Malware Analysis Series (MAS): Article 3

by Alexandre Borges
release date: MAY/05/2022 | rev: A

1. Introduction

Welcome to the third article in the ***MAS (Malware Analysis Series).*** After two articles that, hopefully, provided you with information for an initial foundation and motivation about malware analysis, so let's move forward to learn other interesting aspects of malicious Windows binaries from well-known samples, which are available to download from public sandboxes.

While I'm not sure whether you've read or not the first two articles, you can get them from the following links below:

- **MAS_1**: https://exploitreversing.com/2021/12/03/malware-analysis-series-mas-article-1/
- **MAS_2:** https://exploitreversing.com/2022/02/03/malware-analysis-series-mas-article-2/

I will not review all concepts presented in my last two articles and, if necessary, so I recommend reading them when it's possible. Of course, in practical terms and over the time, several techniques and approaches already explained will be repeated over and over again to provide you with more experience on the proposed topics.

I received several questions from professionals who have asked me about the purpose of this series, so it's time to make it clear: the purpose is to show several malware analysis techniques, approaches, contexts and concepts associated with the topic, as already mentioned in previous articles.

On lab setup, readers could use the procedure of the lab setup and tools that I mentioned in the last two article, and just if need be, so I'll point out any tool that we haven't used previously. Please, in case you need it, I recommend that you read the previous articles in this series.

Anyway, before proceeding, it's recommended to take a snapshot of your virtual machines and turn off any network communication and shared folders. While we aren't handling a ransomware case, avoid exposing your virtual machines to the local network when analyzing malware samples. Additionally, I'll be using **REMnux** and **Windows 8.1/10 (64 bits)** to perform any analysis. Thus, if you have the configured lab proposed in the last article, so you can re-use it.

Now we're ready to start our analysis.

This time, we are analyzing this sample:

SHA 256: **ed22dd68fd9923411084acc6dc9a2db1673a2aab14842a78329b4f5bb8453215**

You can easily get it by using **Malwoverview** (https://github.com/alexandreborges/malwoverview) and downloading it from **Malware Bazaar** as shown in the command below:

- **malwoverview.py -b 5 -B ed22dd68fd9923411084acc6dc9a2db1673a2aab14842a78329b4f5bb8453215**

2. Gathering information

As usual, our first steps are collecting enough information about the given malware threat. There're several tools to accomplish this task, so let's start by checking it against Virus Total:

```
remnux@remnux:~/malware/mas/mas_sample_3$ malwoverview.py -v 2 -V mas_3.bin -o 0

MD5 hash:            4024dad64d53d7f43fd00cdbc8d9519a
SHA1 hash:           7d5cd9062bb3c170efb190b673a77c33ed719ea6
SHA256 hash:         ed22dd68fd9923411084acc6dc9a2db1673a2aab14842a78329b4f5bb8453215

Malicious:           42
Undetected:          22

AV Report:
                     Avast:          Win32:BankerX-gen [Trj]
                     Avira:          TR/AD.Nekark.ljbel
                     BitDefender:    Trojan.Agent.FUCX
                     DrWeb:          Trojan.Emotet.1156
                     Emsisoft:       Trojan.Emotet (A)
                     ESET-NOD32:     Win32/Emotet.CV
                     F-Secure:       CLEAN
                     FireEye:        Generic.mg.4024dad64d53d7f4
                     Fortinet:       W32/Emotet.1156!tr
                     Kaspersky:      HEUR:Trojan-Banker.Win32.Emotet.gen
                     McAfee:         Emotet-FTG!4024DAD64D53
                     Microsoft:      Trojan:Win32/Emotetcrypt.IE!MTB
                     Panda:          Trj/GdSda.A
                     Sophos:         Mal/Generic-R + Troj/Emotet-CYP
                     TrendMicro:     TrojanSpy.Win32.EMOTET.YXCCLZ
                     ZoneAlarm:      CLEAN

Overlay:             NO
```

[Figure 01] First evaluation of the malware sample against Virus Total using Malwoverview.

Great! Using the same **Malwoverview**, it's quite simple to search for our sample on **Triage** and gather further information as shows figures below:

```
remnux@remnux:~/malware/mas/mas_sample_3$ malwoverview.py -x 1 -X ed22dd68fd9923411084acc6dc9a2db167
3a2aab14842a78329b4f5bb8453215 -o 0

                                    TRIAGE OVERVIEW REPORT

-----------------------------------------------------------------------------------------------

id:        220323-plrq2aeab7
status:    reported
kind:      file
filename:  ed22dd68fd9923411084acc6dc9a2db1673a2aab14842a78329b4f5bb8453215
submitted: 2022-03-23T12:25:20Z
completed: 2022-03-23T12:40:26Z
-----------------------------------------------------------------------------------------
```

[Figure 02] Determine the task ID on Triage using Malwoverview

```
remnux@remnux:~/malware/mas/mas_sample_3$ malwoverview.py -x 2 -X 220323-plrq2aeab7 -o 0

                                          TRIAGE SEARCH REPORT
----------------------------------------------------------------------------------------------------

score:      10
extracted:
            botnet:   Epoch5
            c2:
                      51.75.33.122:443
                      186.250.48.5:80
                      168.119.39.118:443
                      207.148.81.119:8080
                      194.9.172.107:8080
                      139.196.72.155:8080
                      78.47.204.80:443
                      159.69.237.188:443
                      45.71.195.104:8080
                      54.37.106.167:8080
                      185.168.130.138:443
                      37.44.244.177:8080
                      185.184.25.78:8080
                      185.148.168.15:8080
                      128.199.192.135:8080
                      37.59.209.141:8080
                      103.41.204.169:8080
                      185.148.168.220:8080

            family:   emotet
            key:      eck1_key
            value:    -----BEGIN PUBLIC KEY-----
                      MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAE2DWT12OLUMXfzeFp+bE2AJubVDsW
                      NqJdRC6yODDYRzYuuNL0i2rI2Ex6RUQaBvqPOL7a+wCWnIQszh42gCRQlg== -----END PUBLIC
                      KEY-----
            key:      ecs1_key
            value:    -----BEGIN PUBLIC KEY-----
                      MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAE9C8agzYaJ1GMJPLKqOyFrlJZUXVI
                      lAZwAnOq6JrEKHtWCQ+8CHuAIXqmKH6WRbnDw1wmdM/YvqKFH36nqC2VNA== -----END PUBLIC
                      KEY-----
            rule:     Emotet4
            dumped:   memory/1680-57-0x0000000000210000-0x0000000000237000-memory.dmp
            resource: behavioral1/memory/1680-57-0x0000000000210000-0x0000000000237000-memory.dmp
            tasks:    behavioral1 behavioral2

id:         220323-plrq2aeab7
target:     ed22dd68fd9923411084acc6dc9a2db1673a2aab14842a78329b4f5bb8453215
size:       700416
md5:        4024dad64d53d7f43fd00cdbc8d9519a
sha1:       7d5cd9062bb3c170efb190b673a77c33ed719ea6
sha256:     ed22dd68fd9923411084acc6dc9a2db1673a2aab14842a78329b4f5bb8453215
completed:  2022-03-23T12:40:26Z
signatures:
            Emotet
            Suspicious behavior: EnumeratesProcesses
            Suspicious use of WriteProcessMemory

targets:
            family:   emotet
            iocs:
                      storesdk.dsx.mp.microsoft.com
                      api.msn.com
                      51.75.33.122
```

[Figure 03] Summarized information collected from Triage by using Malwoverview

Finally, we can try **Capa** from Mandiant (https://github.com/mandiant/capa/releases/tag/v3.2.0), which brings valuable information about the binary:

```
C:\Users\Administrador\Desktop\MAS\MAS_3>capa mas_3.bin
loading : 100%|
matching: 100%|                                              | 2742/2742 [00:19<00:00, 142.22
+------------------------+------------------------------------------------------------------------+
| md5                    | 4024dad64d53d7f43fd00cdbc8d9519a                                       |
| sha1                   | 7d5cd9062bb3c170efb190b673a77c33ed719ea6                               |
| sha256                 | ed22dd68fd9923411084acc6dc9a2db1673a2aab14842a78329b4f5bb8453215       |
| os                     | windows                                                                |
| format                 | pe                                                                     |
| arch                   | i386                                                                   |
| path                   | mas_3.bin                                                              |
+------------------------+------------------------------------------------------------------------+

+------------------------+------------------------------------------------------------------------+
| ATT&CK Tactic          | ATT&CK Technique                                                       |
|------------------------+------------------------------------------------------------------------|
| DEFENSE EVASION        | Modify Registry:: T1112                                                |
|                        | Obfuscated Files or Information:: T1027                                |
| DISCOVERY              | File and Directory Discovery:: T1083                                   |
|                        | Query Registry:: T1012                                                 |
| EXECUTION              | Shared Modules:: T1129                                                 |
+------------------------+------------------------------------------------------------------------+
```

[Figure 04] First information and MITRE tactics presented by Capa

```
+-----------------------------+------------------------------------------------------------------+
| MBC Objective               | MBC Behavior                                                     |
|-----------------------------+------------------------------------------------------------------|
| CRYPTOGRAPHY                | Encrypt Data::RC4 [C0027.009]                                    |
|                             | Generate Pseudo-random Sequence::RC4 PRGA [C0021.004]            |
| DISCOVERY                   | Application Window Discovery::Window Text [E1010.m01]            |
|                             | Code Discovery::Enumerate PE Sections [B0046.001]                |
| FILE SYSTEM                 | Delete File:: [C0047]                                            |
|                             | Move File:: [C0063]                                              |
|                             | Read File:: [C0051]                                              |
| OPERATING SYSTEM            | Registry::Create Registry Key [C0036.004]                        |
|                             | Registry::Delete Registry Key [C0036.002]                        |
|                             | Registry::Open Registry Key [C0036.003]                          |
|                             | Registry::Query Registry Value [C0036.006]                       |
| PROCESS                     | Allocate Thread Local Storage:: [C0040]                          |
|                             | Terminate Process:: [C0018]                                      |
+-----------------------------+------------------------------------------------------------------+

+-------------------------------------------------------+----------------------------------------+
| CAPABILITY                                            | NAMESPACE                              |
|-------------------------------------------------------+----------------------------------------|
| encrypt data using RC4 PRGA (2 matches)               | data-manipulation/encryption/rc4       |
| contain a resource (.rsrc) section                    | executable/pe/section/rsrc             |
| extract resource via kernel32 functions (6 matches)   | executable/resource                    |
| delete file                                           | host-interaction/file-system/delete    |
| get file size                                         | host-interaction/file-system/meta      |
| move file                                             | host-interaction/file-system/move      |
| read .ini file                                        | host-interaction/file-system/read      |
| read file on Windows                                  | host-interaction/file-system/read      |
| get graphical window text (3 matches)                 | host-interaction/gui/window/get-text   |
| allocate thread local storage                         | host-interaction/process               |
| terminate process (2 matches)                         | host-interaction/process/terminate     |
| query or enumerate registry value (2 matches)         | host-interaction/registry              |
| delete registry key                                   | host-interaction/registry/delete       |
| link many functions at runtime                        | linking/runtime-linking                |
| enumerate PE sections                                 | load-code/pe                           |
| parse PE header                                       | load-code/pe                           |
+-------------------------------------------------------+----------------------------------------+
```

[Figure 05] Gathering malware's capabilities information using Capa

So far we have the following important information:

- The binary seems to be **Emotet**.
- The botnet is **Epoch 5**.
- There's a **long list of C2 IP addresses** (the listing above was truncated).
- This Emotet sample seems to be using **Elliptic Curve Cryptography.**
- The composition of **EnumeratesProcess** + **WriteProcessMemory** can indicate that malware is **looking for a target process to perform code injection** (we need to confirm it later).
- There's an indication of the presence of **RC4 (symmetric algorithm)** encrypting information in .data section.
- The malware also enumerates **PE sections**.

There is an relevant point: part of the collected information so far might be associated to the packer itself, so the next step is to understand whether the malware is packed or not to be able to confirm some of these gathered facts.

3. Unpacking

Using **Die tool** (https://github.com/horsicq/Detect-It-Easy) to check further information on the sample, we have the following points:

- The sample was compiled using **MS Visual C++ 2005**.
- It includes **MFC library**.
- Apparently there is a simple **anti-debugging trick** (**IsDebugPresent( )**)
- The entropy is high for **.rsrc** and **.text sections**, but it isn't always a solid fact to confirm that the sample is packed.

Using **PE-Bear** (https://github.com/hasherezade/bearparser) we are able to collect further information:

- It's a **32-bit sample** (from **FileHdr tab**).

- It's a **DLL sample** (from **FileHdr tab**).

- It **doesn't statically load any DLL related to network communication using WinSock2, WinINet, COM and so on.** This fact might be strange because, unless it's a wiper, common malware threats usually establishes a network communication to its creator. Thus**, either the malware loads network API dynamically or it might be packed.**

- The malware imports some resource-related APIs, which could indicate that resources could contain some data configuration and other useful information. Some of these APIs are:

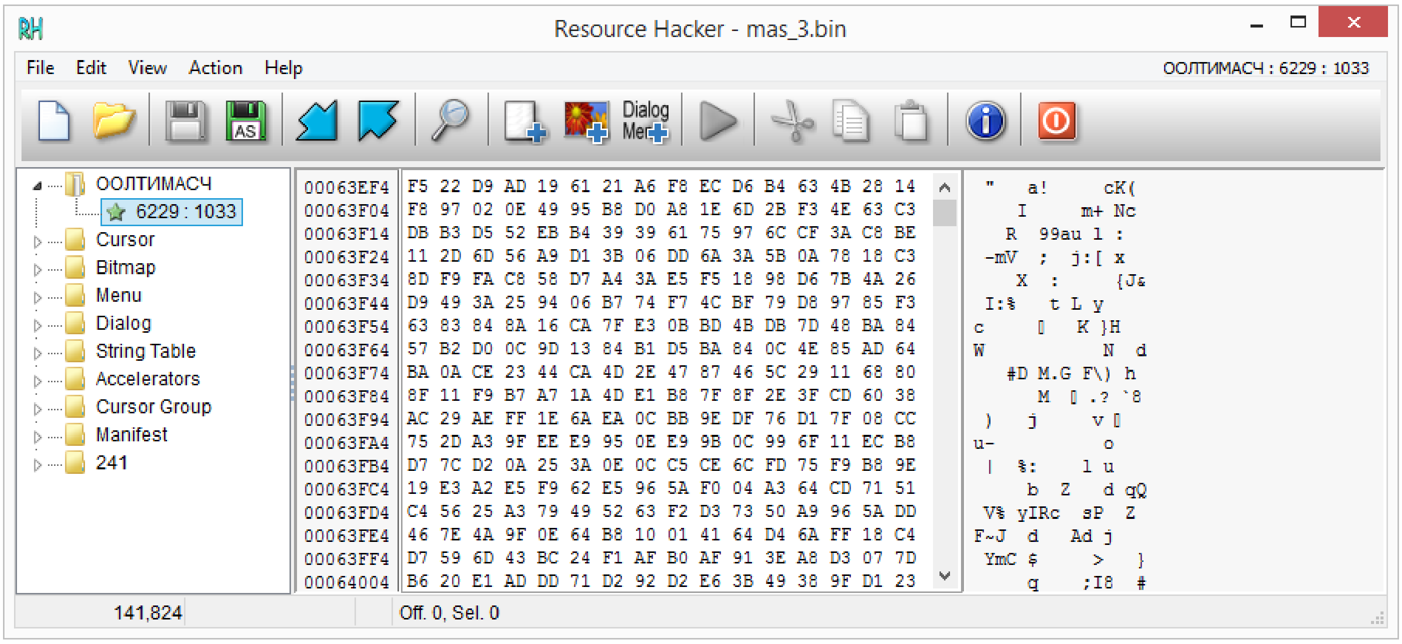
  - **FindResource**
  - **FindResourceExA**
  - **LoadResource**
  - **SizeOfResource**
  - **LockResource**

- **FreeResource**

- The malicious binary **exports two functions**:

  - **DllRegisterServer**
  - **DllUnregisterServer**

Using **Resource Hacker tool** (http://www.angusj.com/resourcehacker/), we confirm that there is some data within resources, but it might not have any relation to the real payload:



[Figure 06] Examining the resource content using Resource Hacker tool

So far, we aren't sure whether the sample is packed or not, so we have to use debugger to confirm it. Remember that it is a DLL, so we need to debug the **rundll32.exe** and provide, as argument, the DLL and one of the exported functions, which is the **DllRegisterServer( )** (function #1).

As I mentioned in the first article of this series, there're many ways to unpack malware samples, which some of them are semi-manual (using debuggers), automatic (**pe-sieve** and **hollows_hunter**) and even completely manual through scripts.

Whatever be your choice, start up your virtual machine (**Windows 8.1 or Windows 10**), open up the **x32dbg** (it's a 32-bit DLL -- https://x64dbg.com/) and load the **rundll32.dll** (**C:\Windows\SysWOW64\rundll32.exe**) for debugging. Go to **File → Change Command Line** and type a similar line, providing the DLL and the first exported function (or its respective ordinal number):

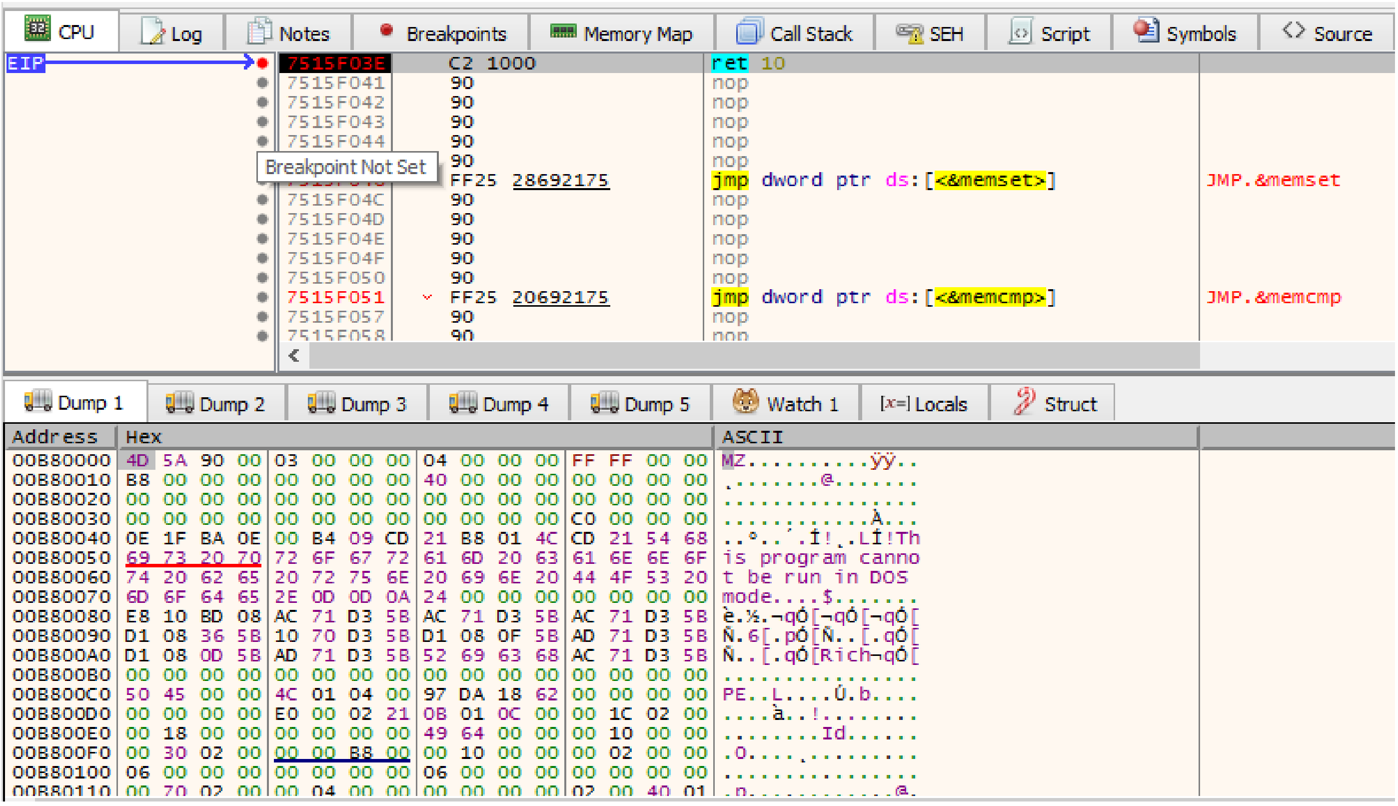
- **"C:\Windows\SysWOW64\rundll32.exe" C:\Users\Administrator\Desktop\MAS_3\mas_3.bin, #1**

Press **CTRL + F2** to reload the binary with the provided argument and, likely, the debugger had stopped on the **System Breakpoint**. **Play F5** once and you'll have stopped one the **Entry Point**.

