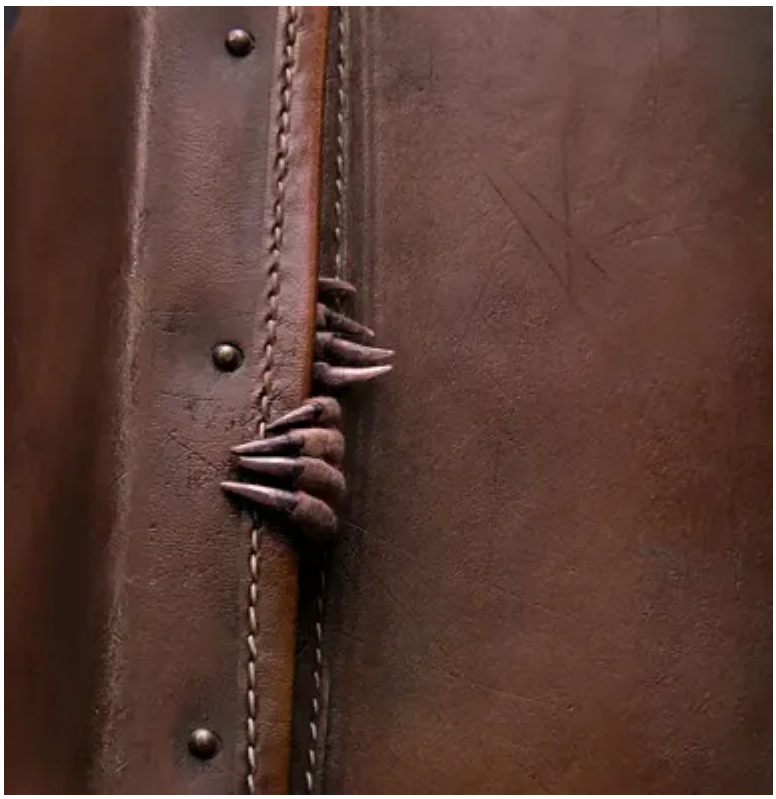
Funtastic Packers And Where To Find Them

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.

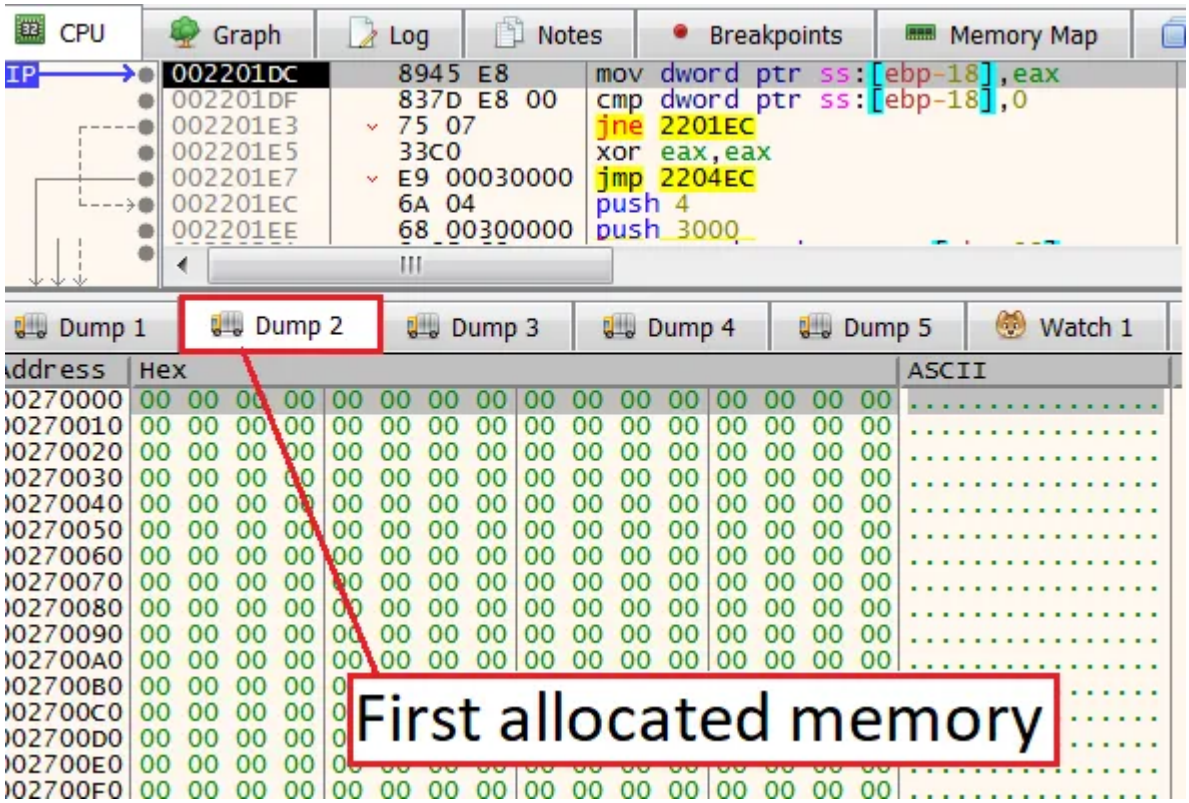
I will also share some theoretical insights regarding the way to approach these packers, and hopefully, shed some light using a step-by-step dynamic observation. The purpose here is to give some guidelines on what to look for when we try to understand how the unpacking mechanism works.

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.

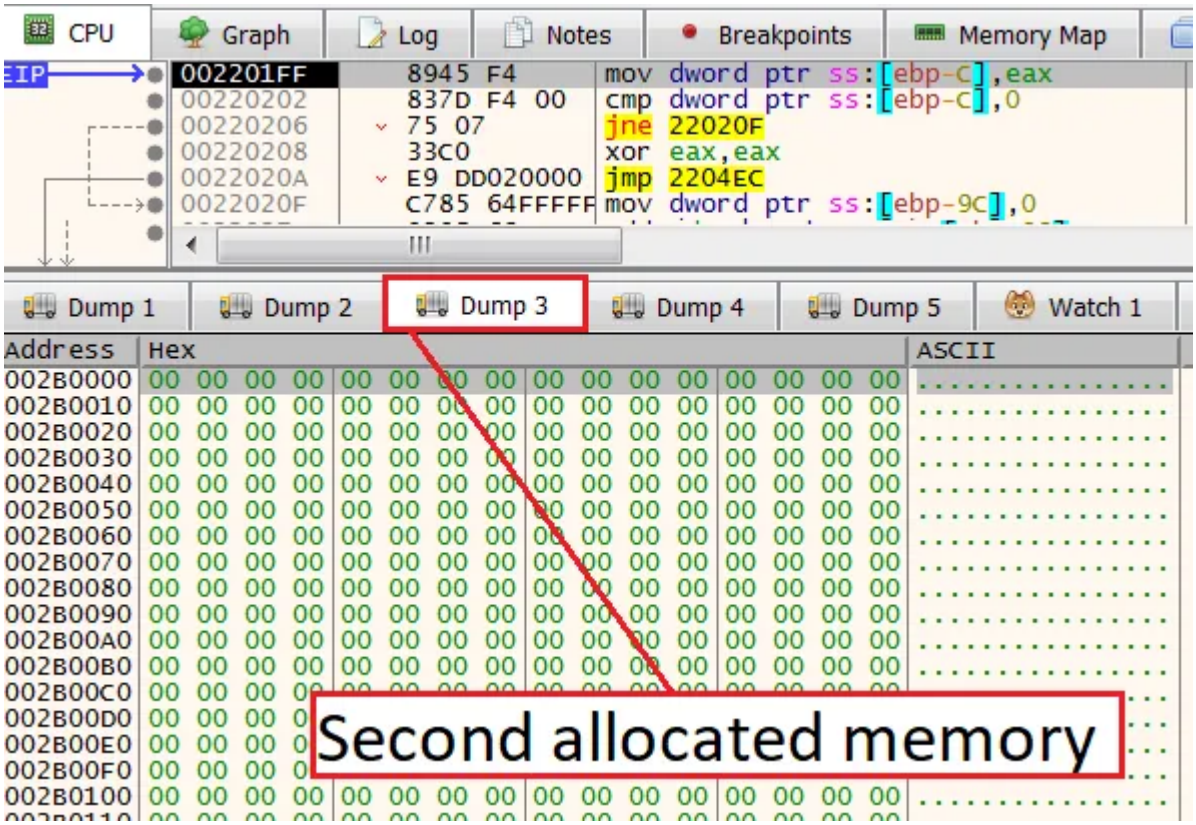
Get2Downloader

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 lt.

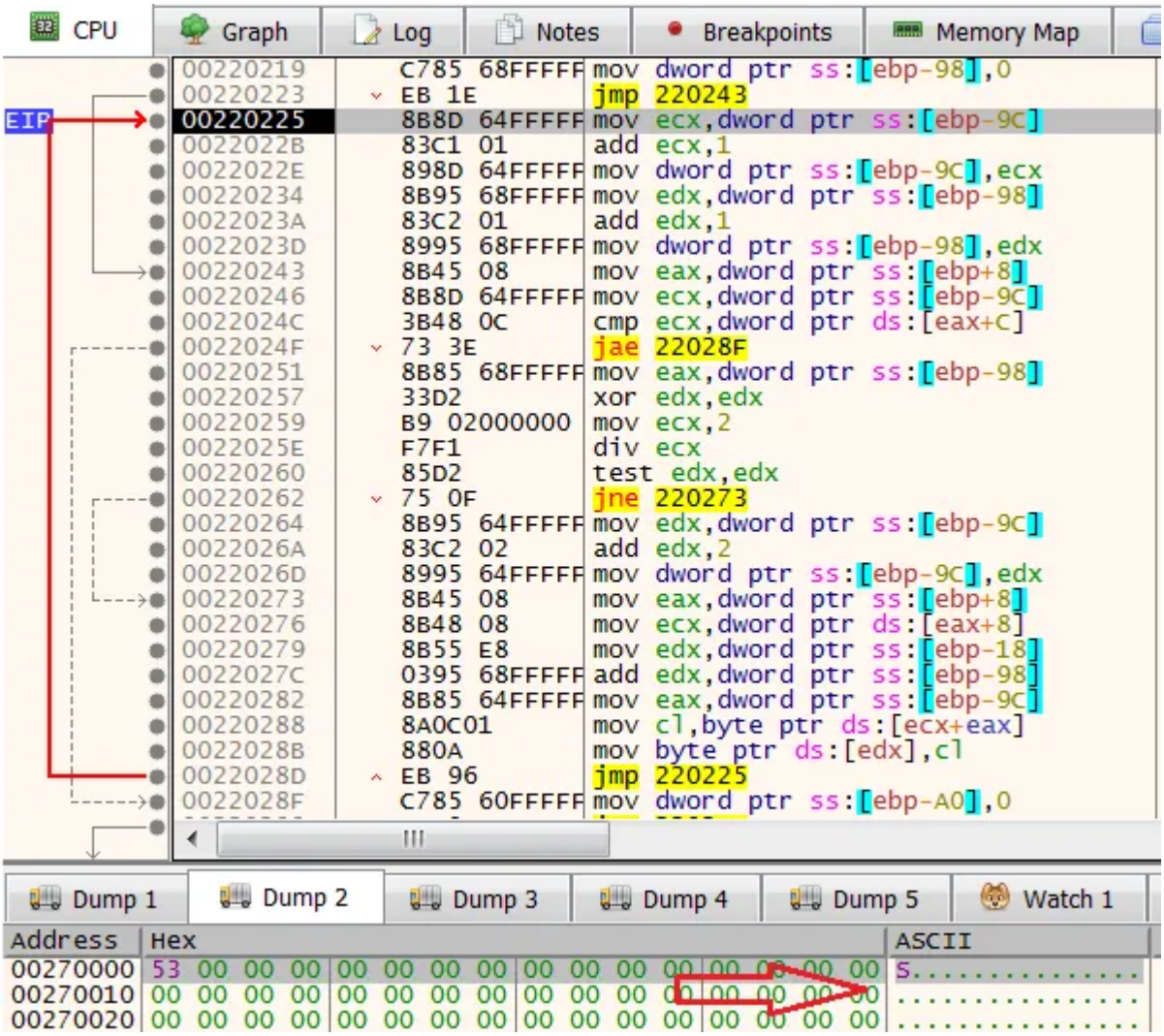


Get2: First Allocated Memory



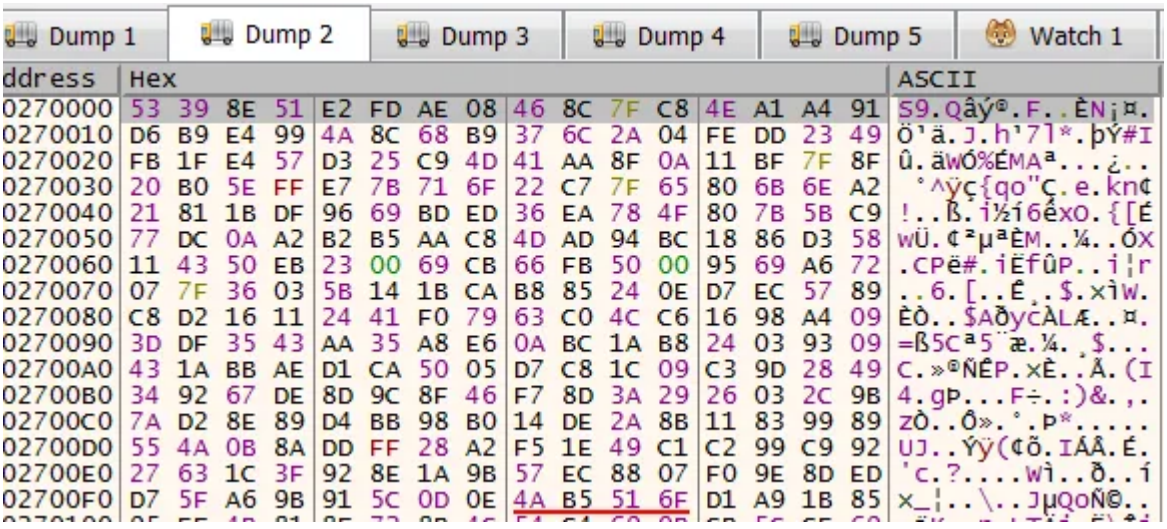
Get2: Second Allocated Memory

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.



