

Unpacking Emotet malware part 02

 [muha2xmad.github.io/unpacking/emotet-part-2/](https://github.com/muha2xmad/unpacking/emotet-part-2/)

January 7, 2022



Muhammad Hasan Ali

Malware Analysis learner

5 minute read

As-salamu Alaykum

Part 01 summary

Download the sample: [Here](#)

found `VirtualAlloc` call in `sub_417D50` and its address.

we search for abnormal jumps. we found `jmp ecx` and its address.

Introduction

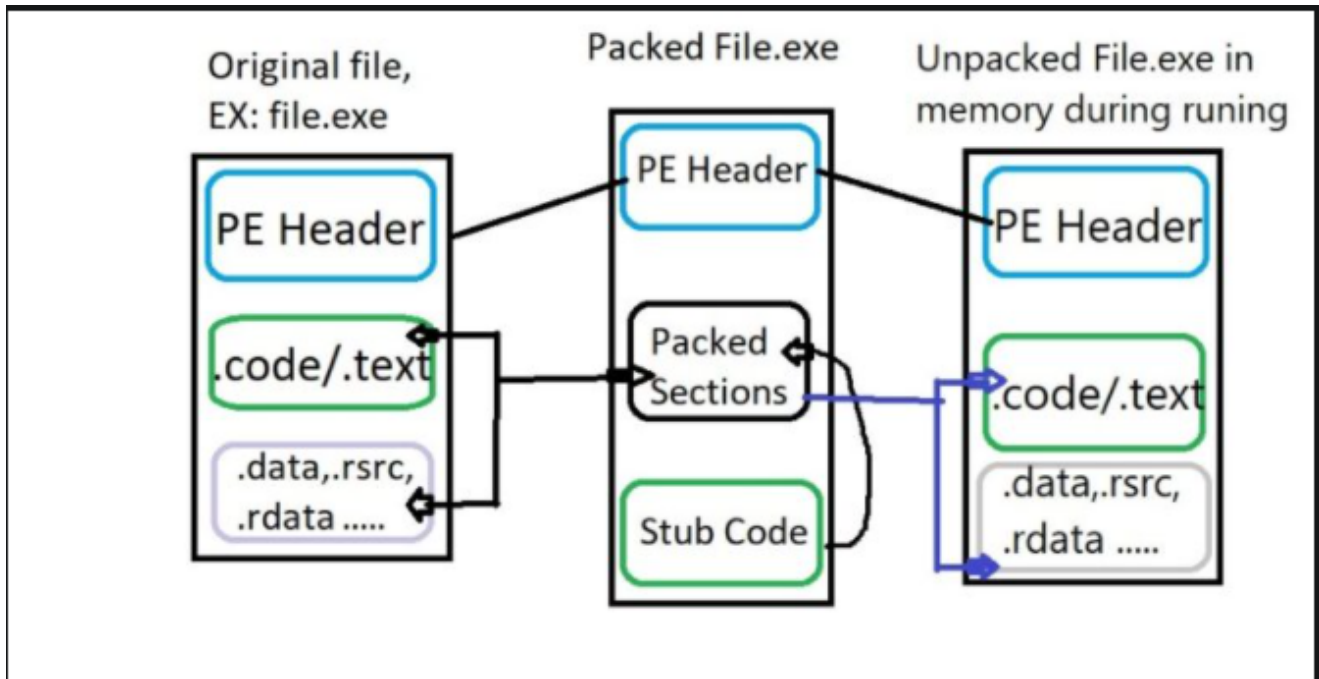
we will debug our sample with x32dbg tool to unpack the Emotet malware

Notes to be taken

- What is Packing: A trick which is Used to avoid AV detection and analysis.
- What is a packer: A tool that compresses, encrypts, and/or modifies a malicious file format. [1](#)
- Why using packers: To avoid AV detection and analysis to make it harder for researchers to analyze the code
- We need to find the the original entry point (OEP). What is the OEP: is the address of the malware's first instruction (where malicious code begins) before it was packed. [2](#)
- How to find the OEP: find the `tail jump` . the tail jump It's an unconditional jump exists in the tail of stub code , it points to address of unpack file. [3](#)

How to the unpack happen? [3](#) As we see in the figure (1). OS create stub code with packed file

What is stub code? [3](#) Stub code is responsible for unpacking packed sections, when you are running the file ,the address of unpack file exists in the stub code to unpack file. So at the end of the stub code we will see an unconditional jump (tail jump), that is meant after execute the stub code will jump to the address of unpacking file.



Figure(1):

What is stack string? [answer](#)

We need to know what is `VirtualAlloc` ? [Here](#)

Says that “Reserves, commits, or changes the state of a region of pages in the virtual address space of the calling process. Memory allocated by this function is automatically initialized to zero.”

Syntax

```
LPVOID VirtualAlloc(
    [in, optional] LPVOID lpAddress,
    [in]           SIZE_T dwSize,
    [in]           DWORD  flAllocationType,
    [in]           DWORD  flProtect
);
```

The most important parameter of this function is `lpaddress` , **which returns the starting offset of the newly allocated memory**. where we will extract the malware then dump it.

Start Debugging

Open our sample by x32dbg and hit the entry point

Address	Hex	Assembly	Comments
00417E00	55	push ebp	EntryPoint
00417E01	8BEC	mov ebp,esp	
00417E03	83EC 0C	sub esp,c	
00417E06	57	push edi	
00417E07	C745 FC 00000000	mov dword ptr ss:[ebp-4],0	
00417E0E	8B55 08	mov edx,dword ptr ss:[ebp+8]	edx:"U<ïfï\FwÇEü"
00417E11	8915 DCC14100	mov dword ptr ds:[41C1DC],edx	
00417E17	892D BCC14100	mov dword ptr ds:[41C1BC],ebp	
00417E1D	C745 FC 00000000	mov dword ptr ss:[ebp-4],0	
00417E24	E8 27FEFFFF	call emotet.417C50	
00417E29	E8 82040000	call emotet.4182B0	
00417E2E	EB 00	jmp emotet.417E30	
00417E30	E8 6B020000	call emotet.4180A0	
00417E35	68 582C0000	push 2C58	
00417E3A	E8 11FFFFFF	call emotet.417D50	
00417E3F	83C4 04	add esp,4	
00417E42	C705 C8C14100 00000000	mov dword ptr ds:[41C1C8],0	
00417E4C	A1 C8C14100	mov eax,dword ptr ds:[41C1C8]	
00417E51	A3 CCC14100	mov dword ptr ds:[41C1CC],eax	
00417E56	C705 C4C14100 02000000	mov dword ptr ds:[41C1C4],2	
00417E60	C745 F4 00000000	mov dword ptr ss:[ebp-c],0	
00417E67	8B0D A8C14100	mov ecx,dword ptr ds:[41C1A8]	
00417E6D	51	push ecx	
00417E6E	8B15 0CC04100	mov edx,dword ptr ds:[41C00C]	edx:"U<ïfï\FwÇEü"
00417E74	52	push edx	edx:"U<ïfï\FwÇEü"
00417E75	E8 B6000000	call emotet.417F30	
00417E7A	83C4 08	add esp,8	
00417E7D	A3 00C24100	mov dword ptr ds:[41C200],eax	
00417E82	A1 C8C14100	mov eax,dword ptr ds:[41C1C8]	
00417E87	3B05 A4C14100	cmp eax,dword ptr ds:[41C1A4]	
00417E8D	72 02	jb emotet.417E91	