Before proceeding, double-check to be sure that the virtual machine **doesn't have any shared folder** and **network communication with any system** (internal or external to your lab). **Typically I disable any network interface.**

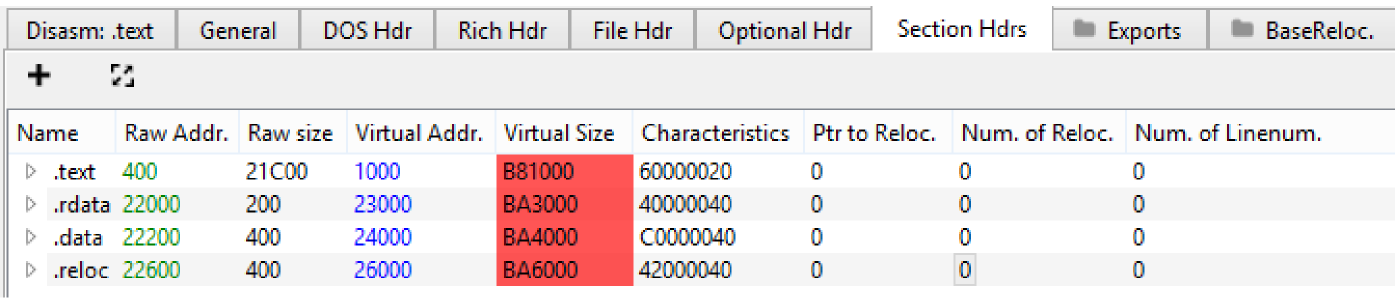
Let’s **set up breakpoints** on the following functions: **VirtualAlloc( )**, **WriteProcessMemory( )**, **CreateProcessInternalW( )** and **ResumeThread( )**

The breakpoint on the **VirtualAlloc( )** is going to be hit soon after the entry point, but take care: each section will be copied into this allocated one by one. In other words, you won’t have the entire malware at first hit, so pay attention to the addresses and, likely, there’re will be four or five hits in a row since the start to “complete” the entire “unpacked binary” in the memory. Right click the dump area and pick up “**Follow in Memory Map**”. **Right-click the memory region** and go to “**Dump Memory to File**”. Give a name and save the unpacked sample.



[Figure 07] Unpacking and extracting the PE binary during a x32dbg session

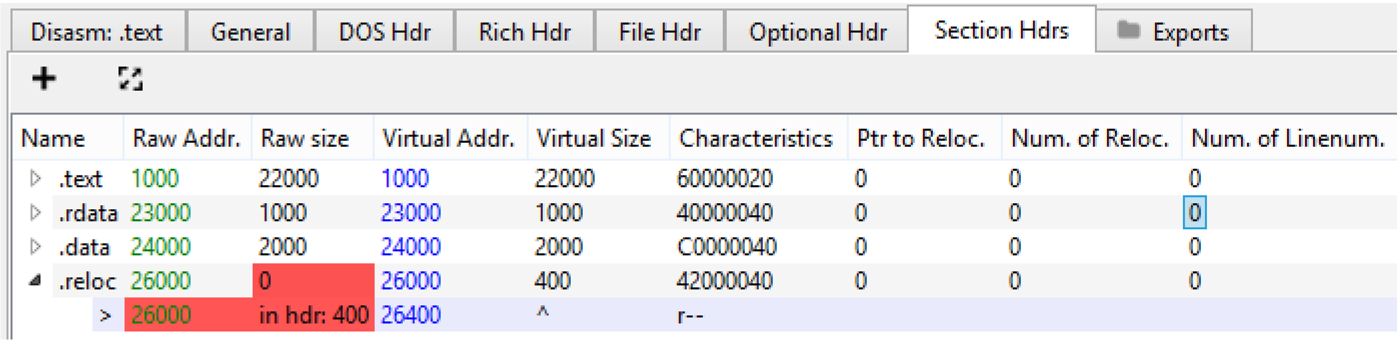
Open up the dumped sample in **PE-Bear** and you’ll notice that sections headers are messed up:



[Figure 08] Messed up section headers

As I mentioned in the second article of this series, you can fix them by copying the same values from **Virtual Address** to **Raw Address** (this is a **mapped file** and **.text section** starts at 0x1000), adjusting its sizes to keep the same sizes in **Raw Size** and **Virtual Size**. If you don’t understand how to do the math, it’s very simple:

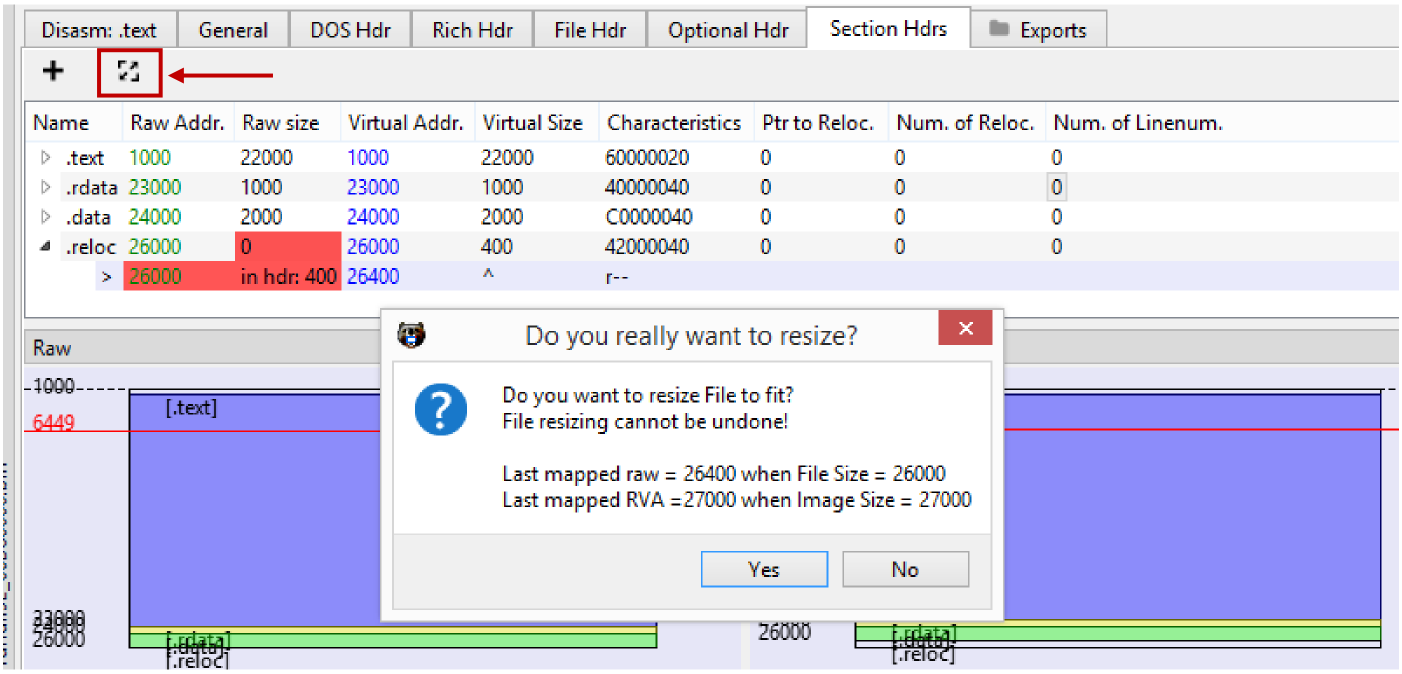
- **.rdata size - .text size == 23000 – 1000 == 22000, so fill .text size with this value.**
- **.data size - .rdata size == 24000 – 23000 == 1000, so fill .rdata size with this value.**
- **.reloc size - .data size == 26000 – 24000 == 2000, so fill .data size with this value.**



Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics	Ptr to Reloc.	Num. of Reloc.	Num. of Linenum.
.text	1000	22000	1000	22000	60000020	0	0	0
.rdata	23000	1000	23000	1000	40000040	0	0	0
.data	24000	2000	24000	2000	C0000040	0	0	0
.reloc	26000	0	26000	400	42000040	0	0	0
>	26000	in hdr: 400	26400	^	r--			

[Figure 09] Fixing section headers using PE-Bear

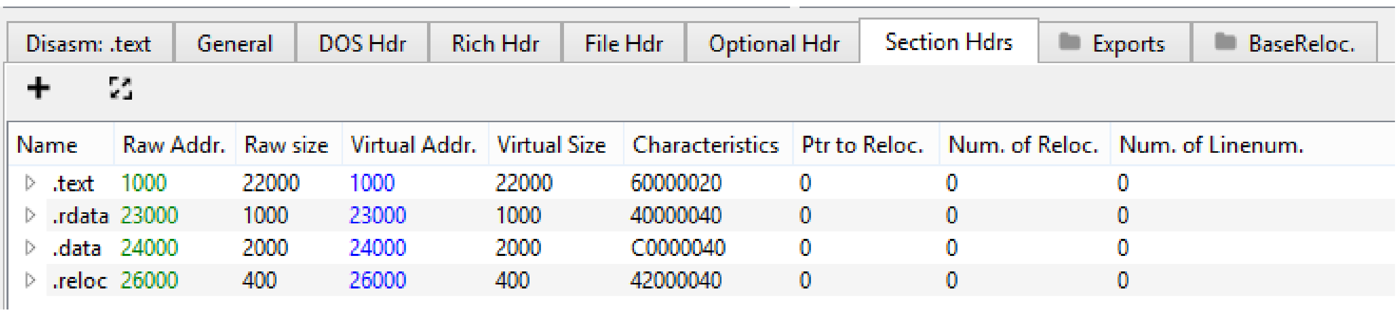
The final result is not clean but it can be managed by resizing the binary using the button pointed below:



Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics	Ptr to Reloc.	Num. of Reloc.	Num. of Linenum.
.text	1000	22000	1000	22000	60000020	0	0	0
.rdata	23000	1000	23000	1000	40000040	0	0	0
.data	24000	2000	24000	2000	C0000040	0	0	0
.reloc	26000	0	26000	400	42000040	0	0	0
>	26000	in hdr: 400	26400	^	r--			

[Figure 10] Resizing the PE binary through PE-Bear

If you open it up again **on PE-Bear** you'll have a clean binary in terms of section headers. Save the clean binary.



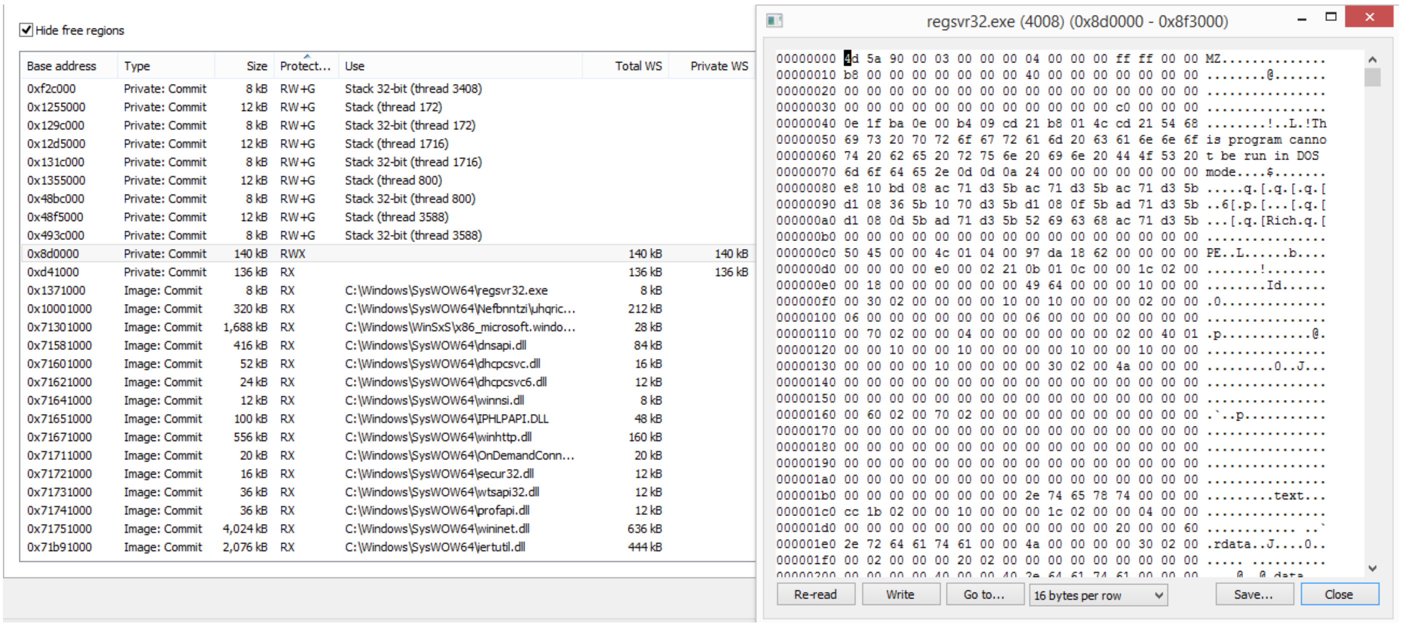
Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics	Ptr to Reloc.	Num. of Reloc.	Num. of Linenum.
.text	1000	22000	1000	22000	60000020	0	0	0
.rdata	23000	1000	23000	1000	40000040	0	0	0
.data	24000	2000	24000	2000	C0000040	0	0	0
.reloc	26000	400	26000	400	42000040	0	0	0

[Figure 11] Section headers list

Another simpler approach to unpack the malware is through **hollows_hunter** tool, which there're versions to x86 and x64 (https://github.com/hasherezade/hollows_hunter). In this case, you should:

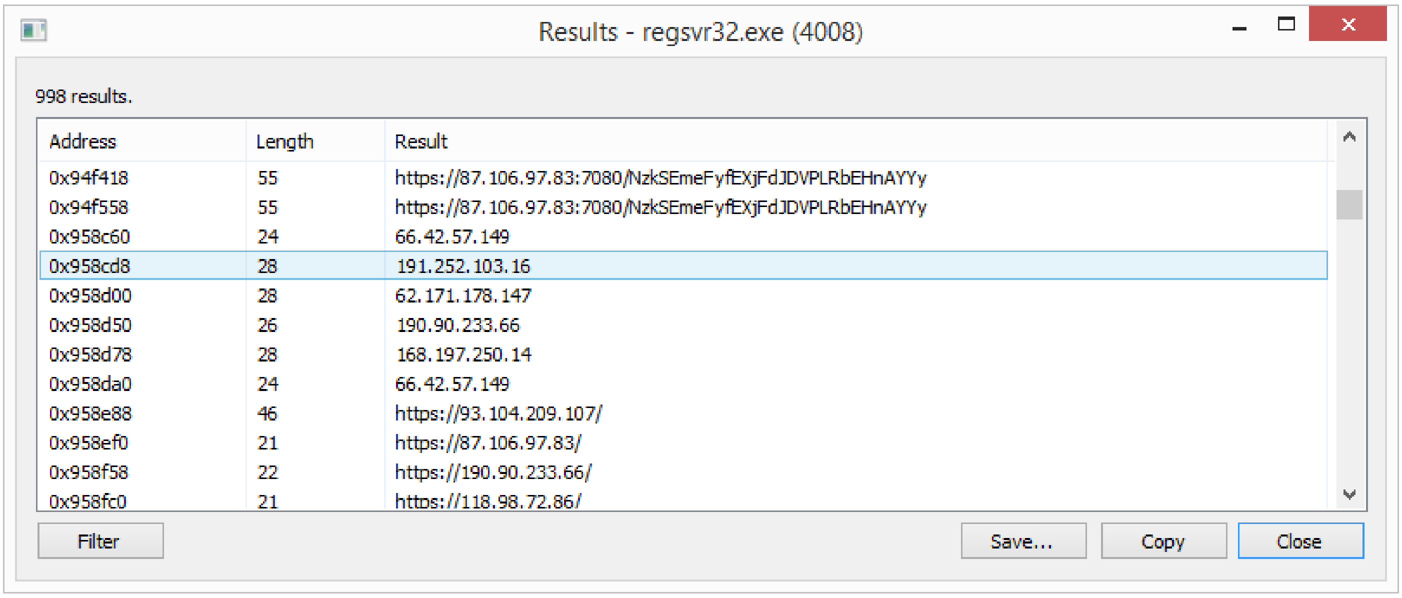
a. run **hollows_hunter64** in loop to ensure to catch any implant on memory: **hollows_hunter64 /loop**
b. run the malware: **rundll32 mas_3.bin,#1** **(take care:** the malware is going to remove itself, so keep a backup of it)

Hollows_hunter64 will provide two DLLs almost of same size, but you should prefer the larger one. Of course, you can observe the injected DLL in the the running regsvr32.exe process:



[Figure 12] Examining injected code on memory through Process Hacker

Listing strings and, afterwards, using a regex expression (**(?:(?:\d|[01]?\d\d|2[0-4]\d|25[0-5])\.){3}(?:25[0-5]|2[0-4]\d|[01]?\d\d|\d)(?:\/\d{1,2})?**) to search URLs on the memory of the potential malicious process makes possible to discover good information:



[Figure 13] Hunting URLs and IP addresses through regex on Process Hacker

4. Reversing

As usual, let's start our reversing session using **IDA Pro 7.7.x** and, just in case you don't have this version, you could follow the reversing session using **IDA Home 7.7.x.** (https://hex-rays.com/ida-home/). We're going to keep the focus on few objectives such as:

- **Renaming variables and functions.**
- **Decrypting strings**
- **Extracting C2 data configuration**
- **Handling hashed functions**
- **Extracting eventual public keys**
- **Fixing calling conventions whether it's necessary**
- **Creating C++ structures whether they are necessary and make our understanding easier.**

Differently from last article, my intention is not to enter in deep details and I'll try to keep this article short. Some professionals asked about reasons why I don't use dynamic analysis. Actually, it's a matter of personal preference for static analysis although I think dynamic analysis very useful and I also use it in several stages such as:

1. **Understanding a network protocol communication for writing an emulation script.**
2. **Confirming whether a given function of the malware works as I think it does.**
3. **Unpacking (in general).**
4. **Handling specific shellcode analysis.**
5. **Analyzing first stages in .NET format (next article).**

At same way, in several consulting services, I usually extract valuable information by performing memory analysis (**Volatility**) and gathering important indicators, information and artifacts such as:

a. **Created services (persistence)**
b. **Network communication information (C2)**
c. **Code injected (evasion)**
d. **Hooked functions (evasion)**
e. **Detecting callbacks (rootkits)**
f. **Unpacked binary (unpacking)**
g. **Created/Changed Registry entries (persistence)**

Certainly I could write a large section using memory analysis before starting the reversing phase, but this article would become so big and, eventually, it's a good opportunity to a near future.

During this section, we'll use the same IDA plugins presented in the second article of **Malware Analysis Series (MAS)**, though there're other good ones I'd like to show you in next articles of this series:

- **Flare Capa Explorer: http://github.com/mandiant/capa.git**
- **ApplyCalleeType: https://github.com/mandiant/flare-ida**
- **StructTyper: https://github.com/mandiant/flare-ida**
- **HashDB: https://github.com/OALabs/hashdb-ida**
- **Findcrypt-yara: https://github.com/polymorf/findcrypt-yara.git**

Please, if you don't know how to install all of these plugins, so read the second article of the this series where I showed further details about how to do it.

Open up the unpacked binary on **IDA Pro** and go to **View → Open Subviews → Type Libraries (SHIFT+F11 hotkey)** and insert important libraries **(INS hotkey)** such as:

- **mssdk_win7** (already inserted automatically)
- **ntapi or ntapi_win7**
- **ntddk_win7**
- **vc10 (not always)**

Although it is not necessary and doesn't make different in this article, it's always advisable to add some signatures, which will help you in most of reversing cases, by going to **View → Open Subviews → Signatures (SHIFT+F5)** and inserting (**INS hotkey**) few library modules such as:

- **vc32rtf**
- **vc32ucrt**
- **vcseh**

As we're going to use decompiler, it's also recommended to decompile the entire file first to avoid misunderstandings while analyzing code. Thus, go to **File → Produce File → Create C File (CTRL+F5)** and save the **.c file** in the same directory of the unpacked malware. The decompiling process take some seconds to finish. Now open up a **Pseudo Code window** and setup if side by side with the **Assembly View** window and synchronize it with the IDA View (**right click → Synchronize with**).

To collect contextualized information, go to **Edit → Plugins → Flare Capa Explorer** and starts the analysis of our first findings, but this time against the assembly code:

Rule Information	Address	Details
encode data using Base64 (2 matches)		data-manipulation/encoding/base64
function(sub_B81B09)	00B81B09	
function(sub_B995A8)	00B995A8	
encode data using XOR (4 matches)		data-manipulation/encoding/xor
basic block(loc_00B84C6C)	00B84C6C	
basic block(loc_00B8B124)	00B8B124	
basic block(loc_00B9ADC4)	00B9ADC4	
basic block(loc_00B9E25F)	00B9E25F	
hash data using murmur3		data-manipulation/hashing/murmur
function(sub_B963F0)	00B963F0	
parse PE header		load-code/pe
function(sub_B8F501)	00B8F501	

[Figure 14] Evaluating malware capabilities through Flare Capa Explorer on IDA Pro

Unfortunately we didn't get too much information, but we learned that:

- There's a parsing of a PE header, which used for hashing functions and shellcode.
- There're a possible **Base64 manipulation.**
- There're some **XOR operations**.
- Finally, a **subroutine (sub_B963F0)** might be using a hashing algorithm named **murmur3**.

Of course, the recommendation is to always check all information presented by **Flare Capa Explorer**, but whether the malware is really using a **hash function as murmur**, so we know that:

- It's a well know **non-cryptographic hash function.**
- Produces a **32-bit or 128-bit hash value.**
- We're able to find its **implementation in several programming languages** on the Internet.

