Malware Analysis Series (MAS): Article 2

by Alexandre Borges
release date: FEB/03/2022 - revision: A

1. Introduction

Welcome to the second article in the ***MAS (Malware Analysis Series).*** In the previous article (https://exploitreversing.files.wordpress.com/2021/12/mas_1_rev_1.pdf) we reviewed few relevant concepts about malware analysis such as unpacking and code injection, which are techniques used for evading and keeping the threat undetected by usual security defenses. Of course, there aren't only these obstacles to circumvent during a reverse engineering session and, as you will learn during this series of articles, we usually have to bypass string encoding (not necessarily encryption), API resolving, anti-analysis techniques (debugging detection techniques, anti-disassembly techniques and virtual machine detection, for example) and different encryption/encoding tricks used by adversaries while analyzing malicious code.

In this current part of MAS we are analyzing **Qakbot** (which is a simple family), but we don't only be focusing on the point of view of threat hunting as it could be expected. This article aims to explain on string decryption, API resolving, C++ structures and C2 data extraction. These procedures can be seem strange for you at first time, but certainly you will get experienced with them as you've analyzed other samples.

Nowadays is quite rare finding malware threats that aren't packed, so that's a good reason to list (last time) important breakpoints to set up while unpacking native code:

- **CreateProcessInternalW( )**
- **VirtualAlloc( ) | VirtualAllocEx( )**
- **VirtualProtect( ) | ZwProtectVirtualMemory( )**
- **WriteProcessMemory( ) | NtWriteProcessMemory( )**
- **ResumeThread( ) | NtResumeThread( )**
- **CryptDecrypt( ) | RtlDecompressBuffer( )**
- **NtCreateSection( )**
- **NtMapViewOfSection( ) | ZwMapViewOfSection( )**
- **UnmapViewOfSection( ) | ZwUnmapViewOfSection( )**
- **NtWriteVirtualMemory( )**
- **NtReadVirtualMemory( )**

As the reader will learn later, the same breakpoints might be useful even while unpacking .NET binaries (next articles) because high-level functions used by programs (native or managed ones) are translated into lower level functions still on userland and, afterwards, they invoke their counterparts on the kernel land.

As certainly readers remember about, main functions used for allocating memory in native binary are:

- **VirtualAlloc( ) → VirtualAllocEx( ) → NtAllocateVirtualMemory( )**

- **HeapCreate( ) → RltCreateHeap( ) → NtAllocateVirtualMemory( )**
- **GlobalAlloc( ) → RtlAllocateHeap( ) → NtAllocateVirtualMemory( )**
- **LocalAlloc( ) → RtlAllocateHeap( ) → NtAllocateVirtualMemory( )**
- **new( ) → HeapAlloc( ) → RtlAllocateHeap( ) → NtAllocateVirtualMemory( )**
- **malloc( ) → HeapAlloc( ) → RtlAllocateHeap( ) → NtAllocateVirtualMemory( )**

This sequence of calls can be also found in "*The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory*" book (by Michael Hale Ligh, Andrew Case, Jamie Levy and Aaron Walters). So, it's appropriate to review them:

- **VirtualAlloc( ):** this function is able to reserve, commit and change state of a region of pages in the virtual address space and it also initializes this region to zero. Furthermore, it is usually used for allocation large regions of memory as, for example, to inject a PE file. It's part of kernel32.dll.

- **VirtualAllocEx( ):** its similar to *VirtualAlloc( )*, but it's able to allocate memory in a remote process. It is part of kernel32.dll.

- **NtAllocateVirtualMemory( ):** this function it's similar to *VirtualAllocEx( )* and it can be called from user-space or kernel-space. This API is part of ntoskrnl.exe.

- **HeapCreate( ):** this function creates a private heap object. This API is part of kernel32.dll and it's usually suitable for small allocations.

- **RtlCreateHeap( ):** this function creates a private heap object that can used by the calling process. This API is found in ntdll.dll (user mode) and ntoskrnl.exe (kernel mode).

- **HeapAlloc( ):** this function, which is found in kernel32.dll, allocates a block of memory from a heap.

- **GlobalAlloc( ):** this function allocates a specified number of bytes from the heap. This API is part of the kernel32.dll.

- **LocalAlloc( ):** this function, which is part of the kernel32.dll, allocates a specified number of bytes from the heap. Additionally, as Windows Memory Manager doesn't provide a separate local heap and global heap, so this function and *GlobalAlloc( )* have the same functionality.

- **RtlAllocateHeap( ):** this routine allocates a block of memory from a heap. It's part of the ntoskrnl.exe.

- **new( ):** this operator allocates and initializes an object or array of objects, returning a pointer to the object. This API is part of the msvcrt.dll.

- **malloc( ):** this function allocates memory blocks. It's part of the msvcrt.dll

Of course, there're other memory APIs that could be used to allocate a memory region or even mapping a file into memory, but they are eventually you might now to be familiarized with them:

- **HeapReAlloc( ):** this function is able to resize the allocated heap region, but it preserves the current data there. It's part of the kernel32.dll.

- **VirtualAlloc2( ):** this API allocates a region memory to the current or remote process. This function uses an interesting array of MEM_EXTENDED_PARAMETER structures and, as the *VirtualAlloc( )*, it's part of the kernel32.dll.

- **CreateFileMapping( ):** this function creates or opens a file mapping object, and it's part of the kernel32.dll.

- **CreateFileMapping2( ):** this function, which was introduced on Windows 10, creates or opens a file mapping object for a specified file, and it's part of the kernel32.dll

- **MapViewOfFile( ):** this function takes a handle from *CreateFileMapping( )* and maps the file (totally or partially) into the process address space. It's part of kernel32.dll.

- **MapViewOfFileEx( ):** this function maps a file into the address space of a calling process and provides the option to specify the base memory address. It's found in the kernel32.dll.

- **MapViewOfFile2( ):** this function, introduced on Windows 10, maps a view of a file into the address space of the specified process. This function is present in mincore.lib, but it's also exported by kernelbase.dll.

- **MapViewOfFIle3( ):** this function, introduced on Windows 10, maps a file into the address space of the specified process and one of its parameters also uses an array of MEM_EXTENDED_PARAMETER structures. This function is exported from kernelbase.dll.

As you can check out, there isn't an infinite array of possibilities for allocating memory and, eventually, even managed code ends passing any memory allocation through one of these APIs above, which makes learning about these functions a fundamental knowledge to unpack native and managed Windows binary as you'll learn in this series of articles.

2. Lab Setup

During this article we'll using the following initial environment:

- **Windows 7 (x86) and Windows 8.1 (x64) or Windows 10 (x64):** If you need a Windows 10 virtual machine, Microsoft continue offering one with expiration time on this website: https://developer.microsoft.com/en-us/windows/downloads/virtual-machines/

- **REMnux:** it's a Ubuntu-derived distribution used for reversing engineering: https://docs.remnux.org/install-distro/get-virtual-appliance

- **x64dbg:** it's a modern debugger composed by x32dbg and x64dbg: https://x64dbg.com/#start

- **IDA Home / IDA Pro** (the best tool for reverse engineering, by far): https://hex-rays.com/ida-pro/#main-differences-between-ida-editions

- **HxD** is an excellent hex-editor that we could be used, for example, to check and fix PE headers manually. It can be downloaded from: https://mh-nexus.de/en/hxd/

- **PEBear** is used to visualize details of a PE Header and fix many binary issues. You can download it tool from: https://github.com/hasherezade/pe-bear-releases

- **Pestudio** is mainly used to triage and collect different information of a potential malware. The tool (free and paid versions) is available here: https://www.winitor.com/features

- **Malwoverview:** https://github.com/alexandreborges/malwoverview

- **Resource Hacker:** http://www.angusj.com/resourcehacker/

If you want to setup a good environment for analysis, you can use **Flare VM** (https://github.com/mandiant/flare-vm), which provides you with several good reversing engineering tools. To set up your virtual machine with **FlareVM**:

a. Install or download a virtual machine running **Windows 10**:

    a. https://developer.microsoft.com/en-us/microsoft-edge/tools/vms/
    b. https://developer.microsoft.com/en-us/windows/downloads/virtual-machines/

b. Make sure that **.NET 4.5** and **PowerShell 5.1** are installed:

    a. (.NET 4.5) https://www.microsoft.com/en-us/download/details.aspx?id=30653
    b. (PowerShell) https://www.microsoft.com/en-us/download/details.aspx?id=54616

c. Make sure your VM has **60 GB free file system** and **2 GB RAM**.
d. Disable **Windows Defender**.

e. Clone the Flare VM repository: **git clone https://github.com/mandiant/flare-vm.git**

    a. Open PowerShell as Administrator.
    b. Unblock-File .\install.ps1
    c. Set-ExecutionPolicy Unrestricted
    d. Set-ExecutionPolicy Unrestricted

According to my experience, the installation script will reboot the virtual machine several times and the entire process takes a significant amount of time to be completed, so you should be patient.

During reversing activities we'll use other tools and few IDA plugins, which some of them demand further details to complete the installation, and we'll explain about them later.

3. Gathering information

The SHA256 hash of the sample is **73e4969db4253f9aeb2cbc7462376fb7e26cc4bb5bd23b82e2af0eaaf5ae66a8**.

This binary can be downloaded from **Malware Bazaar** (https://bazaar.abuse.ch/browse/) directly from any browser or using Malwoverview:

malwoverview.py -b 5 -B 73e4969db4253f9aeb2cbc7462376fb7e26cc4bb5bd23b82e2af0eaaf5ae66a8 -o 0

The first approach is to check this sample against **Virus Total**:

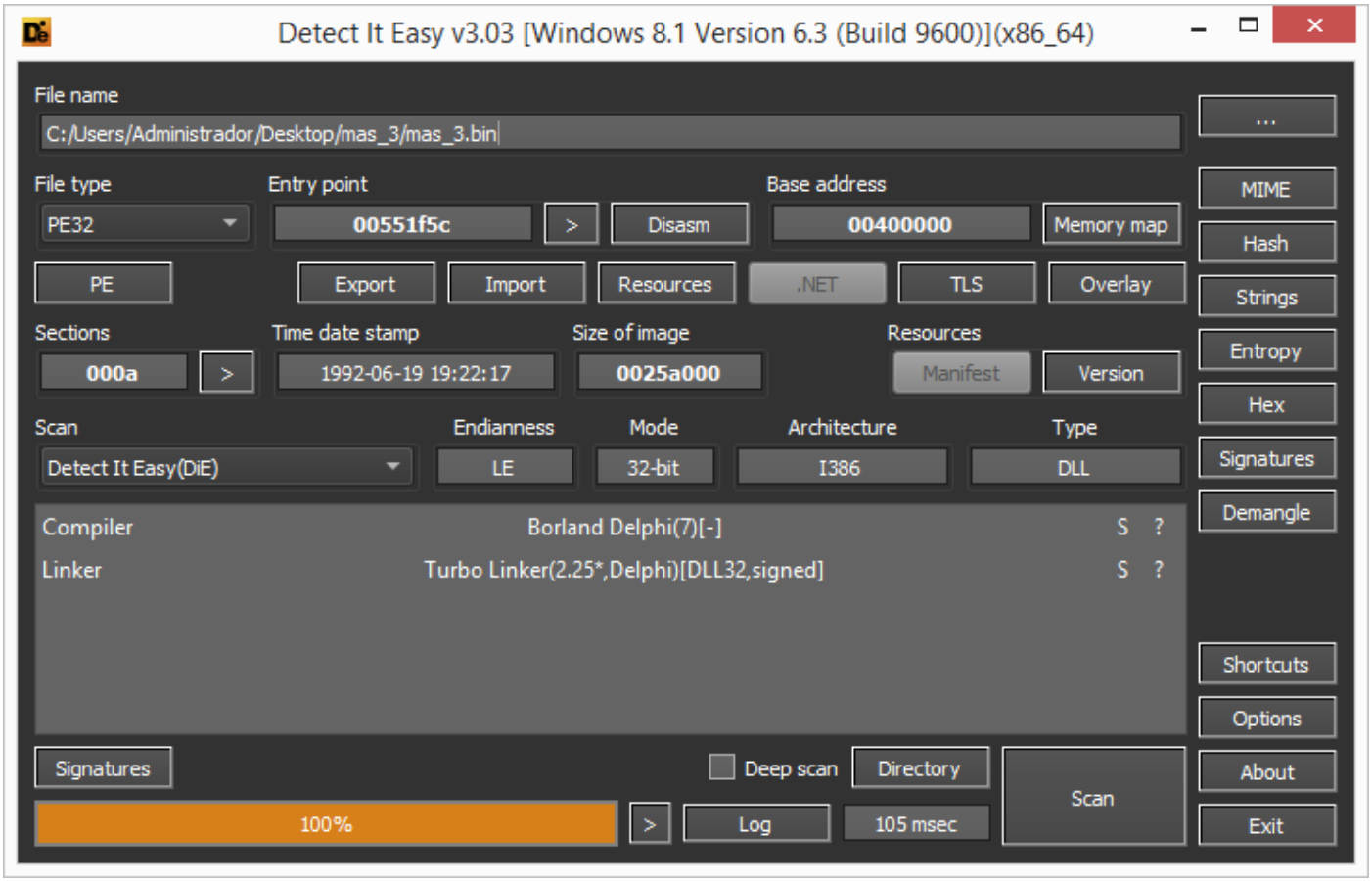
```
remnux@remnux:~/malware/mas$ malwoverview.py -f mas_2.bin -v 2 -o 0

File Name:    mas_2.bin
File Type:    PE32 executable (DLL) (GUI) Intel 80386, for MS Windows

MD5:          74cf47683051f44e6fb55ac9360c717e
SHA256:       73e4969db4253f9aeb2cbc7462376fb7e26cc4bb5bd23b82e2af0eaaf5ae66a8
Imphash:      9ac0f7498466cf5e05cd13d229c7d8c6

entropy:      6.54
Packed?:      PACKED
Overlay?:     OVERLAY
VirusTotal:   38/67


Sections:               Entropy

         CODE            6.54
         DATA            2.97
         BSS             0.00
         .idata          4.98
         .tls            0.00
         .rdata          0.21
         .reloc          6.68
         .rsrc           5.28
         DATAA           7.95
         .fiann          0.00

Main Antivirus Reports:
----------------------------------------
Scan date:    2022-01-01 00:23:55


Avast:        Win32:DangerousSig [Trj]
Avira:        TR/AD.KBot.obmwf
BitDefender:  Trojan.GenericKD.38336966
ESET-NOD32:   Win32/Qbot.DI
F-Secure:     None
FireEye:      Trojan.GenericKD.38336966
Fortinet:     W32/PossibleThreat
Kaspersky:    not-a-virus:HEUR:AdWare.Win32.DExt.gen
McAfee:       Artemis!74CF47683051
Microsoft:    Trojan:Win32/Qakbot.AY!MTB
Panda:        Trj/CI.A
Sophos:       Mal/BadCert-Gen
Symantec:     Trojan Horse
TrendMicro:   TrojanSpy.Win32.QAKBOT.TIAOABFK
```

[Figure 1]

First impression about this output:

- Likely malware is **packed**.
- The malware is really the **Qakbot / Qbot**.
- The binary has an **overlay** (you can think that overlay is a kind of attachment to the binary and doesn't make part of the its sections).
- There are **strange named sections**.

Using **DiE** (Detect It Easy: https://github.com/horsicq/Detect-It-Easy), you learn this sample was compiled with Borland Delphi.



[Figure 2]

Therefore, it would be a reasonable bet to believe this sample was packed using Borland and it is not a native malware written in Delphi. You should pay attention that there're real malware written in Delphi (mainly banker trojans), but hopefully it isn't the case.

Our next step is searching for further information on a public sandbox and my daily choice has been **Triage** (https://tria.ge/dashboard), which is also implemented on **Malwoverview**. The real reason for I usually try to collect IOCs and URL used by the malware (probably C2 communication) is to have enough information on hands that could be useful during unpacking phase, binary reversing and C2 data configuration extraction. Other interesting information got from **Triage** are evasion techniques, persistence techniques, dropped files, Registry entries, processes and all kind of static and dynamic information. For sure you can't blindly believe in all information provided by sandbox reports, but Triage has been very accurate on its analysis.

To use **Triage** on **Malwoverview** you need to get the task IDs associated to the any previously analysis associated to this malware sample and afterwards getting static and dynamic information:

```
remnux@remnux:~/malware/mas$ malwoverview.py -x 1 -X 73e4969db4253f9aeb2cbc7462376fb7e26cc4bb5bd23b82e2af0eaaf5ae66a8 -o 0

                                    TRIAGE OVERVIEW REPORT
-------------------------------------------------------------------------------------------------

id:         211225-nd9q7saac7
status:     reported
kind:       file
filename:   74cf47683051f44e6fb55ac9360c717e.dll.vir
submitted:  2021-12-25T11:18:03Z
completed:  2021-12-25T11:20:13Z
-------------------------------------------------------------------------------------------
id:         211224-p4mj2aebe2
status:     reported
kind:       file
filename:   73e4969db4253f9aeb2cbc7462376fb7e26cc4bb5bd23b82e2af0eaaf5ae66a8
submitted:  2021-12-24T12:53:04Z
completed:  2021-12-24T12:55:43Z
-------------------------------------------------------------------------------------------
id:         211224-jv1gmsddej
status:     reported
kind:       file
filename:   73e4969db4253f9aeb2cbc7462376fb7e26cc4bb5bd23b82e2af0eaaf5ae66a8
submitted:  2021-12-24T08:00:07Z
completed:  2021-12-24T08:02:45Z
-------------------------------------------------------------------------------------------
next:       2021-12-24T08:00:07.794119Z
```

[Figure 3]

```
remnux@remnux:~/malware/mas$ malwoverview.py -x 2 -X 211225-nd9q7saac7 -o 0

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

score:      10
extracted:
            botnet:    obama150
            c2:
                       96.21.251.127:2222
                       70.51.134.181:2222
                       69.14.172.24:443
                       186.64.87.213:443
                       94.62.161.77:995
                       103.139.242.30:990
                       114.79.148.170:443
                       217.164.247.241:2222
                       178.153.86.181:443
                       136.232.34.70:443
                       37.210.226.125:61202
                       173.21.10.71:2222
                       31.219.154.176:32101
                       140.82.49.12:443
                       32.221.229.7:443
                       24.152.219.253:995
                       106.51.48.170:50001
                       114.38.161.124:995
                       96.37.113.36:993
                       190.39.205.165:443
                       45.9.20.200:2211
                       105.198.236.99:995
                       70.163.1.219:443
                       103.139.242.30:995
```

```
id:         211225-nd9q7saac7
target:     74cf47683051f44e6fb55ac9360c717e.dll.vir
size:       2429328
md5:        74cf47683051f44e6fb55ac9360c717e
sha1:       93b1ab0a9e70a546c4b89dcb20a158dfc90b1421
sha256:     73e4969db4253f9aeb2cbc7462376fb7e26cc4bb5bd23b82e2af0eaaf5ae66a8
completed:  2021-12-25T11:20:13Z
signatures:
            Qakbot/Qbot
            Windows security bypass
            Loads dropped DLL
            Creates scheduled task(s)
            Modifies data under HKEY_USERS
            Suspicious behavior: EnumeratesProcesses
            Suspicious behavior: MapViewOfSection
            Suspicious use of WriteProcessMemory

targets:
            family:    qakbot
            iocs:
                       time.windows.com
                       8.8.8.8
                       40.119.148.38
            md5:       74cf47683051f44e6fb55ac9360c717e
            score:     10
            sha1:      93b1ab0a9e70a546c4b89dcb20a158dfc90b1421
            sha256:    73e4969db4253f9aeb2cbc7462376fb7e26cc4bb5bd23b82e2af0eaaf5ae66a8
            size:      2429328bytes
            tags:
                       family:qakbot
                       botnet:obama150
                       campaign:1640256791
                       banker
                       evasion
                       stealer
                       trojan
```

[Figure 4]

```
remnux@remnux:~/malware/mas$ malwoverview.py -x 7 -X 211225-nd9q7saac7 -o 0 | grep cmd
            cmd:       regsvr32 /s C:\Users\Admin\AppData\Local\Temp\74cf47683051f44e6fb55ac9360c717e.dll.vir.dll
            cmd:        /s C:\Users\Admin\AppData\Local\Temp\74cf47683051f44e6fb55ac9360c717e.dll.vir.dll
            cmd:       C:\Windows\SysWOW64\explorer.exe
            cmd:       "C:\Windows\system32\schtasks.exe" /Create /RU "NT AUTHORITY\SYSTEM" /tn nbinlxneus /tr
            cmd:       taskeng.exe {31205A28-6EAB-42D1-AE2C-1E176B460B70} S-1-5-18:NT AUTHORITY\System:Service:
            cmd:       regsvr32.exe -s
            cmd:        -s "C:\Users\Admin\AppData\Local\Temp\74cf47683051f44e6fb55ac9360c717e.dll.vir.dll"
            cmd:       C:\Windows\SysWOW64\explorer.exe
            cmd:       C:\Windows\system32\reg.exe ADD "HKLM\SOFTWARE\Microsoft\Windows
            cmd:       C:\Windows\system32\reg.exe ADD "HKLM\SOFTWARE\Microsoft\Windows
```

[Figure 5]

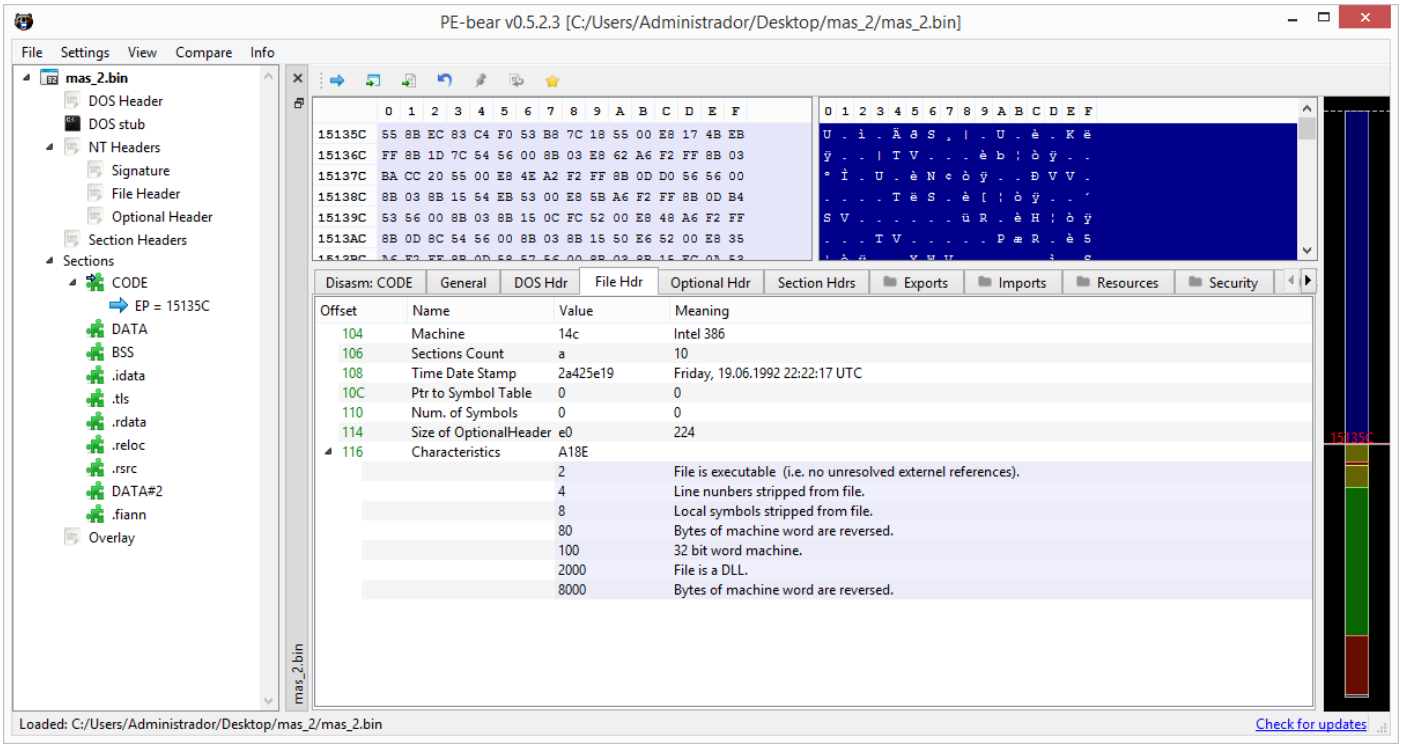
Based on **Figures 3, 4 and 5**, we have learned:

- The botnet ID is "**obama150**".
- The campaign ID is "**1640256791**".
- The binary drops an DLL in the file system in **C:\Users\<user>\AppData\Local\Temp folder**.
- Tasks are scheduled, so it likely uses them as a **persistence** mechanism.
- The sample probably injects some code into a chosen process because APIs such as **EnumerateProcess, MapViewOfSection** and **WriteProcessMemory** and, apparently, **explorer.exe** is the target process.
- The malware has an extensive list of C2 IP addresses (I truncated the output from **Figure 4** and the C2 list is bigger).

4. Unpacking

To unpack this sample we are going to use **x32dbg** (from x64dbg suite) and **HxD editor**. About the excellent **x64dbg/x32dbg** (*written by Duncan Ogilvie*), if you want to recognize and support him, so make a donation to his **x64dbg project** (https://github.com/sponsors/mrexodia). There're also other quite relevant projects that might be supported in the security industry, whose tools have helped a lot during investigations and mainly reverse engineering's sessions.

Before starting the **x64dbg session**, try to open the sample on **PEBear**:



[Figure 6]

As you can read from picture above:

- the binary is a DLL.
- it's a 32-bit PE file.
- there're several non-default sections.
- there's an overlay.

If you check **Exports tab** (not shown), you will learn this **executable has only one exported function** (**DLLRegisterServer**) and its original name seems to be **stager_1.dll**.

Holding this information we know we need to configure the **x64dbg** to debug this DLL. If you remember from first article of this series, we can **set up a DLL debugging session** by:

- Putting up **rundll32.exe** (from *C:\Windows\SysWOW64*) under the debugger's session.
- Going to **File → Change Command Line**
- Changing the text box and passing the malware and its exported function (or its number) as argument.

- The final argument in the text box should be something similar to: ***"C:\Windows\SysWOW64\rundll32.exe" C:\Users\Administrator\Desktop\mas_2\mas_2.bin,#1***
- Please, remember that "**mas_2.bin**" is the name chosen for the article and **#1** is the **ordinal number of the exported function.**
- **Restart the debugger** and play it **(F9)** once, and the debugger will hit the Entry Point's breakpoint.
- **Note:** if you had picked up rundll32.exe from C:\Windows\system32 and restarted the debugger, so it would have changed automatically to C:\Windows\SysWOW64\rundll32.exe anyway.

From this point, the debugging session is configured correctly and we can set up (**CTRL+G**) few breakpoints as shown below:

- **VirtualAlloc**
- **ZwUnmapViewOfSection**
- **WriteProcessMemory**
- **NtResumeThread**

There're some notes about these breakpoints on x64dbg:

- Go to **Options → Preferences** and make sure that both **System Breakpoint and Entry Breakpoint** are checked.

- Set breakpoints on **VirtualAlloc, WriteProcessMemory** and **ResumeThread** APIs when the x64dbg reaches the entry-point (after the system breakpoint).

- I chosen **VirtualAlloc( )** because it's usually used by malware threads to allocate and unpack binaries into the memory (mostly self-injections) and **WriteProcessMemory** because likely according to the Triage's report this binary are performing code injection into another process. Although it isn't important for malware analysis, you should remember that **VirtualAlloc( )** is normally used for big allocations, which it's very appropriate to host an unpacked malicious binary, for example.

- Remember that **VirtualAlloc** breakpoint must be set up on the return point of the API (**ret 10**) and not on its entry point.

- The breakpoint on **ZwUnmapViewOfSection** should be set up only after the debugger execution has hit **WriteProcessMemory's breakpoint.**

- Probably **NtResumeThread's breakpoint** will be hit twice, so continue the execution after the first hit because there will be educational reasons to do it.

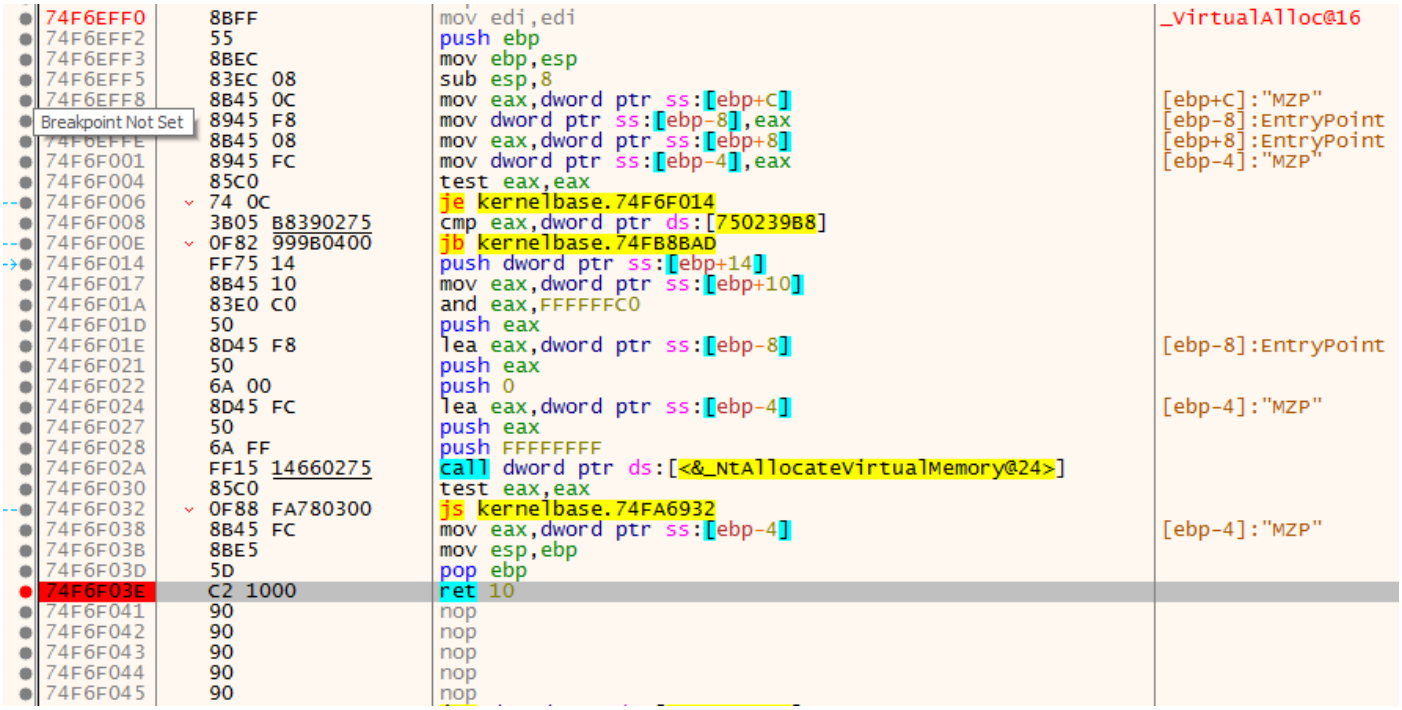
- I've set up a breakpoint on **NtResumeThread** to keep the control of the infection and avoid losing the control of the malware execution.

If you want, it's recommended to install **DbgChild plugin** into **x64dbg/x32dbg** to handle new processes created by the malware. In this article we won't use it (although new processes are launched by the

binary), but it will be useful in the future. **DbgChild** is available from: https://github.com/David-Reguera-Garcia-Dreg/DbgChild

The reason about I'm setting a breakpoint on **ZwUnmapViewOfSection** is because it'd interesting to dump a mapped version of the malware and review the entire process of fixing it manually and automatically. In addition, Triage's report told us about a possible memory mapping using **MapViewOfSection( )**.

As in the first article I received messages asking about "the exact point" of setting a **breakpoint on VirtualAlloc( )** in the first article, so the figure below shows it marked in red and light gray:



[Figure 7]

Now it's time to run (**Play or F9**) the malware under the debugger. The first three hits on **VirtualAlloc breakpoints** won't be useful. Keeping running after the third one likely will rise an exception, but there isn't any problem. **Pass exception to debugger (SHIFT+9)** and, after a while, the debugger will hit another VirtualAlloc's breakpoint. Once this breakpoint is hit, right click EAX register and choose "**Follow in Dump**".

