Unpacking Emotet malware part 01

muha2xmad.github.io/unpacking/emotet-part-1/

January 6, 2022

Muhammad Hasan Ali

Malware Analysis learner

3 minute read

As-salamu Alaykum

Introduction

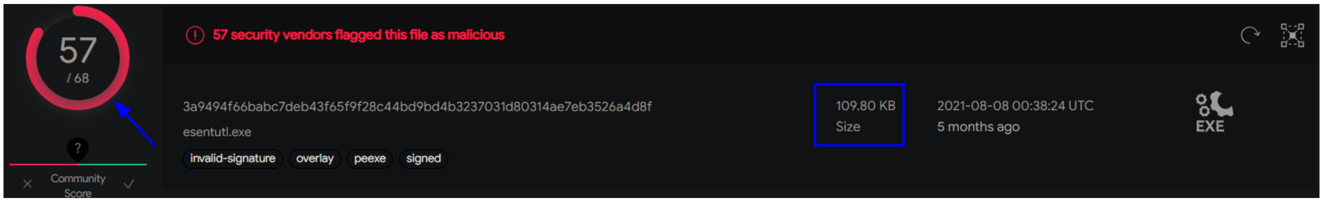
Emotet is a Trojan that spreads through spam emails. The infection may arrive either via malicious script, macro-enabled document files, or malicious link. 1

Download the sample: Here

MD5: `CA06ACD3E1CAB1691A7670A5F23BAEF4`

Virustotal VT

we can see that the malware is detected by 57 out of 68 as a trojan.



57 / 68

57 security vendors flagged this file as malicious

3a9494f66babc7deb43f65f9f28c44bd9bd4b3237031d80314ae7eb3526a4d8f

esentutl.exe

invalid-signature overlay peexe signed

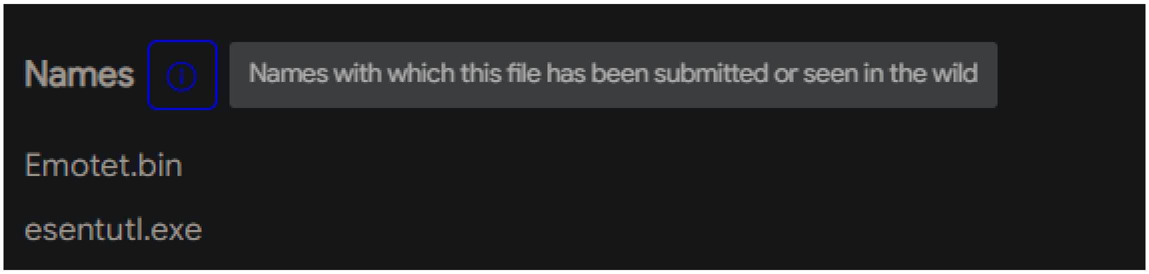
109.80 KB Size

2021-08-08 00:38:24 UTC
5 months ago

Figure(1):

