

★ Get unlimited access to the best of Medium for less than \$1/week. [Become a member](#) X

Funtastic Packers And Where To Find Them



Eli Salem · [Follow](#)

11 min read · Jan 19, 2021

Listen

Share

More



[What's Inside?](#)

In malware, we often see threat actors that tend to obfuscate or encrypt their code in order to slow down the analysis of security researchers. To do so, many authors tend to use open-source packers but also craft their own custom packers.

While custom packers are definitely not a new thing, it is always interesting to observe how they work, and what is the shared similarities between them in different malware.

In this writeup, I will present some known first-stage malware that uses custom packers or other packing mechanisms.

I will also share some theoretical insights regarding the way to approach these packers, and hopefully, shed some light using a step-by-step dynamic observation. The purpose here is to give some guidelines on what to look for when we try to understand how the unpacking mechanism works.

Note: this writeup is about observing the unpacking mechanism, therefore, faster methods to get the final payload such as:

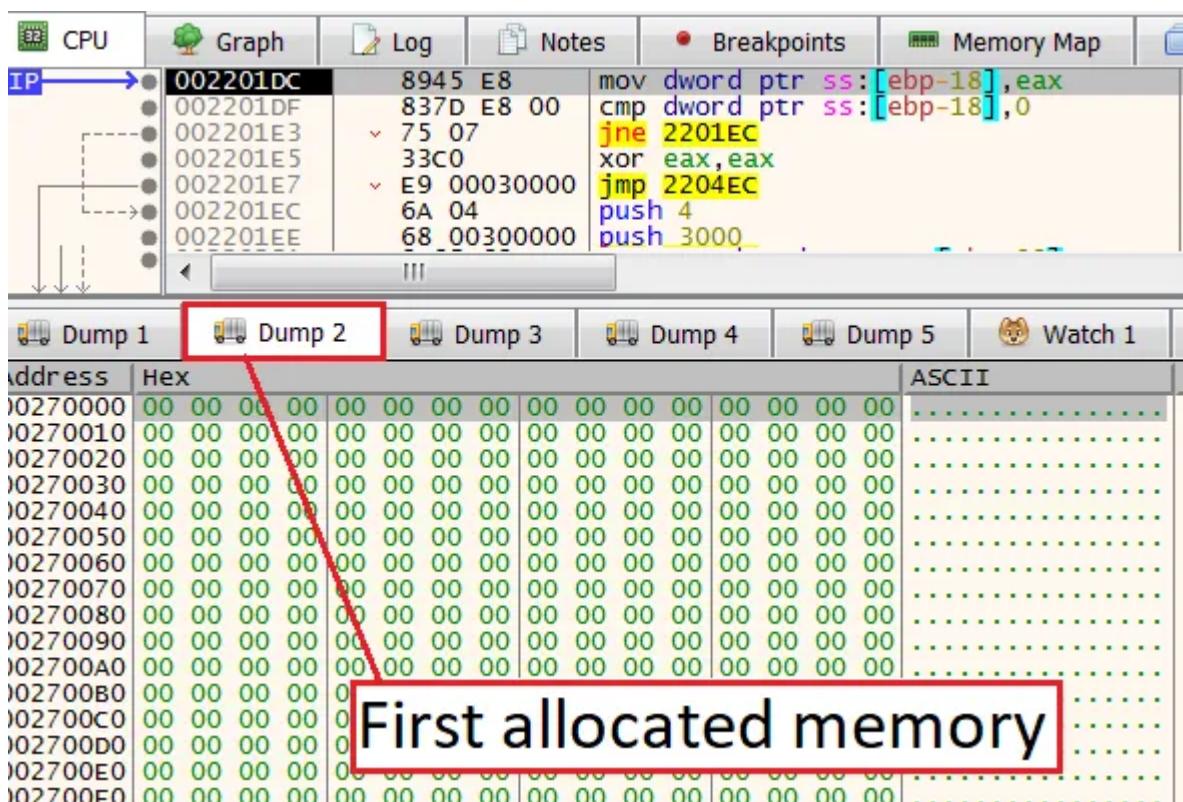
Tracking specific API calls and ignoring others, or, focusing on the code injection parts; won't be mentioned -the goal of this article is to explain the unpacking mechanism **and not** the fastest way to get the final payload.

Get2Downloader

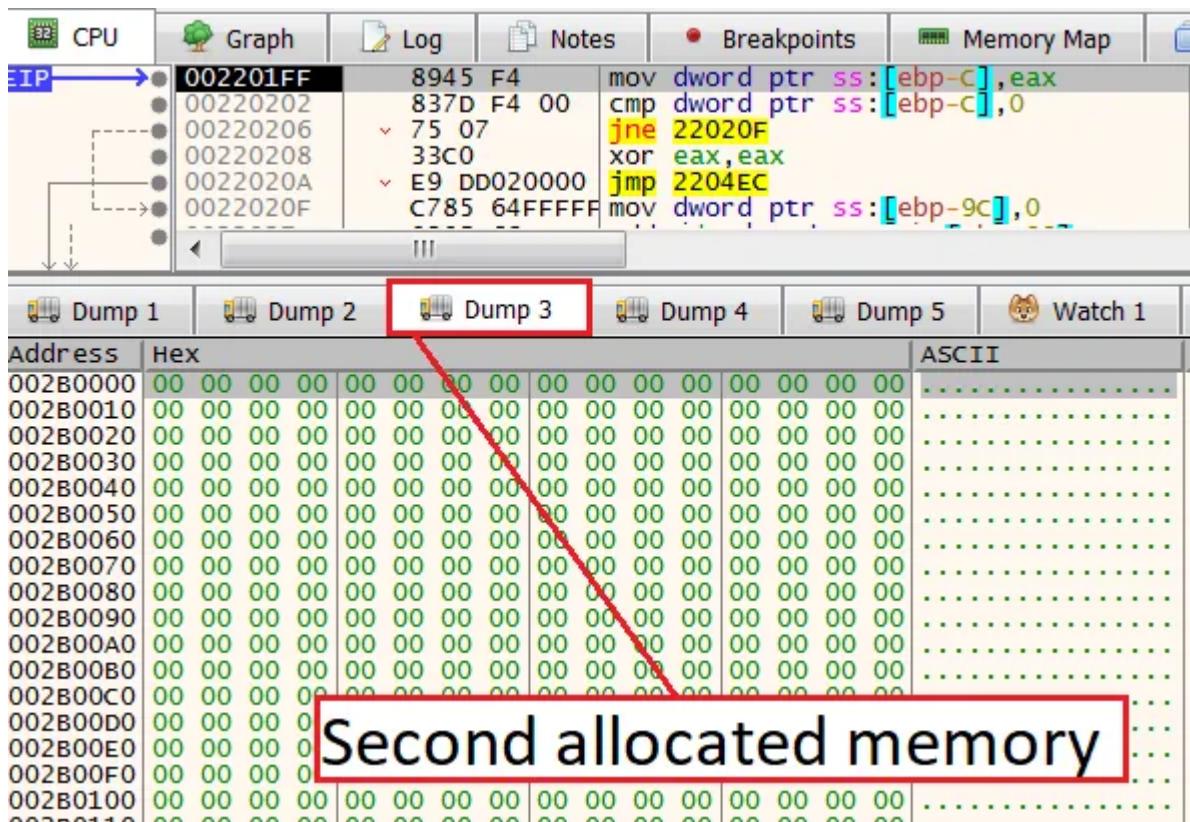
This downloader was first emerged in 2019 and associated with TA505 threat actors. It is known to deliver different malware, most notably Sdbbot.

As in every case, when malware wants to write unpacked content, it first needs to allocate the memory space for it. Get2 uses the API call VirtualAlloc three times, we'll focus only on the last two.

In each allocated memory we'll set a write hardware breakpoint, this assures us that whenever something will be written we'll know about it.



Get2: First Allocated Memory



Get2: Second Allocated Memory

After settings the breakpoints, we'll hit run until we reach the first breakpoint. Then, we'll observe a loop whose objective is to write content to the first allocated memory.

The screenshot shows the Immunity Debugger interface. The CPU tab displays assembly code with several jumps and loops. The memory dump tabs (Dump 1 to Dump 5) show the raw memory content at specific addresses. A red box highlights the first three allocations at addresses 00220219, 00220223, and 00220225.

Address	Hex	ASCII
00220219	53 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	S.....
00220223	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00220225	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Get2: First Allocated Memory being written

To save time, we'll set a breakpoint one step after the loop to see the entire written content. Then, we observe that the memory section is filled with obfuscated data.

The screenshot shows a memory dump tab with many allocations of obfuscated data. A red box highlights the first allocation at address 00220219.

Address	Hex	ASCII
00220219	53 39 8E 51 E2 FD AE 08 46 8C 7F C8 4E A1 A4 91	S9.Qâý®.F..ÈNj¤.
00220223	D6 B9 E4 99 4A 8C 68 B9 37 6C 2A 04 FE DD 23 49	Ö'ä..J.h'71*.þÝ#I
00220225	FB 1F E4 57 D3 25 C9 4D 41 AA 8F 0A 11 BF 7F 8F	Ü.äwÓ%ÉMA¤...¿..
00220229	20 B0 5E FF E7 7B 71 6F 22 C7 7F 65 80 6B 6E A2	°^Ýç{qo"C.e.kn¢
00220233	21 81 1B DF 96 69 BD ED 36 EA 78 4F 80 7B 5B C9	!..B.i½í6éxo.{{[É
00220237	77 DC 0A A2 B2 B5 AA C8 4D AD 94 BC 18 86 D3 58	wÜ.¢zµ¤ÈM..%..ÓX
0022023B	11 43 50 EB 23 00 69 CB 66 FB 50 00 95 69 A6 72	.CPë#.iEFÙP..i r
0022023F	07 7F 36 03 5B 14 1B CA B8 85 24 0E D7 EC 57 89	.6.[..È..\$.xìw.
00220243	C8 D2 16 11 24 41 F0 79 63 CO 4C C6 16 98 A4 09	Èò..\$AðycÀLÆ..¤.
00220247	3D DF 35 43 AA 35 A8 E6 0A BC 1A B8 24 03 93 09	=B5C¤5`æ.%..\$..
0022024B	43 1A BB AE D1 CA 50 05 D7 C8 1C 09 C3 9D 28 49	C.»¤NÉP.xÈ..A.(I
0022024F	34 92 67 DE 8D 9C 8F 46 F7 8D 3A 29 26 03 2C 9B	4.gþ...F÷.:)&.,.
00220253	7A D2 8E 89 D4 BB 98 B0 14 DE 2A 8B 11 83 99 89	zò..ô...þ*....
00220257	55 4A 0B 8A DD FF 28 A2 F5 1E 49 C1 C2 99 C9 92	UJ..Ýy(¢ö.IÀÀ.É.
0022025B	27 63 1C 3F 92 8E 1A 9B 57 EC 88 07 F0 9E 8D ED	'c.?....wì..ð..í
0022025F	D7 5F A6 9B 91 5C 0D 0E 4A B5 51 6F D1 A9 1B 85	x ...JþQoÑ@..

