Unpacking an Emotet trojan

By Otavio M.

Published: 2023-07-23 · Archived: 2026-04-05 16:51:57 UTC

Emotet, in general, is a banking trojan. Identified in-the-wild for the first time in 2014 as a stealth info stealer (mainly targeting banking informations), emotet has evolved to a sofisticated trojan over the years; Now having funcionalities that goes from simply keylogging to self-spreading (as worms do).

The main campaign of the malware is through malspam. Using the “urgency” pretext, emotet make his victims by malicous documents with a macro embedded, malicious scripts or malicious links. Then, the pretext comes with subjects like: “Your Invoice”, “Payment Details” or an upcoming shipment from well-known companies.

Emotet uses some tricks to evade and prevent his detection and analysis. The malware will check for common malware analysis tools (like IDA or Wireshark), check if it is running on a virtual environment and remain sleep, and every sample comes packed or encrypted.

Today we will be covering the unpacking of a sample from emotet family.

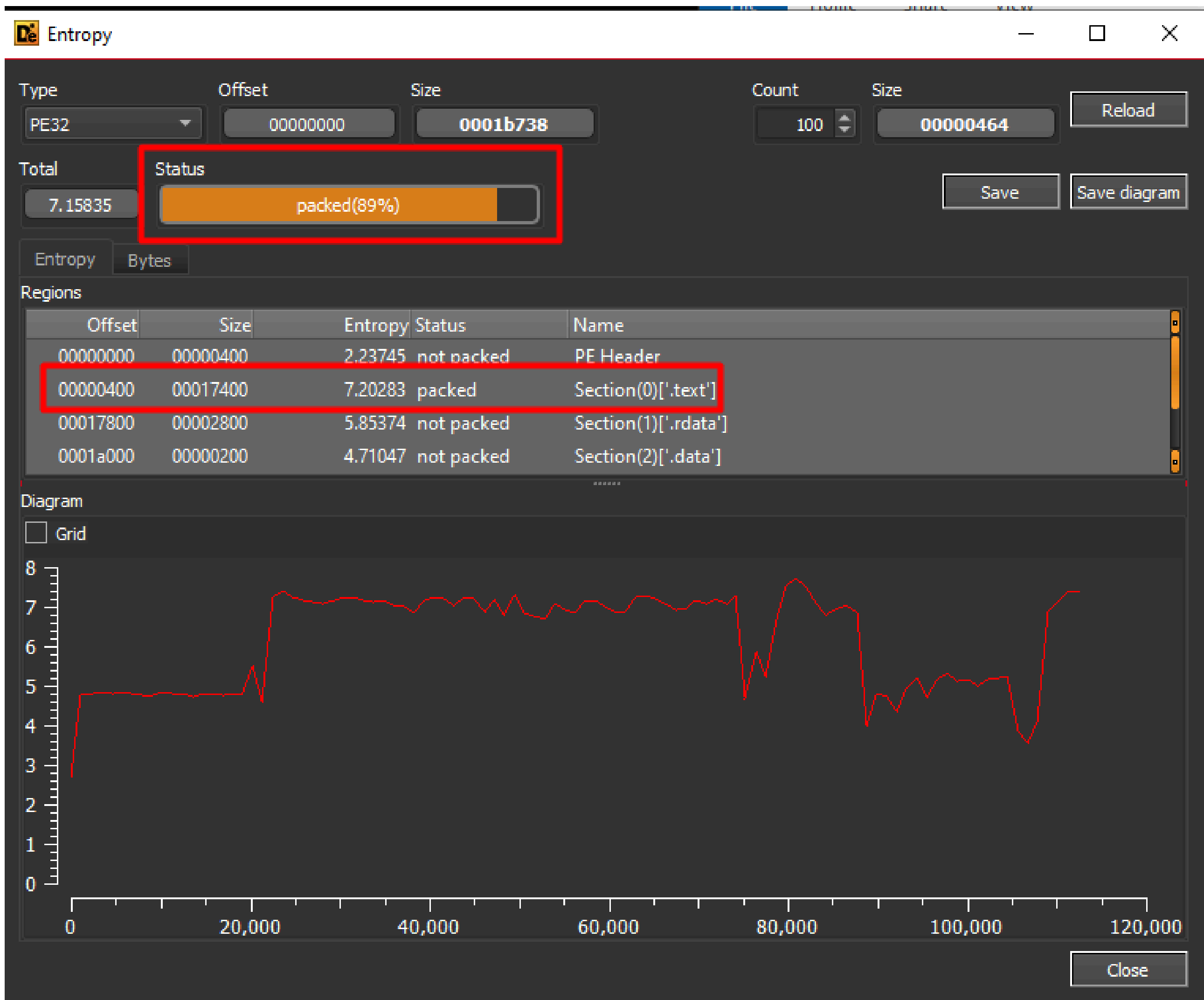
Reconnaissance

```
sha256: 3a9494f66babc7deb43f65f9f28c44bd9bd4b3237031d80314ae7eb3526a4d8f
md5: ca06acd3e1cab1691a7670a5f23baef4
```