There're other weird points about this sample:

- IDA Pro only shows **three strings (SHIFT+F12)**
- There isn't imported functions, so **possibly all of them are resolved dynamically**.
- There're the native **DLLEntryPoint( )** and only one user function exported: **DllRegisterServer( )**

As readers already know, strings usually offer a good guide along reversing tasks, but this time we don't have any one here. If we jump to **DllRegisterServer( )**, the first impression is not good because there're many **XOR** and **ADD operations** with hexadecimal numbers that, initially, we don't have any clue about what they are and do:

```
.text:00B8E1A9 ; HRESULT __stdcall DllRegisterServer()
.text:00B8E1A9                 public DllRegisterServer
.text:00B8E1A9 DllRegisterServer proc near          ; DATA XREF: .rdata:off_BA3028↓o
.text:00B8E1A9
.text:00B8E1A9 var_1C          = dword ptr -1Ch
.text:00B8E1A9 var_18          = dword ptr -18h
.text:00B8E1A9 var_14          = dword ptr -14h
.text:00B8E1A9 var_10          = dword ptr -10h
.text:00B8E1A9 var_C           = dword ptr -0Ch
.text:00B8E1A9 var_8           = dword ptr -8
.text:00B8E1A9 var_4           = dword ptr -4
.text:00B8E1A9
.text:00B8E1A9                 push    ebp
.text:00B8E1AA                 mov     ebp, esp
.text:00B8E1AC                 sub     esp, 1Ch
.text:00B8E1AF                 mov     [ebp+var_10], 713F6Eh
.text:00B8E1B6                 xor     edx, edx
.text:00B8E1B8                 add     [ebp+var_10], 39C2h
.text:00B8E1BF                 or      [ebp+var_10], 94D34AEEh
.text:00B8E1C6                 add     [ebp+var_10], 0FFFF3DC8h
.text:00B8E1CD                 xor     [ebp+var_10], 94F2B9C6h
.text:00B8E1D4                 mov     [ebp+var_14], 183E5Ah
.text:00B8E1DB                 imul    eax, [ebp+var_14], 50h
.text:00B8E1DF                 push    2Fh ; '/'
.text:00B8E1E1                 pop     ecx
.text:00B8E1E2                 push    2Ah ; '*'
.text:00B8E1E4                 mov     [ebp+var_14], eax
.text:00B8E1E7                 mov     eax, [ebp+var_14]
.text:00B8E1EA                 div     ecx
.text:00B8E1EC                 xor     edx, edx
.text:00B8E1EE                 mov     [ebp+var_14], eax
.text:00B8E1F1                 xor     [ebp+var_14], 2854E6h
.text:00B8E1F8                 mov     [ebp+var_1C], 0ED702Ah
.text:00B8E1FF                 xor     [ebp+var_1C], 0F23F01DCh
.text:00B8E206                 xor     [ebp+var_1C], 0F2D991B6h
.text:00B8E20D                 mov     [ebp+var_C], 0F86932h
.text:00B8E214                 imul    eax, [ebp+var_C], 49h
```

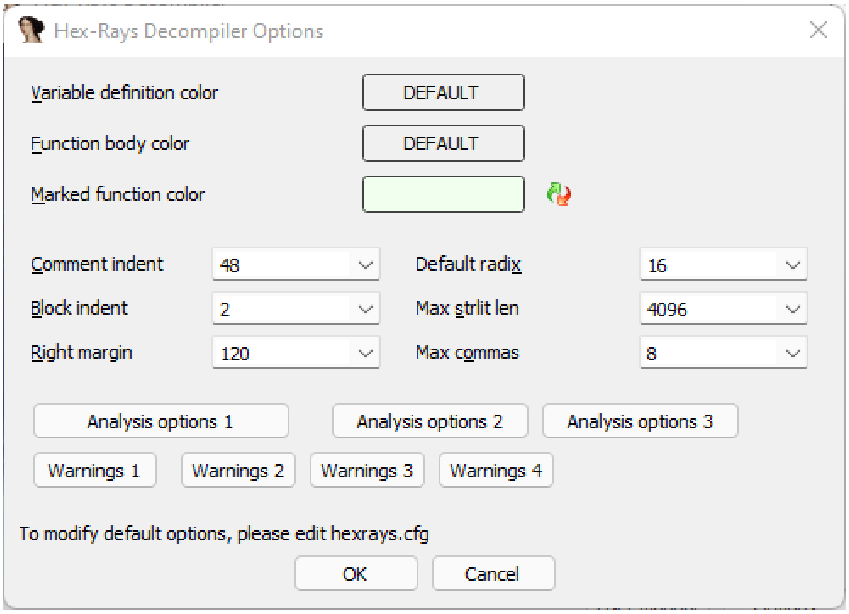
[Figure 15] Hexadecimal constant being manipulated (xor, add) against a structure

A next issue is that, on decompiler, most of constants are represented in decimal format instead of having them in hexadecimal format, as shown in **sub_B91FD0 method**:

```
                case 107845524:
                  v7 = sub_B8D75A();
                  result = sub_B81B09(v17, 436878, v7, 272094, 1, 971167, v18);
                  v0 = result != 0 ? 188355508 : 107845524;
                  break;
                case 112365796:
                  result = sub_B8960B();
                  if ( !result )
                    result = sub_B9C535();
LABEL_53:
                  v0 = 15476468;
                  break;
                case 114181113:
                  result = sub_B8B4FC();
                  v0 = 143923991;
                  break;
                case 125276383:
                  sub_B8E080();
                  v1 = 5411575;
                  result = sub_B8D763(v8, v9, v10, 4000);
                  v3 = result;
LABEL_49:
                  v0 = 44196330;
                  break;
                default:
                  goto LABEL_108;
              }
            }
```

[Figure 16] Constant represented as decimal instead of having them as hexadecimal

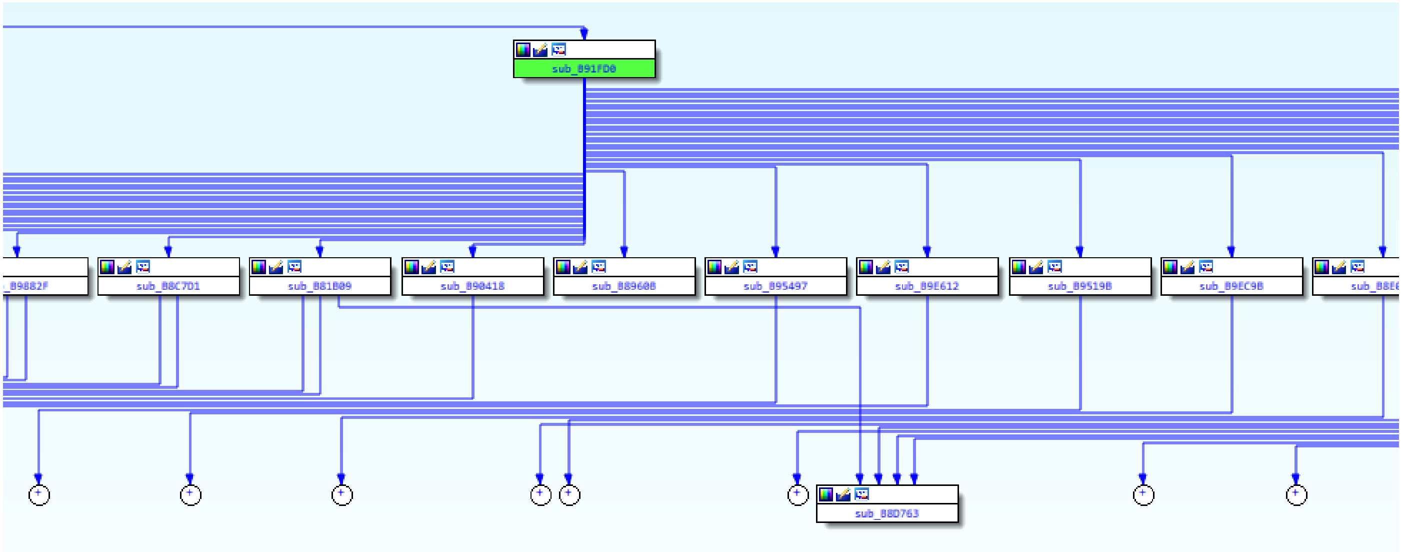
If it’s suitable, we can change decimal representation to hexadecimal representation by pressing **H hotkey**, but it takes so much time to do it in each decimal found over the code, so it’d better go to **Edit → Plugins → Hex-Rays Decompiler → Options** and **change the default radix from 0 to 16**, as shown below:



[Figure 17] Change Decompiler Representation

It'd recommended to produce a new C file again (**File → Produce File → Create C File (CTRL+F5)**) and, if you still see decimal representation, so just refresh the pseudo code representation by pressing **F5 hotkey**.

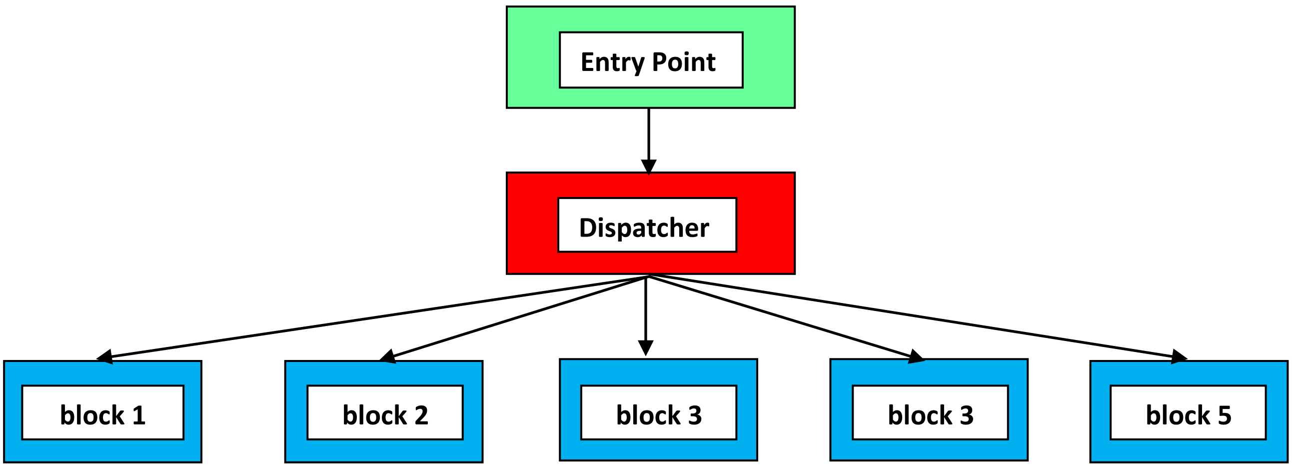
Maybe you might think that there're something very strange in **Figure 16** and, indeed, there's some obfuscation techniques being deployed. In you want to have an overview about what's happening, it's enough to get a graph (**View → Open SubViews → Proximity Browser**) to see "messed up" control flows:



[Figure 18] IDA Pro graph showing several decision branches

Unfortunately, **Emotet have used state variables**, which establishes the next piece of code to be executed. Additionally, the technique used for the Emotet and represented in the graph above is known as **Control Flow Flattening** (also known Code Flattening), which might be considered a sorted of state machine controlled by one or many state variables . In very few and imprecise words, **Control Flow Flattening** transforms a linear execution in a multi-branched execution. This technique is obviously used by many packers and, mainly, by modern protectors that virtualize functions. As examples, obfuscators such as **Obfuscator-LLVM** (https://github.com/obfuscator-llvm/obfuscator/wiki), malware like **FIN7** (https://malpedia.caad.fkie.fraunhofer.de/actor/fin7) and Emotet use this technique.

A simple representation about **Control Flow Flattening** would be:

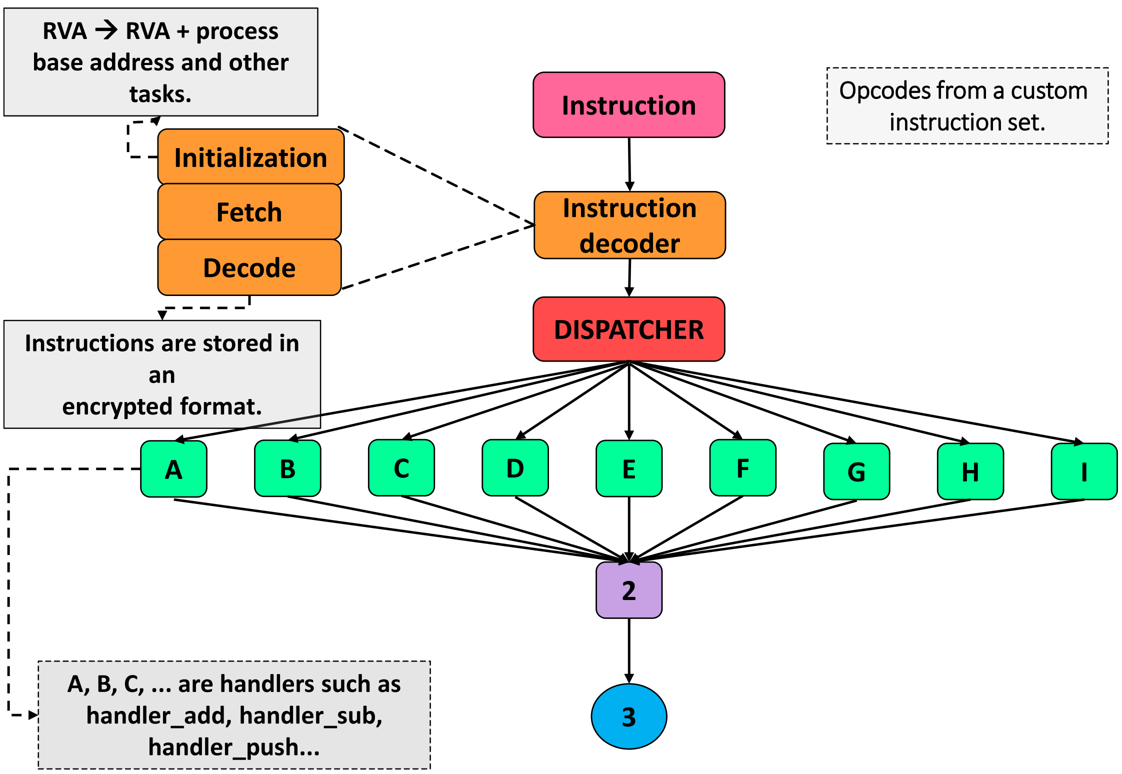


[Figure 19] Control Flow Flattening representation

As readers might have realized, depending on **entry point (state variable)** the dispatcher decides by execution of a different block. The concept of **Control-Flow Flattening** technique is also used for protectors that virtualize function's code. If you remember of first article of this series (**MAS**), modern obfuscators have some interesting characteristics:

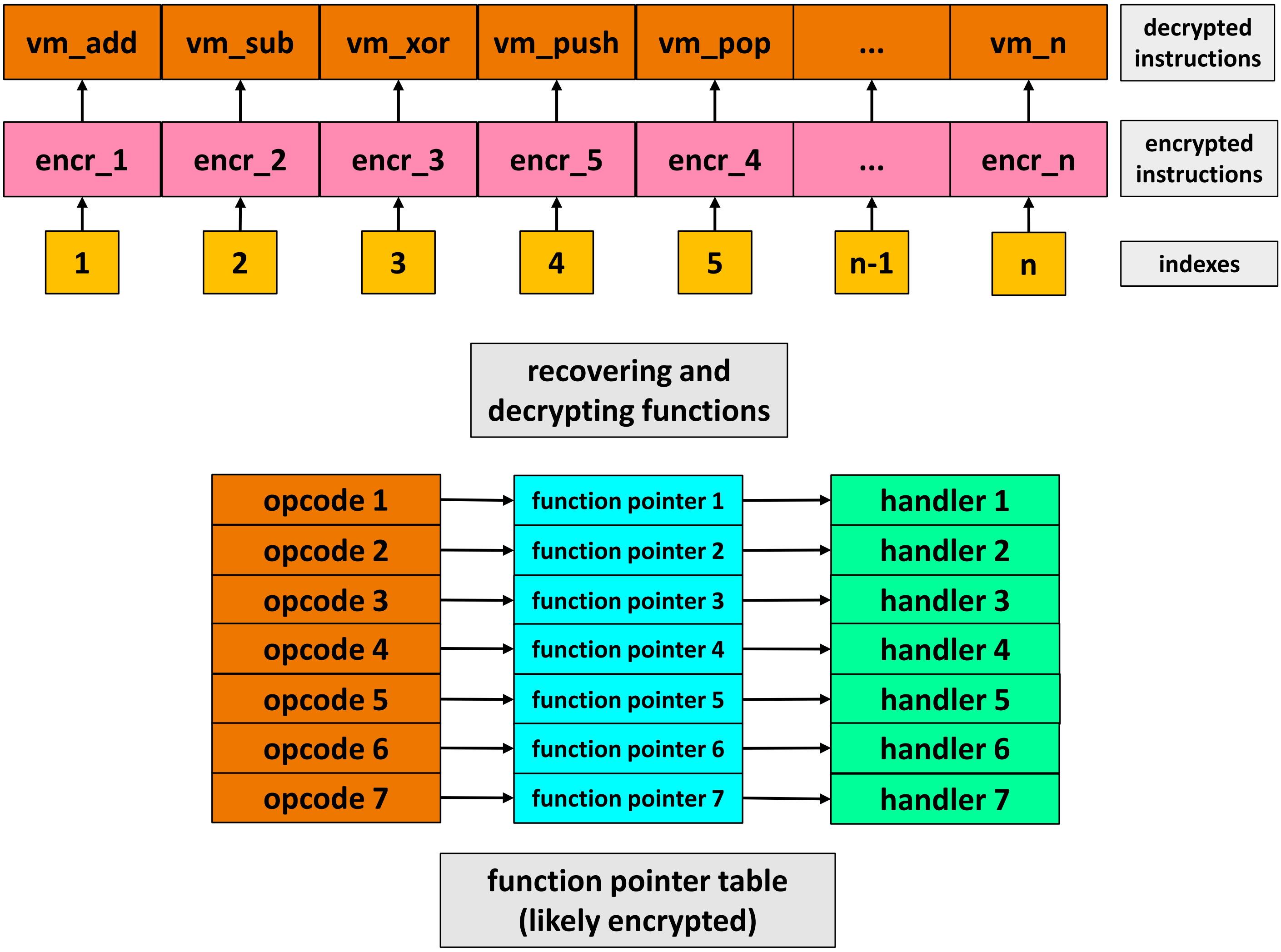
a. They have special **focus on 64-bit code** (but they some of them also cover 32-bit code).
b. **Not all instructions are virtualized.**
c. **Strings are encrypted** (obviously)
d. Native instructions are translated to virtualized ones **(RISC virtual machine instruction set).**
e. DLLs and APIs are **renamed or hashed.**
f. Obfuscation is **stack-based**.
g. There're **fake push instructions**.
h. They use **code re-ordering**.
i. There thousands of **dead-code instructions**.
j. They use **Control Flow Flattening**.
k. Virtualized code is polymorphic, so one **native instruction can be translated to many different virtualized instruction representations**, where one or another could be used anytime.
l. There are usually **critical context switch during the transition from native execution to virtualized execution and vice versa**.

The execution cycle is composed by **fetching, decoding** (translation from x86 to RISC context), **dispatcher** (depending on the instruction a determined handler is executed) and **handler** (the implementation of the virtual machine instruction set). Therefore, **given a decoded instruction, the dispatcher decides which handler will be executed**:



[Figure 20] Virtualized Instruction Execution

Only to supplement the previous explanation (it isn't related to Emoted sample), in cases of malware using virtualized instruction set, these **instructions are usually stored in an array (encrypted form),** and **to execute any virtualized instruction, an index is provided, which refers to array's slot**. So the **instruction is decrypted and the retrieved opcode points to a function pointer (handler) that's is finally executed**, as shown below:



[Figure 21] Virtualized Instruction Execution – part 2

Of course, this topic is really fascinating, but it's out of our scope to this article and there're lots of complex details involved in each showed concept. I've made an introductory presentation at **DEF CON China (2019)** and, just in case readers have curiosity in examining slides, they are available on:

- **abstract:** https://www.defcon.org/html/dc-china-1/dc-cn-1-speakers.html#Borges
- **slides:** https://exploitreversing.files.wordpress.com/2021/12/defcon_china_alexandre-2.pdf

Returning to our Emotet code analysis, we also have **Control Flow Flattening** and the dispatcher is represented as by a switch case construction and, depending on the state variable, a next block of code will be chosen to be executed.