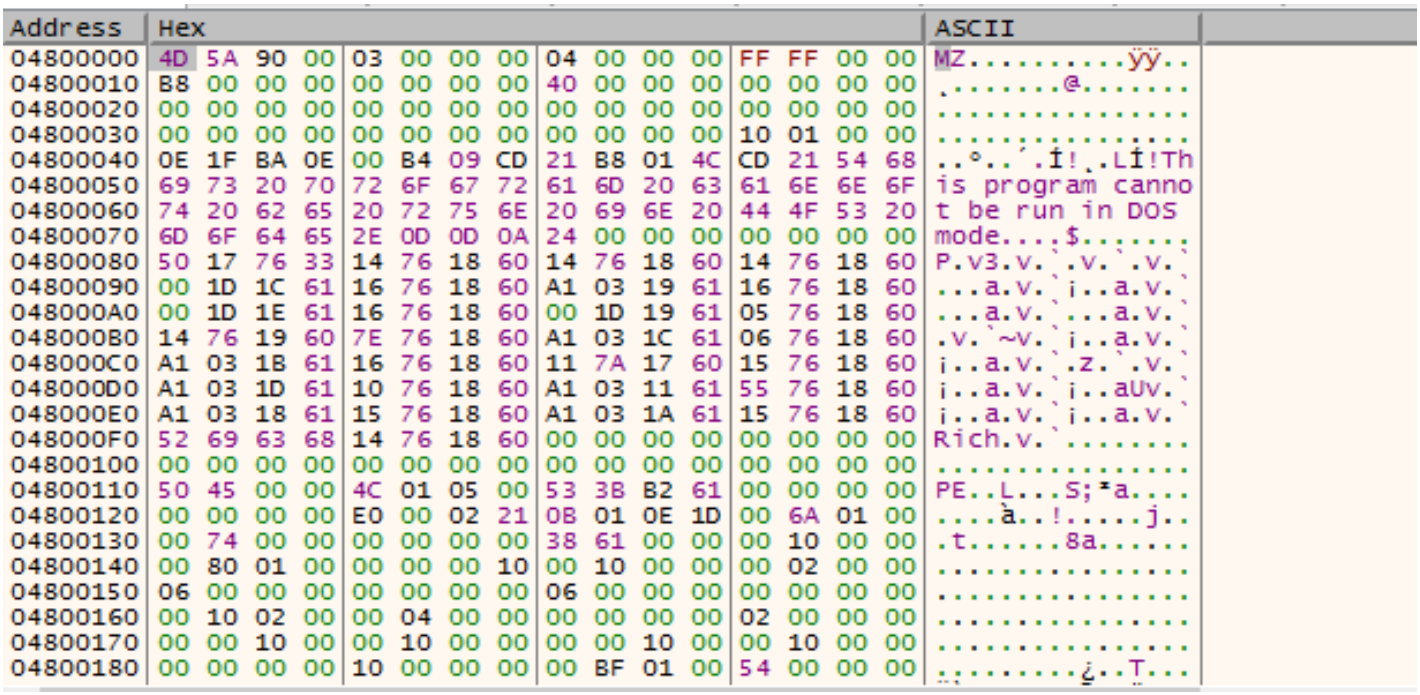
Proceed with execution and a new VirtualAlloc's breakpoint is hit. Once again, right-click in EAX register and choose **Follow in Dump → Dump 2** (to prevent the first dump of being overwritten). Continue the execution and likely you will the an PE file being written in Dump 2.

Nonetheless, it is NOT our executable yet! Why? Because if you examine the Dump2's content, you will find this PE binary has the following sections: **.text, .data, .mrdata, .rsrc and .reloc.** The third section is not a default section, so we could bet this debugging session will show something better later. Therefore, right-click on EAX register and choose **Follow in Dump → Dump 3**.

Continue the execution **(F9).** The debugger should stopped on another VirtualAlloc's breakpoint, but this time there isn't anything really useful. Right-click on EAX register and choose **Follow in Dump → Dump 4**.

Resume the execution **(F9)** and you will find a new PE executable in the Dump 4 tab. However, examining the content of this new PE format file, you'll discover its header is not clean enough, so it wouldn't so

useful for us. Right click the EAX register and pick up Follow in **Dump → Dump 5**. At same way, continue the execution **(F9)** and debugger will hit the **NtResumeThread's breakpoint**. However, a PE format executable will appear in Dump 5 area and, this time, it's the correct one as shown below:

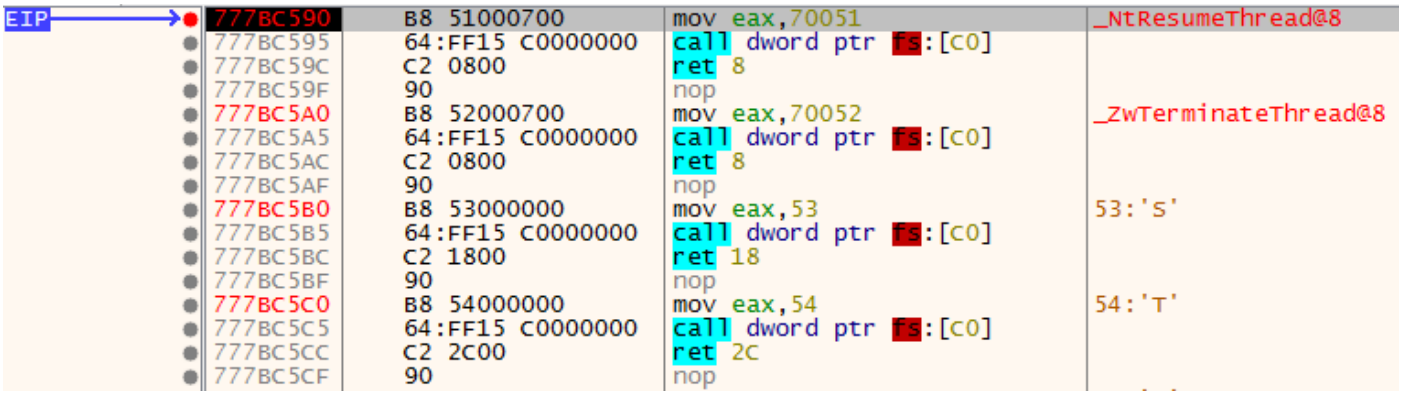


Address	Hex	ASCII
04800000	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00	MZ.....ÿÿ..
04800010	B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00	........@.......
04800020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
04800030	00 00 00 00 00 00 00 00 00 00 00 00 10 01 00 00	....
04800040	0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68	..°..´.Í!¸.LÍ!Th
04800050	69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F	is program canno
04800060	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	t be run in DOS
04800070	6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00	mode....$.......
04800080	50 17 76 33 14 76 18 60 14 76 18 60 14 76 18 60	P.v3.v.`.v.`.v.`
04800090	00 1D 1C 61 16 76 18 60 A1 03 19 61 16 76 18 60	...a.v.`¡..a.v.`
048000A0	00 1D 1E 61 16 76 18 60 00 1D 19 61 05 76 18 60	...a.v.`...a.v.`
048000B0	14 76 19 60 7E 76 18 60 A1 03 1C 61 06 76 18 60	.v.`~v.`¡..a.v.`
048000C0	A1 03 1B 61 16 76 18 60 11 7A 17 60 15 76 18 60	¡..a.v.`.z.`.v.`
048000D0	A1 03 1D 61 10 76 18 60 A1 03 11 61 55 76 18 60	¡..a.v.`¡..aUv.`
048000E0	A1 03 18 61 15 76 18 60 A1 03 1A 61 15 76 18 60	¡..a.v.`¡..a.v.`
048000F0	52 69 63 68 14 76 18 60 00 00 00 00 00 00 00 00	Rich.v.`........
04800100	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....
04800110	50 45 00 00 4C 01 05 00 53 3B B2 61 00 00 00 00	PE..L...S;²a....
04800120	00 00 00 00 E0 00 02 21 0B 01 0E 1D 00 6A 01 00	....à..!.....j..
04800130	00 74 00 00 00 00 00 00 38 61 00 00 00 10 00 00	.t......8a......
04800140	00 80 01 00 00 00 00 10 00 10 00 00 00 02 00 00	....
04800150	06 00 00 00 00 00 00 00 06 00 00 00 00 00 00 00	....
04800160	00 10 02 00 00 04 00 00 00 00 00 00 02 00 00 00	....
04800170	00 00 10 00 00 10 00 00 00 00 10 00 00 10 00 00	....
04800180	00 00 00 00 10 00 00 00 00 BF 01 00 54 00 00 00	..........¿..T...

[Figure 8]

Extract this file from memory by **right clicking the Dump 5 area** and picking **Follow in Memory Map**. From there, **right click on the gray-highlighted region** and **choose "Dump Memory To File"**. Save it wherever you want, but **keep the suggested name** because it contains the base address of the region, which might be useful during the procedure of fixing up the Import Table.

We could have stopped at this point, but let's proceed our debugger session because I'd like to show few concepts that could help you in future articles. Return to **CPU tab** and debugger will be still stopped on **NtResumeThread's breakpoint** as shown below:



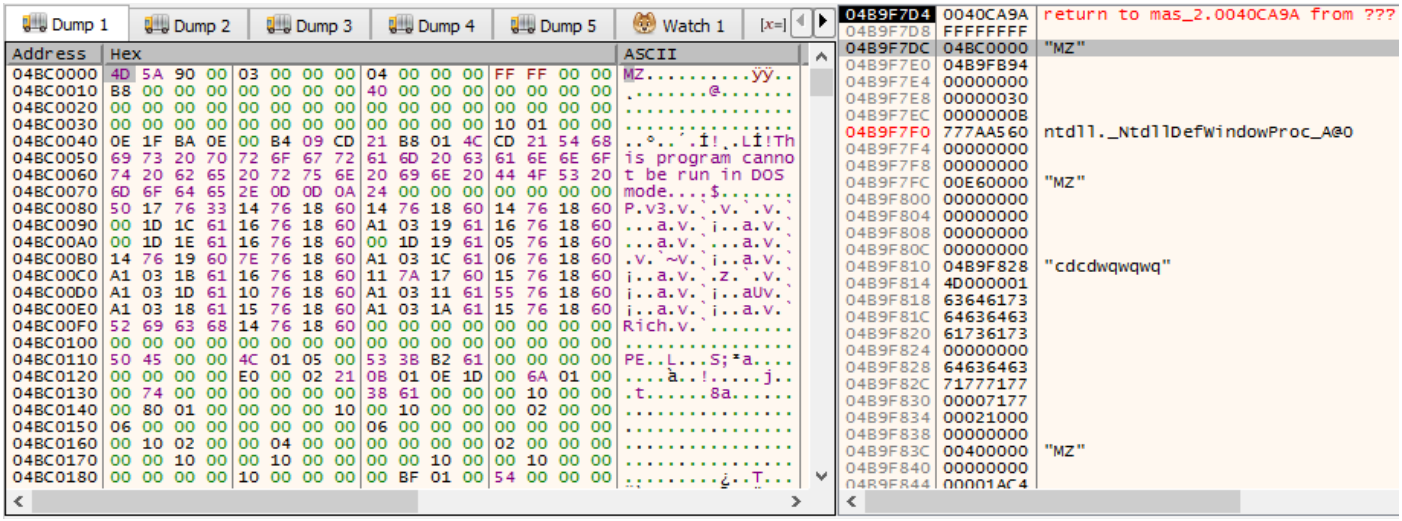
EIP →	777BC590	B8 51000700	mov eax,70051	_NtResumeThread@8
	777BC595	64:FF15 C0000000	call dword ptr fs:[C0]	
	777BC59C	C2 0800	ret 8	
	777BC59F	90	nop	
	777BC5A0	B8 52000700	mov eax,70052	_ZwTerminateThread@8
	777BC5A5	64:FF15 C0000000	call dword ptr fs:[C0]	
	777BC5AC	C2 0800	ret 8	
	777BC5AF	90	nop	
	777BC5B0	B8 53000000	mov eax,53	53:'S'
	777BC5B5	64:FF15 C0000000	call dword ptr fs:[C0]	
	777BC5BC	C2 1800	ret 18	
	777BC5BF	90	nop	
	777BC5C0	B8 54000000	mov eax,54	54:'T'
	777BC5C5	64:FF15 C0000000	call dword ptr fs:[C0]	
	777BC5CC	C2 2C00	ret 2C	
	777BC5CF	90	nop	

[Figure 9]

Continue execution (F9) until the debugger hit **the WriteProcessMemory's breakpoint**. Once it hit it, so setup a breakpoint on the **ZwUnmapViewOfSection( )** and continue the debugger execution **(F9).** Resume the execution and debugger will hit the **ZwUnmapViewOfSection's breakpoint**.

It's interesting to check the second argument (third line) in the Stack area. Why? If you check **ZwUnmapViewOfSection( )**'s information on the MSDN (from Microsoft) you will learn that the second's argument is the base address of the view being unmapped, so this region will contain our file of interest.

Thus, **right click on the second argument's address** (**third line in the Stack area because the first one is the return address**) and choose "**Follow DWORD in Dump → Dump 1**". If everything go correctly, you'll see a image similar to the following one:



[Figure 10]

Once again, repeat the same steps to save the content into a file: **right click on the Dump 1 area** and pick **Follow in Memory Map**. From there, **right click on the gray-highlighted region** and **choose "Dump Memory To File"**. Save it wherever you want, but **keep the suggested name** because it contains the base address of the region that could be useful as I mentioned previously.

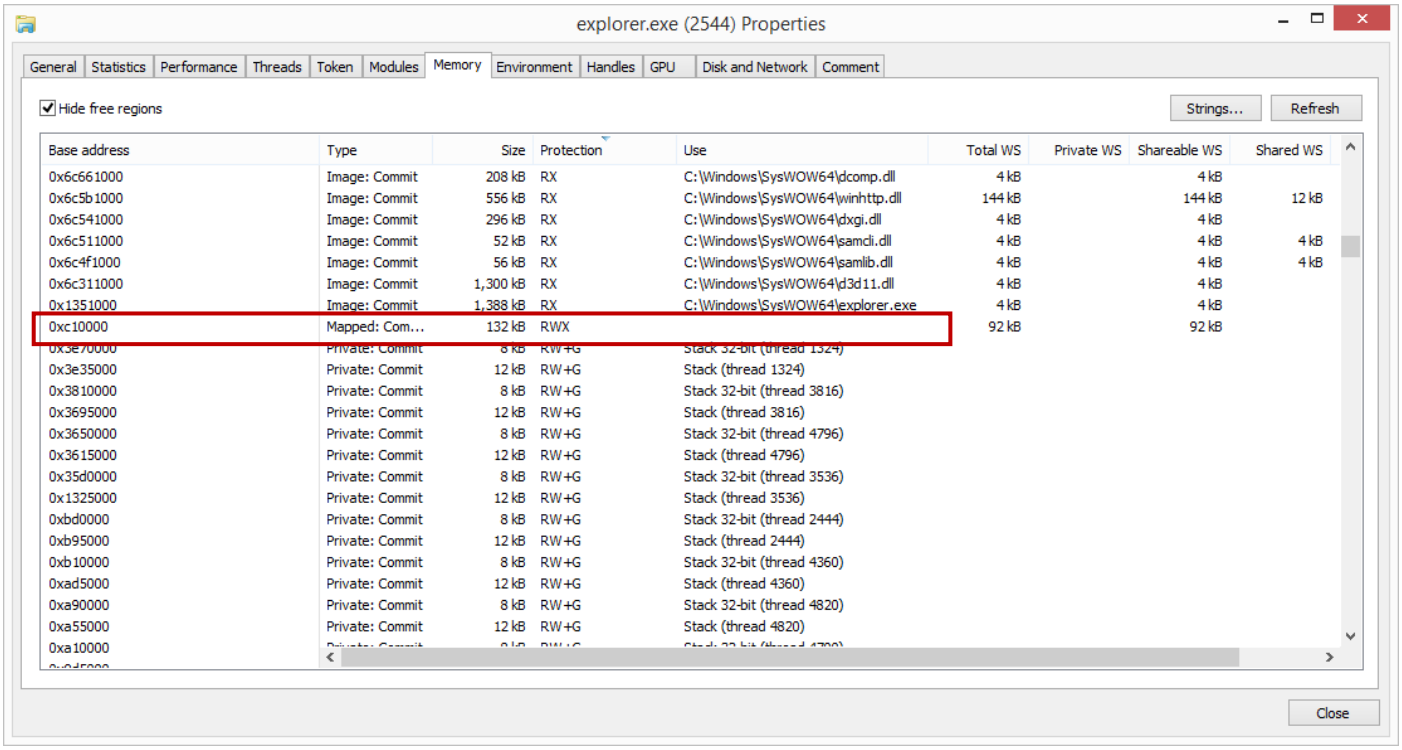
We already have two dumped files: one from the **NtResumeThread's breakpoint** (resulting from VirtualAlloc's breakpoint) and the second one from **ZwUnmapViewOfSection's breakpoint**. These files are enough, but we can continuing the debugger's execution to learn effects caused by the malware. Thus, return to **CPU tab** and resume the execution **(F9)**.

Probably the debugger will hit the **NtResumeThread's breakpoint**, but if you look for created processes in the **Process Hacker** (https://processhacker.sourceforge.io/) or **Process Explorer** from Sysinternals Suite (https://docs.microsoft.com/en-us/sysinternals/downloads/sysinternals-suite) you'll find an new **explorer.exe process** (in suspended mode) being forked from rundll32.exe. Additionally, this explorer.exe is a **32-bit executable** (*C:\Windows\SysWOW64\explorer.exe*).

Continuing the execution of the malware under the debugger control, a new instance of the x64dbg should be started and attached to the **new explorer.exe process** if you've installed the **DbgChild plugin into x32dbg**. Furthermore, checking the **Process Hacker** or **Process Explorer**, you'll **find two new explorer.exe**: the first one that was suspended and second one that forks a **conhost.exe** process. If you proceed with the debugger's execution (**SHIFT+F9** because it raised an exception), the first explorer.exe fill be terminated and only the second one will be kept.

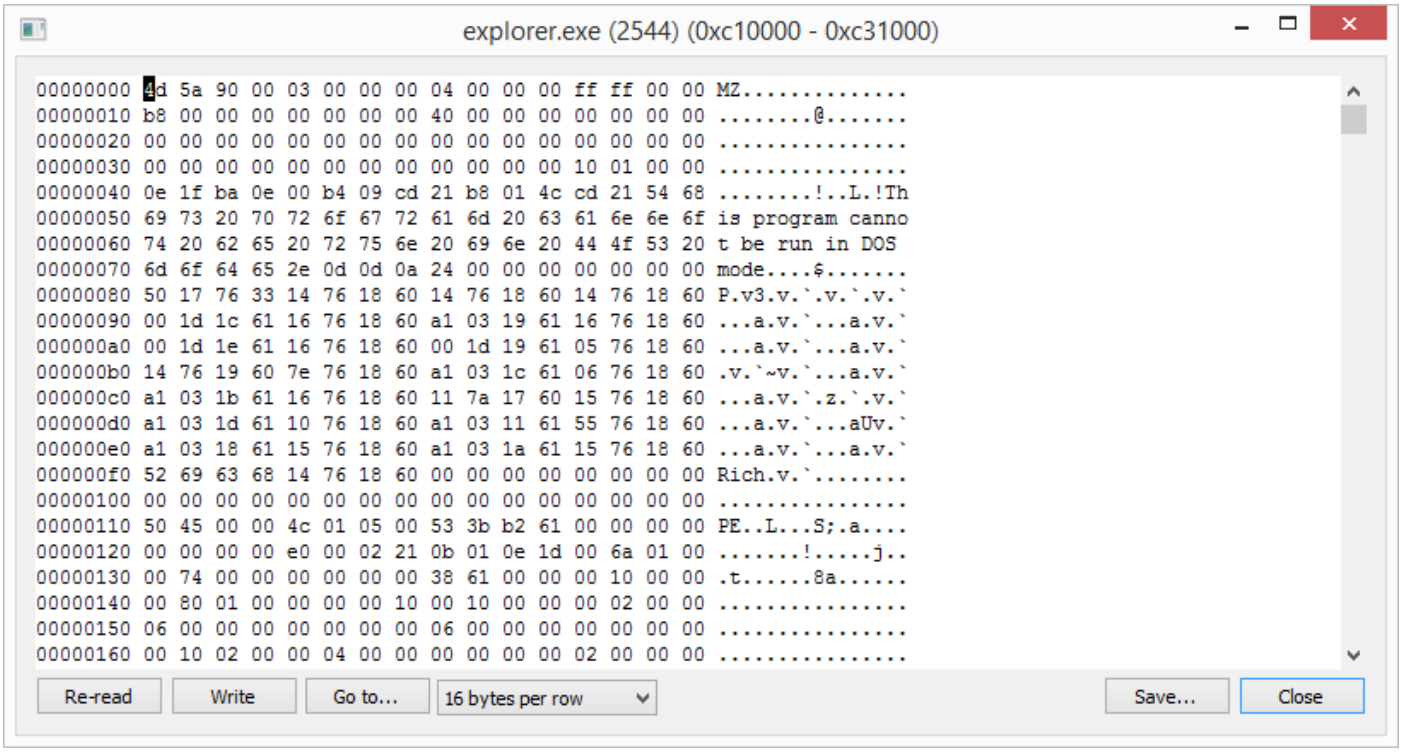
On **Process Hacker double-click the 32-bit explorer.exe** and go to **Memory tab**.

From there, **sort regions according to the Protection column**, look for a **RWX base address** then you'll see the following image:



[Figure 11]

It's interesting to realize that **it's a mapped region** and **there isn't any counterpart on the file system**. **Double-click** it and you'll have the following screen:



[Figure 12]

We have the **injected code by the malware into the new explorer.exe process**! **Save it too**.

In my environment, the three extracted files, **which are the same payload**, were:

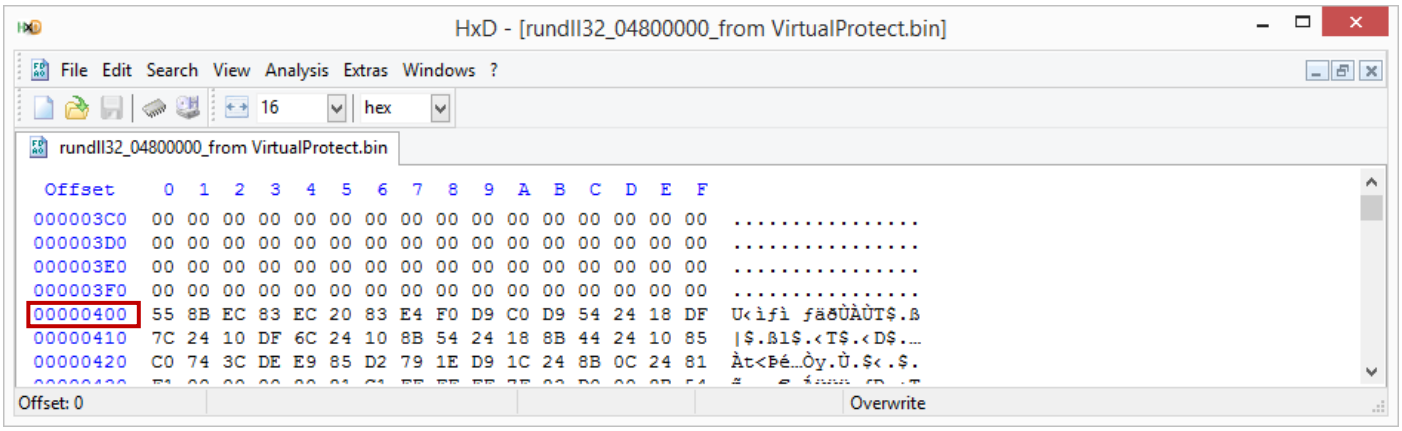
- rundll32_04800000_from VirtualProtect.bin
- rundll32_04BC0000_from_UnmapViewOfSection.bin
- explorer.exe_0xc10000-0x21000.bin

The **first file is an unmapped version of the Qakbot** and two other files are **mapped version of the Qakbot**.

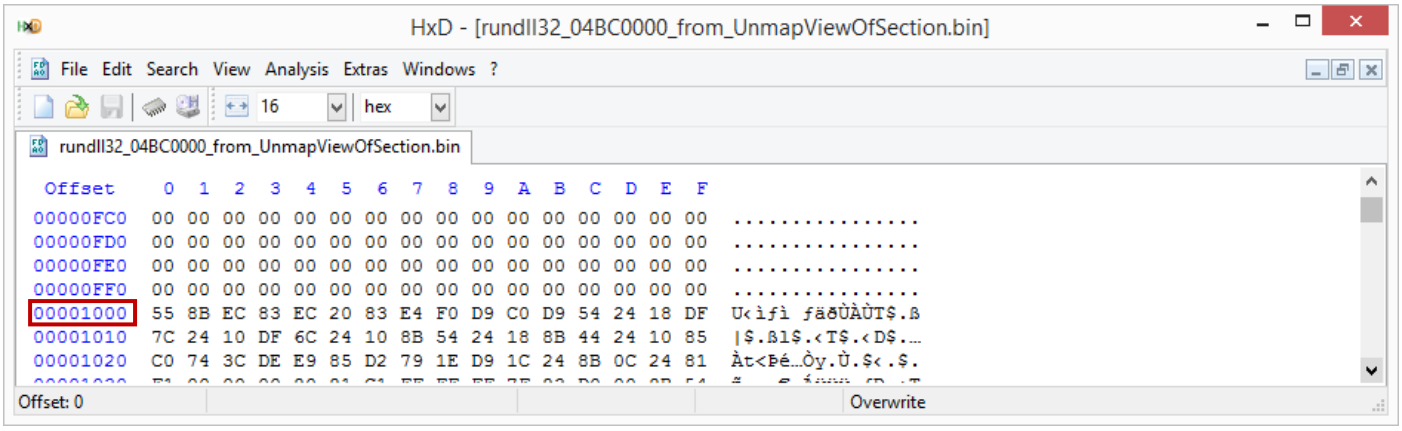
How we can check this information? Try open first two of them in a hexadecimal editor and you'll see that:

- in the **rundll32_04BC0000_from_UnmapViewOfSection.bin the .text section starts at address 0x400**, so it's in **unmapped format**. It'd possible to run it from file system.

- in the **rundll32_04BC0000_from_UnmapViewOfSection.bin** starts at address **0x1000**, so it's in **mapped format**.

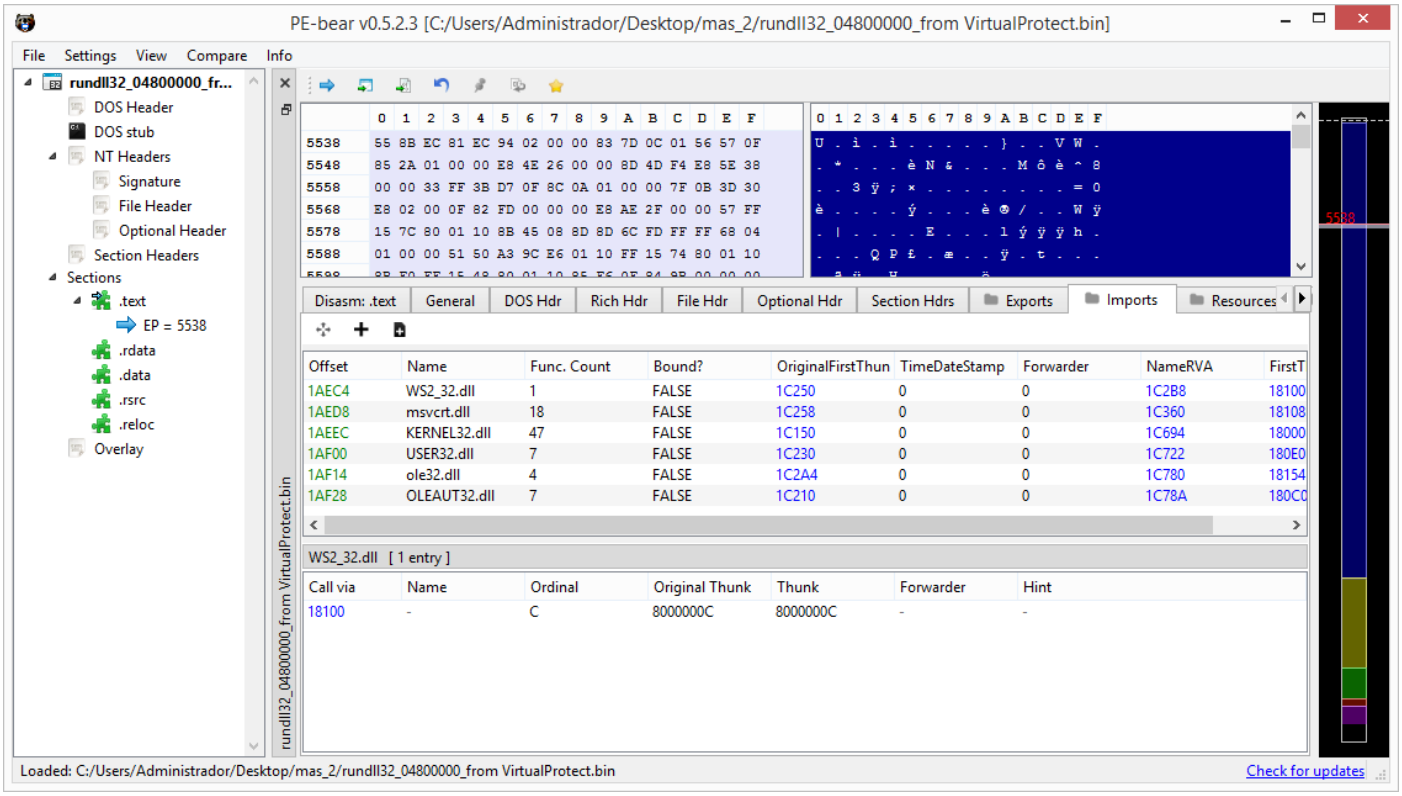


[Figure 13]

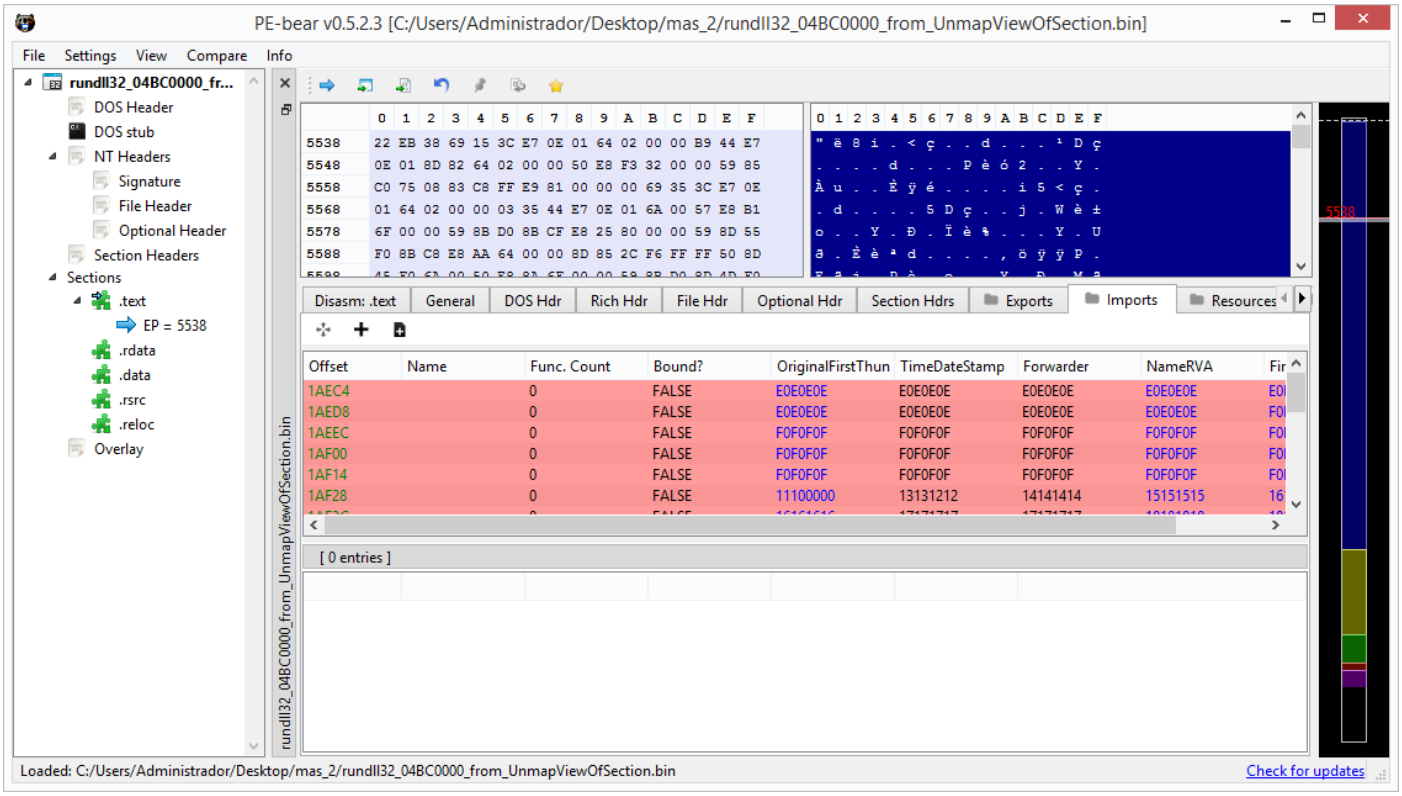


[Figure 14]

Now we can open them up on **PEBear** to check **PE format** details and, mainly, the **imported functions**:



[Figure 15]



[Figure 16]

As you're able to see, the first file (unmapped format) is perfect and we can list all Imported Functions, but in the second one (mapped format) the Import Table is messed up because sections' raw addresses are still reflecting addresses from the disk (unmapped format) rather than memory (mapped format).

You could fix them by executing the following steps:

- **Copy all Virtual Addresses over the Raw Addresses**.
- Calculate the difference between sections' start addresses and use this result as Raw Size and Virtual Size.
- **Change image base address** to reflect the memory's base address. On PE Bear it can done in **Optional Hdr tab → ImageBase**. In my case, the Image Base is 0x04BC0000 (this information comes is available in the name of the extracted file).