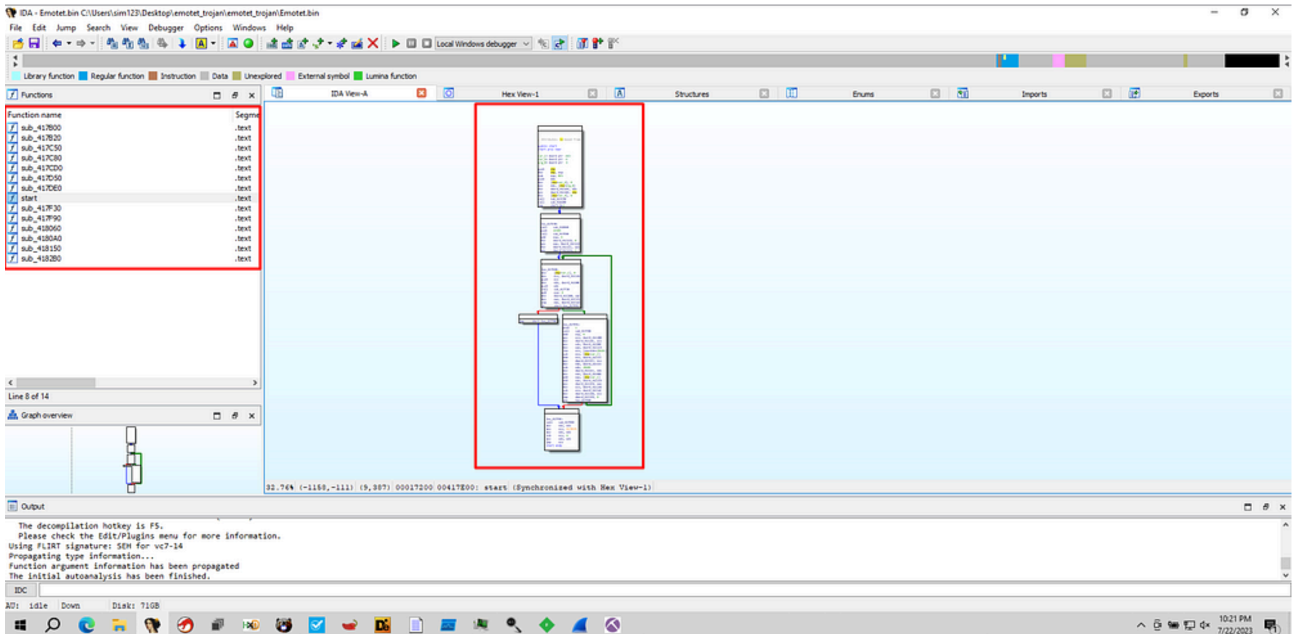
First, we need to know if the sample is definitely packed. Lets open it on DiE.

We can see that it is a 32-bit binary, made in C/C++ and having a certificate stored in the overlay section (WinAuth(2.0))

Looking at the entropy, we see that the binary has 89% chance of being packed.



We can confirm it by looking at the sample on IDA.



IDA shows some indicators of packing, like:

- Lack of subroutines on a malware this sofisticated.
- Lack of internet-related APIs
- Main (start) function seems small and doesn't have any WindowsAPI or indirect calls.

But, we can have the proof of it by looking into common packing APIs:

- CreateProcessInternalW()
- VirtualAlloc()
- VirtualAllocEx()
- VirtualProtect() / ZwProtectVirtualMemory()
- WriteProcessMemory() / NtWriteProcessMemory()
- ResumeThread() / NtResumeThread()
- CryptDecrypt() / RtlDecompressBuffer()
- NtCreateSection() + MapViewOfSection() / ZwMapViewOfSection()
- UnmapViewOfSection() / ZwUnmapViewOfSection()
- NtWriteVirtualMemory()
- NtReadVirtualMemor

Searching for VirtualAlloc(), we will soon find the subroutine that probably is the responsible for unpacking the malware.

```
; 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
```

Now it gets a little bit more complicated. The red box marks an “abnormal epilogue”. An “abnormal epilogue” occurs when we have some pushes into the stack and not a single pop before it returns.

> You can notice that ds:VirtualAlloc is being moved into ecx, then ecx is pushed onto the stack and the return is called, meaning the call of VirtualAlloc().

After calling ecx (VirtualAlloc), the return will execute the second push from the stack (osffset loc_417d9a), executing whatever is present on the second block, and then the real return will come.