In Details section VT Details

1- Different names of the sample



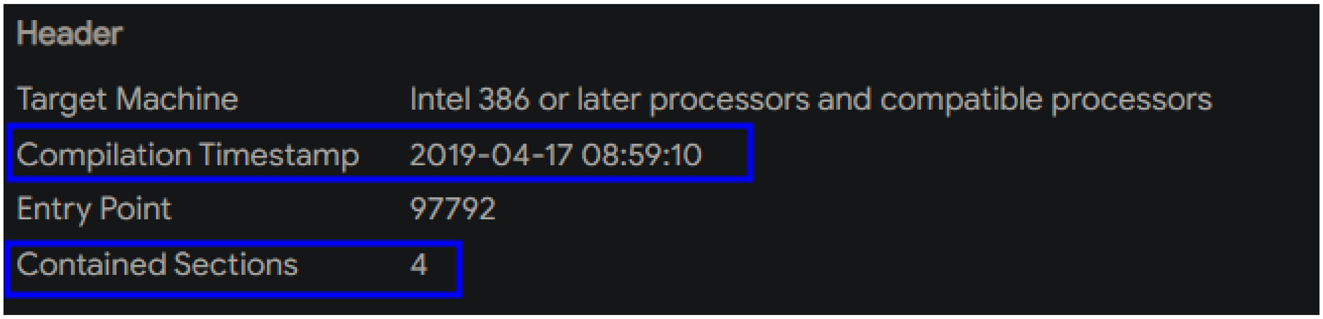
Names

Names with which this file has been submitted or seen in the wild

Emotet.bin

esentutl.exe

Figure(2):
2- Header info



Header

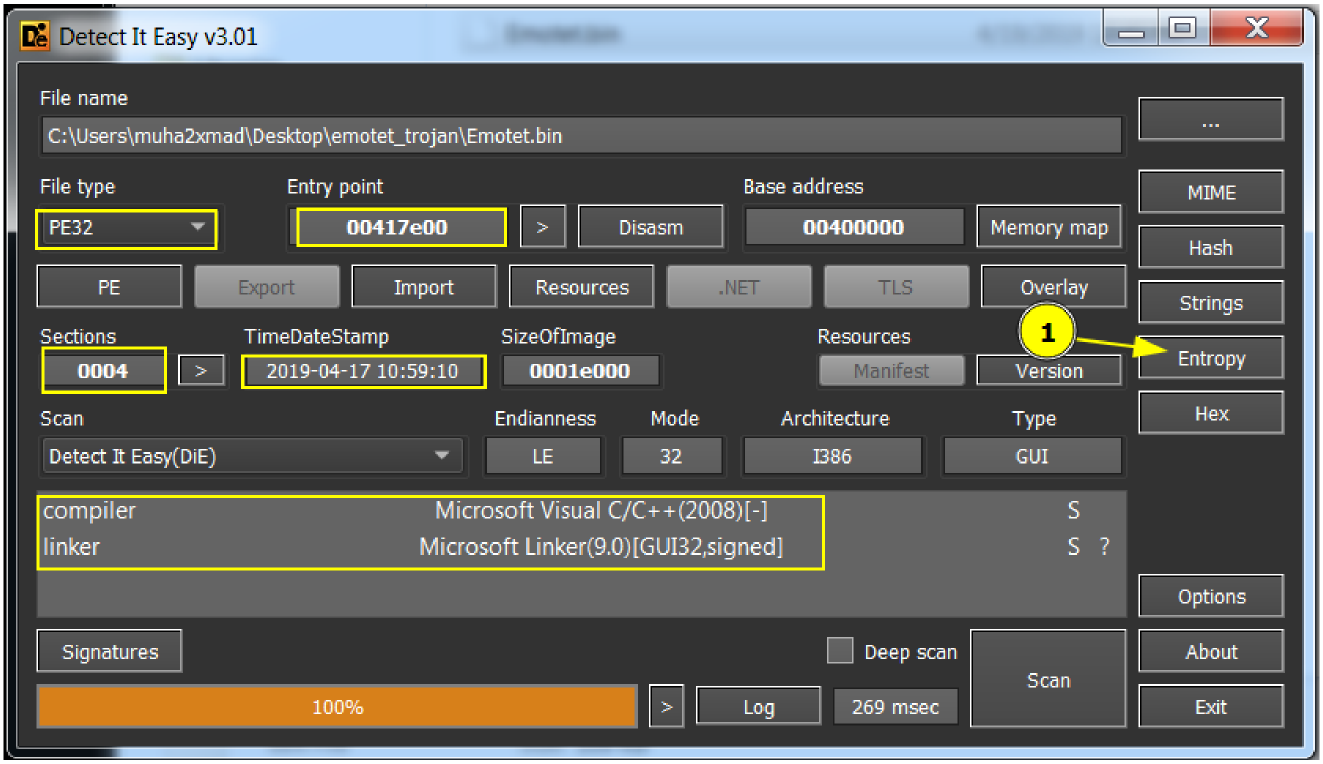
Target Machine	Intel 386 or later processors and compatible processors
Compilation Timestamp	2019-04-17 08:59:10
Entry Point	97792
Contained Sections	4

Figure(3):

Shows compilation Timestamp which can be changed. and Shows number of sections

DiE

open DiE to get more info about the sample



Detect It Easy v3.01

File name: C:\Users\muha2xmad\Desktop\emotet_trojan\Emotet.bin

File type: PE32

Entry point: 00417e00

Base address: 00400000

Sections: 0004

TimeDateStamp: 2019-04-17 10:59:10

SizeOfImage: 0001e000

Scan: Detect It Easy(DiE)

Endianness: LE

Mode: 32

Architecture: I386

Type: GUI

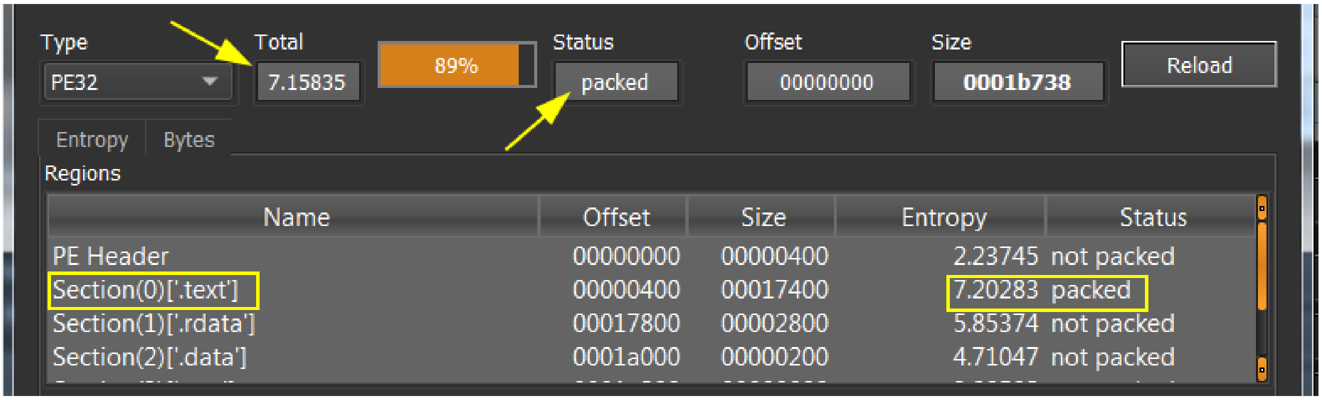
```
compiler    Microsoft Visual C/C++(2008)[-]        S
linker      Microsoft Linker(9.0)[GUI32,signed]    S ?
```

Figure(4):

As we see that info about **file type**, **Entry point**, and **sections**. It will help us in our analysis

Entropy:

press over “Entropy” as in the previous figure(4)