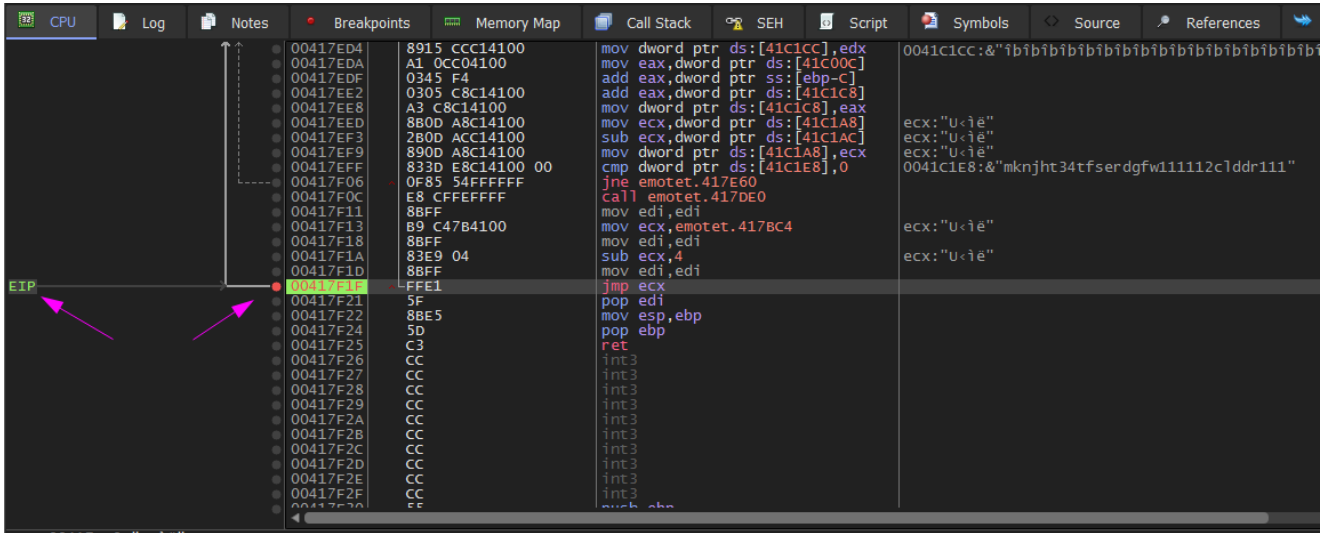
Figure(2):

We set a breakpoint over the jump instruction `jmp ecx` at the address `00417F1F` by pressing `f2`. because after this jump the unpack process will happen.

Address	Hex	Assembly	Comments
00417ED4	8915 CCC14100	mov dword ptr ds:[41C1CC],edx	edx:"U<ïfï\FwÇEü"
00417EDA	A1 0CC04100	mov eax,dword ptr ds:[41C00C]	
00417EDF	0345 F4	add eax,dword ptr ss:[ebp-c]	
00417EE2	0305 C8C14100	add eax,dword ptr ds:[41C1C8]	
00417EE8	A3 C8C14100	mov dword ptr ds:[41C1C8],eax	
00417EED	8B0D A8C14100	mov ecx,dword ptr ds:[41C1A8]	
00417EF3	2B0D ACC14100	sub ecx,dword ptr ds:[41C1AC]	
00417EF9	890D A8C14100	mov dword ptr ds:[41C1A8],ecx	
00417EFF	833D E8C14100 00	cmp dword ptr ds:[41C1E8],0	
00417F06	0F85 54FFFFFF	jne emotet.417E60	
00417F0C	E8 CFFEFFFF	call emotet.417DE0	
00417F11	8BFF	mov edi,edi	
00417F13	B9 C47B4100	mov ecx,emotet.417BC4	
00417F18	8BFF	mov edi,edi	
00417F1A	83E9 04	sub ecx,4	
00417F1D	8BFF	mov edi,edi	
00417F1F	FFE1	jmp ecx	
00417F21	5F	pop edi	
00417F22	8BE5	mov esp,ebp	
00417F24	5D	pop ebp	
00417F25	C3	ret	
00417F26	CC	int3	
00417F27	CC	int3	
00417F28	CC	int3	
00417F29	CC	int3	
00417F2A	CC	int3	
00417F2B	CC	int3	
00417F2C	CC	int3	
00417F2D	CC	int3	
00417F2E	CC	int3	
00417F2F	CC	int3	

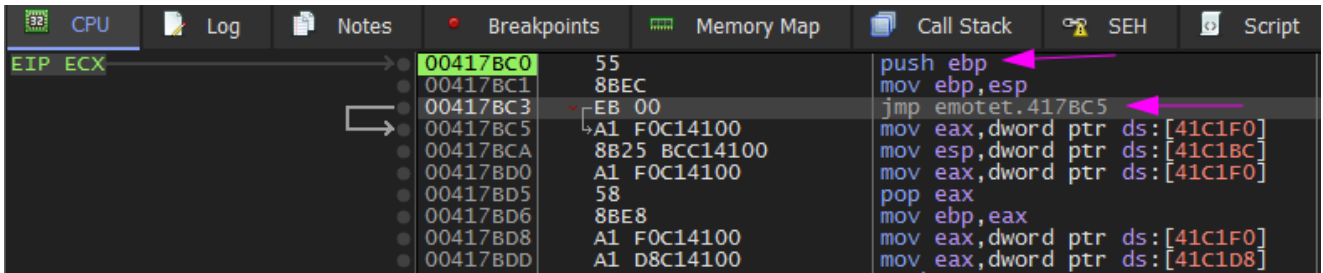
Figure(3):

Then we press `f9` to hit the breakpoint.