Normally, after the code finishes the unpack, we will have a indirect call to it.

```
.text:00417F0C loc_417F0C:                          ; CODE XREF: start+8F↑j
.text:00417F0C                 call    sub_417DE0
.text:00417F11                 mov     edi, edi
.text:00417F13                 mov     ecx, 417BC4h
.text:00417F18                 mov     edi, edi
.text:00417F1A                 sub     ecx, 4
.text:00417F1D                 mov     edi, edi
.text:00417F1F                 jmp     ecx
.text:00417F1F start           endp
```

We can confirm it by looking at the end of the main function, which has an “jmp ecx”.

Again, take notes of the address. 0x00417F1F.

Unpacking

So we got two addresses to set breakpoints in:

We can open it on x64dbg, search for those addresses (Ctrl+G), and then set the breakpoints.

Type	Address	Module/Label/Exception	State	Disassembly	Hits	Summary
Software						
	00417E3F	emotet.bin	Enabled	add esp,4	0	
	00417F1F	emotet.bin	Enabled	jmp ecx	0	

By going to the “jmp” breakpoint and scrolling down, we will encounter another abnormal set of instructions. This time, a subroutine is being pushed into the stack and not popped, the the ret makes the call to that subroutine.

```
00417C34    8B15 B4C14100    mov edx,dword ptr ds:[41C1B4]
00417C3A    52               push edx
00417C3B    C3               ret
```

Following it on the dump, we can see that it is the newly allocated memory.

```
EIP  00417C34   8B15 B4C14100   mov edx,dword ptr ds:[41C1B4]
     00417C3A   52              push edx
     00417C3B   C3              ret
     00417C3C   5D              pop ebp
     00417C3D   C3              ret
     00417C3E   CC              int3
     00417C3F   CC              int3
     00417C40   55              push ebp
     00417C41   8BEC            mov ebp,esp
     00417C43   8B45 08         mov eax,dword ptr ss:[ebp+8]
     00417C46   05 F7130000     add eax,13F7
     00417C4B   5D              pop ebp
     00417C4C   C3              ret
     00417C4D   CC              int3

edx=458 L'j'
dword ptr ds:[emotet.0041C1B4]=021D02F0

.text:00417C34 emotet.bin:$17C34 #17034

Dump 1 | Dump 2 | Dump 3 | Dump 4 | Dump 5 | Watch 1 | [x=] Locals | Struct

Address   Hex                                              ASCII
021D02F0  55 8B EC 81 EC 80 00 00 00 C7 45 F4 00 00 00 00  U.ì.ì....ÇEô....
021D0300  C7 45 F4 00 00 00 00 EB 09 8B 45 F4 83 C0 01 89  ÇEô....ë..Eô.À..
021D0310  45 F4 81 7D F4 D5 DC 32 00 73 12 6A 58 6A 00 8D  Eô.}ôÕÜ2.s.jXj..
021D0320  4D 98 51 E8 58 F8 FF FF 83 C4 0C EB DC 8B 45 04  M.QèXøÿÿ.Ä.ëÜ.E.
021D0330  89 45 F8 8B 45 08 89 45 8C 89 6D 98 8D 55 98 52  .Eø.E..E..m..U.R
021D0340  E8 EB F4 FF FF 83 C4 04 E8 E3 F3 FF FF 89 45 84  èëôÿÿ.Ä.èãóÿÿ.E.
021D0350  [illegible] 5 A0 8B 4D 8C 89 4D AC 8B 55 AC 89  .Eø.E .M..M¬.U¬.
021D0360  [illegible] C 8B 4D F0 03 48 3C 89 4D F0 8B 55  Uð.E¬.Mð.H<.Mð.U
021D0370  F0 89 55 90 8B 45 90 0F B7 48 16 89 4D A4 E8 AD  ð.U..E..·H..M¤è.
021D0380  F3 FF FF 05 20 11 40 00 89 45 88 8B 55 88 8B 02  óÿÿ. .@..E..U...
021D0390  89 45 80 E8 98 F3 FF FF 05 24 11 40 00 89 45 88  .E.è.óÿÿ.$.@..E.
021D03A0  8B 4D 80 51 E8 87 F6 FF FF 83 C4 04 89 45 94 8B  .M.Qè.öÿÿ.Ä..E..
021D03B0  55 80 52 8B 45 88 50 8B 4D 94 51 E8 00 F8 FF FF  U.R.E.P.M.Qè.øÿÿ
021D03C0  83 C4 0C 8B 55 80 52 8B 45 94 50 E8 D0 FE FF FF  .Ä..U.R.E.PèÐþÿÿ
021D03D0  83 C4 08 8B 4D 94 51 8D 55 98 52 E8 A0 FB FF FF  .Ä..M.Q.U.Rè ûÿÿ
021D03E0  85 C0 75 04 33 C0 EB 1A 83 7D A0 00 74 09 8B 45  .Àu.3Àë..} .t..E
021D03F0  B0 8B 4D 98 89 41 10 8B 55 A8 8B 65 98 5D 58 58  °.M..A..U¨.e.]XX
```

Having the knowledge of the abnormal epilogue, we will stepover until the ret is called and take us to another stage of the unpack.