The next picture is the **same function of Figure 15**, but **expanded and including further instructions**, where you can notice the mentioned **state variable**:

```
          {
            while ( v0 <= 0x7C1887A )
            {
              if ( v0 == 0x7C1887A )
              {
                result = sub_B9EBA2();
                v0 = 0x629CB8B;
              }
              else if ( v0 > 0x4F9319B )
              {
                if ( v0 > 0x65F58CA )
                {
                  switch ( v0 )
                  {
                    case 0x66D9794:
                      v7 = sub_B8D75A();
                      result = sub_B81B09(v17, 0x6AA8E, v7, 0x426DE, 1, 0xED19F, v18);
                      v0 = result != 0 ? 0xB3A13B4 : 0x66D9794;
                      break;
                    case 0x6B290E4:
                      result = sub_B8960B();
                      if ( !result )
                        result = sub_B9C535();
LABEL_53:
                      v0 = 0xEC26F4;
                      break;
                    case 0x6CE43F9:
                      result = sub_B8B4FC();
                      v0 = 0x8941B17;
                      break;
                    case 0x77790DF:
                      sub_B8E080();
                      v1 = 0x5292F7;
                      result = sub_B8D763(v8, v9, v10, 0xFA0);
                      v3 = result;
LABEL_49:
                      v0 = 0x2A261EA;
                      break;
                    default:
                      goto LABEL_108;
                  }
                }
                else
                {
                  switch ( v0 )
                  {
                    case 0x65F58CA:
                      result = sub_B8E080();
```

[Figure 22] Emotet Control Flow Flattening (state variable)

Readers can have realized that **v0** is the **state variable**, which is used in several lines of the code and, depending on its value, different **switch case instructions** determine the next block of code (functions and variable state attribution) to be executed.

Is it possible to improve the code representation? Yes, it's. Nonetheless, in my opinion, maybe it isn't worth to invest so much time to analyze this malware sample and we're able to proceed even handling this ugly code. Of course, we'll handle this scenario future articles.

Starting our analysis, we have only two calls inside **DllRegisterServer** function (exported):

- **sub_B91FD0**
- **sub_B8BA9C**

Going inside the first one (**sub_B91FD0**), there're many calls to subroutines and it is a large function. Anyway, there're some methods that could be interesting:

- **sub_B9ACFF (**called many times)
- **sub_B8B9D7** (called many times)
- **sub_B9D14C → sub_B84BB4** (called many times)
- **sub_B86A8D**
    - **sub_B9BFF0** (called many times)
    - **sub_B9B558** (PE parsing)
- **sub_BA1AE9** (DLL related)
- **sub_B9B558** (called many times)
- **sub_B86A8D** (called many times)

Please, I'd like to remember you that I'm showing real steps during a malware analysis because it'd very practical (and non-natural) to go to the "right functions" without providing a reasonable and rational explanation of taken decisions. Furthermore, I'm always focused on explaining how to accomplish the most important reversing steps instead only showing you the final reversed function, so be patient, please.

Certainly I won't reverse the entire malware sample in this article (not even close), but I hope I can show few relevant steps that could help you in your studies. Don't worry: this series (**MAS -- Malware Analysis Series**) will be composed by many articles and we have enough time to discuss different concepts, analysis and details related to reverse engineering and, mainly, malware analysis.

If reader are wondering how to get the number of cross references to each function call, so there two obvious alternatives:

- readers can manually parse each subroutine call and get its cross-references (**X hotkey**).
- readers can write a script to do it automatically.

```
import idautils
import idc

ea = 0xB91FD0

start = idc.get_func_attr(ea, FUNCATTR_START)
end_d = idc.get_func_attr(ea, FUNCATTR_END)
end = end_d -1

INSTR = [idaapi.NN_call]

for function_item in idautils.Functions():
    function_flags = idc.get_func_attr(function_item, FUNCATTR_FLAGS)
    if function_flags & FUNC_LIB or function_flags & FUNC_THUNK:
        continue
    myaddr = list(idautils.FuncItems(function_item))
    for addr in myaddr:
        if (start <= addr <= end):
            instruction = DecodeInstruction(addr)
            if instruction.itype in INSTR:
                print("0x%x %s\t%d" % (addr, idc.generate_disasm_line(addr, 0), len(list(XrefsTo(get_first_fcref_from(addr))))))

```

[Figure 23] Getting number of references using IDA Python/IDC

The result is shown below:

```
Python>
0xb93438 call    sub_B90418        1
0xb93474 call    sub_B95497        1
0xb934a5 call    sub_B86A8D        33
0xb934bc call    sub_B8B401        2
0xb934d9 call    sub_B8DA93        1
0xb93508 call    sub_B8D75A        2
0xb93523 call    sub_B84CB9        3
0xb93545 call    sub_B9D6B1        1
0xb9358a call    sub_B9D14C        1
0xb935c7 call    sub_B9ACFF        3
0xb935f9 call    sub_B9ACFF        3
0xb9362a call    sub_B9AFB0        1
0xb93660 call    sub_B8B9D7        50
0xb9367e call    sub_B8B9D7        50
0xb936e0 call    sub_B9E612        1
0xb936f6 call    sub_B9519B        1
0xb93714 call    sub_B9EC9B        1
0xb93737 call    sub_B8E080        2
0xb9377a call    sub_B8E080        2
0xb937ae call    sub_B8D763        15
0xb937d0 call    sub_B8B4FC        2
0xb937ed call    sub_B8960B        4
0xb937fa call    sub_B9C535        1
0xb93817 call    sub_B8D75A        2
0xb93848 call    sub_B81B09        1
0xb93875 call    sub_B9EBA2        1
0xb938d0 call    sub_B9DAD8        1
0xb938ed call    sub_B9B2FC        1
0xb93903 call    sub_B84700        1
0xb93933 call    sub_B9BAF2        1
0xb93970 call    sub_B8D763        15
0xb939aa call    sub_B8D763        15
0xb93a1c call    sub_B91DA6        1
0xb93a4c call    sub_B8D763        15
0xb93a91 call    sub_B830BE        2
0xb93aae call    sub_B8D79A        1
0xb93ad2 call    sub_B9C16B        1
0xb93af5 call    sub_B86CBB        1
0xb93b49 call    sub_B8BE09        1
0xb93b5c call    sub_B8960B        4
0xb93b7f call    sub_B866B0        1
0xb93b96 call    sub_B8960B        4
0xb93bba call    sub_B8960B        4
0xb93bdd call    sub_B9882F        1
0xb93c01 call    sub_B86A8D        33
0xb93c43 call    sub_B85995        1
0xb93c60 call    sub_B9158A        1
0xb93c8b call    sub_B987E3        1
0xb93cac call    sub_B89E7E        1
0xb93cc1 call    sub_B8C7D1        1
0xb93cd0 call    sub_B88C7C        1
```

[Figure 24] Result: number of references to each call instruction

Returning to our problem, few considerations follow below:

- At this first fast overview**, I wasn't concerned to examining methods being called from sub_B91FD0** (a matter of restricted time).
- My first goal was **finding methods being called many times** (reasons follow below).
- I **quickly inspected only one or other subroutine** and, when I notice something useful, so I wrote down.
- Obviously I lost many good functions and important details, but they don't matter for now.

This slopy approach is usual when I start an analysis because I don't know what expect for, but it could takes to the next step, so pay attention to the following key facts:

- We **have three strings (SHIFT+F12)** throughout the sample (**only two in .rdata and one in .data sections**), so it suggests **that there're one or more subroutines that perform string decryption.**
- We **don't have any explicit DLL or function name in the code**, so probably **there're one or more subroutines responsible for accomplishing this task.**
- Considering that **malware threats usually have many strings, and one or more related subroutines would be called to decode them, so likely these subroutines would be called several times.**
- At same way, **one or more methods would be called many times to decode the DLL name and API name**.
- **Strings, DLLs and API are usually stored in somewhere inside of sections.**
- Code involved with **PE parsing might be an additional indicator of hashing**.

Before proceeding, you can list the available segments (sections) of the malware in IDA Pro by going to **View → Open Subviews → Segments** or pressing **SHIFT + F7 hotkey**:

Segments

Name	Start	End	R	W	X	D	L	Align	Base	Type	Class
.text	00B81000	00BA3000	R	.	X	.	L	para	0001	public	CODE
.rdata	00BA3000	00BA4000	R	.	.	.	L	para	0002	public	DATA
.data	00BA4000	00BA6000	R	W	.	.	L	para	0003	public	DATA

Line 1 of 3

[Figure 25] Segments shown in the IDA Pro

The unpacked sample has only three sections and, curiously, it doesn't have a **.rsrc section**, so possible **strings, DLL names and API functions are encoded and stored in one of these ones available**. Let's go to **sub_B9D14C** (**third one in the list on page 18**) and check it inside:

```
25 LABEL_15:
26       i = (_DWORD *)dword_BA5084;
27     }
28     if ( v1 == 0x2B8A739 )
29     {
30       v5 = sub_B84BB4(0x2F57A, (int)dword_B814C8, 0x4964B);
31       v1 = 0x7D95F8C;
32       if ( !sub_B8D68B(0x3532, 0, 0, v5, 0xB480D, (int)&v7) )
33         v1 = 0x250F141;
34       sub_B8B9D7(0x7B146, 0xA8470, v5);
35       goto LABEL_15;
36     }
37     if ( v1 != 0x7483D93 )
38       break;
39     dword_BA5084 = sub_B9EAA3((void *)0x48);
40     *(_DWORD *)(dword_BA5084 + 0x2C) = 0x4000;
41     v3 = sub_B9EAA3(*(void **)(dword_BA5084 + 0x2C));
42     i = (_DWORD *)dword_BA5084;
43     v1 = 0x2B8A739;
44     v4 = v3 + *(_DWORD *)(dword_BA5084 + 0x2C);
45     *(_DWORD *)(dword_BA5084 + 0x28) = v3;
46     i[4] = v3;
47     i[9] = v3;
```

[Figure 26] Examining the suspicious sub_B9D14C subroutine

The call on **line 30** is interesting because it refers to **dword_B814C8** global variable, which represents a respective address. Checking this place we found out **it's located within the .text section**, as shown below:

```
.text:00B81454 dword_B81454    dd 123FA0BAh, 123FA0B0h, 7F53D2CFh, 7611CED5h, 712FCCD6h
.text:00B81454                                         ; DATA XREF: sub_B9158A+5D↓o
.text:00B81454                 dd 2DA68347h, 155014B1h, 43C1FC57h, 0C0740C97h, 6B8989B2h
.text:00B81454                 dd 168F181h, 0E4D70EB9h, 539B7214h, 3 dup(0)
.text:00B81494 dword_B81494    dd 25481C8Bh, 25481C87h, 443E78EAh, 177B75FBh, 492478A5h
.text:00B81494                                         ; DATA XREF: sub_B9158A+98↓o
.text:00B81494                 dd 270B9292h, 748ED22Ah, 75F9A6BDh, 82C9EB50h, 0E1000129h
.text:00B81494                 dd 4927BB31h, 908C6117h, 22FCDAB0h
.text:00B814C8 dword_B814C8    dd 2139FCACh, 2139FCAFh, 637EB2FEh, 0CB5CD192h
.text:00B814C8                                         ; DATA XREF: sub_B9D14C+46C↓o
.text:00B814D8 dword_B814D8    dd 14691030h, 14691078h, 253A5375h, 14691010h, 97733FC4h
.text:00B814D8                                         ; DATA XREF: sub_B863B8+7D↓o
.text:00B814D8                                         ; sub_B91FD0+160B↓o
```

[Figure 27] Address represented by dword_b814C8 global variable

That's a good result because we've confirmed that **some encrypted data related to string, API name or DLL name is stored there** (we don't know what's exactly) and, additionally, there're many other cross data references (**DATA XREF**) around the given address. If we expand our searching and look at start of the **.text section** by listing segments (**SHIFT + F7 or CTRL + S**) and **double-clicking .text section**, we have the following content shown in the figure below:

```
ext:00B81000 ; Segment type: Pure code
ext:00B81000 ; Segment permissions: Read/Execute
ext:00B81000 _text           segment para public 'CODE' use32
ext:00B81000                 assume cs:_text
ext:00B81000                 ;org 0B81000h
ext:00B81000                 assume es:nothing, ss:nothing, ds:_data, fs:nothing, gs:nothing
ext:00B81000 dword_B81000    dd 2A8EF14Bh, 2A8EF14Ch, 0FD1826Eh, 65D6C97Bh, 0B1DA0BC0h
ext:00B81000                                         ; DATA XREF: sub_BA03F2+2E5↓o
ext:00B81000                 dd 0D7E713DCh, 71CBCB8Bh, 8C1C27D1h, 1121D90Eh, 1121D904h
ext:00B81000                 dd 6552A07Dh, 7F48B46Bh, 9338B668h, 5A48BDCDh, 12E3F9C4h
ext:00B81000                 dd 12E3F9CBh, 779795AAh, 3DC38DB7h, 7B8F9AA0h, 0DDD98DB7h
ext:00B81000                 dd 0C2CFFEFAh, 0F7CBF5BBh, 87C7D9C2h, 1458368Fh, 14583682h
ext:00B81000                 dd 7B3B46E6h, 733150E1h, 783919AFh, 443BBBE3h, 0A7950C98h
ext:00B81000                 dd 65AFC13Fh, 9E8CF327h, 48E9A07Eh
ext:00B81084 dword_B81084    dd 2A59FE51h, 2A59FE5Ah, 0F778B74h, 5F7CD024h, 762CDB7Fh
ext:00B81084                                         ; DATA XREF: sub_BA225A+3BD↓o
ext:00B81084                 dd 39676CBDh, 212BE60Fh, 0AD325786h, 0E0377F13h, 3 dup(0)
ext:00B810B4 dword_B810B4    dd 34366A2Bh, 34366A23h, 191B6026h, 191B390Eh, 0EA391343h
ext:00B810B4                                         ; DATA XREF: sub_B995A8:loc_B9A353↓o
ext:00B810B4                 dd 0C14E99A6h, 0C699CFFBh, 0DAE1541Dh, 94F9B2D6h, 0B8242A40h
ext:00B810B4                 dd 0ADEC3499h, 0
ext:00B810E4 byte_B810E4     db 0Bh, 0EDh            ; DATA XREF: sub_BA27DF+2BF↓o
ext:00B810E6                 dw 2301h
ext:00B810E8                 dd 2301ED04h, 486E8248h, 33B8862h, 63C9E2Eh, 5D0BE078h
ext:00B810E8                 dd 7BAC3F1Bh, 2A5E8BD6h, 875848E5h, 634AF695h, 9EE43C5Ch
ext:00B810E8                 dd 0
ext:00B81114 dword_B81114    dd 319259BAh, 3192598Ah, 45FC36F9h, 1CE637DFh, 54E220EEh
ext:00B81114                                         ; DATA XREF: sub_B995A8+C2B↓o
```

[Figure 28] Possible encrypted data at start of the .text section

That's great! We just have learned **that there're more encrypted data (byte representation)** at **beginning of .text section** and **associated data cross references (DATA XREF)** to each one of these bytes. According to our previous analysis, we've found that one of these references is **sub_B84BB4 subroutine** (**line 30, Figure 26**), which has the following content as first instructions:

```
17  nullsub_1(a2, a1, a3);
18  v4 = (char *)(v3 + 2);
19  v5 = *v3 ^ v3[1];
20  v15 = *v3;
21  v16 = v5;
22  v6 = v5 + 1;
23  if ( (((_BYTE)v5 + 1) & 3) != 0 )
24    v6 = ((v5 + 1) & 0xFFFFFFFC) + 4;
25  v7 = sub_B9EAA3((void *)(2 * v6));
26  if ( v7 )
27  {
28    v8 = &v4[4 * (v6 >> 2)];
29    v9 = (_WORD *)v7;
30    v10 = (unsigned int)(v8 - v4 + 3) >> 2;
31    if ( v4 > v8 )
32      v10 = 0;
33    if ( v10 )
```

[Figure 29] Possible decrypter (sub_B84BB4 subroutine) of referenced data

On line 19 there's an interesting instruction involving an **XOR operation**, which it's a good indication we're handling the "decrypting" subroutine.

Before renaming variables and methods, we have the following context from line 18 onward:

- **v3** seems to be an array of bytes.
- **On line 18**, **(char *)(v3 + 2)** points 8 positions ahead. This value is associated to **v4. Additionally, the cast to (char *) is our strong indication that v4 represents the decrypted string.**
- **One line 19**, the **first four bytes are XOR'd with the next four bytes (*v3 ^ *v3[1])**, and stored into **v5**.
- Notice that, on **line 20, v15 is set with v3 content, so *v15 = *v3.**
- If you examine the remaining of the subroutine below, **v12 is set to v4's content, so *v12 = *v4.**
- **On line 39, v15 (holding v3 content) is XORed with v12 (holding the v4 content).Therefore, so far v3 (first 4 bytes) seems to be the key and v12 (v4) seems to be the encrypted content.**
- **What's the encrypted data's length? Probably it's the *v3[1], but the real value is hidden under a XOR operation (line 19), so we have to execute this XOR operation before getting the real length.**

```
34      {
35        for ( i = 0; i < v10; ++i )
36        {
37          v12 = *(_DWORD *)v4;
38          v4 += 4;
39          v13 = v15 ^ v12;
40          *v9 = (unsigned __int8)v13;
41          v9 += 4;
42          *(v9 - 3) = BYTE1(v13);
43          v13 >>= 16;
44          *(v9 - 2) = (unsigned __int8)v13;
45          *(v9 - 1) = BYTE1(v13);
46        }
47        v5 = v16;
48      }
49      *(_WORD *)(v7 + 2 * v5) = 0;
50    }
51    return v7;
52  }
```

[Figure 30] Sub_B84BB4 subroutine of referenced data (second part)

Therefore, at end, we have:

- **The decrypted stuff is given by: *v3 ^ *(v3 + 2), where *(v3 + 2)'s size is given by (*v3 ^ *v3[1])**.
- Another good hint that the operation **(*v3 ^ *v3[1])** is probably the wished length is provided by the **line 22 (v6 = v5 + 1)** from **Figure 29**, where the operation is adding one because the size of the end of the string **('\x00').**
- The data format is: **[xor key] (4 bytes) + [xored string's length] + [encrypted string]**, where the actual (plain text) string length is given by **(*v3 ^ *v3[1]).**

Based on this interpretation, we can write a simple script in **Python 3** to try to emulate exactly these steps. Additionally, in the second part of this script, once we got the decrypted information (likely strings), we can make comments within IDA idb file using the result as content of such comments. Summarizing our next steps, it's necessary to:

- **Read the encrypted data** from file.
- **Create a variable** holding the first dword **(key).**
- **Create a variable** holding the **second dword (xored string's length).**

- Perform a **XOR operation** between key and the resulting **xored string length**. It will be the plain text string’s length.
- Use the **key to decode the encrypted data from byte 8 onward**.
- At a second moment, alter the script to **create comments next to referring instructions.**

Once again, readers can use any development program or environment to write their Python scripts and one of available options would be to use **Jupyter notebook** to make drafts while programming because it offers good debugging messages and support, which are useful mainly at this drafts. To install and use it, execute the following steps:

1. **pip install jupyterlab**
2. **execute: jupyter-lab**
3. Choose **Python 3 Notebook** (right side)
4. **Rename** the document (left side)

As I’m going to use **IDC/IDA Python functions**, so I will be using the own IDA Pro script environment available in **File → Script Command (SHIFT+F2).** The following script is well-commented, but I’ll leave some additional comments after it:

```python
import binascii
import pefile
import struct
import idautils
import idc

# This routine implements the XOR operation and take the key's size into account.
# In this sample, we're providing the XOR key (first 4 bytes), the data string
# (byte 8 onward) and string's lenght (xored from the second 4 bytes).
# I didn't used byte array, which it would be another possibility, because I wanted
# keep the code simpler as possible.
def decrypter(data_key, data_string, stringlength):

    decoded = ''
    for i in range(0, stringlength):
        decoded += chr((data_string[i]) ^ (data_key[i % len(data_key)]))

    # I'm returning the literal representation because there're some "\r\n"
    # characters, null bytes and Unicode ones, so if we omit this function,
    # so we're going to lost some strings. As I mentioned previously, if we
    # had used byte array, it could be easier to handle this issue here.
    return (repr(decoded))


# This routine extracts data from .text section because, in this case, the
# encrypted strings are stored in the .text section.
def extract_data(filename):

    pe=pefile.PE(filename)
    for section in pe.sections:
        if '.text' in section.Name.decode(encoding='utf-8').rstrip('x00'):
            return (section.get_data(section.VirtualAddress, section.SizeOfRawData))
```

