Unpacking Emotet Trojan

By mov eax, 27

Published: 2023-08-13 · Archived: 2026-04-05 21:27:12 UTC

7 min read

Jul 23, 2023

Press enter or click to view image in full size

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**.

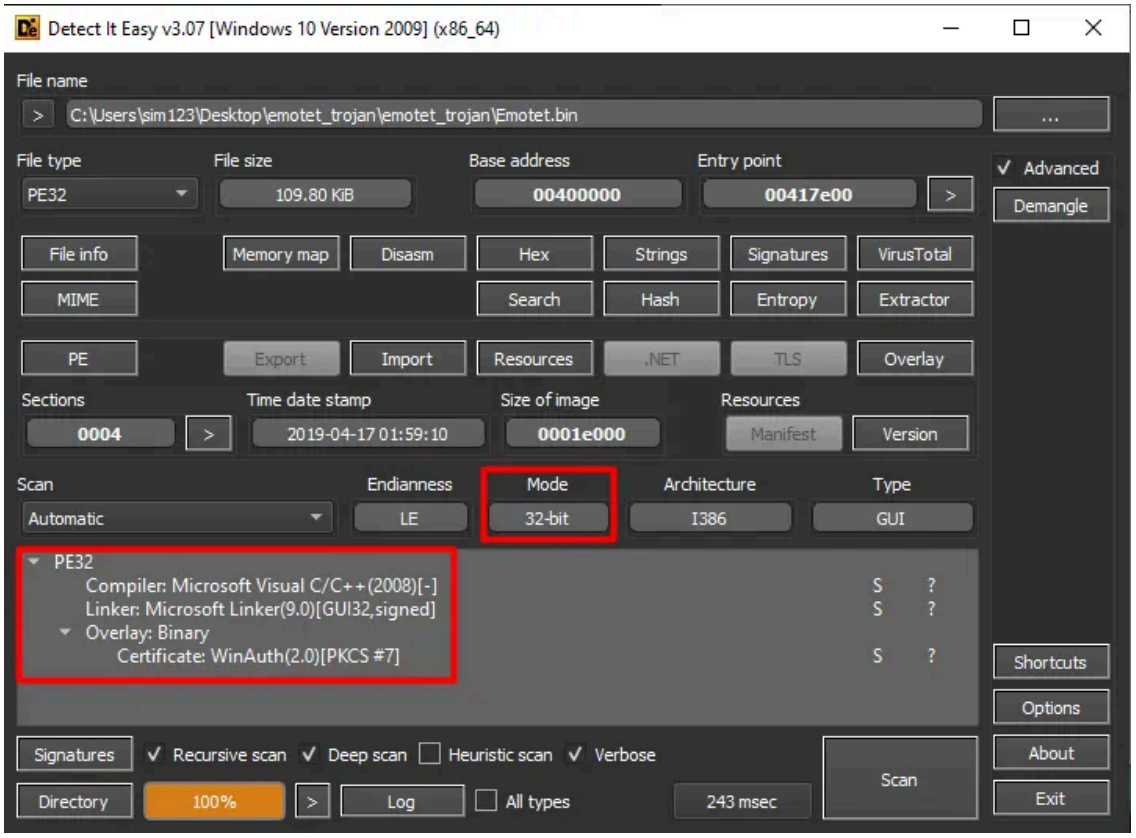
Today we will be covering the **unpacking** of a sample from emotet family.

Reconnaissance

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

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

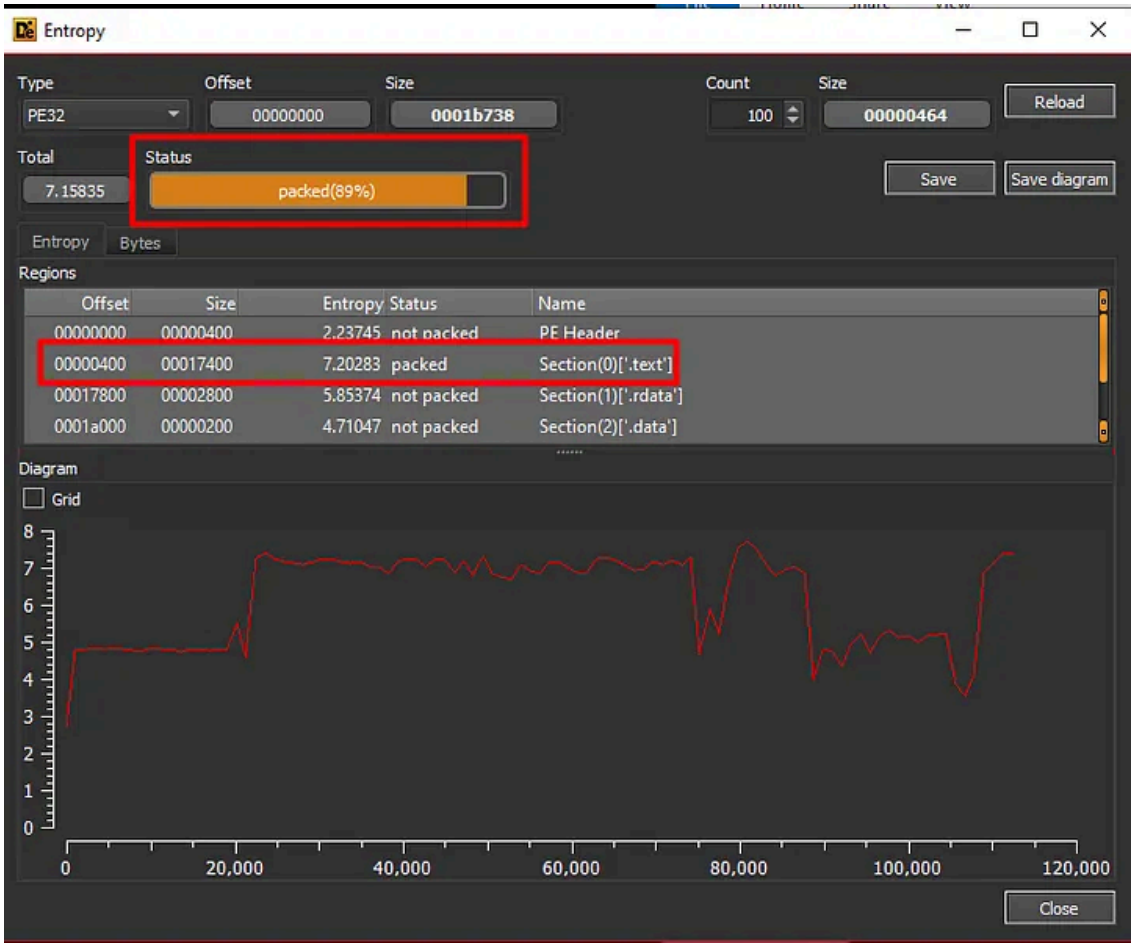
Press enter or click to view image in full size