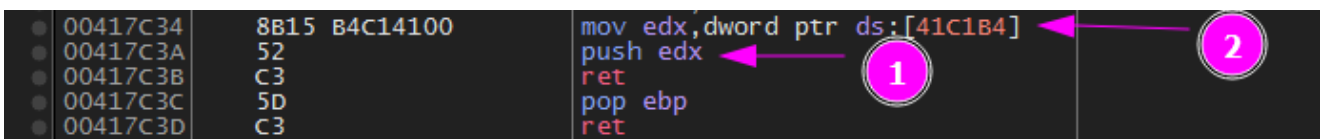
Figure(4):

Then press `f7` we jump to another function. If we analyze this function we will notice that: epilogue



Figure(5):

Abnormal prologue



Figure(6):

The last figure shows the abnormal prologue (1). And (2) is a suspicious instruction which we will know late that is new `VirtualAlloc`.

So we set a breakpoint over this instruction `mov edx,dword ptr ds:[41C1B4]` by `f2` and press `f9` to hit the breakpoint.

	00417C30	8BD2	mov edx,edx
	00417C32	8BD2	mov edx,edx
EIP	00417C34	8B15 B4C14100	mov edx,dword ptr ds:[41C1B4]
	00417C3A	52	push edx
	00417C3B	C3	ret
	00417C3C	5D	pop ebp
	00417C3D	C3	ret

Figure(7):

Now if we follow in dump We see that it's allocating memory.

Then Press f8 it will push edx to the stack which is the value of mov edx,dword ptr ds:[41C1B4] .

Then Press f8 . There is abnormal ret . Normal ret value will get back to wherever it was called from.

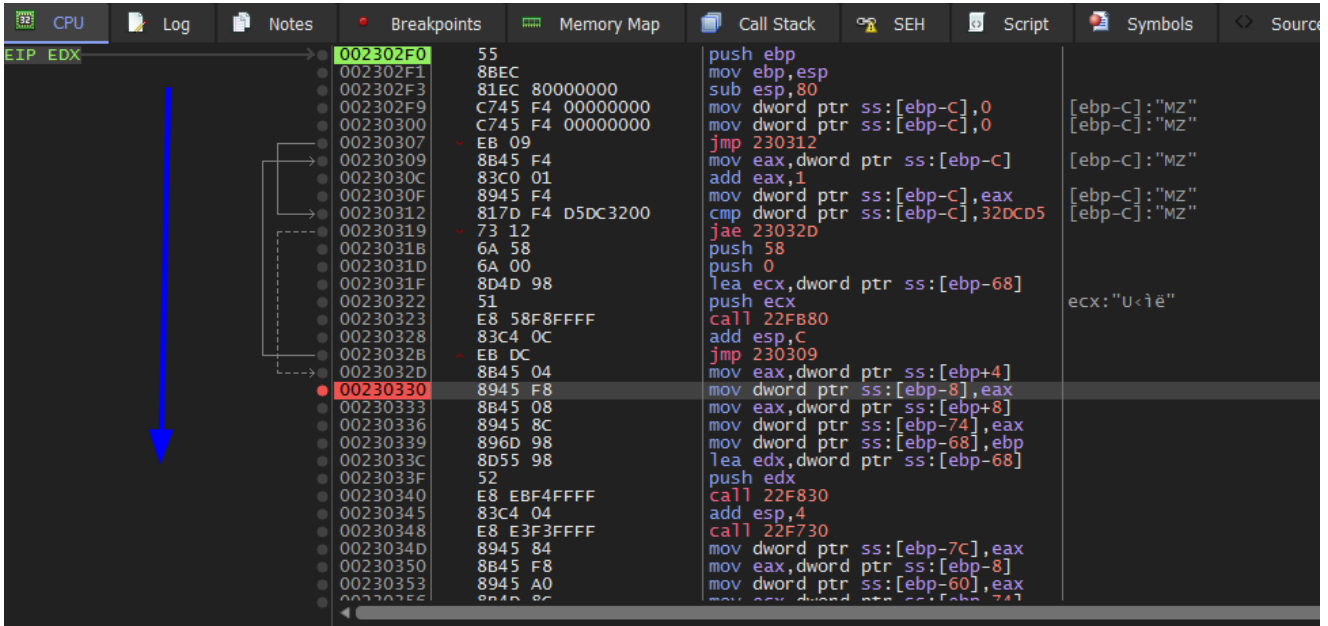
Here it return to this address 002302F0 . Which will be the address of the unpacking section.

0018FF80	002302F0	
0018FF84	00000000	
0018FF88	00400000	emotet.00400000
0018FF8C	75C2343D	return to kernel!32.75C2343D from ???
0018FF90	7EFDE000	
0018FF94	0018FFD4	
0018FF98	778F9832	return to ntdll.778F9832 from ???
0018FF9C	7EFDE000	
0018FFA0	77096289	rpcrt4.77096289
0018FFA4	00000000	
0018FFA8	00000000	
0018FFAC	7EFDE000	

Figure(9):

So step over it.

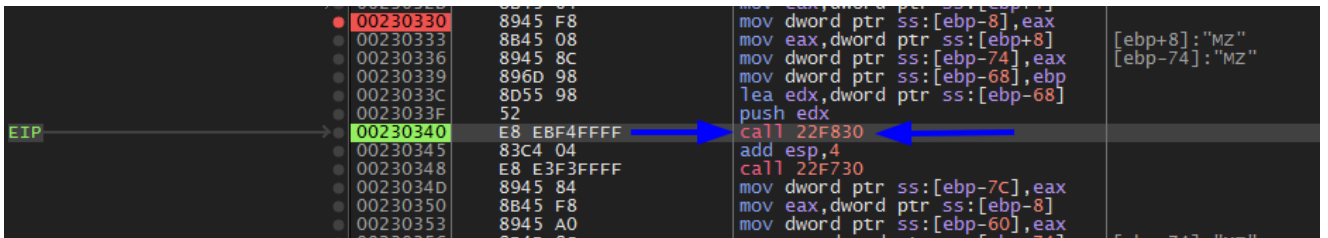
In the next part we will see functions (Unpacking routine) and we will explain it on the fly in the next figure



