informed by another researcher that the malware might not be Redline.

I then revisited my analysis and determined that the sample was Amadey Bot.

I was able to determine this mostly by researching (googling) the decrypted strings.

I thought it would be useful for others to see what this process looked like :)

Decrypted strings are not just useful for C2 information. They are equally as useful for identifying the malware that you are analyzing.

Unless you are analyzing the latest and greatest APT malware, your sample has likely been analyzed and publically documented before. You'd be surprised how much you can determine using Google and the "<u>intext</u>" operator. (Essentially it forces all search results to contain your query string, significantly reducing unrelated content)

From decrypted strings, try to pick something specific.

For example, the following decrypted string &"cred.dll|clip.dll|" can be used to craft a Google query of intext:clip.dll intext:cred.dll malware.

This returns 7 results that reference a combination of Redline Stealer and Amadey Bot.

The first link contains IOC's from an Amadey Bot sample, which align closely with the sample analysed in this blog.

Amadey	
(PID) Process	(2908) mnolyk.exe
C2 (1)	62.204.41.5/Bu58Ngs/index.php
Version	3.66
Options	
Drop directory	5eb6b96734
Drop name	mnolyk.exe
Strings (116)	SCHTASKS
	/Create /SC MINUTE /MO 1 /TN
	/TR "
	" /F
	SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnce
	SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\User Shell Folders
	Startup
	Rem
	cmd /C RMDIR /s/q
	SOFTWARE\Microsoft\Windows\CurrentVersion\Run

In the second link - An additional Amadey sample is analysed with the exact same filename as this one. Albeit with a different C2 server.

Malware configuration

Amadey

(PID) Process	(1692) oneetx.exe		
C2 (1)	http://193.233.20.36		
Version	3.69		
Options			
Drop directory	c5d2db5804		
Drop name	oneetx.exe		
Strings (116)	SCHTASKS		
	/Create /SC MINUTE /MO 1 /TN		

At this point - I would have moderate confidence that the sample is Amadey Bot.

For additional confirmation, I would typically google this family and see if any TTP's are the same or at least similar.

I googled Amadey Bot Analysis and discovered this blog from AhnLab.com.



Amadey Installation Path

> %TEMP%\9487d68b99\bguuwe.exe

Command registered to Task Scheduler

> cmd.exe /C REG ADD "HKCU\Software\Microsoft\Windows\CurrentVersion\Explorer\User Shell Folders" /f /v Startup /t REG_SZ /d %TEMP%\9487d68b99\

> schtasks.exe /Create /SC MINUTE /MO 1 /TN bguuwe.exe /TR "%TEMP%\9487d68b99\bguuwe.exe" /F

The Ahnsec Blog also references a list of AV products that are enumerated by Amadey Bot.

Anti-malware Name	Number
Х	0
Avast Software	1
Avira	2
Kaspersky Lab	3
ESET	4
Panda Security	5
Dr. Web	6
AVG	7
360 Total Security	8
Bitdefender	9
Norton	10
Sophos	11
Comodo	12
Windows Defender (assumed)	13

Coincidentally, almost all of those strings were contained in our sample

'Panda Security"
'AVAST Software"
'Kaspersky Lab"
'ProgramData\\"
'ComputerName"
'CurrentBuild"
'kernel32.dll"
'Bitdefender"
'Doctor Web"
'https://"
'Plugins/"
'SCHTASKS"
'http://"
'/TR \""
'Startup"
'Comodo"
'Sophos"
'Norton"
'Avira"

The <u>Ahnsec blog</u> also references specific parameters that are sent in POST requests made by Amadey Bot.

Item	Data Example	Meaning			
id	129858768759	Infected system's ID			
VS	3.21	Amadey version			
sd	37bbd7	Amadey ID			
OS	9	Windows version ex) Windows 7 – 9 Windows 10 – 1 Windows Server 2012 – 4 Windows Server 2019 – 16			
bi	0	Architecture (x86 – 0, x64 – 1)			
ar	0	Admin privilege status (1 if admin privilege is available)			
рс	PCNAME	Computer name			
un	USERNAME	User name			
dm	DOMAINNAME	Domain name			
av	0	List of installed anti-malware			
lv	0	Set as 0			
og	1	Set as 1			

Table 1. Data sent to the C&C server

Coincidentally, almost all of those same fields (first column) are referenced in our decrypted strings.