Type: PE32

Total: 7.15835

89%

Status: packed

Offset: 00000000

Size: 0001b738

Regions

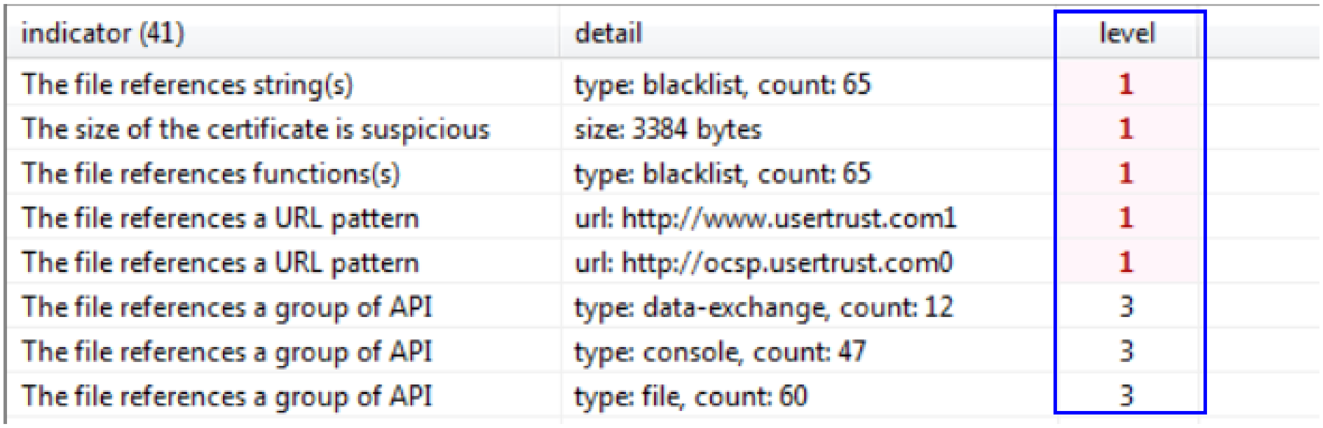
Name	Offset	Size	Entropy	Status
PE Header	00000000	00000400	2.23745	not packed
Section(0)['.text']	00000400	00017400	7.20283	packed
Section(1)['.rdata']	00017800	00002800	5.85374	not packed
Section(2)['.data']	0001a000	00000200	4.71047	not packed

Figure(5):

Shows that it has **high** entropy in **.text** section which is an indicator to be packed

PEstudio analysis

Indicators section:

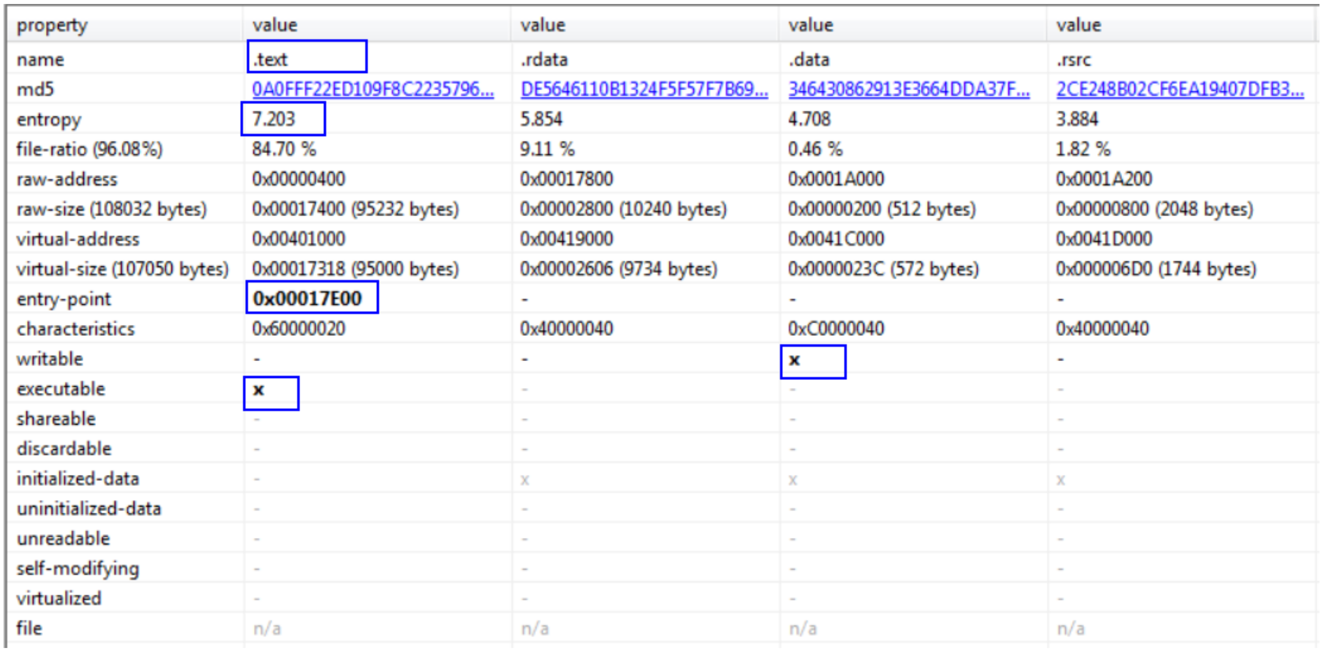


indicator (41)	detail	level
The file references string(s)	type: blacklist, count: 65	1
The size of the certificate is suspicious	size: 3384 bytes	1
The file references functions(s)	type: blacklist, count: 65	1
The file references a URL pattern	url: http://www.usertrust.com1	1
The file references a URL pattern	url: http://ocsp.usertrust.com0	1
The file references a group of API	type: data-exchange, count: 12	3
The file references a group of API	type: console, count: 47	3
The file references a group of API	type: file, count: 60	3

Figure(6):

*Level 1 is most malicious and bigger numbers “3” are less malicious. Shows different malicious indicators that help us in the analysis.