The original sections' headers and changed sections' headers are shown below:

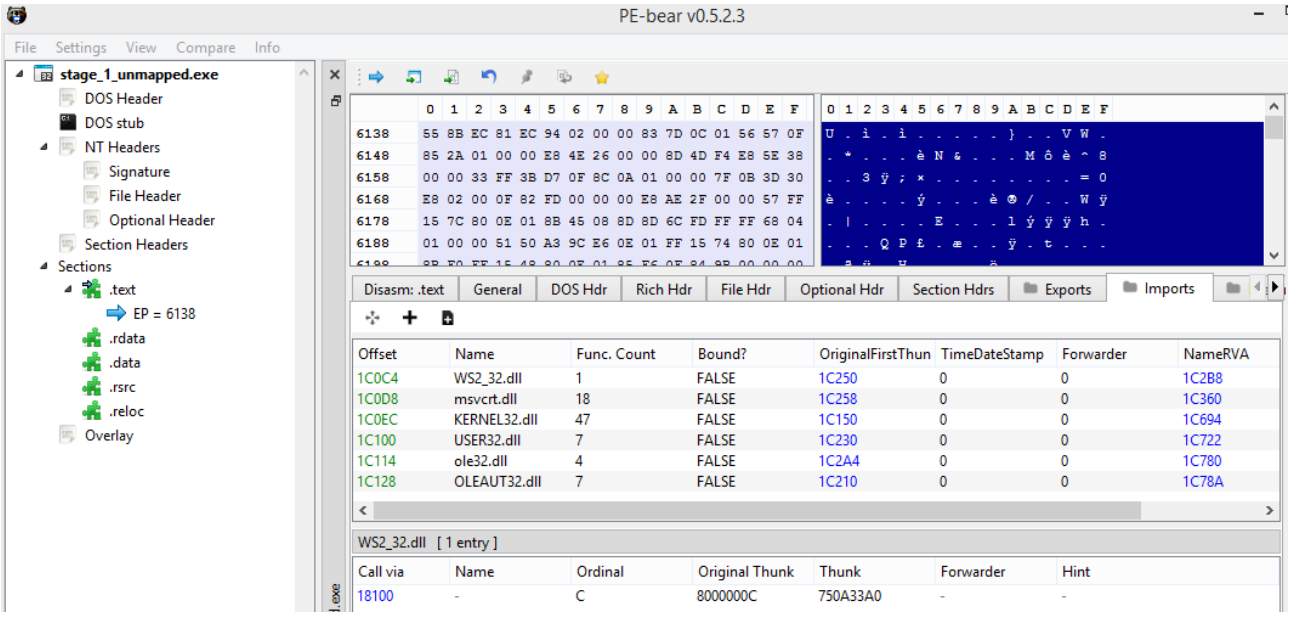
Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics	Ptr to Reloc.	Num. of Reloc.	Num. of Linenum.
.text	400	16A00	1000	16988	60000020	0	0	0
.rdata	16E00	4800	18000	47BE	40000040	0	0	0
.data	1B600	1800	1D000	1790	C0000040	0	0	0
.rsrc	1CE00	600	1F000	4FC	40000040	0	0	0
.reloc	1D400	E00	20000	C04	42000040	0	0	0

[Figure 17]

Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics	Ptr to Reloc.	Num. of Reloc.	Num. of Linenum.
.text	1000	17000	1000	17000	60000020	0	0	0
.rdata	18000	5000	18000	5000	40000040	0	0	0
.data	1D000	2000	1D000	2000	C0000040	0	0	0
.rsrc	1F000	1000	1F000	1000	40000040	0	0	0
.reloc	20000	E00	20000	E00	42000040	0	0	0

[Figure 18]

In the **PEBear**, right click on the filename, choose "**Save the executable as**" and provide a name. Open it up in PEBear and check for **Imports**:



[Figure 19]

If you prefer, it's possible to fix this file using **pe_unmapper tool** that is available on https://github.com/hasherezade/libpeconv/tree/master/pe_unmapper. Execute it as shown below:

- **pe_unmapper /in rundll32_04BC0000_from_UnmapViewOfSection.bin /base 04bc000 /out rundll32_04BC0000_from_UnmapViewOfSection_fixed.bin**

The result is the same and imported table is fixed. The third file file, which was extracted using Process Hacker, is also mapped and the approach is the same.

Once again: all three extracted files are the same payload (Qakbot), so we could use any of them. However, I usually to use the unmapped version when it's available, so let's save it out of the virtual machines, rollback the snapshot and start to analyze it.

5. Reversing

No doubts, reversing any malware might consume a significant amount of time and most of the time it present several challenges:

- Strings are usually encrypted.
- In several opportunities, strings are organized in string tables.
- APIs are hashed using logical operations or even encrypting algorithms.
- Call conventions might be wrong.
- Well-known functions are inline in the code and, sometimes, implemented in a different way.
- There're many C++ structures and, sometimes, we don't have how to guess the exact size of each field/member. Worse, there could be structures inside structures and different STLs.
- Public keys are usually "hidden" in the configuration data.
- C2 data configuration is encrypted.
- The network communication follows a specific protocol and, of course, it's obfuscated and/or encrypted.

As consequences of all these difficulties, it's necessary to have a step-by-step approach while reversing any malware and, in a summarized way, you should have clears goals while doing it. Mostly it'll be useful perform a general markup over all assembly / pseudo code (produced by the decompiler) to make the reversed code easier to understand as well as the general purpose of each subroutine. Certainly, over this markup process, we'll find several hashed functions, encrypted data and crypto-algorithms, which demands writing Python 3 scripts to extract and decrypt valuable information.

Before starting the general code revision, when we are going to rename variables, functions and their respective types, it would be recommended to configure a Python environment for writing our scripts quicky and, no doubts, several IDA Pro plugins that will help us while reversing the malware.

Personally, before releasing a standalone Python script, I use **Jupyter Notebook** to write draft Python scripts because I think the debugging process is better (I learned it while working on fuzzing scripts). In few words, you can setup your environment as shown below:

1. **pip install jupyterlab**
2. **jupyter-lab**

3. Choose **Python 3 Notebook** (right side)
4. Rename the document (left side)
5. If you accept my suggestion, try to create a private GitHub and push all your scripts onto there. The advantage of following this simple procedure is having your scripts everywhere and whenever you need it (and within any virtual machine) Additionally, you can make them public when you feel comfortable with them.
6. At end, export your scripts to standalone versions to be used in your daily work.

About IDA Pro plugins, there're an extensive list of quite useful ones that might help you during reversing malware in your daily job and, of course, each professional has her/his preferences, but eventually the presented plugins are interesting.

Personally, I'm using IDA Pro version 7.7 with decompiler and Python 3.x as default version. To check which Python version is configured with your IDA Pro, open it up and, in the IDA Python prompt, type:

- **import sys**
- **sys.version**

If you need to change the configured Python for IDA Pro, you can do it using "**idapyswitch.exe**", which is available on the IDA Pro installation folder (in my case: *C:\Program Files\IDA Pro 7.7*).

a. Flare Capa Explorer

The plugin is excellent to detect capabilities of executable files inside the IDA Pro. In special, I like it because it helps to detect and identify crypto-algorithms, persistence, evasion techniques and network communication. To install it, execute:

- **pip install wheel**
- **pip install flare-capa**
    - **Note:** at time I'm writing this article, the default PIP package is not supported for IDA 7.7, so you should install it using the following command**: pip install git+https://github.com/mandiant/capa**
- **clone the capa: git clone http://github.com/mandiant/capa.git.**
- **clone the capa-rules: git clone https://github.com/mandiant/capa-rules.git**
- copy the **capa_explorer.py** plugin to *IDA plugin directory*. In my case:
    - C:\github\capa\capa\ida\plugin> **cp capa_explorer.py "C:\Program Files\IDA Pro 7.7\plugins"**
- On IDA Pro, load the binary and, eventually, it'd be recommended to select **Manual Load** and **Load Resources** for getting better results. However, you wouldn't need to load the overlay.
- Go to **Edit → Plugin → Flare capa explorer** and select **"Program Analysis" tab**. From there, click on the "Analysis" button, which will prompt you to select the folder containing the capa-rules (in my case, **C:\github\capa-rules**).

- **Note:** from time to time, don't forget to update Capa and capa-rules using "**git pull**" command.

b. ApplyCalleType and StructTyper plugins

Both plugins are available from the excellent flare-ida project. To install them:

- **git clone** https://github.com/mandiant/flare-ida
- copy **apply_callee_type_plugin.py** and **struct_typer_plugin.py** to **"C:\Program Files\IDA Pro 7.7\plugins"** folder.
- copy the **python folder** (for example: **"C:\github\flare-ida\python"**) to Python folder from IDA directory (for example: **C:\Program Files\IDA Pro 7.7\python**)
- **Notes:**
  - remember to update flare-ida using "**git pull**".
  - After updating it you should **copy the named plugins to the mentioned directory**.
  - There're other two plugins in the directory: **stackstrings_plugin.py** and **shellcode_hashes_search_plugin.py**. The former works only with Python 2.7 (we should change the IDA's python configuration to fill this request) and the second one is a good plugin, but we'll use a recently released plugin from OALabs.

c. Findcrypt-yara

This a simple, but effective IDA Pro plugin to find crypto constant, mainly. Of course, Flare Capa Explorer is also able to detect crypto-algorithms, but it's always recommended to have two methods to do the same task. To install it:

- **pip install yara-python**
- **git clone https://github.com/polymorf/findcrypt-yara.git**
- copy both **findcrypt3.py** and **findcrypt3.rule** to IDA's plugin folder (*C:\Program Files\IDA Pro 7.7\plugins*)

d. HashDB

HashDB is an excellent plugin from OALabs that performs string hash lookup against a remote database on OALabs. Actually, it's a welcome evolution and extension from the idea offered by **shellcode_hashes_search_plugin.py plugin** (created by Mandiant), which I personally used in several opportunities, and it's able to provide an amazing integration with IDA Pro and really handle and detect most hashed strings. Install it by executing the following steps:

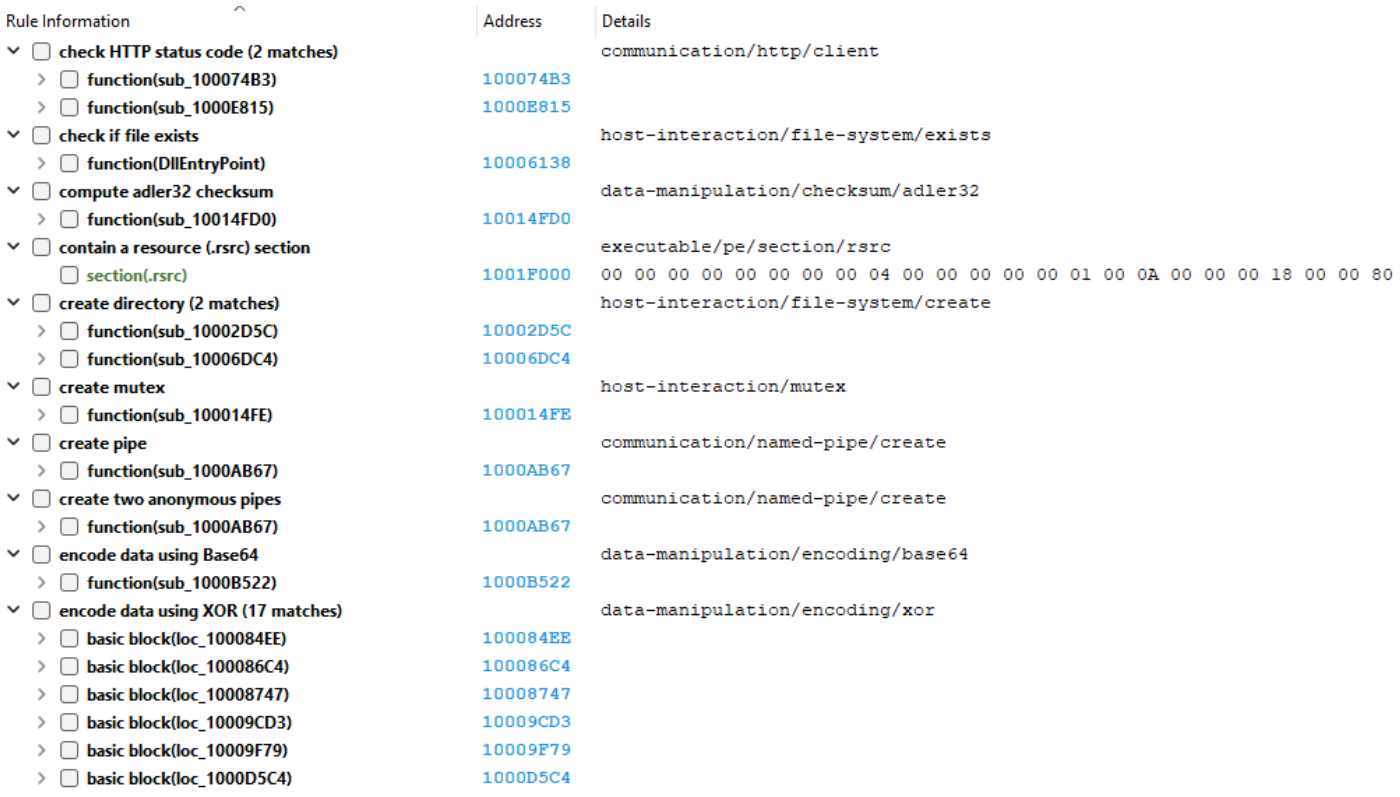
- **git clone** https://github.com/OALabs/hashdb-ida
- copy **hashdb.py** to IDA's plugin directory (*C:\Program Files\IDA Pro 7.7\plugins*)
- **Attention:** as HashDB performs lookup on OALabs server, so you should remember to keep Internet access in your environment.

Over the **Flare Capa Explorer** usage is suggested to load the resource section to its better operation. Furthermore, another good point is that non-sense data and function names (pointing to this section) are also prevented.

Few recommendations just after opening any malware on IDA Pro are:

- Decompile the entire program to avoid any decompiler's issue later: **File → Produce File → Create C File (**or **CTRL+F5).**
- Go to **View → Open Subviews → Type Libraries (or SHIFT-F11)** and confirm whether **mssdk_win7, ntapi_win7 and ntddk_win7** are included. If they aren't, so do it by using **INS key**. It might be useful to add into **Signatures** (**View → Open Subviews → Signatures or SHIFT-F5**) the **vc32rtf library**. Remember that, though all of libraries comes from Windows 7 base foundation, to analyze kernel drivers might be better to use libraries related to Windows 10.
- Run **Flare Capa Explorer plugin** (**Analyze button**) and take a short note about main indications. Note: when you run it at first time, you'll need to point the capa-rules directory.
- Check **Findcrypt's results** to collect further information.

The **Flare Capa Explorer** screenshot follows below:



Rule Information	Address	Details
check HTTP status code (2 matches)		communication/http/client
function(sub_100074B3)	100074B3	
function(sub_1000E815)	1000E815	
check if file exists		host-interaction/file-system/exists
function(DllEntryPoint)	10006138	
compute adler32 checksum		data-manipulation/checksum/adler32
function(sub_10014FD0)	10014FD0	
contain a resource (.rsrc) section		executable/pe/section/rsrc
section(.rsrc)	1001F000	00 00 00 00 00 00 00 00 04 00 00 00 00 00 01 00 0A 00 00 00 18 00 00 80
create directory (2 matches)		host-interaction/file-system/create
function(sub_10002D5C)	10002D5C	
function(sub_10006DC4)	10006DC4	
create mutex		host-interaction/mutex
function(sub_100014FE)	100014FE	
create pipe		communication/named-pipe/create
function(sub_1000AB67)	1000AB67	
create two anonymous pipes		communication/named-pipe/create
function(sub_1000AB67)	1000AB67	
encode data using Base64		data-manipulation/encoding/base64
function(sub_1000B522)	1000B522	
encode data using XOR (17 matches)		data-manipulation/encoding/xor
basic block(loc_100084EE)	100084EE	
basic block(loc_100086C4)	100086C4	
basic block(loc_10008747)	10008747	
basic block(loc_10009CD3)	10009CD3	
basic block(loc_10009F79)	10009F79	
basic block(loc_1000D5C4)	1000D5C4	

[Figure 20]

Notes about the **Flare Capa Explorer's output** (the list from figure above was truncated) that, eventually, might be useful as reference during analysis follow below:

- There's an indication of **Base64 encoding** on **sub_1000B522** subroutine.
- There's a **file enumeration** on **sub_1000B064**.

- The malware has two subroutines (**sub_100124AE** and **sub_100124F3**) using **Mersenne Twister** (a kind pseudorandom number generator – PRNG).
- Subroutine **sub_1000F681** seems to be handling with **SHA1 hashing algorithm**.
- Subroutine **sub_10012C9B** is handling with **Import Table reconstruction**.
- Subroutine **sub_10012862** seems to handle with **PE header**.
- There're indication of **HTTP manipulation** on **sub_100074B3** and **sub_1000E815** subroutines.
- Subroutine **sub_10014FD0** might be using **Adler32 checksum algorithm**, which is used of **zlib compression library** and it's faster than CRC-32 algorithm.
- There're many references to **XOR operations** and, likely, many of them are involved in some kind of **string encoding**.

It's quite relevant to highlight that all findings from **Flare Capa Explorer** must be confirmed because there're can be false positives.

Another IDA plugin that could be used to find crypto-constants is the **Findcrypt plugin** and, no doubts, it might be useful for supplementing information given by other plugins:

Rules file	Name	String	Value
global	CRC32_poly_Constant_10018DF8	$c0	b' \x83\xb8\xed'
global	CRC32_poly_Constant_1001B940	$c0	b' \x83\xb8\xed'
global	CRC32_table_10019BF8	$c0	b'\x00\x00\x00\x00w\x070\x96\xee\x0ea,\x99\tQ\xba\x07m\xc4\x19'
global	RIPEMD160_Constants_1000F6A9	$c5	b'\x01#Eg'
global	RIPEMD160_Constants_1000F6B1	$c6	b'\x89\xab\xcd\xef'
global	RIPEMD160_Constants_1000F6B9	$c7	b'\xfe\xdc\xba\x98'
global	RIPEMD160_Constants_1000F6C1	$c8	b'vT2\x10'
global	RIPEMD160_Constants_1000F6C9	$c9	b'\xf0\xe1\xd2\xc3'
global	SHA1_Constants_1000F6A9	$c5	b'\x01#Eg'
global	SHA1_Constants_1000F6B1	$c6	b'\x89\xab\xcd\xef'
global	SHA1_Constants_1000F6B9	$c7	b'\xfe\xdc\xba\x98'
global	SHA1_Constants_1000F6C1	$c8	b'vT2\x10'
global	SHA1_Constants_1000F6C9	$c9	b'\xf0\xe1\xd2\xc3'
global	SHA1_Constants_1000F5B4	$c10	b'\xd6\xc1b\xca'
global	BASE64_table_1001B880	$c0	b'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/'

[Figure 21]

The output is very interesting because:

- **CRC32** is being used in several places. One of places seems being a CRC32 table.
- **SHA1**, as provided by Capa Explorer, is also shown.
- **Base64 table** is also listed.
- **RIPEMD (RIPE Message Digest)**, a family of cryptographic hash functions, was found in several addresses

Our starting point will be the **DllEntryPoint function** and we'll navigate in several subroutines and moving forward according to the execution flow. Of course, it isn't feasible to comment about all subroutines (there're over 500 of them) in this article and we are going to focus on few of them only.

The general basic approach is **renaming variables, functions and functions' arguments, changing its respective types ('Y' hotkey) in all possible opportunities and creating structures to attend C++ manipulations**.

Thus, a recommended approach is putting **Disassembly View and Pseudocode View side by side** and synchronize both in the IDA Pro because almost all our work will be done on the decompiler.

```
1  BOOL __stdcall DllEntryPoint(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpReserved)
2  {
3    __int64 v3; // rax
4    DWORD ModuleFileNameW; // esi
5    DWORD LastError; // eax
6    int i; // esi
7    __int16 Filename[260]; // [esp+Ch] [ebp-294h] BYREF
8    WCHAR WideCharStr[64]; // [esp+214h] [ebp-8Ch] BYREF
9    char v10[8]; // [esp+294h] [ebp-Ch] BYREF
10   int v11; // [esp+29Ch] [ebp-4h] BYREF
11
12   if ( fdwReason == 1 )
13   {
14     sub_100087A0();
15     LODWORD(v3) = sub_100099B8(v10);
16     if ( v3 >= 190512 )
17     {
18       sub_10009124();
19       GetModuleHandleA(0);
20       dword_1001E69C = (int)hinstDLL;
21       ModuleFileNameW = GetModuleFileNameW(hinstDLL, (LPWSTR)Filename, 0x104u);
22       LastError = GetLastError();
23       if ( !ModuleFileNameW || LastError == 122 )
24         return 0;
25       memset(WideCharStr, 0, sizeof(WideCharStr));
26       for ( i = 0; i < 10000; ++i )
27       {
28         fdwReason = sub_10009773(i);
29         MultiByteToWideChar(0, 0, (LPCCH)fdwReason, -1, WideCharStr, 63);
30         sub_10008773(&fdwReason);
31       }
32       sub_10012C0A(dword_1001E69C);
33       dword_1001E684 = sub_1000E369(1486);
34       fdwReason = sub_1000978D(265);
35       if ( GetFileAttributesW((LPCWSTR)fdwReason) != -1 )
36       {
37         sub_10008786(&fdwReason);
38         return 0;
39       }
40       sub_10008786(&fdwReason);
41       v11 = 0;
```

[Figure 22]

To rename functions I usually follow a kind of informal naming rule by adding a prefix such as "**mw"** (from malware) or **"ab"** (from my own name), and try to reflect the main goal of the function. I also use the word **"wrap"** or **"w"** to point that a function contain a reference to another important function. Other suggestions the might be useful while reserving the code using IDA Pro:

- Try to find the **enumeration** associated to a constant my using the '**M**' hotkey.
- If there's any decimal value (mainly a hash) that you need to convert to hexadecimal, use **'H' hotkey.**
- You should always look for code and data cross-references using **'X' hotkey** to list cross-references to the cursor and **CTRL-X** hotkey to list cross-references to the current address. Cross-references from the current address (**CTRL-J**) and, on decompiler, global cross-references (**CTRL+ALT+X**) to a specific structure field are also very useful.
- In the pseudo code, from Decompiler, if a function doesn't have any argument, so **double-check it against the assembly code because, sometimes, if the malware isn't using a default call convention, the decompiler might miss arguments.**
- As I've mentioned previously, if you find any **unknown constant**, so do the simple action and search it on Google.

On line 28, the subroutine **sub_10009773( )** is a first challenge and we need to analyze to get a better understanding of it:

```
1 char *sub_10009773()
2 {
3   int v1; // [esp+0h] [ebp-8h]
4   unsigned int v2; // [esp+4h] [ebp-4h]
5
6   return sub_1000865C(0xE4Cu, (int)&unk_1001D5A8, (int)&unk_1001E3F8, v1, v2);
7 }
```

[Figure 23]

The first argument is a size, the **second argument** points to a **blob of encoded data (possibly strings)** and the **third argument points to another blob of encoded data (possibly a key because its length).** We don't have information about **v1** and **v2** yet:

```
.data:1001D5A8 unk_1001D5A8    db  8Ch              ; DATA XREF: sub_10009773+C↑
.data:1001D5A8                                      ; sub_1000978D+11↑o
.data:1001D5A9                 db 0DFh
.data:1001D5AA                 db  29h ; )
.data:1001D5AB                 db 0ACh
.data:1001D5AC                 db  4Fh ; O
.data:1001D5AD                 db 0F9h
.data:1001D5AE                 db 0F1h
.data:1001D5AF                 db  36h ; 6
```

[Figure 24]

```
.data:1001E3F8 unk_1001E3F8    db 0EFh             ; DATA XREF: sub_10009773+21↑
.data:1001E3F8                                     ; sub_1000978D+C↑o
.data:1001E3F9                 db 0B0h
.data:1001E3FA                 db  5Bh ; [
.data:1001E3FB                 db 0C9h
.data:1001E3FC                 db  1Ch
.data:1001E3FD                 db  9Ch
.data:1001E3FE                 db  83h
.data:1001E3FF                 db  40h ; @
```

[Figure 25]

We took a note of these address because they will be useful:

- blob 1 (maybe strings): **0x1001D5A8**
- blob 2 (maybe a key): **0x1001E3F8**

So, moving inside subroutine **sub_1000865C** (next page – **Figure 26**), we find a very interesting instruction on **line 25**:

- **v10[v5 - a5] = *(_BYTE *)(v5 + a2) ^ *(_BYTE *)(v5 % 0x5A + a3);**

In this expression, a2 is the blob 1 (probably a string table) and a3 is blob 2 (probably an encrypting key). There's the operation "v5 % 0x5A", which is also useful for our analysis and, strangely, "a4" argument is not being used:

```
1  char *__usercall sub_1000865C@<eax>(unsigned int a1@<edx>, int a2@<ecx>, int a3, int a4, unsigned int a5)
2  {
3    unsigned int v5; // edi
4    unsigned int v6; // esi
5    int v9; // esi
6    char *v10; // eax
7
8    v5 = a5;
9    v6 = a5;
10   if ( a5 >= a1 )
11     return (char *)&unk_1001E6FE;
12   while ( *(_BYTE *)(v6 % 0x5A + a3) != *(_BYTE *)(v6 + a2) )
13   {
14     if ( ++v6 >= a1 )
15       return (char *)&unk_1001E6FE;
16   }
17   v9 = v6 - a5;
18   if ( !v9 )
19     return (char *)&unk_1001E6FE;
20   v10 = (char *)sub_100087B5(v9 + 1);
21   if ( v10 )
22   {
23     do
24     {
25       v10[v5 - a5] = *(_BYTE *)(v5 + a2) ^ *(_BYTE *)(v5 % 0x5A + a3);
26       ++v5;
27       --v9;
28     }
29     while ( v9 );
30     return v10;
31   }
32   return (char *)&unk_1001E6FE;
33 }
```

[Figure 26]

After **renaming ('N' hotkey)** subroutines, arguments and variable, we have the following:

```
1  char *__usercall mw_decode_string_table@<eax>(
2          unsigned int size@<edx>,
3          int string_table_1@<ecx>,
4          int string_table_1_key,
5          int arg_4,
6          unsigned int string_table_offset)
7  {
8    unsigned int arg_5_ref; // edi
9    unsigned int arg_5_ref_1; // esi
10   unsigned int string_table_size; // esi
11   char *prt_mem_HeapAlloc; // eax
12
13   arg_5_ref = string_table_offset;
14   arg_5_ref_1 = string_table_offset;
15   if ( string_table_offset >= size )
16     return (char *)&unk_1001E6FE;
17   while ( *(_BYTE *)(arg_5_ref_1 % 90 + string_table_1_key) != *(_BYTE *)(arg_5_ref_1 + string_table_1) )
18   {
19     if ( ++arg_5_ref_1 >= size )
20       return (char *)&unk_1001E6FE;
21   }
22   string_table_size = arg_5_ref_1 - string_table_offset;
23   if ( !string_table_size )
24     return (char *)&unk_1001E6FE;
25   prt_mem_HeapAlloc = (char *)mw_HeapAlloc(string_table_size + 1);
26   if ( prt_mem_HeapAlloc )
27   {
28     do
29     {
30       prt_mem_HeapAlloc[arg_5_ref - string_table_offset] = *(_BYTE *)(arg_5_ref + string_table_1) ^ *(_BYTE *)(arg_5_ref % 90 + string_table_1_key);
31       ++arg_5_ref;
32       --string_table_size;
33     }
34     while ( string_table_size );
35     return prt_mem_HeapAlloc;
36   }
37   return (char *)&unk_1001E6FE;
38 }
```

[Figure 27]

To manage this scenario and other similar situations, we usually **write a Python script to decode the encrypted data by mimicking the pseudo code**. Honestly, I almost never use debuggers during reversing analysis and prefer doing everything statically, but it's a personal choice. Another approach I'm used to taking into account is extracting the data bytes (for example, **string_table_1** and **string_table_1_key** on the **Figure 27**) reading directly from the database instead of copy it into the Python script. There isn't good or bad choice here and everything is a personal option. The script to decrypt the data blob follows below:

```
import binascii
import pefile

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

# This routine populates the string table. Pay attention to separator '\x00'.
def make_string_table(string_data):
    str_table = []
    for k in string_data.split('\x00'):
        str_table.append(k)
    return str_table

# This routine only prints the string table.
def print_string_table(my_table):
    for j in range(0, len(my_table)):
            print(my_table[j])

# This routine searches for a string in the string table. Pay attention:
# the search is through a given offset in bytes and not a slot of this table.
def string_decrypter_search(arg_string, arg_key, str_addr):
    local_table = []
    for i in range(0, len(arg_string)):
        local_table.append((arg_string[i]) ^ (arg_key[i % len(arg_key)]))
    converted_table = bytes(local_table)[str_addr:].decode('latin').split('\x00')[0]
    return (str_addr, converted_table)

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

# 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 28]

```python
# data_seg_start: start address of the provided .data segment
# encrypted_string_addr: start address of the encrypted strings
# key_data_addr: start address of the key used to decrypt strings

def string_decrypter(data_seg_start, encrypted_string_addr, key_data_addr):

    data_1 = b''
    data_2 = b''

    # Next two lines calculates the offset between data blob and start of the .data segment.
    encrypted_string_addr_rel = calc_offsets(data_seg_start, encrypted_string_addr)
    key_data_addr_rel = calc_offsets(data_seg_start, key_data_addr)

    # Next three lines extracts .data section's information.
    filename = r"C:\Users\Administrador\Desktop\MAS\mas_3\stage_1.bin"
    data_encoded_extracted_1 = extract_data(filename)
    data_encoded_extracted_2 = extract_data(filename)

    # Next six lines calculates the size of the encrypted string table and the XOR key. Pay attention that
    # I've used an approach of searching up to two "end of string" marker is found.
    d1_off = 0x0
    d2_off = 0x0
    if (b'\x00\x00' in data_encoded_extracted_1[int(encrypted_string_addr_rel,16):]):
        d1_off = (data_encoded_extracted_1[int(encrypted_string_addr_rel,16):]).index(b'\x00\x00')
    if (b'\x00\x00' in data_encoded_extracted_2[int(key_data_addr_rel,16):]):
        d2_off = (data_encoded_extracted_2[int(key_data_addr_rel,16):]).index(b'\x00\x00')

    # Uncomment next 4 lines to print the hexadecimal representation of the extracted bytes.
    #print("encrypted_string_addr:")
    #print(binascii.b2a_hex(data_encoded_extracted_1[int(encrypted_string_addr_rel,16):_
    # int(encrypted_string_addr_rel,16) + d1_off]))
    #print("\nkey_data_addr:")
    #print(binascii.b2a_hex(data_encoded_extracted_2[int(key_data_addr_rel,16):int(key_data_addr_rel,16) + d2_off]))

    # Next two lines the meaningful information (encrypted string table and XOR key) are isolated.
    data_1 = data_encoded_extracted_1[int(encrypted_string_addr_rel,16):int(encrypted_string_addr_rel,16) + d1_off]
    data_2 = data_encoded_extracted_2[int(key_data_addr_rel,16):int(key_data_addr_rel,16) + d2_off]

    # Finally the string table is decrypted.
    decoded_data = decrypter(data_1, data_2)

    # The decrypted string table is printed.
    print_string_table(make_string_table(decoded_data))

    # One string is extracted from the string table. You haven't seen the associated malware
    # code yet, but this example number (1486) came from the malware code. Further information on it in next pages.
    item, result = string_decrypter_search(data_1, data_2, 1486)
    print("\nstring[%d]: %s" % (item, result))
```

You're able to find the start address of the .data section by using "CTRL+S" on IDA Pro :)

```python
def main():
    string_decrypter('0x1001D000','0x1001D5A8', '0x1001E3F8')

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

[Figure 29]

I've added comments over the code for helping you to understand the script, but it's suitable leaving few observations:

- I've used **Jupyter notes** while writing a reversing Python script because it's easier to debug the code, though I always export and adapt it (if it's necessary) to a standalone Python script.

- I've **kept support functions separated from the main function**, so that's the reason of the line number sequence has been reset.