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)*)

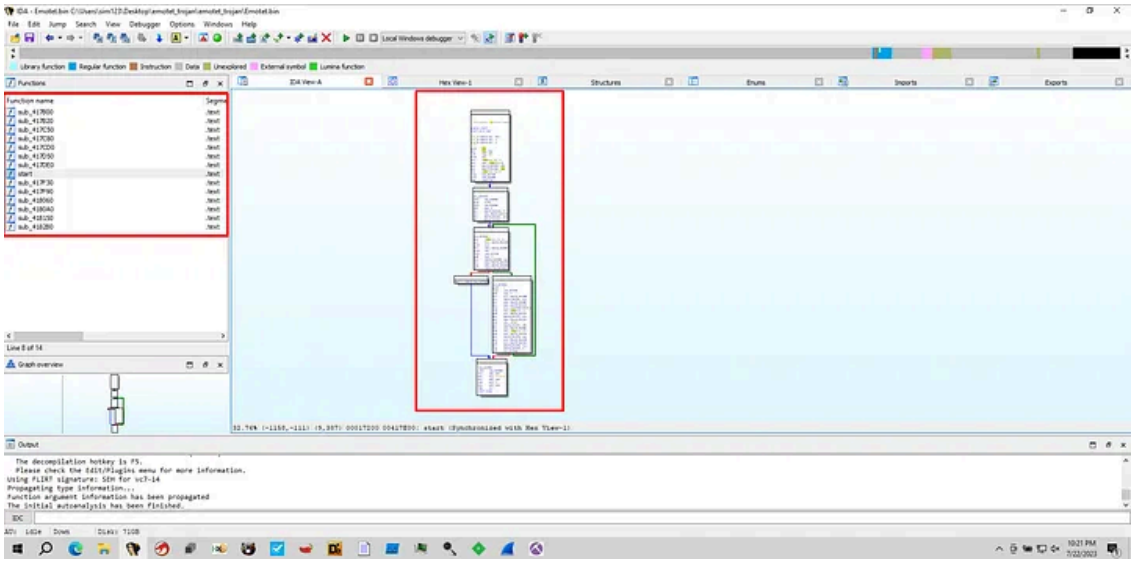
By looking at the **entropy**, we see that the binary has ***89%*** chance of being packed.

Press enter or click to view image in full size



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

Press enter or click to view image in full size



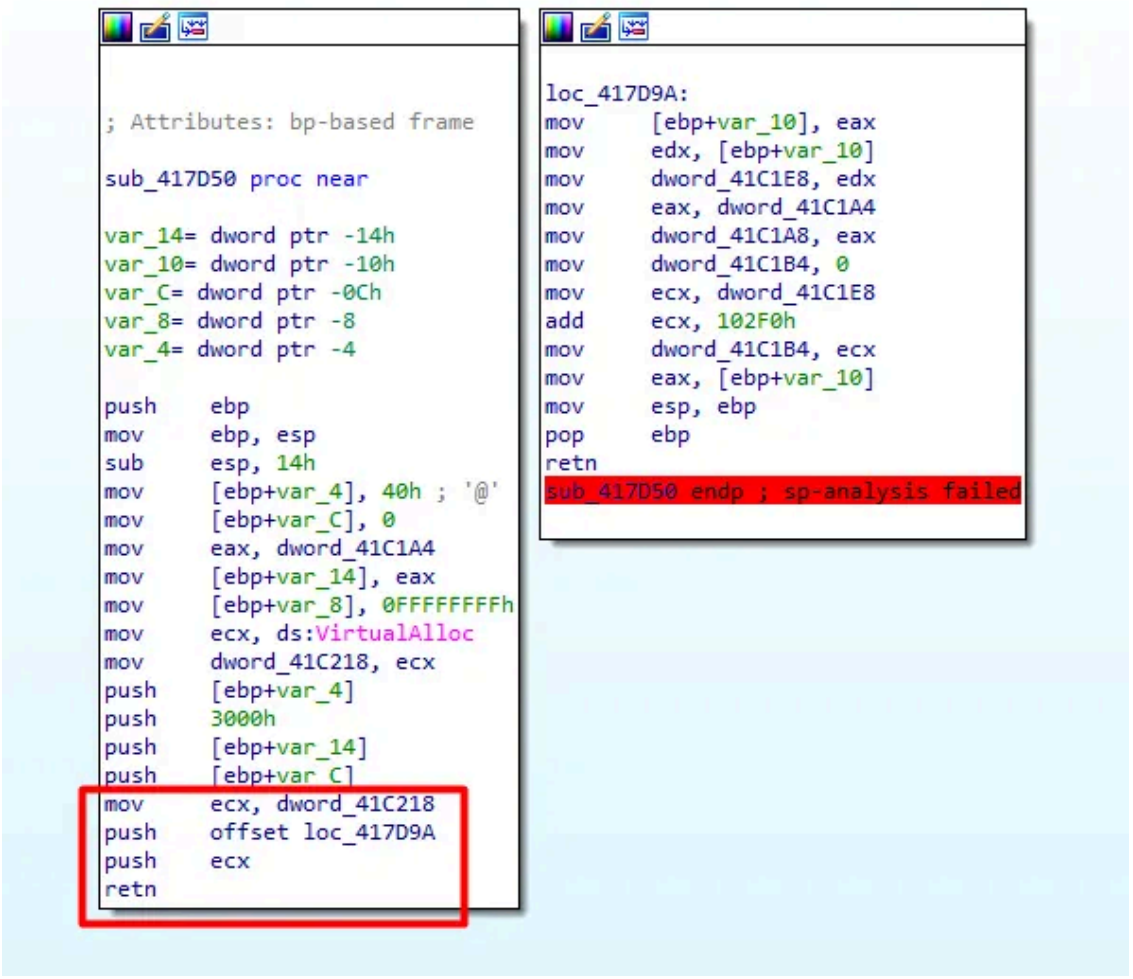
IDA shows some **indicators of packing**, like:

- Lack of functions on a malware this sofisticated.
- Main (start) function seems small and doesn't have any WindowsAPI call.

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()**
- **NtReadVirtualMemory()**

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



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.

```
.text:00417E3A                call    sub_417D50
.text:00417E3F                add     esp, 4
```

So, the return of the subroutine will be on ***0x00417E3F***.

Normally, after the code finishes the unpack, **we will have a jmp 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:

```
0x00417E3F
0x00417F1F
```

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

Press enter or click to view image in full size

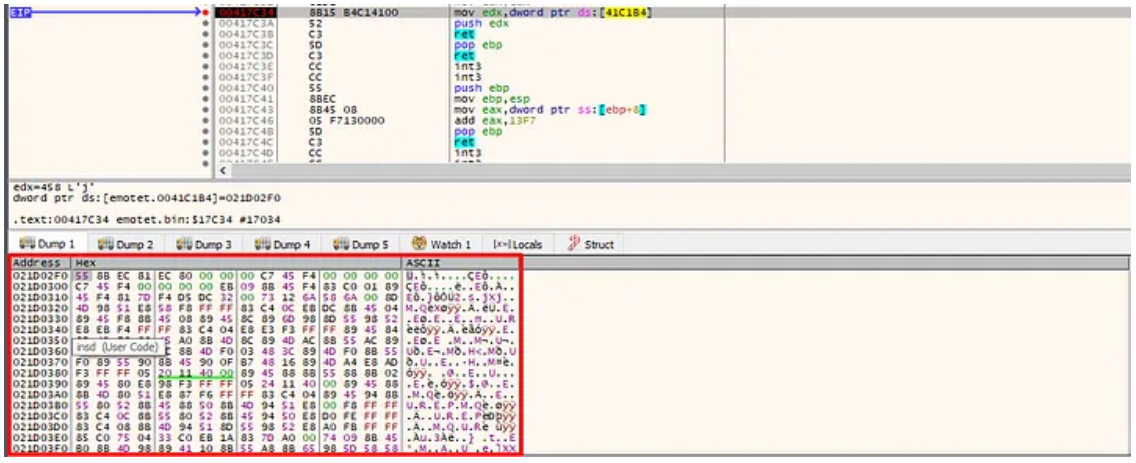
Type	Address	Module/Label/Exception	State	Disassembly	Hits	Summary
Software						
	00417E3F	emotet.bin	Enabled	add esp,4	0	
	00417F1F	emotet.bin	Enabled	jmp ecx	0	

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**.

Press enter or click to view image in full size



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

Press enter or click to view image in full size

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

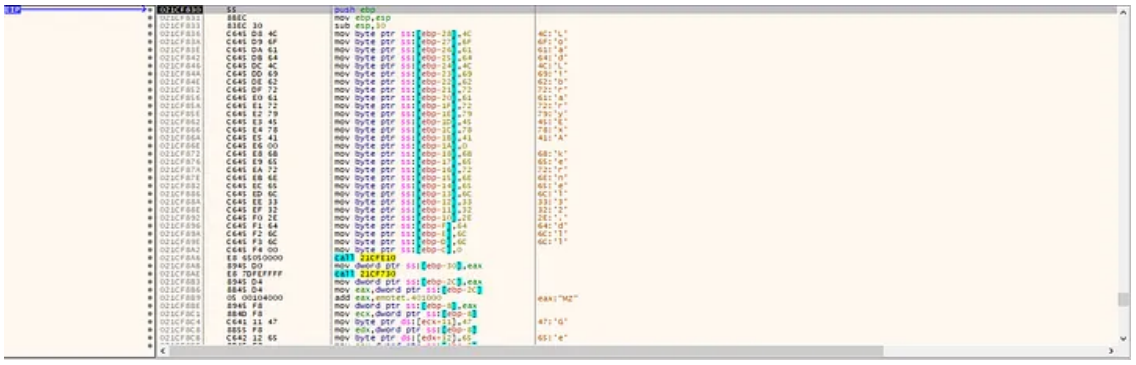
Get mov eax, 27’s stories in your inbox

Join Medium for free to get updates from this writer.

Remember me for faster sign in

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

Press enter or click to view image in full size



We can see that this subroutine is using "***stack string***s" 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.