Now, we will have to stepover that code and analyze all those calls(e. g. call 21CF710).

After some analysis, we eventually will find the call to 3AF830, which has some interesting code:

We can see that this subroutine is using "stack strings" as a form of evasion/obfuscation. This piece of code tries to obfuscate the passing of arguments "LoadLibraryExA" and "kernel32.dll" to the call 21CFE10. And many others.

As we know what is happening here, lets go straight to the return of that subroutine.

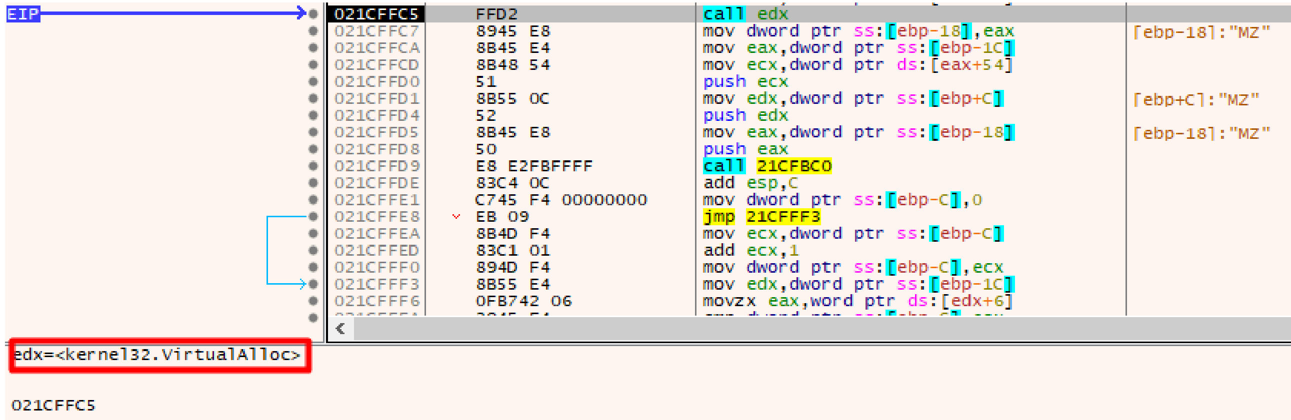
After returning to the main unpacking subroutine, we notice that the upcoming calls has the same mechanism as 3AF830.

But, the last one is important. Since the code inside of it is different, is worth a analysis.

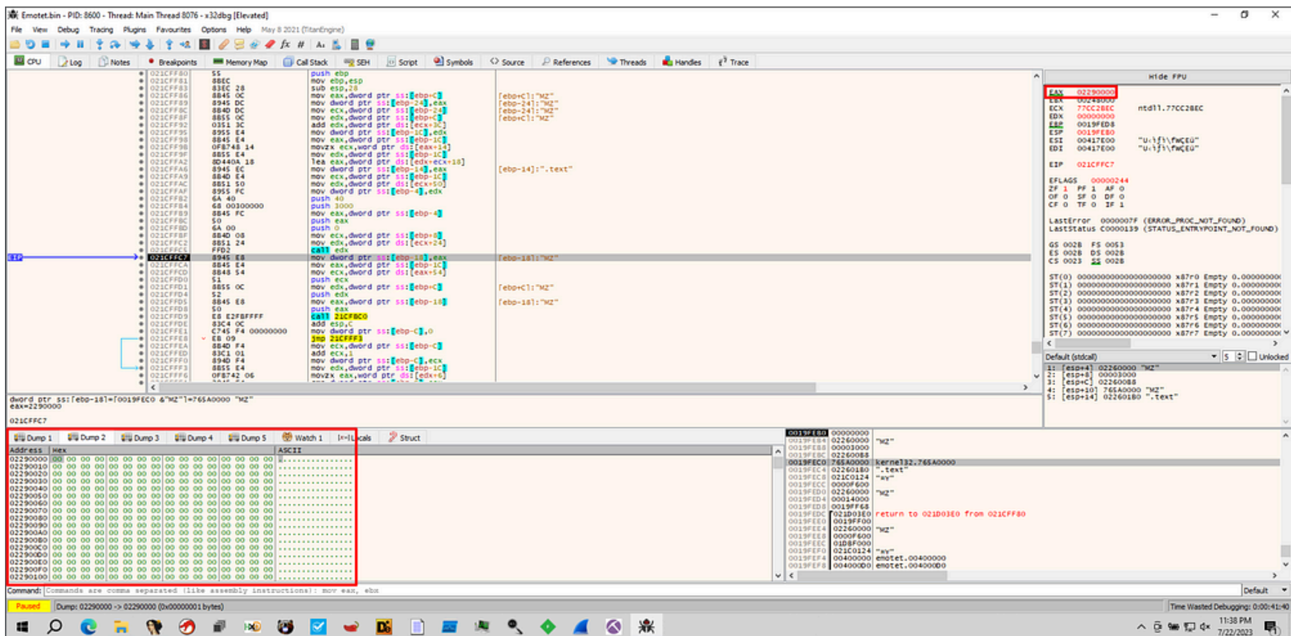
We can assume that edx is a call to VirtualAlloc, because some of the parameters passed to it are common parameters passed to VirtualAlloc itself.