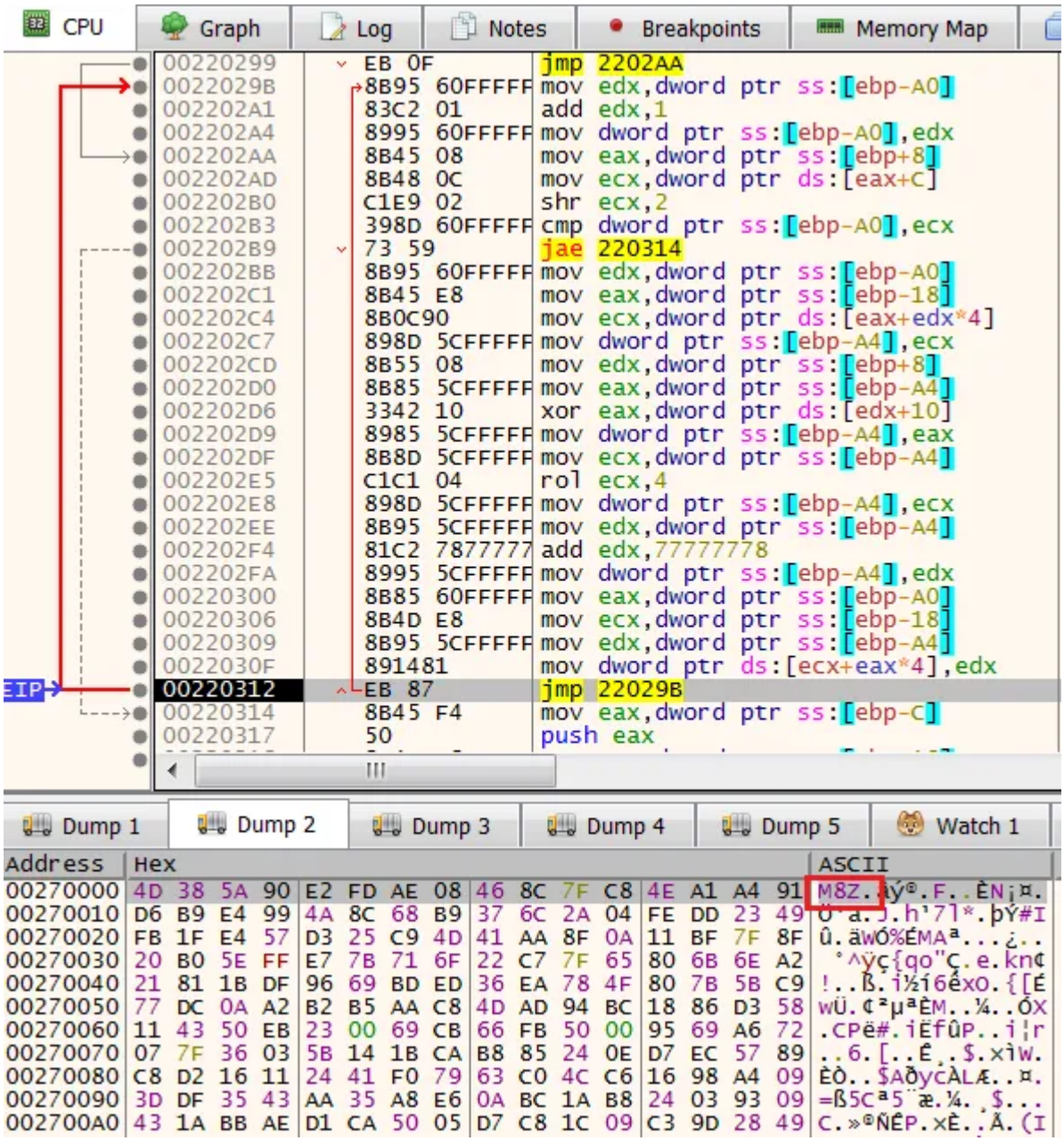
Get2: First Allocated Memory being written

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.



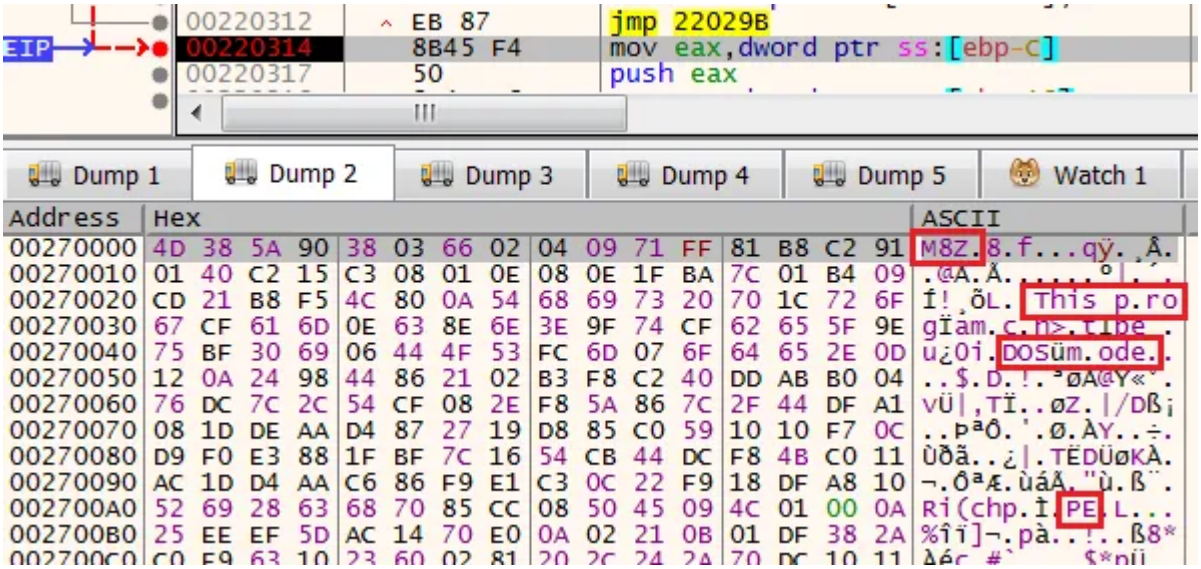
Get2: First Allocated Memory obfuscated content

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.



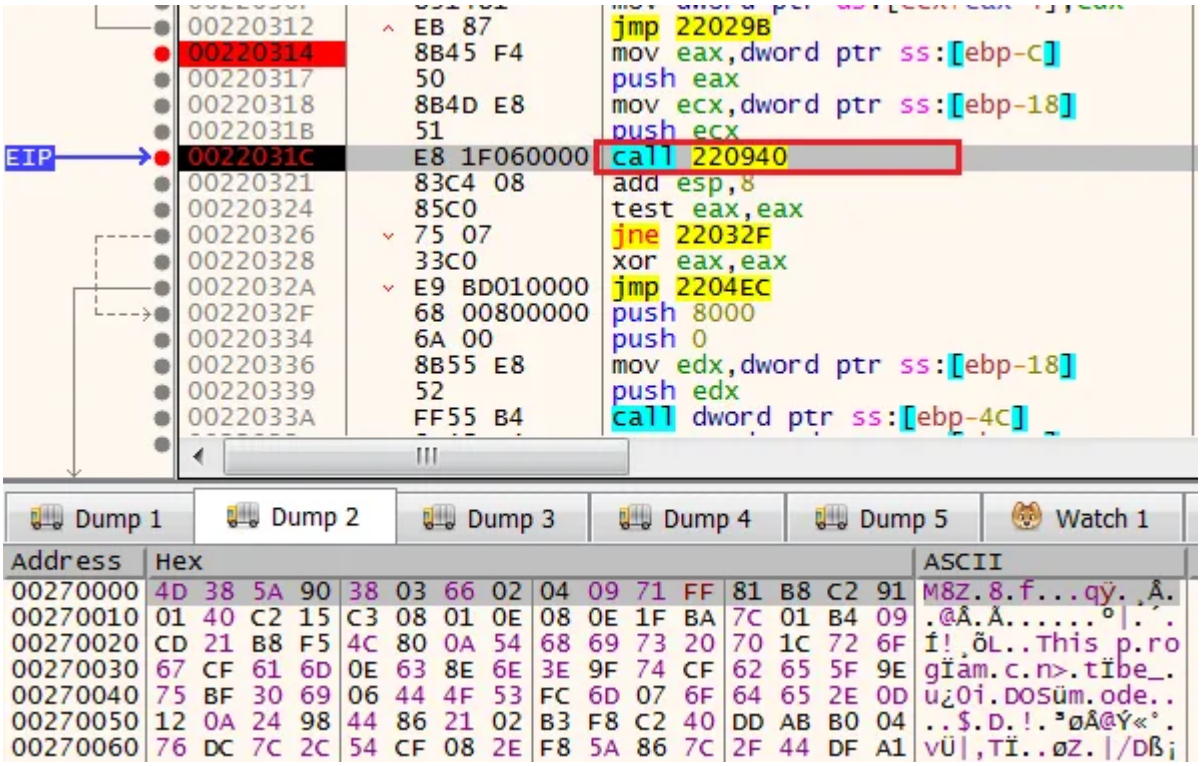
Get2: M8Z string written

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.



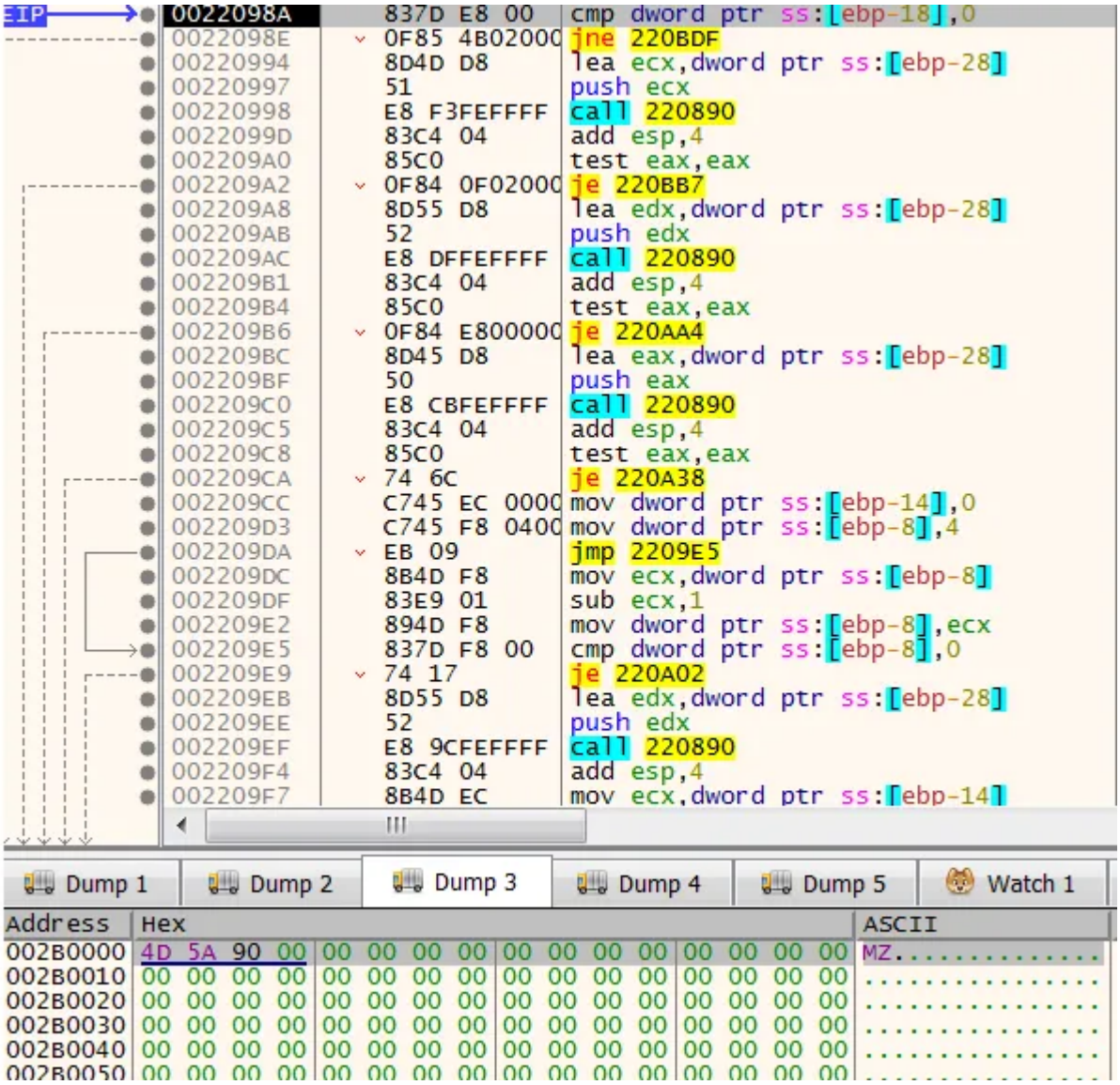
Get2: decryption loop ends

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.