Since POST request parameters are pretty specific - Was confident my sample was actually Amadey bot.

L"\\¬="	
"POST"	
"&vs="	
"3.70"	
"&sd="	
"&os="	
"&bi="	
"&ar="	
"&pc="	
"&un="	
"&dm="	
"&av="	
"&lv="	
"&og="	
"ESET"	
"dll"	
" <c>"</c>	
"id="	
"AVG"	
<u>;;;</u>	
At this point	Lalso reviewed a second blog from Plackborny Which confirmed much of the

At this point, I also reviewed a <u>second blog from Blackberry</u>. Which confirmed much of the <u>same analysis as AhnSec</u>.

At this point, I was comfortable re-classifying the malware as Amadey bot.

(I also learned not to blindly follow tags from Malware Reps)

Conclusion and Recommendations

At this point I'm going to conclude the analysis as we have successfully located the C2 information and identified the malware family. In a real life situation, this analysis could serve multiple purposes.

- Decrypted strings can be googled to aid in malware identification.
- Decrypted strings contain commands and process names that can be used for process-based hunting
- Decrypted Strings contain URL structure which can used to hunt or develop detection rules for proxy logs.
- Decrypted Strings contain an IP that could be used to identify infected machines.
- Decrypted Strings can be used to enhance a Ghidra or IDA database enhancing the decompiler output and leading to better RE analysis.

- Better automation could be used to make a config extractor useful for a threat intel/analysis pipeline. (Replacing x32dbg with Dumpulator would be a great way to do this)
- + lots of fun :D

Virustotal

At the time of this analysis (2023/04/10) - There is only 1/87 detections for the C2 on Virustotal

	① 1 security vendor flagged	d this IP address a	as malicious					
/ 87	77.91.124.207 (77.91.124.0/24 AS 203727 (Daniil Yevchenko)	4)						
Community Score								
DETECTION DETAILS RELATIONS COMMUNITY								
① 1 sec	curity vendor flagged this IP address as malicious			(‡); ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;				
77.91.124 AS 20372	1.207 (77.91.124.0/24) 7 (Daniil Yevchenko)			FI				
Community Score 🕑								
DETECTION DETAILS RELATIONS COMMUNITY								
Crowdsourced context ①								
HIGH 1 MEDIUM 0 LOW 0 INFO 0 SUCCESS 0								
A CnC Panel - according to source ViriBack - 2 days ago								
Security vendors' analysis ① Do you want to automate checks?								
ViriBack (D Malware	Abusix	🚫 Clean					
Acronis (Clean	ADMINUSLabs	✓ Clean					
AICC (MONITORAPP) (Clean	AlienVault	😔 Clean					

Decoded Strings

A full list of strings obtained using the log function of x32dbg.

(Noting that these are in order of length and not location of occurrence.)

```
&"SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Explorer\\User Shell Folders"
&"SYSTEM\\CurrentControlSet\\Control\\ComputerName\\ComputerName"
&"SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion"
&"abcdefghijklmnopqrstuvwxyz0123456789-_"
&"/Create /SC MINUTE /MO 1 /TN "
&"/plays/chapter/index.php"
&"GetNativeSystemInfo"
&"cred.dll|clip.dll|"
"77[.]91[.]124[.]207"
"Panda Security"
"AVAST Software"
"Kaspersky Lab"
"ProgramData\\"
"ComputerName"
"CurrentBuild"
"kernel32.dll"
"Bitdefender"
"Doctor Web"
"https://"
"Plugins/"
"SCHTASKS"
"http://"
" /TR \""
"Startup"
"Comodo"
"Sophos"
"Norton"
"Avira"
"\" /F"
L"\\¬="
"POST"
"&vs="
"3.70"
"&sd="
"&os="
"&bi="
"&ar="
"&pc="
"&un="
"&dm="
"&av="
"&lv="
"&og="
"ESET"
"dll"
"<C>"
"id="
"AVG"
???
```

Useful Links

- AhnSec Labs Blog on Amadey Stealer
- Blackberry Blog <u>Amadey Bot Analysis</u>
- Mandiant <u>Repo for Flare VM Install</u>
- X32dbg Documentation <u>Conditional Breakpoints in X32dbg</u>