Get2: First Allocated Memory obfuscated content

After a couple of instructions, we found ourselves in another loop. At first glance, it seems this loop has characteristics we expect from traditional decryption\encryption routines, such as *shr* (shift right), *xor*, and *rol* (rotate left) opcodes. The loop changes the first bytes of the obfuscated content to “M8Z”, which starts to resemble the classic “MZ” string.

Address	Hex	ASCII
00270000	4D 38 5A 90 E2 FD AE 08 46 8C 7F C8 4E A1 A4 91	M8Z. à®. F.. ÈN..
00270010	D6 B9 E4 99 4A 8C 68 B9 37 6C 2A 04 FE DD 23 49	0. a. J. h'71*. bÝ#I
00270020	FB 1F E4 57 D3 25 C9 4D 41 AA 8F 0A 11 BF 7F 8F	Û. àwÓ%ÉMA... Ê..
00270030	20 B0 5E FF E7 7B 71 6F 22 C7 7F 65 80 6B 6E A2	^yç{qo"Ç. e. kn¢
00270040	21 81 1B DF 96 69 BD ED 36 EA 78 4F 80 7B 5B C9	!.. B. i½í6éxO. { [É
00270050	77 DC 0A A2 B2 B5 AA C8 4D AD 94 BC 18 86 D3 58	wÜ. t²µáÈM..%.. ÓX
00270060	11 43 50 EB 23 00 69 CB 66 FB 50 00 95 69 A6 72	. CPë#. iÉfÙP.. i r
00270070	07 7F 36 03 5B 14 1B CA B8 85 24 0E D7 EC 57 89	. 6. [.. É. \$. xiw.
00270080	C8 D2 16 11 24 41 F0 79 63 C0 4C C6 16 98 A4 09	Èò.. \$AðycALÆ..¤.
00270090	3D DF 35 43 AA 35 A8 E6 0A BC 1A B8 24 03 93 09	=B5Cª5 æ.%.. \$..
002700A0	43 1A BB AE D1 CA 50 05 D7 C8 1C 09 C3 9D 28 49	C. »®ÑEP. xÈ. ;À. (I

Get2: M8Z string written

To see the final stage of this decryption loop, we'll set a breakpoint one step after the jump instruction and hit run. Then, additional string fractures such as: “This program cannot run in DOS mode” and the word “PE”, will be revealed. Overall, we understand that this loop was responsible for taking the obfuscated content and do some decryption. However, it is obvious that this is not the final stage.

The screenshot shows the OllyDbg debugger interface. The assembly window displays the following code snippet:

```

00220312 EB 87 jmp 22029B
00220314 8B45 F4 mov eax,dword ptr ss:[ebp-C]
00220317 50 push eax
    ...

```

The dump window shows memory dump 1 with the following data:

Address	Hex	ASCII
00270000	4D 38 5A 90	M8Z.8.f...qÿ. Å.
00270010	01 40 C2 15	@A.A.....o ..
00270020	CD 21 B8 F5	f!. öL..This p.ro
00270030	67 CF 61 6D	gİam.c.n>.Tİbe_
00270040	75 BF 30 69	uɔi.DOSÜm.ode..
00270050	12 0A 24 98	..\$.D.!..øA@Y<.
00270060	76 DC 7C 2C	vÜ ,Tİ..øZ. /DBi
00270070	08 1D DE AA	.paö..ø.AY..=.
00270080	D9 F0 E3 88	Üðä..ż .TEDÜøKA.
00270090	AC 1D D4 AA	-.ðæ.ùáÅ.ù.ß..
002700A0	52 69 28 63	Ri(chp.í.PE.L...
002700B0	25 EE EF 5D	%ïj~.pà...!..ß8*
002700C0	C0 F9 62 10	åäc #` \$*nii

Get2: decryption loop ends

After the loop ends, we encounter a call to one of the Get2 functions. This function is important for the final decryption stage. We'll set a breakpoint on it and step into it.

The screenshot shows the OllyDbg debugger interface. The assembly window displays the following code snippet:

```

00220312 EB 87 jmp 22029B
00220314 8B45 F4 mov eax,dword ptr ss:[ebp-C]
00220317 50 push eax
00220318 8B4D E8 mov ecx,dword ptr ss:[ebp-18]
0022031B 51 push ecx
0022031C E8 1F060000 call 220940
    ...

```

The dump window shows memory dump 1 with the following data:

Address	Hex	ASCII
00270000	4D 38 5A 90	M8Z.8.f...qÿ. Å.
00270010	01 40 C2 15	@A.A.....o ..
00270020	CD 21 B8 F5	f!. öL..This p.ro
00270030	67 CF 61 6D	gİam.c.n>.Tİbe_
00270040	75 BF 30 69	uɔi.DOSÜm.ode..
00270050	12 0A 24 98	..\$.D.!..øA@Y<.
00270060	76 DC 7C 2C	vÜ ,Tİ..øZ. /DBi

Get2: Second stage decryption function

As we enter, we see that our second allocated memory has been manipulated., and the letter M has been written into it. This is because the function is copying and manipulating the content from the first to the second allocated memory.

The screenshot shows the Immunity Debugger interface. The assembly pane displays a sequence of instructions starting at address 0022098A. The memory dump tabs at the bottom show the first 5 allocations of memory, with Dump 1 containing the MZ header.

Address	Hex	ASCII
002B0000	4D 5A 90 00	MZ.....
002B0010	00 00 00 00
002B0020	00 00 00 00
002B0030	00 00 00 00
002B0040	00 00 00 00
002B0050	00 00 00 00

Get2: Second Allocated Memory Being Written

The entire process of writing content to the second allocated memory is also part of a large loop. Although we get what seems to be a clean portable executable, we didn't finish yet. When we inspect the PE headers we see that it is packed with a UPX packer.

The screenshot shows the OllyDbg debugger interface. The assembly window at the top displays the following assembly code:

```

EIP 00220BDA: E9 ABFDFFFF jmp 22098A
        8B45 DC mov eax,dword ptr ss:[ebp-24]
        2B45 0C sub eax,dword ptr ss:[ebp+C]
        8BE5 mov esp,ebp
        5D pop ebp
        C3 ret
        CC int3
    
```

The memory dump window below shows the executable's memory dump. The first few bytes are the MZ header and file signature. The exploit payload is visible, followed by the UPX compressed executable. Two specific bytes, 55 and 50, are highlighted in red boxes and labeled "UPX0" and "UPX1" respectively.

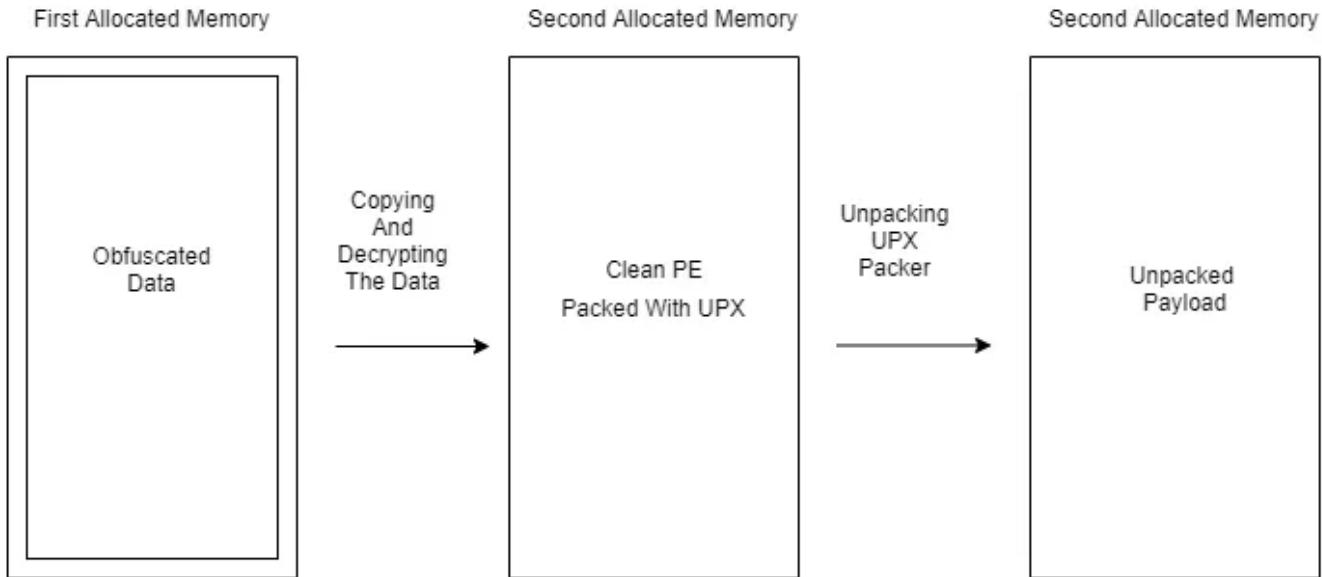
Address		Hex		ASCII	
002B0000	4D 5A 90 00	03 00 00 00 00	04 00 00 00 00	FF FF 00 00	MZ.....ÿ..
002B0010	B8 00 00 00	00 00 00 00 00	40 00 00 00 00	00 00 00 00 00@.....
002B0020	00 00 00 00	00 00 00 00 00	00 00 00 00 00	00 00 00 00 00
002B0030	00 00 00 00	00 00 00 00 00	00 00 00 00 00	08 01 00 00	...o...!..L!Th
002B0040	0E 1F BA 0E	00 B4 09 CD	21 B8 01 4C	CD 21 54 68	is program canno
002B0050	69 73 20 70	72 6F 67 72	61 6D 20 63	61 6E 6E 6F	t be run in DOS
002B0060	74 20 62 65	20 72 75 6E	20 69 6E 20	44 4F 53 20	mode....\$.....
002B0070	6D 6F 64 65	2E 0D 0D 0A	24 00 00 00	00 00 00 00	!`óÀ@Y«À@Y«À@Y«
002B0080	86 21 B3 F8	C2 40 DD AB	C2 40 DD AB	C2 40 DD AB	vÜ,«í@Y«vÜ.«z@Y«
002B0090	76 DC 2C AB	CF 40 DD AB	76 DC 2E AB	5A 40 DD AB	vÜ/«ß@Y«j.þ@ß@Y«
002B00A0	76 DC 2F AB	DF 40 DD AB	A1 1D DE AA	D4 40 DD AB	'.ð@A@Y«j.ð@A@Y«
002B00B0	27 19 D8 AA	C0 40 DD AB	A1 1D D8 AA	F7 40 DD AB	j.Ù@à@Y«.ð.«É@Y«
002B00C0	A1 1D D9 AA	E3 40 DD AB	1F BF 16 AB	CB 40 DD AB	A@Ü«K@Y«-.ð@A@Y«
002B00D0	C2 40 DC AB	4B 40 DD AB	AC 1D D4 AA	C6 40 DD AB	-.Ù@A@Y«-. "«A@Y«
002B00E0	AC 1D DD AA	C3 40 DD AB	AC 1D 22 AB	C3 40 DD AB	-.ð@A@Y«Rich@A@Y«
002B00F0	AC 1D DF AA	C3 40 DD AB	52 69 63 68	C2 40 DD AB	
002B0100	00 00 00 00	00 00 00 00	50 45 00 00	4C 01 03 00	PE L.....
002B0110	25 EE EF 5D	00 00 00 00	00 00 00 00	E0 00 02 21	%íi].....à, !
002B0120	0B 01 0E 00	00 C0 01 00	00 10 00 00	00 60 02 00À.....
002B0130	20 2C 04 00	00 70 02 00	00 30 04 00	00 00 00 10	, p. 0.....
002B0140	00 10 00 00	00 02 00 00	06 00 00 00	00 00 00 00	
002B0150	06 00 00 00	00 00 00 00	00 40 04 00	00 10 00 00	@.....
002B0160	00 00 00 00	02 00 40 01	00 00 10 00	00 10 00 00	@.....
002B0170	00 00 10 00	00 10 00 00	00 00 00 00	10 00 00 00	
002B0180	F0 32 04 00	50 00 00 00	DC 31 04 00	14 01 00 00	ð2. P. Ü1.....
002B0190	00 30 04 00	DC 01 00 00	00 00 00 00	00 00 00 00	0. Ü.....
002B01A0	00 00 00 00	00 00 00 00	40 33 04 00	24 00 00 00	@3. \$.....
002B01B0	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	
002B01C0	00 00 00 00	00 00 00 00	F4 2D 04 00	18 00 00 00	ð-.....
002B01D0	14 2E 04 00	5C 00 00 00	00 00 00 00	00 00 00 00	
002B01E0	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	
002B01F0	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	
002B0200	55 50 58 30	00 00 00 00	00 60 02 00	00 10 00 00	UPX0.....
002B0210	00 00 00 00	00 04 00 00	00 00 00 00	00 00 00 00	
002B0220	00 00 00 00	80 00 00 E0	55 50 58 31	00 00 00 00	UPX1.....