> push 40 as: PAGE_EXECUTE_READWRITE (flProtect) && push 3000 as: MEM_COMMIT / MEM_RESERVE

To confirm it, we stepover to that call.



Knowing the return of that API is the base address of the recent allocated memory, we can follow the address in EAX (return).



After stepping over a little bit more, we will notice that memory being populated, more specifically after the call to 21CFBC0.

Then, we have to continue to stepover until the subroutine finishes populating that memory.

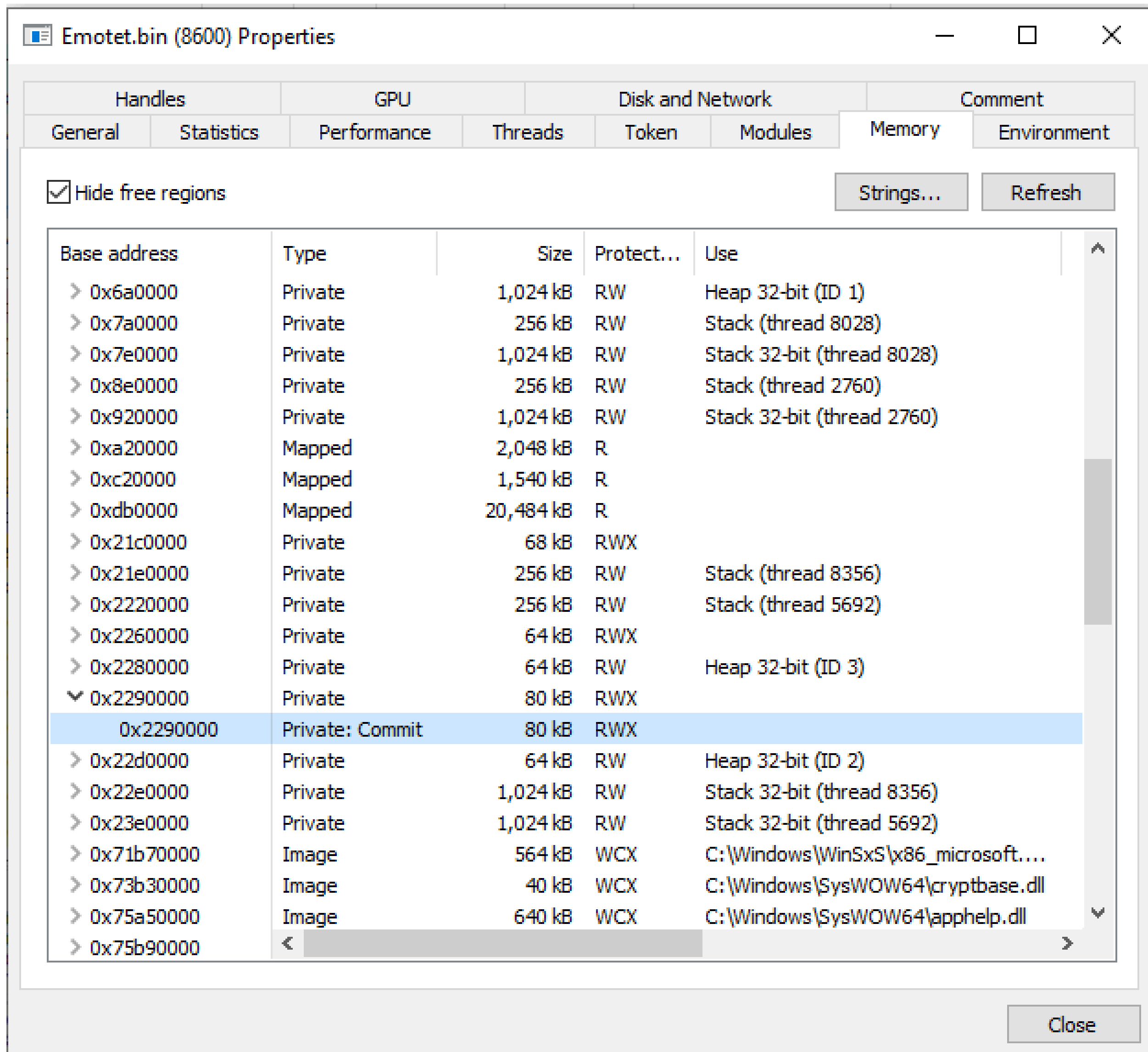
> We will know it finished when we encounter a return.