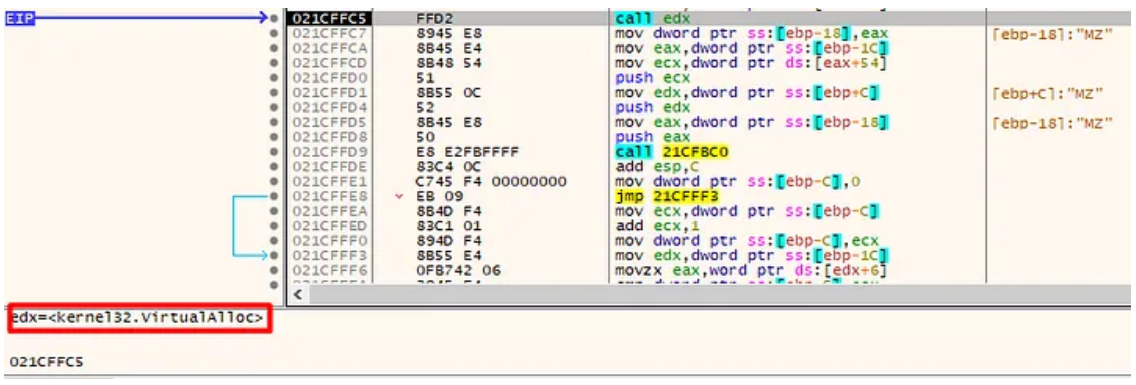
Press enter or click to view image in full size

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.

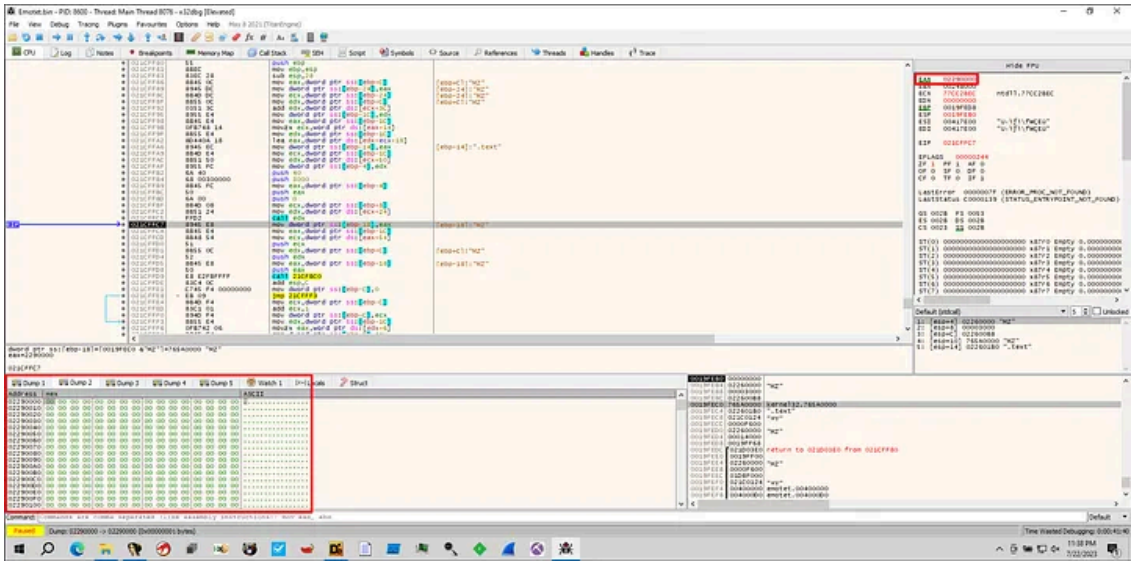
Press enter or click to view image in full size



:)

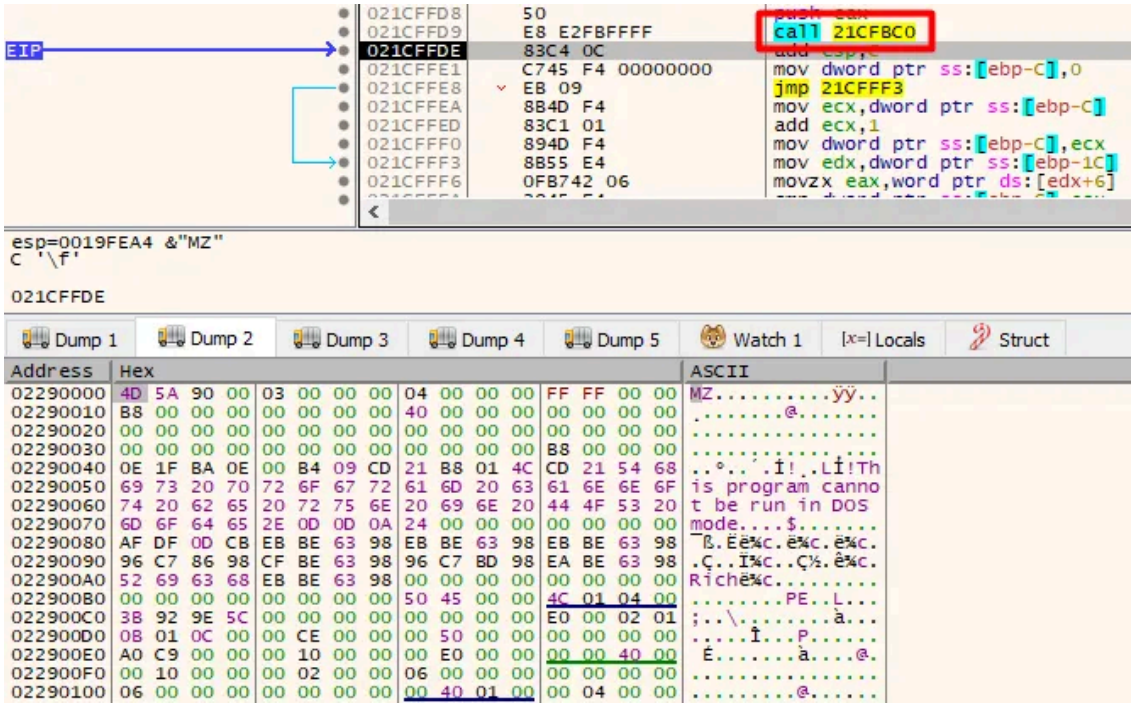
Knowing the return of that API is the **base address of the recent allocated memory**, we can follow the address in ***EAX*** (return).