Get2: Second Allocated Memory Packed With UPX

The UPX packer is one of the most common open-source packers. For attackers, it's easy to use packer and it also provides reliability because of the fact that it is not a complex packer. There are several ways to overcome this packer, most of them are documented on the internet. In this case, I choose to use the UPX utility of CFF Explorer.

Note: In some recent versions of Get2 malware, the UPX part is missing.

The final stages of the unpacking mechanism are portrayed in the following diagram:



While inspecting the final unpacked file, we'll see it has low entropy and a large number of strings, functions names, and data.

indicators (5/33)			
virustotal (warning)			
dos-header (64 bytes)			
dos-stub (200 bytes)			
file-header (Dec.2019)			
optional-header (file-checksum)			
directories (6)			
sections (59.44%)			
libraries (1/4)			
imports (44/120)			
exports (racevr)			
tls-callbacks (n/a)			
resources (manifest)			
strings (59/2044)			
debug (n/a)			
manifest (asInvoker)			
version (n/a)			
certificate (n/a)			
	network	-	WinHttpAddRequestHeaders
	network	-	WinHttpCloseHandle
	network	-	WinHttpConnect
	network	-	WinHttpCrackUrl
	network	-	WinHttpGetIEProxyConfigForCurrentUser
	network	-	WinHttpGetProxyForUrl
	network	-	WinHttpOpen
	network	-	WinHttpOpenRequest
	network	-	WinHttpQueryDataAvailable
	network	-	WinHttpQueryHeaders
	network	-	WinHttpReadData
	network	-	WinHttpReceiveResponse
	network	-	WinHttpSendRequest
	network	-	WinHttpSetOption
	network	-	WinHttpSetTimeouts
	network	-	WinHttpWriteData

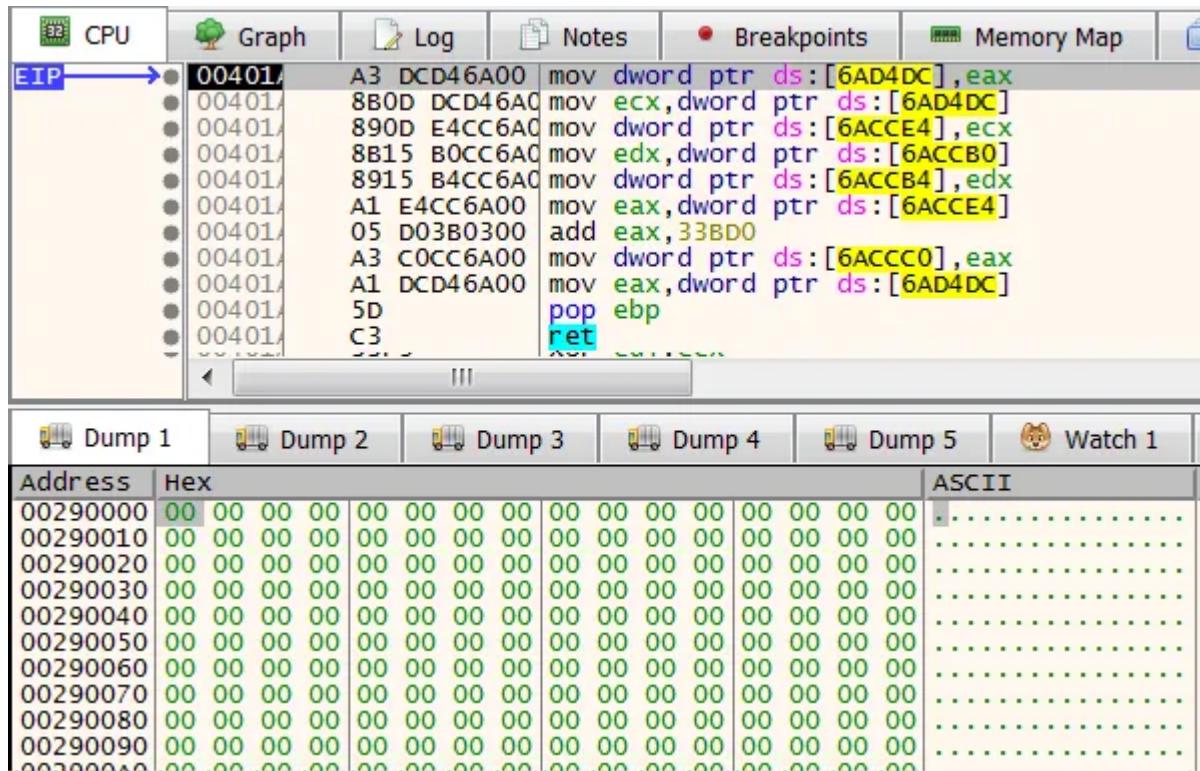
Get2: Final unpacked payload

Qbot

Qbot (aka QakBot, Pinkslipbot, QuakBot) is one of the known and prevalent banking trojans in the last years and been active for years since 2007.

The way to unpack the first dropper of Qbot is relatively simple and already been documented. Setting a breakpoint on VirtualProtect and inspecting several mov instructions after it, will do the trick. Otherwise, we can use automated tools such as PE-Sieve. However, where is the fun in that?

To start, we'll set a breakpoint on `VirtualAllocEx` and `VirtualAlloc`, then, click Run. We'll hit on `VirtualAllocEx`, and to assure nothing slips away, put a write hardware breakpoint on the newly allocated memory. Then, hit run.



Qbot: VirtualAllocEx allocated memory

We reach our breakpoint in a function that writes obfuscated data, byte by byte, into this allocated memory. Similar to the Get2 malware or basically any other malware that has an unpacking mechanism, the data is written inside a loop.

The screenshot shows the Immunity Debugger interface with the CPU tab selected. The assembly window displays the following code:

```
004019: BA F9 120000 mov edx,12F9  
004019: 85D2 test edx,edx  
004019: ^ 74 6E je qbot.4019BD  
004019: A1 9CD46A00 mov eax,dword ptr ds:[6AD49C]  
004019: 3B05 88D46A00 cmp eax,dword ptr ds:[6AD488]  
004019: ^ 72 02 jb qbot.40195E  
004019: ^ EB 5F jmp qbot.4019BD  
004019: 8B35 9CD46A00 mov esi,dword ptr ds:[6AD49C]  
004019: 0375 F4 add esi,dword ptr ss:[ebp-C]  
004019: 68 DC040000 push 4DC  
004019: 6A 00 push 0  
004019: FF15 9CCF6A00 call dword ptr ds:[<&LoadIconA>]  
004019: 03F0 add esi,eax  
004019: 68 DC040000 push 4DC  
004019: 6A 00 push 0  
004019: FF15 9CCF6A00 call dword ptr ds:[<&LoadIconA>]  
004019: 0375 F4 add esi,dword ptr ss:[ebp-C]  
004019: 03C6 add eax,esi  
004019: 0345 F4 add eax,dword ptr ss:[ebp-C]  
004019: 8B0D 9CD46A00 mov ecx,dword ptr ds:[6AD49C]  
004019: 034D F4 add ecx,dword ptr ss:[ebp-C]  
004019: 034D F4 add ecx,dword ptr ss:[ebp-C]  
004019: 034D F4 add ecx,dword ptr ss:[ebp-C]  
004019: 8B15 D0D46A00 mov edx,dword ptr ds:[6AD4D0]  
004019: 8B35 CCD46A00 mov esi,dword ptr ds:[6AD4CC]  
004019: 8A0406 mov al,byte ptr ds:[esi+eax]  
004019: 88040A mov byte ptr ds:[edx+ecx],al  
004019: 8B0D 9CD46A00 mov ecx,dword ptr ds:[6AD49C]  
004019: 83C1 01 add ecx,1  
004019: 890D 9CD46A00 mov dword ptr ds:[6AD49C],ecx  
004019: EB 89 jmp qbot.401946  
004019: 5E pop esi  
004019: 8BE5 mov esp,ebp  
004019: 5D pop ebp  
004019: C3 ret  
004019: CC int3
```

Qbot: Data Written In The Allocated Memory

In order to speed up the process, we'll set a breakpoint just after the loop. Then, an entire obfuscated content will be written.