After all of this, we can finally dump that memory.

The dump is made by:

```
right-clicking the first byte -> following it on memory map -> right-click > dump memory to file
```

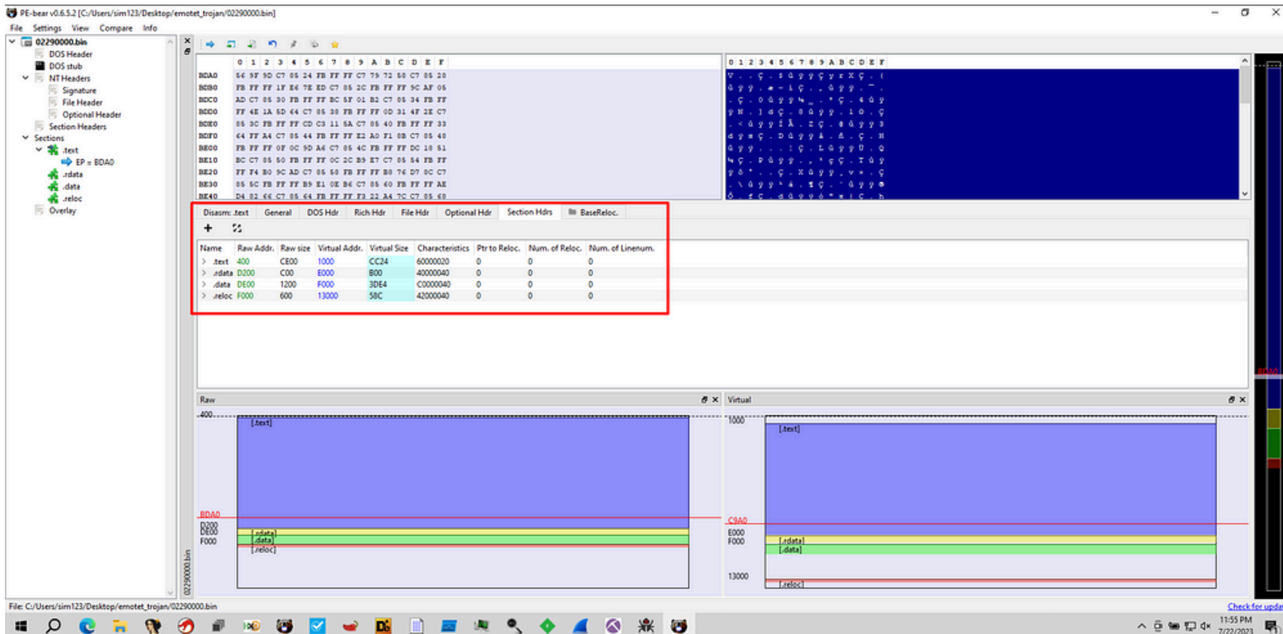
> Another way of doing it is: Opening the process in process hacker (administrator), searching for the address marked in dump, then dumping to a file.



Unmapping

If we open the recently dumped file on IDA for example, it will be completely scrambled. The reason for this is because the binary may be unaligned or mapped, so we need to align and unmap it.

To align and unmap it, we will open the dumped binary on PE Bear.