[Figure 31] Script to decrypt strings (first part)

```python
# This routine calculates the offset between the the start of the .text section and
# and address of the encrypted string. In this case, encrypted strings also starts
# at beginning of the section, but this is a particular case.
def calc_offsets(x_seg_start, x_start):

    data_offset = hex(int(x_start,16) - int(x_seg_start,16))
    return data_offset


# This routine is responsible for calling the routine for extracting the encrypted
# string, and separates the components: the XOR key, the XORed string's length and
# the encrypted string to be decrypted. At end, the routine calls decrypter( ) routine
# to decrypt all found strings.
def string_decrypter(text_seg_start, encrypted_string_addr, encrypted_end):

    # Next line calculates the offset between the start of the .text segment and
    # the address of the provided string.
    encrypted_string_addr_rel = calc_offsets(text_seg_start, encrypted_string_addr)

    # Next two lines extracts .text section's information.
    filename = r"C:\Users\Administrador\Desktop\MAS\mas_3\mas_3_unpacked\mas_3_unpacked.bin"
    text_encoded_extracted = extract_data(filename)

    # Next two lines calculates the size of the encrypted string table. Pay attention that
    # there're two possible approaches here: we can provide the address of the end of the
    # encrypted strings (0x00B81930 -- this is the end + 1) and, of course, the offset is only
    # the different of the addresses and in this case works well. Another possible approach
    # would be searching up to a marker. To get this marker, go to address 00b81801,
    # undefine (U hotkey) the sub_B81801 subroutne and write down the first to two bytes: 7D, 66h.
    # Therefore, in this case, I'm using the easier way that's is only provide the start and end
    # address of the encrypted blob, but readers could examine the second approach (commented below).
    d1_off = 0x0
    d1_off = int(encrypted_end,16) - int(text_seg_start,16)

    # comment only the previous line and uncomment the next two ones to use the marker method.
    # if (b'\x83\xec' in text_encoded_extracted[int(encrypted_string_addr_rel,16):]):
    #    d1_off = (text_encoded_extracted[int(encrypted_string_addr_rel,16):]).index(b'\x83\xec')

    bytes_extracted = (text_encoded_extracted[int(encrypted_string_addr_rel,16):\
    int(encrypted_string_addr_rel,16) + d1_off])

    # print("extracted_encrypted_string:")
    # Uncomment next 2 lines to print the hexadecimal representation of the extracted bytes.
    #print(binascii.b2a_hex(bytes_extracted))

    # In the next lines I'm going to extract the XOR key, the XORed string length and
    # the encrypted blob. No doubts, the advantage of using struct.unpack() routine is that
    # the little indian issue and data's representation are already handled.
    offset = 0
    xorkey = bytes_extracted[offset:(offset+4)]
    xorkey_unpacked = struct.unpack('<I', xorkey)[0]
    xored_length = bytes_extracted[(offset+4):(offset+4+4)]
    xored_length_unpacked = struct.unpack('<I', xored_length)[0]

    # We need to find out the real length of each encrypted string before proceeding,
    # so we have to do an xor operation between the first four bytes (XOR key) and the second
    # four bytes (XORed string's length). This length is used for determining how many bytes to
    # extract.
    string_length = xorkey_unpacked  ^ xored_length_unpacked
    encrypted_string = bytes_extracted[8:8 + string_length]

```

[Figure 32] Script to decrypt strings (second part)

```
96      # Next two lines might seem complicated, but they aren't. The first one call the decrypter( )
97      # routine and removes unnecessary a unnecessary character(') and replace another one. As the
98      # resulting might have some Unicode characters, so we remove them by decoding the string (turning
99      # it to bytes) and returning it to ascii string again, but ignoring any obvious translation
100     #error from Unicode to byte.
101     decoded_data = (decrypter(xorkey, encrypted_string, string_length)).strip("'").replace("\\r\\n","\n")
102     return (decoded_data.encode('utf-8').decode('ascii','ignore'))
103
104
105 # This routine aims to patch any global variable's name representing the address of the
106 # encrypted string by the plain text string representation. There're three small details
107 # here: 1. we have to add "\x00" at end to establish a well-formed string ; 2. We're using
108 # the patch_byte function to perform the change and 3. The function create_strlit is used
109 # to create a string using the patched bytes, which is termined by "\x00". The advantage of
110 # this approach is that we will see string in assembly and pseudo code's view instead of
111 # visualing the global variable's name template (dword_<address>). In the other hand,
112 # this operation alters the idb database and, eventually, it might not be needed.
113 # Although I'll follow this approach here, personally I prefer only creating comments within
114 # the code, though all these comments are visible only in assembly code.
115 def fix_operand(prov_addr, prov_string):
116     addr = prov_addr
117     string_bytes_1 = bytes(prov_string,'utf-8') + b'\x00'
118     for x in string_bytes_1:
119         patch_byte(addr, x)
120         addr = addr + 1
121     create_strlit(prov_addr, idc.BADADDR)
122
123
124 # To get this values (start_addr and end_addr), I've just examine code in .text section.
125 start_addr = 0x00B81000
126 end_addr = 0x00B81930
127 ea = start_addr
128
129 # This loop is calls the string_decrypter routine, make a comment at each instruction
130 # using the decrypted string and, additionally, patch the idb database using the same
131 # decrypted strings. To be clear: we don't need to use both approaches, but I'm doing
132 # it only for education purposes.
133 while (ea < end_addr):
134     for xref in idautils.XrefsTo(ea, 1):
135         # Next line can be uncommented to visualize the address of the decoded strings,
136         # the address of the instruction referring to the given string and, finally,
137         # the own instruction doing such reference.
138         #print("0x%x 0x%x %s" % (xref.frm, xref.to, idc.generate_disasm_line(xref.frm, 0)))
139         final_string = string_decrypter(hex(start_addr), hex(xref.to), hex(end_addr))
140
141         # Obviously, this line prints the decrypted string in the IDA Pro's Output view.
142         print("%s" % final_string)
143
144         # The next line creates a comment next to the instruction referring to the string
145         # using the strings with its content.
146         idc.set_cmt(xref.frm, final_string, 0)
147
148         # The next instruction effectively changes the global variable notation to
149         # the decrypted string in the idb database.
150         fix_operand(xref.to, final_string)
151
152     # We're adding four to ea because all address are 32-bit.
153     ea += 4
154
155 # If readers want to decrypt only one string, so you should comment the
156 # entire while loop above (lines 132 to 152) and uncomment the nex line.
157 # print("\n\n%s" % (string_decrypter('0x00B81000','00B81174','0x00B81930')))
```

[Figure 33] Script to decrypt strings (third and last part)

The content of the **IDA's Output window** is the following one:

```
Python>
%s_%08X
%u.%u.%u.%u

--%S--
Cookie: %s=%s

Content-Type: multipart/form-data; boundary=%s


--%S
Content-Disposition: form-data; name="%S"; filename="%S"
Content-Type: application/octet-stream


%s\\%s
%s\\%s
%s\\%s
%s\\%s
%s\\%s
%s\\%s
%s\\regsvr32.exe /s "%s\\%s"
%s\\regsvr32.exe /s "%s\\%s"
%s\\regsvr32.exe /s "%s\\%s"
%s\\%s%x
SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Run
SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Run
%s\\regsvr32.exe /s "%s\\%s" %s
userenv.dll
bcrypt.dll
shell32.dll
crypt32.dll
shlwapi.dll
wtsapi32.dll
wininet.dll
urlmon.dll
advapi32.dll
RNG
"ECS1 \x00\x00\x00/\x1a\x836\x1a'Q\x8c$\x85RYQuH\x94\x06p\x02s\x9a({V\t\x0f\x08{\x80!z(~\x96E\\&t\x85\x1f~-\x954"
"ECS1 \x00\x00\x00/\x1a\x836\x1a'Q\x8c$\x85RYQuH\x94\x06p\x02s\x9a({V\t\x0f\x08{\x80!z(~\x96E\\&t\x85\x1f~-\x954"
ECK1 \x00\x00\x005\x93c\x8bPi6\x00\x9b\x9bT;\x166]D.80G6.\x8bjLzED\x1a\x06\x8f8\x00\x96\x9c\x84,\x1e6\x80$P\x96
ECK1 \x00\x00\x005\x93c\x8bPi6\x00\x9b\x9bT;\x166]D.80G6.\x8bjLzED\x1a\x06\x8f8\x00\x96\x9c\x84,\x1e6\x80$P\x96
ECCPUBLICBLOB
ECCPUBLICBLOB
ECCPUBLICBLOB
Microsoft Primitive Provider
Microsoft Primitive Provider
Microsoft Primitive Provider
Microsoft Primitive Provider
HASH
SHA256
SHA256
ObjectLength
ObjectLength
AES
KeyDataBlob
ECDH_P256
ECDSA_P256
%s:Zone.Identifier
%s\\*
%s\\%s
WinSta0\\Default
POST
%s%s.dll
%s%s.dll
%s%s.exe
%s%s.exe
%s\\regsvr32.exe /s "%s" %s
%s\\regsvr32.exe /s "%s"
DllRegisterServer
```

[Figure 34] Decrypted strings

An educational experience can be done here. I commented the **line 150** (***fix_operand(xref.to, final_string)***) of our script **(Figure 33).**

If readers to run the script, you will have the following piece of code **including strings used as comment next to instructions** and the result will be similar to the visualized below:

```
.text:00B915CE                 pop     edx
.text:00B915CF                 mov     ecx, offset dword_B81324 ; userenv.dll
.text:00B915D4                 call    sub_B909F9
.text:00B915D9                 mov     eax, 0C28C6FDh
.text:00B915DE                 jmp     short loc_B915A4
.text:00B915E0 ; ---------------------------------------------------------------------
.text:00B915E0
.text:00B915E0 loc_B915E0:                             ; CODE XREF: sub_B9158A+35↑j
.text:00B915E0                 mov     eax, esi
.text:00B915E2                 jmp     short loc_B915A4
.text:00B915E4 ; ---------------------------------------------------------------------
.text:00B915E4
.text:00B915E4 loc_B915E4:                             ; CODE XREF: sub_B9158A+31↑j
.text:00B915E4                 push    5
.text:00B915E6                 pop     edx
.text:00B915E7                 mov     ecx, offset dword_B81454 ; urlmon.dll
.text:00B915EC                 call    sub_B909F9
.text:00B915F1                 mov     eax, 8845D28h
.text:00B915F6                 jmp     short loc_B915A4
.text:00B915F8 ; ---------------------------------------------------------------------
.text:00B915F8
.text:00B915F8 loc_B915F8:                             ; CODE XREF: sub_B9158A+2A↑j
.text:00B915F8                 push    3
.text:00B915FA                 pop     edx
.text:00B915FB                 mov     ecx, offset dword_B81364 ; shell32.dll
.text:00B91600                 call    sub_B909F9
.text:00B91605                 mov     eax, 0C02343Bh
.text:00B9160A                 jmp     short loc_B915A4
.text:00B9160C ; ---------------------------------------------------------------------
.text:00B9160C
.text:00B9160C loc_B9160C:                             ; CODE XREF: sub_B9158A+26↑j
.text:00B9160C                 xor     edx, edx
.text:00B9160E                 mov     ecx, offset dword_B81344 ; bcrypt.dll
.text:00B91613                 inc     edx
.text:00B91614                 call    sub_B909F9
.text:00B91619                 mov     eax, 94106F9h
.text:00B9161E                 jmp     short loc_B915A4
.text:00B91620 ; ---------------------------------------------------------------------
.text:00B91620
.text:00B91620 loc_B91620:                             ; CODE XREF: sub_B9158A+22↑j
.text:00B91620                 xor     edx, edx
.text:00B91622                 mov     ecx, offset dword_B81494 ; advapi32.dll
.text:00B91627                 call    sub_B909F9
.text:00B9162C                 mov     eax, ebx
```

[Figure 35] Code commented using decrypted strings

As readers might notice, data references (***dword_<address>***) haven't been renamed and only comments were created next to respective references, as expected. This is a welcome approach because readers are able to see all decrypted strings in the **Disassembly view** without needing to change instruction operands in the **idb database**. In the other hand, we don't have the same comment on the **pseudocode's view**, which could be an issue for some demanding professionals.

Uncommenting the **line 150** (as in the original script in **Figure 33**), the result is a bit different in **IDA View** and **Pseudocode View**, as shown below:

```
.text:00B915CE                 pop     edx
.text:00B915CF                 mov     ecx, offset aUserenvDll ; userenv.dll
.text:00B915D4                 call    sub_B909F9
.text:00B915D9                 mov     eax, 0C28C6FDh
.text:00B915DE                 jmp     short loc_B915A4
.text:00B915E0 ; ---------------------------------------------------------------------------
.text:00B915E0
.text:00B915E0 loc_B915E0:                             ; CODE XREF: sub_B9158A+35↑j
.text:00B915E0                 mov     eax, esi
.text:00B915E2                 jmp     short loc_B915A4
.text:00B915E4 ; ---------------------------------------------------------------------------
.text:00B915E4
.text:00B915E4 loc_B915E4:                             ; CODE XREF: sub_B9158A+31↑j
.text:00B915E4                 push    5
.text:00B915E6                 pop     edx
.text:00B915E7                 mov     ecx, offset aUrlmonDll ; urlmon.dll
.text:00B915EC                 call    sub_B909F9
.text:00B915F1                 mov     eax, 8845D28h
.text:00B915F6                 jmp     short loc_B915A4
.text:00B915F8 ; ---------------------------------------------------------------------------
.text:00B915F8
.text:00B915F8 loc_B915F8:                             ; CODE XREF: sub_B9158A+2A↑j
.text:00B915F8                 push    3
.text:00B915FA                 pop     edx
.text:00B915FB                 mov     ecx, offset aShell32Dll ; shell32.dll
.text:00B91600                 call    sub_B909F9
.text:00B91605                 mov     eax, 0C02343Bh
.text:00B9160A                 jmp     short loc_B915A4
.text:00B9160C ; ---------------------------------------------------------------------------
.text:00B9160C
.text:00B9160C loc_B9160C:                             ; CODE XREF: sub_B9158A+26↑j
.text:00B9160C                 xor     edx, edx
.text:00B9160E                 mov     ecx, offset aBcryptDll ; bcrypt.dll
.text:00B91613                 inc     edx
.text:00B91614                 call    sub_B909F9
.text:00B91619                 mov     eax, 94106F9h
.text:00B9161E                 jmp     short loc_B915A4
.text:00B91620 ; ---------------------------------------------------------------------------
.text:00B91620
.text:00B91620 loc_B91620:                             ; CODE XREF: sub_B9158A+22↑j
.text:00B91620                 xor     edx, edx
.text:00B91622                 mov     ecx, offset aAdvapi32Dll ; advapi32.dll
.text:00B91627                 call    sub_B909F9
.text:00B9162C                 mov     eax, ebx
```

[Figure 36] Code commented with decrypted strings and data references renamed

As readers are able to notice, this time we can see data references renamed using the name of decrypted strings and, additionally, we kept all comments. Of course, we don't need both ones, but I left them here to show you the final effect. If you return to the **.text section** for any of these strings (for example, **urlmon.dll**), you'll see the following:

```
.text:00B81454 aUrlmonDll      db 'urlmon.dll',0       ; DATA XREF: sub_B9158A+5D↓o
.text:00B8145F                 db  7Fh ;
.text:00B81460                 db 0D5h
.text:00B81461                 db 0CEh
```

[Figure 37] Renamed data reference in .text section

Finally, and maybe the most important, the **pseudocode view** presents the following instructions:

```
        switch ( i )
        {
          case 0x9155FA6:
            sub_B909F9((int)"advapi32.dll", 0);
            i = 0x155153;
            break;
          case 0x155153:
            sub_B909F9((int)"bcrypt.dll", 1);
            i = 0x94106F9;
            break;
          case 0xE45F04:
            sub_B909F9((int)"shell32.dll", 3);
            i = 0xC02343B;
            break;
          case 0x25F6E14:
            sub_B909F9((int)"urlmon.dll", 5);
            i = 0x8845D28;
            break;
          case 0x7746308:
            i = 0x9155FA6;
            break;
          default:
            sub_B909F9((int)"userenv.dll", 6);
            i = 0xC28C6FD;
            break;
        }
      }
```

[Figure 38] Renamed data references in pseudocode view

The pseudo code above shows the expected result, which includes **all strings** as part of instructions.

At end of the script (**Figure 33**), I inserted a comment explaining that readers could comment the entire while loop block (**line 133 to 153**) and uncomment the **line 157** for being able to decrypt only string for testing purposes.

Now readers have seen the entire script and respective results, I'd like to make few considerations about the **IDC/IDA Python script**:

- I used IDC functions because in many opportunities it makes our work much easier, so both **idautils and idc libraries were imported** (**lines 4 and 5**).

- We could use byte arrays and, in this case, I made an option to keep everything as string to keep the code simple.

- If reader doesn't know about the **repr( ) function** on **line 22**, which is a Python **built-in function**, so search about it on: https://docs.python.org/3/library/functions.html#repr

- The **data extraction code (lines 27 to 32)** is exactly the same from second article of this series, but it was adapted to extract data from **.text section**.

- **On lines 85 and 87** the scripts uses **struct.unpack(**). Python struct is a powerful resource to interpret bytes as packed binary data and it's able to do this interpretation according to the byte order (**little-endian, big-endian or even native**). You can read a bit more about Python structs and learn from examples on https://docs.python.org/3/library/struct.html.

- **On line 101,** the **decrypter( ) routine** is called and, once the result is returned, all single quotes are removed and, soon after it, any sequence **"\r\n"** is converted to **"\n".**

- **On line 102, Unicode characters were removed** because we don't understand them and, for this specific purpose, they won't be useful.

- **From line to 115 to 121**, we have the routine used to patch the binary and change the data reference names to the a name represented by the decrypted string. The sequence should be clear: we converted decrypted strings to byte representation, **appended the "\x00" to the end of the sequence of bytes to get a well-formed string**, patched the provided address with each letter of the decrypted strings and, finally, we created a new string using a IDC function named **strlit(long ea, long len)**. Note that we could have specified the length of the string, but we chose using a string delimitator: https://hex-rays.com/products/ida/support/idadoc/207.shtml.

- **On line 134**, **idautils.XrefsTo( ) function** is used to get all references to the address of the given encrypted string, so we are able to **get all instructions' addresses referring to the encrypted string**: https://hex-rays.com/products/ida/support/idapython_docs/idautils.html#idautils.XrefsTo

- **On lines 138 and 139**, it's suitable to highlight that **xref.to** provides the address of the encrypted string and **xref.frm** provides the address of the instruction referring to the encrypted string.

- **One line 146**, the **set_cmt( ) function** is used to set an indented comment: https://hex-rays.com/products/ida/support/idadoc/204.shtml

- On **line 150** the script call **fix_operand( )** that is responsible for **patching the idb database** by **replacing the string reference by the string itself.**

- Finally, **on line 157**, the script offers the option **to decrypt only one given encrypted string**, but it's necessary to **comment out the whole while loop between lines 133 and 153**.

- Readers are able to get both **start** and **end addresses** of encrypted strings by examining the **.text section** (**CTRL+S**) as shown below:

```
.text:00B81000 _text          segment para public 'CODE' use32
.text:00B81000                assume cs:_text
.text:00B81000                ;org 0B81000h
.text:00B81000                assume es:nothing, ss:nothing, ds:_data, fs:nothing, gs:nothing
.text:00B81000 dword_B81000   dd 2A8EF14Bh, 2A8EF14Ch, 0FD1826Eh, 65D6C97Bh, 0B1DA0BC0h
.text:00B81000                                          ; DATA XREF: sub_BA03F2+2E5↓o

.text:00B818F4 dword_B818F4   dd 1C1D82CFh, 1C1D82DEh, 4E71EE8Bh, 6F74E5AAh, 4F6FE7BBh
.text:00B818F4                                          ; DATA XREF: sub_B888E5+17E↓o
.text:00B818F4                dd 796BF0AAh, 0EABF9FBDh, 0F62FB5A6h, 5C49504Bh, 58FD82D0h
.text:00B818F4                dd 6317FDF5h, 116757A6h, 0A6970A9h, 7E6C7D7Eh, 3AEB283h
.text:00B81930
```

[Figure 39] Getting start and end addresses of the encrypted strings

Now we have decrypted strings, let's move forward. Return to the beginning of the malware, which is the exported **DllRegisterServer( ),** and go to **sub_B91FD0( ) subroutine**, which is effectively the first one to be called within **DllRegisterServer( )**, as shown below:

```
HRESULT __stdcall DllRegisterServer()
{
  int v0; // ecx

  sub_B91FD0();
  return sub_B8BA9C(v0, v0, 0);
}
```

[Figure 40] First function to be called in DllRegisterServer( ) (exported by malware)

Going inside the given routine (**sub_B91FD0()**), we have the following:

```
49              while ( v0 <= 0x7C1887A )
50              {
51                if ( v0 == 0x7C1887A )
52                {
53                  result = sub_B9EBA2();
54                  v0 = 0x629CB8B;
55                }
56                else if ( v0 > 0x4F9319B )
57                {
58                  if ( v0 > 0x65F58CA )
59                  {
60                    switch ( v0 )
61                    {
62                      case 0x66D9794:
63                        v7 = sub_B8D75A();
64                        result = sub_B81B09(v17, 0x6AA8E, v7, 0x426DE, 1, 0xED19F, v18);
65                        v0 = result != 0 ? 0xB3A13B4 : 0x66D9794;
66                        break;
67                      case 0x6B290E4:
68                        result = sub_B8960B();
69                        if ( !result )
70                          result = sub_B9C535();
```

[Figure 41] Beginning of sub_B91FD0 subroutine

On line 53 we found a call to **sub_B9EBA2( ) subroutine**, which has the content below:

```
int sub_B9EBA2()
{
  int i; // ecx

  for ( i = 0xE0A0B0E; i != 0xB52FD32; i = 0xB52FD32 )
    dword_BA5080 = sub_B9EAA3((void *)0x118);
  sub_BA03F2(0x80A9B, dword_BA5080 + 8, 0x16460);
  return 1;
}
```

[Figure 42] Sub_B9EBA2 subroutine

There's nothing so attractive, except by few non-sense values. Thus, let's carry on and go into the first subroutine (**sub_B9EAA3**) and next to **sub_B8645E( )**, where we will find such sequence of code:

```
int __thiscall sub_B9EAA3(void *this)
{
  int v2; // eax

  v2 = sub_B8645E((void *)0x43);
  return sub_B91B22(v2, 8, 0xD93FD, (int)this);
}
```

[Figure 43] Sub_B9EAA3 subroutine

```
1 int __thiscall sub_B8645E(void *this)
2 {
3   int (__cdecl *v1)(int, int, _DWORD, int, int, int, int); // eax
4 
5   v1 = (int (__cdecl *)(int, int, _DWORD, int, int, int, int))sub_B9BFF0(0x303, (int)this, (int)this, 0x76FC34E6);
6   return v1(0xB7D5AF, 0xCB85E0, 0, 0xEFD1B, 0x7F43F, 0xFC523, 0x3BD7C);
7 }
```

[Figure 44] Sub_B8645E subroutine

From **Figure 42**, we didn't find anything relevant again, and we have two calls: **sub_B8645E( )** and **sub_B91B22( )**, both including some strange hexadecimal numbers. If we examine the content of **sub_B8645E( ),** we're going to discover the content of **Figure 44** and things starts to be interesting, so we can do first considerations:

- Initially it seems we have a C++ function call, but soon below the **cdecl calling convention** is being used on the **sub_B9BFF0( )** function call.