Figure(10):

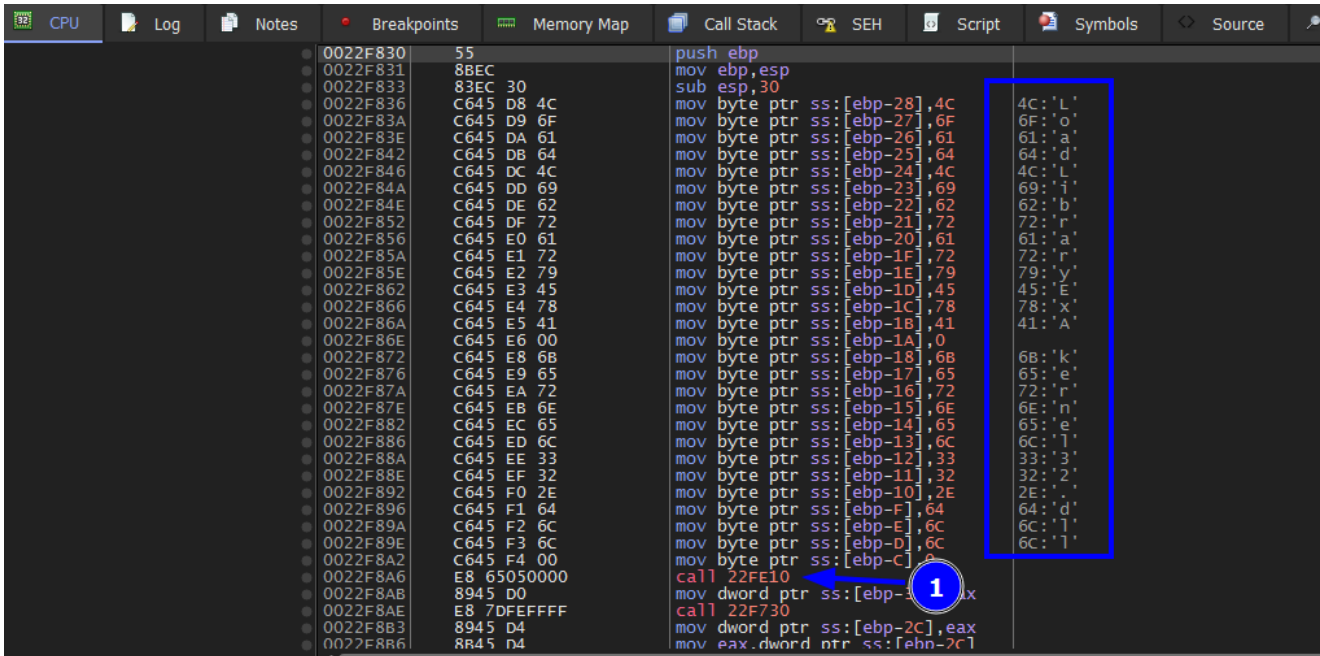
Keep stepping over until you reach the breakpoint.

Then we see this function and step into `f7`.



Figure(11):

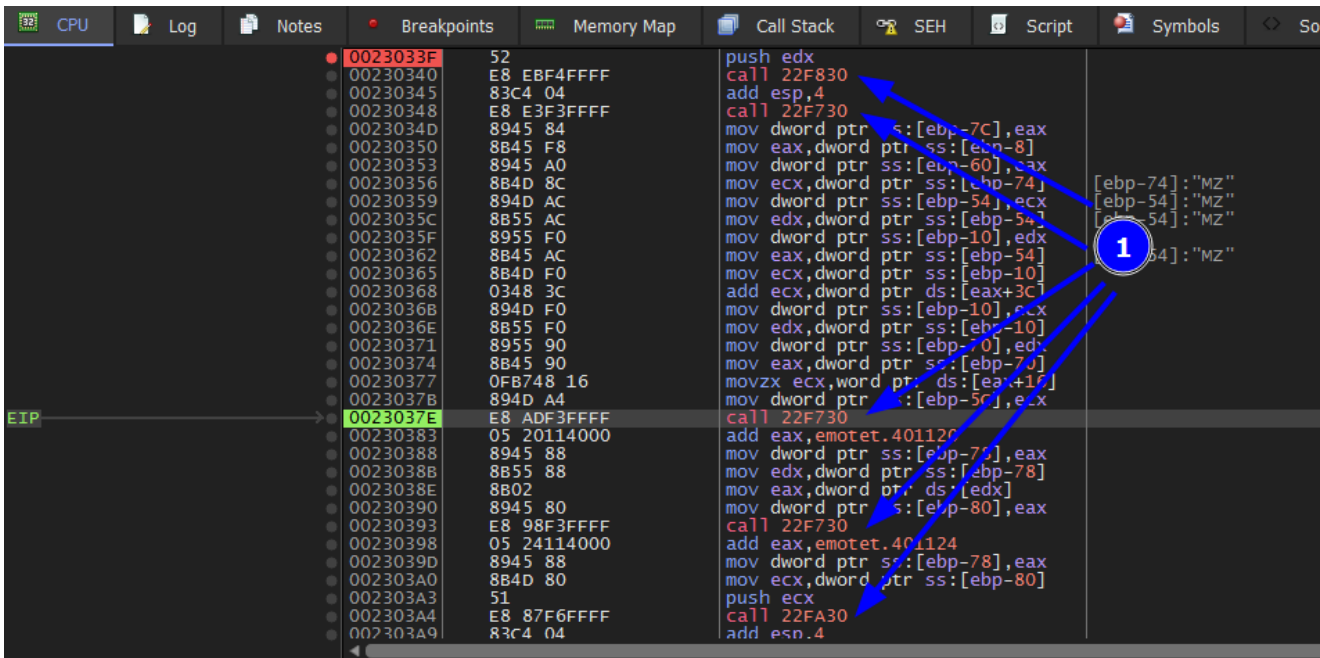
It uses stack strings. which is mentioned above In introduction. (1) pushes them on the stack.



Figure(12):

To get out from this function find `ret` and set a breakpoint then press `f9`

And these functions do the same as above. So step over them `f8` . to see what inside a function without executing it: Double click over a function and press `-` button to get out.



Figure(13):

Until we get to this last function. step into `f7` .