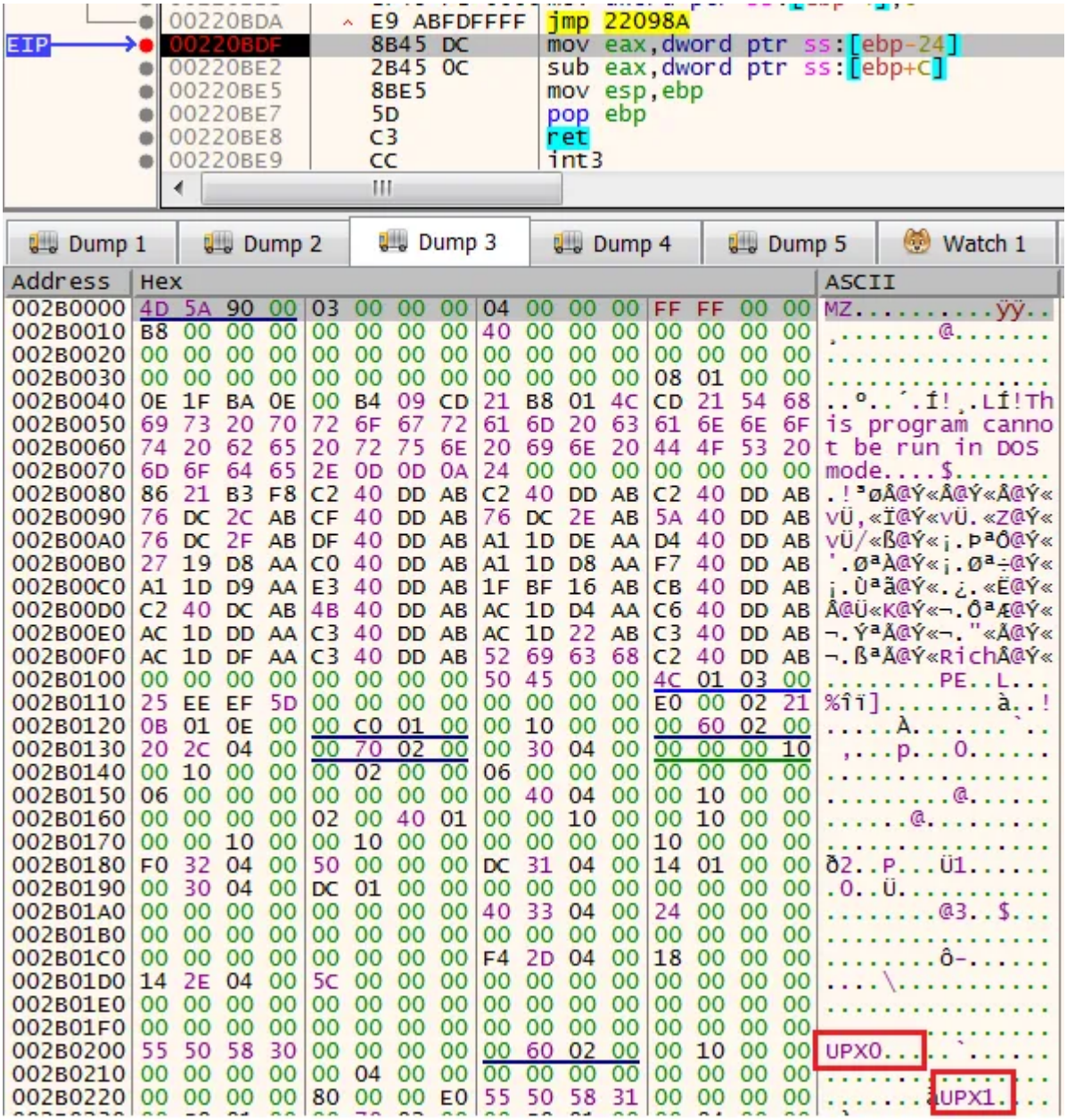
Get2: Second stage decryption function

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.



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.

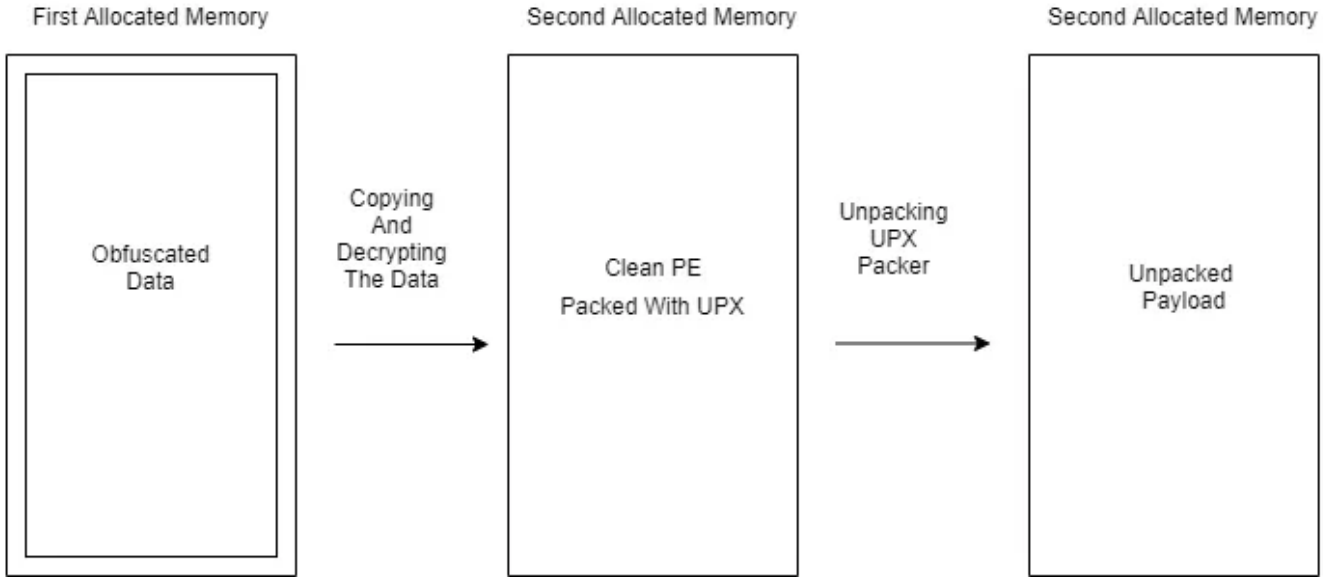


Get2: Second Allocated Memory Packed With UPX

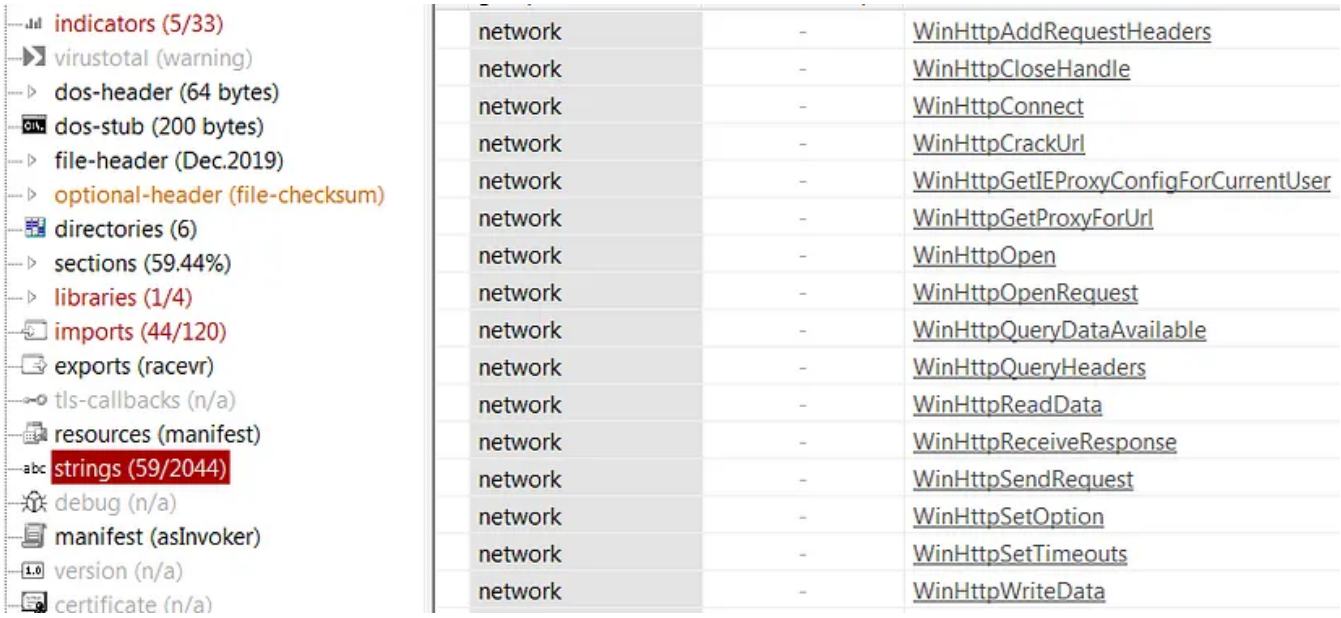
The UPX packer is one of the most common open-source packers. For attackers, it's easy to use packer and it also provides reliability because of the fact that it is not a complex packer. There are several ways to overcome this packer, most of them are documented on the internet. In this case, I choose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



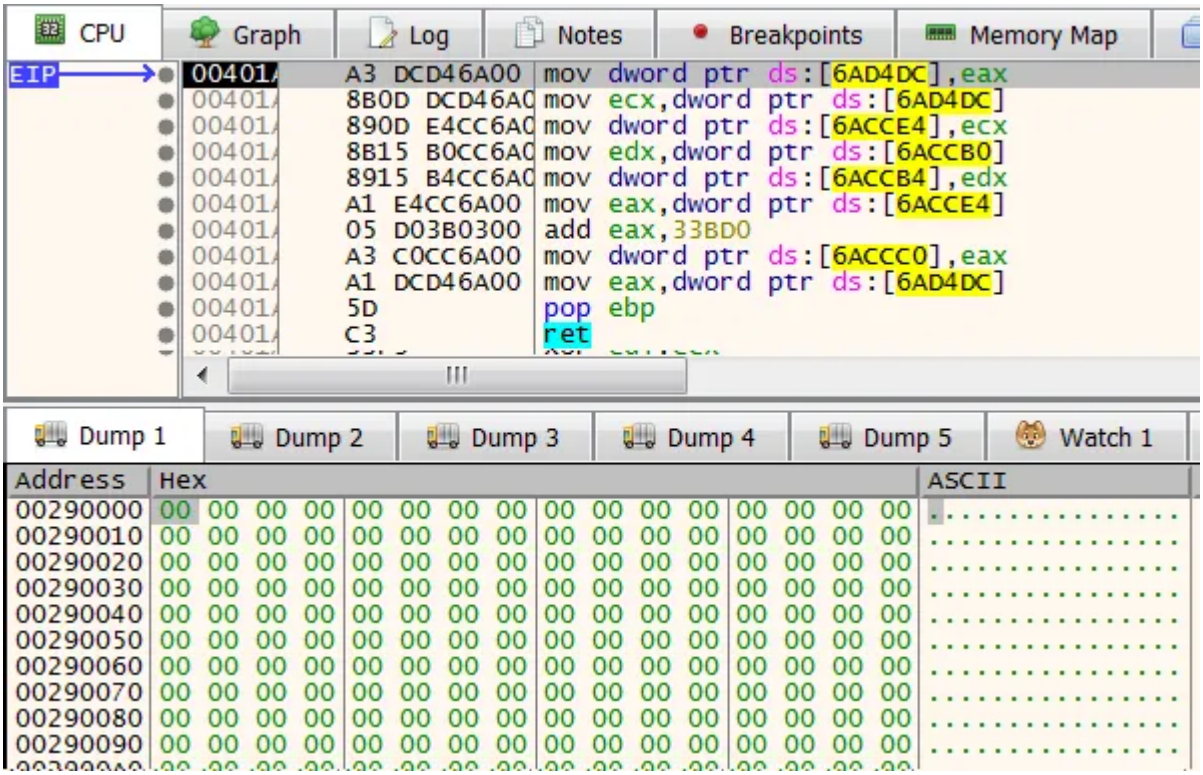
Get2: Final unpacked payload

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.

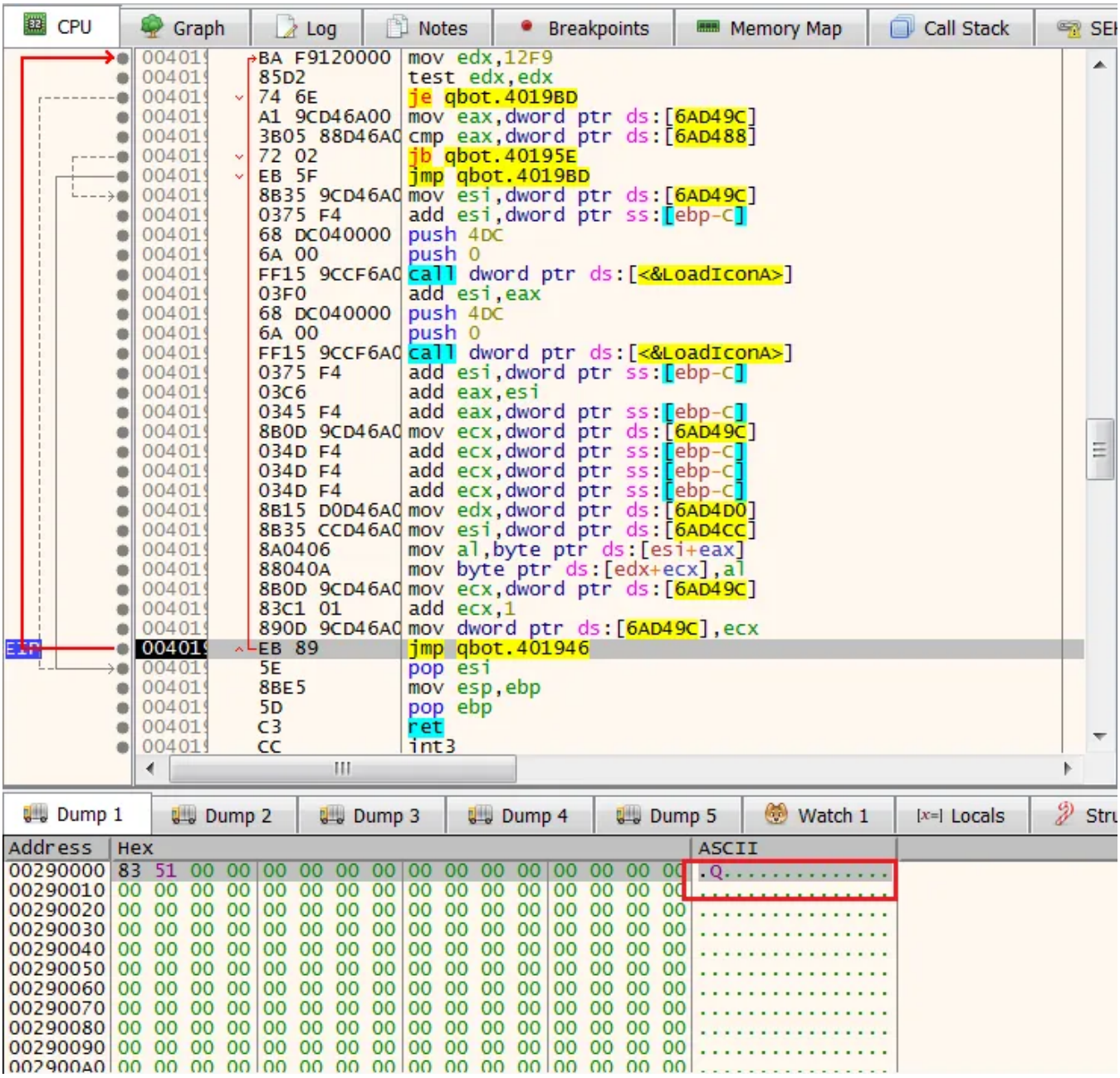
The way to unpack the first dropper of Qbot is relatively simple and already been documented. Setting a breakpoint on *VirtualProtect* and inspecting several *mov* instructions after it, will do the trick. Otherwise, we can use automated tools such as PE-Sieve. However, where is the fun in that?