- The **sub_B9BFF0( ) subroutine call (line 5**) has several arguments, where **the last one seems to be a hash** and, usually, this is expected when analyzing malware samples with **obfuscation techniques.**

- Returning a value/string to a **local variable (v1)** is an indication that there's something related to hash resolution (DLL or API hashing) and, as readers are going to see in this case, an **API hashing name resolution**.

- Finally, it seems that **v1 is contains the name of a function (API**) because on line 6 the **v1's content is used as the name of the called function, which includes several (and fake) arguments.**

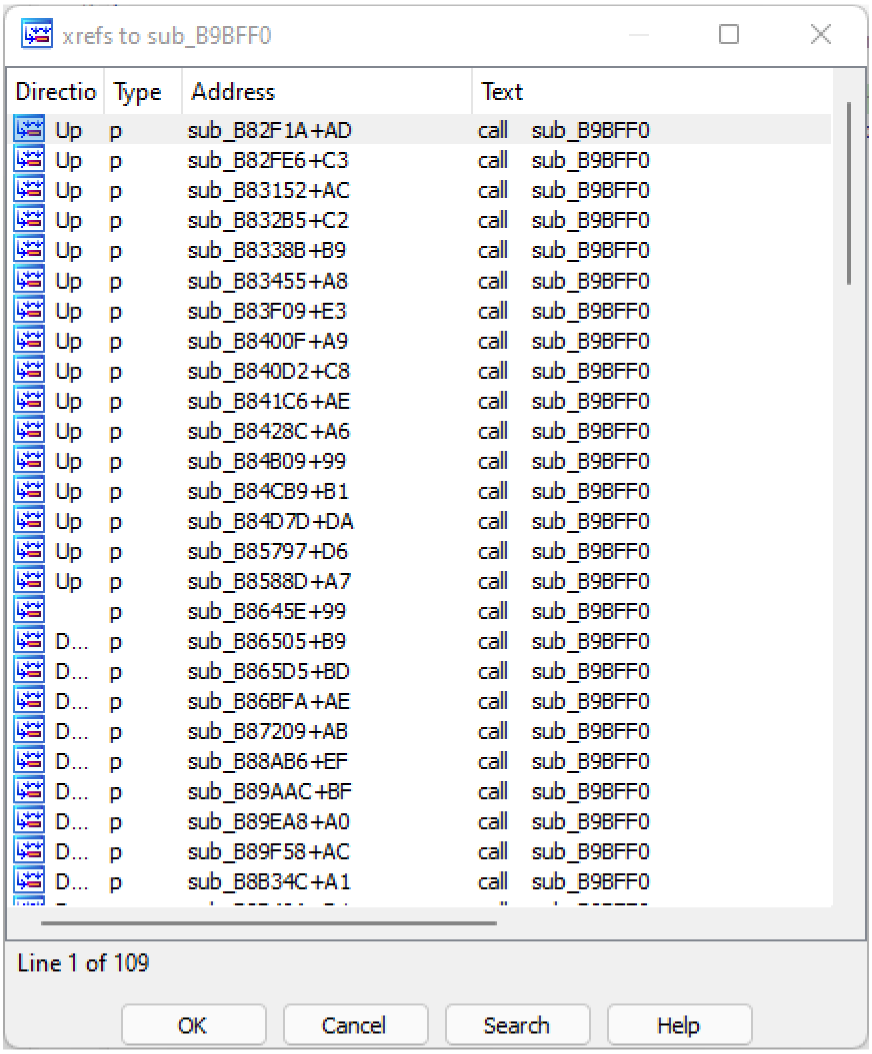
- Reading all 7 lines, the general idea is that **the function on line 1 is a wrapper/proxy, where first an API name is resolved for a given hash and, after being resolved, it's called.** As this **wrapper function on line doesn't have any useful arguments, so the calling on line 6 doesn't have any concrete argument neither**.

- Readers can easily to confirm that **v1( ) call (line 6) doesn't have arguments** by checking the assembly code and, as you're able to see, **between sub_B9BFF0( ) on line 5 and this one on line 06, there's only a stack adjustment**:

```
.text:00B864F2                 mov     ecx, 0AC802C42h
.text:00B864F7                 call    sub_B9BFF0
.text:00B864FC                 add     esp, 14h
.text:00B864FF                 call    eax
```

[Figure 45] No arguments for sub_B8645E (call eax)

In any malware analysis case where there's API hash resolution, **the responsible routine is called many times (once for each function hash),** so the obvious step it to check how many time **sub_B9BFF0( )** subroutine is called (**X hotkey**):



[Figure 46] Cross-references to sub_B9BFF0( ) subroutine

There're **109 cross-references** to it, so it seems a promising subroutine. Stepping into it, we have:

```
int __cdecl sub_B9BFF0(int a1, int a2, int a3, int a4)
{
  int v4; // ecx
  struct _LIST_ENTRY *v5; // eax

  if ( !dword_BA4218[a1] )
  {
    v5 = sub_BA1AE9(v4);
    dword_BA4218[a1] = sub_B9B558(0xDF7EE, a4, 0xB21A0, (int)v5);
  }
  return dword_BA4218[a1];
}
```

[Figure 47] sub_B9BFF0( ) subroutine

In this subroutine we have the following artifacts:

- a possible array (**lines 6, 9 and 11**), which it seems being used to hold API's names.
- a call to **sub_BA1AE9**, which the v4 argument comes from stack (ecx).
- a call to **sub_B9B558** using **v5 local variable** (returned from **sub_BA1AE9**) as argument and **a4** argument that, according to **Figure 44 (sub_B8645E subroutine)** is an hexadecimal and possible an API hash.

Going into **sub_BA1AE9( )** subroutine we see:

```
1  struct _LIST_ENTRY *__cdecl sub_BA1AE9(int a1)
2  {
3    struct _LIST_ENTRY *p_InLoadOrderModuleList; // edi
4    struct _LIST_ENTRY *i; // esi
5
6    p_InLoadOrderModuleList = &sub_B9AA52()->Ldr->InLoadOrderModuleList;
7    for ( i = p_InLoadOrderModuleList->Flink; ; i = i->Flink )
8    {
9      if ( i == p_InLoadOrderModuleList )
10       return 0;
11     if ( (sub_B940AF(0x582E9, 0xBF580, 0x9991, i[6].Flink) ^ 0x23FECA30) == a1 )
12       break;
13   }
14   return i[3].Flink;
15 }
```

[Figure 47] sub_BA1AE9( ) subroutine

Wow! We found the **subroutine responsible for DLL hashing resolving**, **which after finding the right DLL name, it returns its respective address.** Thus, some eventual considerations follows below:

- On line 6, the **sub_B9AA52 subroutine** gets the **PEB (Process Environment Block)** and it's trivial to understand it because of **NtCurrentPeb( )** call (instruction **mov eax, large fs:30h** at address **0x00B9AA52**).

- At **same line 6**, the **_PEB struct** has a field named **Ldr (offset 0xC)**, which is a pointer to **PEB_LDR_DATA structure** (https://www.nirsoft.net/kernel_struct/vista/PEB.html):

```
typedef struct _PEB
{
     UCHAR InheritedAddressSpace;
     UCHAR ReadImageFileExecOptions;
     UCHAR BeingDebugged;
     UCHAR BitField;
     ULONG ImageUsesLargePages: 1;
     ULONG IsProtectedProcess: 1;
     ULONG IsLegacyProcess: 1;
     ULONG IsImageDynamicallyRelocated: 1;
     ULONG SpareBits: 4;
     PVOID Mutant;
     PVOID ImageBaseAddress;
     PPEB_LDR_DATA Ldr;
     PRTL_USER_PROCESS_PARAMETERS ProcessParameters;
```

[Figure 48] _PEB structure

- You can see the same **_PEB structure** on IDA Pro by going to **Structure tab (SHIFT+F9)**, **pressing INSERT key, clicking on Add Standard Structure**, **searching for _PEB structure** and adding it:

```
00000000 _PEB             struc ; (sizeof=0x248, align=0x8, copyof_1)
00000000 InheritedAddressSpace db ?
00000001 ReadImageFileExecOptions db ?
00000002 BeingDebugged    db ?
00000003 anonymous_0      _PEB::$D57935FE5756AF9F9B84A66E67E8019A ?
00000004 Mutant           dd ?                    ; offset
00000008 ImageBaseAddress dd ?                    ; offset
0000000C Ldr              dd ?                    ; offset
00000010 ProcessParameters dd ?                   ; offset
```

[Figure 49] _PEB structure (from IDA Pro)

- If you want to learn a bit more about **PEB** and navigate within its fields, a good reference follow: https://processhacker.sourceforge.io/doc/struct___p_e_b.html.

- The **_PEB_LDR_DATA structure**, pointed by **Ldr field** from **_PEB structure**, is the **representation of a DLL module loaded in the process**. Its internal composition has the following content according to the IDA Pro, which readers can have access by repeating the same mentioned method**: go to Structure tab (SHIFT+F9) → press Insert → go to Add Standard Structure** and search for **PEB_LDR_DATA structure** (alternatively, you can check the same information, but presented in a different format, on: https://www.nirsoft.net/kernel_struct/vista/PEB_LDR_DATA.html):

```
00000000 _PEB_LDR_DATA   struc ; (sizeof=0x30, align=0x4, copyof_5)
00000000 Length          dd ?
00000004 Initialized     db ?
00000005                 db ? ; undefined
00000006                 db ? ; undefined
00000007                 db ? ; undefined
00000008 SsHandle        dd ?                    ; offset
0000000C InLoadOrderModuleList _LIST_ENTRY ?
00000014 InMemoryOrderModuleList _LIST_ENTRY ?
0000001C InInitializationOrderModuleList _LIST_ENTRY ?
00000024 EntryInProgress dd ?                    ; offset
00000028 ShutdownInProgress db ?
00000029                 db ? ; undefined
0000002A                 db ? ; undefined
0000002B                 db ? ; undefined
0000002C ShutdownThreadId dd ?                   ; offset
00000030 _PEB_LDR_DATA   ends
```

[Figure 50] PEB_LDR_DATA structure (from IDA Pro)

- The **InLoadOrderModuleList** points to a **_LIST_ENTRY structure**, which represents a double linked list, as shown below (extracted from IDA Pro -- **Structure tab**). Of course, **InMemoryOrderModuleList** and **InInitializationOrderModuleList** has the same representation:

```
00000000 _LIST_ENTRY     struc ; (sizeof=0x8, align=0x4, copyof_6)
00000000                                         ; XREF: _PEB/r
00000000                                         ; _PEB_LDR_DATA/r ...
00000000 Flink           dd ?                    ; offset
00000004 Blink           dd ?                    ; offset
00000008 _LIST_ENTRY     ends
```

[Figure 51] _LIST_ENTRY structure (from IDA Pro)

- Although readers probably already know meaning of these field, it's worth to remember them here:

- **InLoadModuleList:** it's a double-linked list that organizes all modules (DLLs) in the the exact order that they were loaded into a process on memory.

- **InMemoryOrderList:** it's a double-linked list that organizes all modules (DLLs) in the order that they appear on the process's memory.

- **InInitializationOrderModuleList:** it's a double-linked list that organizes all modules (DLLs) in the order that they were initialized.

- All these fields from the **_PEB_LDR_DATA structure** are head of **LDR_DATA_TABLE_ENTRY structures** (shown below), which are one represents a **loaded DLL module**:

```
00000000 LDR_DATA_TABLE_ENTRY struc ; (sizeof=0x80, align=0x8, copyof_34)
00000000 InLoadOrderLinks _LIST_ENTRY ?
00000008 InMemoryOrderLinks _LIST_ENTRY ?
00000010 InInitializationOrderLinks _LIST_ENTRY ?
00000018 DllBase         dd ?                        ; offset
0000001C EntryPoint      dd ?                        ; offset
00000020 SizeOfImage     dd ?
00000024 FullDllName     _UNICODE_STRING ?
0000002C BaseDllName     _UNICODE_STRING ?
00000034 Flags           dd ?
00000038 LoadCount       dw ?
0000003A TlsIndex        dw ?
0000003C anonymous_0     _LDR_DATA_TABLE_ENTRY::$2AEAD4F93EB0D2D784A8BC67C8E3780B ?
00000044 CheckSum        dd ?
00000048 anonymous_1     _LDR_DATA_TABLE_ENTRY::$07E50B91B6BB7B4A220DB818F2334DE2 ?
0000004C EntryPointActivationContext dd ?        ; offset
00000050 PatchInformation dd ?                   ; offset
00000054 ForwarderLinks  _LIST_ENTRY ?
0000005C ServiceTagLinks _LIST_ENTRY ?
00000064 StaticLinks     _LIST_ENTRY ?
0000006C ContextInformation dd ?                 ; offset
00000070 OriginalBase    dd ?
00000074                 db ? ; undefined
00000075                 db ? ; undefined
00000076                 db ? ; undefined
00000077                 db ? ; undefined
00000078 LoadTime        _LARGE_INTEGER ?
00000080 LDR_DATA_TABLE_ENTRY ends
```

[Figure 52] _PEB_LDR_DATA structure (from IDA Pro)

- There're several and quite interesting fields such as **FullDllName** (it holds the full path of DLL on disk), **BaseDllName** (it holds the DLL name), **LoadCount** (contains the number of times this DLL was loaded using **LoadLibrary( )**) and, finally, **DllBase** (contains the base address of the DLL).

- Therefore, the basic idea is: the code parses all modules loaded in the memory, gets the respective DLL name, calculates the associated hash by using **sub_B940AF subroutine**, performs an XOR operation with the given key (**0x23FECA30**) and compares each result with the calculated hash. **If there's a match, so the DLL's base address is returned.**

- In fact, all hashing mechanisms have a similar modus-operandi and, basically, they changes only the algorithm (obviously) and eventually have further logical operation as, in this case, **an additional step is done by performing a XOR instruction with an extra XOR key**.

Now that readers refreshed few important concepts, the altered code after having done a minimal work on it follows:

```
1 struct _LIST_ENTRY *__cdecl mw_dll_hashing(int a1)
2 {
3   LDR_DATA_TABLE_ENTRY **p_InLoadOrderModuleList; // edi
4   LDR_DATA_TABLE_ENTRY *ptr_dll_representation; // esi
5 
6   p_InLoadOrderModuleList = (LDR_DATA_TABLE_ENTRY **)&getPEB()->Ldr->InLoadOrderModuleList;
7   for ( ptr_dll_representation = *p_InLoadOrderModuleList;
8         ;
9         ptr_dll_representation = (LDR_DATA_TABLE_ENTRY *)ptr_dll_representation->InLoadOrderLinks.Flink )
10  {
11    if ( ptr_dll_representation == (LDR_DATA_TABLE_ENTRY *)p_InLoadOrderModuleList )
12      return 0;
13    if ( (mw_dll_hashing_algo(0x582E9, 0xBF580, 0x9991, ptr_dll_representation->BaseDllName.Buffer) ^ 0x23FECA30) == a1 )
14      break;
15  }
16  return (struct _LIST_ENTRY *)ptr_dll_representation->DllBase;
17 }
```

[Figure 53] _PEB_LDR_DATA structure (from IDA Pro)

Surprisingly, I made few changes in the code above whether compared to **Figure 47**:

- In the **Structure view** (**SHIFT+F9**), I inserted the **LDR_DATA_TABLE_ENTRY structure**.
- I renamed (**N hotkey**) **sub_A1AE9 subroutine** to **mw_dll_hashing**.
- I renamed (**N hotkey**) "i" local variable to **ptr_dll_representation** (you can give whatever name you want).
- **(trick)** I changed **ptr_dll_representation type** (using **Y hotkey**) to **LDR_DATA_TABLE_ENTRY*.**
- I renamed (**N hotkey**) **sub_B940AF subroutine** to **mw_dll_hashing_algo.**
- Press **F5** to "recompile" the code and **update the pseudo code**.
- Observe that the **DLL base address is returned on line 16**.

As readers can notice, it wasn't hard. The code confirms the previous explanation about how it works and, much better, including field names certainly makes the understanding easier.

The **sub_B940AF subroutine** (renamed to **mw_dll_hashing_algo**), which is used to **DLL hashing**, it's quite simple as readers can see below:

```
int __usercall mw_dll_hashing_algo@<eax>(int a1@<edx>, int a2@<ecx>, int a3, _WORD *ptr_provided_name_1)
{
  unsigned int ptr_provided_name; // eax
  int hash; // [esp+20h] [ebp+8h]

  nullsub_1(a2, a1, a3);
  for ( hash = 0; *ptr_provided_name_1; hash = (hash << 0x10) + (hash << 6) + ptr_provided_name - hash )
  {
    ptr_provided_name = (unsigned __int16)*ptr_provided_name_1;
    if ( ptr_provided_name >= 'A' && ptr_provided_name <= 'Z' )
      ptr_provided_name += 0x20;
    ++ptr_provided_name_1;
  }
  return hash;
}
```

[Figure 54] sub_B940AF subroutine

The algorithm above performs the following operations:

- receives a **pointer to the DLL name.**

- Parses each letter (indexed by k) of the given name and calculates the hash by summing up three operations: **hash << 0x10, hash << 6 and (ptr_provided_name[k] – hash)**.
- Checks whether a letter of DLL name is in upper case and, if it's, so **change it to lower case before continuing the interaction with each remaining letter**.
- Finally, **it returns the calculated hash.**

Let's go up two levels back to **sub_B9BFF0 subroutine** and move inside **sub_B9B558 subroutine** that, supposedly has 4 arguments (of course, it doesn't have):

```
1  int __usercall sub_B9B558@<eax>(int a1@<edx>, int a2@<ecx>, int a3, int a4)
2  {
3    char *v4; // esi
4    int v5; // ebp
5    char *v6; // edi
6    int v7; // ecx
7    int v10; // [esp+20h] [ebp-1Ch]
8    int v11; // [esp+24h] [ebp-18h]
9    int v12; // [esp+28h] [ebp-14h]
10   int v13; // [esp+2Ch] [ebp-10h]
11
12   nullsub_1(a2, a1, a3);
13   v4 = 0;
14   v5 = 0;
15   v13 = *(_DWORD *)(a4 + 0x3C);
16   v6 = (char *)(a4 + *(_DWORD *)(v13 + a4 + 0x78));
17   v12 = a4 + *((_DWORD *)v6 + 7);
18   v7 = a4 + *((_DWORD *)v6 + 8);
19   v10 = v7;
20   v11 = a4 + *((_DWORD *)v6 + 9);
21   if ( *((_DWORD *)v6 + 6) )
22   {
23     while ( (sub_B8B099(0x8FE71, 0x5E5FB, (_BYTE *)(a4 + *(_DWORD *)(v7 + 4 * v5)), 0xB6EDA) ^ 0x32C9DB43) != a2 )
24     {
25       v7 = v10;
26       if ( (unsigned int)++v5 >= *((_DWORD *)v6 + 6) )
27         return (int)v4;
28     }
29     v4 = (char *)(a4 + *(_DWORD *)(v12 + 4 * *(unsigned __int16 *)(v11 + 2 * v5)));
30     if ( v4 >= v6 && v4 < &v6[*(_DWORD *)(v13 + a4 + 0x7C)] )
31       return sub_B9B384(v4);
32   }
33   return (int)v4;
34 }
```

[Figure 55] sub_B9B558 subroutine

No doubts, this routine is very similar to the previous one about **DLL hashing**, but it's used to **API hashing**. Clearly there's a **PE parsing operation** happening in the routine and, if you reader to pay attention, there're first few relevant facts:

- a similar operation of "hash comparison", as we seen on **Figure 53** for DLL, it's happening on **line 23** and comparing the found hash against the provided one (**a2**), which comes from **a4** in the parent subroutine (**sub_B9B558**) and, finally, comes from **a4** from subroutine **sub_B9BFF0** that has the hash **0x76FC34E6** as its fourth argument (**Figure 44**).

- The **XOR key** is **0x32C9DB43**, which it's different from previous one used for DLL hashing.

- The possible subroutine handling **the actual hashing operation** is **sub_B8B099**.

