

# Funtastic Packers And Where To Find Them

By Eli Salem

Published: 2021-01-18 · Archived: 2026-04-05 20:37:45 UTC



11 min read

Jan 18, 2021



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.

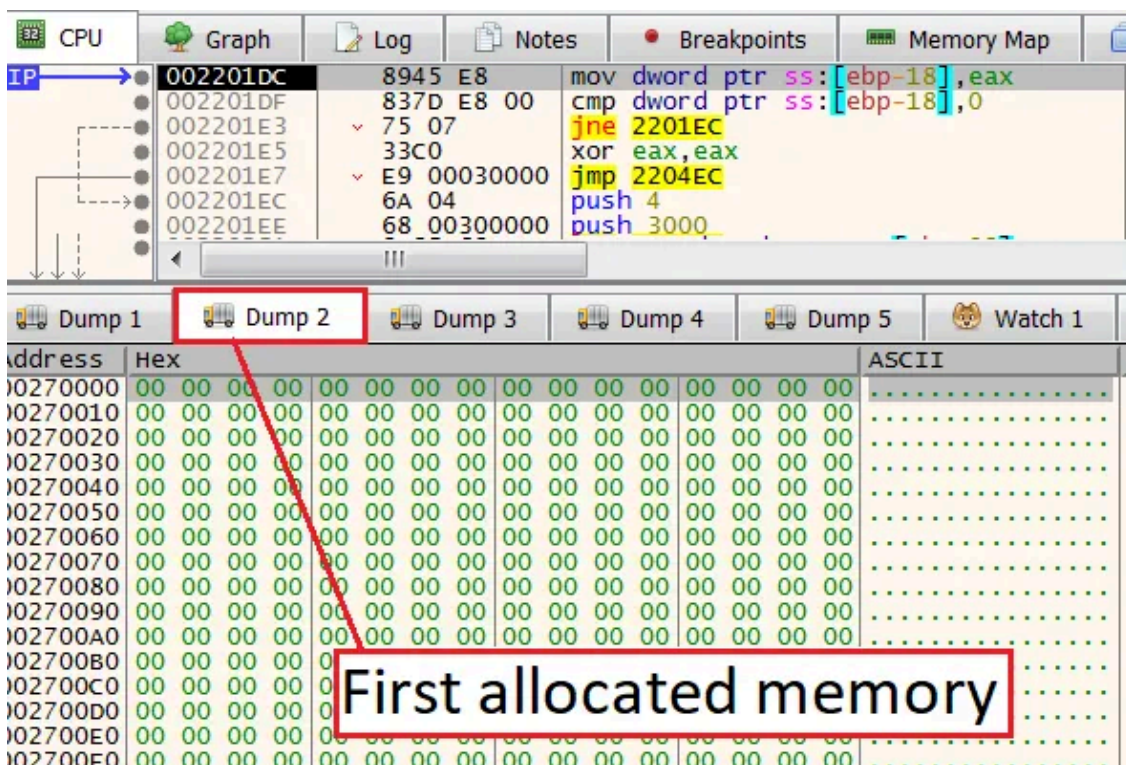
## Get2Downloader

[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.

Press enter or click to view image in full size



Get2: First Allocated Memory

The screenshot shows a debugger interface with the following components:

- Assembly View:** Shows instructions at addresses 002201FF to 0022020F. The instruction at 00220206 is `jne 22020F`, and the instruction at 0022020A is `jmp 2204EC`.
- Dump Tabs:** Includes 'Dump 1', 'Dump 2', 'Dump 3' (highlighted with a red box), 'Dump 4', 'Dump 5', and 'Watch 1'.
- Memory Dump Table:** A table with columns for Address, Hex, and ASCII. The Hex column shows a sequence of 00 values. A red box containing the text 'Second allocated memory' is overlaid on the dump data.

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 a debugger window with the following assembly code:

```

00220219 C785 68FFFFFF mov dword ptr ss:[ebp-98],0
00220223 EB 1E jmp 220243
00220225 8B8D 64FFFFFF mov ecx, dword ptr ss:[ebp-9C]
0022022B 83C1 01 add ecx,1
0022022E 898D 64FFFFFF mov dword ptr ss:[ebp-9C],ecx
00220234 8B95 68FFFFFF mov edx, dword ptr ss:[ebp-98]
0022023A 83C2 01 add edx,1
0022023D 8995 68FFFFFF mov dword ptr ss:[ebp-98],edx
00220243 8B45 08 mov eax, dword ptr ss:[ebp+8]
00220246 8B8D 64FFFFFF mov ecx, dword ptr ss:[ebp-9C]
0022024C 3B48 0C cmp ecx, dword ptr ds:[eax+C]
0022024F 73 3E jae 22028F
00220251 8B85 68FFFFFF mov eax, dword ptr ss:[ebp-98]
00220257 33D2 xor edx,edx
00220259 B9 02000000 mov ecx,2
0022025E F7F1 div ecx
00220260 85D2 test edx,edx
00220262 75 0F jne 220273
00220264 8B95 64FFFFFF mov edx, dword ptr ss:[ebp-9C]
0022026A 83C2 02 add edx,2
0022026D 8995 64FFFFFF mov dword ptr ss:[ebp-9C],edx
00220273 8B45 08 mov eax, dword ptr ss:[ebp+8]
00220276 8B48 08 mov ecx, dword ptr ds:[eax+8]
00220279 8B55 E8 mov edx, dword ptr ss:[ebp-18]
0022027C 0395 68FFFFFF add edx, dword ptr ss:[ebp-98]
00220282 8B85 64FFFFFF mov eax, dword ptr ss:[ebp-9C]
00220288 8A0C01 mov cl, byte ptr ds:[ecx+eax]
0022028B 880A mov byte ptr ds:[edx],cl
0022028D EB 96 jmp 220225
0022028F C785 60FFFFFF mov dword ptr ss:[ebp-A0],0
    
```

The memory dump below shows the following data:

Address	Hex	ASCII
00270000	53 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	S.....
00270010	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00270020	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 memory dump shows the following obfuscated content:

Address	Hex	ASCII
0270000	53 39 8E 51 E2 FD AE 08 46 8C 7F C8 4E A1 A4 91	59.Qây®.F..ÈÑjµ.
0270010	D6 B9 E4 99 4A 8C 68 B9 37 6C 2A 04 FE DD 23 49	Ó'ã.J.h'7l*.pY#I
0270020	FB 1F E4 57 D3 25 C9 4D 41 AA 8F 0A 11 BF 7F 8F	û.äw0%EMAª...¿..
0270030	20 B0 5E FF E7 7B 71 6F 22 C7 7F 65 80 6B 6E A2	°^ÿç{qo"C.e.kñç
0270040	21 81 1B DF 96 69 BD ED 36 EA 78 4F 80 7B 5B C9	!..B.i½i6éxo.{{É
0270050	77 DC 0A A2 B2 B5 AA C8 4D AD 94 BC 18 86 D3 58	wÜ.ç²µªÈM..¼..ÖX
0270060	11 43 50 EB 23 00 69 CB 66 FB 50 00 95 69 A6 72	.CPë#.iÉfÛP..i r
0270070	07 7F 36 03 5B 14 1B CA 8B 85 24 0E D7 EC 57 89	..6.[..É..\$.xiw.
0270080	C8 D2 16 11 24 41 F0 79 63 C0 4C C6 16 98 A4 09	ÈÖ..\$AðycALÆ..µ.
0270090	3D DF 35 43 AA 35 A8 E6 0A BC 1A B8 24 03 93 09	=B5Cª5 æ.¼. \$....
02700A0	43 1A BB AE D1 CA 50 05 D7 C8 1C 09 C3 9D 28 49	C.»®ÑÉP.xÈ..Á.(I
02700B0	34 92 67 DE 8D 9C 8F 46 F7 8D 3A 29 26 03 2C 9B	4.gp...F÷.:.)&...
02700C0	7A D2 8E 89 D4 BB 98 B0 14 DE 2A 8B 11 83 99 89	zò..ò»..°P*.....
02700D0	55 4A 0B 8A DD FF 28 A2 F5 1E 49 C1 C2 99 C9 92	UJ..Ýÿ(çö.IÁÁ.É.
02700E0	27 63 1C 3F 92 8E 1A 9B 57 EC 88 07 F0 9E 8D ED	'c.?....wì..ð..í
02700F0	D7 5F A6 9B 91 5C 0D 0E 4A B5 51 6F D1 A9 1B 85	x_ ...\.JµQoÑè..

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.