Press enter or click to view image in full size



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

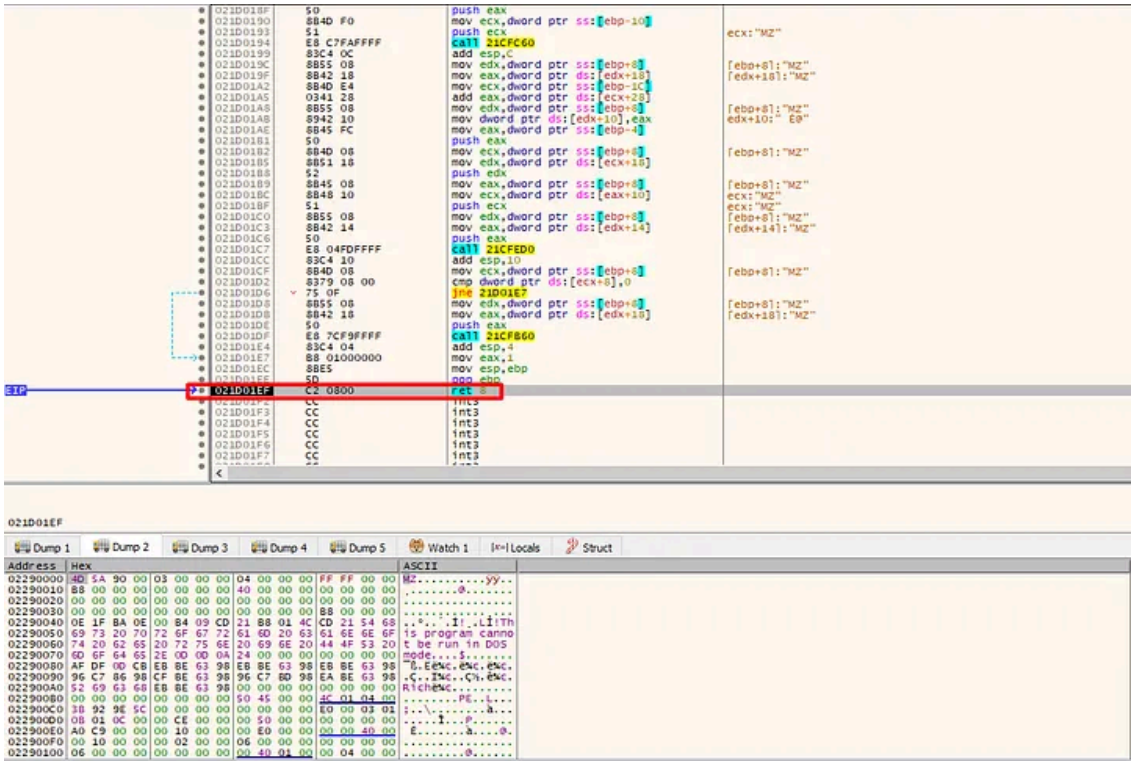
Press enter or click to view image in full size