Sections section:



property	value	value	value	value
name	.text	.rdata	.data	.rsrc
md5	0A0FFF22ED109F8C2235796...	DE5646110B1324F5F57F7B69...	346430862913E3664DDA37F...	2CE248B02CF6EA19407DFB3...
entropy	7.203	5.854	4.708	3.884
file-ratio (96.08%)	84.70 %	9.11 %	0.46 %	1.82 %
raw-address	0x00000400	0x00017800	0x0001A000	0x0001A200
raw-size (108032 bytes)	0x00017400 (95232 bytes)	0x00002800 (10240 bytes)	0x00000200 (512 bytes)	0x00000800 (2048 bytes)
virtual-address	0x00401000	0x00419000	0x0041C000	0x0041D000
virtual-size (107050 bytes)	0x00017318 (95000 bytes)	0x00002606 (9734 bytes)	0x0000023C (572 bytes)	0x000006D0 (1744 bytes)
entry-point	**0x00017E00**	-	-	-
characteristics	0x60000020	0x40000040	0xC0000040	0x40000040
writable	-	-	x	-
executable	x	-	-	-
shareable	-	-	-	-
discardable	-	-	-	-
initialized-data	-	x	x	x
uninitialized-data	-	-	-	-
unreadable	-	-	-	-
self-modifying	-	-	-	-
virtualized	-	-	-	-
file	n/a	n/a	n/a	n/a

Figure(7):

The previous figure shows:

1- .text section is packed

2- .text section contains the entry point for the executable. This means that, in addition to holding the compressed data, .text section also contains the stub code responsible for unpacking. 2

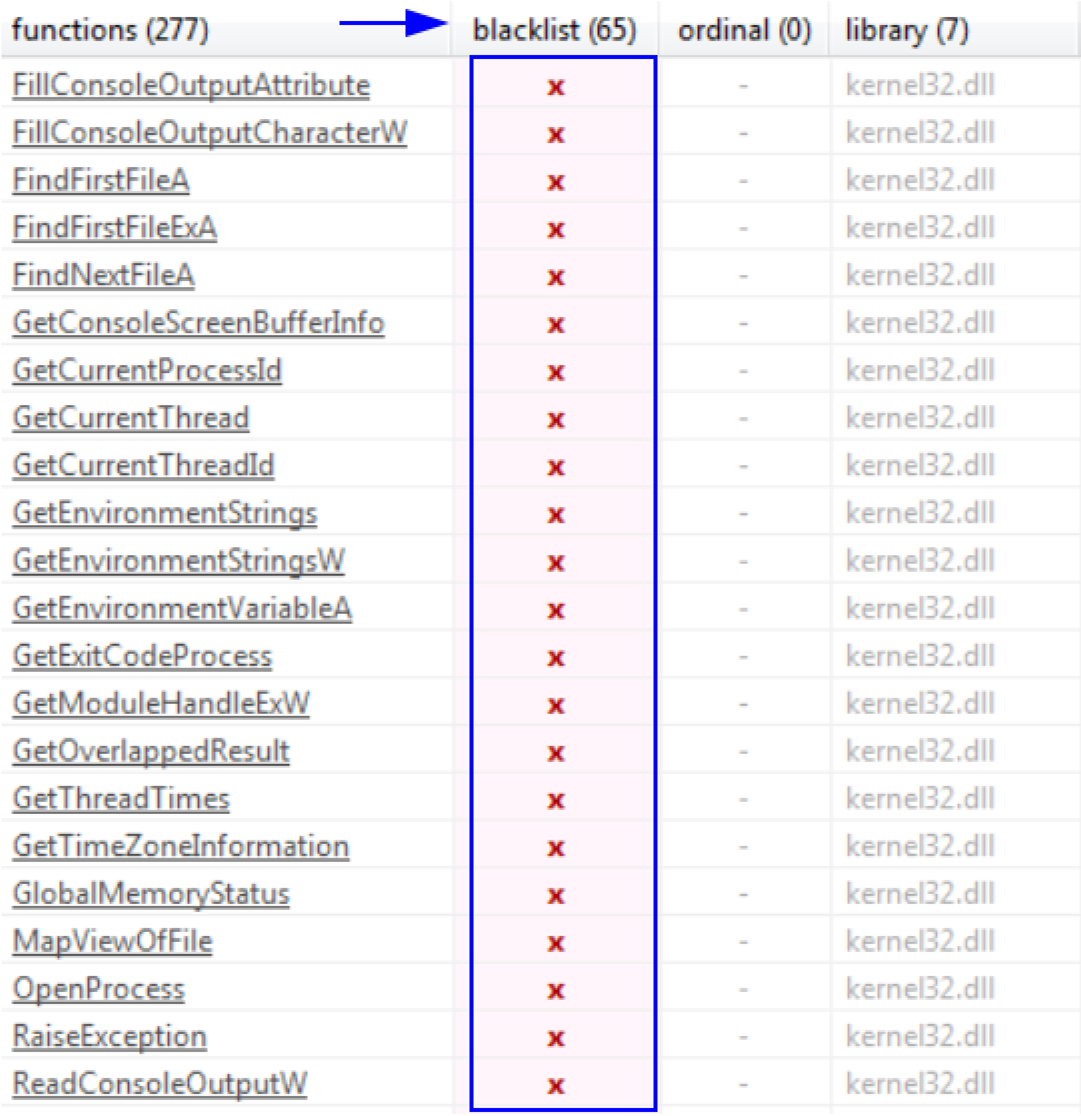
*The section which is responsible for unpacking can vary as in UPX packing

3- .text section is executable

4- .data section is writable

Strings section:

press over `blacklist` to list them



functions (277)	blacklist (65)	ordinal (0)	library (7)
FillConsoleOutputAttribute	x	-	kernel32.dll
FillConsoleOutputCharacterW	x	-	kernel32.dll
FindFirstFileA	x	-	kernel32.dll
FindFirstFileExA	x	-	kernel32.dll
FindNextFileA	x	-	kernel32.dll
GetConsoleScreenBufferInfo	x	-	kernel32.dll
GetCurrentProcessId	x	-	kernel32.dll
GetCurrentThread	x	-	kernel32.dll
GetCurrentThreadId	x	-	kernel32.dll
GetEnvironmentStrings	x	-	kernel32.dll
GetEnvironmentStringsW	x	-	kernel32.dll
GetEnvironmentVariableA	x	-	kernel32.dll
GetExitCodeProcess	x	-	kernel32.dll
GetModuleHandleExW	x	-	kernel32.dll
GetOverlappedResult	x	-	kernel32.dll
GetThreadTimes	x	-	kernel32.dll
GetTimeZoneInformation	x	-	kernel32.dll
GlobalMemoryStatus	x	-	kernel32.dll
MapViewOfFile	x	-	kernel32.dll
OpenProcess	x	-	kernel32.dll
RaiseException	x	-	kernel32.dll
ReadConsoleOutputW	x	-	kernel32.dll