Address	Hex	ASCII
00290000	83 51 61 34 CF 06 37 38 CB 00 31 32 D1 04 00 00	.Qa4Í.78È.12Ñ...
00290010	E2 30 00 31 8B 42 74 75 83 5C 41 6C 76 5F 63 00	à0.1.Btu.\Alv_c.
00290020	E2 30 00 00 E2 30 56 69 50 45 75 61 36 77 72 65	å0..å0ViPEuå6wre
00290030	47 31 00 00 E2 30 00 00 E2 63 6E 6D 43 41 56 69	G1..å0..åcnmCAVi
00290040	07 46 4F 66 1C 58 6C 65 E2 30 00 00 F5 57 72 74	.Fof.xleå0..öwr
00290050	F7 4F 6C 50 F0 5D 74 65 01 45 00 00 E2 30 00 4C	÷olPô]te.E..å0.L
00290060	CD 4F 64 4C CB 52 72 61 D0 47 45 78 A3 30 00 00	fodLÈRraåDGEx£0..
00290070	E2 30 47 65 AE 7B 6F 64 B7 5C 65 48 C3 5E 64 6C	å0Ge®{od.\ehA^dl
00290080	87 6F 00 00 E2 75 65 74 9F 5D 64 75 76 53 48 61	..åuet.]duvSHA
00290090	8C 54 6C 65 8D 30 00 00 A1 42 65 61 6E 53 46 69	.Tle...BeanSF
002900A0	4E 54 41 00 E2 30 00 00 E2 30 00 53 3E 45 46 69	NTA å0.å0.52EEF

Qbot: Obfuscated Data Written In The Allocated Memory

Note: Packed binaries tend to contain an embedded obfuscated content that will be read and written into newly allocated memory sections. Initial static analysis of Qbot shows the obfuscated content to be written.

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
0004A7B0	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90
0004A7C0	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90
0004A7D0	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90	90
0004A7E0	00	3E	03	00	83	51	61	34	CF	06	37	38	CB	00	31	32	.>..fQa4I.78E.12
0004A7F0	D1	04	00	00	E2	30	00	31	8B	42	74	75	83	5C	41	6C	Ñ...â0.1<Btuf\Al
0004A800	76	5F	63	00	E2	30	00	00	E2	30	56	69	50	45	75	61	v_c.â0..â0ViPEua
0004A810	36	77	72	65	47	31	00	00	E2	30	00	00	E2	63	6E	6D	6wreg1..â0..âcnm
0004A820	43	41	56	69	07	46	4F	66	1C	58	6C	65	E2	30	00	00	CAVi.FOf.Xleâ0..
0004A830	F5	57	72	74	F7	4F	6C	50	F0	5D	74	65	01	45	00	00	ÖWrt=OlPõ]te.E..
0004A840	E2	30	00	4C	CD	4F	64	4C	CB	52	72	61	D0	47	45	78	â0.LÍodLÉRraDGEx
0004A850	A3	30	00	00	E2	30	47	65	AE	7B	6F	64	B7	5C	65	48	£0..â0Ge@{od.\eH
0004A860	C3	5E	64	6C	87	6F	00	00	E2	75	65	74	9F	5D	64	75	Ã^dl+o..âuetÝ]du
0004A870	76	53	48	61	8C	54	6C	65	8D	30	00	00	A1	42	65	61	vSHaETle.0..;Bea
0004A880	6E	53	46	69	4E	54	41	00	E2	30	00	00	E2	30	00	53	nSFINTA.â0..â0.S
0004A890	3F	45	46	69	4E	54	50	6F	4B	5F	74	65	30	31	00	00	?EFINTPoK_te01..
0004A8A0	E2	30	57	72	0B	45	65	46	0B	5D	65	00	00	00	00	00	â0Wr.EeF.]e.....
0004A8B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Qbot: Static analysis — obfuscated content

Interestingly, this memory section is being manipulated again and again by several loops. Then, it being overwritten until the upper part is filled with the unknown string “aaa45678901234” followed by several names of API calls.

Address	Hex	ASCII
00290000	61 61 61 34 35 36 37 38 39 30 31 32 33 34 00 00	aaa45678901234..
00290010	00 00 00 31 69 72 74 75 61 6C 41 6C 6C 6F 63 00	...lirtualAlloc.
00290020	00 00 00 00 00 00 56 69 72 74 75 61 6C 46 72 65VirtualFre
00290030	65 00 00 00 00 00 00 00 00 00 55 6E 6D 61 70 56 69	e.....UnmapVi
00290040	65 77 4F 66 46 69 6C 65 00 00 00 00 6F 69 72 74	ewOfFile....oirt
00290050	75 61 6C 50 72 6F 74 65 63 74 00 00 00 00 00 4C	ualProtect....L
00290060	6F 61 64 4C 69 62 72 61 72 79 45 78 41 00 00 00	oadLibraryExA...
00290070	00 00 47 65 74 4D 6F 64 75 6C 65 48 61 6E 64 6C	..GetModuleHandl
00290080	65 41 00 00 00 47 65 74 4D 6F 64 75 6C 65 48 61	eA...GetModuleHa
00290090	6E 64 6C 65 57 00 00 00 43 72 65 61 74 65 46 69	ndlew...CreateFi
002900A0	6C 65 41 00 00 00 00 00 00 00 00 53 65 74 46 69	leA.....SetFi
002900B0	6C 65 50 6F 69 6E 74 65 72 00 00 00 00 00 57 72	lePointer....Wr
002900C0	69 74 65 46 69 6C 65 00 00 00 00 00 00 00 00 00	iteFile.....
002900D0	00 43 6C 6F 73 65 48 61 6E 64 6C 65 00 00 00 00	.CloseHandle....
002900E0	00 00 00 00 47 65 74 54 65 6D 70 50 61 74 68 41GetTempPathA
002900F0	00 00 00 00 00 00 00 6C 73 74 72 6C 65 6E 41 00lstrlenA.
00290100	00 00 00 00 00 00 00 00 00 00 6C 73 74 72 63 61lstrca

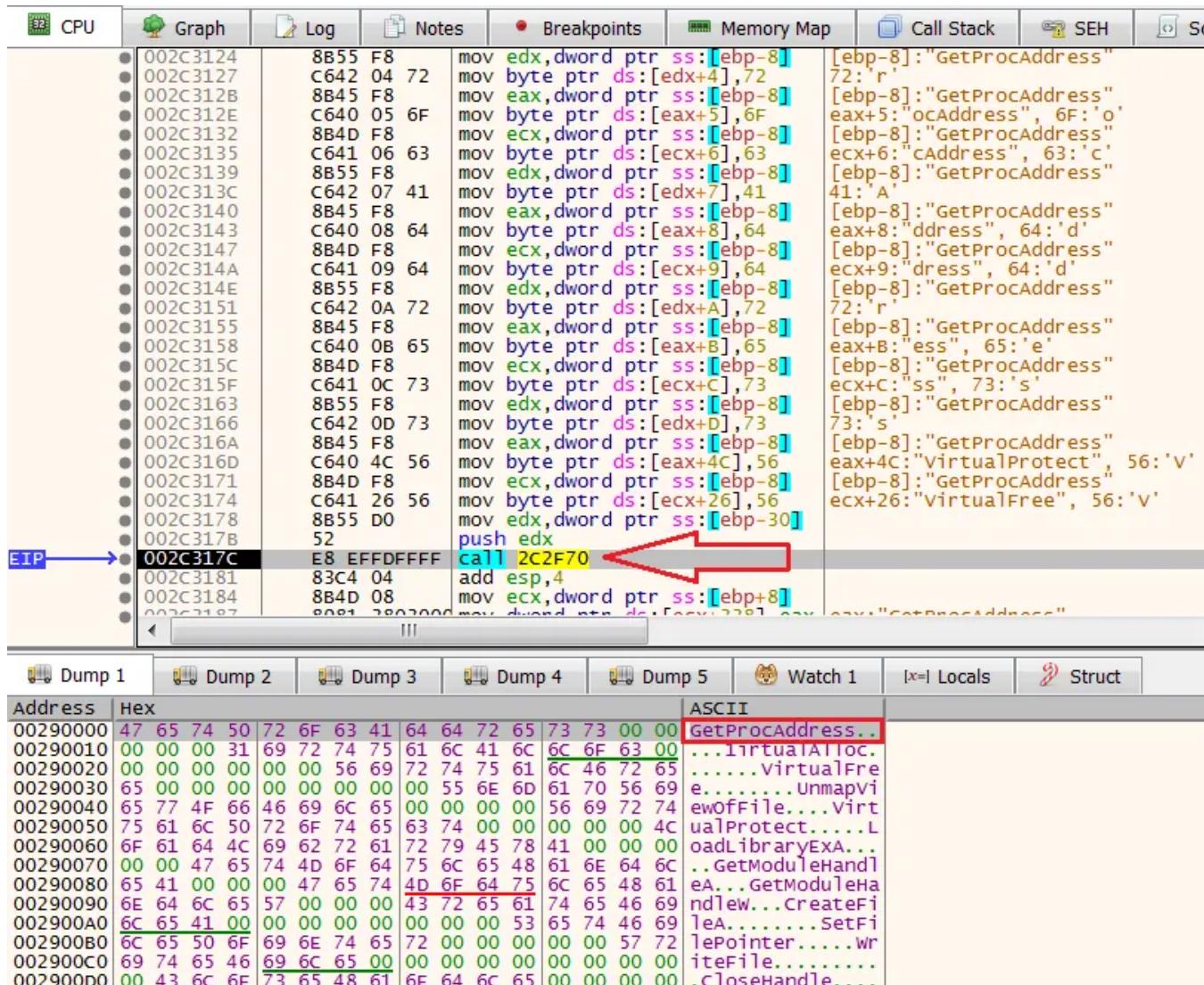
Qbot: Data being Overwritten In The Allocated Memory

The rest of the memory section appears to be more obfuscated. However, when taking a closer look, it does have fragments of strings that can indicate a potential encrypted PE file.

Qbot: Signs Of Obfuscated PE

Several instructions later, in the CPU window, we observe the string “aaa45678901234” is being modified by several *mov* instructions.

These instructions assign the HEX value “47 65 74 50 72 6f 63 41 64 64 72 65 73 73” to the ECX register (in ASCII, these bytes are translated to “*GetProcAddress*”). This technique is called Stack-Strings and will appear several times during the Qbot unpacking process. Then, the *GetProcAddress* string being used by another function.



Qbot: GetProcAddress Stack-Strings

Next, the packer call to another function that deals with *VirtualAlloc* by using the Stack-strings concept. As observed in the image below, it will store the *VirtualAlloc* string from *EBP-18* to *EBP-C*.