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

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

Press enter or click to view image in full size

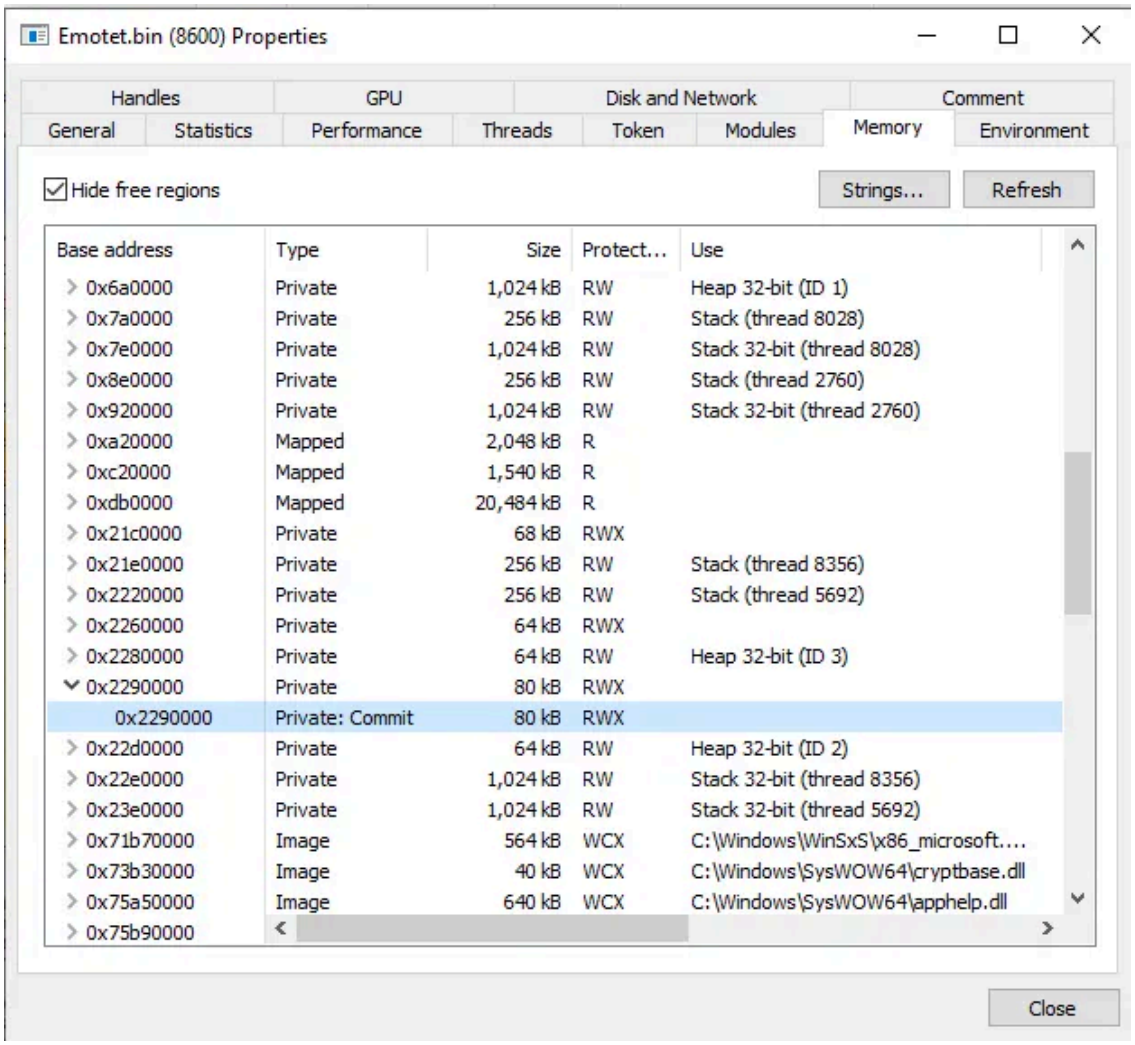


After all of this, we can finally **dump that memory**.

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.

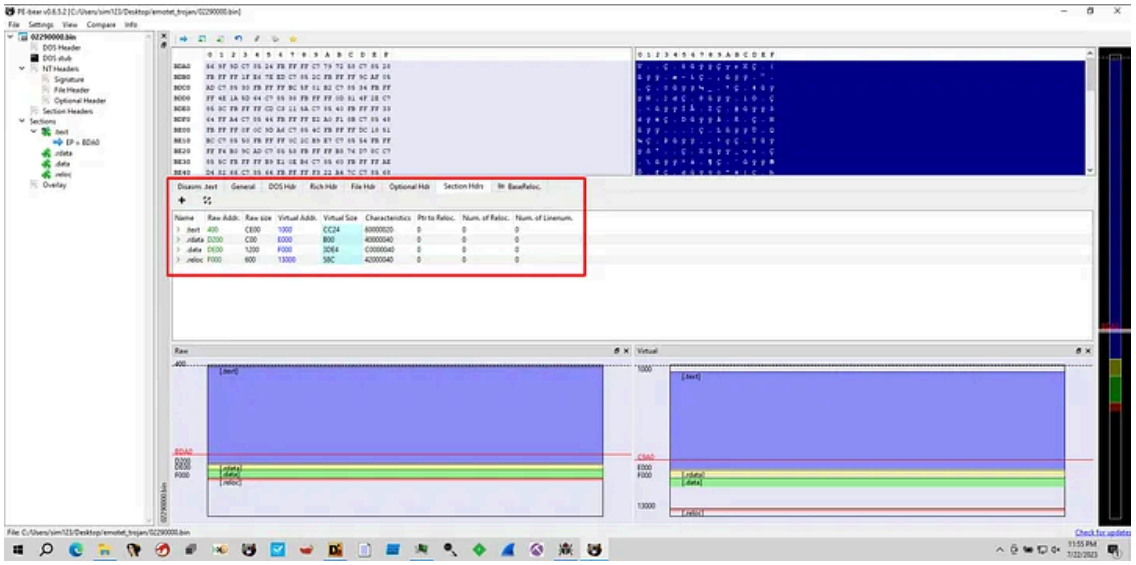


Unmapping

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**.

Press enter or click to view image in full size



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

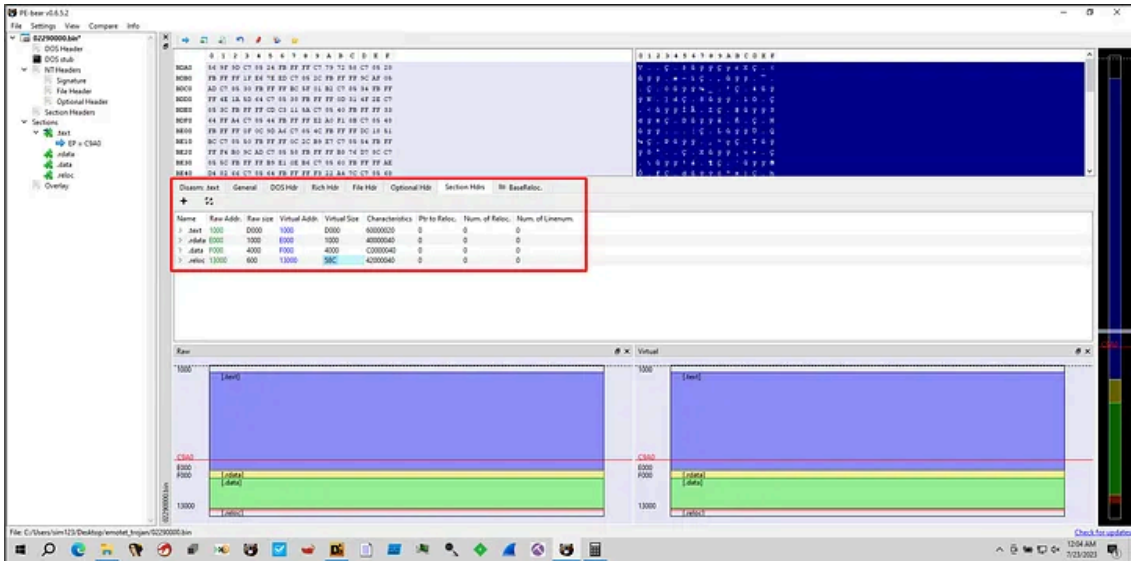
The unmap process is simples. It will take 4 steps:

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**).

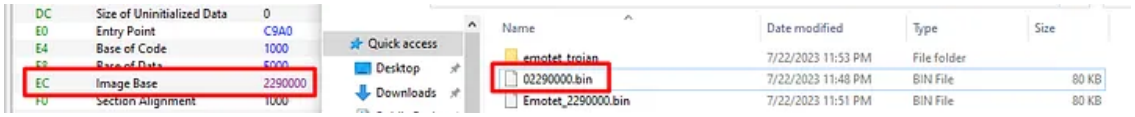
> You can subtract those values with the windows calculator and the “**programmer**” option.