The **result from script execution** is extensive and, usually, I wouldn't put it here (waste of pages), but maybe is important to the reader being able to check own results.

```
coreServiceShell.exe;PccNTMon.exe;NTRTScan.exe
MBAMService.exe;mbamgui.exe
%SystemRoot%\SysWOW64\explorer.exe
MsMpEng.exe
SOFTWARE\Microsoft\Microsoft Antimalware\Exclusions\Paths
WBJ_IGNORE
mpr.dll
LocalLow
bdagent.exe;vsserv.exe;vsservppl.exe
avp.exe;kavtray.exe
C:\INTERNAL\__empty
cmd.exe
SOFTWARE\Wow6432Node\Microsoft\Windows Defender\Spynet
\\.\pipe\
{%02X%02X%02X%02X-%02X%02X-%02X%02X-%02X%02X-%02X%02X%02X%02X%02X%02X}
wbj.go
.cfg
Packages
iphlpapi.dll
image/pjpeg
%SystemRoot%\SysWOW64\OneDriveSetup.exe
Win32_PnPEntity
user32.dll
%SystemRoot%\System32\msra.exe
WRSA.exe
vbs
cscript.exe
egui.exe;ekrn.exe
FALSE
.dll
tcpdump.exe;windump.exe;ethereal.exe;wireshark.exe;ettercap.exe;rtsniff.exe;packetcapture.exe;capturenet.exe
crypt32.dll
ALLUSERSPROFILE
shlwapi.dll
setupapi.dll
vkise.exe;isesrv.exe;cmdagent.exe
1234567890
avgcsrvx.exe;avgsvcx.exe;avgcsrva.exe
open
t=%s time=[%02d:%02d:%02d-%02d/%02d/%d]
SubmitSamplesConsent
Winsta0
Create
%SystemRoot%\SysWOW64\explorer.exe
%S.%06d
*/*
mcshield.exe
application/x-shockwave-flash
SOFTWARE\Wow6432Node\Microsoft AntiMalware\SpyNet
ws2_32.dll
%SystemRoot%\SysWOW64\xwizard.exe
reg.exe ADD "HKLM\%s" /f /t %s /v "%s" /d "%s"
```

[Figure 30]

```
snxhk_border_mywnd
Set objWMIService = GetObject("winmgmts:" & "{impersonationLevel=impersonate}!\\.\%coot\cimv2")
Set objProcess = GetObject("winmgmts:root\cimv2:Win32_Process")
errReturn = objProcess.Create("%s", null, nul, nul)
wpcap.dll
select
Win32_ComputerSystem
SysWOW64
fmon.exe
AvastSvc.exe
kernel32.dll
%SystemRoot%\System32\mobsync.exe
netapi32.dll
Content-Type: application/x-www-form-urlencoded
%ProgramFiles(x86)%\Internet Explorer\iexplore.exe
advapi32.dll
SOFTWARE\Microsoft\Windows Defender\Exclusions\Paths
c:\\
%SystemRoot%\explorer.exe
aswhookx.dll
%SystemRoot%\explorer.exe
https
Software\Microsoft
%ProgramFiles%\Internet Explorer\iexplore.exe
System32
dwengine.exe;dwarkdaemon.exe;dwwatcher.exe
Win32_PhysicalMemory
type=0x%04X
%SystemRoot%\System32\xwizard.exe
wtsapi32.dll
.dat
aswhooka.dll
LastBootUpTime
SELECT * FROM Win32_Processor
Mozilla/5.0 (Windows NT 6.1; rv:77.0) Gecko/20100101 Firefox/77.0
Win32_Process
Win32_DiskDrive
Name
fshoster32.exe
SpyNetReporting
S:(ML;;NW;;;LW)
WQL
c:\hiberfil.sysss
SOFTWARE\Microsoft\Microsoft AntiMalware\SpyNet
Set objWMIService = GetObject("winmgmts:" & "{impersonationLevel=impersonate}!\\.\%coot\cimv2")
Set colFiles = objWMIService.ExecQuery("Select * From CIM_DataFile Where Name = '%s'")
For Each objFile in colFiles
objFile.Copy("%s")
Next
aaebcdeeifghiiojklmnooupqrstuuyvwxyyaz
Caption,Description,Vendor,Version,InstallDate,InstallSource,PackageName
```

[Figure 31]

```
Caption,Description,Vendor,Version,InstallDate,InstallSource,PackageName
SystemRoot
SELECT * FROM AntiVirusProduct
%SystemRoot%\SysWOW64\mobsync.exe
wininet.dll
CommandLine
SELECT * FROM Win32_OperatingSystem
winsta0\default
ROOT\CIMV2
Caption
SOFTWARE\Microsoft\Windows Defender\SpyNet
NTUSER.DAT
Caption,Description,DeviceID,Manufacturer,Name,PNPDeviceID,Service,Status
ntdll.dll
TRUE
SAVAdminService.exe;SavService.exe
.exe
image/jpeg
wmic process call create 'expand "%S" "%S"'

displayName
 from
userenv.dll
urlmon.dll
Initializing database...
ccSvcHst.exe
%SystemRoot%\System32\OneDriveSetup.exe
ByteFence.exe
Win32_Product
WScript.Sleep %u
Set objWMIService = GetObject("winmgmts:" & "{impersonationLevel=impersonate}!\\.\%coot\cimv2")
Set objProcess = GetObject("winmgmts:root\cimv2:Win32_Process")
errReturn = objProcess.Create("%s", null, nul, nul)
WSCript.Sleep 2000
Set fso = CreateObject("Scripting.FileSystemObject")
fso.DeleteFile("%s")
image/gif
abcdefghijklmnopqrstuvwxyz
root\SecurityCenter2
%SystemRoot%\SysWOW64\msra.exe
\sf2.dll
aabcdeefghiijklmnoopqrstuuvwxyyz
Win32_Bios
%s\system32\
shell32.dll


string[1486]: kernel32.dll
```

[Figure 32]

If you check the output, there're several quite meaningful strings that provide a good indication of techniques and checking done by the binary. Additionally, and as occurs in any modern malware, strings are encrypted, so it is not longer possible easily to find and follow the malware's code using clear strings.

Note the last line of the output: the script extracts correctly a string at byte 1486 of the string table, so this possibility will be useful later.

A very simple function that is useful is **sub_10008773 (mw_ww_string_length) → sub_100087CB (mw_w_string_length)**, which calculates a string length and its used frequently on the code (list cross-references on **sub_10008773/mw_ww_string_length** using **'X' hotkey**). Additionally, it's a good opportunity to use an useful resource of IDA Pro that's the user **enumeration** creation. As in this piece of code **"-1" means "CHAR" and "-2" means "WCHAR"**, so I created a enumeration (**Figure 36**) that contains only two fields and used **'M' hotkey** to change the values. How did I discover about "**char**" and "**wchar**"? The subroutine being called on **line 17 (sub_1000C52C/mw_str_length_char)** accepts **_BYTE** as argument and subroutine **sub_1000C545** (**mw_str_length_wchar**) on **line 21** accepts **_WORD** as argument.

```
1  _BYTE **__cdecl mw_w_string_length(_BYTE **string, int char_or_wchar)
2  {
3    _BYTE **result; // eax
4    _BYTE *v3; // esi
5    int v4; // eax
6
7    result = string;
8    if ( string )
9    {
10     v3 = *string;
11     if ( *string )
12     {
13       *string = 0;
14       v4 = char_or_wchar;
15       if ( char_or_wchar == CHAR )
16       {
17         v4 = mw_str_length_char(v3);
18       }
19       else if ( char_or_wchar == WCHAR )
20       {
21         v4 = mw_str_length_wchar(v3);
22       }
23       mw_memset(v3, 0, v4);
24       return (_BYTE **)HeapFree(var_heap_hndl, 0, v3);
25     }
26   }
27   return result;
28 }
```

[Figure 33]

```
1  int __cdecl mw_str_length_char(_BYTE *string_1)
2  {
3    int counter; // eax
4
5    counter = 0;
6    if ( string_1 && *string_1 )
7    {
8      do
9        ++counter;
10     while ( string_1[counter] );
11   }
12   return counter;
13 }
```

[Figure 34]

```
1  int __thiscall mw_str_length_wchar(_WORD *string_1)
2  {
3    int counter; // eax
4
5    counter = 0;
6    if ( string_1 && *string_1 )
7    {
8      do
9        ++counter;
10     while ( string_1[counter] );
11   }
12   return counter;
13 }
```

[Figure 35]

```
FFFFFFFF ; enum char_wchar, mappedto_138
FFFFFFFF WCHAR                  = 0FFFFFFFEh
FFFFFFFF CHAR                   = 0FFFFFFFFh
```

[Figure 36]

The subroutine **sub_10012C0A (mw_ww_construct_imports) → sub_10012C0A (mw_ww_fill_imports)** brings something new in this series of article that's the **PE format manipulation**:

```
1  int __cdecl sub_10012C0A(int a1)
2  {
3    int result; // eax
4    int v2; // [esp+Ch] [ebp-8h]
5    int v3; // [esp+10h] [ebp-4h]
6
7    v3 = *(_DWORD *)(a1 + *(_DWORD *)(a1 + 60) + 128) + a1;
8    v2 = 0;
9    while ( *(_DWORD *)(v3 + 12) )
10   {
11     v3 += 20;
12     v2 += 20;
13   }
14   dword_1001E664 = (int)sub_1000881A((_BYTE *)(*(_DWORD *)(a1 + *(_DWORD *)(a1 + 60) + 128) + a1), v2);
15   if ( !dword_1001E664 )
16     return 0;
17   result = v2;
18   dword_1001E668 = v2;
19   return result;
20 }
```

[Figure 37]

If you convert these numbers (60 and 128) to hexadecimal (**'H' hotkey**), you'll find two important numbers: 0x3C and 0x80. The **0x3C** is an indication of **e_lfanew** field from **IMAGE_DOS_HEADER structure** and **0x80** is an indication of **Import Directory (_IMAGE_IMPORT_DESCRIPTOR** structure**)** within **IMAGE_DATA_DIRECTORY structure**. Two excellent references about **PE Format** follow below:

- **PE 102 (by Corkami):** https://github.com/corkami/pics/blob/master/binary/pe102/pe102.pdf
- **PE File Format Offsets (by Sunshine):** http://www.sunshine2k.de/reversing/tuts/tut_pe.htm

Sometimes it isn't so simple to figure out what's happening during a PE manipulation, so both these references above will be quite useful and, if you can, work with them together. Our first step is trying to improve this pseudo code and, as a requested step, we have to add few PE structures into IDA Pro and change some variables' types to point to them.

To add new standard structures, go to **Structures tab (SHIFT-F9)**, press **"INS" key**, click on "**Add standard structure**" and type **"_IMAGE_DOS_HEADER".** Once IDA found it, choose it and click on OK button. Repeat the same steps to **_IMAGE_IMPORT_DESCRIPTOR.**

Although only these two PE structures are used in this specific threat (for example, during the **Remcos malware** further structures are necessary) , there are other ones you might want to insert into IDA Pro when it's needed. A shortened list of them follow:

- **_IMAGE_DOS_HEADER**
- **_IMAGE_NT_HEADERS**
- **_IMAGE_DATA_DIRECTORY**
- **_IMAGE_EXPORT_DIRECTORY**
- **_IMAGE_OPTIONAL_HEADER**
- **_IMAGE_SECTION_HEADER**
- **_IMAGE_IMPORT_DESCRIPTOR**

After adding the structures mentioned previously, let's perform a sequence of tasks:

a. check them at **Structures tab** to confirm their presence there then put the cursor on offset **60 (or 0x3C)**, press **"T" hotkey** (**Select a structure** functionality) and choose **_IMAGE_DOS_HEADER**. Likely the **IDA Decompiler** has changed the representation of the offset to **e_lfanew** from **_IMAGE_DOS_HEADER structure**. **Update the decompiler with F5**.

b. Using both documents about PE format that I left as reference, it seems that taking **0x3C** as reference, the offset **0x80 (128)** seems to indicate the structure **_IMAGE_IMPORT_DESCRIPTOR** (**ImportDirectory)**, which belongs to **_IMAGE_DATA_DIRECTORY** structure.

c. Therefore, **click on "v3"** and **press 'Y' hotkey** to change the type from "**int v3**" to "**_IMAGE_IMPORT_DESCRIPTOR *v3**" (please: note that is a pointer to the structure, so it includes a '*' symbol). **Update the decompiler with F5**.

d. Repeat the same step on variable **dword_1001E664** by changing its type from "**int**" to "**_IMAGE_IMPORT_DESCRIPTOR *"**. **Update the decompiler with F5**.

e. Finally, click on number "**20**" and press **"T" hotkey** (**Select a structure**) and pick up **_IMAGE_IMPORT_DESCRIPTOR**. **Update the decompiler with F5**.

After having renamed some variables and a function (to be commented in next paragraphs), the IDA pseudo code looks a bit better:

```
int __cdecl mw_prepare_Import_Structures(int hndl_DLL)
{
  int var_Import_Directory_size_1; // eax
  int var_Import_Directory_size; // [esp+Ch] [ebp-8h]
  _IMAGE_IMPORT_DESCRIPTOR *ptr_IMPORT_IMPORT_DESCRIPTOR; // [esp+10h] [ebp-4h]

  ptr_IMPORT_IMPORT_DESCRIPTOR = (_IMAGE_IMPORT_DESCRIPTOR *)(*(_DWORD *)(hndl_DLL
                                                                          + *(_DWORD *)(hndl_DLL
                                                                                        + offsetof(_IMAGE_DOS_HEADER, e_lfanew))
                                                                          + 0x80)
                                                              + hndl_DLL);
  var_Import_Directory_size = 0;
  while ( ptr_IMPORT_IMPORT_DESCRIPTOR->Name )
  {
    ++ptr_IMPORT_IMPORT_DESCRIPTOR;
    var_Import_Directory_size += sizeof(_IMAGE_IMPORT_DESCRIPTOR);
  }
  ptr_Import_Structures = (_IMAGE_IMPORT_DESCRIPTOR *)mw_w_construct_import_structures(
                                                        (_BYTE *)(*(_DWORD *)(hndl_DLL
                                                                              + *(_DWORD *)(hndl_DLL
                                                                                            + offsetof(_IMAGE_DOS_HEADER, e_lfanew))
                                                                              + 0x80)
                                                                  + hndl_DLL),
                                                        var_Import_Directory_size);
  if ( !ptr_Import_Structures )
    return 0;
  var_Import_Directory_size_1 = var_Import_Directory_size;
  import_offset_counter = var_Import_Directory_size;
  return var_Import_Directory_size_1;
}
```

[Figure 38]

Moving inside the subroutine named **sub_1000881A** (**mw_w_construct_import_structures**), I recommend you to rename all arguments according to argument's names from **Figure 38**.

After accomplishing this task and renaming few variables, we have:

```
_BYTE *__cdecl mw_w_construct_import_structures(_BYTE *ptr_IMPORT_IMPORT_DESCRIPTOR, int var_Import_Directory_size)
{
  _BYTE *prt_to_Import_Descriptors_heap; // eax
  _BYTE *ptr_Import_Structures; // esi

  prt_to_Import_Descriptors_heap = mw_HeapAlloc(var_Import_Directory_size + 2);
  ptr_Import_Structures = prt_to_Import_Descriptors_heap;
  if ( prt_to_Import_Descriptors_heap )
  {
    mw_construct_import_structures(
      (int)prt_to_Import_Descriptors_heap,
      ptr_IMPORT_IMPORT_DESCRIPTOR,
      var_Import_Directory_size);
    ptr_Import_Structures[var_Import_Directory_size] = 0;
  }
  return ptr_Import_Structures;
}
```

[Figure 39]

The **mw_HeapAlloc API** has the following code after having renamed function's name, arguments and **applied a standard enumeration** by using **'M' hotkey** (look for 'HEAP' and the exact constant name appears):

```
LPVOID __cdecl mw_HeapAlloc(SIZE_T dwBytes)
{
  return HeapAlloc(var_heap_hndl, HEAP_ZERO_MEMORY, dwBytes);
}
```

[Figure 40]

Proceeding with our analysis, go into subroutine **sub_10008892** (**mw_construct_import_structures**) , rename variables and arguments (**'N' hotkey**), and you will have something similar to this:

```
int __cdecl mw_construct_imports(
        int prt_to_Import_Descriptors_heap,
        _BYTE *ptr_IMPORT_IMPORT_DESCRIPTOR,
        int var_Import_Directory_size)
{
  int updated_ptr_Import_Descriptors; // eax
  int var_Import_Directory_counter; // esi
  _BYTE *ptr_IMPORT_IMPORT_DESCRIPTOR_1; // edx

  updated_ptr_Import_Descriptors = prt_to_Import_Descriptors_heap;
  var_Import_Directory_counter = var_Import_Directory_size;
  if ( var_Import_Directory_size )
  {
    ptr_IMPORT_IMPORT_DESCRIPTOR_1 = ptr_IMPORT_IMPORT_DESCRIPTOR;
    do
    {
      ptr_IMPORT_IMPORT_DESCRIPTOR_1[prt_to_Import_Descriptors_heap - (_DWORD)ptr_IMPORT_IMPORT_DESCRIPTOR] = *ptr_IMPORT_IMPORT_DESCRIPTOR_1;
      ++ptr_IMPORT_IMPORT_DESCRIPTOR_1;
      --var_Import_Directory_counter;
    }
    while ( var_Import_Directory_counter );
  }
  return updated_ptr_Import_Descriptors;
}
```

[Figure 40]

Now we can return to **DllEntryPoint routine** (press **CTRL + E** and choose **DllEntryPoint**), analyze the next subroutine that, no doubts, is the most important one so far because it is going to introduce a new concept in this series of articles that's **dynamic API resolving** because, as you'll see, **APIs functions are represented by hashes**.

Go into **sub_1000E369 subroutine**, which is **called 13 times** (**'X' hotkey**), and there you are going to find **sub_10009773 subroutine** (**mw_w_decode_string_table**), which we've already analyzed and, if you check its cross-references (**'X' hotkey**), you will discover that it's **called 23 times**! Furthermore, you're going to find an API being called, but it doesn't have a well-formed name (**dword_1001E684**) and points to **zeroed** data at **.data section**. That's a big indicator of **dynamic API resolution**! Your first view of this piece of code should be the following one:

```
1  _DWORD *__usercall sub_1000E369@<eax>(SIZE_T a1@<edx>, int a2@<ecx>, int a3)
2  {
3    _DWORD *v5; // esi
4    HMODULE ModuleHandleA; // eax
5    char *v8; // [esp+Ch] [ebp-4h] BYREF
6
7    v5 = 0;
8    v8 = sub_10009773();
9    if ( a3 == 1486 )
10     ModuleHandleA = GetModuleHandleA(v8);
11   else
12     ModuleHandleA = (HMODULE)(*(int (__stdcall **)(char *))dword_1001E684)(v8);
13   if ( ModuleHandleA )
14     v5 = sub_1000E31E(a1, a2, (int)ModuleHandleA);
15   sub_10008773(&v8);
16   return v5;
17 }
```

[Figure 41]

Have you remember about the routine "**string_decrypter_search**" from our the Python script (**Figure 28**)? As a test, we've used "**1486**" as offset to look for the exact string on the string table and we found "**kernel32.dll**" (last line on **Figure 32**). Thus, we could insert a comment here to remember later.

Go into **sub_1000E31E** and you're see the subroutine **sub_100087B5**, which is our previously renamed wrapper to **HeapAlloc( ).** You'll be able to rename few **local variables**, but there is not much more than it.

Once you enter into **sub_1000E15A**, few secrets start to be revealed because we'll see several hints about what's happening:

- several well-known decimal number being used (**60, 120, 124** and so on).
- two calls subroutines: **sub_1000C52C** and **sub_1000D5AA**.
- a very interesting and essential **XOR operation** happening.
- another hexadecimal being used: **0x6C6C642E**
- Calls to **LoadLibrary( )** and **GetProcAddress( )**

Before explaining anything, you are already able to guess what's happening: **API resolving**! Yes, we finally reached our first API resolving case in our series and you'll learn how every information is really important here.

Therefore, lets me show both **sub_1000E31E** and **sub_1000E15A** (without any renaming and just like you'll be seeing) before proceeding with our analysis:

```
_DWORD *__usercall sub_1000E31E@<eax>(SIZE_T a1@<edx>, int a2@<ecx>, int a3)
{
  _DWORD *result; // eax
  _DWORD *v5; // edi
  SIZE_T v6; // esi
  FARPROC *v7; // ebx
  int v8; // edi
  _DWORD *v9; // [esp+8h] [ebp-8h]

  result = sub_100087B5(a1);
  v5 = result;
  v9 = result;
  if ( result )
  {
    v6 = a1 >> 2;
    if ( v6 )
    {
      v7 = (FARPROC *)result;
      v8 = a2 - (_DWORD)result;
      do
      {
        *v7 = sub_1000E15A(a3, *(int *)((char *)v7 + v8));
        ++v7;
        --v6;
      }
      while ( v6 );
      return v9;
    }
    return v5;
  }
  return result;
}
```

[Figure 42]

```
FARPROC __fastcall sub_1000E15A(int a1, int a2)
{
  int v3; // ebx
  int v4; // eax
  int v5; // edx
  int v6; // edx
  unsigned int v7; // ecx
  unsigned int v8; // eax
  _BYTE *v9; // esi
  int v10; // eax
  FARPROC result; // eax
  unsigned int v12; // ecx
  const CHAR *v13; // esi
  unsigned int i; // eax
  CHAR v15; // cl
  bool v16; // zf
  HMODULE LibraryA; // eax
  CHAR LibFileName[4]; // [esp+Ch] [ebp-58h] BYREF
  char v19[60]; // [esp+10h] [ebp-54h]
  int v20; // [esp+4Ch] [ebp-18h]
  int v21; // [esp+50h] [ebp-14h]
  int v22; // [esp+54h] [ebp-10h]
  unsigned int v23; // [esp+58h] [ebp-Ch]
  int v24; // [esp+5Ch] [ebp-8h]
  unsigned int v25; // [esp+60h] [ebp-4h]

```

[Figure 43]

```
27  v24 = a2;
28  v3 = *(_DWORD *)(a1 + 60);
29  v4 = *(_DWORD *)(v3 + a1 + 120);
30  if ( !v4 )
31    return 0;
32  v5 = *(_DWORD *)(v4 + a1 + 32);
33  v21 = a1 + *(_DWORD *)(v4 + a1 + 36);
34  v6 = a1 + v5;
35  v7 = *(_DWORD *)(v4 + a1 + 24);
36  v20 = a1 + *(_DWORD *)(v4 + a1 + 28);
37  v8 = 0;
38  v22 = v6;
39  v25 = 0;
40  v23 = v7;
41  if ( !v7 )
42    return 0;
43  while ( 1 )
44  {
45    v9 = (_BYTE *)(a1 + *(_DWORD *)(v6 + 4 * v8));
46    v10 = sub_1000C52C(v9);
47    if ( (sub_1000D5AA(v10, (int)v9, 0) ^ 0x218FE95B) == v24 )
48      break;
49    v6 = v22;
50    v8 = v25 + 1;
51    v25 = v8;
52    if ( v8 >= v23 )
53      return 0;
54  }
55  v12 = a1 + *(_DWORD *)(v3 + a1 + 120);
56  v13 = (const CHAR *)(a1 + *(_DWORD *)(v20 + 4 * *(unsigned __int16 *)(v21 + 2 * v25)));
57  if ( (unsigned int)v13 < v12 || (unsigned int)v13 >= v12 + *(_DWORD *)(v3 + a1 + 124) )
58    return (FARPROC)(a1 + *(_DWORD *)(v20 + 4 * *(unsigned __int16 *)(v21 + 2 * v25)));
59  for ( i = 0; i < 0x40; ++i )
60  {
61    v15 = v13[i];
62    if ( v15 == 46 )
63      break;
64    if ( !v15 )
65      break;
66    LibFileName[i] = v15;
67  }
68  v16 = v13[i] == 0;
69  *(_DWORD *)&LibFileName[i] = 0x6C6C642E;
70  v19[i] = 0;
71  if ( !v16 )
72    v13 += i + 1;
73  LibraryA = LoadLibraryA(LibFileName);
74  if ( !LibraryA )
75    return 0;
76  result = GetProcAddress(LibraryA, v13);
77  if ( !result )
78    return 0;
79  return result;
80 }
```

[Figure 44]

As reversing tasks become more elaborated, it's advisable to rule a better name convention, so it might use the suffix "**arg**" for every argument and "**var**" for all local variables. Sometimes I get out of using suffix "**var**" because it might make the code very polluted, but it depends on the situation and context.

I believe that code associated to **sub_100087B5** subroutine (on **Figure 42**) is really understandable and you don't have any issues.

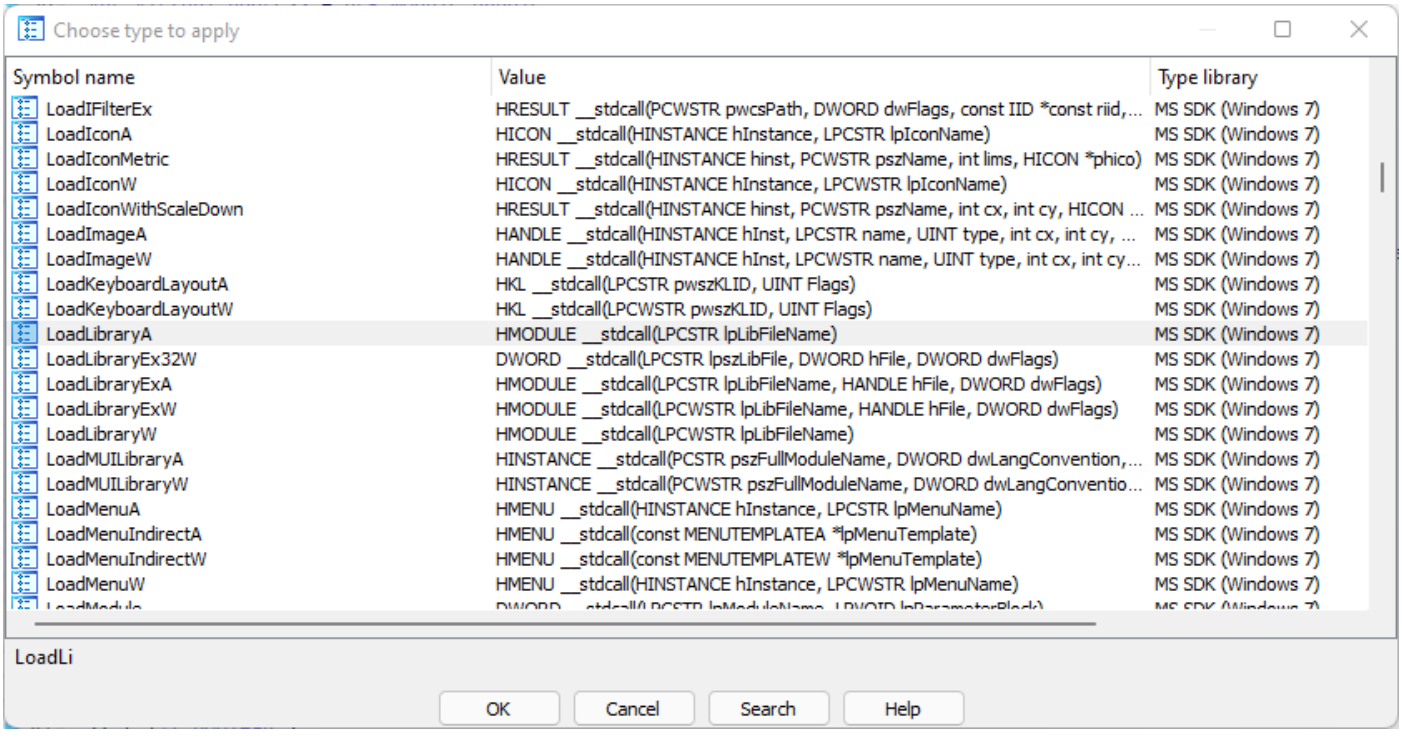
Within **sub_100087B5** subroutine, there's only a **HeapAlloc( )** call, which you can change the second argument to a symbolic representation (**'M' hotkey**), which results in **HEAP_ZERO_MEMORY**.

The **sub_1000E15A** is different and presents relevant details. Once again, you need to read those mentioned documents about PE format and, according to offsets, change the associated types. For example, the decimal **"60" (0x3C – remember: to convert to hexadecimal, use 'H' hotkey)** refers to **e_lfanew field** (from **IMAGE_DOS_HEADERS**) that points to **_IMAGE_NT_HEADERS**, so you can change the type of "v3" (int) to **_IMAGE_NT_HEADERS***. A similar approach can be done with decimal **"120" (0x78)**, which points to **Export Directory,** so you should change the **v4's type (int)** to **_IMAGE_EXPORT_DIRECTORY *** (**you must import this standard structure into your IDA before using it).** Likely the aspect of your code will become a bit better.

Another quite interesting aspect is the **XOR operation** with a fixed value (**0x218FE95B**) in one of the lines because the malware is calling the **sub_1000D5AA subroutine** and performing a XOR with its returned value. As you'll learn about it, this constant value (**0x218FE95B**) is essential for us.

Near to end of the code (**Figure 44**), there're two API calls: **LoadLibrary( )** and **GetProcAddress( )**. To make code better and use standard argument types, it's recommended to invoke a plugin named "**ApplyCalleeType**" that we mentioned previously.

Thus go to **Edit → Plugins → ApplyCalleeType → Use Standard Type** and **search for the given API name** as shown below. Afterwards, **update the IDA Decompiler's view (F5)**. Repeat this procedure to each API calls over the code.



[Figure 45]

I've renamed all subroutines, arguments and variables, so the reversed code after working on instructions from **Figure 43** and **Figure 44** follows below:

```
FARPROC __fastcall mw_hashing_api(int arg_module_handle, int arg_api_offset)
{
  _IMAGE_NT_HEADERS *rva_IMAGE_NT_HEADERS; // ebx
  _IMAGE_EXPORT_DIRECTORY *rva_IMAGE_EXPORT_DIRECTORY; // eax
  int array_AddressOfNames; // edx
  int ptr_AddressOfNames; // edx
  unsigned int rva_NumberOfNames; // ecx
  unsigned int counter; // eax
  _BYTE *var_api_name; // esi
  int api_name_length; // eax
  FARPROC result; // eax
  DWORD var_virtual_address; // ecx
  const CHAR *rva_AddressOfNameOrdinals; // esi
  unsigned int counter_2; // eax
  CHAR var_NameOrdinal; // cl
  bool zf_boolean; // zf
  HMODULE LibraryA; // eax
  CHAR LibFileName[4]; // [esp+Ch] [ebp-58h] BYREF
  char array_size_60[60]; // [esp+10h] [ebp-54h]
  DWORD array_AddressOfFunctions; // [esp+4Ch] [ebp-18h]
  DWORD ptr_AddressOfNameOrdinals; // [esp+50h] [ebp-14h]
  int ptr_heap_allocated; // [esp+54h] [ebp-10h]
  unsigned int var_NumberOfNames; // [esp+58h] [ebp-Ch]
  int var_api_name_offset; // [esp+5Ch] [ebp-8h]
  unsigned int updated_counter; // [esp+60h] [ebp-4h]

  var_api_name_offset = arg_api_offset;
  rva_IMAGE_NT_HEADERS = *(_IMAGE_NT_HEADERS **)(arg_module_handle + offsetof(_IMAGE_DOS_HEADER, e_lfanew));
  rva_IMAGE_EXPORT_DIRECTORY = *(_IMAGE_EXPORT_DIRECTORY **)((char *)&rva_IMAGE_NT_HEADERS->OptionalHeader.DataDirectory[0].VirtualAddress
                                                           + arg_module_handle);
  if ( !rva_IMAGE_EXPORT_DIRECTORY )
    return 0;
  array_AddressOfNames = *(DWORD *)((char *)&rva_IMAGE_EXPORT_DIRECTORY->AddressOfNames + arg_module_handle);
  ptr_AddressOfNameOrdinals = arg_module_handle
                            + *(DWORD *)((char *)&rva_IMAGE_EXPORT_DIRECTORY->AddressOfNameOrdinals + arg_module_handle);
  ptr_AddressOfNames = arg_module_handle + array_AddressOfNames;
  rva_NumberOfNames = *(DWORD *)((char *)&rva_IMAGE_EXPORT_DIRECTORY->NumberOfNames + arg_module_handle);
  array_AddressOfFunctions = arg_module_handle
                           + *(DWORD *)((char *)&rva_IMAGE_EXPORT_DIRECTORY->AddressOfFunctions + arg_module_handle);
  counter = 0;
  ptr_heap_allocated = ptr_AddressOfNames;
  updated_counter = 0;
  var_NumberOfNames = rva_NumberOfNames;
  if ( !rva_NumberOfNames )
    return 0;
  while ( 1 )
  {
    var_api_name = (_BYTE *)(arg_module_handle + *(_DWORD *)(ptr_AddressOfNames + 4 * counter));
    api_name_length = mw_str_length_char(var_api_name);
    if ( (mw_crc32(api_name_length, (int)var_api_name, 0) ^ 0x218FE95B) == var_api_name_offset )
      break;
    ptr_AddressOfNames = ptr_heap_allocated;
    counter = updated_counter + 1;
    updated_counter = counter;
    if ( counter >= var_NumberOfNames )
      return 0;
  }
  var_virtual_address = arg_module_handle
                      + *(DWORD *)((char *)&rva_IMAGE_NT_HEADERS->OptionalHeader.DataDirectory[0].VirtualAddress
                                   + arg_module_handle);
  rva_AddressOfNameOrdinals = (const CHAR *)(arg_module_handle
                                           + *(_DWORD *)(array_AddressOfFunctions
                                                       + 4
                                                       * *(unsigned __int16 *)(ptr_AddressOfNameOrdinals
                                                                             + 2 * updated_counter)));
  if ( (unsigned int)rva_AddressOfNameOrdinals < var_virtual_address
    || (unsigned int)rva_AddressOfNameOrdinals >= var_virtual_address
                                                + *(DWORD *)((char *)&rva_IMAGE_NT_HEADERS->OptionalHeader.DataDirectory[0].Size
                                                           + arg_module_handle) )
  {
    return (FARPROC)(arg_module_handle
                   + *(_DWORD *)(array_AddressOfFunctions
                               + 4 * *(unsigned __int16 *)(ptr_AddressOfNameOrdinals + 2 * updated_counter)));
  }
  for ( counter_2 = 0; counter_2 < 64; ++counter_2 )
  {
    var_NameOrdinal = rva_AddressOfNameOrdinals[counter_2];
    if ( var_NameOrdinal == 0x2E )
      break;
    if ( !var_NameOrdinal )
```

```
81        break;
82      LibFileName[counter_2] = var_NameOrdinal;
83    }
84    zf_boolean = rva_AddressOfNameOrdinals[counter_2] == 0;
85    *(_DWORD *)&LibFileName[counter_2] = _dll;
86    array_size_60[counter_2] = 0;
87    if ( !zf_boolean )
88      rva_AddressOfNameOrdinals += counter_2 + 1;
89    LibraryA = LoadLibraryA(LibFileName);
90    if ( !LibraryA )
91      return 0;
92    result = GetProcAddress(LibraryA, rva_AddressOfNameOrdinals);
93    if ( !result )
94      return 0;
95    return result;
96  }
```

[Figure 46]

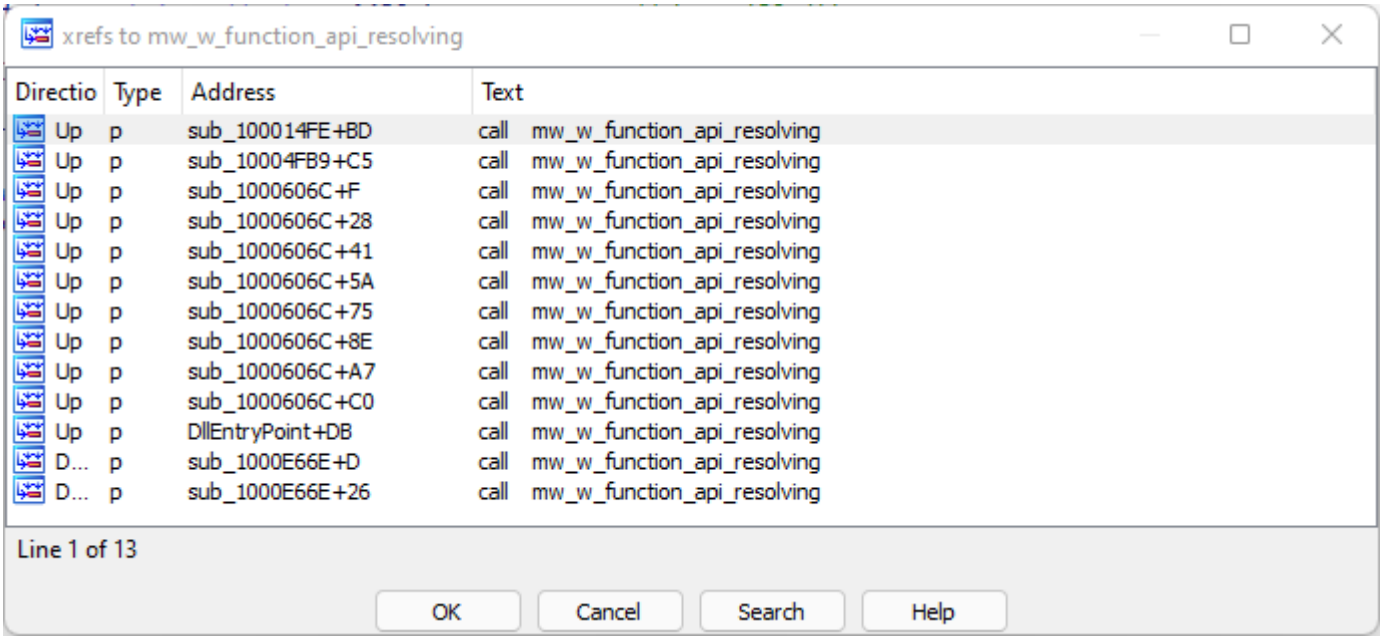
Going into the pending **sub_ 1000C52C subroutine**, we have the following reversed code:

```
int __usercall mw_crc32@<eax>(unsigned int crc32_data_length@<edx>, int crc32_data@<ecx>, int crc32_start_arg)
{
  int crc32; // esi
  unsigned int counter; // ecx
  unsigned int var_crc32_lookup; // esi

  crc32 = ~crc32_start_arg;
  if ( !crc32_data_length )
    return 0;
  for ( counter = 0; counter < crc32_data_length; ++counter )
  {
    var_crc32_lookup = mw_crc_constants[(*(_BYTE *)(counter + crc32_data) ^ (unsigned __int8)crc32) & 0xF] ^ ((*(unsigned __int8 *)(counter + crc32_data) ^ (unsigned int)crc32) >> 4);
    crc32 = mw_crc_constants[var_crc32_lookup & 0xF] ^ (var_crc32_lookup >> 4);
  }
  return ~crc32;
}
```

[Figure 47]

I used this reference to **CRC32 algorithm** to interpret the code on **Figure 47**: https://en.wikipedia.org/wiki/Cyclic_redundancy_check. Furthermore**, IDA marks CRC32 constants** and **Findcrypt plugin** also reports CRC32 usage.