The screenshot shows the Qbot debugger interface. The assembly window displays the following code snippet:

```

    push ebp
    mov ebp,esp
    sub esp,1C
    mov byte ptr ss:[ebp-18],56
    mov byte ptr ss:[ebp-17],69
    mov byte ptr ss:[ebp-16],72
    mov byte ptr ss:[ebp-15],74
    mov byte ptr ss:[ebp-14],75
    mov byte ptr ss:[ebp-13],61
    mov byte ptr ss:[ebp-12],6C
    mov byte ptr ss:[ebp-11],41
    mov byte ptr ss:[ebp-10],6C
    mov byte ptr ss:[ebp-F],6C
    mov byte ptr ss:[ebp-E],6F
    mov byte ptr ss:[ebp-D],63
    mov byte ptr ss:[ebp-C],0
    push 0
    call 2C3600
    add esp,4
    mov dword ptr ss:[ebp-1C],eax
    cmp dword ptr ss:[ebp-1C],0
    jne 2C326A
    push 1
    call 2C3600
    add esp,4
    mov dword ptr ss:[ebp-1C],eax
    mov eax,dword ptr ss:[ebp-1C]
    push eax
    call 2C2F70
    add esp,4
    mov dword ptr ss:[ebp-4],eax
    lea ecx,dword ptr ss:[ebp-18]
    push ecx
    mov dword ptr ss:[ebp-1C],1C
  
```

The memory dump window shows the string "VirtualAlloc" at address 0018FB98:

Address	Hex	ASCII
0018FB98	F0 31 2C 00 00 00 59 75 A8 34 5A 75 00 00 59 75	ð1,... Yu ``4Zu..Yu
0018FBA8	00 F0 E8 FF 4C 6F 61 64 4C 69 62 72 61 72 79 45	.ðéùloadlibraryE
0018FBB8	78 41 00 75 56 69 72 74 75 61 6C 41 6C 6C 6F 63	xA. ðVirtualAlloc
0018FBC8	00 03 00 00 0F 00 00 00 0F 00 00 00 80 FF 18 00Y.
0018FBDB8	DF 3C 2C 00 00 2E 03 00 00 2E 03 00 00 F0 E8 FF	Þ<,...,.ðéý
0018FBE8	54 01 29 00 00 00 40 00 80 00 40 00 6F E3 49 75	T.)...@...@.oäIU
0018FBF8	80 FF 18 00 00 00 00 00 00 00 00 00 0F 01 00 00	.ý.....
0018FC08	00 00 00 00 D0 1C 40 00 00 00 40 00 00 F0 2A 00ð@...@.ð*
0018FC18	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0018FC28	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0018FC38	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0018FC48	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0018FC58	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Qbot: VirtualAlloc Stack-Strings

Then, the function will do the following:

1. Calling *GetProcAddress* (the string resides in *EBP-4*) and request *VirtualAlloc*.

2. *GetProcAddress* returns the address of *VirtualAlloc* (which is stored in *EAX*), and move it to *EBP-8*.
3. *VirtualAlloc* being called with Read-Write-Execute permissions.

Surprisingly, this call did not trigger the *VirtualAlloc* breakpoint we set at the beginning of our investigation.

```

002C326A 8B45 E4 mov eax,dword ptr ss:[ebp-1c]
002C326D 50 push eax
002C326E E8 FDFCFFFF call 2C2F70
002C3273 83C4 04 add esp,4
002C3276 8945 FC mov dword ptr ss:[ebp-4],eax
002C3279 CD4D E8 lea ecx,dword ptr ss:[ebp-18]
002C327C 51 push ecx
002C327D 8B55 E4 mov edx,dword ptr ss:[ebp-1c]
002C3280 52 push edx
002C3281 FF55 FC call dword ptr ss:[ebp-4]
002C3284 8945 F8 mov dword ptr ss:[ebp-8],eax
002C3287 6A 40 push 40
002C3289 68 00300000 push 3000
002C328E 8B45 08 mov eax,dword ptr ss:[ebp+8]
002C3291 50 push eax
002C3292 6A 00 push 0
002C3294 FF55 F8 call dword ptr ss:[ebp-8]
002C3297 8BE5 mov esp,ebp
002C3299 5D pop ebp
    
```

Qbot: Call For VirtualAlloc Steps

Similar to previous times, we'll set a breakpoint on the newly allocated memory, just in case we'll not miss any content that will be written there.

Address	Hex	ASCII
002D0000	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002D0010	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002D0020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002D0030	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002D0040	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002D0050	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002D0060	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002D0070	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002D0080	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002D0090	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002D00A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002D00B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002D00C0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Qbot: Newly Allocated Memory

Fortunately, after several instructions, we encounter a loop that starts to write content to the newly allocated memory. It starts copying the data stored in the *EDX* register, which points to an address found several offsets further in the first allocated memory. As we remember, we suspected it might contain encrypted PE.

The screenshot shows the Immunity Debugger interface. The CPU tab is active, displaying assembly code. The instruction at address 002C33EB is highlighted with a red box and an arrow from the EIP register. This instruction is a jump to address 002C33F4. The assembly code includes various stack operations like push and pop, and memory moves. Below the assembly window is a memory dump table with columns for Address, Hex, and ASCII. The first row shows memory starting at 002D0000 with hex values A4 59 90 00 EA 03 00 00 and ASCII Y..é.

Address	Hex	ASCII
002D0000	A4 59 90 00 EA 03 00 00 00 00 00 00 00 00 00 00	Y..é.....
002D0010	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002D0020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002D0030	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002D0040	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002D0050	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002D0060	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002D0070	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002D0080	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002D0090	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002D00A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002D00B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002D00C0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Qbot: Data Copied From First To Second Allocated Memory

As always, to speed things up, we'll set a breakpoint one step after the loop and hit Run. We can see that the entire part from the first allocated memory has been copied into the second allocated memory.

Dump 1	Dump 2	Dump 3	Dump 4	Dump 5	Watch 1
Address	Hex				
002D0000	A4 59 90 00	EA 03 00 00	ED 03 00 00	FE FB 00 00	Y..é..í..þ..
002D0010	31 03 00 00	E9 03 00 00	29 04 00 00	E9 03 00 00	1..é..).. é...
002D0020	E9 03 00 00	E9 03 00 00	E9 03 00 00	E9 03 00 00	é...é...é...é...
002D0030	E9 03 00 00	E9 03 00 00	E9 03 00 00	89 04 00 00	é...é...é.....
002D0040	E7 1A BA 0E	E9 AF 09 CD	C8 BB 01 4C	AC 25 54 68	ç..é..fÈ».L-%Th
002D0050	00 77 20 70	FB 6A 67 72	C8 68 20 63	C8 69 6E 6F	.w pûjgrÈh cÈino
002D0060	DD 23 62 65	09 76 75 6E	09 6D 6E 20	A5 4A 53 20	Ý#be.vun.mn ¥J5
002D0070	C4 6A 64 65	FF 08 0D 0A	CD 03 00 00	E9 03 00 00	Ájdeý...f..é..
002D0080	B6 53 4E 0F	F2 34 20 5C	E2 34 20 5C	E2 34 20 5C	ÍSN.ó4 \â4 \â4 \
002D0090	7D 50 25 5D	D3 34 20 5C	5D 51 23 5D	00 35 20 5C	}P%]ó4 \]Q#[.5 \
002D00A0	5D 51 24 5D	DE 34 20 5C	E2 34 21 5C	03 35 20 5C]Q\$]P4 \â4!.5 \
002D00B0	3D 51 21 5D	CD 34 20 5C	1D 51 28 5D	03 35 20 5C	=Q!]f4 \.Q(.5 \
002D00C0	1D 51 DF 5C	F3 34 20 5C	FD 50 22 5D	E3 34 20 5C	.Qß\ó4 \ýP"]ã4 \

Qbot: Data Copied From First To Second Allocated Memory

Then, we encounter another loop. This loop deobfuscates the entire second allocated memory. It is noticed by observing the magic string -"MZ" that appears after the first iterations.

The screenshot shows the Immunity Debugger interface. The CPU tab is selected, displaying assembly code. The instruction at address 002C3ADA is highlighted in yellow and has a blue arrow pointing to it from the left. The assembly code for this instruction is:

```

    EB C1 jmp 2C3A9D
    8BE5
    5D
    C3
    55
    8BEC
    51
    8B45 08
    8945 FC
    EB 09
    8B4D 5C
  
```

The instruction at address 002C3A9D is also highlighted in yellow and has a blue arrow pointing to it from the left. The assembly code for this instruction is:

```

    C745 FC 0000 mov dword ptr ss:[ebp-4],0
    EB 09 jmp 2C3AA6
    8B45 FC mov eax,dword ptr ss:[ebp-4]
    83C0 04 add eax,4
    8945 FC mov dword ptr ss:[ebp-4],eax
    8B4D FC mov ecx,dword ptr ss:[ebp-4]
    3B4D 0C cmp ecx,dword ptr ss:[ebp+C]
    73 2E jae 2C3ADC
    8B55 08 mov edx,dword ptr ss:[ebp+8]
    0355 FC add edx,dword ptr ss:[ebp-4]
    8B02 mov eax,dword ptr ds:[edx]
    0345 FC add eax,dword ptr ss:[ebp-4]
    8B4D 08 mov ecx,dword ptr ss:[ebp+8]
    034D FC add ecx,dword ptr ss:[ebp-4]
    8901 mov dword ptr ds:[ecx],eax
    8B55 FC mov edx,dword ptr ss:[ebp-4]
    81C2 E9030000 add edx,3E9
    8B45 08 mov eax,dword ptr ss:[ebp+8]
    0345 FC add eax,dword ptr ss:[ebp-4]
    3310 xor edx,dword ptr ds:[eax]
    8B4D 08 mov ecx,dword ptr ss:[ebp+8]
    034D FC add ecx,dword ptr ss:[ebp-4]
    8911 mov dword ptr ds:[ecx],edx
  
```

The memory dump tab (Dump 1) shows the first 16 bytes of memory starting at address 002D0000. The MZ header is highlighted with a red box.

Address	Hex	ASCII
002D0000	4D 5A 90 00 EA 03 00 00 ED 03 00 00 FE FB 00 00	MZ .ê...í...þü..
002D0010	31 03 00 00 E9 03 00 00 29 04 00 00 E9 03 00 00	1...é...é...é...
002D0020	E9 03 00 00 E9 03 00 00 E9 03 00 00 E9 03 00 00	é...é...é...é...
002D0030	E9 03 00 00 E9 03 00 00 E9 03 00 00 89 04 00 00	é...é...é...é....
002D0040	E7 1A BA 0E E9 AF 09 CD C8 BB 01 4C AC 25 54 68	ç.º.é..fÈ».L-%Th
002D0050	00 77 20 70 FB 6A 67 72 C8 68 20 63 C8 69 6E 6F	.w þújgrÈh cÈino
002D0060	DD 23 62 65 09 76 75 6E 09 6D 6E 20 A5 4A 53 20	Ý#be.vun.mn ¥JS
002D0070	C4 6A 64 65 FF 08 0D 0A CD 03 00 00 E9 03 00 00	Ájdeý...f...é...
002D0080	B6 53 4E 0F F2 34 20 5C E2 34 20 5C E2 34 20 5C	SN.ð4 \â4 \â4 \
002D0090	7D 50 25 5D D3 34 20 5C 5D 51 23 5D 00 35 20 5C	}P%]ð4 \]Q#].5 \
002D00A0	5D 51 24 5D DE 34 20 5C E2 34 21 5C 03 35 20 5C]Q\$]þ4 \â4!\.5 \
002D00B0	3D 51 21 5D CD 34 20 5C 1D 51 28 5D 03 35 20 5C	=Q!]f4 \.Q(.5 \
002D00C0	1D 51 DF 5C F3 34 20 5C FD 50 22 5D E3 34 20 5C	.QB\ð4 \yP"]â4 \

Qbot: Second Allocated Memory Deobfuscated

This loop contains several characteristics that already observed in the Get2 malware unpacking mechanism. The use of *xor* and *add* opcodes.