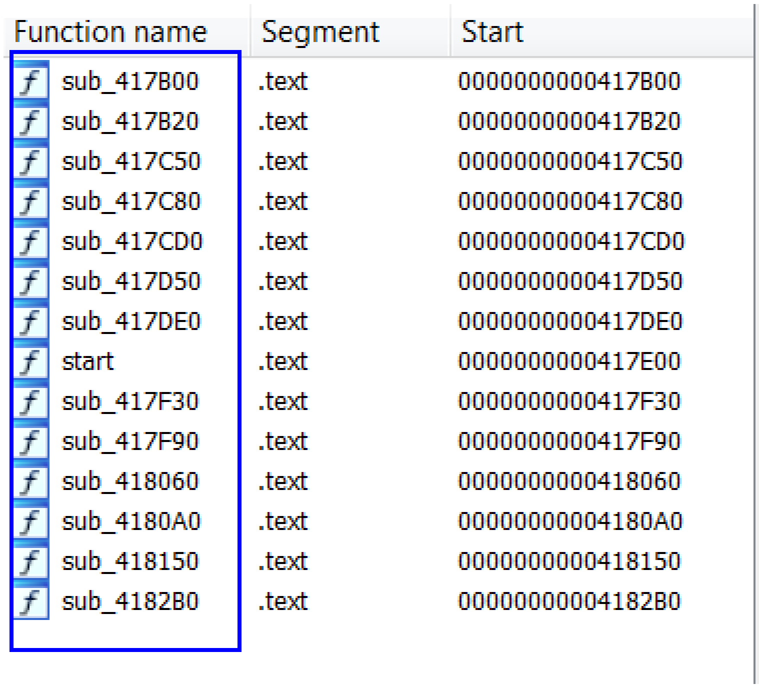
Figure(8):

Strings are good indicators to know what this malware is trying to do on the system

IDA analysis

To analyze the assemble code to know how to unpack and where to start the debugging

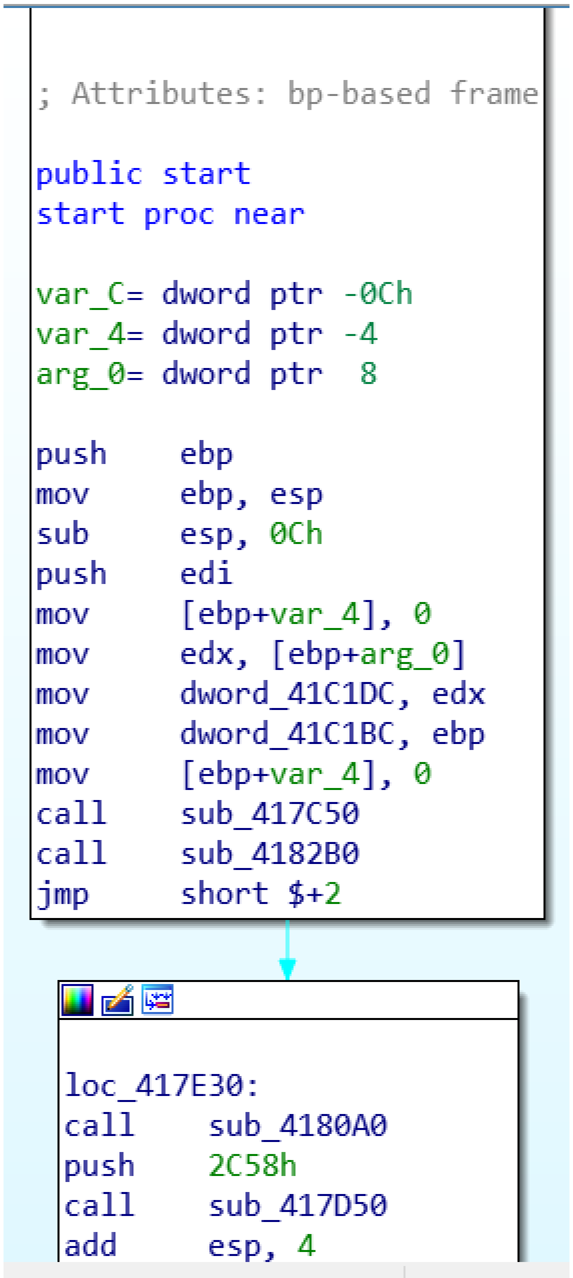
Open it in IDA: It shows that is low number of functions which another indicator that is packed



Function name	Segment	Start
sub_417B00	.text	0000000000417B00
sub_417B20	.text	0000000000417B20
sub_417C50	.text	0000000000417C50
sub_417C80	.text	0000000000417C80
sub_417CD0	.text	0000000000417CD0
sub_417D50	.text	0000000000417D50
sub_417DE0	.text	0000000000417DE0
start	.text	0000000000417E00
sub_417F30	.text	0000000000417F30
sub_417F90	.text	0000000000417F90
sub_418060	.text	0000000000418060
sub_4180A0	.text	00000000004180A0
sub_418150	.text	0000000000418150
sub_4182B0	.text	00000000004182B0

Figure(9):

Press over “start” which located in the function as in the previous figure to get started



```
; Attributes: bp-based frame

public start
start proc near

var_C= dword ptr -0Ch
var_4= dword ptr -4
arg_0= dword ptr  8

push    ebp
mov     ebp, esp
sub     esp, 0Ch
push    edi
mov     [ebp+var_4], 0
mov     edx, [ebp+arg_0]
mov     dword_41C1DC, edx
mov     dword_41C1BC, ebp
mov     [ebp+var_4], 0
call    sub_417C50
call    sub_4182B0
jmp     short $+2
```