Therefore, we know that this malware is probably using **CRC32** as hashing function for **resolving API names**, but how can be find further evidence and manage this API resolution statically? Returns to subroutine **sub_1000E369 (mw_w_function_api_resolving)** and ask for cross-references (**'X' hotkey**):



[Figure 48]

All these subroutines are calling **sub_1000E369**, so there's a good chance to discover further evidences within these subroutines. Analyzing the references above, there're several calls coming from **sub_1000606C**, so it's a good choice to go and examine the code:

```
1  DWORD *sub_1000606C()
2  {
3    _DWORD *result; // eax
4
5    dword_1001E684 = (int)sub_1000E369(0x11Cu, (int)&unk_1001BA30, 1486);
6    dword_1001E6BC = (int)sub_1000E369(0x28u, (int)&unk_1001BB50, 2912);
7    dword_1001E694 = (int)sub_1000E369(0x48u, (int)&unk_1001BB80, 531);
8    dword_1001E6B4 = (int)sub_1000E369(0x18u, (int)&unk_1001BBCC, 1533);
9    dword_1001E68C = (int)sub_1000E369(0xCCu, (int)&unk_1001BBE8, 1645);
10   dword_1001E690 = (int)sub_1000E369(0x2Cu, (int)&unk_1001BCB8, 764);
11   dword_1001E698 = (int)sub_1000E369(8u, (int)&unk_1001BCE8, 3648);
12   result = sub_1000E369(4u, (int)&unk_1001BCF4, 3042);
13   dword_1001E6B8 = (int)result;
14   return result;
15 }
```

[Figure 49]

If you check all unexplored bytes references (suffix **unk_**), there're many encrypted bytes that, once you convert some of them to double word, you will have the following bytes:

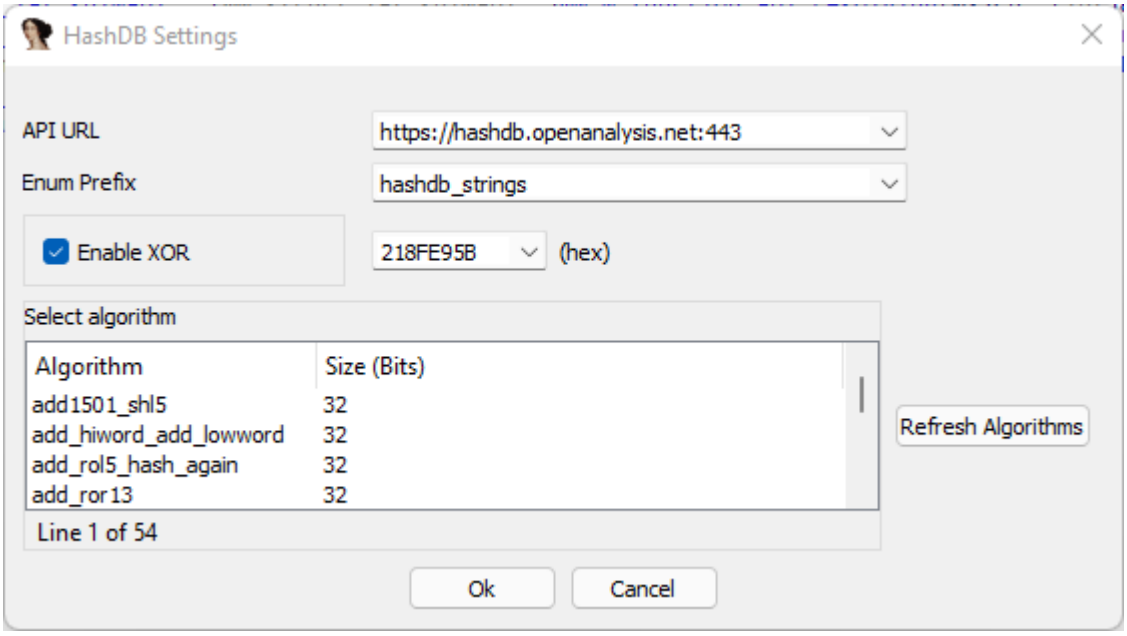
```
.rdata:1001BA30 dword_1001BA30  dd 1E4E54D6h            ; DATA XREF: sub_1000606C+A↑o
.rdata:1001BA30                                         ; DllEntryPoint+D6↑o
.rdata:1001BA34                 dd 0EA9AE187h
.rdata:1001BA38                 dd 0FBE7CAD4h
.rdata:1001BA3C                 dd 0E8F3F6A4h
.rdata:1001BA40                 dd 90098C2Bh
.rdata:1001BA44                 dd 0E07C512Dh
.rdata:1001BA48                 dd 1906F55Bh
.rdata:1001BA4C                 dd 0D71EF109h
.rdata:1001BA50                 dd 6ED77E75h
.rdata:1001BA54                 dd 0FEA8B810h
.rdata:1001BA58                 dd 4AC7C978h
.rdata:1001BA5C                 dd 0C1D7521Eh
.rdata:1001BA60                 db 0AFh
.rdata:1001BA61                 db 0FCh
.rdata:1001BA62                 db  1Ch
.rdata:1001BA63                 db  91h
.rdata:1001BA64                 db 0D0h
.rdata:1001BA65                 db  1Eh
```

[Figure 50]

According to our previous analysis, it is likely these double words are CRC32 hashes. Throughout of my career, I was used to writing scripts for decrypting any kind of encrypted bytes, but it was a time consuming work. Afterwards, I used the **make_sc_hash_db.py** script (from flare-ida project: https://github.com/mandiant/flare-ida/tree/master/shellcode_hashes) and it really helped me a lot. Since last year we have available the excellent **HashDB IDA plugin** authored by OALabs, which it's a welcome evolution and significant improvement from **make_sc_hash_db.py** idea, and it have been very useful.

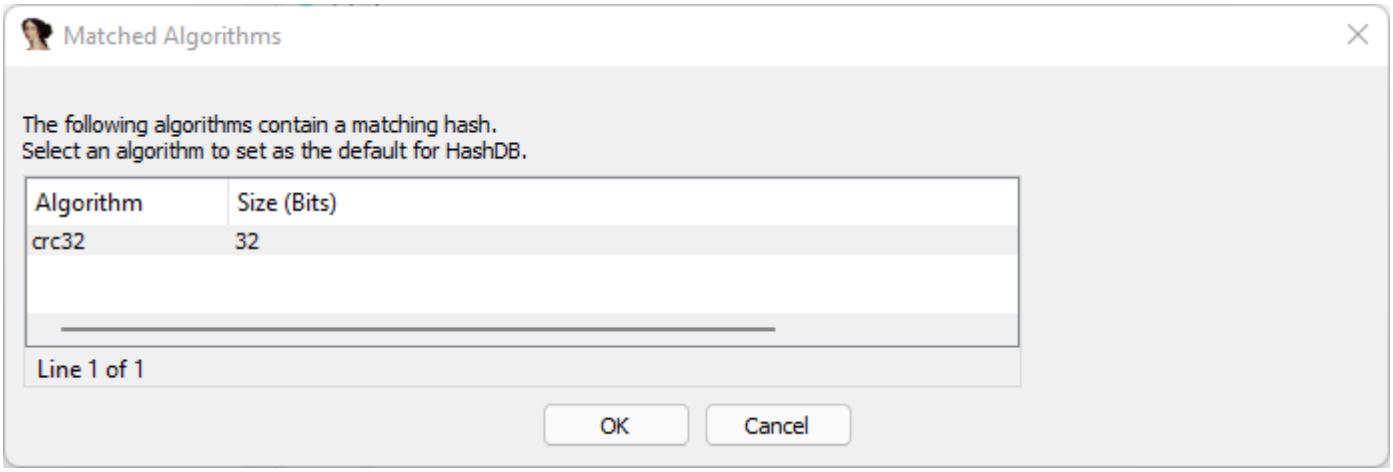
How can we use **HashDB plugin** to tackle our resolving API issue? The first step is to remember of a prior and critical information that's the **XOR key** used in the CRC32 operation: **0x218FE95B**.

Therefore, our first step is going to **Edit → Plugins → HashDB**, **setting that XOR key** (personally, I think better right clicking on the **XOR key** and choose **"HashDB set XOR key")** and click on **Refresh Algorithms**.



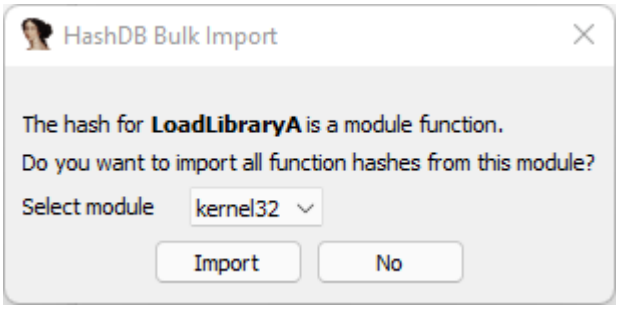
[Figure 51]

Now you must right click on the first hash (**Figure 50**) and choose "**HashDB Hunt Algorithm**":



[Figure 52]

As expected, the CRC32 was detected. You must **mark** the CRC32 algorithm above and click **OK**. Now right click again on the first CRC32 hash and choose **HashDB Lookup** that you see the following message:



[Figure 52]

That's great! **HashDB plugin**, which communicates with OALabs server, discovered that the first hash stands for **LoadLibraryA( )** from **kernel32.dll**, so probably all of the following hashes (up to a sequence of four zeroed bytes) are also **CRC32 hashes** from the same module. In this case, it's worth to press **"Import" button** to bring all of hashes into the IDA database. Please, be patient because it takes some time and, while it's working, IDA Pro seems to hang.

Now you must **select all of the remaining bytes until the sequence of zeros**, **right click** and **choose HuntDB Scan IAT**. This operation take some seconds to finish. Once the task is completed, you'll see the following:

```
.rdata:1001BA30 dword_1001BA30  dd LoadLibraryA_0       ; DATA XREF: sub_1000606C+A↑o
.rdata:1001BA30                                         ; DllEntryPoint+D6↑o
.rdata:1001BA34 ptr_LoadLibraryW dd LoadLibraryW_0
.rdata:1001BA38 ptr_FreeLibrary dd FreeLibrary_0
.rdata:1001BA3C ptr_GetProcAddress dd GetProcAddress_0
.rdata:1001BA40 ptr_GetModuleHandleA dd GetModuleHandleA_0
.rdata:1001BA44 ptr_CreateToolhelp32Snapshot dd CreateToolhelp32Snapshot_0
.rdata:1001BA48 ptr_Module32First dd Module32First_0
.rdata:1001BA4C ptr_Module32Next dd Module32Next_0
.rdata:1001BA50 ptr_WriteProcessMemory dd WriteProcessMemory_0
.rdata:1001BA54 ptr_OpenProcess dd OpenProcess_0
.rdata:1001BA58 ptr_VirtualFreeEx dd VirtualFreeEx_0
.rdata:1001BA5C ptr_WaitForSingleObject dd WaitForSingleObject_0
.rdata:1001BA60 ptr_CloseHandle dd CloseHandle_0
.rdata:1001BA64 ptr_LocalFree   dd LocalFree_0
.rdata:1001BA68 ptr_CreateProcessW dd CreateProcessW_0
.rdata:1001BA6C ptr_ReadProcessMemory dd ReadProcessMemory_0
.rdata:1001BA70 ptr_Process32First dd Process32First_0
.rdata:1001BA74 ptr_Process32Next dd Process32Next_0
.rdata:1001BA78 ptr_Process32FirstW dd Process32FirstW_0
.rdata:1001BA7C ptr_Process32NextW dd Process32NextW_0
.rdata:1001BA80 ptr_CreateProcessAsUserW dd CreateProcessAsUserW_0
.rdata:1001BA84 ptr_VirtualAllocEx dd VirtualAllocEx_0
.rdata:1001BA88 ptr_VirtualAlloc dd VirtualAlloc_0
.rdata:1001BA8C ptr_OpenThread  dd OpenThread_0
.rdata:1001BA90 ptr_Wow64DisableWow64FsRedirection dd Wow64DisableWow64FsRedirection_0
.rdata:1001BA94 ptr_Wow64EnableWow64FsRedirection dd Wow64EnableWow64FsRedirection_0
.rdata:1001BA98 ptr_GetVolumeInformationW dd GetVolumeInformationW_0
.rdata:1001BA9C ptr_IsWow64Process dd IsWow64Process_0
.rdata:1001BAA0 ptr_CreateThread dd CreateThread_0
.rdata:1001BAA4 ptr_CreateFileW dd CreateFileW_0
.rdata:1001BAA8 ptr_FindClose   dd FindClose_0
.rdata:1001BAAC ptr_GetFileAttributesW dd GetFileAttributesW_0
.rdata:1001BAB0 ptr_SetFilePointer dd SetFilePointer_0
.rdata:1001BAB4 ptr_WriteFile   dd WriteFile_0
.rdata:1001BAB8 ptr_ReadFile    dd ReadFile_0
.rdata:1001BABC ptr_CreateMutexA dd CreateMutexA_0
.rdata:1001BAC0 ptr_ReleaseMutex dd ReleaseMutex_0
.rdata:1001BAC4 ptr_FindResourceA dd FindResourceA_0
.rdata:1001BAC8 ptr_SizeofResource dd SizeofResource_0
.rdata:1001BACC ptr_LoadResource dd LoadResource_0
.rdata:1001BAD0 ptr_GetTickCount64 dd GetTickCount64_0
.rdata:1001BAD4 ptr_ExpandEnvironmentStringsW dd ExpandEnvironmentStringsW_0
```

[Figure 53]

That's excellent! If you search down from this address, you'll figure out that there're many sequence of bytes that represent API's names from other functions, so you should hunt them and repeat the same step by marking all block of bytes until a sequence of zeroed bytes, right clicking and choosing **HuntDB Scan IAT**.

Although it might seem a bit tedious, this task is very worthwhile and produces an amazing result. All of hashes (and its respective function's names) are in .**rdata** section.

It's quite relevant to mention that all that **HashDB plugin** creates an enumeration named *"hashdb_strings_<algorithm_name>"* and you're able to check it up going **to View → OpenSubviews → Enumerations** (**SHIFT+F10**).

The figure follows only for illustrating few other blocks of hashes and their respective function names:

```
.rdata:1001BB50 ptr_RtlAllocateHeap dd RtlAllocateHeap_0
.rdata:1001BB50                                         ; DATA XREF: sub_1000606C+1B↑o
.rdata:1001BB50                                         ; sub_1000C681+10D↑o
.rdata:1001BB54 ptr_RtlFreeHeap dd RtlFreeHeap_0
.rdata:1001BB58 ptr_RtlGetVersion dd RtlGetVersion_0
.rdata:1001BB5C ptr_NtCreateSection dd NtCreateSection_0
.rdata:1001BB60 ptr_NtUnmapViewOfSection dd NtUnmapViewOfSection_0
.rdata:1001BB64 ptr_NtMapViewOfSection dd NtMapViewOfSection_0
.rdata:1001BB68 ptr_NtWriteVirtualMemory dd NtWriteVirtualMemory_0
.rdata:1001BB6C ptr_NtProtectVirtualMemory dd NtProtectVirtualMemory_0
.rdata:1001BB70 ptr_NtClose     dd NtClose_0
.rdata:1001BB74 ptr_ZwQueryInformationThread dd ZwQueryInformationThread_0
.rdata:1001BB78                 db    0
.rdata:1001BB79                 db    0
.rdata:1001BB7A                 db    0
.rdata:1001BB7B                 db    0
.rdata:1001BB7C                 db    0
.rdata:1001BB7D                 db    0
.rdata:1001BB7E                 db    0
.rdata:1001BB7F                 db    0
.rdata:1001BB80 ptr_MessageBoxA dd MessageBoxA_0        ; DATA XREF: sub_1000606C+34↑o
.rdata:1001BB84 ptr_EnumWindows dd EnumWindows_0
.rdata:1001BB88 ptr_RegisterClassExA dd RegisterClassExA_0
.rdata:1001BB8C ptr_CreateWindowExA dd CreateWindowExA_0
.rdata:1001BB90 ptr_ChangeWindowMessageFilter dd ChangeWindowMessageFilter_0
.rdata:1001BB94 ptr_ShowWindow  dd ShowWindow_0
.rdata:1001BB98 ptr_UpdateWindow dd UpdateWindow_0
.rdata:1001BB9C ptr_GetMessageA dd GetMessageA_0
.rdata:1001BBA0 ptr_TranslateMessage dd TranslateMessage_0
.rdata:1001BBA4 ptr_DispatchMessageA dd DispatchMessageA_0
.rdata:1001BBA8 ptr_DestroyWindow dd DestroyWindow_0
.rdata:1001BBAC ptr_UnregisterClassA dd UnregisterClassA_0
.rdata:1001BBB0 ptr_PostQuitMessage dd PostQuitMessage_0
.rdata:1001BBB4 ptr_DefWindowProcA dd DefWindowProcA_0
.rdata:1001BBB8 ptr_GetKeyboardLayoutList dd GetKeyboardLayoutList_0
.rdata:1001BBBC ptr_GetSystemMetrics dd GetSystemMetrics_0
.rdata:1001BBC0 ptr_GetProcessWindowStation dd GetProcessWindowStation_0
.rdata:1001BBC4 ptr GetUserObjectInformationW dd GetUserObjectInformationW 0
```

[Figure 54]

I won't reproduce each "API block" here, but you're going to find APIs from **ntdll.dll, user32.dll, ntapi32.dll, advapi32.dll, shlwapi.dll, shell32.dll, userenv.dll** and so on.

After having run **HashDB plugin** to decrypt all API names, the next step is creating structures for holding the API block from each DLL. For example, all APIs represented in **Figure 53** comes from kernel32.dll, so you should mark all of them, right-click and choose **"Create struct from selection"** . Click on the name of the just created struct and use '**N' hotkey** to give a name that reflects the structure's purpose and DLL's name where all these APIs are found (for example: **mw_struct_iat_kernel32**). Repeat the same steps for

other blocks of API's names and, of course, if you don't remember the DLL's name from each block of API's names come from, so search them on Google.

After creating structures (one for each API names' block) you can verify them on **Structures tab** (**SHIFT-F9**) as shown below:

```
> 00000000 ; [0000011C BYTES. COLLAPSED STRUCT mw_struct_iat_kernel32. PRESS CTRL-NUMPAD+ TO EXPAND]
> 00000000 ; [00000028 BYTES. COLLAPSED STRUCT mw_struct_iat_ntdll. PRESS CTRL-NUMPAD+ TO EXPAND]
> 00000000 ; [00000048 BYTES. COLLAPSED STRUCT mw_struct_iat_user32. PRESS CTRL-NUMPAD+ TO EXPAND]
> 00000000 ; [00000018 BYTES. COLLAPSED STRUCT mw_struct_iat_netapi32. PRESS CTRL-NUMPAD+ TO EXPAND]
> 00000000 ; [000000CC BYTES. COLLAPSED STRUCT mw_struct_iat_advapi32. PRESS CTRL-NUMPAD+ TO EXPAND]
> 00000000 ; [0000002C BYTES. COLLAPSED STRUCT mw_struct_iat_shlwapi. PRESS CTRL-NUMPAD+ TO EXPAND]
> 00000000 ; [00000008 BYTES. COLLAPSED STRUCT mw_struct_iat_shell32. PRESS CTRL-NUMPAD+ TO EXPAND]
> 00000000 ; [00000004 BYTES. COLLAPSED STRUCT mw_struct_iat_userenv. PRESS CTRL-NUMPAD+ TO EXPAND]
```

[Figure 55]

The code presented on **Figure 49** should have changed to the following one:

```
1  _DWORD *sub_1000606C()
2  {
3    _DWORD *result; // eax
4
5    dword_1001E684 = (int)sub_1000E369(0x11Cu, (int)&ptr_LoadLibraryA, 1486);
6    dword_1001E6BC = (int)sub_1000E369(0x28u, (int)&ptr_RtlAllocateHeap, 2912);
7    dword_1001E694 = (int)sub_1000E369(0x48u, (int)&ptr_MessageBoxA, 531);
8    dword_1001E6B4 = (int)sub_1000E369(0x18u, (int)&ptr_NetShareEnum, 1533);
9    dword_1001E68C = (int)sub_1000E369(0xCCu, (int)&ptr_SetFileSecurityW, 1645);
10   dword_1001E690 = (int)sub_1000E369(0x2Cu, (int)&ptr_StrStrIA, 764);
11   dword_1001E698 = (int)sub_1000E369(8u, (int)&ptr_ShellExecuteW, 3648);
12   result = sub_1000E369(4u, (int)&ptr_GetUserProfileDirectoryW, 3042);
13   dword_1001E6B8 = (int)result;
14   return result;
15 }
```

[Figure 56]

We don't finished it yet and there are further procedures as shown below:

- **Rename** (**'N' hotkey**) subroutine **sub_1000E369** to something better (in my case, as I've mentioned previously, **mw_w_function_api_resolving**).

- **Rename** (**'N' hotkey**) **variables** on left to reflect the structure's content. For example, dword_1001E684 to **ptr_struct_iat_kernel32**.

- **Change the type of each renamed variable of left** from '**int**' to '**mw_struct_iat_kernel32 ***'. Pay attention: it's a pointer, so there must be the '*' at end.

- **Remember:** if you committed any mistake, you can undo the operation (**CTRL+Z**) anytime.

This name and type scheme are very recommended to make the pseudo code more readable and easier to understand what's going on. After having finished these steps, the code on **Figure 56** should be much better:

```
mw_struct_iat_userenv *mw_iat_construction()
{
  mw_struct_iat_userenv *result; // eax

  ptr_struct_iat_kernel32 = (mw_struct_iat_kernel32 *)mw_w_function_api_resolving(
                                                        0x11Cu,
                                                        (int)&ptr_LoadLibraryA_1,
                                                        1486);
  ptr_struct_iat_ntdll = (mw_struct_iat_ntdll *)mw_w_function_api_resolving(0x28u, (int)&ptr_RtlAllocateHeap, 2912);
  ptr_struct_iat_user32 = (mw_struct_iat_user32 *)mw_w_function_api_resolving(0x48u, (int)&ptr_MessageBoxA, 531);
  ptr_struct_iat_netapi32 = (mw_struct_iat_ntapi32 *)mw_w_function_api_resolving(0x18u, (int)&ptr_NetShareEnum, 1533);
  ptr_struct_iat_advapi32 = (mw_struct_iat_advapi32 *)mw_w_function_api_resolving(
                                                        0xCCu,
                                                        (int)&ptr_SetFileSecurityW,
                                                        1645);
  ptr_struct_iat_shlwapi = (mw_struct_iat_shlwapi *)mw_w_function_api_resolving(0x2Cu, (int)&ptr_StrStrIA, 764);
  ptr_struct_iat_shell32 = (mw_struct_iat_shell32 *)mw_w_function_api_resolving(8u, (int)&ptr_ShellExecuteW, 3648);
  result = (mw_struct_iat_userenv *)mw_w_function_api_resolving(4u, (int)&ptr_GetUserProfileDirectoryW, 3042);
  ptr_struct_iat_userenv = result;
  return result;
}
```

[Figure 57]

Having managed these issues we can continue our analysis. From this point onwards, we'll use the same **API hash resolving** technique by concerning to hash's resolving and structure types only.

The **mw_iat_construction subroutine** is called from three places (**'X' hotkey**), so we can examine one of them (for example, **sub_10005FBC** subroutine):

```
int __userpurge sub_10005FBC@<eax>(int a1@<edi>, int a2)
{
  struct _OSVERSIONINFOA *v2; // eax

  mw_iat_construction();
  v2 = (struct _OSVERSIONINFOA *)sub_1000D1C9(a1);
  dword_1001E688 = v2;
  if ( !v2 )
    return 1;
  v2[1].dwBuildNumber = 1;
  sub_10012B10(*(_DWORD *)&dword_1001E688[3].szCSDVersion[84]);
  if ( (*(_DWORD *)&dword_1001E688[42].szCSDVersion[60] & 0x10000) != 0 )
  {
    dword_1001E688[1].dwPlatformId = 1;
  }
  else
  {
    if ( sub_1000CBDA(*(_DWORD *)&dword_1001E688[3].szCSDVersion[84], a1) )
    {
      if ( *(_DWORD *)&dword_1001E688[3].szCSDVersion[68] != 3 )
        return 0;
LABEL_10:
      sub_10003184();
      return 0;
    }
    dword_1001E688[1].dwPlatformId = 1;
  }
  if ( *(_DWORD *)&dword_1001E688[3].szCSDVersion[68] == 3 )
    goto LABEL_10;
  sub_10005DDC(a1);
  return 0;
}
```

[Figure 58]

In the decompiler, before continue analyzing the code, **update it by pressing F5**. Go inside **sub_1000D1C9**, you'll have a large piece of code and part of it follows below (I renamed only the three first lines):

```
32  var_hinstDLL_ref = var_hinstDLL;
33  ptr_allocated_memory_1 = mw_HeapAlloc(6852u);
34  ptr_allocated_memory_1_ref = ptr_allocated_memory_1;
35  if ( ptr_allocated_memory_1 )
36  {
37    ptr_allocated_memory_1[1424] = GetCurrentProcessId();
38    v4 = ((int (__stdcall *)(int))ptr_struct_iat_kernel32->ptr_GetTickCount64)(a1);
39    sub_100124AE(v4 + ptr_allocated_memory_1_ref[1424], ptr_allocated_memory_1_ref + 402);
40    if ( GetModuleFileNameW(0, (LPWSTR)ptr_allocated_memory_1_ref + 2850, 0x105u) )
41      ptr_allocated_memory_1_ref[1557] = sub_1000916A((__int16 *)ptr_allocated_memory_1_ref + 2850);
42    CurrentProcess = GetCurrentProcess();
43    v6 = (void **)sub_1000BBB8(CurrentProcess);
44    ptr_allocated_memory_1_ref[68] = v6;
45    if ( sub_1000BD40(*v6) )
46      ptr_allocated_memory_1_ref[133] = 3;
47    else
48      ptr_allocated_memory_1_ref[133] = (sub_1000BC15() > 0) + 1;
49    ptr_allocated_memory_1_ref[134] = sub_1000E59E(ptr_allocated_memory_1_ref + 136);
50    ptr_allocated_memory_1_ref[135] = sub_1000E563();
51    ptr_allocated_memory_1_ref[137] = var_hinstDLL_ref;
52    v27 = 128;
53    v28 = 256;
54    if ( !((int (__stdcall *)(_DWORD, _DWORD, _DWORD *, int *, char *, int *, char *))ptr_struct_iat_advapi32->ptr_LookupAccountSidW)(
55            0,
56            *(_DWORD *)ptr_allocated_memory_1_ref[68],
57            ptr_allocated_memory_1_ref + 69,
58            &v27,
59            v23,
60            &v28,
61            v26) )
62      GetLastError();
63    v7 = ((int (__stdcall *)(int))ptr_struct_iat_user32->ptr_GetSystemMetrics)(4096);
64    v20 = (HMODULE)ptr_allocated_memory_1_ref[137];
65    ptr_allocated_memory_1_ref[1556] = v7 > 0;
66    GetModuleFileNameW(v20, (LPWSTR)ptr_allocated_memory_1_ref + 276, 0x105u);
67    GetLastError();
68    ptr_allocated_memory_1_ref[269] = sub_1000916A((__int16 *)ptr_allocated_memory_1_ref + 276);
69    v8 = sub_1000B958((LPCWSTR)ptr_allocated_memory_1_ref + 138, v23);
70    ptr_allocated_memory_1_ref[43] = v8;
71    sub_1000B82D(v8, (int)(ptr_allocated_memory_1_ref + 44));
72    if ( ptr_allocated_memory_1_ref != (_DWORD *)-176 )
```

[Figure 59]

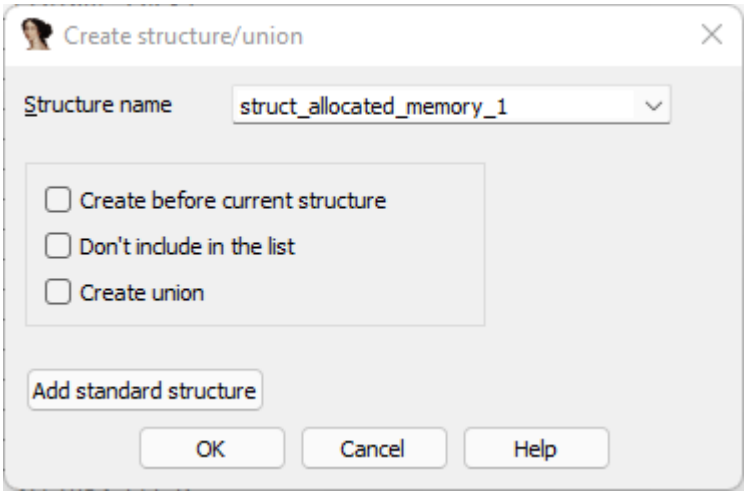
We won't analyze all this code right now, but only interesting parts. Furthermore, it's suitable opportunity to show few well-known techniques when we have large allocation of heap and its respective usage (as an array) spread over several parts of the code. In this case, the malware **allocates 6852 bytes and soon afterwards it does several attribution of values to slots of this array** and, definitely, working with indexes (1557, 133, 134, 135, 137…) is not a good idea. Based on this information, **we can create a structure of 1713 DWORDs (it's huge!)** and assume that each one can hold a pointer (32-bit). Of course, there're many doubts about this statement:

- How do we know **whether the structure is only composed by dwords**?
- How do we know **whether the structure is not composed by other structures too**?

Pay attention here: initially, we're going to generate a simple structure here containing **1713 double words (dd)**. Usually **creating a simple structure that contains double words is what we do and almost always works**, but **our decision is WRONG** in this case and there's a clear hint on **Figure 58**. **Why am I doing it? To show how things really work in a real malware analysis**. Later I'll return to this point, but I want to be clear in saying that the **technique which I'm going to show you is CORRECT**, but in this case the **interpretation is WRONG**. Thus, we have two ways to create a simple structure composed by dwords:

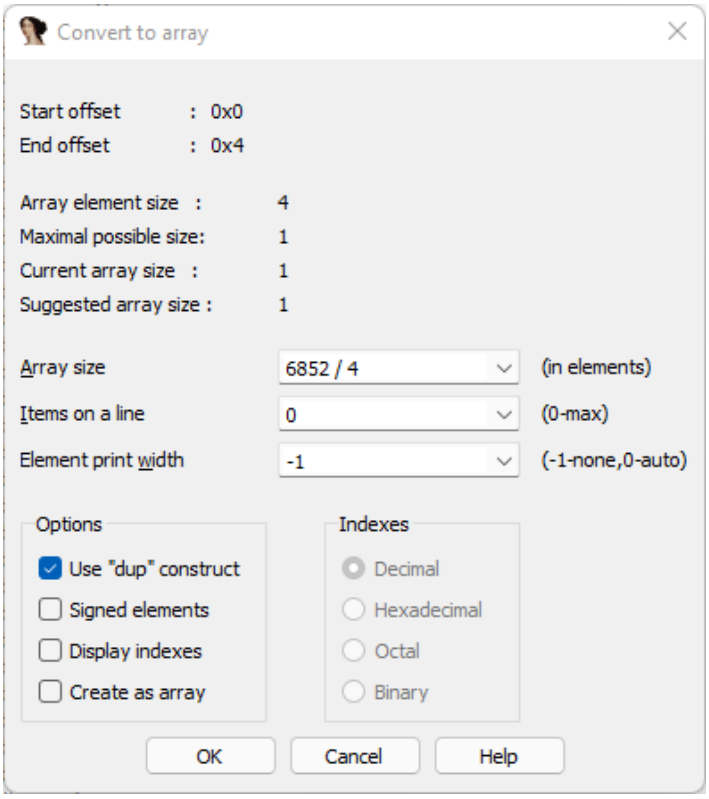
- Use the own **IDA Pro features to accomplish the task automatically**.
- Create your **own structure as a C code** then **import it** into IDA Pro's database.