As always, when we encounter these types of loops we'll set a breakpoint step after the loop to get the final result, which is a clean portable executable.

The screenshot shows the assembly and memory dump windows of IDA Pro. The assembly window displays the following code:

```

002C3ADA EB C1 jmp 2C3A9D
002C3ADC 8BE5 mov esp,ebp
002C3ADE 5D pop ebp
002C3ADF C3 ret
002C3AE0 55 push ebp
002C3AE1 8BEC mov ebp,esp
002C3AE3 51 push ecx

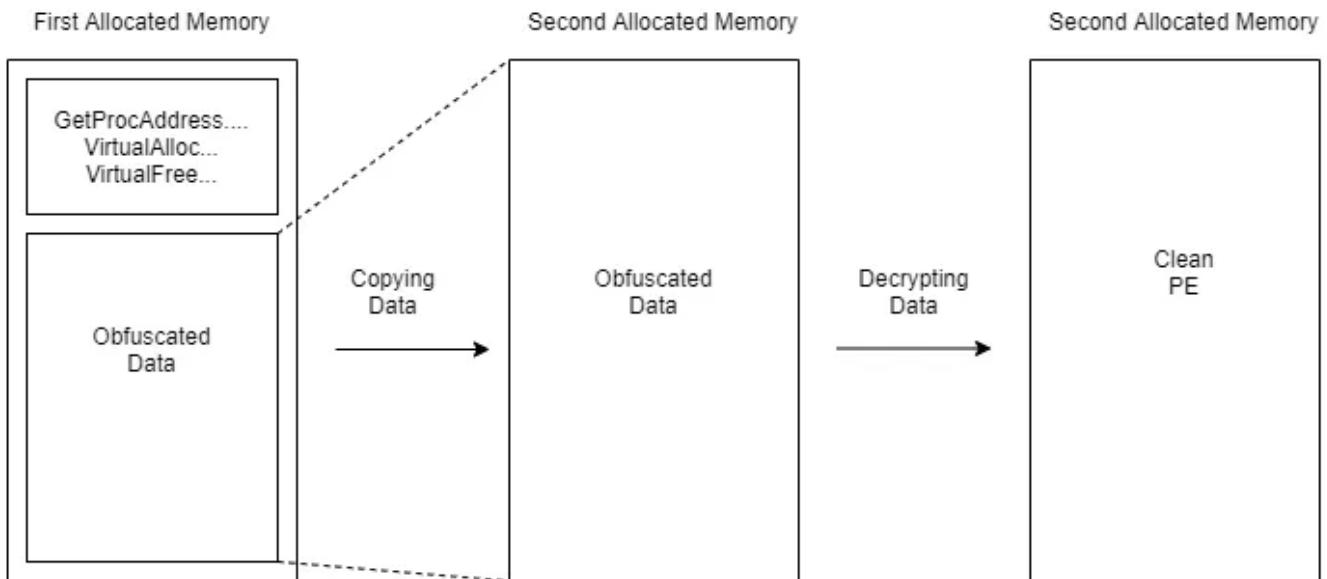
```

The memory dump window shows the raw hex and ASCII data for the program's memory space, starting with the MZ header:

Address	Hex	ASCII
002D0000	4D 5A 90 00	MZ.....
002D0010	B8 00 00 00	YY...@.....
002D0020	00 00 00 00@.....
002D0030	00 00 00 00@.....à
002D0040	0E 1F BA 0E	..°..!..Lf!Th
002D0050	69 73 20 70	is program canno
002D0060	72 6F 67 72	t be run in DOS
002D0070	61 6D 20 63	mode....\$
002D0080	61 6E 6E 6F	_PN..1 \.1 \.1 \
002D0090	74 55 25 5D	TU%].1 \tu#].1 \
002D00A0	1A 31 20 5C	tu\$].1 \.1!\:1 \
002D00B0	74 55 21 5D	tu!].1 \tuC].1 \
002D00C0	1C 31 20 5C	tuB\].1 \tu"].1 \
002D00D0	74 55 DF 5C	Rich.1 \.....
002D00E0	1B 31 20 5C	PE..L...z2_
002D00F0	00 00 00 00à.....;

Qbot: Unpacked PE

The final stages of the unpacking mechanism are portrayed in the following diagram:



Now, we'll dump this memory section and inspect it with tools such as [IDA](#) or PEstudio. The second stage of Qbot is known to contain its further payload in its resource section, and also contain [RC4](#) encryption and BLZPack decompression.

	file type	size	status	hashes	entropy	language	first bytes hex	first bytes text
1	PE32 executable for Win32 (GUI) (S)	102400	neutral	D3AD502FB57B6...	2.748	neutral	00 00 01 00 01 00 20 40 00 00...@.....
2	PE32 executable for Win32 (GUI) (S)	102400	neutral	F6C5998970A01...	2.944	neutral	00 00 01 00 01 00 10 20 00 00...(..
3	PE32 executable for Win32 (GUI) (S)	102400	neutral	92BC614EBE97C...	1.891	neutral	00 00 01 00 01 00 30 60 00 00...0`.....
4	PE32 executable for Win32 (GUI) (S)	102400	neutral	LC8AADAC8BFAB...	1.611	neutral	00 00 01 00 01 00 28 50 00 00...(P.....
5	PE32 executable for Win32 (GUI) (S)	102400	neutral	171146D2255AA...	1.333	neutral	00 00 01 00 01 00 14 28 00 00...(.....
6	PE32 executable for Win32 (GUI) (S)	102400	neutral	DB20D0FFF006AB...	1.440	neutral	00 00 01 00 01 00 10 20 00 00...h.....
7	PE32 executable for Win32 (GUI) (S)	102400	neutral	A73182CF0C90...	7.998	neutral	96 73 A2 6E B4 72 93 D0 6D 3...	s..n..r...m 9.....;

	file type	size	status	hashes	entropy	language	first bytes hex	first bytes text
1	PE32 executable for Win32 (GUI) (S)	102400	neutral	D3AD502FB57B6...	2.748	neutral	00 00 01 00 01 00 20 40 00 00...@.....
2	PE32 executable for Win32 (GUI) (S)	102400	neutral	F6C5998970A01...	2.944	neutral	00 00 01 00 01 00 10 20 00 00...(..
3	PE32 executable for Win32 (GUI) (S)	102400	neutral	92BC614EBE97C...	1.891	neutral	00 00 01 00 01 00 30 60 00 00...0`.....
4	PE32 executable for Win32 (GUI) (S)	102400	neutral	LC8AADAC8BFAB...	1.611	neutral	00 00 01 00 01 00 28 50 00 00...(P.....
5	PE32 executable for Win32 (GUI) (S)	102400	neutral	171146D2255AA...	1.333	neutral	00 00 01 00 01 00 14 28 00 00...(.....
6	PE32 executable for Win32 (GUI) (S)	102400	neutral	DB20D0FFF006AB...	1.440	neutral	00 00 01 00 01 00 10 20 00 00...h.....
7	PE32 executable for Win32 (GUI) (S)	102400	neutral	A73182CF0C90...	7.998	neutral	96 73 A2 6E B4 72 93 D0 6D 3...	s..n..r...m 9.....;

Qbot: Qbot's Second Stage Payload At The Resource Section

IcedID

IcedID aka Bokbot is also one of the most prevalent banking trojans in the last years. It is known to be associated with Lunar Spider threat actors.

Unpacking the first dropper of IcedID is pretty simple, similar to Qbot, we only need to set breakpoints on *VirtualAlloc* and *VirtualProtect*. In the unpacking mechanism, the second time *VirtualProtect* will be called it will indicate the unpacked IcedID. As said, we'll focus on how the IcedID unpacking mechanism works rather than getting the fastest way to unpack it.

As we start, by setting a breakpoint on *VirtualAlloc* and *VirtualProtect*, we'll notice a difference between the IcedID and Qbot, Get2 unpacking mechanisms.

Traditionally, we expect to see a memory allocation before everything else. In IcedID it's not the case- *VirtualProtect* is being called before *VirtualAlloc*.

By inspecting the “*lpAddress*” argument in *VirtualProtect* we'll notice that it appears to deal with shellcode execution. This is visible with the byte sequence E8 00 00 00 00 which is a relative call for the next instruction.

In this writeup, I won't cover the entire shellcode investigation, only the unpacking mechanism. For those of you who want to explore it, you can click on the shellcode memory address, press “follow in disassembler”, set a breakpoint on the relative call, and hit Run.

edi=509F9C63

.text:755A4347 kernel32.dll:\$14347 #14347 <virtualProtect>

Address	Hex	ASCII
01039068	B9 FB 31 00 00	E8 00 00 00 00
01039078	C7 EF 0A BF BC	4A 5E 29 31 C1 DB C3 00 00 B9 BC
01039088	09 00 00 BA 97 FD 00 00	89 FA 31 DB 66 81 EE 41
01039098	13 89 CE 83 E6 03 75 18	66 BB 87 38 89 FB 66 01
010390A8	DA F6 DA 6B D2 03 C1 CA	08 66 81 C7 45 09 89 D7
010390B8	30 10 40 E2 D7 E9 46 05	00 90 3F 3F 3F B1
010390C8	AC A6 A6 56 C9 E5 E5 2B	01 2B 2B 92 28 31 31 83
010390D8	E2 F8 FB B3 3B 3B 3B 8D	61 A7 D0 35 36 21 36 62
010390E8	7D 6A 63 6E 62 76 69 68	68 6D 6C 51 55 5D 55 2D
010390F8	2E 26 2A 8C 8F FF FC 93	93 93 93 CF C9 C9 C9 82
01039108	14 0R 0C 24 C9 20 18 AC	6A 02 02 38 49 F6 C7 22

IcedID: Shellcode

Then, we'll hit Run again and encounter *VirtualAlloc* three different times. In the third time, we'll set a write hardware breakpoint on the newly allocated memory.

dword_ptr [ebx+17A5A2]=[002A0574]=0
eax=002C0000

002A06FF

Address	Hex	ASCII
002C0000	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002C0010	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002C0020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002C0030	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002C0040	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002C0050	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002C0060	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002C0070	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002C0080	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002C0090	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002C00A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

IcedID: Call To VirtualAlloc

Next, we encounter a function, which has a loop that copies data to the newly allocated memory. Interestingly, in the previous malware, when we saw data moved

it always copied byte by byte. Nonetheless, in this case, we see that the copy is managed by an instruction called *rep movsb*. This instruction by default moving data from the *ESI* to the *EDI* register, which points to the newly allocated memory.