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.



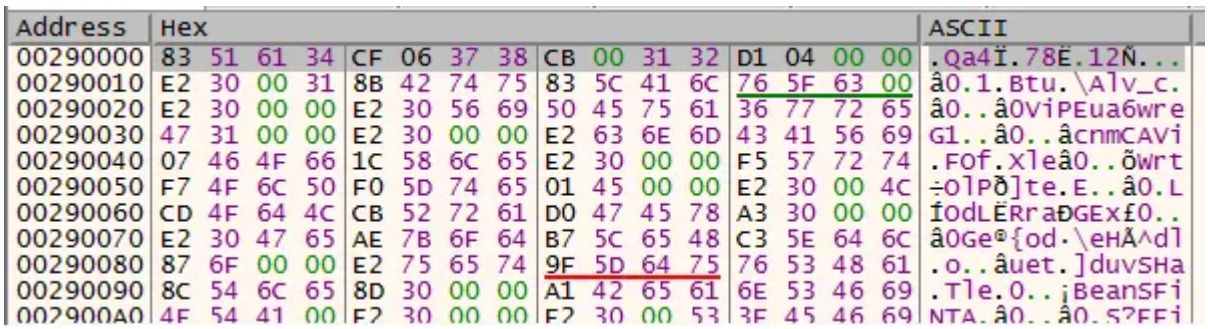
Qbot: VirtualAllocEx allocated memory

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.



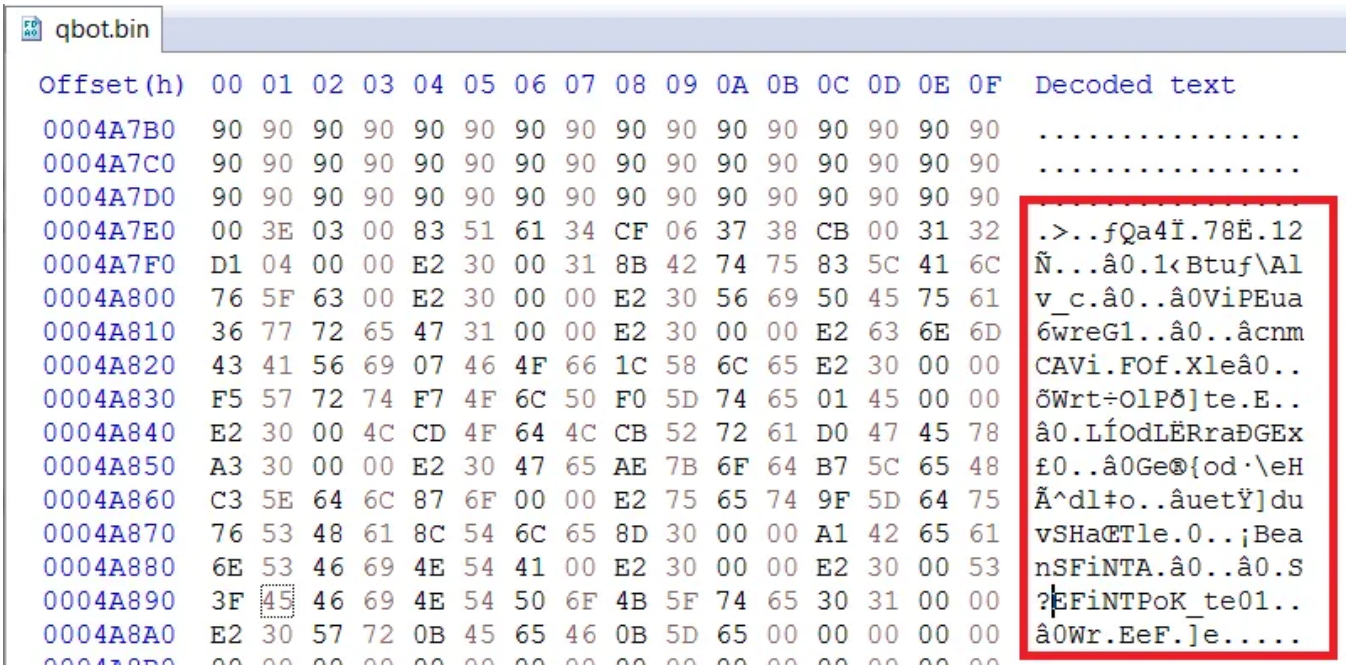
Qbot: Data Written In The Allocated Memory

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.

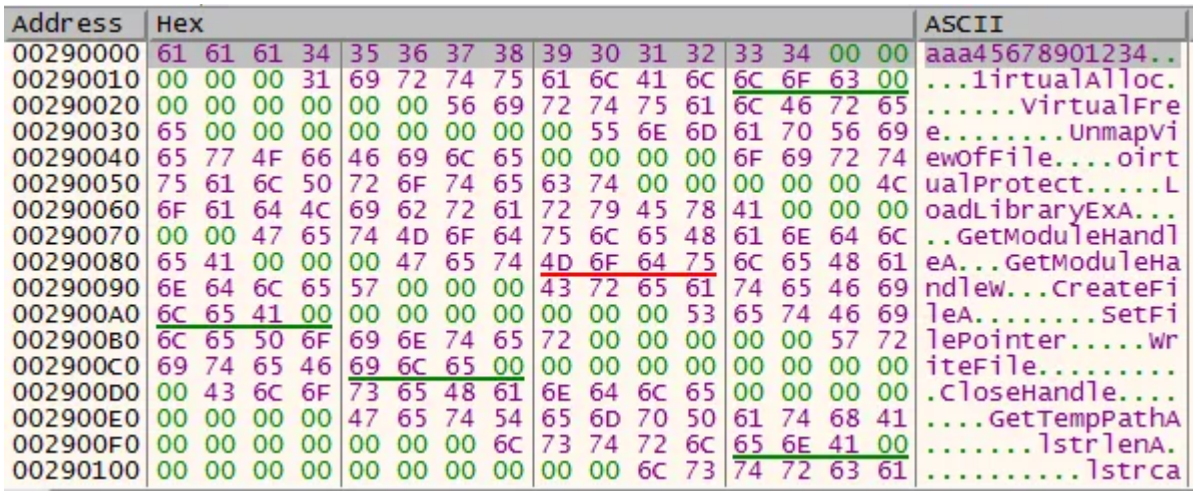


qbot.bin

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	.>..ƒQa4Ï.78Ë.12
0004A7F0	D1	04	00	00	E2	30	00	31	8B	42	74	75	83	5C	41	6C	Ñ...â0.1‹Btuƒ\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÷OlPð]te.E..
0004A840	E2	30	00	4C	CD	4F	64	4C	CB	52	72	61	D0	47	45	78	â0.LÍOdLËRraÐGEx
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	vSHaŒTle.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.....

Qbot: Static analysis — obfuscated content

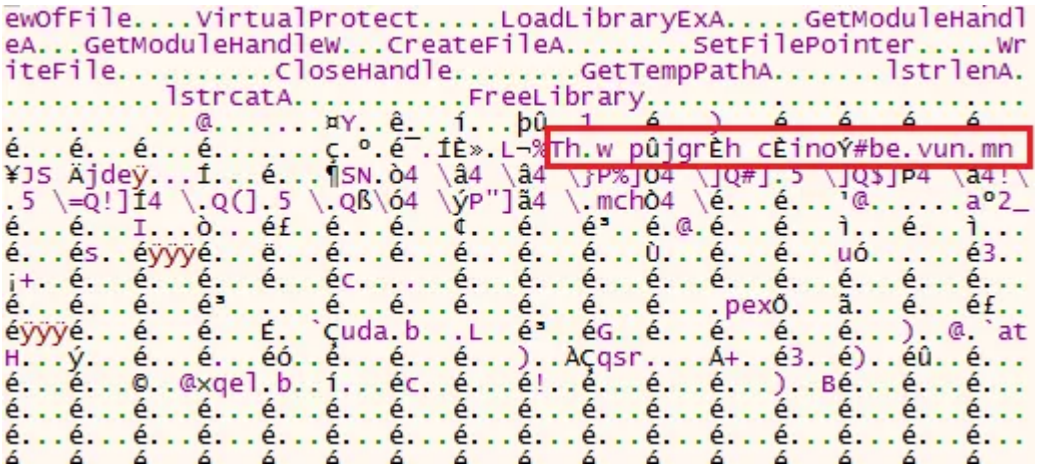
Interestingly, this memory section is being manipulated again and again by several loops. Then, it being overwritten until the upper part is filled with the unknown string “aaa45678901234” followed by several names of API calls.



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	...1irtualAlloc.
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

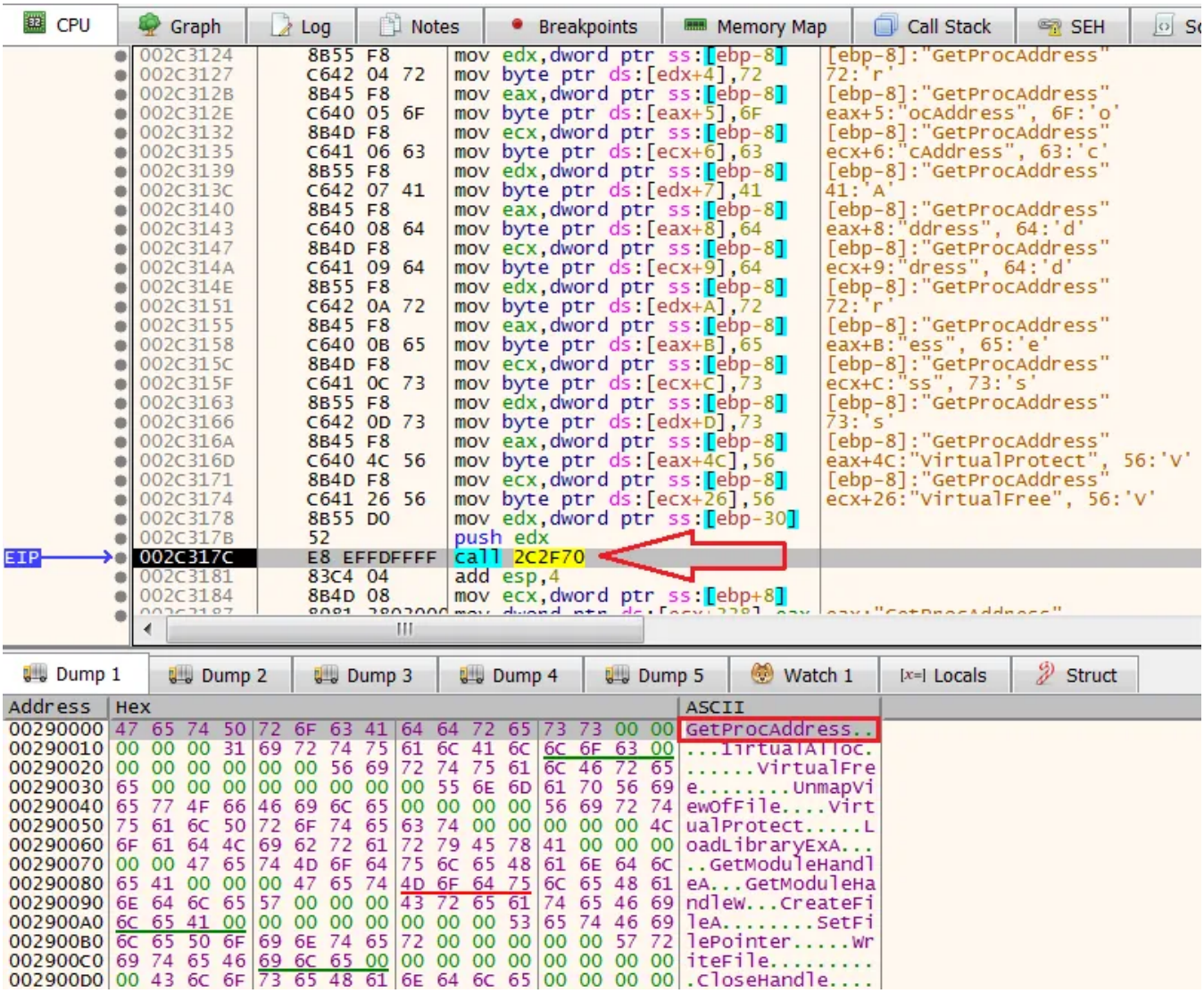
Qbot: Data being Overwritten In The Allocated Memory

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.