	002303D0	83C4 08	add esp,8	
	002303D3	8B4D 94	mov ecx,dword ptr ss:[ebp-6C]	[ebp-6C]: "MZ"
	002303D6	51	push ecx	ecx: "MZ"
	002303D7	8D55 98	lea edx,dword ptr ss:[ebp-68]	
	002303DA	52	push edx	
EIP	002303DB	E8 A0FBFFFF	call 22FF80	
	002303E0	85C0	test eax,eax	
	002303E2	75 04	jne 2303E8	
	002303E4	33C0	xor eax,eax	
	002303E6	EB 1A	jmp 230402	
	002303E8	837D A0 00	cmp dword ptr ss:[ebp-60],0	
	002303EC	74 09	je 2303F7	

Figure(14):

After we get into the function we need to analyze it **carefully**

	002303D0	83C4 08	add esp,8	
	002303D3	8B4D 94	mov ecx,dword ptr ss:[ebp-6C]	[ebp-6C]: "MZ"
	002303D6	51	push ecx	ecx: "MZ"
	002303D7	8D55 98	lea edx,dword ptr ss:[ebp-68]	
	002303DA	52	push edx	
EIP	002303DB	E8 A0FBFFFF	call 22FF80	
	002303E0	85C0	test eax,eax	
	002303E2	75 04	jne 2303E8	
	002303E4	33C0	xor eax,eax	
	002303E6	EB 1A	jmp 230402	
	002303E8	837D A0 00	cmp dword ptr ss:[ebp-60],0	
	002303EC	74 09	je 2303F7	

Figure(14):

As we can see `call edx` is calling `VirtualAlloc` :

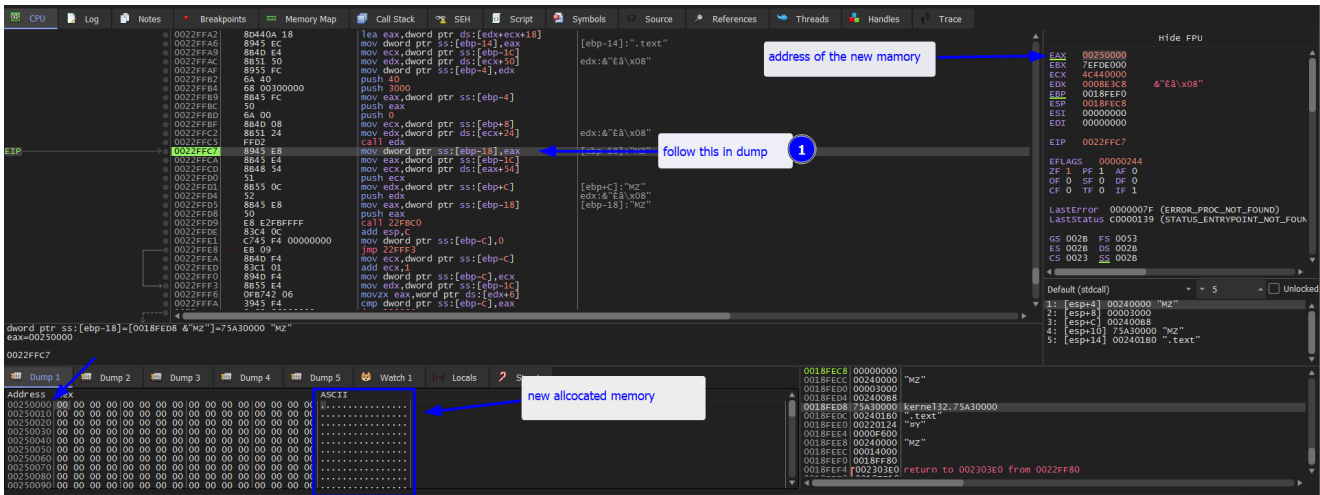
`push 40` `RWX` which is our indicator to know that this call could be `VirtualAlloc`

	0022FF80	55	push ebp	
	0022FF81	8BEC	mov ebp,esp	
	0022FF83	83EC 28	sub esp,28	
	0022FF86	8B45 0C	mov eax,dword ptr ss:[ebp+C]	[ebp+C]: "MZ"
	0022FF89	8945 DC	mov dword ptr ss:[ebp-24],eax	[ebp-24]: "MZ"
	0022FF8C	8B4D DC	mov ecx,dword ptr ss:[ebp-24]	[ebp-24]: "MZ"
	0022FF8F	8B55 0C	mov edx,dword ptr ss:[ebp+C]	[ebp+C]: "MZ"
	0022FF92	0351 3C	add edx,dword ptr ds:[ecx+3C]	
	0022FF95	8955 E4	mov dword ptr ss:[ebp-1C],edx	
	0022FF98	8B45 E4	mov eax,dword ptr ss:[ebp-1C]	
	0022FF9B	0FB748 14	movzx ecx,word ptr ds:[eax+14]	
	0022FF9F	8B55 E4	mov edx,dword ptr ss:[ebp-1C]	
	0022FFA2	8D440A 18	lea eax,dword ptr ds:[edx+ecx+18]	
	0022FFA6	8945 EC	lea dword ptr ss:[ebp-14],eax	[ebp-14]: ".text"
	0022FFA9	8B4D E4	mov ecx,dword ptr ss:[ebp-1C]	
	0022FFAC	8B51 50	mov edx,dword ptr ds:[ecx+50]	ecx+50: "A{u}]#uB"
	0022FFAF	8955 FC	mov dword ptr ss:[ebp-4],edx	
	0022FFB2	6A 40	push 40	
	0022FFB4	68 00300000	push 3000	
	0022FFB8	8B45 FC	mov eax,dword ptr ss:[ebp-4]	
	0022FFBC	50	push eax	
	0022FFBD	6A 00	push 0	
	0022FFBF	8B4D 08	mov ecx,dword ptr ss:[ebp+8]	
	0022FFC2	8B51 24	mov edx,dword ptr ds:[ecx+24]	
EIP	0022FFC5	FFD2	call edx	
	0022FFC7	8945 E8	mov dword ptr ss:[ebp-18],eax	[ebp-18]: "MZ"
	0022FFCA	8B45 E4	mov eax,dword ptr ss:[ebp-1C]	
	0022FFCD	8B48 54	mov ecx,dword ptr ds:[eax+54]	
	0022FFD0	51	push ecx	
	0022FFD1	8B55 0C	mov edx,dword ptr ss:[ebp+C]	[ebp+C]: "MZ"
	0022FFD4	52	push edx	