The screenshot shows a debugger interface with several tabs: CPU, Graph, Log, Notes, Breakpoints, Memory Map, Call Stack, and Registers. The CPU tab displays assembly code, and the Registers tab shows various processor registers with their current values. A red box highlights the *rep movsb* instruction at address 002A02C0, which is part of a larger sequence of instructions for copying data from memory to the stack.

Address	Hex	ASCII
002C0000	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002C0010	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002C0020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
002C0030	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

IcedID: Data Copied To The Allocated Memory

This opcode of data transfer will be used multiple times in the IcedID unpacking mechanism.

Dump 1	Dump 2	Dump 3	Dump 4	Dump 5	Watch 1
Address	Hex	ASCII			
010094DE	7F 0A 68 A5 0D 00 00 FF 00 00 36 53 BF B9 B4 CC	.h¥..ÿ..65z'í			
010094EE	00 00 00 00 8B F5 CC B6 55 C5 84 06 1B C8 D6 07	...öíUA...ÉÖ.			
010094FE	C6 D4 C5 60 AF FF FF 00 00 00 FF 00 00 00 57 E3	ÆÓA`ÿy..ÿ..wã			
0100950E	5E D9 35 AC 00 FF 00 00 FF 00 00 00 FF 00 00 00 00	^Ù5~.ÿ.yÿ..ÿ..			
0100951E	00 15 1C 90 5A 04 71 70 6A 39 04 7A 78 65 BB BBZ.qpj9.zxe»»			
0100952E	00 FF 00 FF 00 BD B1 5C B8 9C B5 6E 3E 05 96 3C	.ÿ.ÿ.%±,.µn>..<			
0100953E	3B F1 91 E9 C3 00 00 00 00 00 00 00 00 FF FF 00	;ñ.éÁ..ÿy.			
0100954E	FF 00 04 F5 ED AF A4 95 44 00 FF 00 FF 00 FF 00	ÿ..öí~¤.D.ÿ.ÿ.ÿ.			
0100955E	14 7A 78 74 75 3E 1D 23 A0 8E E6 DD D6 98 00 00	.zxtu>.# .æÝO...			
0100956E	00 FF 8D 20 FE 0C 72 93 70 82 00 00 00 FF 00 00	.ÿ.þ.r.p..ÿ..			
0100957F	FF 00 00 FF 00 FF DF 1C C1 95 FF 00 00 00 FF B2	v..v.vþ.Á.v..v²			

ESI Register

Dump 1	Dump 2	Dump 3	Dump 4	Dump 5	Watch 1
Address	Hex	ASCII			
002C0000	7F 0A 68 A5 0D 00 00 FF 00 00 36 53 BF B9 B4 CC	.h¥..ÿ..65z'í			
002C0010	00 00 00 00 8B F5 CC B6 55 C5 84 06 1B C8 D6 07	...öíUA...ÉÖ.			
002C0020	C6 D4 C5 60 AF FF FF 00 00 00 FF 00 00 00 57 E3	ÆÓA`ÿy..ÿ..wã			
002C0030	5E D9 35 AC 00 FF 00 FF FF 00 00 00 FF 00 00 00 00	^Ù5~.ÿ.yÿ..ÿ..			
002C0040	00 15 1C 90 5A 04 71 70 6A 39 04 7A 78 65 BB BBZ.qpj9.zxe»»			
002C0050	00 FF 00 FF 00 BD B1 5C B8 9C B5 6E 3E 05 96 3C	.ÿ.ÿ.%±,.µn>..<			
002C0060	3B F1 91 E9 C3 00 00 00 00 00 00 00 00 FF FF 00	;ñ.éÁ..ÿy.			
002C0070	FF 00 04 F5 ED AF A4 95 44 00 FF 00 FF 00 FF 00	ÿ..öí~¤.D.ÿ.ÿ.ÿ.			
002C0080	14 7A 78 74 75 3E 1D 23 A0 8E E6 DD D6 98 00 00	.zxtu>.# .æÝO...			
002C0090	00 FF 8D 20 FE 0C 72 93 70 82 00 00 00 FF 00 00	.ÿ.þ.r.p..ÿ..			
002C00A0	FF 00 00 FF 00 FF DF 1C C1 95 FF 00 00 00 FF B2	v..v.vþ.Á.v..v²			

EDI register

IcedID: Data Comparison ESI vs EDI (Identical)

We'll hit Run again until we found ourselves in another data manipulating loop. This loop is significantly smaller and appears to modify the data using *xor* and *ror* (rotate right) opcodes. While modifying, we notice some signs of a portable executable, with the strings "M8Z" (as we saw in the Get2 malware).

The screenshot shows the Immunity Debugger interface. The assembly window displays the following code snippet:

```

002A01 5F          pop edi
002A01 5E          pop esi
002A01 01E8        add eax,ebp
002A01 8D342A      lea esi,dword ptr ds:[edx+ebp]
002A01 ^ EB 9F     jmp 2A026B
002A01 5F          pop edi
002A01 5E          pop esi
002A01 5D          pop ebp
002A01 5B          pop ebx
002A01 83C4 08     add esp,8
002A01 C3          ret
002A01 55          push ebp
002A01 89E5        mov ebp,esp
002A01 8B45 08     mov eax,dword ptr ss:[ebp+8]
002A01 8B4D 0C     mov ecx,dword ptr ss:[ebp+C]
002A01 8B55 10     mov edx,dword ptr ss:[ebp+10]
002A01 31DB        xor ebx,ebx
002A01 89CE        mov esi,ecx
002A01 83E6 03     and esi,3
002A01 ^ 75 11     jne 2A02FA
002A01 8B5D 10     mov ebx,dword ptr ss:[ebp+10]
002A01 66:01DA     add dx,bx
002A01 F6DA        neg dl
002A01 6BD2 03     imul edx,edx,3
002A01 C1CA 08     ror edx,8
002A01 8955 10     mov dword ptr ss:[ebp+10],edx
002A01 3010        xor byte ptr ds:[eax],dl
002A01 40          inc eax
EIP → 002A02 E2 E3  Loop 2A02E2
002A02 C9          leave
002A02 C2 0C00      ret c
002A02 83EC 10     sub esp,10
002A02 53          push ebx
002A02 55          push ebp
002A02 56          push esi
002A02 8B7424 24    mov esi,dword ptr ss:[esp+24]

```

The memory dump window shows the following hex and ASCII data:

Address	Hex	ASCII
002C0000	4D 38 5A A5 0D 36 53 BF B9 B4 CC 8B F5 CC B6 55	M8Z...65i'í.öí!u
002C0010	C5 84 06 1B C8 D6 07 C6 D4 C5 60 AF 57 E3 5E D9 A...E0.ÆØA wà^Ù	
002C0020	35 AC 15 1C 90 5A 04 71 70 6A 39 04 7A 78 65 BB 5...z.qpj9.zxe»	
002C0030	BB BD B1 5C B8 9C B5 6E 3E 05 96 3C 3B F1 91 E9 >%±\..µn>..<;ñ.é	
002C0040	C3 04 F5 ED AF A4 95 44 14 7A 78 74 75 3E 1D 23 Å.öí¤.D.zxtu>.#	
002C0050	A0 8E E6 DD D6 98 8D 20 FE 0C 72 93 70 82 DE 1C .æYö.. b.r.p.p.	

IcedID: Data being decrypted

As always, to speed up the process we'll set a breakpoint one step after this loop and we'll hit Run. Once we did that, we can see that some bytes have been changed. Still, it's not a clean portable executable.

002C0000	4D 38 5A 90 38 03 66 02 04 09 71 FF 81 B8 C2 91	M8Z.8.f...qý.Å.
002C0010	01 40 C2 15 C6 D8 09 1C 0E 1F BA F8 00 B4 09 CD	.@Å.ÆØ....°ø. .í
002C0020	21 B8 01 4C C0 0A 54 68 69 73 20 0E 70 72 6F 67	!. LÀ. This .prog
002C0030	67 61 6D 87 63 47 6E 1F 4F 74 E7 62 65 AF CF 75	gäm.cGn.Otçbe -ü
002C0040	5F 98 69 06 44 4F 7E 53 03 6D 6F 64 65 2E 0D 89	_.i.DO~S.mode..
002C0050	0A 24 4C 44 4F 01 14 27 F9 0B 75 49 AA 58 04 E3	.\$LDO.. 'ù.uI¤X.å
002C0060	3E 6A 4D 30 09 11 78 17 48 70 AB AB 14 C7 C6 39	>jMO..x.Hp¤¤.ç¤9
002C0070	11 04 D9 11 4A AB 0A 4F ED 0F 41 08 02 6F 94 10	..Û. J¤.Oí.A..o..
002C0080	CA 4B 08 82 52 69 63 68 88 40 9C 50 50 45 80 4C	ÉK..Rich.@.PPE.L
002C0090	01 A0 D6 80 59 2B 8C 5E 14 1C E0 00 02 21 0B 01	. Ö.Y+.^.à..!..
002C00A0	0F 0C F0 18 A4 B9 DF 09 F0 37 10 14 11 5C 09 30	..à.¤íR.à7...!.0

IcedID: Data being decrypted

Then, we observe something unusual. We can see again the *movsb* opcode in a larger loop, but now it copies data from the upper part of the allocated memory (the one from the image above) which is stored in the *ESI* register, to an address in the same memory space located several offsets after (which is also stored in the *EDI* register).

After a few iterations and data manipulation functions, we observe the bytes 4D 5A and the string “MZ” at the beginning of the memory to which the *movsb* opcode writes.