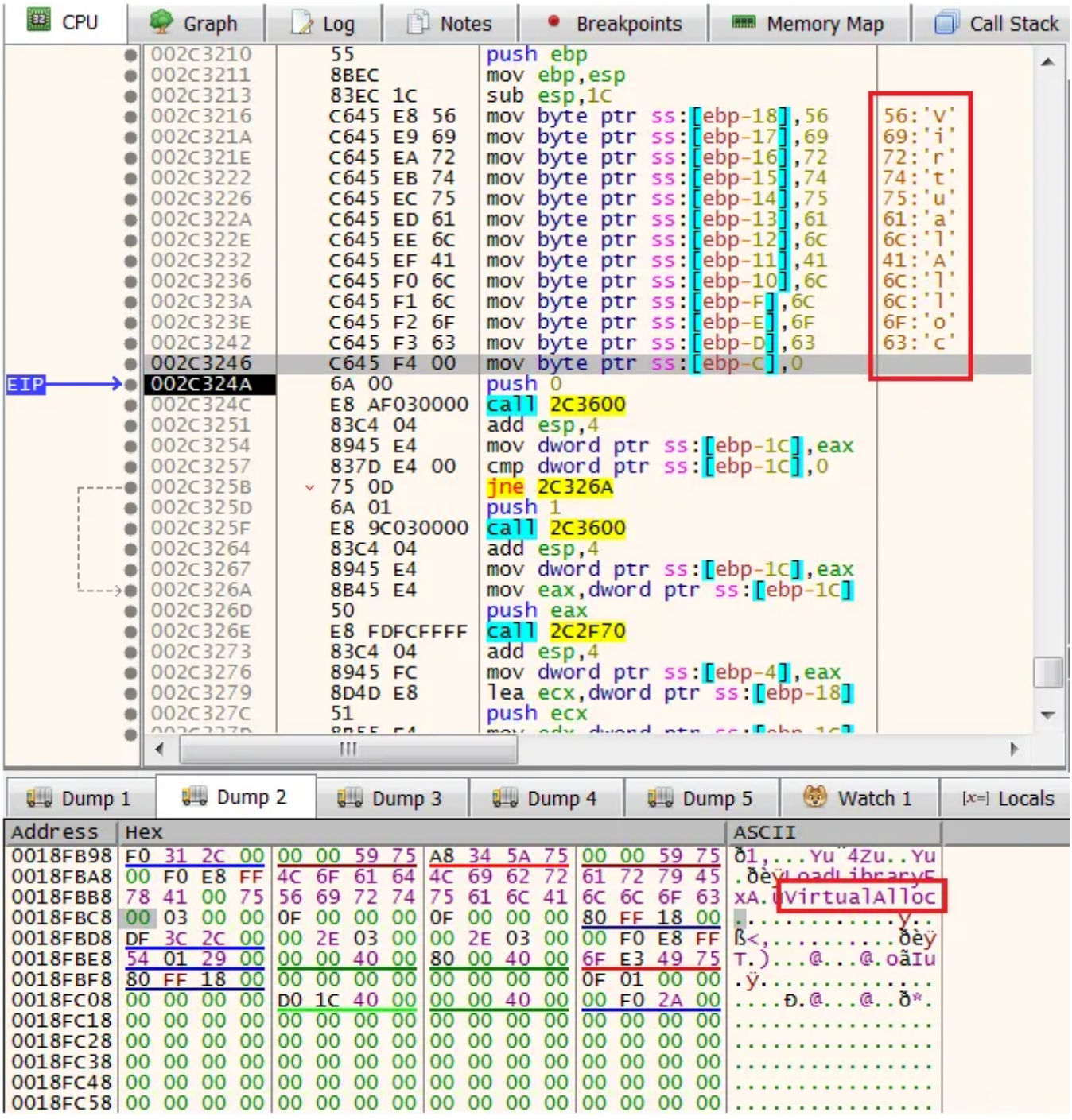
Qbot: Signs Of Obfuscated PE

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.



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.



Qbot: VirtualAlloc Stack-Strings

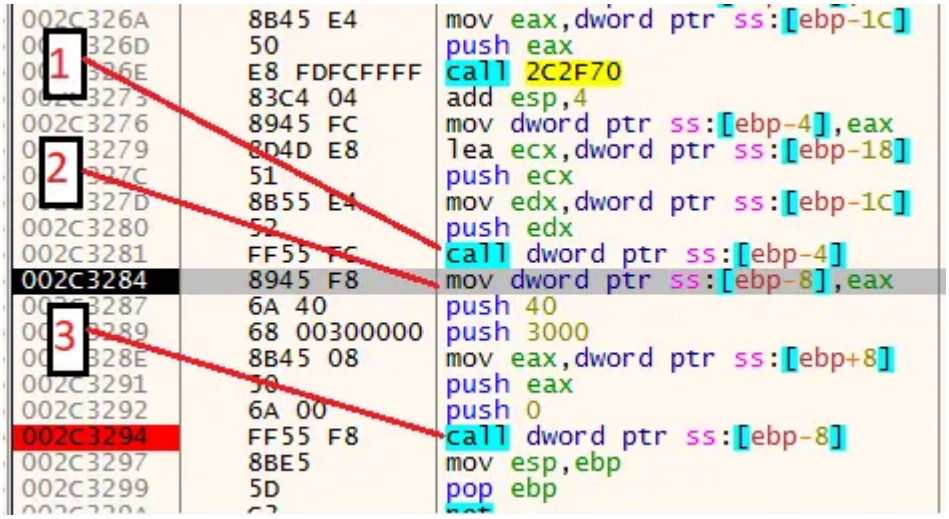
Then, the function will do the following:

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.

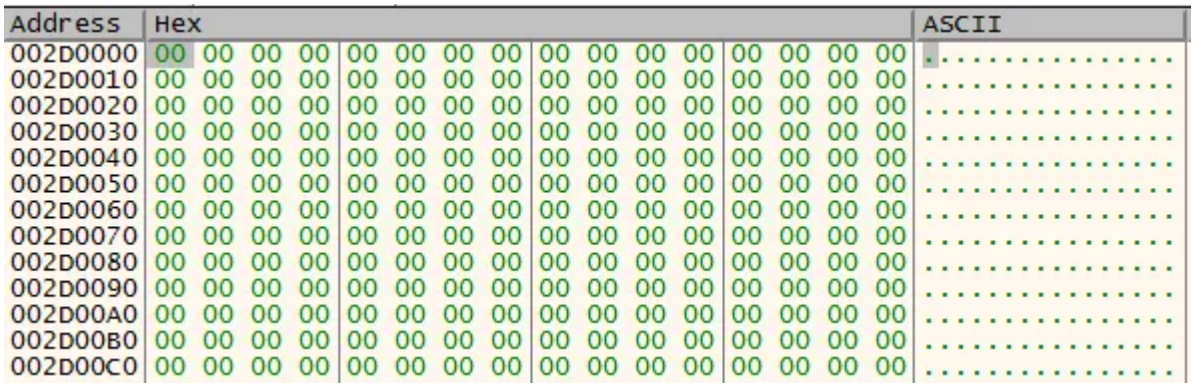
3. *VirtualAlloc* being called with Read-Write-Execute permissions.

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



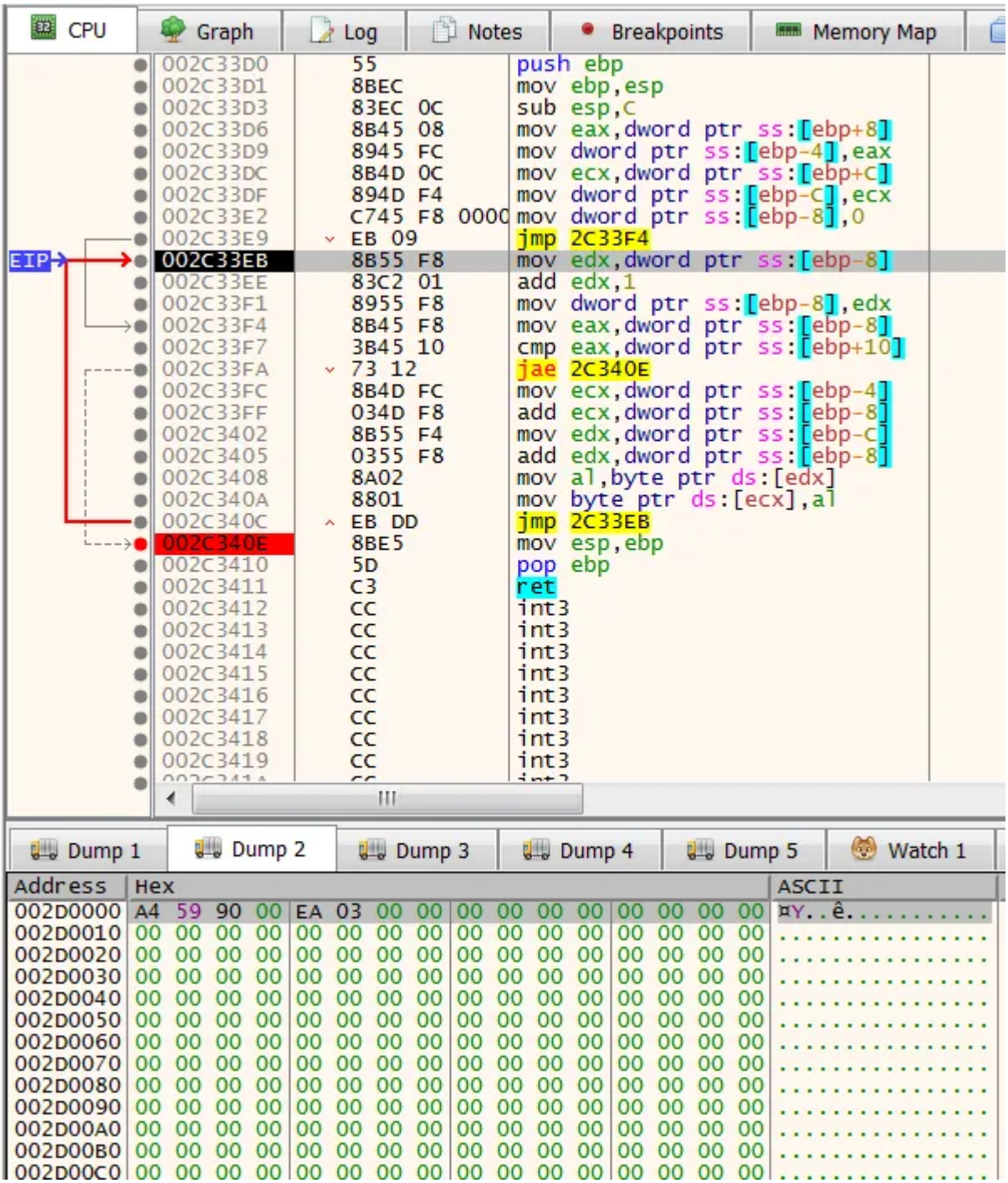
Qbot: Call For VirtualAlloc Steps

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.



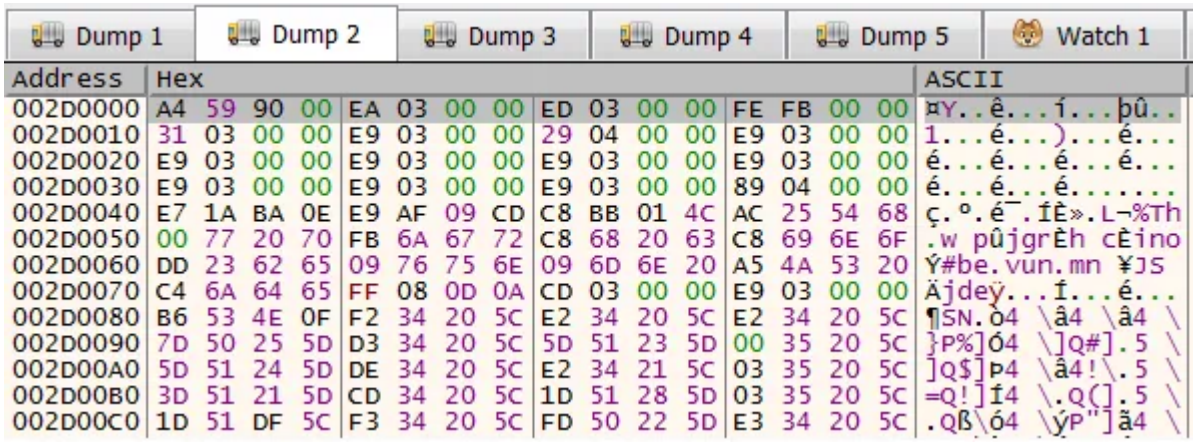
Qbot: Newly Allocated Memory

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.



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.