The screenshot shows a debugger window with the following assembly code:

```

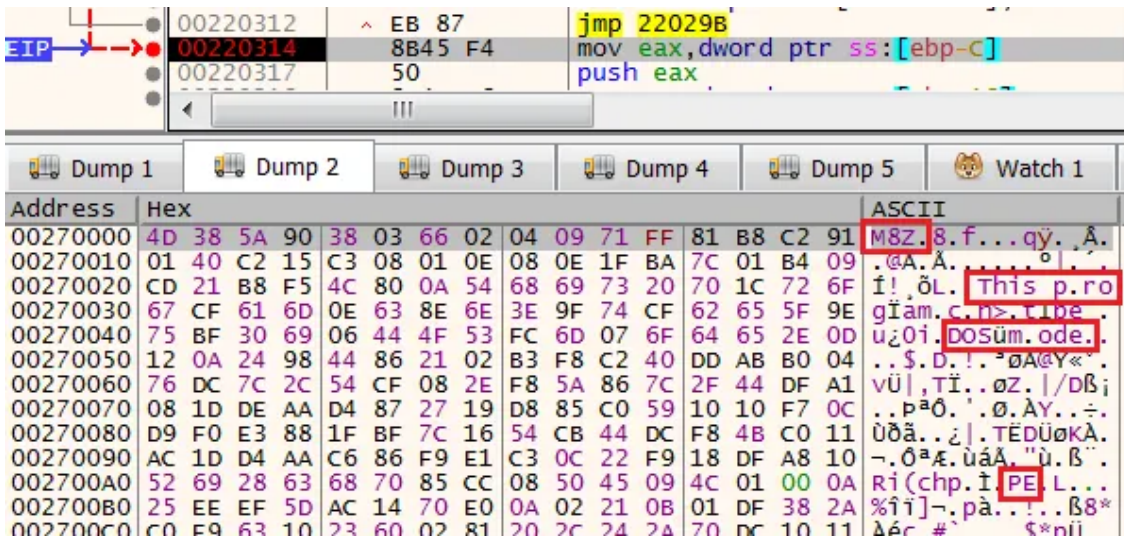
00220299 EB 0F jmp 2202AA
0022029B 8B95 60FFFFFF mov edx,dword ptr ss:[ebp-A0]
002202A1 83C2 01 add edx,1
002202A4 8995 60FFFFFF mov dword ptr ss:[ebp-A0],edx
002202AA 8B45 08 mov eax,dword ptr ss:[ebp+8]
002202AD 8B48 0C mov ecx,dword ptr ds:[eax+C]
002202B0 C1E9 02 shr ecx,2
002202B3 398D 60FFFFFF cmp dword ptr ss:[ebp-A0],ecx
002202B9 73 59 jae 220314
002202BB 8B95 60FFFFFF mov edx,dword ptr ss:[ebp-A0]
002202C1 8B45 E8 mov eax,dword ptr ss:[ebp-18]
002202C4 8B0C90 mov ecx,dword ptr ds:[eax+edx*4]
002202C7 898D 5CFFFFFF mov dword ptr ss:[ebp-A4],ecx
002202CD 8B55 08 mov edx,dword ptr ss:[ebp+8]
002202D0 8B85 5CFFFFFF mov eax,dword ptr ss:[ebp-A4]
002202D6 3342 10 xor eax,dword ptr ds:[edx+10]
002202D9 8985 5CFFFFFF mov dword ptr ss:[ebp-A4],eax
002202DF 8B8D 5CFFFFFF mov ecx,dword ptr ss:[ebp-A4]
002202E5 C1C1 04 rol ecx,4
002202E8 898D 5CFFFFFF mov dword ptr ss:[ebp-A4],ecx
002202EE 8B95 5CFFFFFF mov edx,dword ptr ss:[ebp-A4]
002202F4 81C2 78777777 add edx,77777778
002202FA 8995 5CFFFFFF mov dword ptr ss:[ebp-A4],edx
00220300 8B85 60FFFFFF mov eax,dword ptr ss:[ebp-A0]
00220306 8B4D E8 mov ecx,dword ptr ss:[ebp-18]
00220309 8B95 5CFFFFFF mov edx,dword ptr ss:[ebp-A4]
0022030F 891481 mov dword ptr ds:[ecx+eax*4],edx
00220312 EB 87 jmp 22029B
00220314 8B45 F4 mov eax,dword ptr ss:[ebp-C]
00220317 50 push eax
    
```

The memory dump below shows the following data:

Address	Hex	ASCII
00270000	4D 38 5A 90 E2 FD AE 08 46 8C 7F C8 4E A1 A4 91	M8Z. ky°. F. ËNj#.
00270010	D6 B9 E4 99 4A 8C 68 B9 37 6C 2A 04 FE DD 23 49	Ü a. J. h'7l*. pY#I
00270020	FB 1F E4 57 D3 25 C9 4D 41 AA 8F 0A 11 BF 7F 8F	û. äw0%ÉMAª...¿..
00270030	20 B0 5E FF E7 7B 71 6F 22 C7 7F 65 80 6B 6E A2	° ^ÿç{qo"ç. e.knt
00270040	21 81 1B DF 96 69 BD ED 36 EA 78 4F 80 7B 5B C9	!..B. i½i6éxo. {[É
00270050	77 DC 0A A2 B2 B5 AA C8 4D AD 94 BC 18 86 D3 58	wÜ. Ç²µªÉM..¼..ÖX
00270060	11 43 50 EB 23 00 69 CB 66 FB 50 00 95 69 A6 72	.CPë#. iÉfÛP..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	ÈÖ..\$AðycALÆ..ª.
00270090	3D DF 35 43 AA 35 A8 E6 0A BC 1A B8 24 03 93 09	=B5Cª5 æ.¼. \$....
002700A0	43 1A BB AE D1 CA 50 05 D7 C8 1C 09 C3 9D 28 49	C. »ªÑÉP. xÈ..Ä. (I

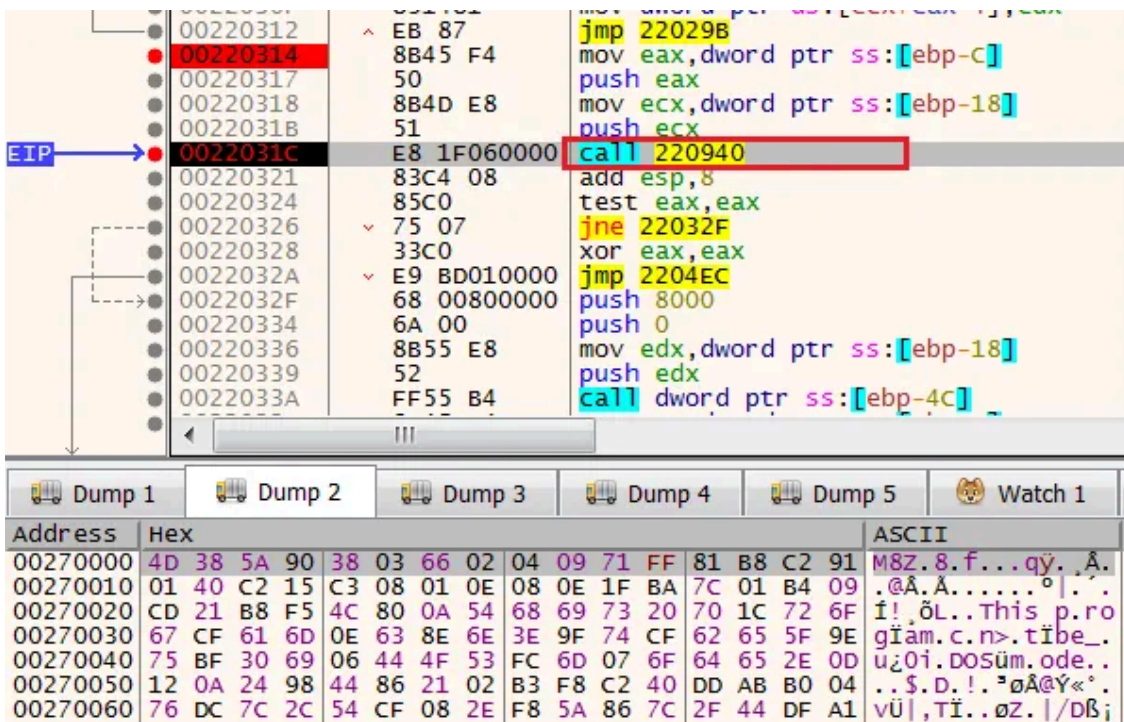
Get2: M8Z string written

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.



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.