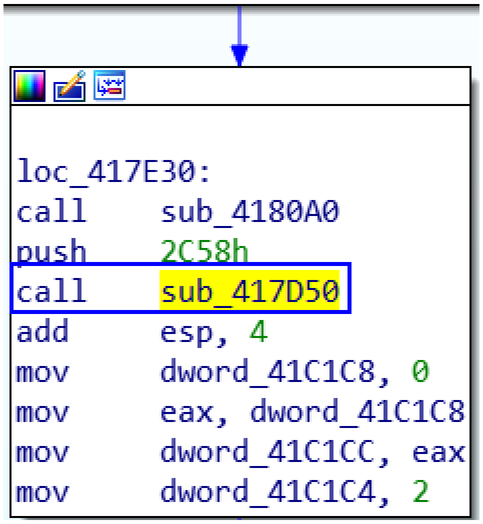
```
loc_417E30:
call    sub_4180A0
push    2C58h
call    sub_417D50
add     esp, 4
```

Figure(10):

Because Emotet malware uses a customized packer. we can try to unpack it through **dynamic analysis**. Through **dynamic analysis** the malware does the unpacking process. **The process will need to allocate memory for the next stage**.

So it’s a good assumption that we will see a **call to VirtualAlloc**. We need to search which function has VirtualAlloc call. 3

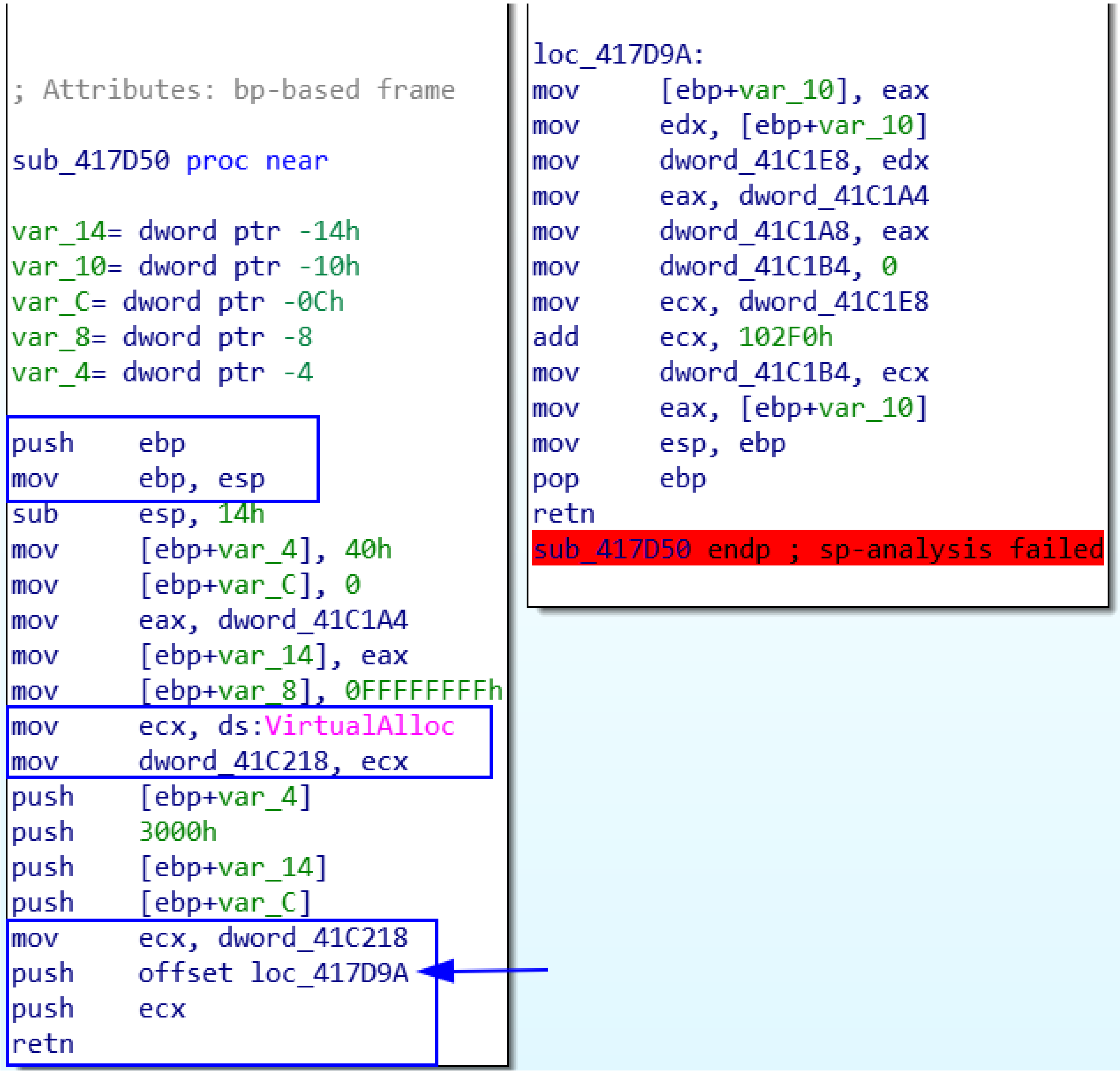
If you searched you will find that **call** `sub_417D50` is the unpacking routine



```
loc_417E30:
call    sub_4180A0
push    2C58h
call    sub_417D50
add     esp, 4
mov     dword_41C1C8, 0
mov     eax, dword_41C1C8
mov     dword_41C1CC, eax
mov     dword_41C1C4, 2
```

Figure(11):