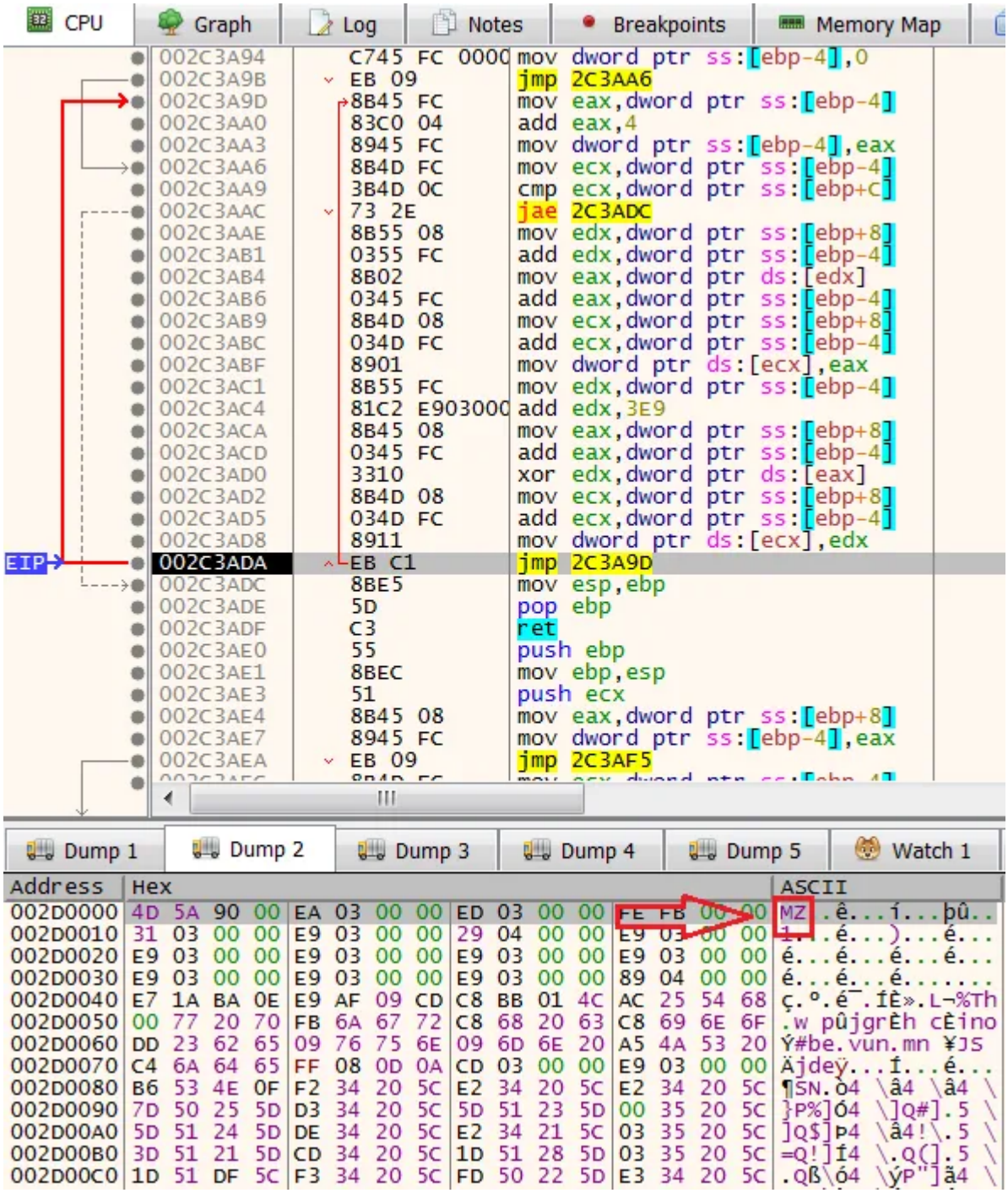


Dump 1 | Dump 2 | Dump 3 | Dump 4 | Dump 5 | Watch 1

Address	Hex	ASCII
002D0000	A4 59 90 00 EA 03 00 00 ED 03 00 00 FE FB 00 00	¤Y..ê...í...þû..
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.º.é¯.ÍÈ».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 ¥JS
002D0070	C4 6A 64 65 FF 08 0D 0A CD 03 00 00 E9 03 00 00	Äjdeÿ...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$]Þ4 \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 \ýP"]ã4 \

Qbot: Data Copied From First To Second Allocated Memory

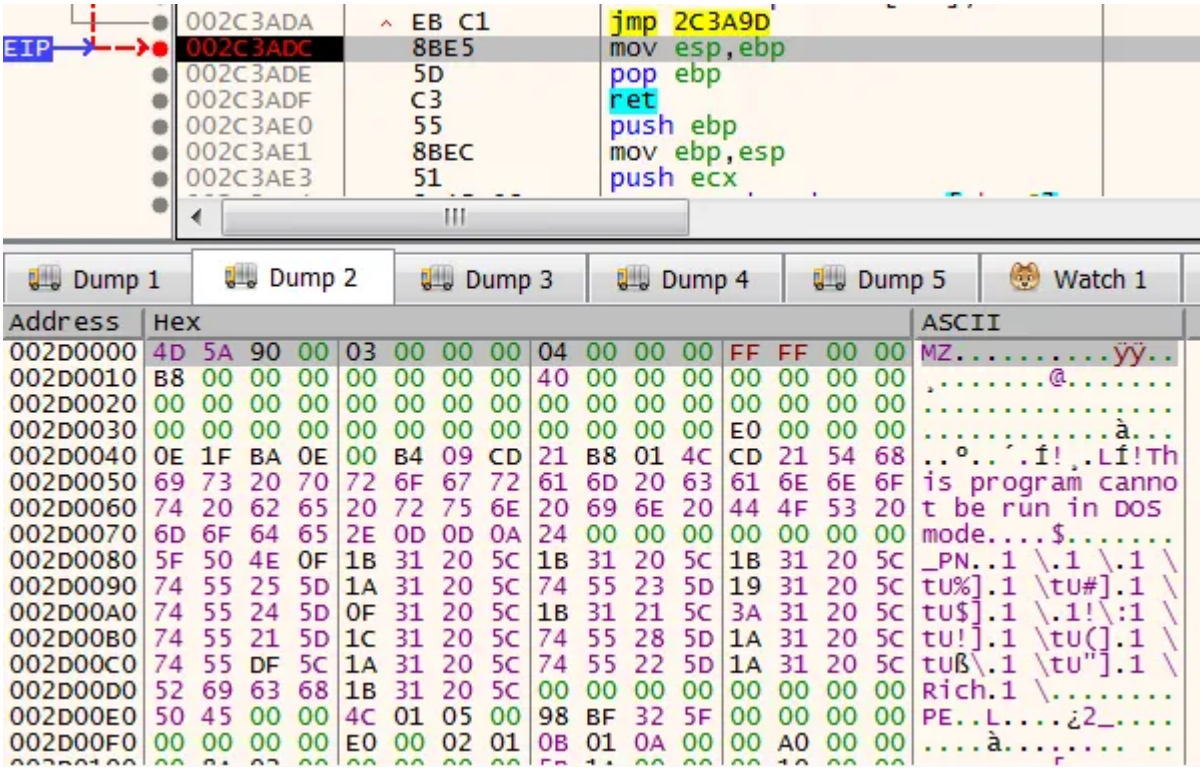
Then, we encounter another loop. This loop deobfuscates the entire second allocated memory. It is noticed by observing the magic string -”MZ” that appears after the first iterations.



Qbot: Second Allocated Memory Deobfuscated

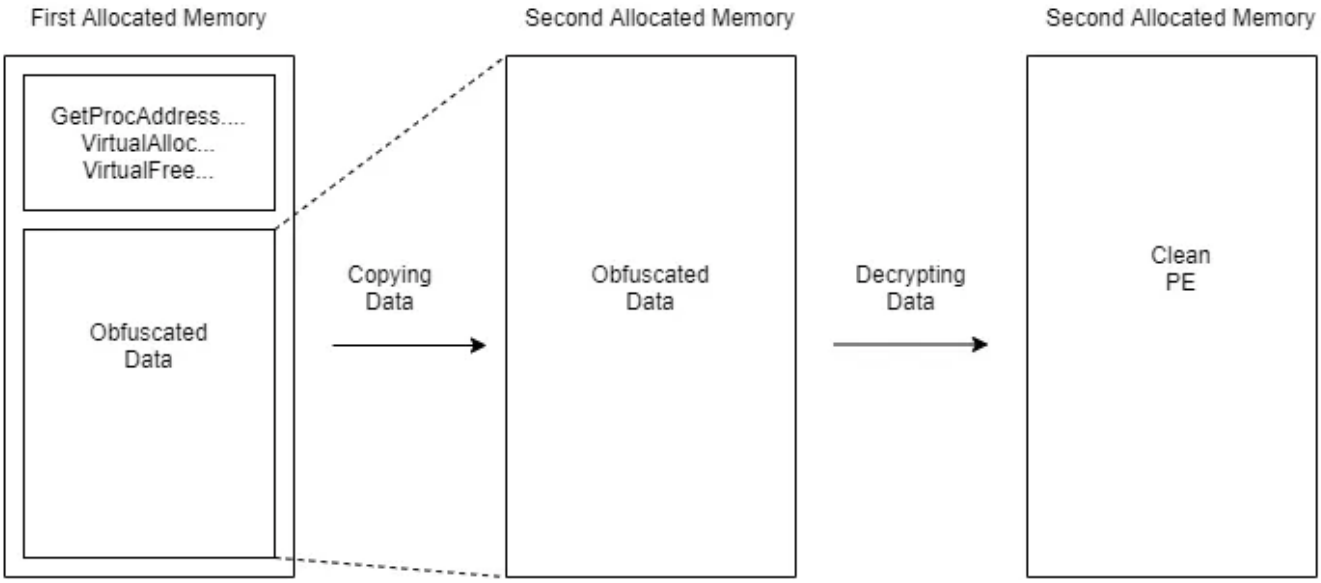
This loop contains several characteristics that already observed in the Get2 malware unpacking mechanism. The use of *xor* and add opcodes.

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.

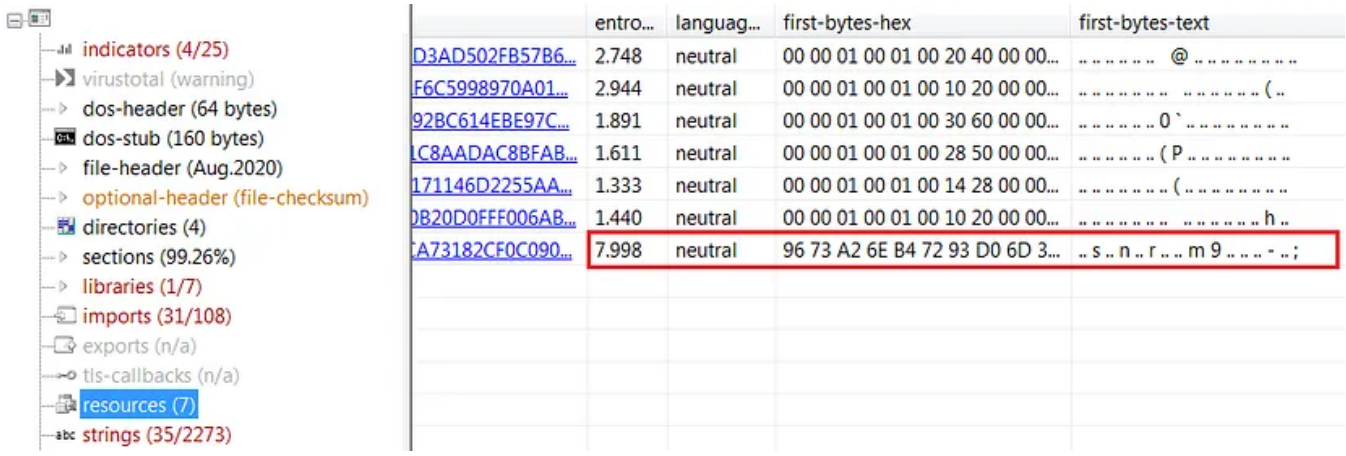


Qbot: Unpacked PE

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.



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

IcedID

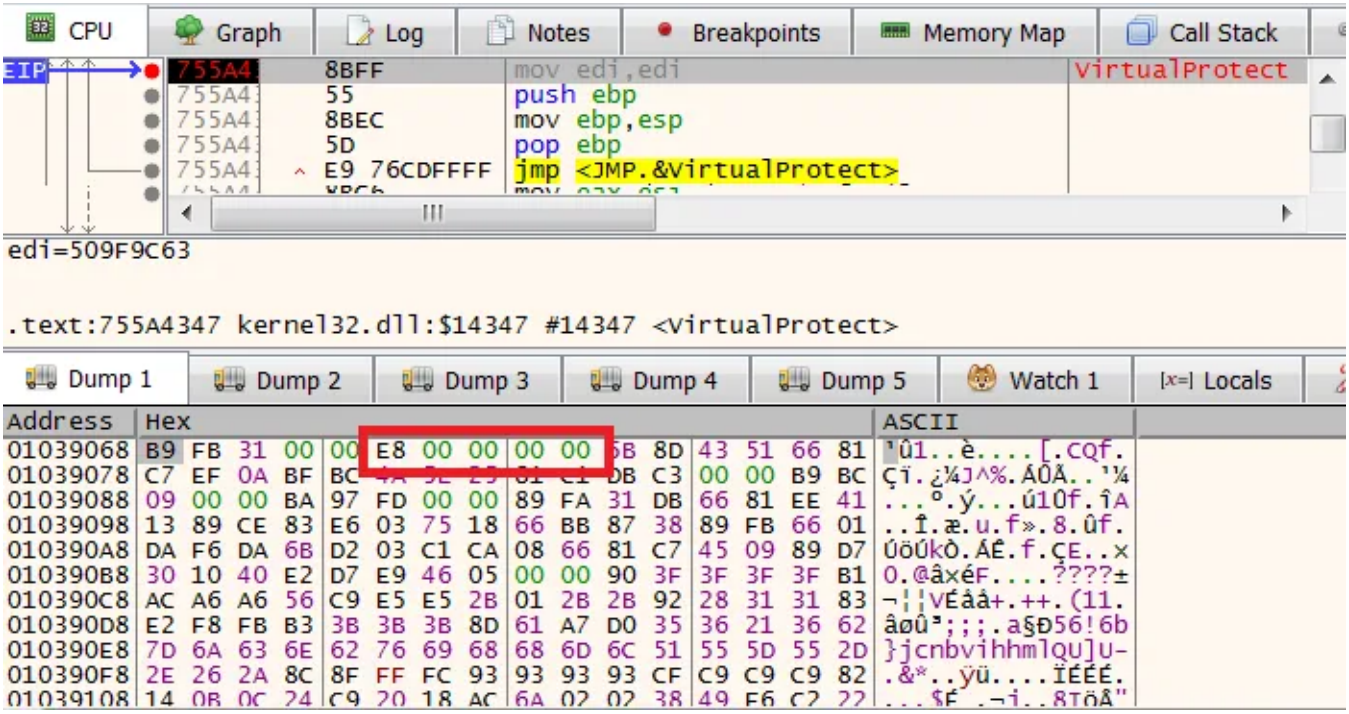
IcedID aka Bokbot is also one of the most prevalent banking trojans in the last years. It is known to be associated with Lunar Spider threat actors.

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*.