edx=<kerne132.VirtualAlloc>

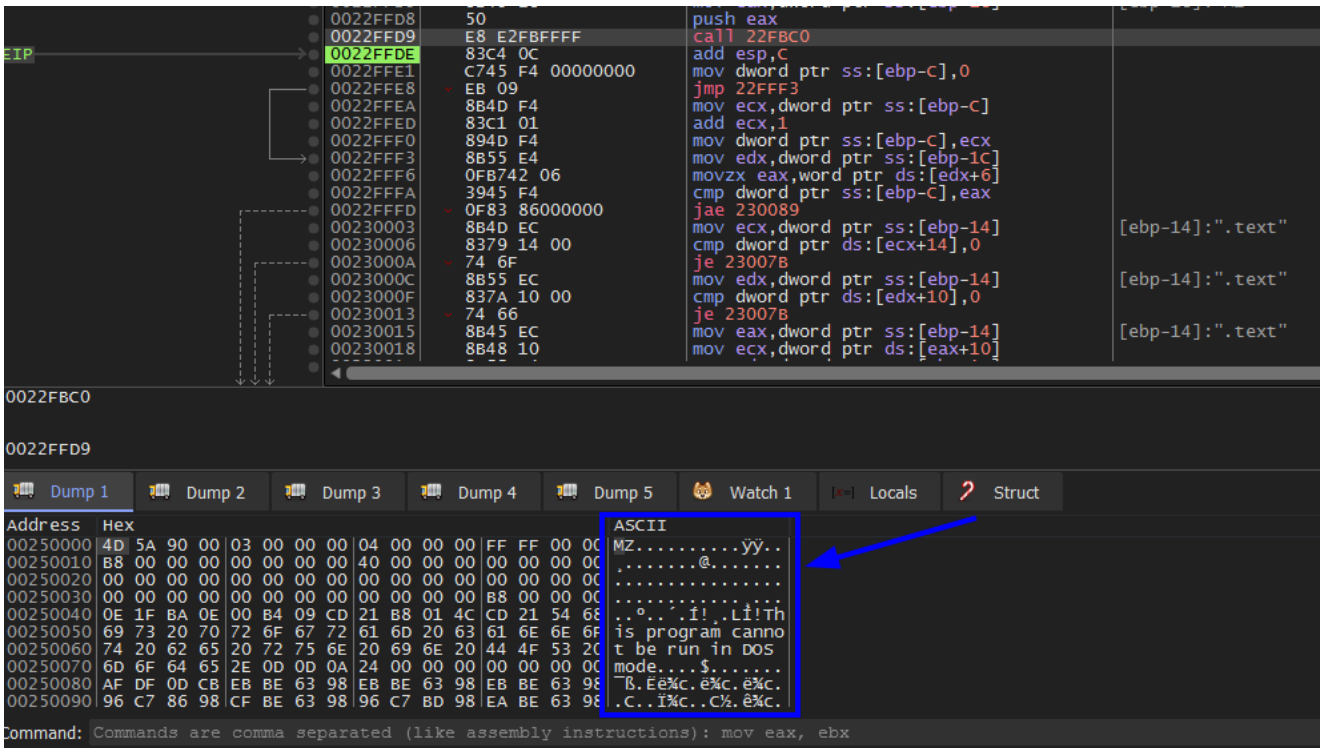
Figure(15):

One step over `f8` and we will get the address of newly memory allocated in `eax`



Figure(16):

Then keep stepping over and get to this function `call 22FBC0` and then one more step over. As we see in the dump section, the function writes over the newly memory allocate with the exe file.



Figure(17):

When keep stepping we see that it's copying files to the exe file

`.text`

0022FFD9	E8 E2FBFFFF	call 22FB00	
0022FFDE	83C4 0C	add esp,c	
0022FFE1	C745 F4 00000000	mov dword ptr ss:[ebp-c],0	
0022FFE8	EB 09	jmp 22FFF3	
0022FFEA	8B4D F4	mov ecx,dword ptr ss:[ebp-c]	ecx:"MZ"
0022FFED	83C1 01	add ecx,1	
0022FFF0	894D F4	mov dword ptr ss:[ebp-c],ecx	
0022FFF3	8B55 E4	mov edx,dword ptr ss:[ebp-1c]	
0022FFF6	0FB742 06	movzx eax,word ptr ds:[edx+6]	eax:".text"
0022FFFA	3945 F4	cmp dword ptr ss:[ebp-c],eax	
0022FFFD	0F83 86000000	jae 23007B	
00230003	8B4D EC	mov ecx,dword ptr ss:[ebp-14]	[ebp-14]:".text"
00230006	8379 14 00	cmp dword ptr ds:[ecx+14],0	
0023000A	74 6F	je 23007B	
0023000C	8B55 EC	mov ecx,dword ptr ss:[ebp-14]	[ebp-14]:".text"
0023000F	837A 10 00	cmp dword ptr ds:[edx+10],0	
00230013	74 66	je 23007B	
00230015	8B45 EC	mov eax,dword ptr ss:[ebp-14]	[ebp-14]:".text"
00230018	8B48 10	mov ecx,dword ptr ds:[eax+10]	ecx:"MZ"
0023001B	8B55 E4	mov edx,dword ptr ss:[ebp-1c]	
0023001E	8B42 3C	mov eax,dword ptr ds:[edx+3c]	eax:".text"
00230021	8D4401 FF	lea eax,dword ptr ds:[ecx+eax-1]	eax:".text"
00230025	8B4D E4	mov ecx,dword ptr ss:[ebp-1c]	
00230028	33D2	xor edx,edx	
0023002A	F771 3C	div dword ptr ds:[ecx+3c]	
0023002D	8B55 E4	mov edx,dword ptr ss:[ebp-1c]	
00230030	0FAF42 3C	imul eax,dword ptr ds:[edx+3c]	eax:".text"
00230034	50	push eax	eax:".text"
00230035	8B45 EC	mov eax,dword ptr ss:[ebp-14]	[ebp-14]:".text"
00230038	8B4D 0C	mov ecx,dword ptr ss:[ebp-c]	[ebp+c]:".MZ"
0023003B	0348 14	add ecx,dword ptr ds:[eax+14]	ecx:"MZ"

Figure(18):

then `.rdata` then `.data` then `.reloc`

Until we get to the last `ret 8` as shown.