The screenshot shows the Immunity Debugger interface with the CPU tab selected. The assembly window displays the following code:

```

002A0: 41           inc    ecx
002A0: B0 10        mov    al, 10
002A0: E8 4F000000  call   2A0236
002A0: 10C0         adc    al, al
002A0: ^ 73 F7      jae   2A01E2
002A0: < 75 3F      jne   2A022C
002A0: AA           stosb 
002A0: ^ EB D4      jmp   2A01C4
002A0: E8 4D000000  call   2A0242
002A0: 29D9         sub    ecx, ebx
002A0: < 75 10      jne   2A0209
002A0: E8 42000000  call   2A0240
002A0: < EB 28      jmp   2A0228
002A0: AC           lodsb 
002A0: D1E8         shr    eax, 1
002A0: < 74 4D      je    2A0252
002A0: 11C9         adc    ecx, ecx
002A0: < EB 1C      jmp   2A0225
002A0: 91           xchg   ecx, eax
002A0: 48           dec    eax
002A0: C1E0 08       shl    eax, 8
002A0: AC           lodsb 
002A0: E8 2C000000  call   2A0240
002A0: 3D 007D0000  cmp    eax, 7D00
002A0: < 73 0A      jae   2A0225
002A0: 80FC 05       cmp    ah, 5
002A0: < 73 06      jae   2A0226
002A0: 83F8 7F       cmp    eax, 7F
002A0: < 77 02      ja    2A0227
002A0: 41           inc    ecx
002A0: 41           inc    ecx
002A0: 95           xchg   ebp, eax
002A0: 89E8         mov    eax, ebp
002A0: B3 01         mov    bl, 1
002A0: 56           push   esi
002A0: 89FE         mov    esi, edi
002A0: 29C6         sub    esi, eax
002A0: F3:A4         rep    movsb
002A0: 5E           pop    esi
002A0: ^ EB 8E        jmp   2A01C4
002A0: 00D2         add    dl, dl

```

The Registers pane shows the following values:

- EAX: 00000001
- EBX: 00000002
- ECX: 00000000
- EDX: 002C1980
- EBP: 00000004
- ESP: 0010E28C
- ESI: 002C000C (highlighted)
- EDI: 002C19B1 (highlighted)
- EIP: 002A0234
- EFLAGS: 00000216
- ZF: 0 PF: 1 AF: 1
- OF: 0 SF: 0 DF: 0
- CF: 0 TF: 0 IF: 1
- Last Error: 000000
- Last Status: C00001
- GS: 002B FS: 0053
- ES: 002B DS: 002B
- CS: 0023 SS: 002B
- DR0: 002C0000
- DR1: 002C19A4
- DR2: 00000000
- DR3: 00000000
- DR6: FFFF4FF0
- DR7: 00110005

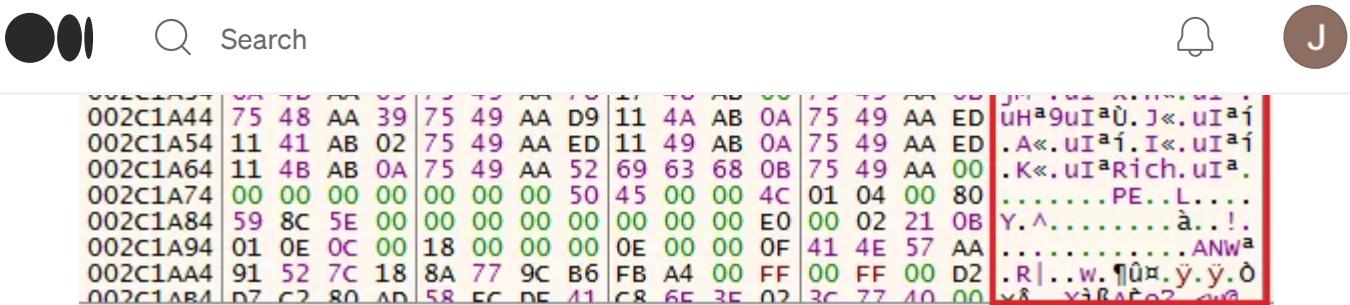
The Stack pane shows the following dump:

Address	Hex	ASCII
002C1974	A3 56 B3 5B C3 72 D3 79 E3 9F F3 C8 F3 CF F2 F1	fv [ArÓyå. óÉóÍöô
002C1984	49 38 18 C9 6F D1 96 D9 9C E1 A7 E9 AE F1 DD F9	T8. ËoÑ. l. ásé®ñýù
002C1994	E4 F9 F6 F9 36 C5 3D CD CA D6 47 FF D4 C0 00 4D	äùöù6A=fÉÖGyðA.M
002C19A4	5A 90 00 03 00 00 00 04 00 00 00 FF FF 4C DD 08	z.....yyLý.
002C19B4	94 1E 12 00 FF 00 FF 00 8B D9 D0 6C 4E 75 EE 72	...y.y..UÐINuIr
002C19C4	20 AD 9B C4 E5 E1 CA 00 00 00 00 88 FC 65 F9 F8	..ÁaaÆ.....üeùø
002C19D4	0E 22 36 B4 F4 EE 44 A4 4A 07 A5 00 FF 00 00 00	.."6' ÓID¤J.¥.ý...
002C19E4	00 00 FF FF B3 A7 C8 CA CF B2 86 00 FF 00 00 00	ÿy'SÉEÍ²..ý...
002C19F4	FF 00 00 00 FF 00 00 FF 66 C0 C4 73 39 32 CF 43	ÿ..y..ýfääs92Íc
002C1A04	91 B7 C8 C1 E3 46 FF 00 00 FF 00 4B 8D CE 9C E6	.ÉAäFy..ý.K.íæ
002C1A14	F4 51 0E 00 00 00 00 FF 00 00 00 FF 00 FF 00 00	ôQ.....ý..ý..ý..
002C1A24	64 5A 40 8A 00 FF FF 00 00 FF 00 DF B0 14 48 53	dz@..ýy..ý.B..HS
002C1A34	A9 9B EB 79 1E EE B5 0A C7 40 62 00 FF FF 00 F3	@.ëy.íµ.C@b.ýy.ó
002C1A44	A8 2C 63 42 35 D0 7C 6E A5 1C DD 46 D1 15 00 00	..,cB5D n¥.YFN..
002C1A54	FF 00 00 00 00 00 FF 00 FF 00 DD 78 5A 45 A5 FF	ÿ.....ý.ý.ÝxZEÝy
002C1A64	00 00 00 00 F3 90 D2 C9 27 99 FF 00 00 FF 29 E4ó.OÉ .ý..ý)å
002C1A74	DF 27 3C E7 6E 81 8E 00 01 64 FF A1 55 85 79 00	ß'<çn....dyiU:y

IcedID: Data Copied And Decrypted In The Upper Part Of The Memory

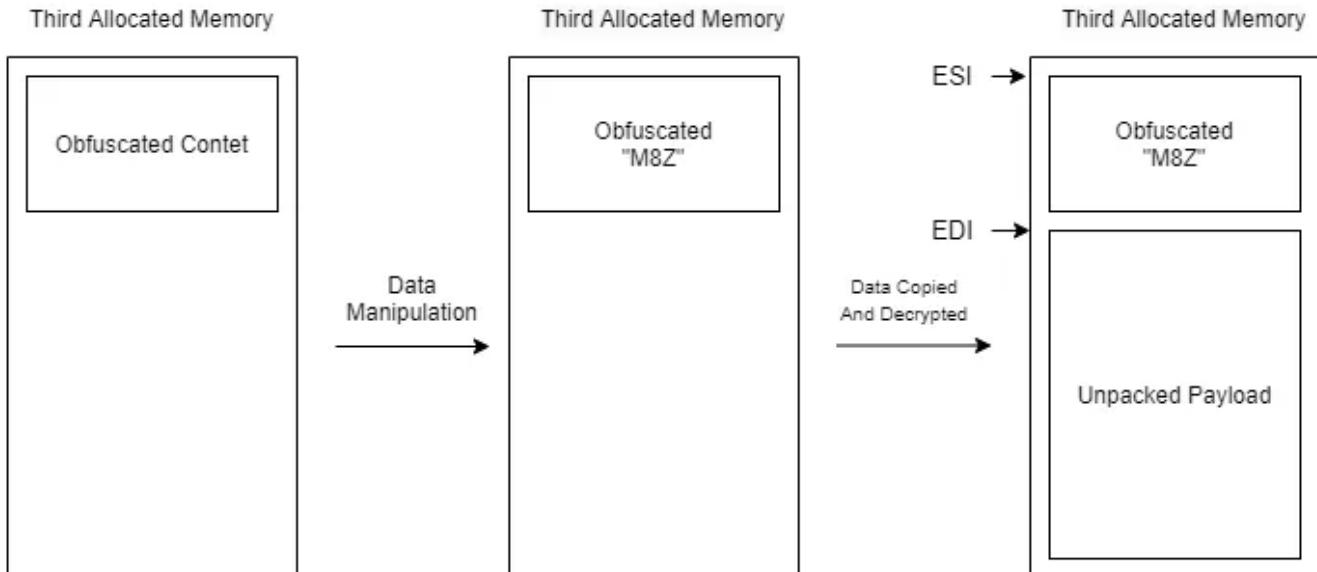
Eventually, we understand the purpose of this loop -writing a clean portable executable in this memory part.

Open in app ↗



IcedID: Clean PE File

The final stages of the unpacking mechanism are portrayed in the following diagram:



IcedID: Unpacking Diagram

After the loop ends, we'll dump the file and inspect it with other tools. This file, which is actually a module, is the second stage of IcedID, which is a downloader for the final payload.

property	value
md5	E98AC0534F01FA34C3A107727009D3D3
sha1	82F17A51EFF45A1AEC75103BE2187DCFD4B651D
sha256	BE5CDDA4888B59E9EDF83D7186EED4EA7428EBB570283E8E0C5C0BEA789BB4EB
md5-without-overlay	067D2AE1952CBF36049A618C75F2E5FC
sha1-without-overlay	9F7D1DE6FA98A314A1EFCF62C5B70A774C1A241C
sha256-without-ove...	7B2A10C43AF496AE350C01BFD70E0C05249DFAA921B88976B8BC6D62727F479C
first-bytes-hex	4D 5A 90 00 03 00 00 04 00 00 FF FF 00 00 B8 00 00 00 00 00 00 40 00 00...
first-bytes-text	M Z@.....
file-size	26182 (bytes)
size-without-overlay	10752 (bytes)
entropy	3.118
imphash	9A31CA7834D07273DABB36E82CAF0A31
signature	n/a
entry-point	83 7C 24 08 01 75 1B 8B 44 24 04 33 C9 51 51 51 68 DD 13 00 10 51 51 A3 00 30 ...
file-version	n/a
description	n/a
file-type	dynamic-link-library
cpu	32-bit
subsystem	GUI

IcedID: Unpacked PE In Pestudio

Conclusion

In this writeup, we observed three first-stage malware unpacking mechanisms. And although each one has its own way to unpack itself, we saw several characteristics they all shared.

Similarly, whether its unpacking or decrypting, the following features will most likely occur:

- Traditionally, packers tend to start with reading an obfuscated data embedded in the PE, and writing it to a newly allocated memory.
- Writing to, or reading from newly allocated memory will most likely happen inside a loop. So loops can be a good starting point when we search for decryption routines.
- Most of the time, data will be copied or modified byte by byte. Therefore, it is important to pay attention to moves of one byte (*mov byte ptr*)
- Some opcodes such as *xor, rol, ror, shl, shr* are likely to be found in the decryption loop.

[Malware](#)[Malware Analysis](#)[Packers](#)[Reverse Engineering](#)[Debugging](#)