After that, we have the Unmapped Binary:

Press enter or click to view image in full size



Press enter or click to view image in full size



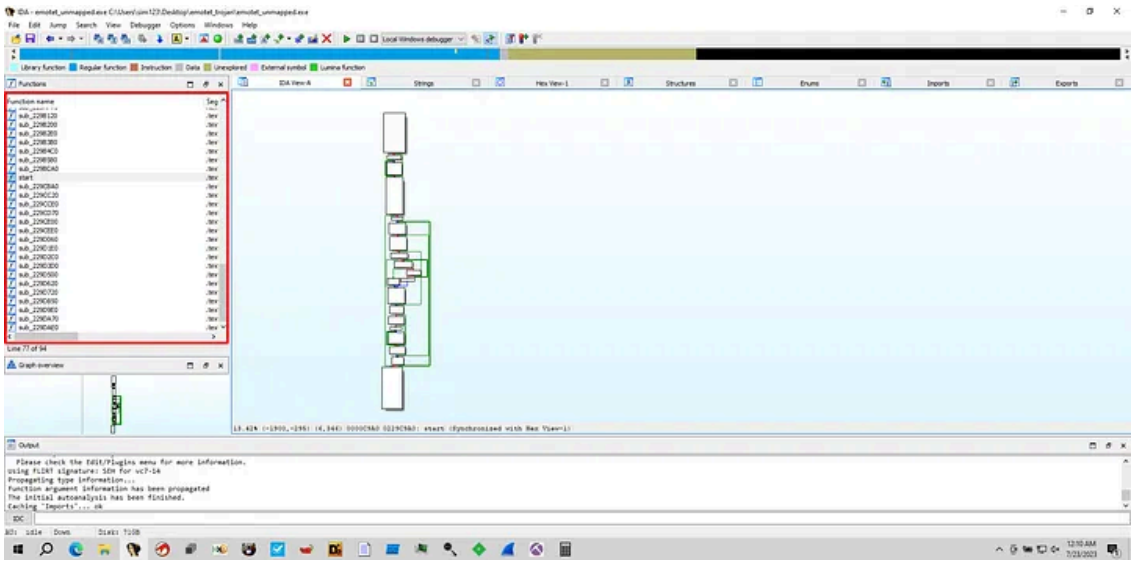
Now, all we have to do is save the new binary.