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 displays a debugger's assembly and memory dump windows. The assembly window shows the following instructions:

```

0022098A 837D E8 00 cmp dword ptr ss:[ebp-18],0
0022098E 0F85 4B020000 jne 220BDF
00220994 8D4D D8 lea ecx,dword ptr ss:[ebp-28]
00220997 51 push ecx
00220998 E8 F3FEFFFF call 220890
0022099D 83C4 04 add esp,4
002209A0 85C0 test eax,eax
002209A2 0F84 0F020000 je 220BB7
002209A8 8D55 D8 lea edx,dword ptr ss:[ebp-28]
002209AB 52 push edx
002209AC E8 DFFEFFFF call 220890
002209B1 83C4 04 add esp,4
002209B4 85C0 test eax,eax
002209B6 0F84 E8000000 je 220AA4
002209BC 8D45 D8 lea eax,dword ptr ss:[ebp-28]
002209BF 50 push eax
002209C0 E8 CBFEEFFF call 220890
002209C5 83C4 04 add esp,4
002209C8 85C0 test eax,eax
002209CA 74 6C je 220A38
002209CC C745 EC 0000 mov dword ptr ss:[ebp-14],0
002209D3 C745 F8 0400 mov dword ptr ss:[ebp-8],4
002209DA EB 09 jmp 2209E5
002209DC 8B4D F8 mov ecx,dword ptr ss:[ebp-8]
002209DF 83E9 01 sub ecx,1
002209E2 894D F8 mov dword ptr ss:[ebp-8],ecx
002209E5 837D F8 00 cmp dword ptr ss:[ebp-8],0
002209E9 74 17 je 220A02
002209EB 8D55 D8 lea edx,dword ptr ss:[ebp-28]
002209EE 52 push edx
002209EF E8 9CFEFFFF call 220890
002209F4 83C4 04 add esp,4
002209F7 8B4D EC mov ecx,dword ptr ss:[ebp-14]
    
```

The memory dump window shows the following data:

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.

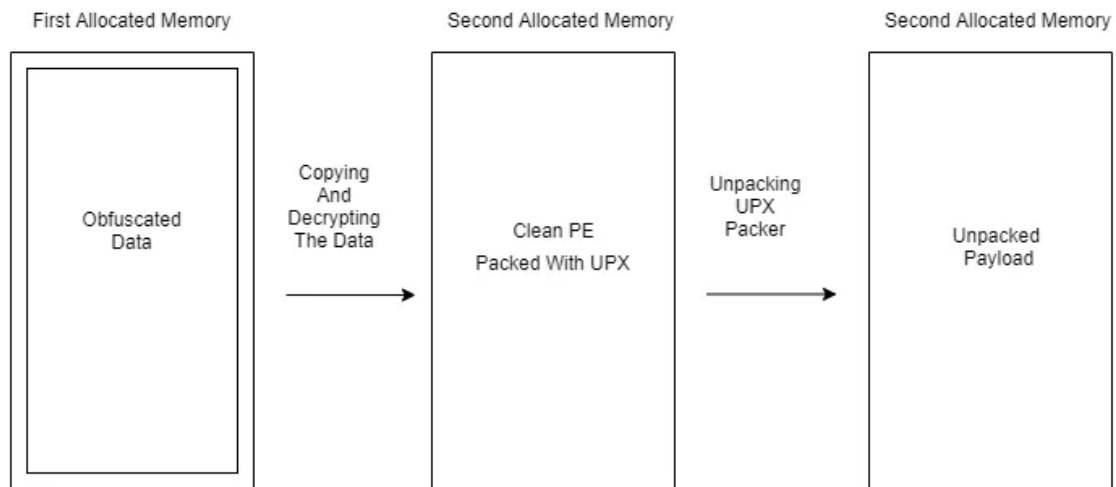
The screenshot displays a debugger interface with two main windows. The top window shows assembly code with the instruction pointer (EIP) at 002208DF. The code includes instructions for moving data from the stack, subtracting, adjusting the stack pointer, popping the stack pointer, returning, and calling the Windows API function `int3`. The bottom window shows a memory dump starting at address 002B0000. The dump is organized into columns for hex values and ASCII characters. Two specific memory locations are highlighted with red boxes: one containing the text 'UPX0...' and another containing 'UPX1...'. The ASCII column shows various characters, including letters, symbols, and control characters, indicating a packed executable format.

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.

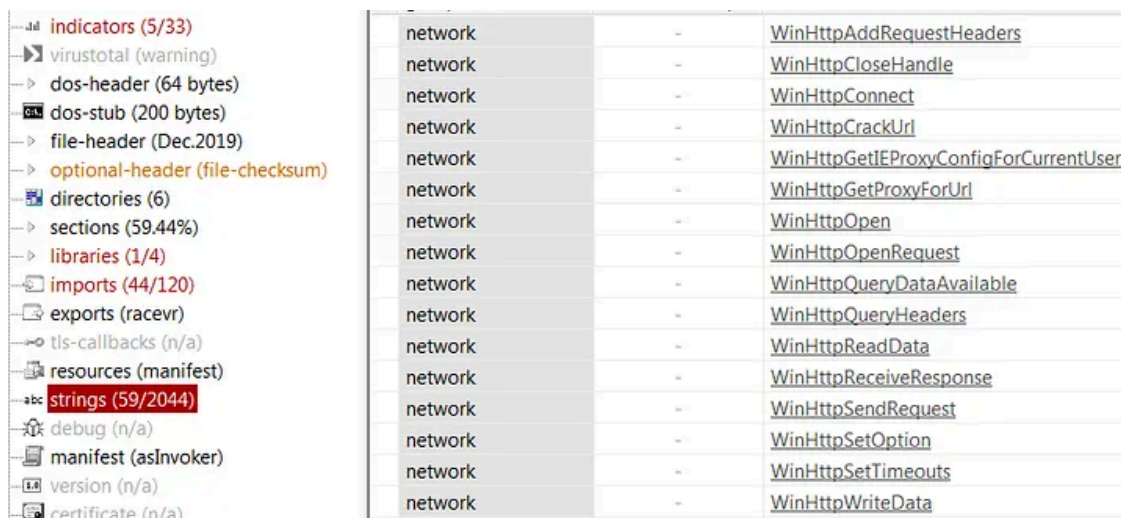
**Note:** In some recent versions of Get2 malware, the UPX part is missing.

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.

Press enter or click to view image in full size



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.

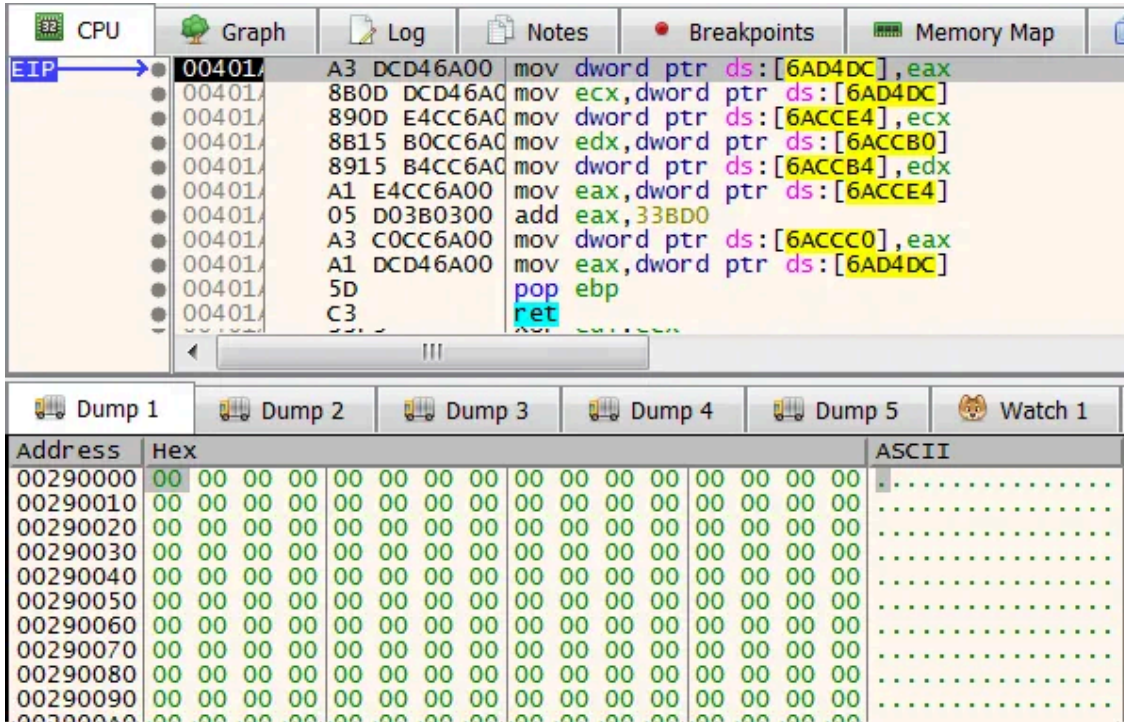
The way to unpack the first dropper of Qbot is relatively simple and already been documented. Setting a breakpoint on [VirtualProtect](#) and inspecting several *mov* instructions after it, will do the trick. Otherwise, we can use automated tools such as [PE-Sieve](#). However, where is the fun in that?

## Get Eli Salem's stories in your inbox

Join Medium for free to get updates from this writer.