Before proceeding in our analysis, we need to add structures which will be necessary to improve our reversing experience, so go to **Structure view (SHIFT+F9) → Insert key → Add standard structure** and **add** the following ones:

- **_IMAGE_DOS_HEADER**
- **_IMAGE_NT_HEADERS**
- **_IMAGE_EXPORT_DIRECTORY**
- **_IMAGE_FILE_HEADER** (automatically loaded by the first three ones)
- **_IMAGE_OPTIONAL_HEADERS32** (automatically loaded by the first three ones)
- **_IMAGE_DATA_DIRECTORY** (automatically loaded by the first three ones)

A good reference to **Windows executable structure** is available on: https://github.com/corkami/pics/blob/master/binary/pe102/pe102.pdf.

First I'm going to show the final code and, afterwards, I'll be explaining all necessary steps for that readers are able to get to the same result:

```
1 char *__usercall mw_api_hash_resolving@<eax>(int a1@<edx>, int a2@<ecx>, int a3, int dll_base_address)
2 {
3   char *ptr_api_name; // esi
4   int counter; // ebp
5   _IMAGE_EXPORT_DIRECTORY *ptr_IMAGE_EXPORT_DIRECTORY; // edi
6   DWORD ptr_AddressOfNames; // ecx
7   DWORD ptr_AddressOfNames_1; // [esp+20h] [ebp-1Ch]
8   DWORD ptr_AddressOfNameOrdinals; // [esp+24h] [ebp-18h]
9   DWORD ptr_AddressOfFunctions; // [esp+28h] [ebp-14h]
10  _IMAGE_NT_HEADERS *ptr_IMAGE_NT_HEADERS; // [esp+2Ch] [ebp-10h]
11
12  nullsub_1(a2, a1, a3);
13  ptr_api_name = 0;
14  counter = 0;
15  ptr_IMAGE_NT_HEADERS = *(_IMAGE_NT_HEADERS **)(dll_base_address + offsetof(_IMAGE_DOS_HEADER, e_lfanew));
16  ptr_IMAGE_EXPORT_DIRECTORY = (_IMAGE_EXPORT_DIRECTORY *)(dll_base_address
17                                                       + *(DWORD *)((char *)&ptr_IMAGE_NT_HEADERS->OptionalHeader.DataDirectory[0].VirtualAddres
18                                                                    + dll_base_address));
19  ptr_AddressOfFunctions = dll_base_address + ptr_IMAGE_EXPORT_DIRECTORY->AddressOfFunctions;
20  ptr_AddressOfNames = dll_base_address + ptr_IMAGE_EXPORT_DIRECTORY->AddressOfNames;
21  ptr_AddressOfNames_1 = ptr_AddressOfNames;
22  ptr_AddressOfNameOrdinals = dll_base_address + ptr_IMAGE_EXPORT_DIRECTORY->AddressOfNameOrdinals;
23  if ( ptr_IMAGE_EXPORT_DIRECTORY->NumberOfNames )
24  {
25    while ( (mw_api_hashing_algo(
26               0x8FE71,
27               0x5E5FB,
28               (_BYTE *)(dll_base_address + *(_DWORD *)(ptr_AddressOfNames + 4 * counter))) ^ 0x32C9DB43) != a2 )
29    {
30      ptr_AddressOfNames = ptr_AddressOfNames_1;
31      if ( ++counter >= ptr_IMAGE_EXPORT_DIRECTORY->NumberOfNames )
32        return ptr_api_name;
33    }
34    ptr_api_name = (char *)(dll_base_address
35                          + *(_DWORD *)(ptr_AddressOfFunctions
36                                        + 4 * *(unsigned __int16 *)(ptr_AddressOfNameOrdinals + 2 * counter)));
37    if ( ptr_api_name >= (char *)ptr_IMAGE_EXPORT_DIRECTORY
38      && ptr_api_name < (char *)ptr_IMAGE_EXPORT_DIRECTORY
39                      + *(DWORD *)((char *)&ptr_IMAGE_NT_HEADERS->OptionalHeader.DataDirectory[0].Size + dll_base_address) )
40    {
41      return (char *)mw_w_api_hash_resolving(ptr_api_name);
42    }
43  }
44  return ptr_api_name;
45 }
```

[Figure 56] sub_B9B558 subroutine after some reversing actions

Due explanations follow (pay attention that line numbers might not be the same):

- Click on **v13** variable, press **N hotkey** and rename it to **ptr_IMAGE_NT_HEADER**S. Afterwards, press **Y hotkey** and **change its type to _IMAGE_NT_HEADERS*** instead of **int**.
- On **0x3C** (**line 15**), press **T hotkey** and **choose _IMAGE_DOS_HEADER**.

- If **a4** argument changed its type, click on it, press **Y hotkey** and change it back to **int a4**.
- On the same **a4 argument**, rename it to **dll_base_address**.
- On line 16, **press Y** on **v6** local variable and change its type to **_IMAGE_EXPORT_DIRECTORY***.
- On line 16, rename **v6** local variable to **ptr_IMAGE_EXPORT_DIRECTORY**.
- On line 19, rename **v12** local variable to **ptr_AddressOfFunctions**.
- On line 20, rename **v7** local variable to **ptr_AddressOfNames**.
- On line 21, rename **v10** local variable to **ptr_AddressOfNames** (again).
- On line 22, rename **v11** local variable to **ptr_AddressOfNameOrdinals**.
- **On line 28, rename v5** local variable to **counter**.
- **On line 29**, as **v4's content** is a **pointer to the API name**, so **rename** it to **ptr_api_name**.
- **On line 38**, **rename sub_B9B384 to mw_w_api_hash_resolving** because, basically, it's the usage of the recent defined routines.
- **On line 25** is the calling of **sub_B8B099**, which is the actual hashing function to calculates the hash value and almost identical to the respective DLL hashing function. Thus, **rename it to mw_api_resolving_algo**, which is shown below:

```
int __usercall mw_api_hashing_algo@<eax>(int a1@<edx>, int a2@<ecx>, _BYTE *a3)
{
  _BYTE *ptr_provided_name; // ebx
  int hash; // [esp+1Ch] [ebp+8h]

  ptr_provided_name = a3;
  nullsub_1(a2, a1, a3);
  for ( hash = 0; *ptr_provided_name; ++ptr_provided_name )
    hash = (hash << 0x10) + (hash << 6) + (char)*ptr_provided_name - hash;
  return hash;
}
```

[Figure 57] Subroutine containing the API hashing algorithm

- As readers can notice it, it's the **same algorithm of DLL hashing**, **but without having lines to convert eventual upper case letters to lower case.**
- Finally, let's return to the **sub_B9B558 subroutine** and rename it to **mw_api_hash_resolving**.

We've done our quick analysis of subroutines directly related to DLL and API hashing, but our analysis so far is only of a very small piece of the puzzle because, for example, we don't have any API name yet.

There're two ways to handle this issue:

- We can use a plugin like **HashDB** to help us and, no doubts, it will save time during our analysis.
- We could **write our own script to handle all API hashes** and find the associated names.

Using a plugin, mainly during working days, it's the recommended approach. However, there're small side effects that, eventually, might be not suitable for you:

- You need an **Internet connection** to the **HashDB plugin** to communicate with OALabs servers and, in critical premises, this access could be not available or allowed.

- **HashDB** could not have the wished **algorithm** for that particular sample / malware family and you would need to write a script to manage API hash resolving anyway.

In this article I'm going to continue using HashDB, but eventually I will show how doing your own script to calculate and markup the **idb file** from IDA Pro in future articles.

Returning to **sub_B8645E subroutine**, we had the following:

```
1 int __thiscall sub_B8645E(void *this)
2 {
3   int (__cdecl *v1)(int, int, _DWORD, int, int, int, int); // eax
4
5   v1 = (int (__cdecl *)(int, int, _DWORD, int, int, int, int))sub_B9BFF0(0x303, (int)this, (int)this, 0x76FC34E6);
6   return v1(0xB7D5AF, 0xCB85E0, 0, 0xEFD1B, 0x7F43F, 0xFC523, 0x3BD7C);
7 }
```

[Figure 58] sub_B8645E subroutine

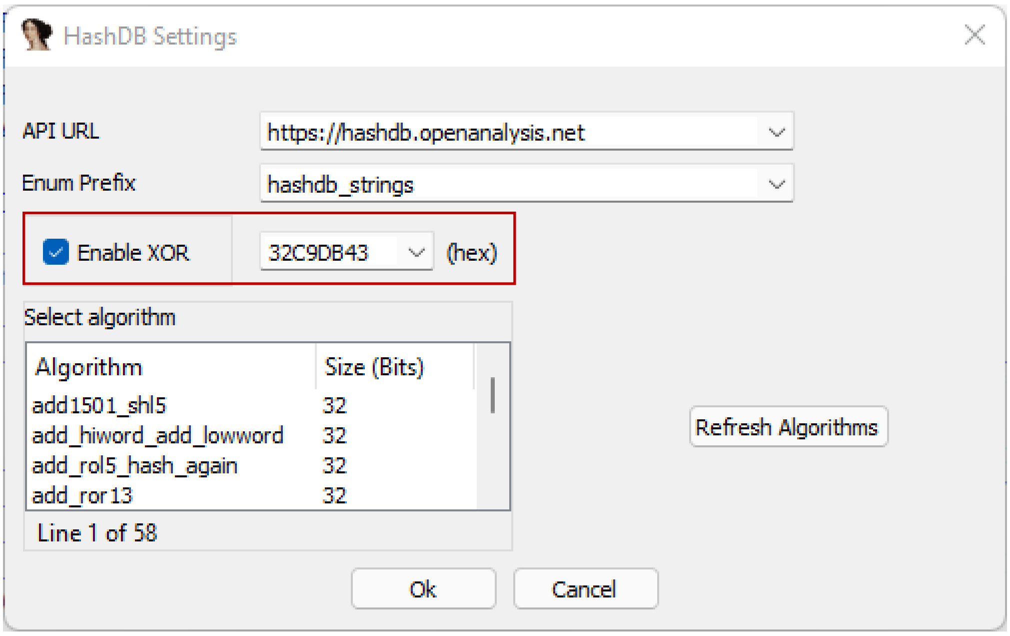
We already know that:

- The prototype at the first line is a stub/proxy used for passing useful arguments when required.
- The first argument of the subroutine is a sort of index used to locate the API in the "API table".
- The last argument is the API hash.
- On **line 7**, the resolved API (**from line 6**) doesn't have any real argument and all supposed arguments are garbage (in this specific function in the image above).
- As we explained on **page 38**, the **XOR key** for decrypting API hashes is **0x32C9DB43.**

Therefore, before proceeding the decryption, we must to set the XOR key on HashDB, so readers has two options:

- Return to **sub_B9B558 subroutine (Figure 54)**, right click the XOR value and choose "**HashDB set XOR key**".

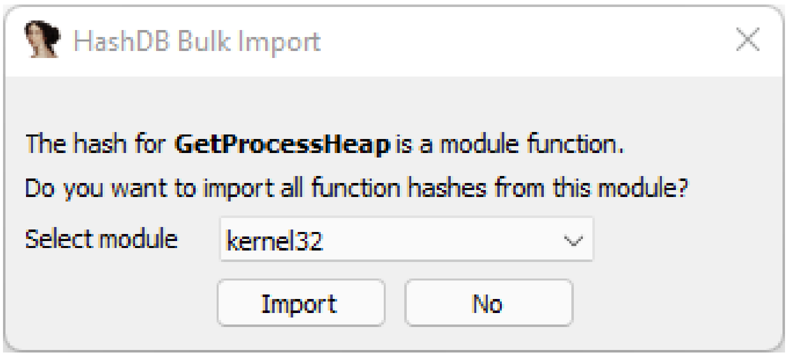
- Go to **Edit → Plugins → HashDB** and **set the XOR key** (shown below).



[Figure 59] sub_B8645E subroutine

After having set up the **XOR key** (take care: not always exist a XOR key), you should do the following:

- **Right-click the API hash** and choose **HashDB Hunting Algorithm**.
- Probably one or two algorithms will be returned, but as we know that is an Emotet sample, so mark "**emotet**".
- **Right click the API hash** again and choose **HashDB Lookup**.
- You'll see the following window and choose **kernel32**:



[Figure 60] HashDB Bulk Import

- Click on the **Import button** and **wait** few seconds until the hash importing task has been finished.
- If you go to **Enumerations view** (**SHIFT + F10**) you'll see something similar to the following image:

```
FFFFFFFF ; enum hashdb_strings_emotet, mappedto_38, width 4 bytes
FFFFFFFF RtlUnicodeStringToOemString_0  = 2272Ah
FFFFFFFF LocalFree_0      = 116374h
FFFFFFFF WriteConsoleOutputAttribute_0  = 157EE0h
FFFFFFFF NtQueryMultipleValueKey_0  = 25CCE3h
FFFFFFFF GetProcessDEPPolicy_0  = 28DC4Bh
FFFFFFFF IsTimeZoneRedirectionEnabled_0  = 32543Bh
FFFFFFFF AddLocalAlternateComputerNameA_0  = 33A648h
FFFFFFFF AddLocalAlternateComputerNameW_0  = 33A662h
FFFFFFFF ZwIsProcessInJob_0  = 3C8713h
FFFFFFFF RtlCompareUnicodeStrings_0  = 704963h
FFFFFFFF GetNumaHighestNodeNumber_0  = 9CB81Dh
FFFFFFFF ZwAlpcCancelMessage_0  = 0B0DC71h
FFFFFFFF ZwLockProductActivationKeys_0  = 0BAF072h
FFFFFFFF GetLocalTime_0    = 0D9BFC1h
FFFFFFFF NtOpenThreadToken_0  = 0E7CE7Ch
FFFFFFFF GetConsoleInputWaitHandle_0  = 1009405h
FFFFFFFF RtlInterlockedFlushSList_0  = 13F47CAh
FFFFFFFF RtlDllShutdownInProgress_0  = 14D5901h
FFFFFFFF ZwCreateTransactionManager_0  = 15B49EBh
```

[Figure 61] Part of an enumeration created by HashDB

- Put the cursor on **sub_B9BFF0 subroutine**, press "**Y hotkey**" and **change the type of the last argument**, which is the API hash, to **hashdb_strings_emotet** (the name of the enumeration as shown in the figure above):

    - **int __cdecl sub_B9BFF0(int, int, int, hashdb_strings_emotet)**

- **Press F5**.

You should see the following result:

```
int __thiscall sub_B8645E(void *this)
{
  int (__cdecl *v1)(int, int, _DWORD, int, int, int, int); // eax

  v1 = (int (__cdecl *)(int, int, _DWORD, int, int, int, int))sub_B9BFF0(0x303, (int)this, (int)this, GetProcessHeap_0);
  return v1(0xB7D5AF, 0xCB85E0, 0, 0xEFD1B, 0x7F43F, 0xFC523, 0x3BD7C);
}
```

[Figure 62] API hash resolved to GetProcessHeap_0

Finally:

- rename **v1** to **GetProcessHeap.**
- rename **sub_B8645E** to **mw_GetProcessHeap.**
- rename the API hash resolving routine (**sub_B9BFF0**) to **mw_api_resolving**.

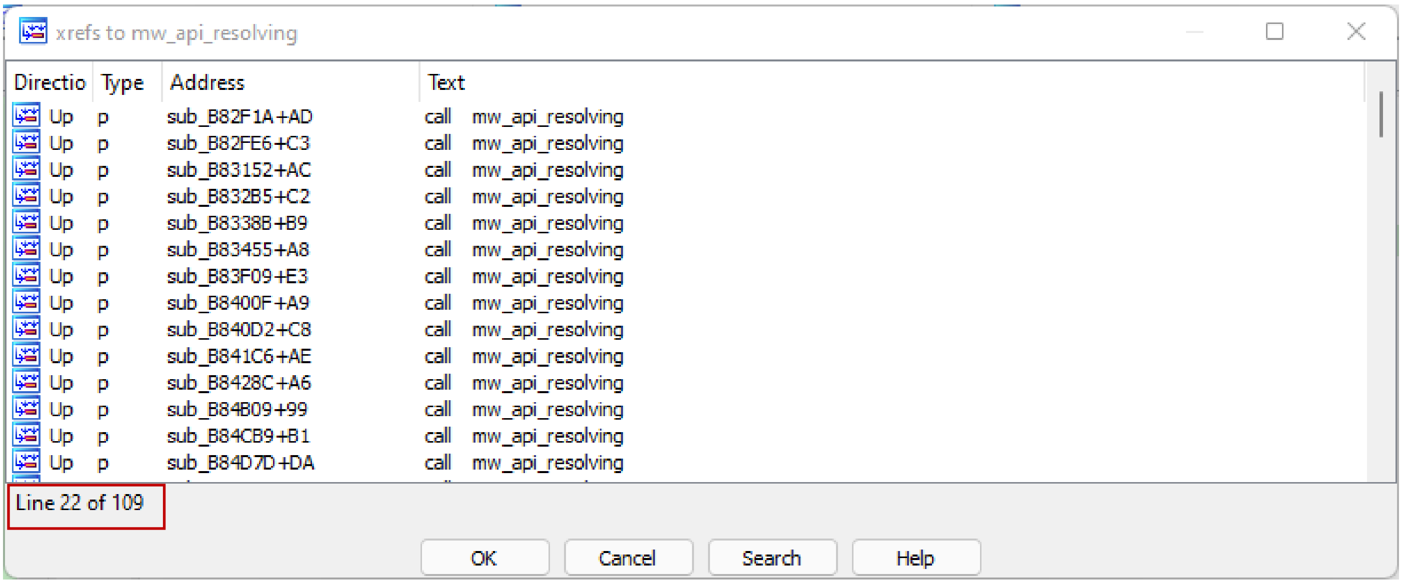
You you have something like:

```
int __thiscall mw_GetProcessHeap(void *this)
{
  int (__cdecl *GetProcessHeap)(int, int, _DWORD, int, int, int, int); // eax

  GetProcessHeap = (int (__cdecl *)(int, int, _DWORD, int, int, int, int))mw_api_resolving(
                                                                               771,
                                                                               (int)this,
                                                                               (int)this,
                                                                               GetProcessHeap_0);
  return GetProcessHeap(12047791, 13338080, 0, 982299, 521279, 1033507, 245116);
}
```

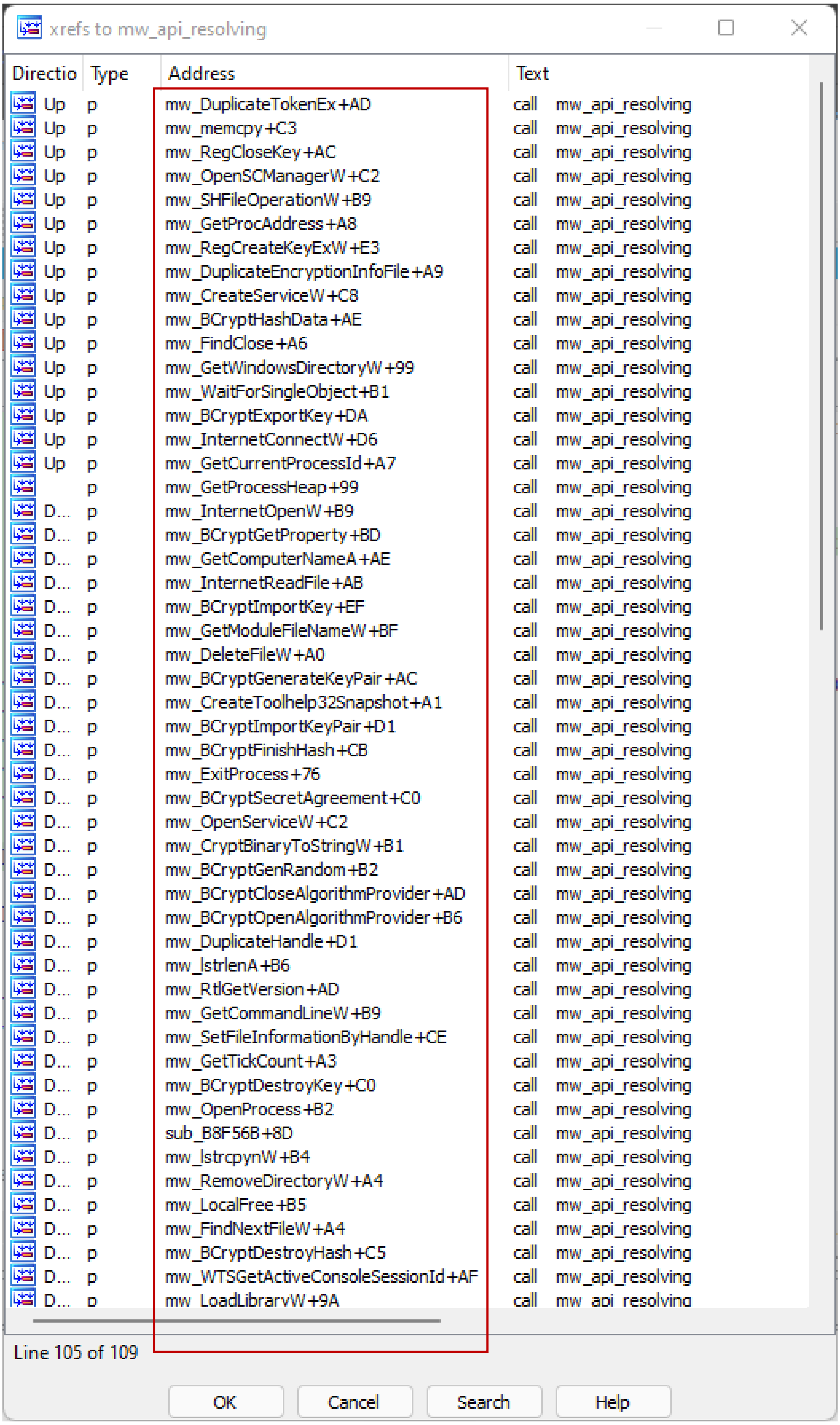
[Figure 63] API hash resolved and renamed