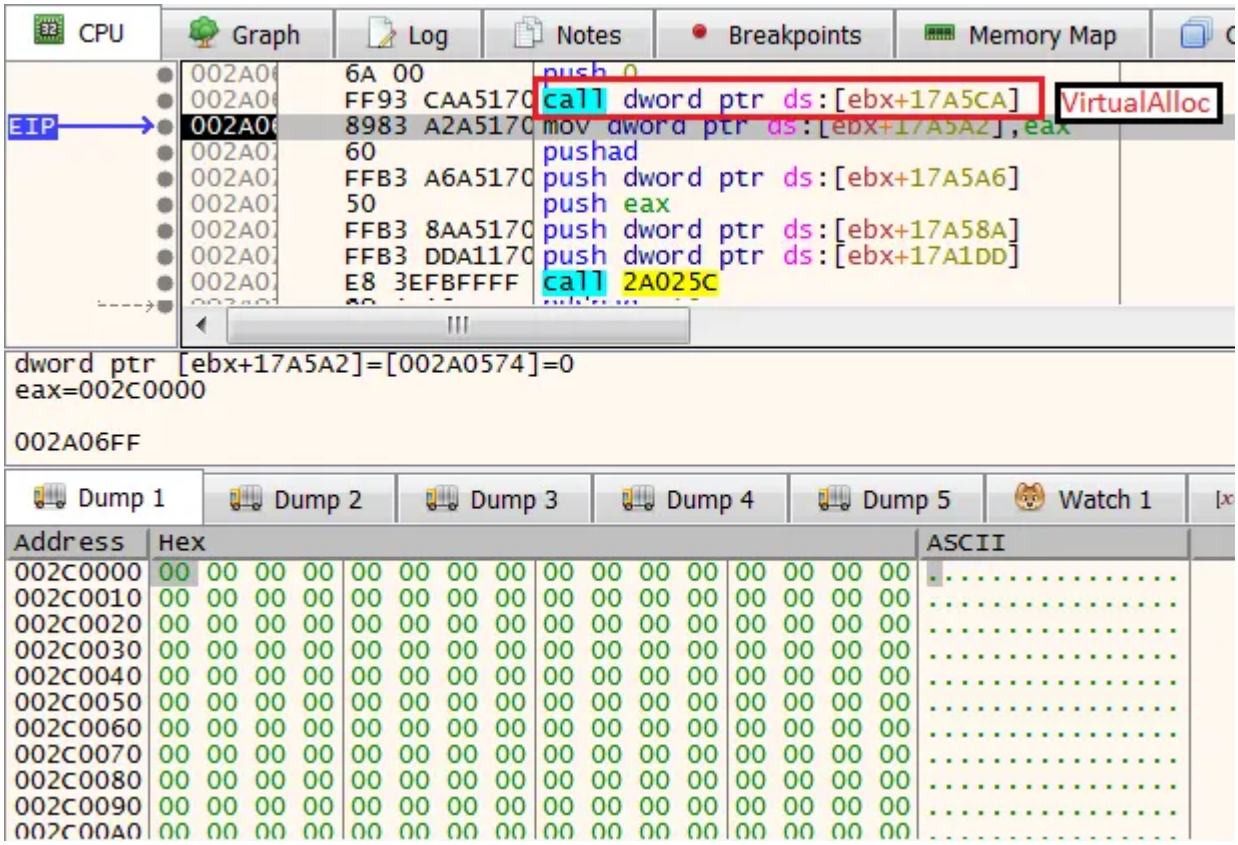
By inspecting the "*lpAddress*" argument in *VirtualProtect* we'll notice that it appears to deal with shellcode execution. This is visible with the byte sequence E8 00 00 00 00 which is a relative call for the next instruction.

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.



IcedID: Shellcode

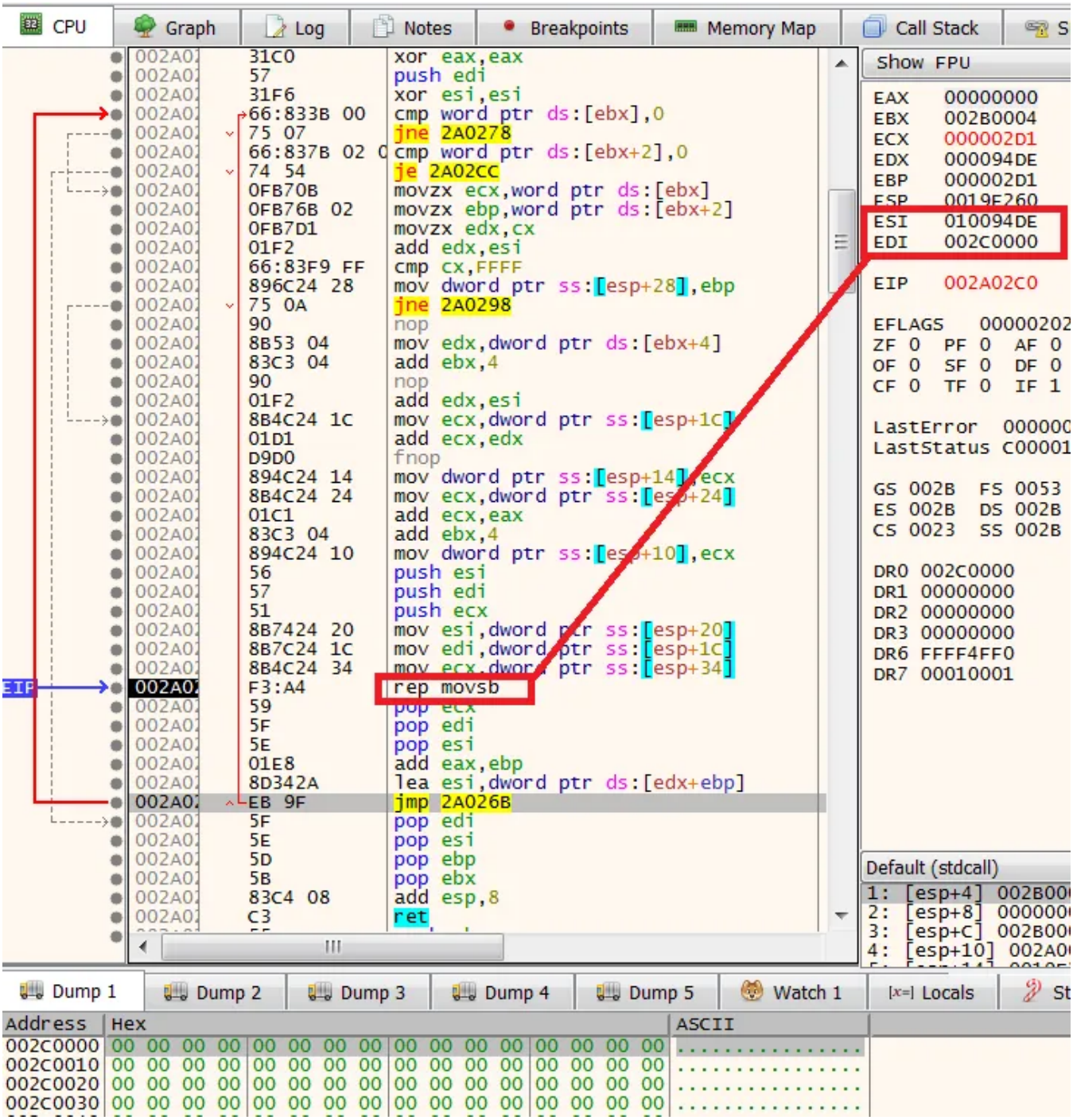
Then, we'll hit Run again and encounter *VirtualAlloc* three different times. In the third time, we'll set a write hardware breakpoint on the newly allocated memory.



IcedID: Call To VirtualAlloc

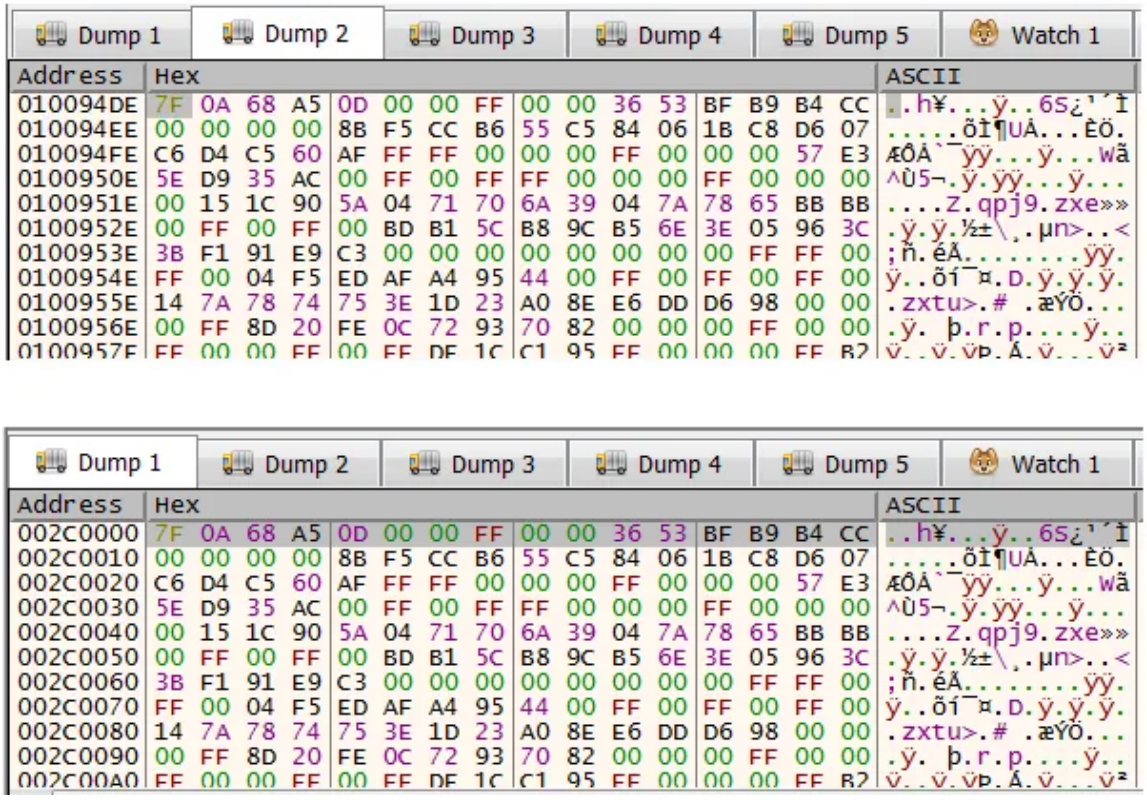
Next, we encounter a function, which has a loop that copies data to the newly allocated memory. Interestingly, in the previous malware, when we saw data moved

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.



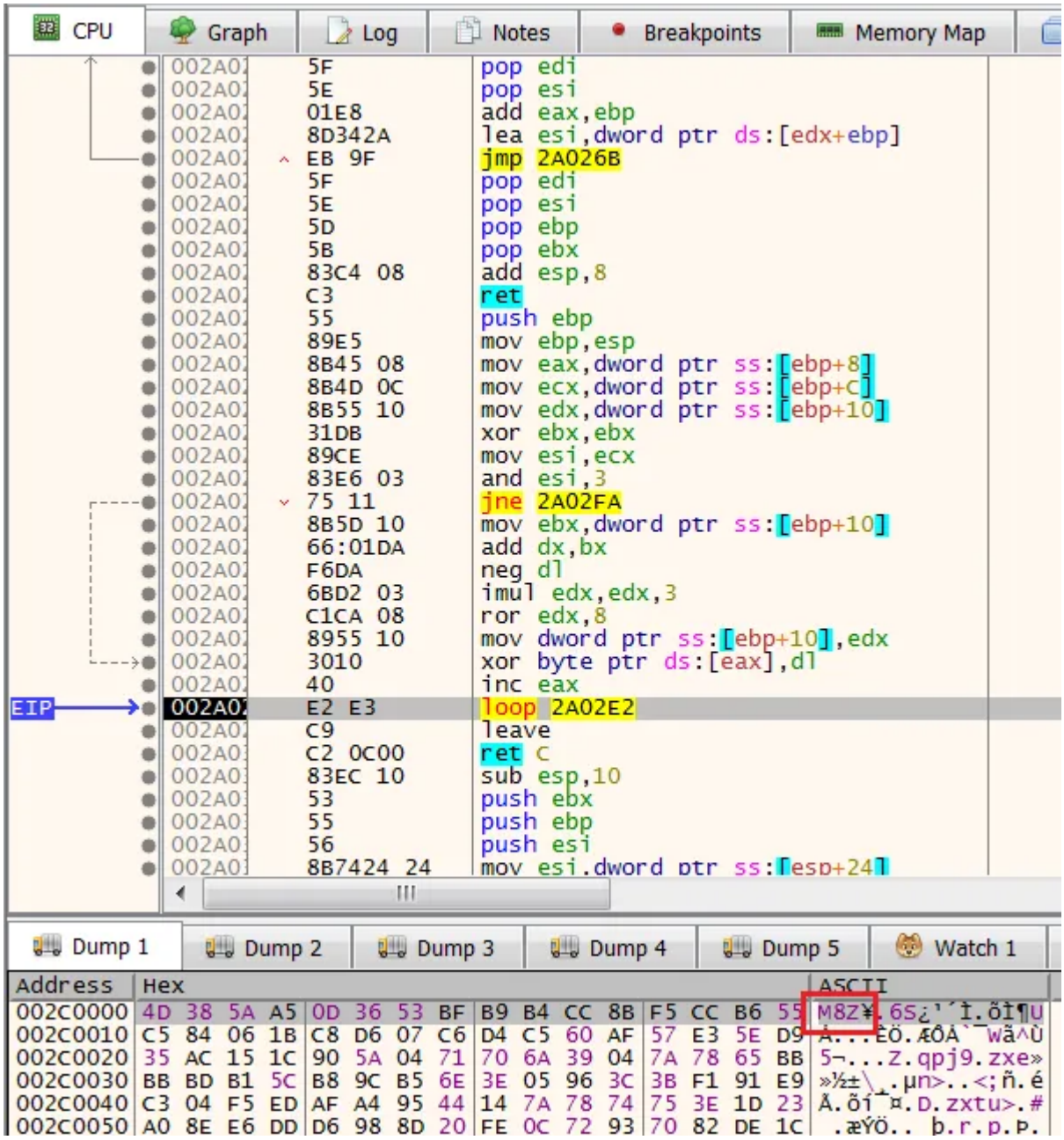
IcedID: Data Copied To The Allocated Memory

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



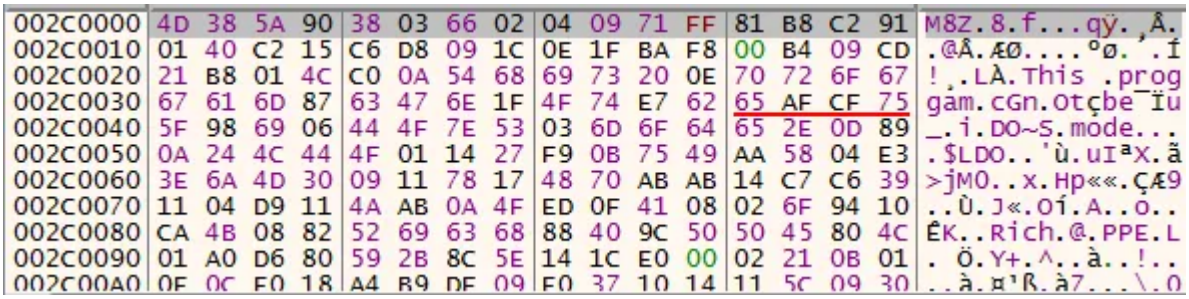
IcedID: Data Comparison ESI vs EDI (Identical)

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).