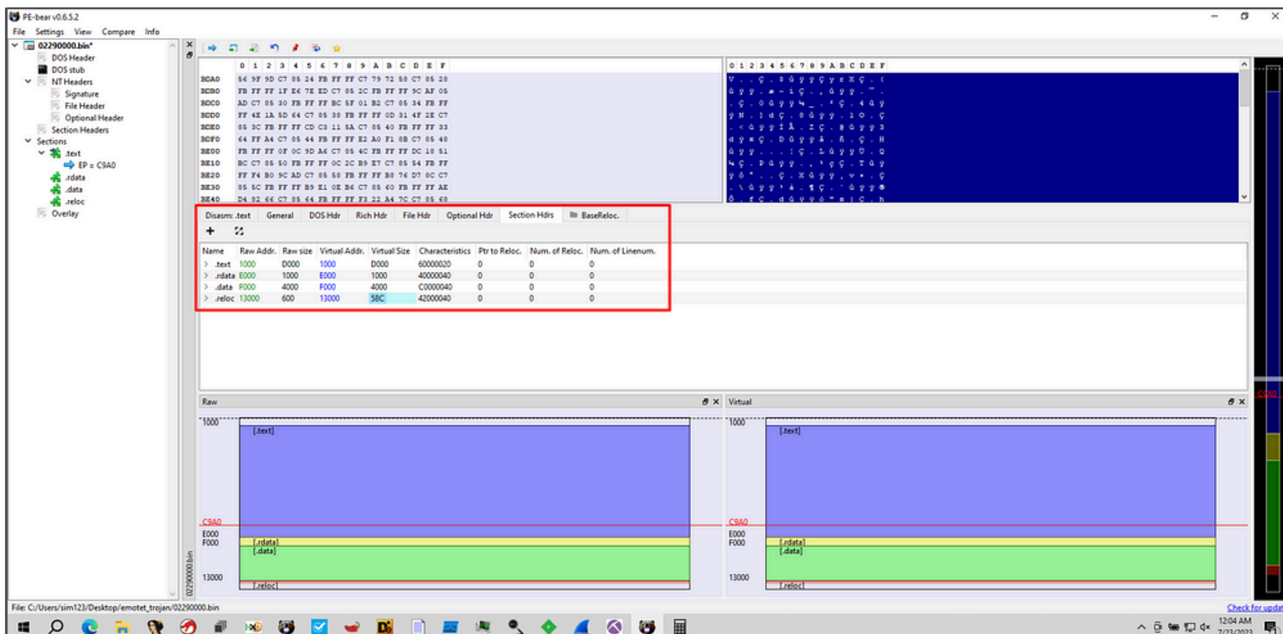
As you can see, we don’t have the import tab, meaning we need to unmap the binary.

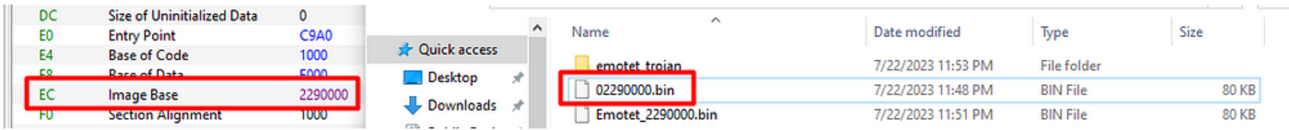
The unmap process is simples. It will take 4 steps:

1. Change the Raw Address to the same as Virtual Address.
2. Ajust the Raw Size by subtracting the first sectin by the second, and so on.
3. Copy the adjusted Raw Size to the Virtual Size.
4. Fix the Base Address in the Optional Header (The same as the dump address).

> You can subtract those values with the windows calculator and the “programmer” option.

After that, we have the Unmapped Binary:



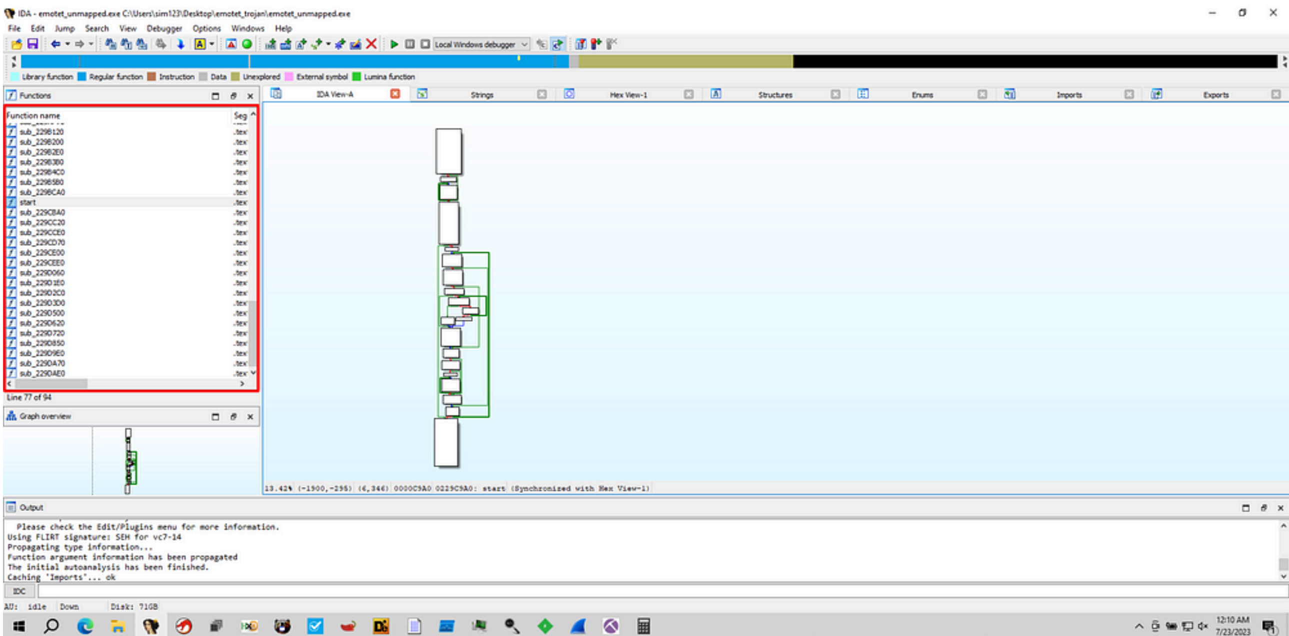


Now, all we have to do is save the new binary.