Remember me for faster sign in

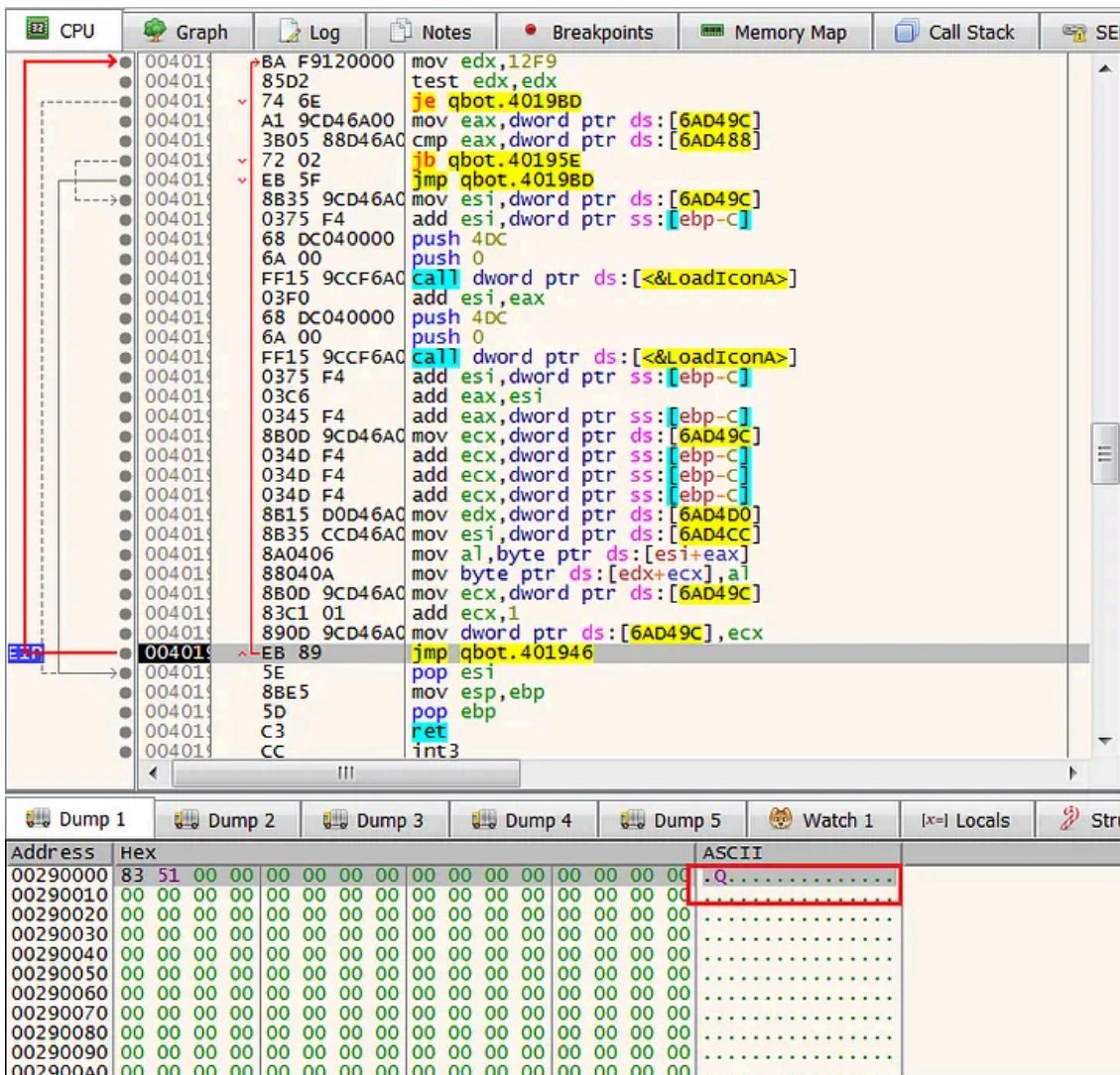
To start, we'll set a breakpoint on *VirtualAllocEx* and *VirtualAlloc*, then, click Run. We'll hit on *VirtualAllocEx*, and to assure nothing slips away, put a write hardware breakpoint on the newly allocated memory. Then, hit run.



Qbot: *VirtualAllocEx* allocated memory

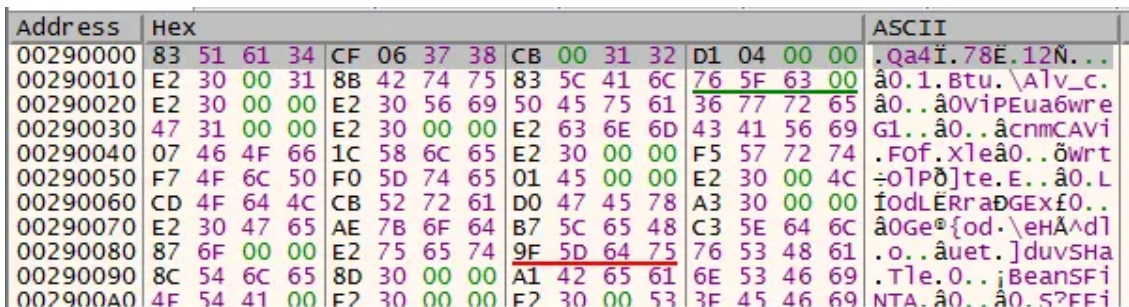
We reach our breakpoint in a function that writes obfuscated data, byte by byte, into this allocated memory. Similar to the Get2 malware or basically any other malware that has an unpacking mechanism, the data is written inside a loop.

Press enter or click to view image in full size



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.



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.

Press enter or click to view image in full size

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	.....
0004A7E0	00	3E	03	00	83	51	61	34	CF	06	37	38	CB	00	31	32	.>..fQa4İ.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	õWrt+OlPõ]te.E..
0004A840	E2	30	00	4C	CD	4F	64	4C	CB	52	72	61	D0	47	45	78	â0.LíOdLÈRraDGEx
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..âuetÿ]du
0004A870	76	53	48	61	8C	54	6C	65	8D	30	00	00	A1	42	65	61	vSHaETle.0..jBea
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.....

Qbot: Static analysis — obfuscated content

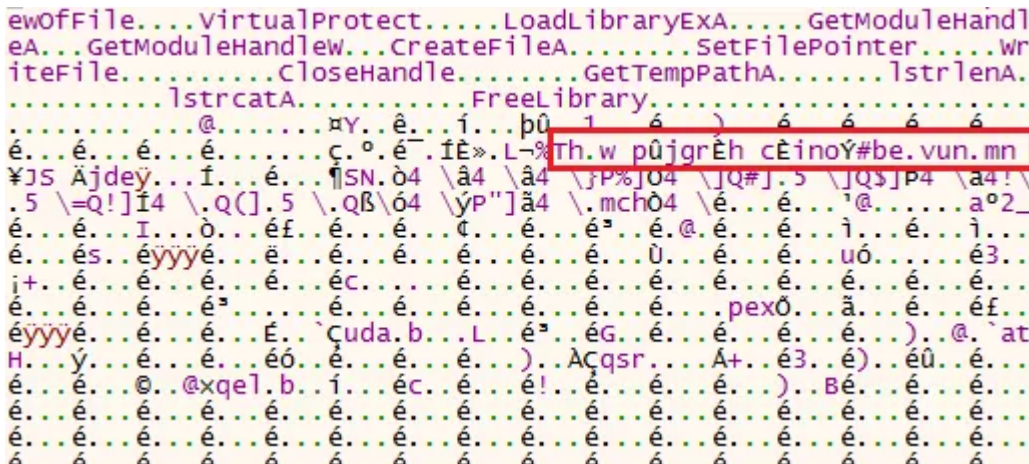
Interestingly, this memory section is being manipulated again and again by several loops. Then, it being overwritten until the upper part is filled with the unknown string “aaa45678901234” followed by several names of API calls.

Press enter or click to view image in full size

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 65	.....virtualFre
00290030	65 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 41	....GetTempPathA
002900F0	00 00 00 00 00 00 00 6C 73 74 72 6C 65 6E 41 00	.....lstrlenA.
00290100	00 00 00 00 00 00 00 00 00 00 6C 73 74 72 63 61	.....lstrca

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 has fragments of strings that can indicate a potential encrypted PE file.