002301DF	E8 7CF9FFFF	call 22FB60	
002301E4	83C4 04	add esp,4	
002301E7	B8 01000000	mov eax,1	
002301EC	8B55	mov esp,ebp	
002301EE	5D	pop ebp	
002301EF	C2 0800	ret 8	
002301F2	CC	int3	
002301F3	CC	int3	
002301F4	CC	int3	
002301F5	CC	int3	

Figure(19):

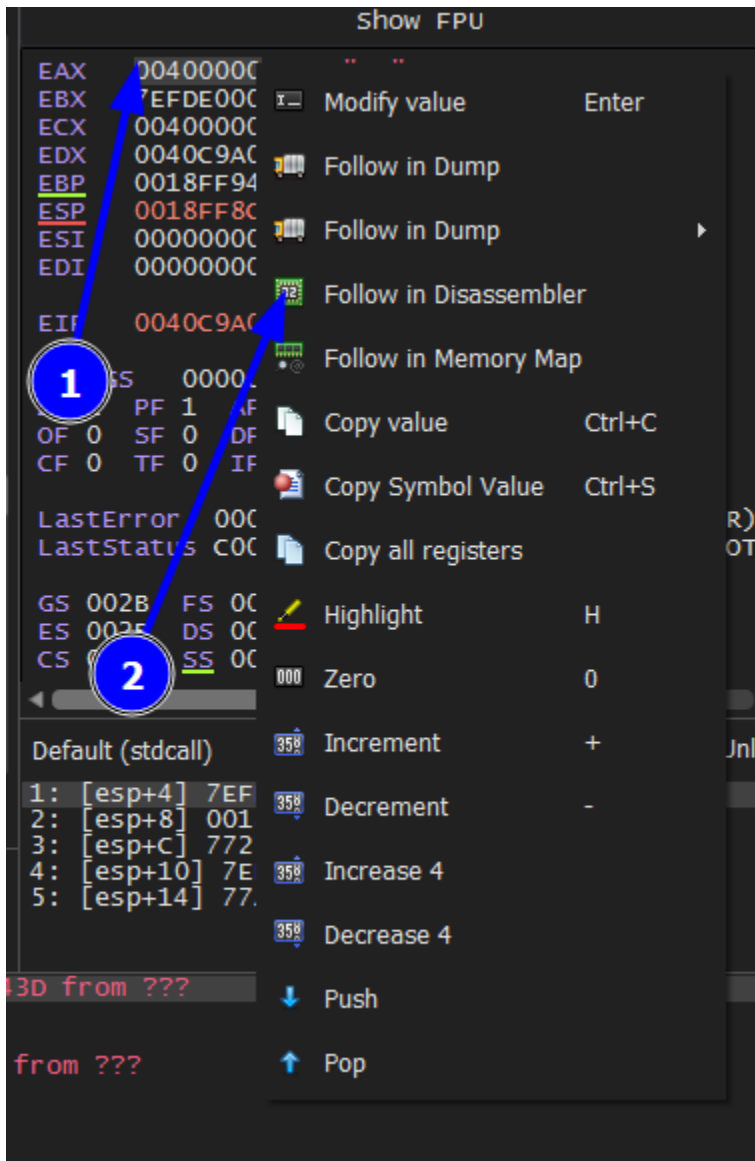
Stay awake our file is almost finished. After the second `ret .`

The screenshot shows the Immunity Debugger interface. The main window displays assembly code for a function. The call stack window is open, showing the current function and its callers. A blue box highlights the OEP address (0018FF94) in the call stack. A blue arrow points to the OEP address in the call stack. The memory dump window at the bottom shows the contents of the memory at the OEP address, which is the ASCII string "MZ".

Figure(20):

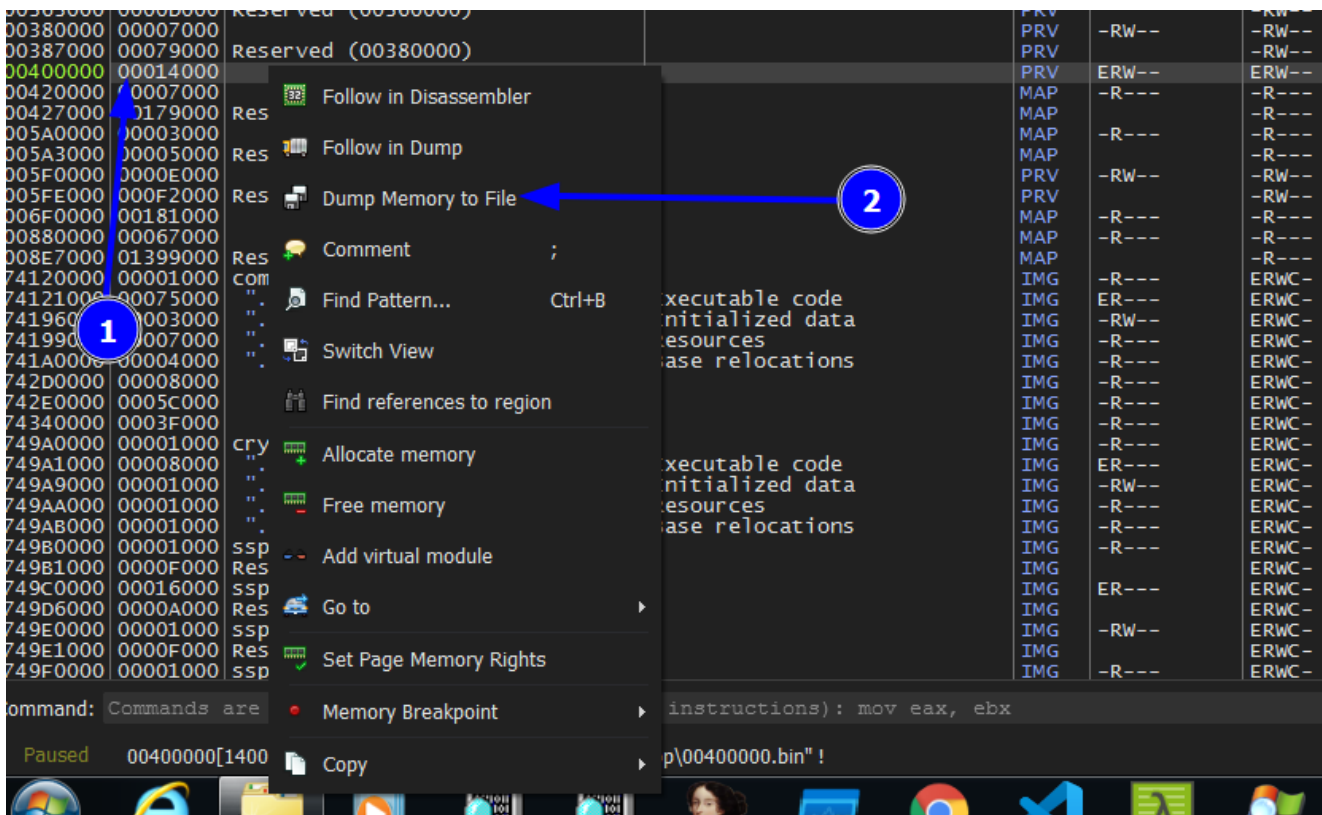
Now we can dump the unpacked exe. **right click** over **eax** and press **Follow in Mwmory map**