That's ok! The problem is that there're many other API hashes in the code and we need to apply the same procedure for all of them. Keep the cursor on **mw_api_resolving** and **press X** to list **all references:**



[Figure 64] References to mw_api_resolving subroutine

Yes, **there're 109 references**, unfortunately. After you have resolved all API hashes, you'll have:



xrefs to mw_api_resolving

Directio	Type	Address	Text
Up	p	mw_DuplicateTokenEx+AD	call mw_api_resolving
Up	p	mw_memcpy+C3	call mw_api_resolving
Up	p	mw_RegCloseKey+AC	call mw_api_resolving
Up	p	mw_OpenSCManagerW+C2	call mw_api_resolving
Up	p	mw_SHFileOperationW+B9	call mw_api_resolving
Up	p	mw_GetProcAddress+A8	call mw_api_resolving
Up	p	mw_RegCreateKeyExW+E3	call mw_api_resolving
Up	p	mw_DuplicateEncryptionInfoFile+A9	call mw_api_resolving
Up	p	mw_CreateServiceW+C8	call mw_api_resolving
Up	p	mw_BCryptHashData+AE	call mw_api_resolving
Up	p	mw_FindClose+A6	call mw_api_resolving
Up	p	mw_GetWindowsDirectoryW+99	call mw_api_resolving
Up	p	mw_WaitForSingleObject+B1	call mw_api_resolving
Up	p	mw_BCryptExportKey+DA	call mw_api_resolving
Up	p	mw_InternetConnectW+D6	call mw_api_resolving
Up	p	mw_GetCurrentProcessId+A7	call mw_api_resolving
	p	mw_GetProcessHeap+99	call mw_api_resolving
D...	p	mw_InternetOpenW+B9	call mw_api_resolving
D...	p	mw_BCryptGetProperty+BD	call mw_api_resolving
D...	p	mw_GetComputerNameA+AE	call mw_api_resolving
D...	p	mw_InternetReadFile+AB	call mw_api_resolving
D...	p	mw_BCryptImportKey+EF	call mw_api_resolving
D...	p	mw_GetModuleFileNameW+BF	call mw_api_resolving
D...	p	mw_DeleteFileW+A0	call mw_api_resolving
D...	p	mw_BCryptGenerateKeyPair+AC	call mw_api_resolving
D...	p	mw_CreateToolhelp32Snapshot+A1	call mw_api_resolving
D...	p	mw_BCryptImportKeyPair+D1	call mw_api_resolving
D...	p	mw_BCryptFinishHash+CB	call mw_api_resolving
D...	p	mw_ExitProcess+76	call mw_api_resolving
D...	p	mw_BCryptSecretAgreement+C0	call mw_api_resolving
D...	p	mw_OpenServiceW+C2	call mw_api_resolving
D...	p	mw_CryptBinaryToStringW+B1	call mw_api_resolving
D...	p	mw_BCryptGenRandom+B2	call mw_api_resolving
D...	p	mw_BCryptCloseAlgorithmProvider+AD	call mw_api_resolving
D...	p	mw_BCryptOpenAlgorithmProvider+B6	call mw_api_resolving
D...	p	mw_DuplicateHandle+D1	call mw_api_resolving
D...	p	mw_lstrlenA+B6	call mw_api_resolving
D...	p	mw_RtlGetVersion+AD	call mw_api_resolving
D...	p	mw_GetCommandLineW+B9	call mw_api_resolving
D...	p	mw_SetFileInformationByHandle+CE	call mw_api_resolving
D...	p	mw_GetTickCount+A3	call mw_api_resolving
D...	p	mw_BCryptDestroyKey+C0	call mw_api_resolving
D...	p	mw_OpenProcess+B2	call mw_api_resolving
D...	p	sub_B8F56B+8D	call mw_api_resolving
D...	p	mw_lstrcpynW+B4	call mw_api_resolving
D...	p	mw_RemoveDirectoryW+A4	call mw_api_resolving
D...	p	mw_LocalFree+B5	call mw_api_resolving
D...	p	mw_FindNextFileW+A4	call mw_api_resolving
D...	p	mw_BCryptDestroyHash+C5	call mw_api_resolving
D...	p	mw_WTSGetActiveConsoleSessionId+AF	call mw_api_resolving
D...	p	mw_LoadLibraryW+9A	call mw_api_resolving

Line 105 of 109

OK Cancel Search Help

[Figure 65] All API hashes resolved and respective subroutines renamed

Certainly, the routine of **marking up process, resolving all these API hashes and renaming functions**, though take a huge time, it makes the analysis much easier. Additionally, I searched for all necessary APIs on MSDN and renamed all arguments for each of these resolved APIs to the correct name:

```
int __usercall mw_GetTempFileNameW@<eax>(
        int lpTempFileName@<edx>,
        int a2@<ecx>,
        int a3,
        int a4,
        int a5,
        int uUnique,
        int lpPathName,
        int lpPrefixString)
{
  int (__stdcall *GetTempFileNameW)(int, int, int, int); // eax

  nullsub_1(a2, lpTempFileName, a3);
  GetTempFileNameW = (int (__stdcall *)(int, int, int, int))mw_api_resolving(0x75, 0x63, 0x63, GetTempFileNameW_0);
  return GetTempFileNameW(lpPathName, lpPrefixString, uUnique, lpTempFileName);
}
```

[Figure 66] All API hashes resolved and respective subroutines renamed

Following the same steps we took to extract and decode strings, let's examine the content of **the .data section** using **CTRL+S hotkey** and going to there:

```
BA4000 dword_BA4000    dd 176A19CCh, 176A18ACh, 6D4B52FFh, 166AA2CDh, 125AE376h
BA4000                                         ; DATA XREF: sub_BA225A+32A↑o
BA4000                 dd 166A49CCh, 614D6E64h, 166AA2CDh, 603B8D03h, 166A89D3h
BA4000                 dd 7CC6100Eh, 166A89D3h, 8C22DD47h, 166A89D3h, 47A63682h
BA4000                 dd 166AA2CDh, 0AB875C53h, 166AA2CDh, 7FA95EE1h, 166A89D3h
BA4000                 dd 0B0003CFAh, 166A89D3h, 9DE8B175h, 166AA2CDh, 0A69E35E9h
BA4000                 dd 166A89D3h, 5973A175h, 166A89D3h, 18C28D75h, 166A89D3h
BA4000                 dd 90AADE4Ch, 166A89D3h, 9ABB22E9h, 166A89D3h, 0BEA630ABh
BA4000                 dd 166A89D3h, 0CBC28D75h, 166A89D3h, 6F5033ABh, 166AB1D7h
BA4000                 dd 6A233782h, 166AA2CDh, 0ED37AE88h, 166AA2CDh, 55834372h
BA4000                 dd 166AA2CDh, 0A6EE21C9h, 166A89D3h, 84D8B2F2h, 166A89D3h
BA4000                 dd 0A9083508h, 166A89D3h, 1990DC64h, 166A49CCh, 8253338Eh
BA4000                 dd 166AA2CDh, 0D5978DF7h, 166AA2CDh, 27549AA4h, 166A89D3h
BA4000                 dd 70DE573h, 166A49CCh, 6D8E3CFAh, 166AA2CDh, 0B2C6C094h
BA4000                 dd 166A89D3h, 3085540Fh, 166A89D3h, 0D9EA65B8h, 166A89D3h
BA4000                 dd 7CBB7191h, 166A89D3h, 41227BBAh, 166AA2CDh, 0D8E5AF15h
BA4000                 dd 166AA2CDh, 440B739Bh, 166AB1D7h, 99BB201Eh, 166A89D3h
BA4000                 dd 0AE983FFAh, 166AA2CDh, 34F8830Fh, 166AA2CDh, 39B28007h
BA4000                 dd 166AA2CDh, 5908DE0Ah, 166A89D3h, 0DC29CF99h, 166A89D3h
BA4000                 dd 6730C7FEh, 13C30863h, 376F5D2Ch, 0C3067705h, 92766931h
BA4000                 dd 4A94AB31h, 55FE408h, 0F3D53973h, 8AF26DF6h, 2F7B31B8h
BA4000                 dd 1B073FA4h, 44F29992h, 0A39B4AF0h, 15268244h, 0D53092F3h
BA4000                 dd 53484B7h, 4C4B779h, 0DD9FDFE6h, 0EF56E861h, 16AFFEEDh
BA4000                 dd 0D58F6E3Ah, 20410F68h, 0E45645B8h, 0B3468EDAh, 0E2F87AC2h
BA4000                 dd 4B6D8A9h, 0E06D8F59h, 6E840B57h, 0A7D16839h, 598AC20Bh
BA4000                 dd 51566204h, 22763359h, 0B0563294h, 78128B11h, 5784087Eh
BA4000                 dd 98317209h, 1DF55F28h, 3F4A8F37h, 2800201Bh, 218187B6h
BA4208 dword_BA4208    dd 6D0500h              ; DATA XREF: sub_B84700+386↑r
BA4208                                         ; sub_B84700+39F↑r ...
BA420C dword_BA420C    dd 0                    ; DATA XREF: sub_B83210+7F↑r
BA420C                                         ; sub_B83210+8C↑r ...
BA4210 dword_BA4210    dd 0                    ; DATA XREF: sub_B85C9A+5F9↑r
```

[Figure 67] Content of .data section

It's interesting to notice that the data blob ends with **double "\x00"**, so we're are going to use this fact later.

Following the data cross-reference at the start of the .data section, we have lines of code from this subroutine (**sub_BA225A**) as shown below:

```
      {
        v19 = sub_B9ACFF(0x375B8, 0xF73AC, 0xF6F4C, 0xB942, dword_BA4000, &v18);
        v1 = v19;
        v0 = 0x909989E;
        v3 = v18 + v19;
        v17 = v18 + v19;
        goto LABEL_2;
      }
```

[Figure 68] Code from sub_BA225A referring to .data section's bytes

If you examine the content of **sub_B9ACFF( )**, you'll find the following code:

```
int __usercall sub_B9ACFF@<eax>(int a1@<edx>, int a2@<ecx>, int a3, int a4, _DWORD *a5, int *a6)
{
  char *v6; // esi
  int v7; // edx
  unsigned int v8; // edi
  int v9; // ebp
  char *v10; // ecx
  unsigned int v11; // edi
  unsigned int v12; // edx
  int v13; // ecx
  int v15; // [esp+14h] [ebp-8h]
  int v16; // [esp+18h] [ebp-4h]

  nullsub_1(a2, a1, a3);
  v6 = (char *)(a5 + 2);
  v7 = *a5 ^ a5[1];
  v15 = *a5;
  v16 = v7;
  v8 = v7;
  if ( (v7 & 3) != 0 )
    v8 = (v7 & 0xFFFFFFFC) + 4;
  v9 = mw_Heap((void *)v8);
  if ( v9 )
  {
    v10 = &v6[4 * (v8 >> 2)];
    v11 = 0;
    v12 = (unsigned int)(v10 - v6 + 3) >> 2;
    if ( v6 > v10 )
      v12 = 0;
    if ( v12 )
    {
      v13 = v9 - (_DWORD)v6;
      do
      {
        ++v11;
        *(_DWORD *)&v6[v13] = v15 ^ *(_DWORD *)v6;
        v6 += 4;
      }
```

[Figure 69] First lines of sub_B9ACFF subroutine

That's exactly the same decrypting routine used for decoding strings, but in this **case bytes that are stored within the .data section aren't strings**. Analyzing additional lines of code from previous subroutine (**sub_BA225A**), we have:

```
42 LABEL_15:
43       v3 = v17;
44     }
45     if ( v0 == 0x2FD4739 )
46     {
47       v5 = *(_BYTE *)v1;
48       v6 = *(_BYTE *)(v1 + 1);
49       v14 = *(_BYTE *)(v1 + 2);
50       v15 = *(_BYTE *)(v1 + 3);
51       v7 = sub_B84BB4(0x3FFB6, (int)"%u.%u.%u.%u", 0xC14DA);
52       mw_snwprintf(0x71679, 0x7882F, 0x5F67, v15, 0x2A5F9, 0x7E69F, v14, 0x18939, v2 + 0x10, v6, 0x10, v7, 0xFF226, v5);
53       mw_w_GetFreeHeap(0xFE047, 0xDDC38, v7);
54       *(_WORD *)(v2 + 0x42) = _byteswap_ushort(*(_WORD *)(v1 + 4));
55       v8 = *(_BYTE *)(v1 + 6);
56       v9 = *(_BYTE *)(v1 + 7);
57       v1 += 8;
58       v10 = v8;
59       v11 = v9;
60       v0 = 0x7F8B24;
61       *(_WORD *)(v2 + 0x40) = v11 | (v10 << 8);
```

[Figure 70] Further lines of code from sub_BA225A, which holds references to .data section

This is a typical code used for **IP address formatting** and the string "**%u.%u.%u.%u**" on **line 51** confirms that we're right. Therefore, all bytes stored from **0x00ba4000** to **0x00ba4208** from **.data section are encrypted/encoded IP addresses.**

I wrote a simple Python script, without using IDA Python or IDC instructions, **to extract, decode and format all IP addresses used as C2 by Emotet**. This **script assumes that encrypted IP address are stored at start of the .data section** and, just in case it changes, so it's quite simple to adapt it.

```python
import binascii
import pefile
import struct
import ipaddress

# This routine implements the XOR operation and take the key's size into account.
def decrypter(data_key, data_string, stringlength):
    decoded = []
    for i in range(0, stringlength):
        decoded.append((data_string[i]) ^ (data_key[i % len(data_key)]))
    return decoded

# This routine extracts data from .data section.
def extract_data(filename):
    pe=pefile.PE(filename)
    imagebase = pe.OPTIONAL_HEADER.ImageBase
    for section in pe.sections:
        if '.data' in section.Name.decode(encoding='utf-8').rstrip('x00'):
            return (section.get_data(section.VirtualAddress, section.SizeOfRawData),
                    (section.VirtualAddress + imagebase))

# This routine calculates the offset between the current address of the targeted
# data and the start address of the .data section section.
def calc_offsets(x_seg_start, x_start):

    data_offset = hex(int(x_start,16) - int(x_seg_start,16))
    return data_offset
```

[Figure 71] Script to extract, decrypt and formatting C2 IP Addresses (first part)

```python
def data_decrypter():

    # Next two lines extracts .data section's information.
    filename = r"C:\Users\Administrador\Desktop\MAS\mas_3\mas_3_unpacked\mas_3_unpacked.bin"
    data_extracted, virtualaddress = extract_data(filename)

    # Convert the (image base + section rva) to hexadecimal.
    encrypted_string_addr = hex(virtualaddress)

    # Next two lines calculate the offset between text blob and start of the .text segment.
    # In this specific case, the result will be zero because we're extracting data from the
    # start of the segment. However, in many cases, it isn't true. Anyway, I kept the routine
    # calc_offsets just in case readers need to use it.
    encr_data_rel = calc_offsets(encrypted_string_addr, encrypted_string_addr)

    # Next two lines calculates the size of the encrypted string table. Pay attention that
    # I've used the approach of searching up to a marker, which is this case is "\x00\x00".
    d1_off = 0x0
    if (b'\x00\x00' in data_extracted[int(encr_data_rel,16):]):
        d1_off = (data_extracted[int(encr_data_rel,16):]).index(b'\x00\x00')

    bytes_extracted = (data_extracted[int(encr_data_rel,16):int(encr_data_rel,16) + d1_off])

    # Uncomment the next line whether you need to print the extracted bytes in hexdecimal
    # to confirm the extraction is correct. Additionally, you can match this result against
    # SHIFT+E from IDA Pro.
    #print(binascii.b2a_hex(bytes_extracted))

    # In the next lines I'm going to extract the XOR key, the XORed string length and
    # the encrypted blob. No doubts, the advantage of using struct.unpack() routine is that
    # the little indian issue and data's representation are already handled.
    offset = 0
    xorkey = bytes_extracted[offset:(offset+4)]
    xorkey_unpacked = struct.unpack('<I', xorkey)[0]
    xored_length = bytes_extracted[(offset+4):(offset+4+4)]
    xored_length_unpacked = struct.unpack('<I', xored_length)[0]
    string_length = xorkey_unpacked  ^ xored_length_unpacked
    encrypted_string = bytes_extracted[8:8 + string_length]
    decoded_bytes = bytes(decrypter(xorkey, encrypted_string, string_length))

    # Next 10 lines of code are responsible for extracting and presenting
    # the C2 IP Address list.
    # The format is: [4 bytes for IP address][2 bytes of port]
    print('\nC2 IP ADDRESS LIST:')
    print(30 * '-')
    k = 0
    i = 0
    while (k < len(decoded_bytes)):
        ip_item = decoded_bytes[k:k+4]
        ip_port = decoded_bytes[k+4:k+6]
        print("IP[%d]: %s" % (i,ipaddress.IPv4Address(ip_item)),end=':')
        print(int(binascii.hexlify(ip_port),16))
        k = k + 8
        i = i + 1

```

```
def main():

    # Call the string_decrypter routine
    data_decrypter()
    return

if __name__ == '__main__':
    main( )
```

[Figure 72] Script to extract, decrypt and formatting C2 IP Addresses (second and last part)

```
C2 IP ADDRESS LIST:
------------------------------
IP[0]: 51.75.33.122:443
IP[1]: 186.250.48.5:80
IP[2]: 168.119.39.118:443
IP[3]: 207.148.81.119:8080
IP[4]: 194.9.172.107:8080
IP[5]: 139.196.72.155:8080
IP[6]: 78.47.204.80:443
IP[7]: 159.69.237.188:443
IP[8]: 45.71.195.104:8080
IP[9]: 54.37.106.167:8080
IP[10]: 185.168.130.138:443
IP[11]: 37.44.244.177:8080
IP[12]: 185.184.25.78:8080
IP[13]: 185.148.168.15:8080
IP[14]: 128.199.192.135:8080
IP[15]: 37.59.209.141:8080
IP[16]: 103.41.204.169:8080
IP[17]: 185.148.168.220:8080
IP[18]: 103.42.58.120:7080
IP[19]: 78.46.73.125:443
IP[20]: 68.183.93.250:443
IP[21]: 190.90.233.66:443
IP[22]: 5.56.132.177:8080
IP[23]: 62.171.178.147:8080
IP[24]: 196.44.98.190:8080
IP[25]: 168.197.250.14:80
IP[26]: 66.42.57.149:443
IP[27]: 59.148.253.194:443
IP[28]: 104.131.62.48:8080
IP[29]: 191.252.103.16:80
IP[30]: 54.37.228.122:443
IP[31]: 88.217.172.165:8080
IP[32]: 195.77.239.39:8080
IP[33]: 116.124.128.206:8080
IP[34]: 93.104.209.107:8080
IP[35]: 118.98.72.86:443
IP[36]: 217.182.143.207:443
IP[37]: 87.106.97.83:7080
IP[38]: 210.57.209.142:8080
IP[39]: 54.38.242.185:443
IP[40]: 195.154.146.35:443
IP[41]: 203.153.216.46:443
IP[42]: 198.199.98.78:8080
IP[43]: 85.214.67.203:8080
```

[Figure 73] Extracted C2 IP addresses: exactly equal to Triage's output (Figure 03 – page 03)

5. Conclusion

This article follows the same educational path from first articles and the choice for the Emotet is due the fact it offers interesting concepts and tasks such as extracting and decrypting strings and C2 IP addresses. In the other side, analyzing the entire malware can take a significant time because of control flow flattening obfuscation, but it isn't hard. Probably we'll return to this topic in the future when analyzing similar malware samples.

My goal continue being to offer a review of malware analysis to make possible for that reverse engineers can learn something new, have a sort of guideline to follow and source of research when and whether it's necessary. Of course, it isn't a course about malware analysis, but I think it could be helpful by offering something really applied and practical, which tries to explain taken decisions and how to proceed when analyzing similar contexts.

I could have chosen a more complex malware sample, but it wasn't the idea. If the final objective is writing a series of articles explaining important concepts, strategies, techniques and approaches used during malware analysis of different threats, so proposing hard samples wouldn't help anyone and it would be useless, in my opinion.

Probably this article will have errors, but it isn't big deal. Soon I find them, I'll release a new revision of this document.

6. Acknowledgments

I'd like to publicly thank **Ilfak Guilfanov (@ilfak)** and **Hex-Rays (@HexRaysSA)** for supporting this project by providing me with a personal license of the IDA Pro.

My gratitude is endless because certainly I couldn't keep writing this series without a personal license (not depending on corporate licenses). Honestly, I don't have enough words to say how much I got happy in last JAN/06/2022 when he replied my message and agreed with this project. As I promised him, I will keep writing this series of articles for the next months and years.

Once again: **thank you for everything, Ilfak.**

Just in case you want to keep in touch:

- **Twitter: @ale_sp_brazil**
- **Blog:** https://exploitreversing.com

Keep reversing and I see you at next time!

Alexandre Borges