This our unpacking function: `sub_417D50`



```
; Attributes: bp-based frame

sub_417D50 proc near

var_14= dword ptr -14h
var_10= dword ptr -10h
var_C= dword ptr -0Ch
var_8= dword ptr -8
var_4= dword ptr -4

push    ebp
mov     ebp, esp
sub     esp, 14h
mov     [ebp+var_4], 40h
mov     [ebp+var_C], 0
mov     eax, dword_41C1A4
mov     [ebp+var_14], eax
mov     [ebp+var_8], 0FFFFFFFFh
mov     ecx, ds:VirtualAlloc
mov     dword_41C218, ecx
push    [ebp+var_4]
push    3000h
push    [ebp+var_14]
push    [ebp+var_C]
mov     ecx, dword_41C218
push    offset loc_417D9A
push    ecx
retn
```

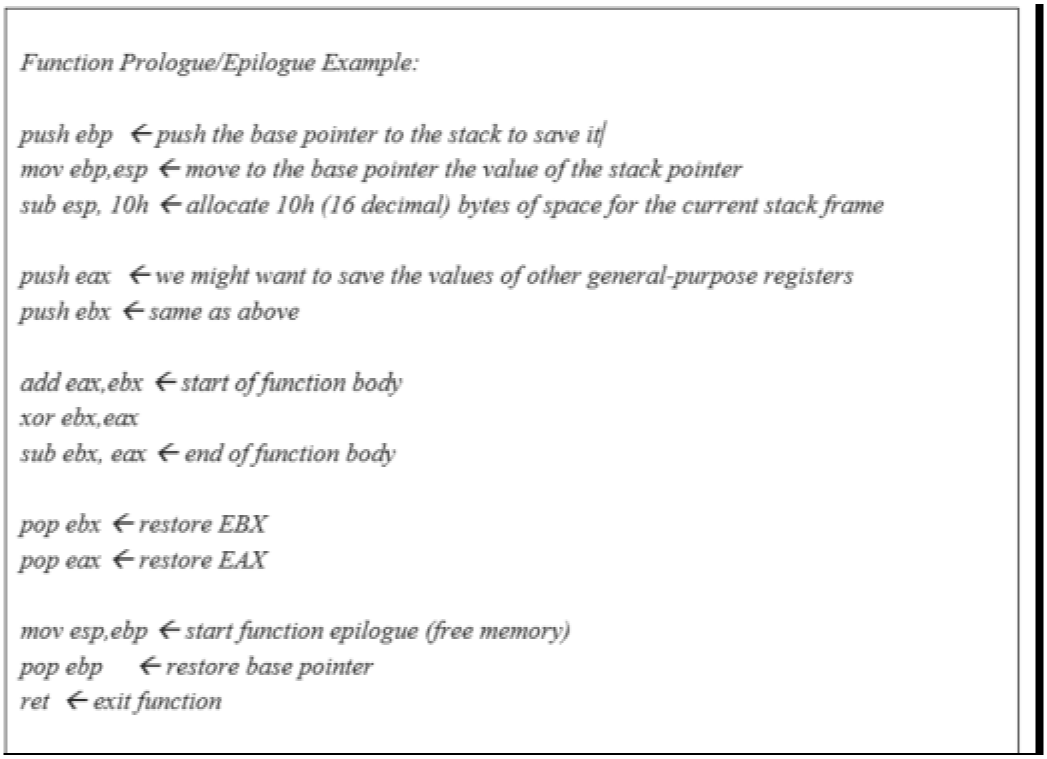
```
loc_417D9A:
mov     [ebp+var_10], eax
mov     edx, [ebp+var_10]
mov     dword_41C1E8, edx
mov     eax, dword_41C1A4
mov     dword_41C1A8, eax
mov     dword_41C1B4, 0
mov     ecx, dword_41C1E8
add     ecx, 102F0h
mov     dword_41C1B4, ecx
mov     eax, [ebp+var_10]
mov     esp, ebp
pop     ebp
retn
sub_417D50 endp ; sp-analysis failed
```

Figure(12):

Abnormal epilogue

First we need to clear **what normal prologue and epilogue are?**

The procedure prologue and epilogue are standard initialization sequences that compilers generate for almost all of their functions.

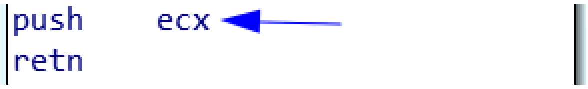


Function Prologue/Epilogue Example:

```
push ebp   ←push the base pointer to the stack to save it
mov ebp,esp ←move to the base pointer the value of the stack pointer
sub esp, 10h ←allocate 10h (16 decimal) bytes of space for the current stack frame

push eax   ←we might want to save the values of other general-purpose registers
push ebx ←same as above

add eax,ebx ←start of function body
xor ebx,eax
sub ebx, eax ←end of function body

pop ebx ←restore EBX
pop eax ←restore EAX

mov esp,ebp ←start function epilogue (free memory)
pop ebp     ←restore base pointer
ret  ←exit function
```

Figure(13):

What is **NOT normal** here is epilogue in the last figure:



```
push    ecx
retn
```

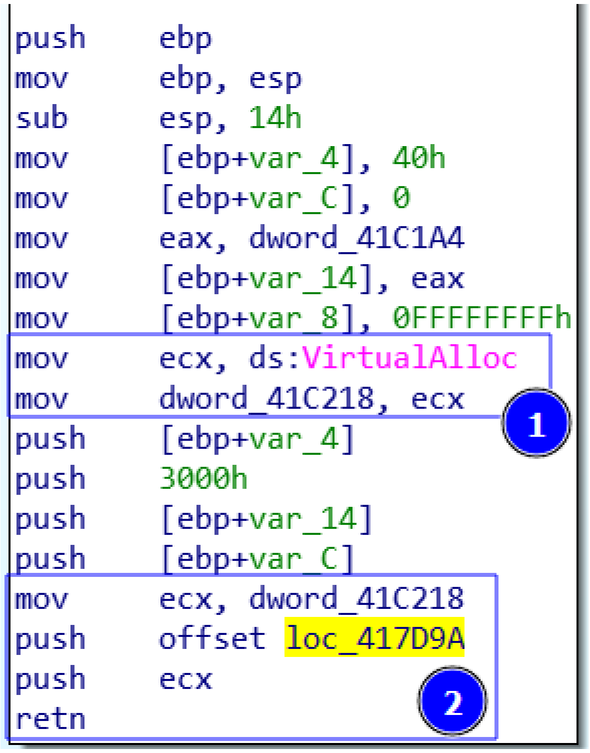
Figure(14):

You don’t `push` anything before `ret` this called abnormal.

normal epilogue is to `pop EBP` before `ret` . Here it will return `ecx` because it executes the last instruction- top of the stack-.