IcedID: Data being decrypted

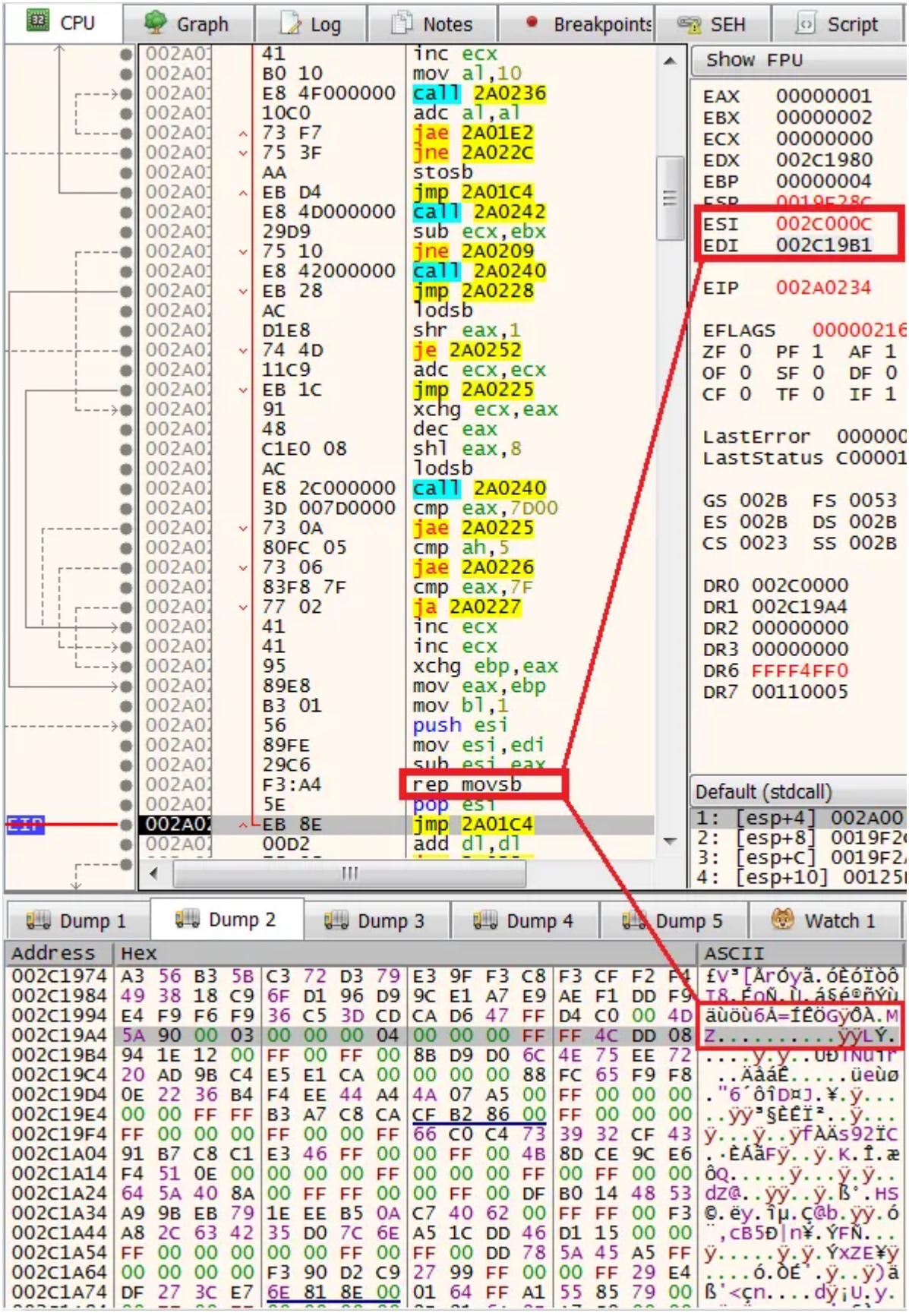
As always, to speed up the process we'll set a breakpoint one step after this loop and we'll hit Run. Once we did that, we can see that some bytes have been changed. Still, it's not a clean portable executable.



IcedID: Data being decrypted

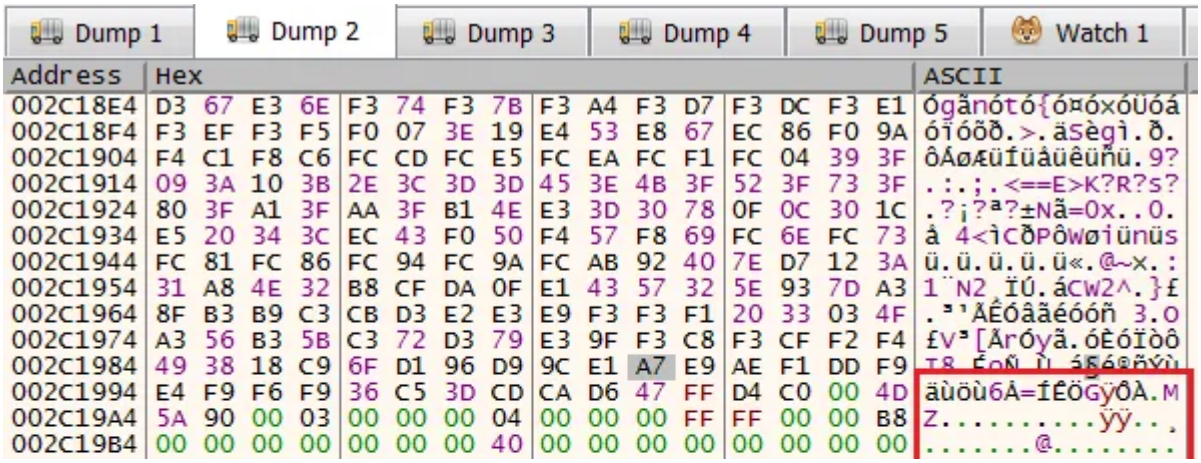
Then, we observe something unusual. We can see again the *movsb* opcode in a larger loop, but now it copies data from the upper part of the allocated memory (the one from the image above) which is stored in the *ESI* register, to an address in the same memory space located several offsets after (which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

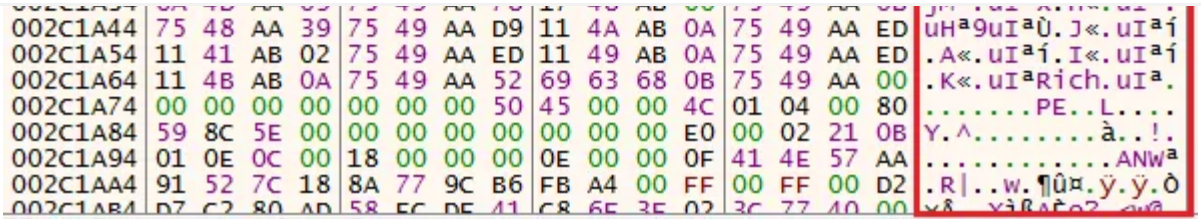
After a few iterations and data manipulation functions, we observe the bytes 4D 5A and the string “MZ” at the beginning of the memory to which the *movsb* opcode writes.



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

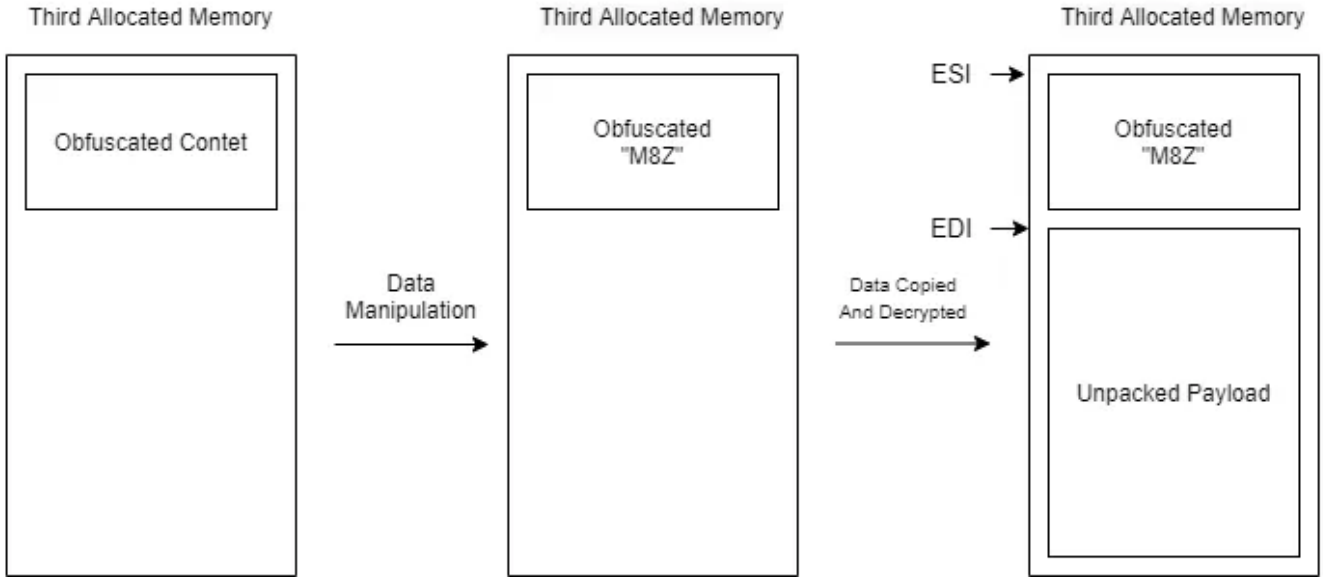
Eventually, we understand the purpose of this loop -writing a clean portable executable in this memory part.





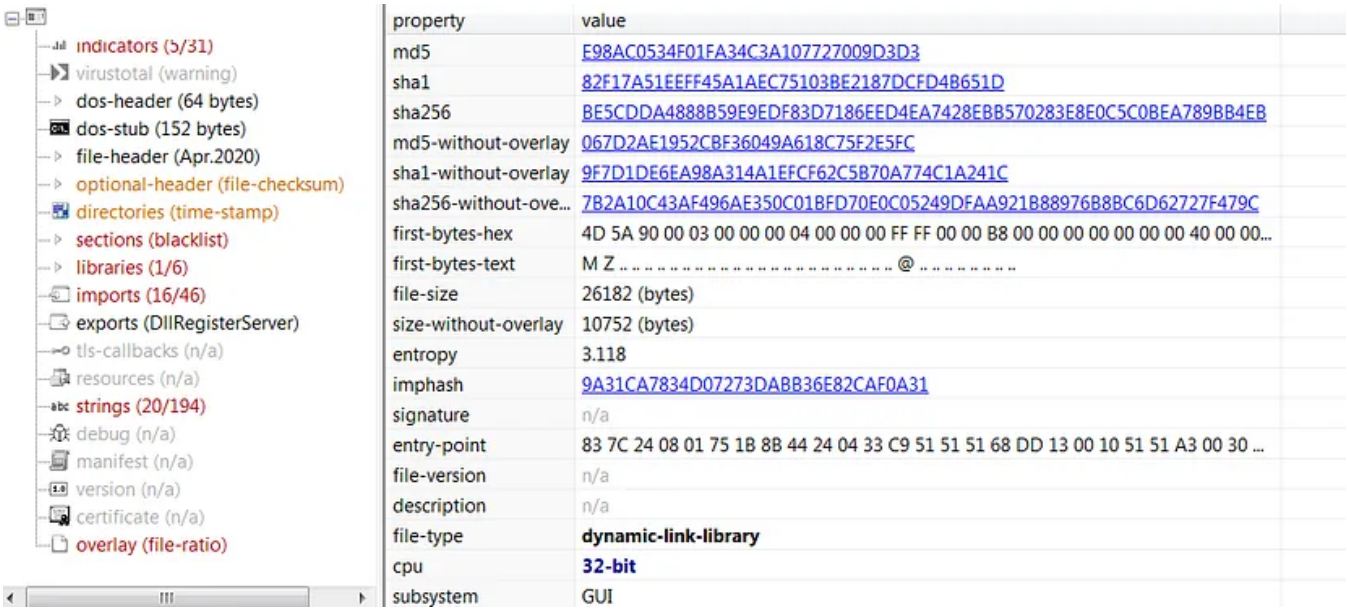
IcedID: Clean PE File

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.



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