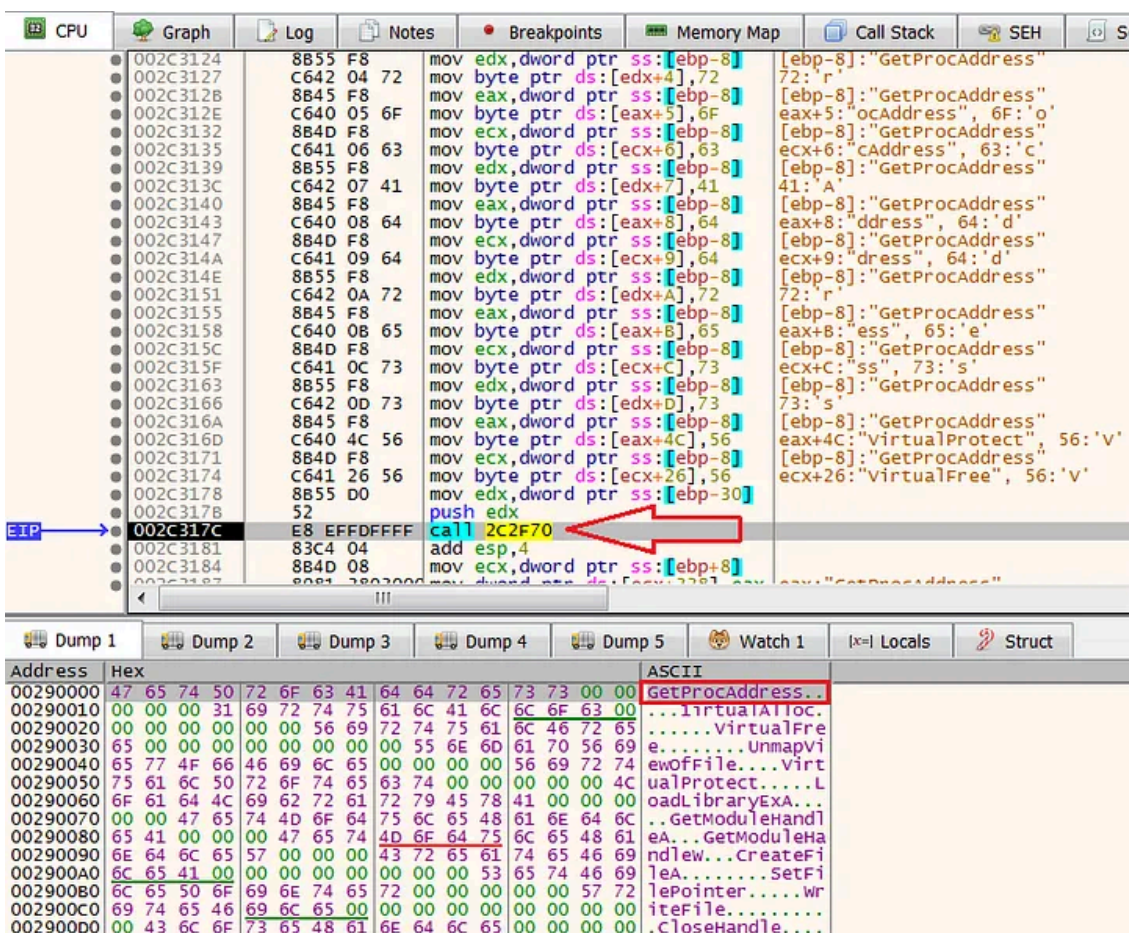


Qbot: Signs Of Obfuscated PE

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.

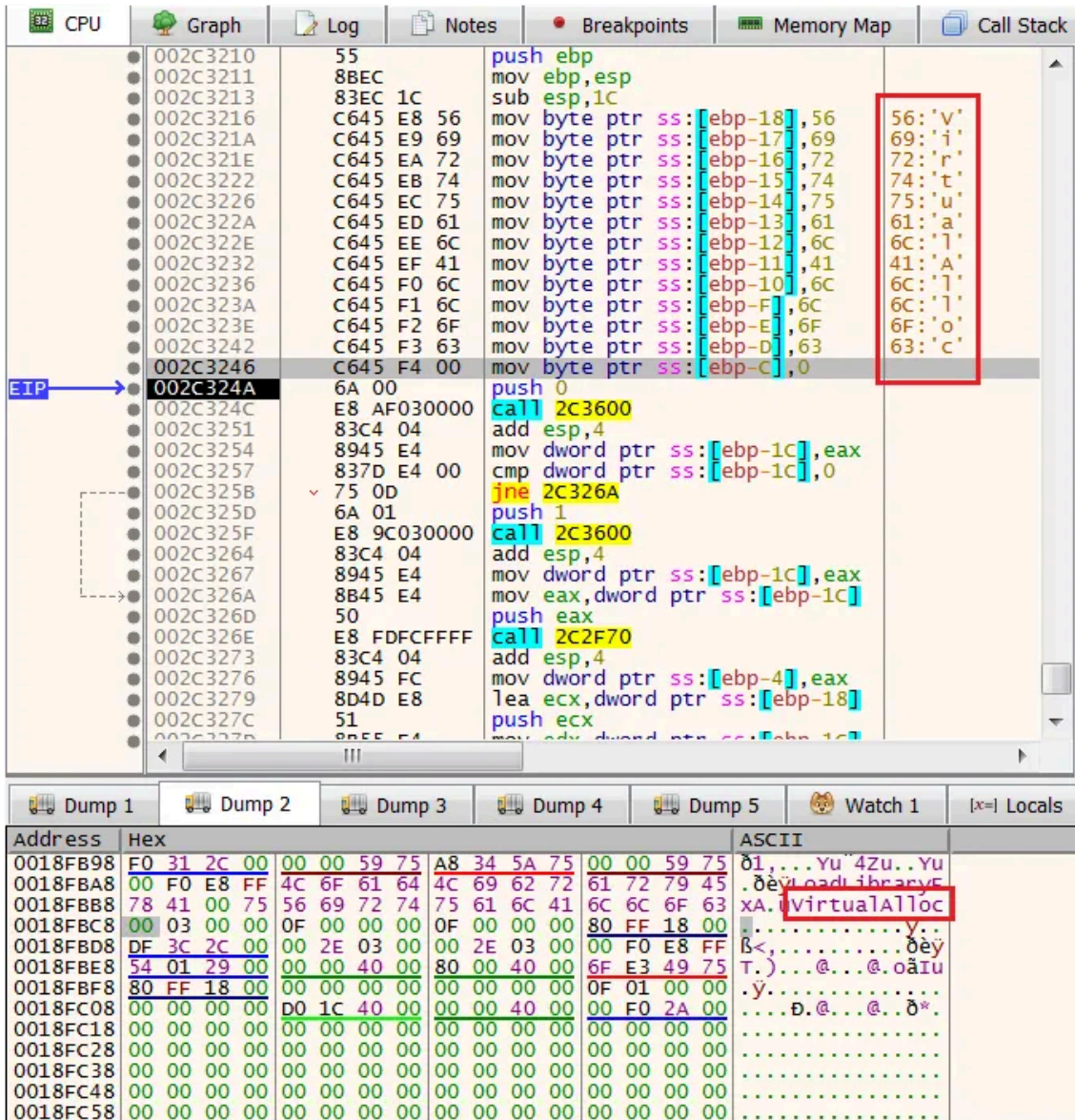
Press enter or click to view image in full size



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*.

Press enter or click to view image in full size



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 8D4D 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
002C329A C3 ret
    
```

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.

**Assembly Code:**

```

002C33D0 55          push ebp
002C33D1 8BEC       mov ebp, esp
002C33D3 83EC 0C    sub esp, C
002C33D6 8B45 08    mov eax, dword ptr ss:[ebp+8]
002C33D9 8945 FC    mov dword ptr ss:[ebp-4], eax
002C33DC 8B4D 0C    mov ecx, dword ptr ss:[ebp+C]
002C33DF 894D F4    mov dword ptr ss:[ebp-C], ecx
002C33E2 C745 F8 0000 mov dword ptr ss:[ebp-8], 0
002C33E9 EB 09     jmp 2C33F4
002C33EB 8B55 F8    mov edx, dword ptr ss:[ebp-8]
002C33EE 83C2 01    add edx, 1
002C33F1 8955 F8    mov dword ptr ss:[ebp-8], edx
002C33F4 8B45 F8    mov eax, dword ptr ss:[ebp-8]
002C33F7 3B45 10    cmp eax, dword ptr ss:[ebp+10]
002C33FA 73 12     jae 2C340E
002C33FC 8B4D FC    mov ecx, dword ptr ss:[ebp-4]
002C33FF 034D F8    add ecx, dword ptr ss:[ebp-8]
002C3402 8B55 F4    mov edx, dword ptr ss:[ebp-C]
002C3405 0355 F8    add edx, dword ptr ss:[ebp-8]
002C3408 8A02     mov al, byte ptr ds:[edx]
002C340A 8801     mov byte ptr ds:[ecx], al
002C340C EB DD     jmp 2C33EB
002C340E 8BE5     mov esp, ebp
002C3410 5D      pop ebp
002C3411 C3      ret
002C3412 CC      int3
002C3413 CC      int3
002C3414 CC      int3
002C3415 CC      int3
002C3416 CC      int3
002C3417 CC      int3
002C3418 CC      int3
002C3419 CC      int3
002C341A CC      int3
    