Sorry for this Mistake in the next figure. It's **Follow in Mwmory map**



Figure(21):

Then **right click** and then press **Dump memory to File**



Figure(22):

Now if we tried to open it in IDA. We will notice that's can't be analyzed


```
public start
start proc near
push    esi
lahf
popf
mov     dword ptr [ebp-4DCh], 587279C7h
mov     dword ptr [ebp-4D8h], 0ED7EE61Fh
mov     dword ptr [ebp-4D4h], 0AD05AF9Ch
mov     dword ptr [ebp-4D0h], 0B2015FBCh
mov     dword ptr [ebp-4CCh], 645D1A4Eh
mov     dword ptr [ebp-4C8h], 2E4F310Dh
mov     dword ptr [ebp-4C4h], 5A11C3CDh
mov     dword ptr [ebp-4C0h], 0A4FF6433h
mov     dword ptr [ebp-4BCh], 8BF1A0E2h
mov     dword ptr [ebp-4B8h], 0A69D0C0Fh
mov     dword ptr [ebp-4B4h], 0BC5118DCh
mov     dword ptr [ebp-4B0h], 0E7B92C0Ch
mov     dword ptr [ebp-4ACh], 0AD9CB0F4h
mov     dword ptr [ebp-4A8h], 8CD776B8h
mov     dword ptr [ebp-4A4h], 0B60EE1B9h
mov     dword ptr [ebp-4A0h], 6682D4AEh
mov     dword ptr [ebp-49Ch], 7CA422F3h
mov     dword ptr [ebp-498h], 0A37A83F0h
mov     dword ptr [ebp-494h], 647CC756h
mov     dword ptr [ebp-490h], 0E5155930h
mov     dword ptr [ebp-48Ch], 4E97325Fh
mov     dword ptr [ebp-488h], 0D2A17ECh
mov     dword ptr [ebp-484h], 0D1FFE464h
mov     dword ptr [ebp-480h], 710B8734h
mov     dword ptr [ebp-47Ch], 0F493AA9Fh
```

with Hex View-1)

Figure(23):

So we need to repair section headers using `PE bear` tool.

Before

Disasm: .text		General	DOS Hdr	Rich Hdr	File Hdr	Optional Hdr	Section Hdrs	BaseReloc.
Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics	Ptr to Reloc.	Num. of Reloc.	Num. of Linenur
▷ .text	400	CE00	1000	CC24	60000020	0	0	0
▷ .rdata	D200	C00	E000	B00	40000040	0	0	0
▷ .data	DE00	1200	F000	3DE4	C0000040	0	0	0
▷ .reloc	F000	600	13000	58C	42000040	0	0	0

Figure(24):

After editing

Disasm: .text		General	DOS Hdr	Rich Hdr	File Hdr	Optional Hdr	Section Hdrs	BaseReloc.
Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics	Ptr to Reloc.	Num. of Reloc.	Num. of Linen
▷ .text	1000	D000	1000	D000	60000020	0	0	0
▷ .rdata	E000	1000	E000	1000	40000040	0	0	0
▷ .data	F000	4000	F000	4000	C0000040	0	0	0
▷ .reloc	13000	600	13000	600	42000040	0	0	0

Figure(25):

Then change the image base: if it's different value of the OEP.

Disasm: .text		General	DOS Hdr	Rich Hdr	File Hdr	Optional Hdr	Section Hdrs	BaseReloc.
Offset	Name	Value	Value					
D0	Magic	10B	NT32					
D2	Linker Ver. (Major)	C						
D3	Linker Ver. (Minor)	0						
D4	Size of Code	CE00						
D8	Size of Initialized Data	5000						
DC	Size of Uninitialized Data	0						
E0	Entry Point	C9A0						
E4	Base of Code	1000						
E8	Base of Data	E000						
EC	Image Base	400000						
F0	Section Alignment	1000						
F4	File Alignment	200						
F8	OS Ver. (Major)	6	Windows Vista / Server 2008					
FA	OS Ver. (Minor)	0						
FC	Image Ver. (Major)	0						
FE	Image Ver. (Minor)	0						
100	Subsystem Ver. (Major)	6						
102	Subsystem Ver. Minor)	0						
104	Win32 Version Value	0						
108	Size of Image	14000						
10C	Size of Headers	400						

Figure(26):

Unmap the unpacked file

How we edit the section headers? ordered steps.

first: copy `Virtual address` values into `Raw address` values.

second: `Raw size` Raw size of `.text` = Raw adress of `.rdata` - Raw adress of `.text`

$$E000 - 1000 = D000$$

Raw size of `.rdata` = Raw adress of `.data` - Raw adress of `.rdata`

$$F000 - E000 = 1000$$

Raw size of `.data` = Raw adress of `.reloc` - Raw adress of `.data`

$$13000 - F000 = 4000$$

Raw size of `.reloc` = still the same third: copy `Raw size` values into `Virtual size` values.

After changing save the file. This is our unpacked malware

See you in the next article. inshAllah

Article quote

على الضفة الأخرى لن نخشى الغرق |

Refernces

1- <https://www.mcafee.com/blogs/enterprise/malware-packers-use-tricks-avoid-analysis-detection/>

2-<https://www.oreilly.com/library/view/learning-malware-analysis/9781788392501/12556df2-7825-4e43-8811-c0fabeab78d8.xhtml>

3- <https://www.0xbyte.com/unpacking-mzsp-ransomware-manually/>

4- <https://isc.sans.edu/forums/diary/Stackstrings+type+2/26192/#:~:text=This%20is%20a%20technique%20that,the%20allocated%20chunk%20of%20memory>

5- [https://msdn.microsoft.com/en-us/library/windows/desktop/aa366887\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/aa366887(v=vs.85).aspx)