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

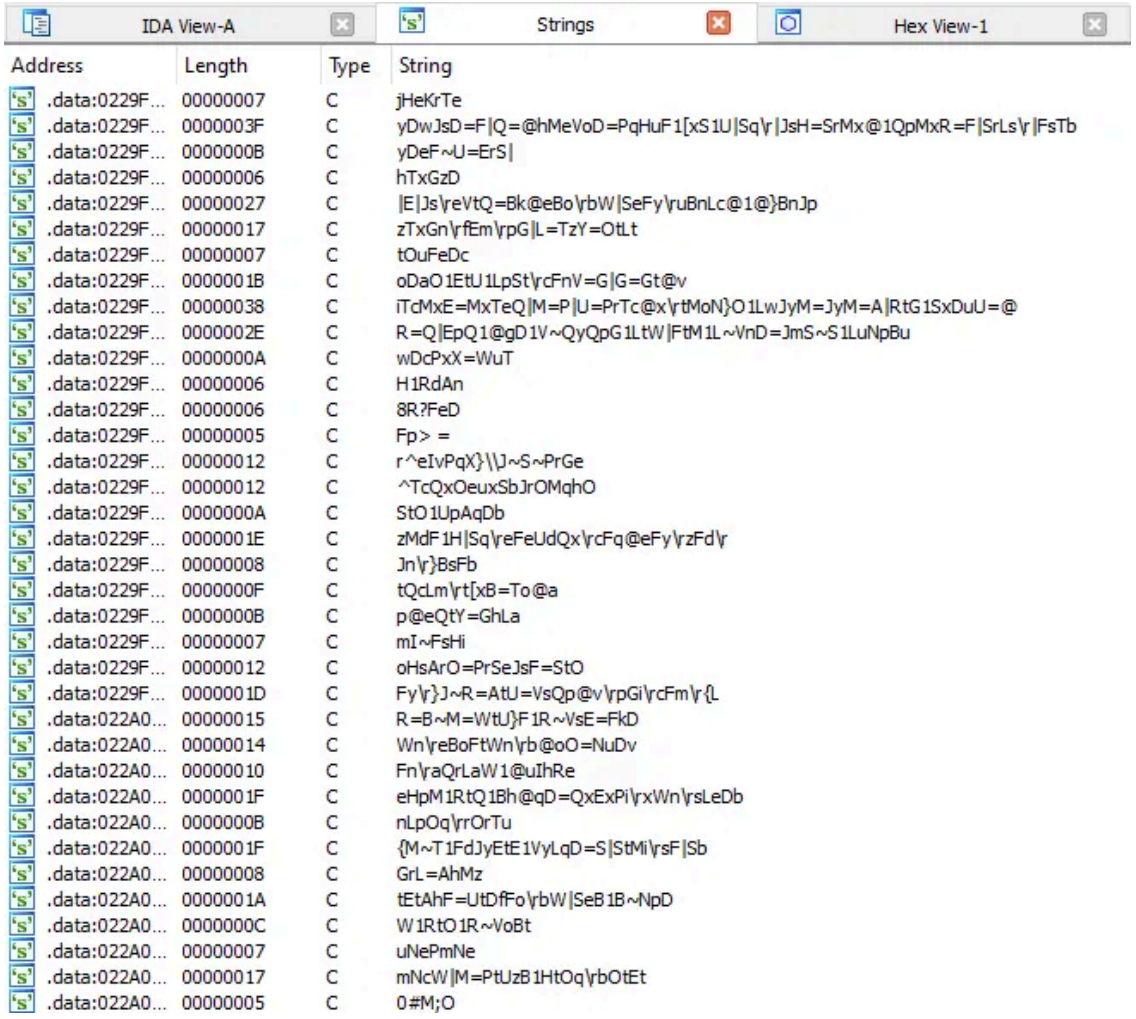
Confirming

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

Press enter or click to view image in full size



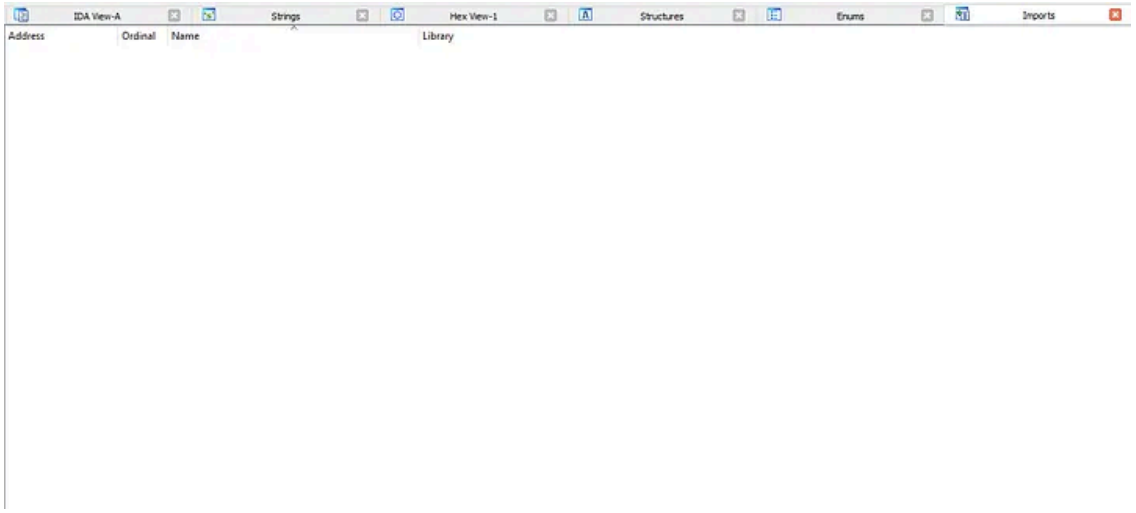
As you notice, now we have plenty of functions to analyze, and assuming by it's strings and imports, the malware is unpacked but has a severe method of **encryption** and **dynamic linking**.



IDA View-A | Strings | Hex View-1

Address	Length	Type	String
.data:0229F...	00000007	C	jHeKrTe
.data:0229F...	0000003F	C	yDwJsD=F|Q=@hMeVoD=PqHuF1[xS1U|Sq\r|JsH=SrMx@1QpMxR=F|SrLs\r|FsTb
.data:0229F...	0000000B	C	yDeF~U=ErS|
.data:0229F...	00000006	C	hTxGzD
.data:0229F...	00000027	C	|E|Js\reVtQ=Bk@eBo\rbW|SeFy\ruBnLc@1@}BnJp
.data:0229F...	00000017	C	zTxGn\rfEm\rpG|L=TzY=OtLt
.data:0229F...	00000007	C	tOuFeDc
.data:0229F...	0000001B	C	oDaO1EtU1LpSt\rcFnV=G|G=Gt@v
.data:0229F...	00000038	C	iTcMxE=MxTeQ|M=P|U=PrTc@x\rtMoN}O1LwJyM=JyM=A|RtG1SxDuU=@
.data:0229F...	0000002E	C	R=Q|EpQ1@gD1V~QyQpG1LtW|FtM1L~VnD=JmS~S1LuNpBu
.data:0229F...	0000000A	C	wDcPxX=WuT
.data:0229F...	00000006	C	H1RdAn
.data:0229F...	00000006	C	8R?FeD
.data:0229F...	00000005	C	Fp> =
.data:0229F...	00000012	C	r^eIvPqX}\\J~S~PrGe
.data:0229F...	00000012	C	^TcQxOeuxSbJrOMqhO
.data:0229F...	0000000A	C	StO1UpAqDb
.data:0229F...	0000001E	C	zMdF1H|Sq\reFeUdQx\rcFq@eFy\rzFd\r
.data:0229F...	00000008	C	Jn\r}BsFb
.data:0229F...	0000000F	C	tQcLm\rt[xB=To@a
.data:0229F...	0000000B	C	p@eQtY=GhLa
.data:0229F...	00000007	C	mI~FsHi
.data:0229F...	00000012	C	oHsArO=PrSeJsF=StO
.data:0229F...	0000001D	C	Fy\r}J~R=AtU=VsQp@v\rpGi\rcFm\r{L
.data:022A0...	00000015	C	R=B~M=WtU}F1R~VsE=FkD
.data:022A0...	00000014	C	Wn\reBoFtWn\rb@oO=NuDv
.data:022A0...	00000010	C	Fn\raQrLaW1@uIhRe
.data:022A0...	0000001F	C	eHpM1RtQ1Bh@qD=QxExPi\rxWn\rsLeDb
.data:022A0...	0000000B	C	nLpOq\rrOrTu
.data:022A0...	0000001F	C	{M~T1FdJyEtE1VyLqD=S|StMi\rsF|Sb
.data:022A0...	00000008	C	GrL=AhMz
.data:022A0...	0000001A	C	tEtAhF=UtDfFo\rbW|SeB1B~NpD
.data:022A0...	0000000C	C	W1RtO1R~VoBt
.data:022A0...	00000007	C	uNePmNe
.data:022A0...	00000017	C	mNcW|M=PtUzB1HtOq\rbOtEt
.data:022A0...	00000005	C	0#M;O

Press enter or click to view image in full size



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

Source: https://infosecwriteups.com/unpacking-emotet-trojan-dac7e6119a0a