```

**Memory Dump:**

Address	Hex	ASCII
002D0000	A4 59 90 00 EA 03 00 00 00 00 00 00 00 00 00	AY...ë.....
002D0010	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	.....
002D0030	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	.....
002D0050	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	.....
002D0070	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	.....
002D0090	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	.....
002D00B0	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	.....

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	ASCII								
002D0000	A4 59 90 00	EA 03 00 00	ED 03 00 00	FE FB 00 00	pY..è...í...pú..					
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...é...					
002D0040	E7 1A BA 0E	E9 AF 09 CD	C8 BB 01 4C	AC 25 54 68	ç.°.é-.îÊ».L-%Th					
002D0050	00 77 20 70	FB 6A 67 72	C8 68 20 63	C8 69 6E 6F	.w pùjgrÈh cÈino					
002D0060	DD 23 62 65	09 76 75 6E	09 6D 6E 20	A5 4A 53 20	Y#be.vun.mn ¥JS					
002D0070	C4 6A 64 65	FF 08 0D 0A	CD 03 00 00	E9 03 00 00	Åjdey...í...é...					
002D0080	B6 53 4E 0F	F2 34 20 5C	E2 34 20 5C	E2 34 20 5C	[SN.ò4 \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!]í4 \.Q(.5 \					
002D00C0	1D 51 DF 5C	F3 34 20 5C	FD 50 22 5D	E3 34 20 5C	.qß\ó4 \ÿP" ]â4 \					

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.

Press enter or click to view image in full size

The screenshot displays a debugger interface with two main windows. The top window shows assembly code with the following instructions:

```

002C3A94 C745 FC 0000 mov dword ptr ss:[ebp-4],0
002C3A9B EB 09 jmp 2C3AA6
002C3A9D 8B45 FC mov eax,dword ptr ss:[ebp-4]
002C3AA0 83C0 04 add eax,4
002C3AA3 8945 FC mov dword ptr ss:[ebp-4],eax
002C3AA6 8B4D FC mov ecx,dword ptr ss:[ebp-4]
002C3AA9 3B4D 0C cmp ecx,dword ptr ss:[ebp+C]
002C3AAC 73 2E jae 2C3ADC
002C3AAE 8B55 08 mov edx,dword ptr ss:[ebp+8]
002C3AB1 0355 FC add edx,dword ptr ss:[ebp-4]
002C3AB4 8B02 mov eax,dword ptr ds:[edx]
002C3AB6 0345 FC add eax,dword ptr ss:[ebp-4]
002C3AB9 8B4D 08 mov ecx,dword ptr ss:[ebp+8]
002C3ABC 034D FC add ecx,dword ptr ss:[ebp-4]
002C3ABF 8901 mov dword ptr ds:[ecx],eax
002C3AC1 8B55 FC mov edx,dword ptr ss:[ebp-4]
002C3AC4 81C2 E90300 add edx,3E9
002C3ACA 8B45 08 mov eax,dword ptr ss:[ebp+8]
002C3ACD 0345 FC add eax,dword ptr ss:[ebp-4]
002C3AD0 3310 xor edx,dword ptr ds:[eax]
002C3AD2 8B4D 08 mov ecx,dword ptr ss:[ebp+8]
002C3AD5 034D FC add ecx,dword ptr ss:[ebp-4]
002C3AD8 8911 mov dword ptr ds:[ecx],edx
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
002C3AE4 8B45 08 mov eax,dword ptr ss:[ebp+8]
002C3AE7 8945 FC mov dword ptr ss:[ebp-4],eax
002C3AEA EB 09 jmp 2C3AF5
    