And the real return is from this function `loc_417D9A` because this is 2nd top of the stack.

We need to know what is happening in this function?



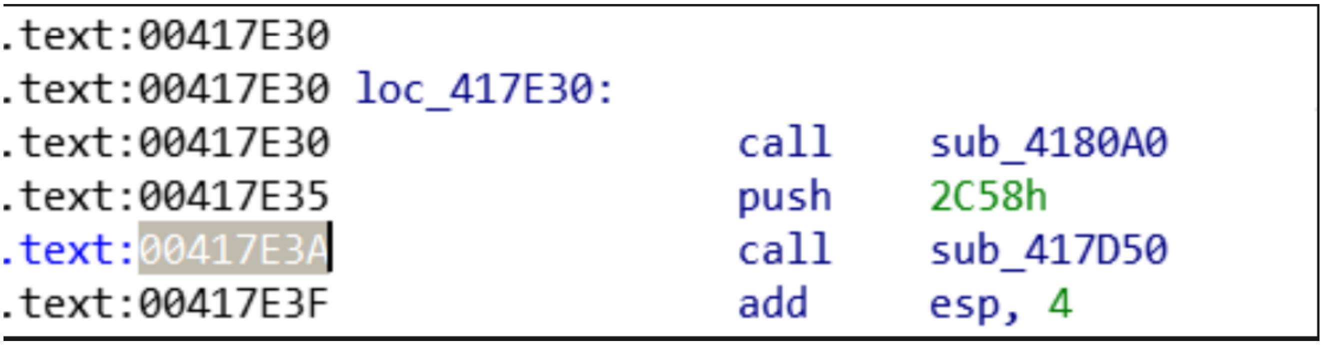
```
push    ebp
mov     ebp, esp
sub     esp, 14h
mov     [ebp+var_4], 40h
mov     [ebp+var_C], 0
mov     eax, dword_41C1A4
mov     [ebp+var_14], eax
mov     [ebp+var_8], 0FFFFFFFFh
mov     ecx, ds:VirtualAlloc
mov     dword_41C218, ecx
push    [ebp+var_4]
push    3000h
push    [ebp+var_14]
push    [ebp+var_C]
mov     ecx, dword_41C218
push    offset loc_417D9A
push    ecx
retn
```

Figure(15):

In the last figure we see the coming:

- `VirtualAlloc` is moved to `ECX` , then
- `ECX` is moved to `dword_41C218` , then
- `dword_41C218` is moved to `ECX`
- then `push ECX` and then `ret`
- And the real return is from this function `loc_417D9A`

So we need to know the address of this function to set a Breakpoint in x64dbg by pressing `space` .



```
.text:00417E30
.text:00417E30 loc_417E30:
.text:00417E30                 call    sub_4180A0
.text:00417E35                 push    2C58h
.text:00417E3A                 call    sub_417D50
.text:00417E3F                 add     esp, 4
```

Figure(16):

We know that code is packed. **We search for abnormal jumps:**

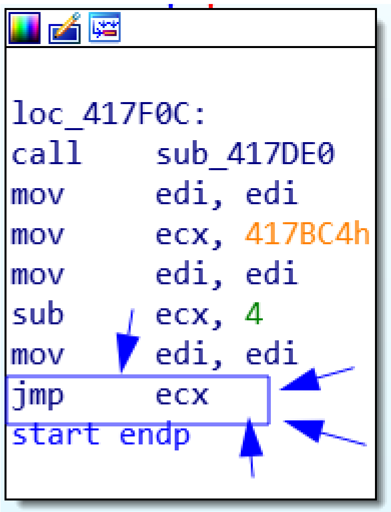
- `jmp` or `call` Instructions to registers
- `Jmp` to strange memory addresses (long jump)

Why searching for abnormal jumps? the address to the location of where data is being unpacked to is stored in a register (such as `ecx`), and that memory address is often in an entirely different section.

I will write an article about “indicators of packed file”. InshAllah

If we return to **start function** and search you will find it.

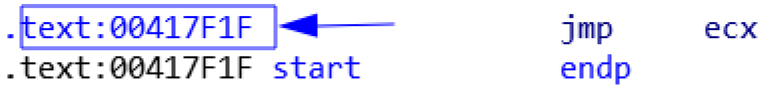
Here we see our abnormal `jmp ecx`:



```
loc_417F0C:
call    sub_417DE0
mov     edi, edi
mov     ecx, 417BC4h
mov     edi, edi
sub     ecx, 4
mov     edi, edi
jmp     ecx
start endp
```

Figure(17):

Press `space` to get its address: `00417F1F`.



```
.text:00417F1F                jmp     ecx
.text:00417F1F start          endp
```

Figure(18):

How to Unpack in the next part. InshAllah

Edit: part 02

Article quote

> المنازل العليا لا تُنال إلّا بالبلاء

References

Inspired by: https://malgamy.github.io/malware-analysis/Emotet-Malware-0x01/

1- https://www.darkreading.com/edge-articles/emotet-101-how-the-ransomware-works—-and-why-it-s-so-darn-effective

2- https://malware.news/t/the-basics-of-packed-malware-manually-unpacking-upx-executables/35961

3- https://distributedcompute.com/2020/02/20/unpacking-emotet/