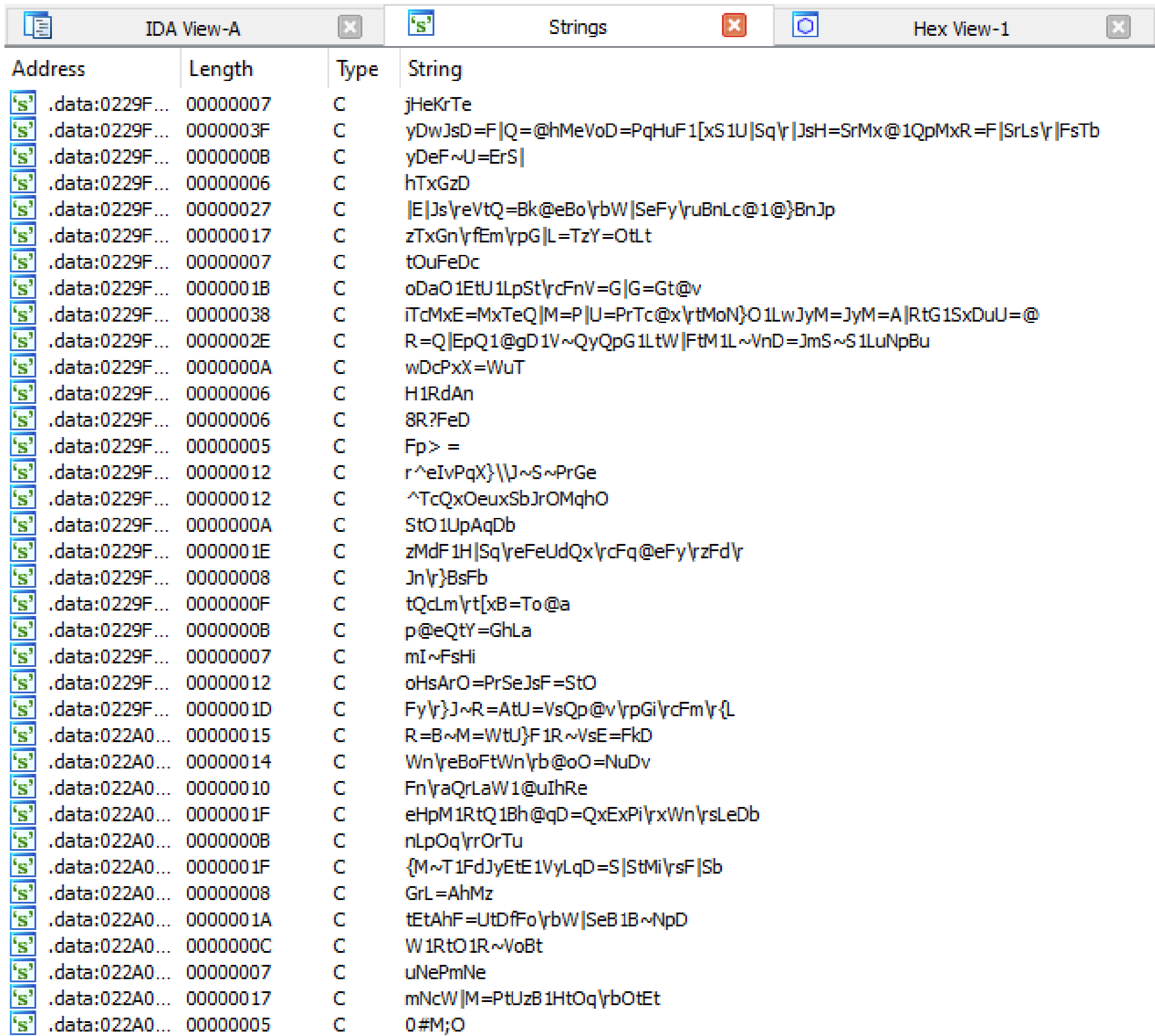
```
Binary name -> right-click -> Save the executable as...
```

Confirming

To confirm all we’ve done , let’s open it on IDA.



As you notice, now we have plenty of functions to analyze, and assuming by it’s strings and imports, the malware is unpacked but has a severe method of encryption and dynamic linking.



And that's it for today, hope you liked and learned something from this article. Thank you!!!

Source: https://estr3llas.github.io/unpacking-an-emotet-trojan/