```

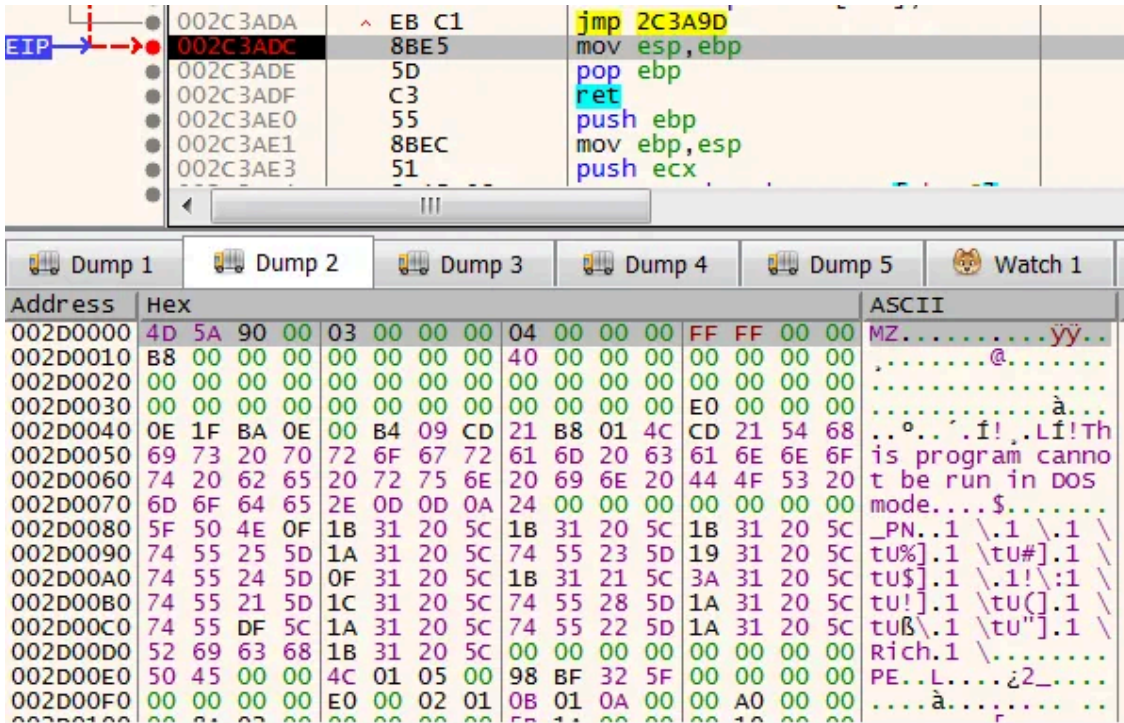
The bottom window shows a memory dump with the following data:

Address	Hex	ASCII
002D0000	4D 5A 90 00 EA 03 00 00 ED 03 00 00 FE FB 00 00	MZ .é...í...bû..
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 E9 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.°.é.îÊ».L-%Th
002D0050	00 77 20 70 FB 6A 67 72 C8 68 20 63 C8 69 6E 6F	.w pûjgrêh cÊino
002D0060	DD 23 62 65 09 76 75 6E 09 6D 6E 20 A5 4A 53 20	Y#be.vun.mn ¥J5
002D0070	C4 6A 64 65 FF 08 0D 0A CD 03 00 00 E9 03 00 00	Äjdeý...í...é...
002D0080	B6 53 4E 0F F2 34 20 5C E2 34 20 5C E2 34 20 5C	¶5N.04 \â4 \â4 \
002D0090	7D 50 25 5D D3 34 20 5C 5D 51 23 5D 00 35 20 5C	}P%]04 \]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!]î4 \.Q(.5 \
002D00C0	1D 51 DF 5C F3 34 20 5C FD 50 22 5D E3 34 20 5C	.Qß\04 \ÿP"]â4 \

Qbot: Second Allocated Memory Deobfuscated

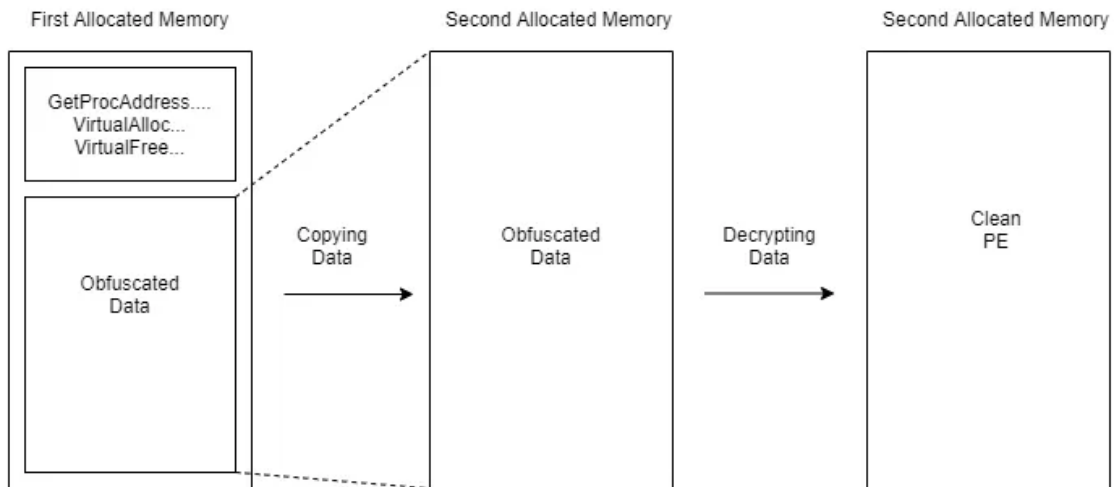
This loop contains several characteristics that already observed in the Get2 malware unpacking mechanism. The use of *xor* and *add* opcodes.

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.



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.

Press enter or click to view image in full size

	entrop...	languag...	first-bytes-hex	first-bytes-text
D3AD502F857B6...	2.748	neutral	00 00 01 00 01 00 20 40 00 00...	..... @ .....
F6C5998970A01...	2.944	neutral	00 00 01 00 01 00 10 20 00 00...	..... ( ..
92BC614EBE97C...	1.891	neutral	00 00 01 00 01 00 30 60 00 00...	..... 0 ` .....
C8AADAC8BFAB...	1.611	neutral	00 00 01 00 01 00 28 50 00 00...	..... ( P .....
171146D2255AA...	1.333	neutral	00 00 01 00 01 00 14 28 00 00...	..... ( .....
0B20D0FFF006AB...	1.440	neutral	00 00 01 00 01 00 10 20 00 00...	..... .. h ..
A73182CF0C090...	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.

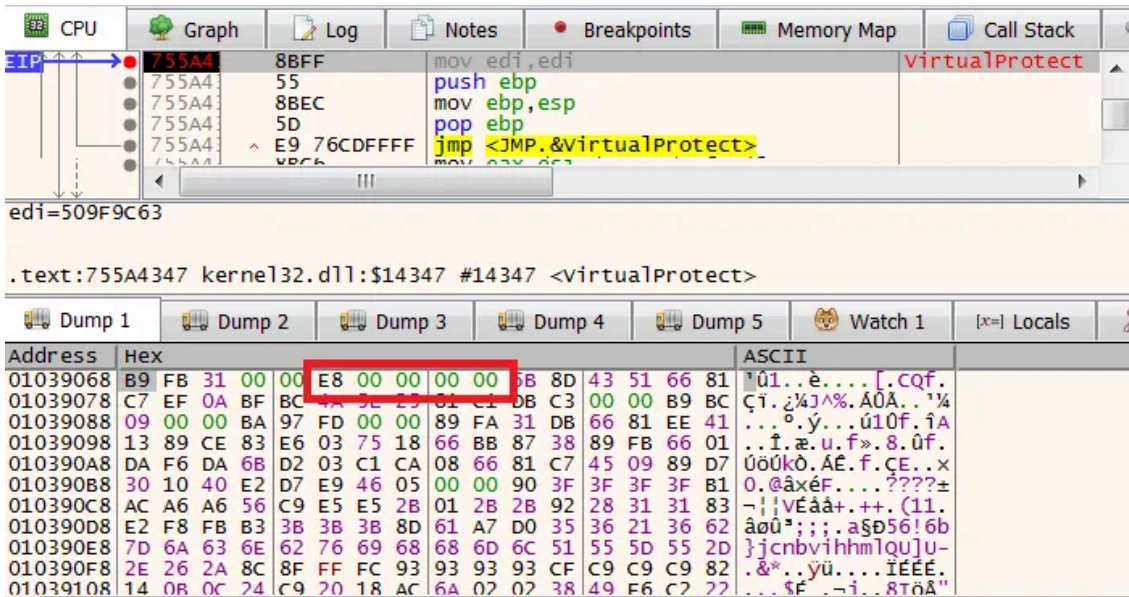
Unpacking the first dropper of IcedID is pretty simple, similar to Qbot, we only need to set breakpoints on *VirtualAlloc* and *VirtualProtect*. In the unpacking mechanism, the second time *VirtualProtect* will be called it will indicate the unpacked IcedID. As said, we’ll focus on how the IcedID unpacking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

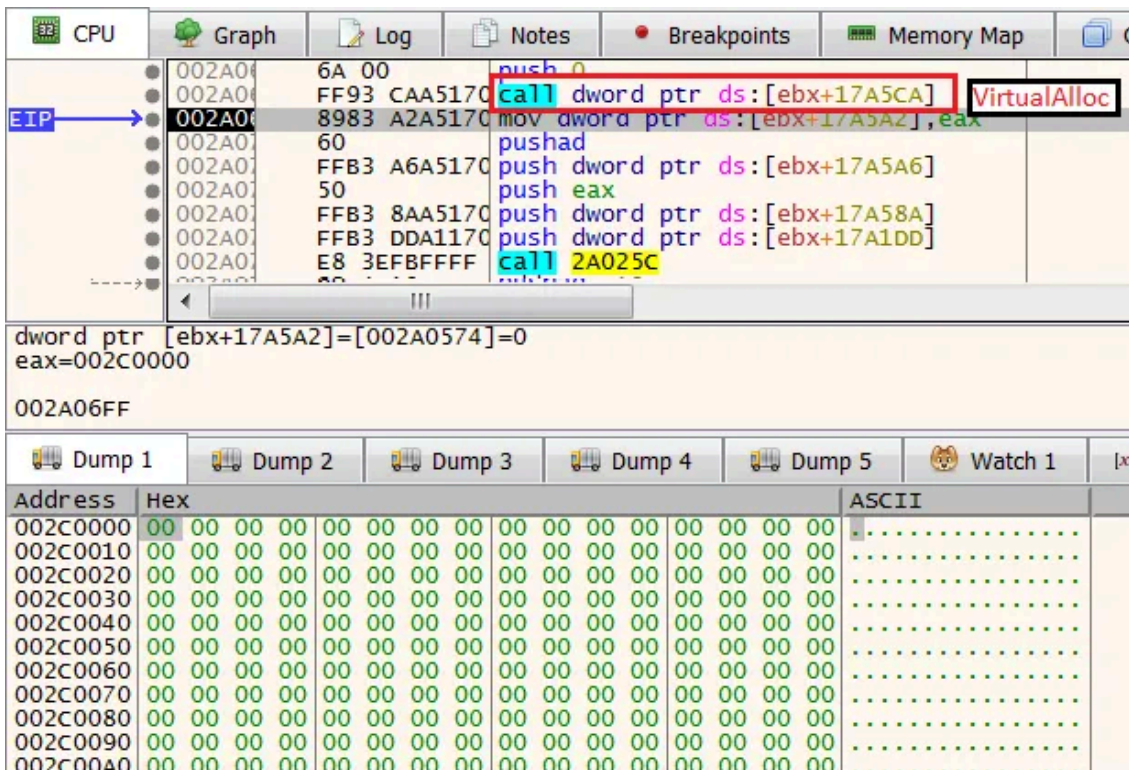
In this writeup, I won’t cover the entire shellcode investigation, only the unpacking mechanism. For those of you who want to explore it, you can click on the shellcode memory address, press “follow in disassembler”, set a breakpoint on the relative call, and hit Run.

Press enter or click to view image in full size



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.



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.

Press enter or click to view image in full size

The screenshot shows a debugger window with the following assembly code and registers:

```
002A0131C0 xor eax, eax
002A0131F6 push edi
002A0166:833B 00 cmp word ptr ds:[ebx], 0
002A0175 07 jne 2A0278
002A0166:837B 02 0 cmp word ptr ds:[ebx+2], 0
002A0174 54 je 2A02CC
002A010FB70B movzx ecx, word ptr ds:[ebx]
002A010FB76B 02 movzx ebp, word ptr ds:[ebx+2]
002A010FB7D1 movzx edx, cx
002A0101F2 add edx, esi
002A0166:83F9 FF cmp cx, FFFF
002A01896C24 28 mov dword ptr ss:[esp+28], ebp
002A0175 0A jne 2A0298
002A0190 nop
002A018B53 04 mov edx, dword ptr ds:[ebx+4]
002A0183C3 04 add ebx, 4
002A0190 nop
002A0101F2 add edx, esi
002A018B4C24 1C mov ecx, dword ptr ss:[esp+1C], ecx
002A0101D1 add ecx, edx
002A01D9D0 fnop
002A01894C24 14 mov dword ptr ss:[esp+14], ecx
002A018B4C24 24 mov ecx, dword ptr ss:[esp+24]
002A0101C1 add ecx, eax
002A0183C3 04 add ebx, 4
002A01894C24 10 mov dword ptr ss:[esp+10], ecx
002A0156 push esi
002A0157 push edi
002A0151 push ecx
002A018B7424 20 mov esi, dword ptr ss:[esp+20]
002A018B7C24 1C mov edi, dword ptr ss:[esp+1C]
002A018B4C24 34 mov ecx, dword ptr ss:[esp+34]
002A01F3:A4 rep movsb
002A0159 pop ecx
002A015F pop edi
002A015E pop esi
002A0101E8 add eax, ebp
002A018D342A lea esi, dword ptr ds:[edx+ebp]
002A01EB 9F jmp 2A026B
002A015F pop edi
002A015E pop esi
002A015D pop ebp
002A015B pop ebx
002A0183C4 08 add esp, 8
002A01C3 ret
```

Registers (Show FPU):

- EAX: 00000000
- EBX: 002B0004
- ECX: 000002D1
- EDX: 000094DE
- EBP: 000002D1
- ESP: 0019E260
- ESI: 010094DE
- EDI: 002C0000
- EIP: 002A02C0

Other registers: EFLAGS: 00000202, ZF: 0, PF: 0, AF: 0, OF: 0, SF: 0, DF: 0, CF: 0, TF: 0, IF: 1. LastError: 000000, LastStatus: C00001. GS: 002B, FS: 0053, ES: 002B, DS: 002B, CS: 0023, SS: 002B. DR0: 002C0000, DR1: 00000000, DR2: 00000000, DR3: 00000000, DR6: FFFF4FF0, DR7: 00010001.

Default (stdcall) stack frame:

- 1: [esp+4] 002B0004
- 2: [esp+8] 00000000
- 3: [esp+C] 002B0004
- 4: [esp+10] 002A01F3

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.

Press enter or click to view image in full size

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%.y..6S;`i						
010094EE	00 00 00 00	8B F5 CC B6	55 C5 84 06	1B C8 D6 07	....øiUA...EÖ.						
010094FE	C6 D4 C5 60	AF FF FF 00	00 00 FF 00	00 00 57 E3	Å0A`yy...y...wä						
0100950E	5E D9 35 AC	00 FF 00 FF	FF 00 00 00	FF 00 00 00	^05~.y.yy...y...						
0100951E	00 15 1C 90	5A 04 71 70	6A 39 04 7A	78 65 BB BB	...Z.qpj9.zxe»»						
0100952E	00 FF 00 FF	00 BD B1 5C	B8 9C B5 6E	3E 05 96 3C	.y.y.%±\..µn>..<						
0100953E	3B F1 91 E9	C3 00 00 00	00 00 00 00	00 FF FF 00	;ñ.éÅ.....yy.						
0100954E	FF 00 04 F5	ED AF A4 95	44 00 FF 00	FF 00 FF 00	y..øi`a.D.y.y.y.						
0100955E	14 7A 78 74	75 3E 1D 23	A0 8E E6 DD	D6 98 00 00	.zxtu>.#.æYÖ...						
0100956E	00 FF 8D 20	FE 0C 72 93	70 82 00 00	00 FF 00 00	.y. p.r.p...y..						
0100957E	FF 00 00 FF	00 FF DF 1C	C1 95 FF 00	00 00 FF B2	v..v.vp.A.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%.y..6S;`i						
002C0010	00 00 00 00	8B F5 CC B6	55 C5 84 06	1B C8 D6 07	....øiUA...EÖ.						
002C0020	C6 D4 C5 60	AF FF FF 00	00 00 FF 00	00 00 57 E3	Å0A`yy...y...wä						
002C0030	5E D9 35 AC	00 FF 00 FF	FF 00 00 00	FF 00 00 00	^05~.y.yy...y...						
002C0040	00 15 1C 90	5A 04 71 70	6A 39 04 7A	78 65 BB BB	...Z.qpj9.zxe»»						
002C0050	00 FF 00 FF	00 BD B1 5C	B8 9C B5 6E	3E 05 96 3C	.y.y.%±\..µn>..<						
002C0060	3B F1 91 E9	C3 00 00 00	00 00 00 00	00 FF FF 00	;ñ.éÅ.....yy.						
002C0070	FF 00 04 F5	ED AF A4 95	44 00 FF 00	FF 00 FF 00	y..øi`a.D.y.y.y.						
002C0080	14 7A 78 74	75 3E 1D 23	A0 8E E6 DD	D6 98 00 00	.zxtu>.#.æYÖ...						
002C0090	00 FF 8D 20	FE 0C 72 93	70 82 00 00	00 FF 00 00	.y. p.r.p...y..						
002C00A0	FF 00 00 FF	00 FF DF 1C	C1 95 FF 00	00 00 FF B2	v..v.vp.A.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 a debugger interface with several tabs: CPU, Graph, Log, Notes, Breakpoints, and Memory Map. The CPU tab is active, displaying assembly instructions with their addresses and hex values. The instruction at address 002A01E3 is highlighted, showing 'loop 2A02E2'. Below the assembly window, there are tabs for Dump 1 through Dump 5 and Watch 1. The Dump 1 window shows a memory dump with hex and ASCII values. The ASCII column contains the text 'M8Z', which is highlighted with a red box.

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.

The screenshot shows a memory dump with hex and ASCII values. The ASCII column contains the text 'M8Z.8.f...qy.,A.' and other characters. The hex values are shown in columns. The text 'M8Z' is highlighted in red in the original image.

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.

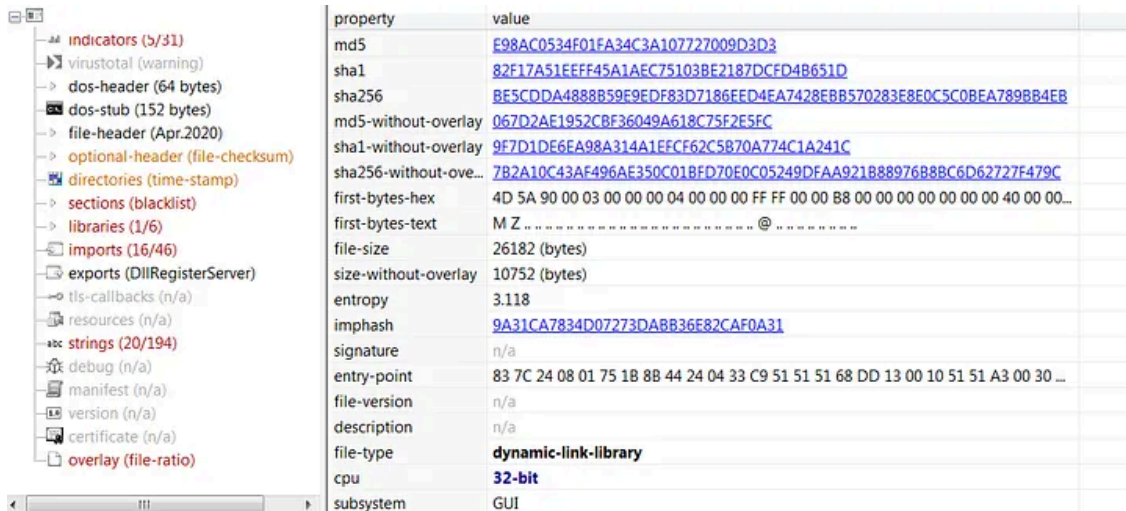
The screenshot shows a debugger window with the following components:

- Assembly List:** A list of instructions with their addresses and mnemonics. The instruction `rep movsb` at address `002A015E` is highlighted with a red box. Other instructions include `inc ecx`, `mov al,10`, `call 2A0236`, `adc al,al`, `jae 2A01E2`, `jne 2A022C`, `stosb`, `jmp 2A01C4`, `call 2A0242`, `sub ecx,ebx`, `jne 2A0209`, `call 2A0240`, `jmp 2A0228`, `lods b`, `shr eax,1`, `je 2A0252`, `adc ecx,ecx`, `jmp 2A0225`, `xchg ecx,eax`, `dec eax`, `shl eax,8`, `lods b`, `call 2A0240`, `cmp eax,7D00`, `jae 2A0225`, `cmp ah,5`, `jae 2A0226`, `cmp eax,7F`, `ja 2A0227`, `inc ecx`, `inc ecx`, `xchg ebp,eax`, `mov eax,ebp`, `mov bl,1`, `push esi`, `mov esi,edi`, `sub esi,eax`, `pop esi`, and `jmp 2A01C4`.
- Registers:** A panel on the right showing register values. `ESI` is `002C000C` and `EDI` is `002C19B1`, both highlighted with red boxes. Other registers include `EAX` (00000001), `EBX` (00000002), `ECX` (00000000), `EDX` (002C1980), `EBP` (00000004), `EIP` (002A0234), and `EFLAGS` (00000216).
- Memory Dump:** A table at the bottom showing memory addresses, hex values, and ASCII characters. The address `002C19A4` contains the hex value `5A 90 00 03` and the ASCII string `Z.....yyLY.`, with the `5A` and `Z` highlighted by red boxes.

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.





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.

---

Source: <https://elis531989.medium.com/funtastic-packers-and-where-to-find-them-41429a7ef9a7>