Using IDA Pro features directly is easier (no doubts), so we **can create a structure in Structures tab (SHIFT+F9)** with given name of name "**struct_allocated_memory_1**" (I'll change it for something better later) and also create a first field (**field_0**) being dword (**dd**). If you don't know how to create a structure, don't freak out because it's quite simple. Press **"INS"** to create the structure, provide a **name (in my case was struct_allocated_memory_1)** , press **OK**. The exact window is shown below:



[Figure 60]

Click on the "end" of the structure (***00001AC4 struct_allocated_memory_1 ends***) and **press "D"** to create a new field (field_0). **Click on this first and unique field_0 and press "D"** twice until its type has changed to double word (dd). Finally, **press '*'** and the following window will come up:



[Figure 61]

Please, pay attention to details:

- **The array size is 6852/4.**
- **The checkbox "Create as array" is unchecked.**

Finally, click on any **ptr_allocated_memory_1_ref** variable and press **"Y"** hotkey. Change its type to the same type of the structure you just created previously (**Figure 60**). For example, in my case:

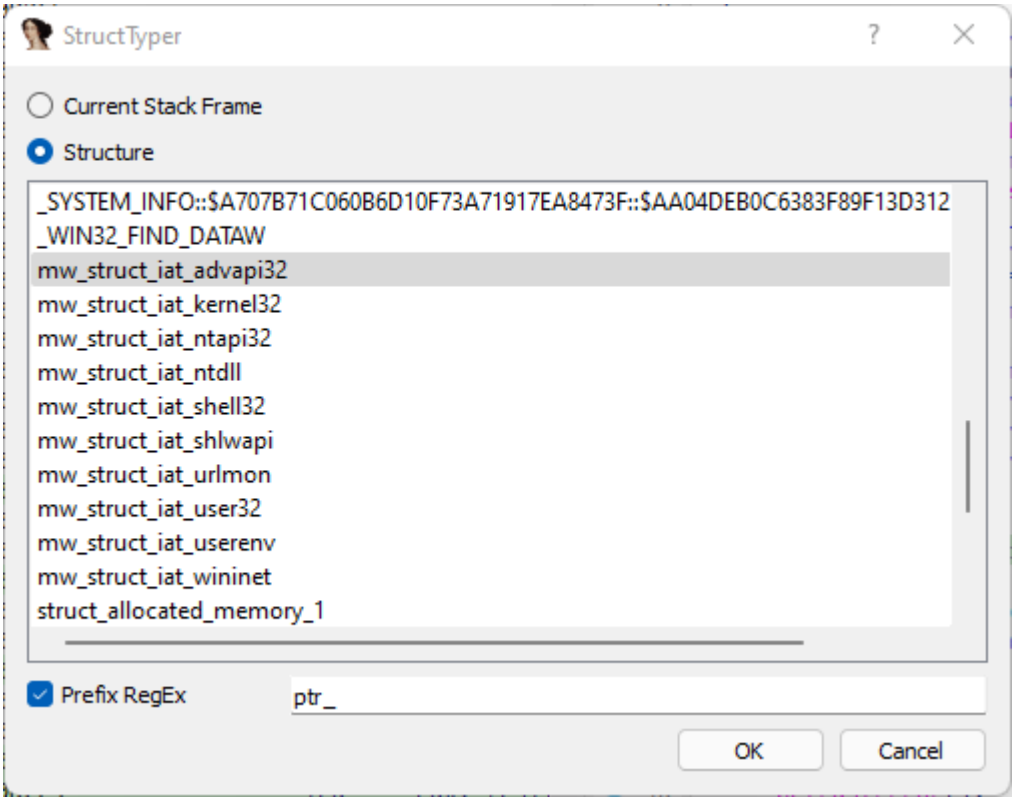
- (before the change): **_DWORD*** ptr_allocated_memory_1_ref
- (after the change): **struct_allocated_memory_1*** ptr_allocated_memory_1_ref

Press F5 to update the Decompiler view.

We can do other interesting improvement. On **line 54** from **Figure 59** you can see the following code:

- if (!((int (__stdcall *)(_DWORD, _DWORD, int *, int *, char *, int *, char *))ptr_struct_iat_advapi32->**ptr_LookupAccountSidW**)(…

Click on the the respective structure's member (**ptr_LookupAccountSidW)**, which is a well-known Windows API. Go to **Edit → Plugins → StructTyper**:



[Figure 62]

Find the structure (**mw_struct_iat_advapi32**), mark "**Prefix RegEx**" and fill it up with any suffix (usually I use initial of my name, but here I'm using '**ptr_**') and press **OK**.

Only a note: remember that **StructTyper plugin** is recommended to apply **argument's type** to well-known structures, so it's perfect for our case. Anyway, you do NOT need to use this plugin because, even without using it, our pseudo code is OK.

Press F5 to update the compiler and, as you'll see, the final code will be much better:

```
32  var_hinstDLL_ref = var_hinstDLL;
33  ptr_allocated_memory_1 = (struct_allocated_memory_1 *)mw_HeapAlloc(0x1AC4u);
34  ptr_allocated_memory_1_ref = ptr_allocated_memory_1;
35  if ( ptr_allocated_memory_1 )
36  {
37    ptr_allocated_memory_1->current_process_id = GetCurrentProcessId();
38    time_miliseconds = ((int (__stdcall *)(int))ptr_struct_iat_kernel32->ptr_GetTickCount64)(a1);
39    marsenne_twister(
40      time_miliseconds + ptr_allocated_memory_1_ref->current_process_id,
41      &ptr_allocated_memory_1_ref->field_648);
42    if ( GetModuleFileNameW(0, (LPWSTR)&ptr_allocated_memory_1_ref->field_1644, 0x105u) )
43      ptr_allocated_memory_1_ref->field_1854 = (int)mw_maybe_cmp((__int16 *)&ptr_allocated_memory_1_ref->field_1644);
44    CurrentProcess = GetCurrentProcess();
45    ptr_buffer_TokenInformation = (void **)mw_w_GetTokenInformation(CurrentProcess);
46    ptr_allocated_memory_1_ref->ptr_SID_2 = (int)ptr_buffer_TokenInformation;
47    if ( mw_AllocateAndInitializeSid(*ptr_buffer_TokenInformation) )
48      ptr_allocated_memory_1_ref->ptr_SID = 3;
49    else
50      ptr_allocated_memory_1_ref->ptr_SID = ((int)mw_AllocateAndInitializeSid_0() > 0) + 1;
51    ptr_allocated_memory_1_ref->join_status = (int)mw_NetGetJoinInformation(&ptr_allocated_memory_1_ref->field_220);
52    ptr_allocated_memory_1_ref->primary_domain_controller = (int)mw_NetGetDCName();
53    ptr_allocated_memory_1_ref->hinstDLL_ref = var_hinstDLL_ref;
54    var_value_128_size_lpName = 128;
55    Wow64Process = 256;
56    if ( !ptr_struct_iat_advapi32->ptr_LookupAccountSidW(
57            0,
58            *(PSID *)ptr_allocated_memory_1_ref->ptr_SID_2,
59            (LPWSTR)&ptr_allocated_memory_1_ref->ptr_AccountName,
60            (LPDWORD)&var_value_128_size_lpName,
61            (LPWSTR)v23,
62            (LPDWORD)&Wow64Process,
63            (PSID_NAME_USE)v26) )
64      GetLastError();
65    var_SystemMetrics = ptr_struct_iat_user32->ptr_GetSystemMetrics(4096);
66    field_224_hinstDLL_ref = (HMODULE)ptr_allocated_memory_1_ref->hinstDLL_ref;
67    ptr_allocated_memory_1_ref->thereis_system_metrics = var_SystemMetrics > 0;
68    GetModuleFileNameW(field_224_hinstDLL_ref, (LPWSTR)&ptr_allocated_memory_1_ref->lpFilename, 0x105u);
69    GetLastError();
70    ptr_allocated_memory_1_ref->field_434 = (int)mw_maybe_cmp((__int16 *)&ptr_allocated_memory_1_ref->lpFilename);
71    v8 = mw_Computer_Volume_Information((LPCWSTR)&ptr_allocated_memory_1_ref->ptr_AccountName, v23);
72    ptr_allocated_memory_1_ref->field_AC = v8;
73    mw_randomized_string_marsenne(v8, (int)&ptr_allocated_memory_1_ref->field_B0);
```

[Figure 63]

Therefore, two general recommendations are:

- If there's an API call, so try the **ApplyCalleeType plugin** on the API to improve arguments representation to well-defined types and names according (or similar) to MSDN.

- If there's a call (usually seen when handling API hashing) where the API's name is a member of a structure, so try to use **StructTyper plugin** on the **structure's member** and choose the structure containing the API member.

Continuing our analysis, there're other places in the code that also use encrypted strings, so our script to decrypt strings would be useful again. For example, going to **sub_10004C5A → sub_10002D5C → sub_1000109A**, which calls the **mw_string_decryptor** subroutine using different arguments:

```
1  _WORD *__cdecl sub_1000109A(unsigned int a1)
2  {
3    int v1; // ecx
4
5    return mw_string_decryptor(0x4F3u, (int)&unk_1001D0B0, (int)&unk_1001D050, v1, a1);
6  }
```

[Figure 64]

As shown above, the encrypted strings is at address **0x1001D0B0** and the decryption key is at address **0x1001D050**, so calling our script is enough to decrypt them:

```
def main():
    string_decrypter('0x1001D000','0x1001D0B0', '0x1001D050')

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

```
%s "$%s = \"%s\"; & $%s"
powershell.exe
schtasks.exe /Create /RU "NT AUTHORITY\SYSTEM" /SC ONSTART /TN %u /TR "%s" /NP /F
Self test FAILED!!!
ProgramData
Microsoft
whoami /all
Red Hat VirtIO;QEMU
net localgroup
SOFTWARE\Microsoft\Windows NT\CurrentVersion\ProfileList
error res='%s' err=%d len=%u
Self test OK.
arp -a
"%s\system32\schtasks.exe" /Create /RU "NT AUTHORITY\SYSTEM" /tn %s /tr "%s" /SC ONCE /Z /ST %02u:%02u /ET %02u:%02u
schtasks.exe /Delete /F /TN %u
srvpost.exe;frida-winjector-helper-32.exe;frida-winjector-helper-64.exe
route print
nslookup -querytype=ALL -timeout=10 _ldap._tcp.dc._msdcs.%s
nltest /domain_trusts /all_trusts
5812
3719
.lnk
netstat -nao
c:\ProgramData
at.exe %u:%u "%s" /I
\System32\WindowsPowerShell\v1.0\powershell.exe
amstream.dll
net view /all
qwinsta
%s %04x.%u %04x.%u res: %s seh_test: %u consts_test: %d vmdetected: %d createprocess: %d
cmd /c set
 /c ping.exe -n 6 127.0.0.1 &  type "%s\System32\calc.exe" > "%s"
\System32\WindowsPowerShell\v1.0\powershell.exe
SOFTWARE\Microsoft\Windows\CurrentVersion\Run
jHxastDcds)oMc=jvh7wdUhxcsdt2
artifact.exe;mlwr_smpl;sample;sandbox;cuckoo-;virus
Virtual
ProfileImagePath
VIRTUAL-PC
ipconfig /all
net share
A3E64E55_pr;VBoxVideo
%s \"$%s = \\\"%s\\\\; & $%s\"
/t4
regsvr32.exe -s
```

[Figure 65]

As we've already decrypted two string tables using the same script, so it'd better to improve it to make our lives easier. Therefore, let's use IDA Python to **make comments** at each call for **mw_w_decode_string_table** subroutine with the exact string given by the offset address in table string. Does it sound complicated? Certainly it's much easier you might imagine as you will learn below:

```python
import idaapi
import idautils
import binascii
import pefile

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

# This routine populates the string table. Pay attention to separator '\x00'.
def make_string_table(string_data):
    str_table = []
    for k in string_data.split('\x00'):
        str_table.append(k)
    return str_table

# This routine only prints the string table.
def print_string_table(my_table):
    for j in range(0, len(my_table)):
            print(my_table[j])

# This routine searches for a string in the string table. Pay attention:
# the search is through a given offset in bytes and not a slot of this table.
def string_decrypter_search(arg_string, arg_key, str_addr):
    local_table = []
    for i in range(0, len(arg_string)):
        local_table.append((arg_string[i]) ^ (arg_key[i % len(arg_key)]))
    converted_table = bytes(local_table)[str_addr:].decode('latin').split('\x00')[0]
    return (str_addr, converted_table)

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

# 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

# data_seg_start: start address of the provided .data segment
# encrypted_string_addr: start address of the encrypted strings
# key_data_addr: start address of the key used to decrypt strings

def string_decrypter(arg_encrypted_string_addr, arg_key_data_addr, str_offset):

    data_1 = b''
    data_2 = b''

    # Converts arguments the appropriated format and notation.
    encrypted_string_addr = hex(int(arg_encrypted_string_addr))
    key_data_addr = hex(int(arg_key_data_addr))

```

[Figure 66]

```
    # Find the start address of the .data segment.
    for segment in idautils.Segments():
        if '.data' == idc.get_segm_name(segment):
            data_seg_start = hex(int(idc.get_segm_start(segment)))

    # Next two lines calculates the offset between data blogbs and start of the .data segment.
    encrypted_string_addr_rel = calc_offsets(data_seg_start, encrypted_string_addr)
    key_data_addr_rel = calc_offsets(data_seg_start, key_data_addr)

    # Next three lines extracts .data section's information.
    filename = r"C:\Users\Administrador\Desktop\MAS\mas_3\stage_1.bin"
    data_encoded_extracted_1 = extract_data(filename)
    data_encoded_extracted_2 = extract_data(filename)

    # Next six lines calculates the size of the encrypted string table and the XOR key. Pay attention that
    # I've used an approach of searching up to two "end of string" marker is found.
    d1_off = 0x0
    d2_off = 0x0
    if (b'\x00\x00' in data_encoded_extracted_1[int(encrypted_string_addr_rel,16):]):
        d1_off = (data_encoded_extracted_1[int(encrypted_string_addr_rel,16):]).index(b'\x00\x00')
    if (b'\x00\x00' in data_encoded_extracted_2[int(key_data_addr_rel,16):]):
        d2_off = (data_encoded_extracted_2[int(key_data_addr_rel,16):]).index(b'\x00\x00')

    # Uncomment next 4 lines to print the hexadecimal representation of the extracted bytes.
    #print("encrypted_string_addr:")
    #print(binascii.b2a_hex(data_encoded_extracted_1[int(encrypted_string_addr_rel,16):_
    # int(encrypted_string_addr_rel,16) + d1_off]))
    #print("\nkey_data_addr:")
    #print(binascii.b2a_hex(data_encoded_extracted_2[int(key_data_addr_rel,16):int(key_data_addr_rel,16) + d2_off]))

    # Next two lines the meaningful information (encrypted string table and XOR key) are isolated.
    data_1 = data_encoded_extracted_1[int(encrypted_string_addr_rel,16):int(encrypted_string_addr_rel,16) + d1_off]
    data_2 = data_encoded_extracted_2[int(key_data_addr_rel,16):int(key_data_addr_rel,16) + d2_off]

    # Finally the string table is decrypted.
    decoded_data = decrypter(data_1, data_2)

    # The decrypted string table is printed.
    # print_string_table(make_string_table(decoded_data))

    # One string is extracted from the string table. You haven't seen the associated malware
    # code yet, but this example number (1486) came from the malware code. Further information on it in next pages.
    item, result = string_decrypter_search(data_1, data_2, str_offset)
    return ("string[%d]: %s" % (item, result))

def comment_string_offset(arg_encrypted_string_addr, arg_key_data_addr, arg_str_offset):

    str_function = idc.get_name_ea_simple(arg_str_offset)
    print("\n\n")
    for k in idautils.CodeRefsTo(str_function,0):
        p = idc.prev_head(k)
        my = idc.print_insn_mnem(p)
        if my == 'mov':
            if idc.get_operand_type(p,1) == 5:
                str_off_1 = int(idc.print_operand(p, 1)[:-1], 16)
                local_result_1 = string_decrypter(arg_encrypted_string_addr, arg_key_data_addr, str_off_1)
                final_result_1 = (local_result_1[local_result_1.find(' '):]).strip()
                idc.set_cmt(k, final_result_1, 0)
            else:
                j = idc.prev_head(p)
                if my2 == 'mov':
                    if idc.get_operand_type(j,1) == 5:
                        str_off_2 = int(idc.print_operand(j, 1)[:-1], 16)
                        local_result_2 = string_decrypter(arg_encrypted_string_addr, arg_key_data_addr, str_off_2
                        final_result_2 = (local_result_2[local_result_2.find(' '):]).strip()
                        idc.set_cmt(k, final_result_2, 0)

string_slot = string_decrypter(0x1001D5A8, 0x1001E3F8, 1486)
string_slot = string_decrypter(0x1001D0B0, 0x1001D050, 708)
print("\n" + string_slot)

comment_string_offset(0x1001D5A8, 0x1001E3F8, "mw_w_decode_string_table")
comment_string_offset(0x1001D0B0, 0x1001D050, "mw_w_decode_string_table_0")
```

[Figure 67]

Few changes happened to this new version of the script:

- As it's using **IDA Python libraries**, so run it from: **File → Script Command (SHIFT+F2)**
- The script import **idaapi** and **idautils libraries** on **lines 1 and 2**. Further information about them:
  - https://www.hex-rays.com/products/ida/support/idapython_docs/ida_idaapi.html#ida_idaapi
  - https://hex-rays.com/products/ida/support/idapython_docs/idautils.html
- **Lines 57 to 64** was inserted to **handle input values** and **find the start address of the .data section** automatically.
- **Line 99** was commented out because we aren't longer interested in listing all decrypted strings.
- **Line 104** returns the a formatted and indexed string similar to "*string[1486]: kernel32.dll*" (for example).
- A new routine named **comment_string_offset** has been introduced.

About the new routine, some comments follow below:

- On the **line 108**, the **get_name_ea_simple** function gets the linear address of the function. **Reference:** https://hex-rays.com/products/ida/support/idadoc/255.shtml
- On the **line 110**, this script is interacting with each cross-reference, via **CodeRefsTo** function, to the given function. In other words, we aren't interested in learning the address of the function, but each line of code calling it. **Referece:** https://hex-rays.com/products/ida/support/idapython_docs/idautils.html#idautils.CodeRefsTo
- On the **line 111**, the **idc.prev_head** function gets the address of the previous instruction right before the calling instruction for string table's decryptor subroutine. Reference: https://hex-rays.com/products/ida/support/idadoc/265.shtml
- On the **line 112**, given the address of the previous instruction, the **idc.print_insn_mnem** function gets its mnemonics. **Reference:** https://hex-rays.com/products/ida/support/idadoc/274.shtml
- On the **line 113**, the script is only interested in **mov instructions**. Explanation: I confirmed that most (but not all as we're going to learn soon) strings offsets for the string table were associated with a particular instruction: mov ecx, <offset>. Thus, the script focuses on mov instructions.
- On the **line 114**, the **idc.get_operand_type** function filters instructions that the second part of the mnemonic is an immediate value. **Reference:** https://hex-rays.com/products/ida/support/idadoc/276.shtml

- On the **line 115**, the **int(idc.print_operand(p, 1)[:-1], 16)** nested functions gets the text representation of the operand (an immediate value) associated to instruction and convert it to decimal, which will be used as offset in the string table. **Reference:** https://hex-rays.com/products/ida/support/idadoc/275.shtml

- On the **line 116**, it calls **string_decrypter** function.

- On the **line 117**, a filtering is done to only isolate the decrypted string and nothing more.

- On the **line 118**, **the idc.set_cmt** function sets the comment at the same line of the string table decrypter function's call. No doubts, we could have set the comment at same line of the immediate value, but I thought it'd better to follow the first approach. **Reference:** https://hex-rays.com/products/ida/support/idadoc/204.shtml

- **Lines 119 to 127** contain the second part of the algorithm. I've verified that, sometimes, the string offset loading (**mov eax, <string offset>**) wouldn't happen at the instruction right before the string table decrypter subroutine call, but two instructions before. In few word, the script tests whether the instruction before the string decrypter function call has the format "**mov <reg>, <immediate value**". If it doesn't, so it goes up one level and test the second previous.

- **On lines 129 and 130** the same routine (**string_decrypter**) is called twice, but with arguments different, from distinguished string tables, and string offsets also different. In this case, only the string associated to that offset is going to be shown on the output.

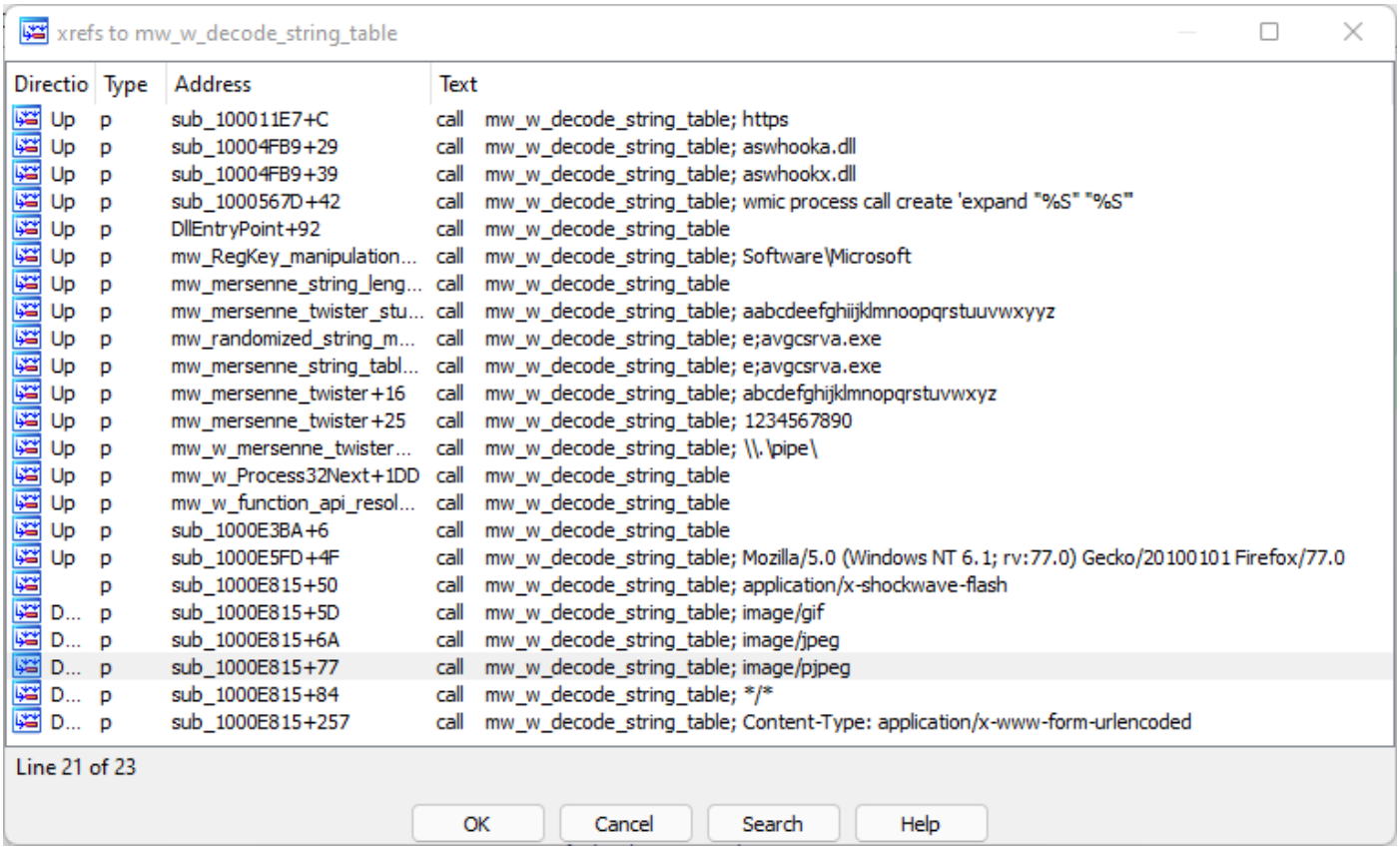
- **Lines 133 and 134** performs comment markup on the code. Please, pay attention to the fact that we need to provide the name of the wrapper subroutine used to call the string table decrypter subroutine.

How can we confirm it worked? We can check the code or check cross-references associated with the wrap of the string table decryptor routine:

```
text:1000E865                call    mw_w_decode_string_table ; application/x-shockwave-flash
text:1000E86A                mov     ecx, 0DA5h
text:1000E86F                mov     [ebp+var_3C], eax
text:1000E872                call    mw_w_decode_string_table ; image/gif
text:1000E877                mov     ecx, 0B97h
text:1000E87C                mov     [ebp+var_38], eax
text:1000E87F                call    mw_w_decode_string_table ; image/jpeg
text:1000E884                mov     ecx, 1CFh
text:1000E889                mov     [ebp+var_34], eax
text:1000E88C                call    mw_w_decode_string_table ; image/pjpeg
text:1000E891                mov     ecx, 3E4h
text:1000E896                mov     [ebp+var_30], eax
text:1000E899                call    mw_w_decode_string_table ; */*
text:1000E89E                and     [ebp+var_28], ebx
```

[Figure 68]

As you're can confirm, decrypted strings from the strings' table was added as a comment. To confirm them using **cross-references** it quite simple:



[Figure 69]

Unfortunately, not all strings table decrypter function had an instruction with an immediate value working as string table offset close to them. Of course, the script can be improved a lot.

I could have followed the same approach to **mark up comments on the decompiler's side**, but I think it wouldn't bring a major value to our analysis. Just in case you want to do, the base script is the following one:

```
def make_decompiler_comments(addr, comment):
    c_function = idaapi.decompile(addr)
    treeloc_struct = idaapi.treeloc_t()
    treeloc_struct.ea = addr
    treeloc_struct.itp = idaapi.ITP_SEMI
    if c_function:
        c_function.set_user_cmt(treeloc_struct, comment)
        c_function.save_user_cmts()
```

Few words and references that might help you:

- **Decompile(ea, hf=None, flags=0)** and **decompile_func(*args)** functions. **Reference**: https://www.hex-rays.com/products/ida/support/idapython_docs/ida_hexrays.html#ida_hexrays.decompile

- **c_function** is a **reference** to a decompiled function (the result) represented as a **cfunc_t structure**, which makes part of a **c_tree representation**. **References**:

- https://hex-rays.com/products/decompiler/manual/sdk/structcfunc_t.shtml
- https://hex-rays.com/products/decompiler/manual/sdk/hexrays_8hpp_source.shtml

```
6846 // Tags to find this location quickly: #cfunc_t #func_t
6847 //-------------------------------------------------------------------------
6848 /// Decompiled function. Decompilation result is kept here.
6849 struct cfunc_t
6850 {
6851   ea_t entry_ea;               ///< function entry address
6852   mba_t *mba;                  ///< underlying microcode
6853   cinsn_t body;                ///< function body, must be a block
6854   intvec_t &argidx;            ///< list of arguments (indexes into vars)
6855   ctree_maturity_t maturity; ///< maturity level
6856   // The following maps must be accessed using helper functions.
6857   // Example: for user_labels_t, see functions starting with "user_labels_"
6858   user_labels_t *user_labels;///< user-defined labels.
6859   user_cmts_t *user_cmts;     ///< user-defined comments.
6860   user_numforms_t *numforms; ///< user-defined number formats.
6861   user_iflags_t *user_iflags;///< user-defined item flags \ref CIT_
6862   user_unions_t *user_unions;///< user-defined union field selections.
```

[Figure 70]

- **Ctree location structure** is used to denote comments. **References**:

  - https://hex-rays.com/products/decompiler/manual/sdk/structtreeloc_t.shtml
  - https://hex-rays.com/products/decompiler/manual/sdk/hexrays_8hpp_source.shtml#l05902

```
5902 struct treeloc_t
5903 {
5904   HEXRAYS_MEMORY_ALLOCATION_FUNCS()
5905   ea_t ea;
5906   item_preciser_t itp;
5907   bool operator < (const treeloc_t &r) const
5908   {
5909     return ea < r.ea
5910         || (ea == r.ea && itp < r.itp);
5911   }
5912   bool operator == (const treeloc_t &r) const
5913   {
5914     return ea == r.ea && itp == r.itp;
5915   }
5916 };
```

[Figure 71]

- **ITP_SEMI** value is a member of **item_preciser_t enumeration** that is used to assign comments to ctree items. **References:**

  - https://www.hex-rays.com/products/decompiler/manual/sdk/hexrays_8hpp_source.shtml
  - https://www.hex-rays.com/products/decompiler/manual/sdk/hexrays_8hpp.shtml#a219c95f85c085e6f539b8d3b96074aee
  - https://www.hex-rays.com/products/decompiler/manual/sdk/hexrays_8hpp.shtml#a219c95f85c085e6f539b8d3b96074aee

- The **set_user_cmt function** set the comment on the function determined by the C tree location structure. **Reference:** https://hex-rays.com/products/decompiler/manual/sdk/structcfunc_t.shtml#ac69c7a795de52c7858c88084dae91fa6

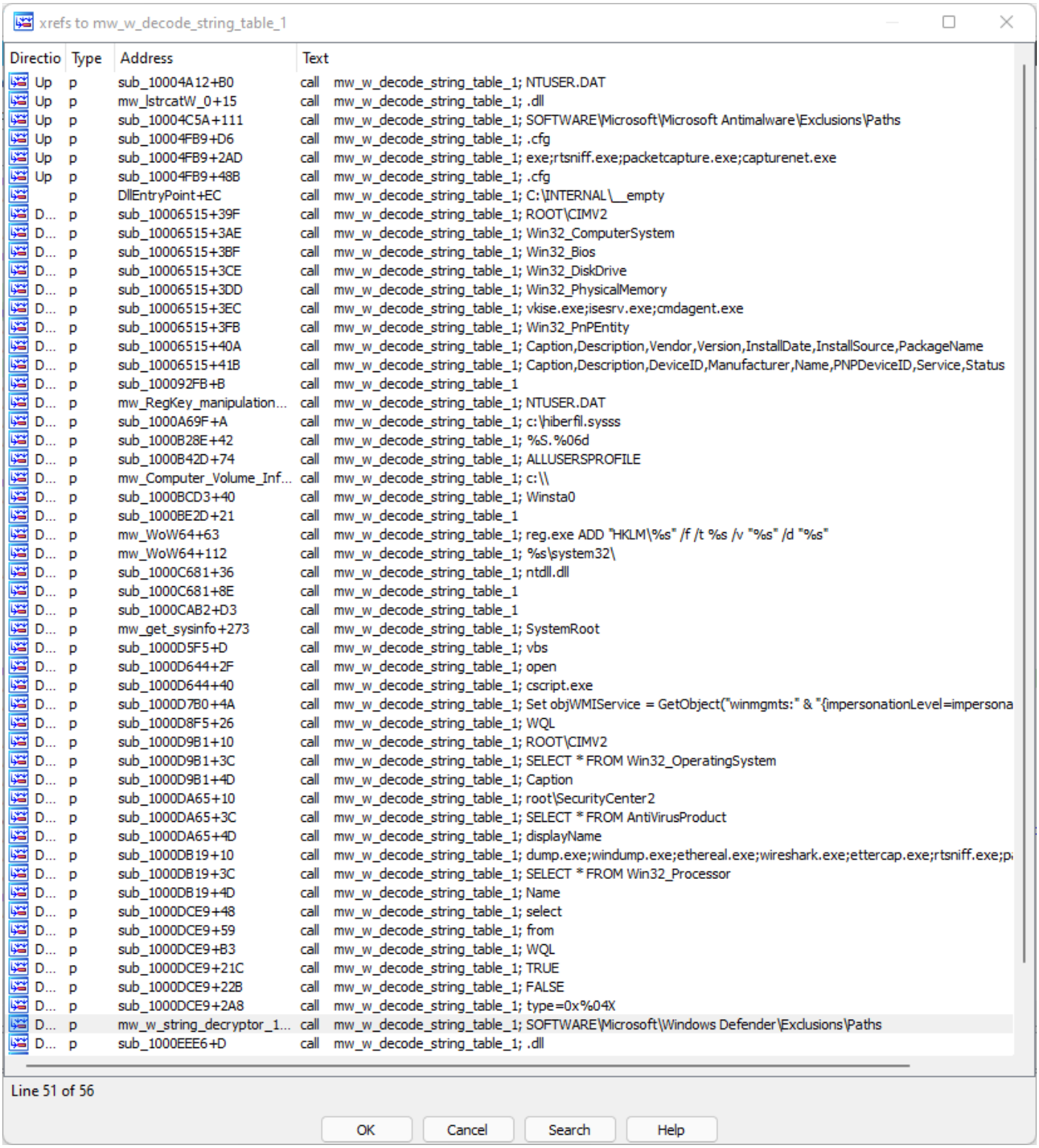
- The **save_user_cmts function** saves comments into the database. **Reference**: https://hex-rays.com/products/decompiler/manual/sdk/structcfunc_t.shtml#a5750da6d0b4a3dfc9e606ecbe565ee65

Returning to the **entry point (CTRL+E)** and going down to the code, you can confirm that **sub_1000978D** subroutine is calling **sub_100086E2**, which is essentially the same decrypter that we saw previously. However, **when I tried using the same IDA Python script from Figures 66 and 67, many cross-references calling the wrapper function weren't commented** and soon I found the reason: **most of string offsets were being passed as argument by using a 'push' instruction and not a 'mov' instruction**. Therefore, I adapted the function **comment_string_offset function** (**Figure 67 – line 106**) to **check the existence of both mnemonics** as shown below:

```
106 def comment_string_offset(arg_encrypted_string_addr, arg_key_data_addr, arg_str_offset):
107
108     str_function = idc.get_name_ea_simple(arg_str_offset)
109     print("\n\n")
110     for k in idautils.CodeRefsTo(str_function,0):
111         p = idc.prev_head(k)
112         my = idc.print_insn_mnem(p)
113         if my in ('mov','push'):
114             if my == 'mov':
115                 if idc.get_operand_type(p,1) == 5:
116                     str_off_1 = int(idc.print_operand(p, 1)[:-1], 16)
117                     local_result_1 = string_decrypter(arg_encrypted_string_addr, arg_key_data_addr, str_off_1)
118                     final_result_1 = (local_result_1[local_result_1.find(' '):]).strip()
119                     idc.set_cmt(k, final_result_1, 0)
120             if my == 'push':
121                 if idc.get_operand_type(p,0) == 5:
122                     str_off_1 = int(idc.print_operand(p, 0)[:-1], 16)
123                     local_result_1 = string_decrypter(arg_encrypted_string_addr, arg_key_data_addr, str_off_1)
124                     final_result_1 = (local_result_1[local_result_1.find(' '):]).strip()
125                     idc.set_cmt(k, final_result_1, 0)
126             else:
127                 j = idc.prev_head(p)
128                 my2 = idc.print_insn_mnem(j)
129                 if my2 in ('mov','push'):
130                     if my2 == 'mov':
131                         if idc.get_operand_type(j,1) == 5:
132                             str_off_2 = int(idc.print_operand(j, 1)[:-1], 16)
133                             local_result_2 = string_decrypter(arg_encrypted_string_addr, arg_key_data_addr, str_off_2)
134                             final_result_2 = (local_result_2[local_result_2.find(' '):]).strip()
135                             idc.set_cmt(k, final_result_2, 0)
136                     if my2 == 'push':
137                         if idc.get_operand_type(j,0) == 5:
138                             str_off_2 = int(idc.print_operand(j, 0)[:-1], 16)
139                             local_result_2 = string_decrypter(arg_encrypted_string_addr, arg_key_data_addr, str_off_2)
140                             final_result_2 = (local_result_2[local_result_2.find(' '):]).strip()
141                             idc.set_cmt(k, final_result_2, 0)
```

[Figure 72]

As you can confirm, changes were small and the only observation is that on **lines 121 and 137** we're interested in getting the **first operand because 'push' instruction has only one operand**. After these changes, we've got much better results as shown on the next page:



[Figure 73]

We're able to notice that, at same way, it isn't possible to get all calls commented because some of them **either are resolved dynamically or the string offset is found many instructions before the calling the function** and trying to catch them might cause side effects without any real gain for our task.

There're few **additional tips** that I'd like to provide with readers before continuing the reversing session:

- If you're renaming any Windows API's parameters, **my suggestion is to try to use the same parameter's name that you find on MSDN**. At first time, it might sounds like a huge and time consuming task, but you'll learn that it's worth and save significant time while trying to figure out what's going on.

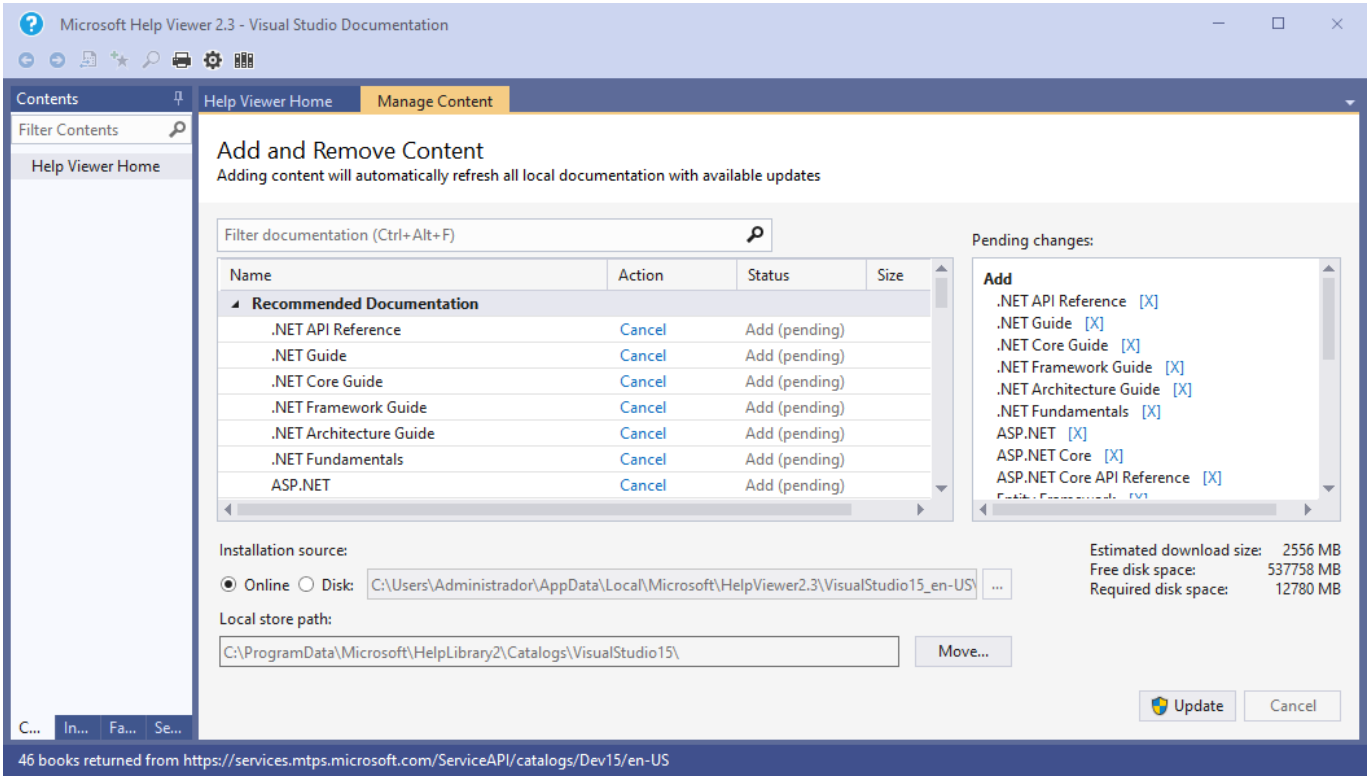
- During your reversing session, you should consider to save your work in regular times by using the **CTRL+W hotkey** to avoid losing your mark up.

- As you've noticed, we have to constantly googling for APIs definitions, parameters and lots of constants, and everything is done using a browser. Nonetheless, according to my experience, you might consider to use the **MSDN offline version** and, as I like programming so much, my suggestion of environment is:

    - **Visual Studio:** https://visualstudio.microsoft.com/

    - **Windows SDK:** you can get it installed with Visual Studio or apart from it: https://developer.microsoft.com/en-us/windows/downloads/windows-sdk/

    - **Install the MSDN offline:** To get it installed, install the Help Viewer from Visual Studio installer by going to **Individual Components tab** and marking **Help Viewer**. Afterwards, open the **Visual Studio** and go to **Help → Add and Remove Help Content**. Mark **"Programming reference for Windows API"** under Windows, remove everything else you don't want (depends your experience), click on **Update** and wait some minutes.



[Figure 74]

Proceeding with our analysis, we returned once more to the entry point (**CTRL+E**) and pay attention to the call to **CreateThread( )** on **line 45**:

```
38      if ( GetFileAttributesW((LPCWSTR)fdwReason) != -1 )
39      {
40        mw_w_string_length_0((_BYTE **)&fdwReason);
41        return 0;
42      }
43      mw_w_string_length_0((_BYTE **)&fdwReason);
44      v11 = 0;
45      mw_var_CreateThread = ptr_struct_iat_kernel32->ptr_CreateThread(0, 0, mw_thread_function, 0, 0, &v11);
46      if ( !mw_var_CreateThread )
47        return 0;
48    }
49  }
50  else if ( !fdwReason )
51  {
52    SetLastError(ERROR_BUSY);
53    return 0;
54  }
55  return 1;
56 }
```

[Figure 75]

The **CreateThread function** has, as you should already know, the third parameter as a pointer to the function to be executed which is **mw_threat_function**. Going inside of it, the function is the same from **Figure 58**, but that created structure was renamed from **struct _allocated_memory_1** to **struct struct_sysinfo** and renamed **ptr_allocated_memory_1** to **ptr_struct_sysinfo**, as shown below :

```
1 int __userpurge mw_thread_function@<eax>(void *a1@<edi>, int a2)
2 {
3   struct _OSVERSIONINFOA *ptr_struct_sysinfo; // eax
4
5   mw_iat_construction();
6   ptr_struct_sysinfo = (struct _OSVERSIONINFOA *)mw_get_sysinfo((int)a1);
7   ptr_struct_sysinfo_ref = ptr_struct_sysinfo;
8   if ( !ptr_struct_sysinfo )
9     return 1;
10  ptr_struct_sysinfo[1].dwBuildNumber = 1;
11  mw_VirtualProtect(*(_DWORD *)&ptr_struct_sysinfo_ref[3].szCSDVersion[84]);
12  if ( (*(_DWORD *)&ptr_struct_sysinfo_ref[42].szCSDVersion[60] & 0x10000) != 0 )
13  {
14    ptr_struct_sysinfo_ref[1].dwPlatformId = 1;
15  }
16  else
17  {
18    if ( mw_CreateEvent_ResumeThread(*(_DWORD *)&ptr_struct_sysinfo_ref[3].szCSDVersion[84], a1) )
19    {
20      if ( *(_DWORD *)&ptr_struct_sysinfo_ref[3].szCSDVersion[68] != 3 )
21        return 0;
22 LABEL_10:
23      mw_StartServiceCtrlDispatcherA();
24      return 0;
25    }
26    ptr_struct_sysinfo_ref[1].dwPlatformId = 1;
27  }
28  if ( *(_DWORD *)&ptr_struct_sysinfo_ref[3].szCSDVersion[68] == 3 )
29    goto LABEL_10;
30  sub_10005DDC((int)a1);
31  return 0;
32 }
```

[Figure 76]

It’s likely you’ve noticed few points of interest in this **Figure 76** such as:

- **on line 3**, the code tells us that **ptr_struct_sysinfo** (formerly **ptr_allocated_memory_1)** is a pointer to a **_OSVERSIONINFOA structure.**

- **On line 6**, this pointer receives the return of function **mw_get_sysinfo (sub_1000D1C9**). If we examine this function, on its **line 33** is called the wrapper to **HeapAlloc** and **it allocates 6852 bytes**. The return of this operation, which is a pointer, is saved into a local variable (here named **ptr_struct_sysinfo**). At end of the function this pointer is returned, so it should be a pointer related to **_OSVERSIONINFOA** or a similar structure.

- **on line 11**, we have something not usual so far that's the notation **mw_VirtualProtect(*(_DWORD *)&ptr_struct_sysinfo_ref[3].szCSDVersion[84]);**. The index 3 suggests there are many structures (this particular one is the number 4) and, because the field's name, it seems to be of type **_OSVERSIONINFOA** or containing same size. A similar effect occurs on **line 12**.

- If you get references (**'X' hotkey**) to **ptr_struct_sysinfo_ref**, you'll find there're **158 cross-references**.

Therefore, we can concludes that **ptr_struct_sysinfo** is actually pointing to a structure of structures and, likely, all these structures might be **_OSVERSIONINFOA** or similar size. As you already know, **_OSVERSIONINFOA** is a structure defined in **winnt.h** that contains operating system version information such as **major and minor version number, build number , platform ID, of the operating system** and **service pack (szCSDVersion)**. Its composition is shown below:

```
typedef struct _OSVERSIONINFOA {
  DWORD dwOSVersionInfoSize;
  DWORD dwMajorVersion;
  DWORD dwMinorVersion;
  DWORD dwBuildNumber;
  DWORD dwPlatformId;
  CHAR  szCSDVersion[128];
} OSVERSIONINFOA, *POSVERSIONINFOA, *LPOSVERSIONINFOA;
```

[Figure 77]

The information above was extracted from MSDN, and you can get it from Internet (https://docs.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-osversioninfoa) or locally, as I've already explained, so another hint about accessing the **local (offline) version of the** MSDN follows: if you want to call **Help Viewer** out of **Visual Studio** environment, so you should create a shortcut setting as target the following command: ***"C:\Program Files (x86)\Microsoft Help Viewer\v2.3\HlpViewer.exe" /catalogName VisualStudio15 /launchingApp Microsoft,VisualStudio,15***. Of course, this command line depends on your **Help Viewer** version **and Visual Studio** version.

The structure above has **148 bytes** and, based on considerations above, it confirms what I had already mentioned previously (**page 47**) that the **struct_sysinfo** should be nested structure (structure composed by structures) and not a simple one.

As the **_OSVERSIONINFOA structure** has **148 bytes** and **mw_get_sysinfo** subroutine (originally named sub_1000D1C9) has a call "**mw_HeapAlloc (6852)**" (check Figure 59, page 47, line 33), so we need to do a simple math: **6852 / 148 = 46**. Thus, **struct_sysinfo** is composed by **46 generic and similar structures**.

Here's decision point and it's relevant to underscore one aspect: we're trying to understand pieces of the malware and improve the IDA pseudo code based on our analysis, but there isn't a "right or wrong" here and, eventually, that's an interesting experience to comment about because some people believe we should be always exact and, sometimes, it isn't possible because there're necessary context to improve the interpretation. For example, if I believe that a field has a goal and, after some analysis and learned contexts, I realize that field's name or even its type wasn't correct, so I return, fix it and move forward. It's ways hard reversing code and try to make it more "readable".

On this case, we might take three different approaches:

a. Creating **generic structure of structures** and renaming fields along the analysis are usually the best bet because provides flexibility with us. Personally, I always try to follow this approach whether the situation and context allow to do.

b. We might use the own **_OSVERSIONINFOA structure** and its default names. A strange fact is that **szCSDVersion field** should hold a string related to applied service packs, but it seems being used and sliced for any other goal, and the final meaning it's a bit weird and hard to follow, so I don't like it, though the code is already using _OSVERSIONINFOA structure.

c. A third possibility would keeping the a huge structure of **DWORDs (6852 / 4 = 1713**) and work with them without facing any problem.

I don't believe the malware's author is using a structure of _OSVERSIONINFOA structures, but a similar structure with the same size. Anyway, this won't do a big difference on our static analysis (believe me). I could use the **approach C**, but I'll choose the **approach A** because there seem be structure of structures in our code. Additionally, I'm guessing assuming that each member structure has the same size of _OSVERSIONINFOA structure and, of course, there isn't any certainty about it and my assumption is based on **Figure 76** because, on the top of the code, there's a reference to _OSVERSIONINFOA structure. Therefore, as the **_OSVERSIONINFOA has 148 bytes** then our generic structure will also have 148 bytes and, based on this information, we know it'll have **148 / 4 = 37** dword fields.

According to explaining so far, we have to create two structures:

- **SYS_INFO:** composed by **37 dword fields**.
- **SYS_INFO_ALL:** composed by **46 SYS_INFO structures**.

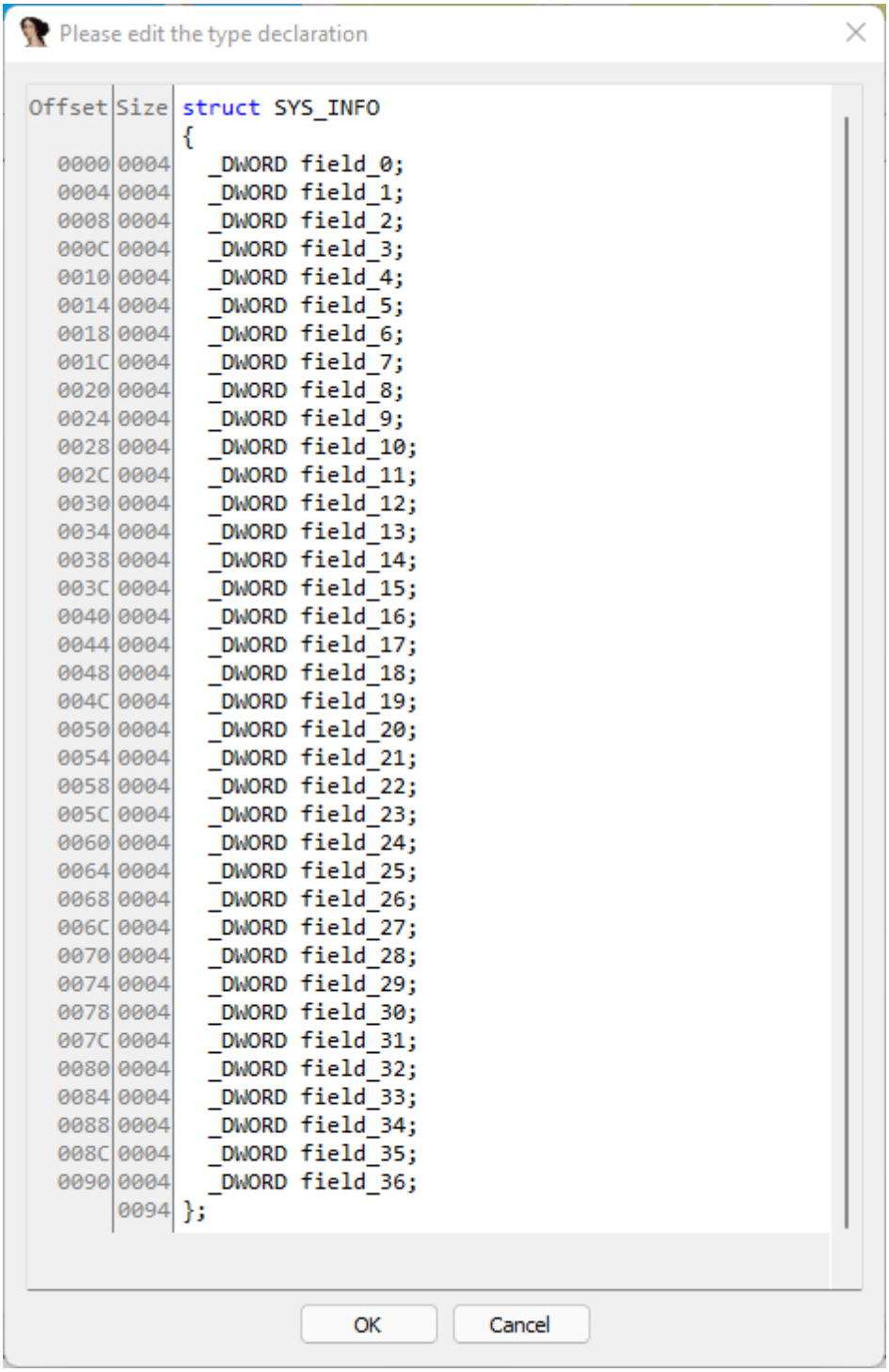
To manage this context, we need to do few steps to get a good pseudo code. First, remove the **struct_sysinfo** on **page 48** because, as we've learned, it isn't a simple structure composed only by dwords fields.

To accomplish the task:

- Go to **View → Open Subviews → Structures (SHIFT+F9)**
- Find the **struct_sysinfo**, right click on it and choose **"Delete structure type…"** and confirm the operation.

Next task is **creating** a new structure (named **SYS_INFO**), which is composed by 37 dwords. This time, we are going to use the second approach mentioned on the top of **page 48.** Thus, to create this structure of structures, execute the following steps:

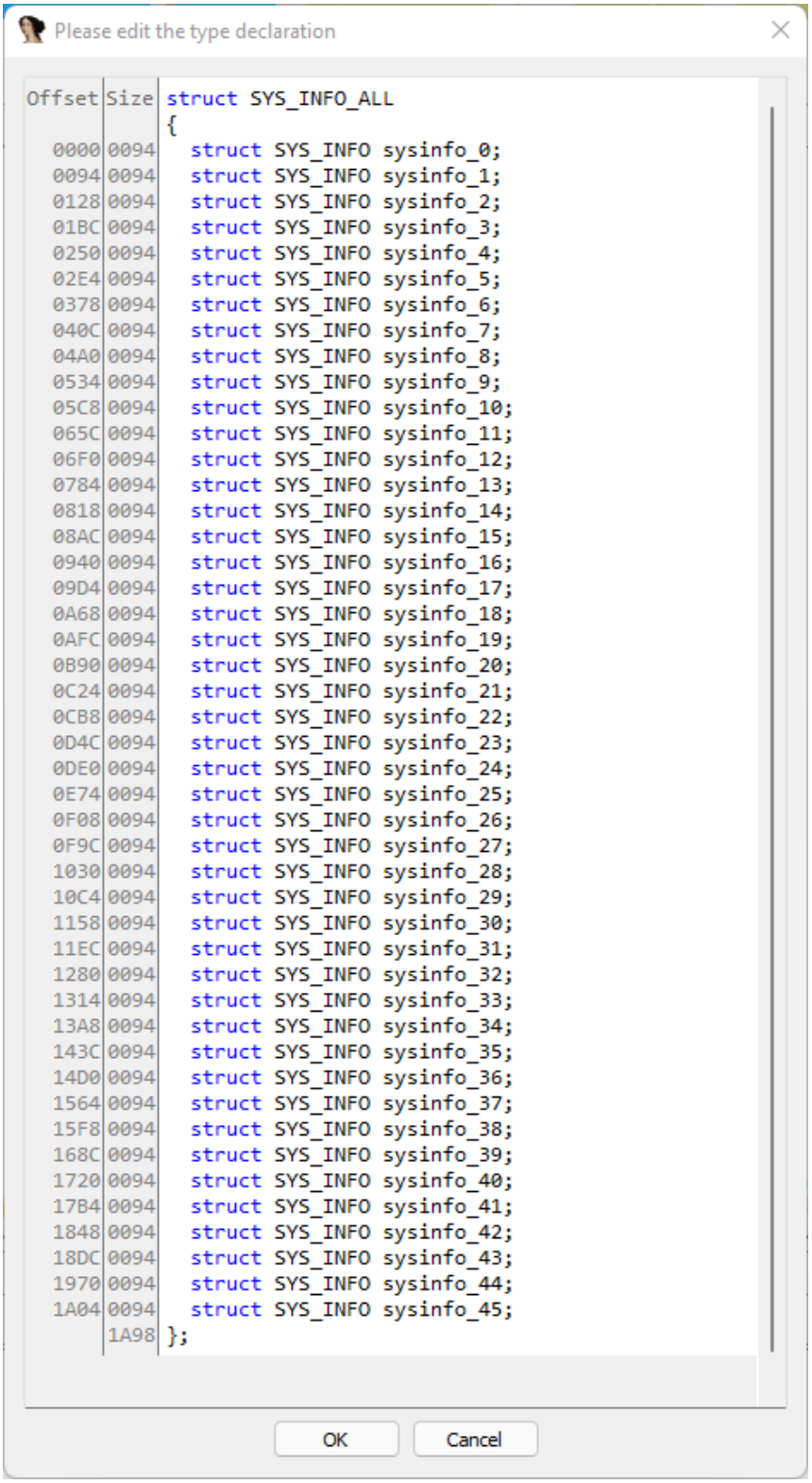
- go to **View → Open Subviews → Local Types (SHIFT+F1)**
- press **INSERT key** from your keyboard.
- Now fill it up with the following content and take care with the syntax.
- **Hint:** write a two-line python script to generate all fields automatically.



[Figure 78]

Basically, we need to repeat steps from page 64, but this time creating a new structure (SYS_INFO_ALL) composed by **46 SYS_INFO structures**. Once again, use Python for help you:

- go to **View → Open Subviews → Local Types (SHIFT+F1)**
- press **INSERT key** from your keyboard.
- Now fill it up with the following content and take care with the syntax.



[Figure 79]

Once the new type was created, so you should **import** both into **Structures tab**. Right click on the **SYS_INFO local type** and pick up **"Synchronize to idb"**. Confirm the import. Now **SYS_INFO** local type has been imported as a structure into the **Structure window**. Do the same with **SYS_INFO_ALL.**

On the next step, you should **apply** the new structure as a type for few variables and, as we've done similar steps previously, so no news here. Based on **Figure 76**, double click on **mw_get_sysinfo (sub_1000D1C9)** and, as you already examined previously (**page 48, picture 63**) there's a call to wrapper of HeapAlloc on **line 33** as shown below:

- ptr_struct_sysinfo = (#198 *)mw_HeapAlloc(6852u);

Click on **ptr_struct_sysinfo** variable, press **'Y' hotkey** and change its signature from "int *ptr_struct_sysinfo" to "**SYS_INFO_ALL** *ptr_struct_sysinfo". It there was a signature such as "#<number *ptr_struct_sysinfo, it's due the fact of you have deleted the previous simple structure. There isn't any problem and make the change normally. Repeat the same procedure for **ptr_struct_sysinfo_ref** variable (**line 34**). Finally, **press F5** and check how our code is better and, this time, correct:

```
32  var_hinstDLL_ref = var_hinstDLL;
33  ptr_struct_sysinfo = (SYS_INFO_ALL *)mw_HeapAlloc(6852u);
34  ptr_struct_sysinfo_1 = ptr_struct_sysinfo;
35  if ( ptr_struct_sysinfo )
36  {
37    ptr_struct_sysinfo->sysinfo_38.seed = GetCurrentProcessId();
38    time_miliseconds = ((int (__stdcall *)(int))ptr_struct_iat_kernel32->ptr_GetTickCount64)(a1);
39    mersenne_twister_initialization(
40      time_miliseconds + ptr_struct_sysinfo_1->sysinfo_38.seed,
41      &ptr_struct_sysinfo_1->sysinfo_10.state);
42    if ( GetModuleFileNameW(0, (LPWSTR)&ptr_struct_sysinfo_1->sysinfo_38.lpFilename, 0x105u) )
43      ptr_struct_sysinfo_1->sysinfo_42.field_3 = mw_m_cmp((__int16 *)&ptr_struct_sysinfo_1->sysinfo_38.lpFilename);
44    CurrentProcess = GetCurrentProcess();
45    ptr_SID = (void **)mw_w_GetTokenInformation(CurrentProcess);
46    ptr_struct_sysinfo_1->sysinfo_1.ptr_buffer_TokenInformation = ptr_SID;
47    if ( mw_AllocateAndInitializeSid(*ptr_SID) )
48      ptr_struct_sysinfo_1->sysinfo_3.token_handle = 3;
49    else
50      ptr_struct_sysinfo_1->sysinfo_3.token_handle = ((int)mw_AllocateAndInitializeSid_0() > 0) + 1;
51    ptr_struct_sysinfo_1->sysinfo_3.network_join_status = mw_NetGetJoinInformation(&ptr_struct_sysinfo_1->sysinfo_3.join_status);
52    ptr_struct_sysinfo_1->sysinfo_3.field_24 = mw_NetGetDCName();
53    ptr_struct_sysinfo_1->sysinfo_3.hDLL = var_hinstDLL_ref;
54    cchName = 128;
55    Wow64Process = 256;
56    if ( !ptr_struct_iat_advapi32->ptr_LookupAccountSidW(
57            0,
58            *(PSID *)ptr_struct_sysinfo_1->sysinfo_1.ptr_buffer_TokenInformation,
59            (LPWSTR)&ptr_struct_sysinfo_1->sysinfo_1.state,
60            (LPDWORD)&cchName,
61            (LPWSTR)ReferencedDomainName,
62            (LPDWORD)&Wow64Process,
63            (PSID_NAME_USE)ptr_account_type) )
64      GetLastError();
65    var_SystemMetrics = ptr_struct_iat_user32->ptr_GetSystemMetrics(4096);
66    mod_handle = (HMODULE)ptr_struct_sysinfo_1->sysinfo_3.hDLL;
67    ptr_struct_sysinfo_1->sysinfo_42.processor_arch_info = var_SystemMetrics > 0;
68    GetModuleFileNameW(mod_handle, (LPWSTR)&ptr_struct_sysinfo_1->sysinfo_3.lpFilename, 0x105u);
69    GetLastError();
70    ptr_struct_sysinfo_1->sysinfo_7.field_10 = mw_m_cmp((__int16 *)&ptr_struct_sysinfo_1->sysinfo_3.lpFilename);
71    crc32_value = mw_Computer_Volume_Information((LPCWSTR)&ptr_struct_sysinfo_1->sysinfo_1.state, ReferencedDomainName);
72    ptr_struct_sysinfo_1->sysinfo_1.crc32_value = crc32_value;
73    mw_mersenne_string(crc32_value, (int)&ptr_struct_sysinfo_1->sysinfo_1.ptr_buffer);
```

[Figure 80]

The reader can wonder reasons for I having took the wrong way on **page 47** (it wasn't a simple structure, but a structure of structures) and afterwards I have fixed it. Simple: **I wanted showing you how real analysis actually happen.** Yes, I already knew the structure was a bit more complex (I said it on page 47), but taking a wrong interpretation I could explain a new technique to create automatically a new structure

(and we'll use this technique several times in this article or next ones) and to explain that, as a reverse engineering, you can easily re-interpret an information, fix it and everything OK.

Let's move forward to next pieces of code. The code on **Figure 80** is a good start point and on **line 39** you see the function **mersenne_twister_initialization**, which is also pointed by the **Capa Explorer** as being Mersenne related.

Mersenne Twister is a well-known **PRNG (Pseudo Random Number Generator**) , but it presents some changes when compared to a default one.

In a **classical PRNG**, a seed is provided to initialize a generator into an initial state (**state 0**). This initial state (**state 0**) is used as input of a one-way **function X** to generate the first random number (**random 0**). To generate a second random number, the previous state (**state 0**) is transformed into a new state (**state 1**) by using a second **function Y**, which is used as input of the **one-way function X** to generate a new random number (**random 1**). If you want a third random number, the previous state (**state 1**) is transformed into a new state (**state 2**) by using the the same **function Y.** This new state (**state 2**) is used as input of the same **function X** to generate a new random number (**random 2**). And process keep going on.

The **Mersenne Twister** has a working slightly different from the **default PRNG**. One of first changes is that the initial state (**state 0**) is not used as input of a first random number (**random 0** at the prior paragraph).

The Mersenne procedure starts providing a seed, which is used as input by an initialization function and the initial state (**state 0**) is generated. A **Twist function** is applied on this initial state (**state 0**) and the **state 1** is generated. To generate the first random number, a **function Y** is applied on the **state 1**. Is the procedure equal to a default PRNG? Almost. The **function Y**, which is used to generate the random number, is reversible (not a one-way function) and this function is **able to generate 624 random numbers with the same state as input**. When all random numbers was generated using the **state 0** as input, so a new state can be created. Appling the **Twister function** on the **state 0**, the **state 1** is created. The **function Y** is applied on the **state 1** and new random numbers (up to 624) can be created. The process keep going on.

In malware analysis we'll see **Mersenne Twister** algorithm being used in many malware samples. This generated random number may be used in different scenarios as **string generation, C2 communication,** and so on. The general algorithm is composed by few parts as initialization, random number generator and the **twist function** that is responsible for transforming the previous state into the next one.

The **mersenne_twister_initialization** function called on **line 39** / **Figure 80** is shown below:

```
1  int __cdecl mersenne_twister_initialization(int seed, _DWORD *state)
2  {
3    int i; // ecx
4    int mersenne_state; // eax
5
6    *state = seed;
7    state[624] = 1;
8    do
9    {
10     i = state[624];
11     mersenne_state = i + 1812433253 * (state[i - 1] ^ (state[i - 1] >> 30));
12     state[i] = mersenne_state;
13     ++state[624];
14   }
15   while ( (int)state[624] < 624 );
16   return mersenne_state;
17 }
```

[Figure 81]

Constants helps us to detect and identify this code on **Figure 81** as being related to **Mersenne Twister**. Returning to code shown on **Figure 80** (that was truncated because the routine is longer), there're tons of APIs being called directly or through a wrapper function such as **GetCurrentProcess( ), GetTokenInformation( ), AllocateAndInitializeSid( ), NetGetJoinInformation( ), NetGetDCName( ), LookupAccountSidW( ), GetVolumeInformationW( ), GetSystemEnvironment( ), GetComputerName( )**, and so on. The remaining part of subroutine shown in **Figure 80** follows below:

```
74    if ( ptr_struct_sysinfo_1 != (SYS_INFO_ALL *)-176 )
75    {
76      num_bytes_written = MultiByteToWideChar(
77                            0,
78                            0,
79                            (LPCCH)&ptr_struct_sysinfo_1->sysinfo_1.ptr_buffer,
80                            -1,
81                            (LPWSTR)&ptr_struct_sysinfo_1->sysinfo_1.lpWideCharStr,
82                            32);
83      if ( num_bytes_written > 0 )
84        *((_WORD *)&ptr_struct_sysinfo_1->sysinfo_1.lpWideCharStr + num_bytes_written) = 0;
85    }
86    mw_search_char((char *)&ptr_struct_sysinfo_1->sysinfo_3.lpFilename, &ptr_struct_sysinfo_1->sysinfo_7.field_11);
87    str_length_char = mw_str_length_char(&ptr_struct_sysinfo_1->sysinfo_1.ptr_buffer);
88    seed = mw_crc32(str_length_char, (int)&ptr_struct_sysinfo_1->sysinfo_1.ptr_buffer, 0);
89    mw_mersenne_twister(seed, (int)&ptr_struct_sysinfo_1->sysinfo_27.state);
90    current_process_id = GetCurrentProcess();
91    ptr_struct_sysinfo_1->sysinfo_27.state = mw_GetSidSubAuthority(current_process_id);
92    memset(ptr_struct_sysinfo_1, 0, 156u);
93    ptr_struct_sysinfo_1->sysinfo_0.field_0 = 156;
94    GetVersionExA((LPOSVERSIONINFOA)ptr_struct_sysinfo_1);
95    ptr_struct_iat_kernel32_ref = ptr_struct_iat_kernel32;
96    var_is_Wow64Process = 0;
97    Wow64Process = 0;
98    if ( ptr_struct_iat_kernel32->ptr_IsWow64Process )
99    {
100     hProcess = GetCurrentProcess();
101     ptr_struct_iat_kernel32_ref->ptr_IsWow64Process(hProcess, &Wow64Process);
102     var_is_Wow64Process = (_BYTE *)Wow64Process;
103   }
104   ptr_struct_sysinfo_1->sysinfo_1.is_Wow64Process = var_is_Wow64Process;
105   if ( var_is_Wow64Process )
106   {
107     wProcessorArchitecture = PROCESSOR_ARCHITECTURE_AMD64;
108   }
109   else
110   {
111     GetSystemInfo(&SystemInfo);
112     wProcessorArchitecture = SystemInfo.wProcessorArchitecture;
113   }
114   LOWORD(ptr_struct_sysinfo_1->sysinfo_1.processor_arch_info) = wProcessorArchitecture;
115   GetWindowsDirectoryW((LPWSTR)&ptr_struct_sysinfo_1->sysinfo_27.lpBuffer, 0x104u);
116   SystemRoot_string = mw_w_decode_string_table_1(0xA35u);// SystemRoot
117   Wow64Process = (int)SystemRoot_string;
118   if ( !ptr_struct_iat_kernel32->ptr_GetEnvironmentVariableW(SystemRoot_string, (LPWSTR)lpBuffer, 260) )
119     ptr_struct_iat_kernel32->ptr_SetEnvironmentVariableW(
120       SystemRoot_string,
121       (LPCWSTR)&ptr_struct_sysinfo_1->sysinfo_27.lpBuffer);
122   mw_w_string_length_0((_BYTE **)&Wow64Process);
123   if ( !ptr_struct_iat_kernel32->ptr_GetEnvironmentVariableW(
124           L"USERPROFILE",
125           (LPWSTR)&ptr_struct_sysinfo_1->sysinfo_34.lpBuffer,
126           521) )
127   {
128     mw_w_str_length_wchar(
129       (wchar_t *)&ptr_struct_sysinfo_1->sysinfo_34.lpBuffer,
130       0x105u,
131       (wchar_t *)L"%s\\%s",
132       &ptr_struct_sysinfo_1->sysinfo_27.lpBuffer,
133       "TEMP");
134     ptr_struct_iat_kernel32->ptr_SetEnvironmentVariableW(
135       L"USERPROFILE",
136       (LPCWSTR)&ptr_struct_sysinfo_1->sysinfo_34.lpBuffer);
```

```
137     }
138     if ( !ptr_struct_iat_kernel32->ptr_GetEnvironmentVariableW(
139             L"TEMP",
140             (LPWSTR)((char *)&ptr_struct_sysinfo_1->sysinfo_31.lpWideCharStr + 2),
141             522) )
142       ptr_struct_iat_kernel32->ptr_SetEnvironmentVariableW(L"TEMP", (LPCWSTR)&ptr_struct_sysinfo_1->sysinfo_34.lpBuffer);
143     if ( !ptr_struct_iat_kernel32->ptr_GetEnvironmentVariableW(L"SystemDrive", (LPWSTR)lpBuffer_2, 64) )
144       ptr_struct_iat_kernel32->ptr_SetEnvironmentVariableW(L"SystemDrive", L"C:");
145     Wow64Process = 127;
146     ((void (__stdcall *)(_DWORD *))ptr_struct_iat_kernel32->ptr_GetComputerNameW)(&ptr_struct_sysinfo_1->sysinfo_44.field_11);
147     var_str_length = mw_str_length_char(&ptr_struct_sysinfo_1->sysinfo_27.state);
148     var_crc_32 = mw_crc32(var_str_length, (int)&ptr_struct_sysinfo_1->sysinfo_27.state, 0);
149     mersenne_twister_initialization(var_crc_32, state);
150     mw_w_mersenne_random(state, (int)&ptr_struct_sysinfo_1->sysinfo_42.field_4, 0x20u);
151     mw_MT_manipulation((int)&ptr_struct_sysinfo_1->sysinfo_42.field_12, 1, 20, 30, state);
152     ptr_struct_sysinfo_1->sysinfo_42.field_20 = mw_w_Process32Next();
153     return ptr_struct_sysinfo_1;
154   }
155   return ptr_struct_sysinfo;
156 }
```

[Figure 82]

Obviously, there isn't enough and space to analyze each line and function here, but let's look inside **mw_mersenne_string** on **line 73**:

```
1  unsigned int __fastcall mw_randomized_string_mersenne(int arg_ComputerVolumeInformation, int arg_buffer_ptr)
2  {
3    char *ptr_decoded_string_table; // ebx
4    unsigned int counter; // esi
5    int var_str_length_char; // eax
6    int mersenne_state[625]; // [esp+Ch] [ebp-9CCh] BYREF
7    char *ptr_decoded_string_table_ref; // [esp+9D0h] [ebp-8h] BYREF
8    unsigned int var_marsenne_random_number; // [esp+9D4h] [ebp-4h]
9
10   mersenne_twister_initialization(arg_ComputerVolumeInformation, mersenne_state);
11   ptr_decoded_string_table = mw_w_decode_string_table();
12   ptr_decoded_string_table_ref = ptr_decoded_string_table;
13   var_marsenne_random_number = mw_mersenne_random(mersenne_state, 5, 8);
14   for ( counter = 0; counter < var_marsenne_random_number; ++counter )
15   {
16     if ( counter >= 0x20 )
17       break;
18     var_str_length_char = mw_str_length_char(ptr_decoded_string_table);
19     *(_BYTE *)(counter + arg_buffer_ptr) = ptr_decoded_string_table[mw_mersenne_random(
20                                                                     mersenne_state,
21                                                                     0,
22                                                                     var_str_length_char - 1)];
23   }
24   *(_BYTE *)(counter + arg_buffer_ptr) = 0;
25   mw_ww_string_length(&ptr_decoded_string_table_ref);
26   return counter;
27 }
```

[Figure 83]

On line 10, the previously explained **mersenne_twister_initialization** subroutine is called. So after this point, we have the **mw_mersenne_random** subroutine being called on **line 13** and, the random number returned by this function is used as a maximum counter in the following loop construction. Additionally, the same **mw_mersenne_random subroutine** is called as an index of the **decoded_string_table**. Therefore, the general idea seems being provide randomized strings.

However, the part most interesting is the own **mw_mersenne_random_subroutine**. If you have any issues about the **Mersenne Twister algorithm**, so maybe the **Wikipedia reference** might be a start point in your learning: https://en.wikipedia.org/wiki/Mersenne_Twister. Anyway, the code is the following one:

```
1  int __cdecl mw_mersenne_random(_DWORD *mt_state, int a2, int a3)
2  {
3    unsigned int random_number; // esi
4
5    random_number = mw_MT_twist_and_random(mt_state) & 0xFFFFFFF;
6    mw_str_length_char("ole32.dll");
7    return a2 - (int)((double)(unsigned int)(a3 - a2 + 1) * ((double)random_number * -0.00000000372529029846191406));
8  }
```

[Figure 84]

Going inside **the mw_MT_twist_and_random subroutine**, we have:

```
1  unsigned int __cdecl mw_MT_twist_and_random(_DWORD *state)
2  {
3    int next; // edi
4    int i; // esi
5    unsigned int curr_state; // ecx
6    int random_number; // ecx
7
8    next = state[624];
9    if ( next >= 624 )
10   {
11     next = 0;
12     for ( i = 0; i < 227; ++i )
13       state[i] = state[i + 397] ^ dword_1001D02C[state[i + 1] & 1] ^ ((state[i] ^ (state[i] ^ state[i + 1]) & 0x7FFFFFFFu) >> 1);
14     do
15     {
16       state[i] = state[i - 227] ^ dword_1001D02C[state[i + 1] & 1] ^ ((state[i] ^ (state[i] ^ state[i + 1]) & 0x7FFFFFFFu) >> 1);
17       ++i;
18     }
19     while ( i < 623 );
20     state[623] = ((state[623] ^ (*state ^ state[623]) & 0x7FFFFFFFu) >> 1) ^ state[396] ^ dword_1001D02C[*(_BYTE *)state & 1];
21     state[624] = 0;
22   }
23   curr_state = state[next];
24   state[624] = next + 1;
25   random_number = ((((curr_state >> 11) ^ curr_state) & 4282013869) << 7) ^ (curr_state >> 11) ^ curr_state;
26   return ((random_number & 4294958988) << 15) ^ random_number ^ ((((random_number & 4294958988) << 15) ^ (unsigned int)random_number) >> 18);
27 }
```

[Figure 85]

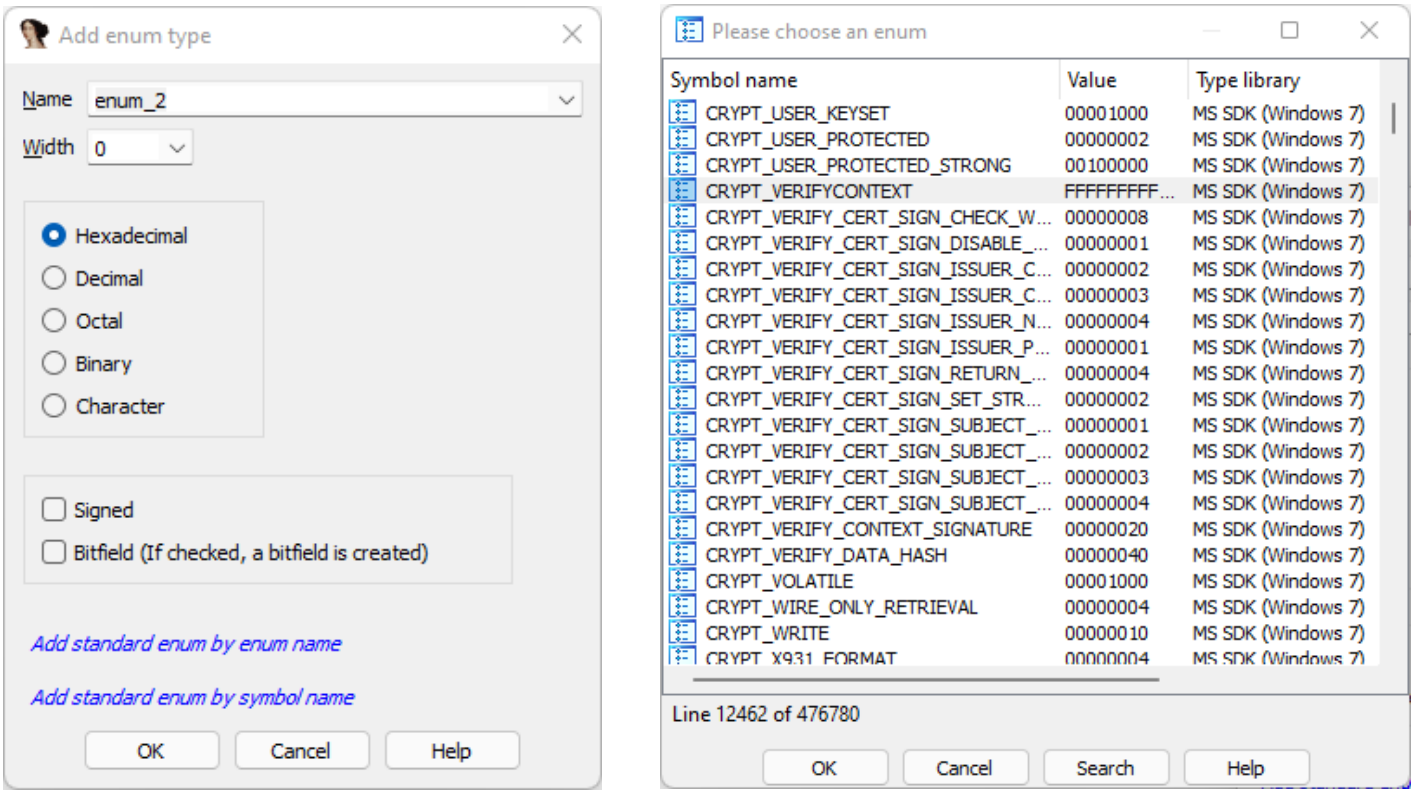
Nearly the code **up to line 22** is related to generation of the next state and code **from line 23 up to the end** it's related to providing a return of the random number.

Looking around for other attractive subroutines, **sub_10011865 subroutine** might be an interesting place to take some the advantage of including a standard enumeration because there're few **Microsoft Crypto APIs** such as:

- **CryptAcquireContextA:** acquire a handle to a ***key container*** within a given ***CSP (cryptographic service provider).***
- **CryptGenRandom**: fill a buffer with cryptographically random bytes.
- **CryptReleaseContext:** releases the handle of a cryptographic service provider (CSP) and a key container.

As expected, there is always one or more numeric constants to be converted into symbolic constants and, in subroutine, we found that **CryptAcquireContextA( )** has in its four parameter (**dwFlags**) a numeric constant (*0xF0000000*). If you read about this function on MSDN (https://docs.microsoft.com/en-us/windows/win32/api/wincrypt/nf-wincrypt-cryptacquirecontexta) you'll learn that all flags has a value with **prefix *'CRYPT_'*** and it could be interesting to add all these **dwFlags** values at once to make your reversing quicker and simpler.

Thus, to do it you should go to **Enumeration tab** (**SHIFT-F10**) and press **"INSERT" key**. Once there, choose "***Add standard enum by symbol name***" and look for any of possible values for **dwFlags**. In this case, I picked up **CRYPT_VERIFYCONTEXT** (the first one in **dwFlag** possible values) and clicked "OK";



[Figure 86]

If you check in **Enumeration tab**, so you'll find our recent added enumeration and all **CRYPT_* constants**, as shown below:

```
00000001 ; -------------------------------------------------------
00000001
00000001 ; enum MACRO_CRYPT_VERIFYCONTEXT, copyof_207, bitfield
00000001 CRYPT_EXPORTABLE   = 1
00000002 CRYPT_USER_PROTECTED   = 2
00000004 CRYPT_CREATE_SALT   = 4
00000008 CRYPT_NEWKEYSET   = 8
00000008 CRYPT_UPDATE_KEY   = 8
00000010 CRYPT_DELETEKEYSET   = 10h
00000010 CRYPT_NO_SALT     = 10h
00000010 CRYPT_RECIPIENT   = 10h
00000020 CRYPT_MACHINE_KEYSET   = 20h
00000040 CRYPT_SILENT      = 40h
00000040 CRYPT_PREGEN      = 40h
00000040 CRYPT_INITIATOR   = 40h
00000080 CRYPT_DEFAULT_CONTAINER_OPTIONAL   = 80h
00000080 CRYPT_ONLINE      = 80h
00000100 CRYPT_SF          = 100h
00000200 CRYPT_CREATE_IV   = 200h
00000400 CRYPT_KEK         = 400h
00000800 CRYPT_DATA_KEY    = 800h
00001000 CRYPT_VOLATILE    = 1000h
00002000 CRYPT_SGCKEY      = 2000h
00004000 CRYPT_ARCHIVABLE   = 4000h
00008000 CRYPT_FORCE_KEY_PROTECTION_HIGH   = 8000h
00100000 CRYPT_USER_PROTECTED_STRONG   = 100000h
F0000000 CRYPT_VERIFYCONTEXT   = 0F0000000h
```

[Figure 87]

Now you can put the mouse on the constant, press **'M' hotkey** and picked the **offered crypto-constant**:

```
14  ptr_addr = 0;
15  phProv = 0;
16  ModuleHandleA = GetModuleHandleA("advapi32.dll");
17  var_advapi_dll = ModuleHandleA;
18  if ( !ModuleHandleA )
19    return 1;
20  CryptAcquireContextA = (BOOL (__stdcall *)(HCRYPTPROV *, LPCSTR, LPCSTR, DWORD, DWORD))GetProcAddress(
21                                                                                          ModuleHandleA,
22                                                                                          "CryptAcquireContextA");
23  if ( !CryptAcquireContextA )
24    return 1;
25  CryptGenRandom = (BOOL (__stdcall *)(HCRYPTPROV, DWORD, BYTE *))GetProcAddress(var_advapi_dll, "CryptGenRandom");
26  if ( !CryptGenRandom )
27    return 1;
28  CryptReleaseContext = (BOOL (__stdcall *)(HCRYPTPROV, DWORD))GetProcAddress(var_advapi_dll, "CryptReleaseContext");
29  if ( !CryptReleaseContext )
30    return 1;
31  if ( !CryptAcquireContextA(&phProv, 0, 0, 1, CRYPT_VERIFYCONTEXT) )
32    return 1;
33  v5 = CryptGenRandom(phProv, 4, (BYTE *)pbBuffer);
34  CryptReleaseContext(phProv, 0);
35  if ( !v5 )
36    return 1;
37  for ( counter = 0; counter < 4; ++counter )
38    ptr_addr = (unsigned __int8)pbBuffer[counter] | (ptr_addr << 8);
39  *buffer_content = ptr_addr;
40  return 0;
41 }
```

[Figure 88]

By the way, if you remember, on **Figure 80 / line 51**, under **mw_get_sysinfo** (**sub_1000D1C9**), we have a wrapper to API **NetGetJoinInformation( )** (named **mw_NetGetJoinInformation**). Moving inside this subroutine, there're some conditional instructions related to the **BufferType**, which has the following possible values:

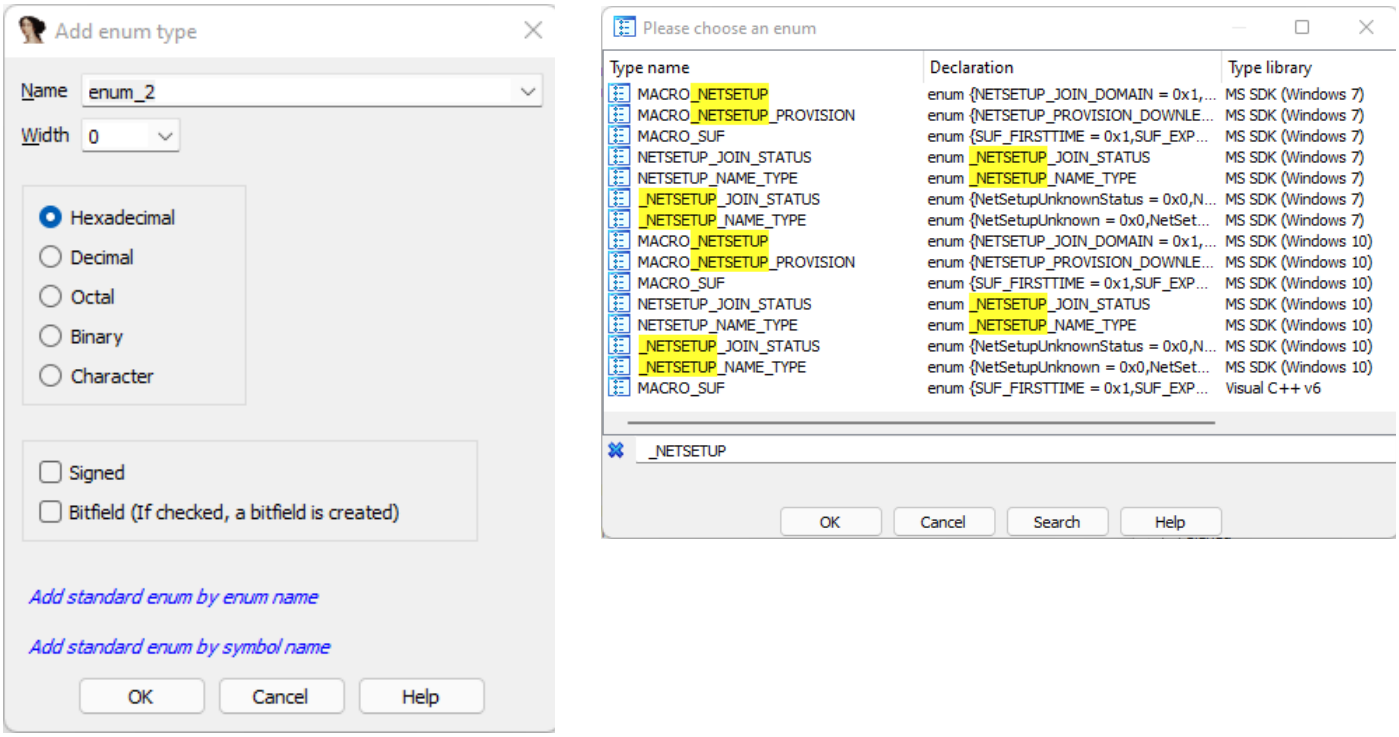
```
typedef enum _NETSETUP_JOIN_STATUS {

    NetSetupUnknownStatus = 0,
    NetSetupUnjoined,
    NetSetupWorkgroupName,
    NetSetupDomainName

} NETSETUP_JOIN_STATUS, *PNETSETUP_JOIN_STATUS;
```

[Figure 89]

The procedure is always the same, but a little detail might change:

a. Go to **Enumeration tab** (**SHIFT-F10**)
b. press **"INSERT" key**.
c. Once there, choose "***Add standard enum by enum name***".
d. Search for the enumeration **_NETSETUP_JOIN_STATUS**.
e. As we're focusing the analysis on APIs from **MS SDK Windows 7**, so pick it.
f. Click "**OK**";

After adding a standard enumeration, go to the code and apply enumeration constants using '**M' hotkey**:



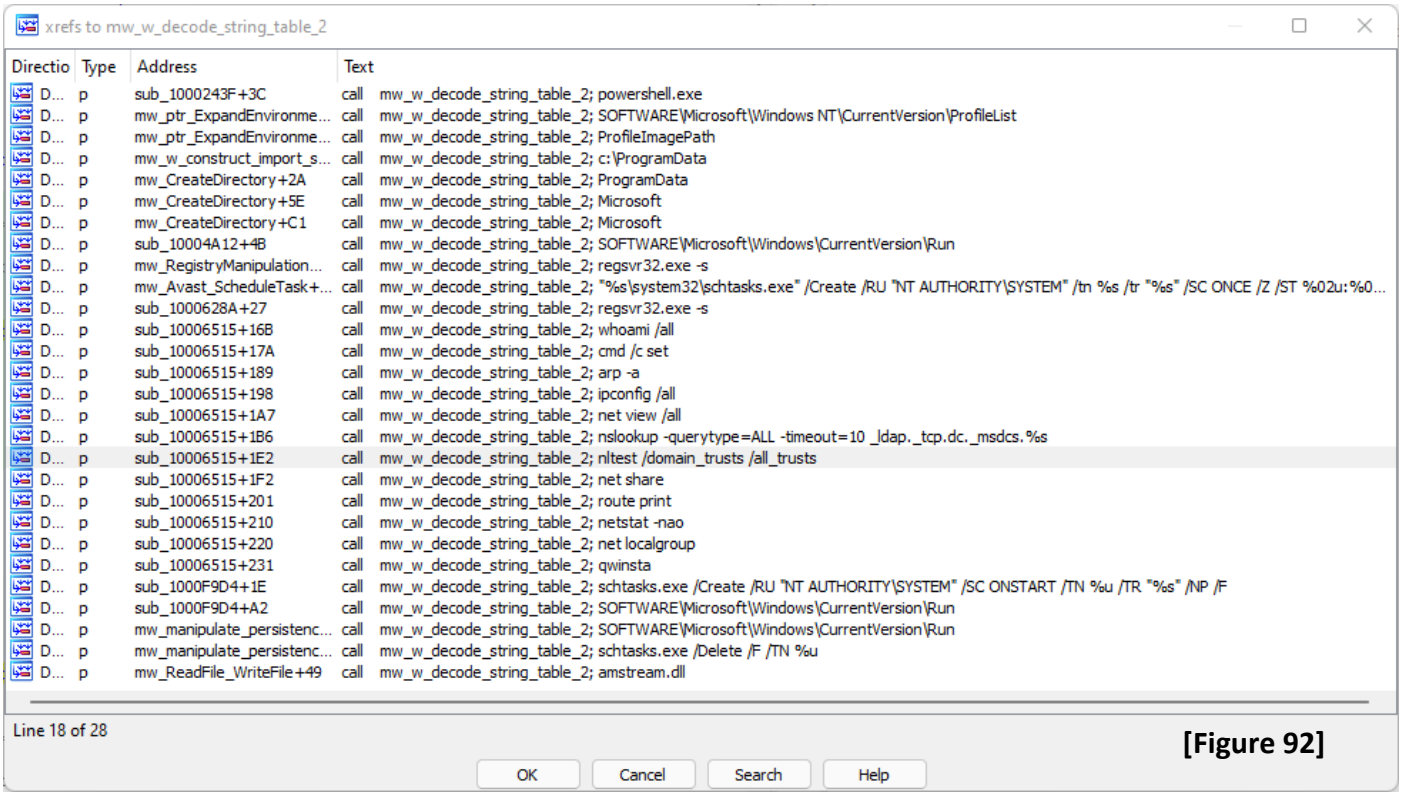
[Figure 90]

```
1  LPVOID __thiscall mw_NetGetJoinInformation(_DWORD *join_status)
2  {
3    unsigned int BufferType; // [esp+8h] [ebp-8h] BYREF
4    _WORD *lpNameBuffer; // [esp+Ch] [ebp-4h] BYREF
5
6    lpNameBuffer = (_WORD *)NetSetupUnknownStatus;
7    if ( ((int (__stdcall *)(_DWORD, _WORD **, unsigned int *))ptr_struct_iat_netapi32->ptr_NetGetJoinInformation)(
8           NetSetupUnknownStatus,
9           &lpNameBuffer,
10          &BufferType) )
11   {
12     return 0;
13   }
14   if ( BufferType < NetSetupWorkgroupName )
15   {
16     *join_status = NetSetupUnknownStatus;
17   }
18   else if ( BufferType == NetSetupWorkgroupName )
19   {
20     *join_status = NetSetupUnjoined;
21   }
22   else if ( BufferType == NetSetupDomainName )
23   {
24     *join_status = NetSetupWorkgroupName;
25   }
26   if ( !lpNameBuffer )
27     return 0;
28   else
29     return mw_w_construct_PE_Import_Structures_0(lpNameBuffer);
30 }
```

[Figure 91]

Keeping our searching for interesting stuff, the **sub_1000FB74 subroutine** calls once again the decoder of the string table (**mw_decode_string_table_2**) and we used our previous IDA Python script again to mark up as comment all of strings over the code:

- **comment_string_offset(0x1001D0B0, 0x1001D050, "mw_w_decode_string_table_2")**



[Figure 92]

Decrypting strings is always useful to understand the context of subroutines such **sub_1000FABA**, which handles and remove existing persistence (are they fake?) before start a process. In this case, strings tells about persistence on Registry (*SOFTWARE\Microsoft\Windows\CurrentVersion\Run*) and scheduled tasks (*schtasks.exe /Delete /F /TN %u*) as shown below:

```
1  int mw_manipulate_persistence()
2  {
3    int v0; // esi
4    wchar_t *var_persistence_tasks; // eax
5    _BYTE *var_persistence_registry; // [esp+0h] [ebp-8h] BYREF
6    wchar_t *Buffer; // [esp+4h] [ebp-4h] BYREF
7
8    var_persistence_registry = mw_w_decode_string_table_2(0x3EDu);// SOFTWARE\Microsoft\Windows\CurrentVersion\Run
9    mw_RegEnum_RegDeleteValueW(
10     HKEY_CURRENT_USER,
11     (int)var_persistence_registry,
12     (int)&ptr_struct_sysinfo_ref->sysinfo_7.pszSrch,
13     1);
14   mw_w_string_length_0(&var_persistence_registry);
15   if ( ptr_struct_sysinfo_ref->sysinfo_3.token_handle == 3 )
16   {
17     v0 = mw_sha1_stuff(60u);
18     if ( v0 != -1 )
19     {
20       Buffer = (wchar_t *)mw_HeapAlloc(0x100u);
21       if ( Buffer )
22       {
23         var_persistence_tasks = mw_w_decode_string_table_2(0x1B3u);// schtasks.exe /Delete /F /TN %u
24         mw_w_str_length_wchar(Buffer, 0x80u, var_persistence_tasks, v0);
25         if ( mw_CreateProcess((int)Buffer, 0, 0xBB8, 1) )
26           mw_w_RegDeleteValueA(60);
27         mw_w_string_length((_BYTE **)&Buffer, -2);
28       }
29     }
30   }
31   dword_1001E650 = 0;
32   return 0;
33 }
```

[Figure 93]

Analyzing the **CAPA Explorer**'s output, it suggests that there're two references to subroutines **checking of HTTP status code**. The second one is **sub_1000E815 subroutine** (renamed as **mw_HTTP_stuff**), which it's huge and has lots of **Wininet APIs** as shown below:

```
39  v29[0] = 0;
40  v32 = 4;
41  v29[1] = 1;
42  hConnect = 0;
43  memset(v25, 0, sizeof(v25));
44  memset(lpszObjectName, 0, sizeof(lpszObjectName));
45  lplpszAcceptTypes[0] = (int)mw_w_decode_string_table();// application/x-shockwave-flash
46  lplpszAcceptTypes[1] = (int)mw_w_decode_string_table();// image/gif
47  lplpszAcceptTypes[2] = (int)mw_w_decode_string_table();// image/jpeg
48  lplpszAcceptTypes[3] = (int)mw_w_decode_string_table();// image/pjpeg
49  v9 = mw_w_decode_string_table();              // */*
50  lplpszAcceptTypes[5] = 0;
51  lplpszAcceptTypes[4] = (int)v9;
52  memset(lpUrlComponents, 0, sizeof(lpUrlComponents));
53  lpUrlComponents[2] = 16;
54  lpUrlComponents[1] = (int)&v26;
55  lpUrlComponents[0] = 60;
56  lpUrlComponents[4] = (int)v25;
57  lpUrlComponents[5] = 256;
58  lpUrlComponents[11] = (int)lpszObjectName;
59  lpUrlComponents[12] = 256;
60  dwUrlLength = mw_str_length_char(lpszUrl);
61  if ( !((int (__stdcall *)(_BYTE *, int, _DWORD, int *))ptr_struct_iat_wininet->ptr_InternetCrackUrlA)(
62          lpszUrl,
63          dwUrlLength,
64          0,
65          lpUrlComponents) )
66    return -4;
67  v12 = 0;
68  v33 = 0;
69  while ( 1 )
70  {
71    dwAccessType = v29[v12];
72    lpdwBufferLength = 0x8404F700;
73    handle_Internet = ((int (__stdcall *)(int, int, _DWORD, _DWORD, _DWORD))ptr_struct_iat_wininet->ptr_InternetOpenA)(
74                        ptr_InternetOpenA_0,    // Mozilla/5.0 (Windows NT 6.1; rv:77.0) Gecko/20100101 Firefox/77.0
75                        dwAccessType,
76                        0,
77                        0,
78                        0);
```

[Figure 94]

This **sub_1000E815 subroutine** has a sequence of calls to well-known **WinINet APIs** such as:

- **InternetCrackUrlA**
- **InternetOpenA**
- **InternetSetOptionA**
- **InternetConnectA**
- **HttpOpenRequestA**
- **InternetQueryOptionA**
- **InternetSetOptionA**
- **HttpSendRequestA**
- **HttpQueryInfoA**

At time moment a critical question comes up: where IP addresses are? We haven't seen any IP address over the code and, worse, all decrypted strings so far don't have any IP address too. However, we just learned that the malware is communicating with C2 by using all these WinINet APIs, so at any point prior this subroutine should have references to IP addresses. Where are they?

Using cross-references (**'X' hotkey**), you can "go up" in the sequence of subroutine calls and try to understand what's the path to get to this WinINet APIs and, along the path, look around for interesting subroutines.

Therefore, using cross-references, we should jump to **sub_1000EBE2 → sub_1000EB81 → sub_1000729B.** At the middle of the path, we learn that **sub_1000EB81** has an interesting **Sleep( ) function** managing intervals of calls for **sub_1000EBE2 subroutine** and restricting it in something **between 2 and 6 seconds**.

In **sub_1000729B subroutine** we have a call for **sub_1000719F** that is before the sequence of calls to APIs mentioned in the previous page. Moving inside **sub_1000729B**, the code seems being interesting:

```
1  LPVOID __usercall mw_SHA_RC4@<eax>(int data_len@<edx>, _BYTE *a2@<ecx>, _DWORD *a3)
2  {
3    LPVOID ptr_data_buffer; // eax
4    int data_buffer; // edi
5    char key_stream[264]; // [esp+10h] [ebp-214h] BYREF
6    char v7[260]; // [esp+118h] [ebp-10Ch] BYREF
7    _BYTE *v8; // [esp+21Ch] [ebp-8h]
8
9    v8 = a2;
10   ptr_data_buffer = mw_HeapAlloc(data_len + 17);
11   data_buffer = (int)ptr_data_buffer;
12   if ( ptr_data_buffer )
13   {
14     mw_w_mersenne_random(&ptr_struct_sysinfo_ref->sysinfo_10.state, (int)v7, 0x10u);
15     mw_w_SHA1((int)key_stream);
16     mw_construct_PE_Import_Structures(data_buffer, v7, 16);
17     mw_construct_PE_Import_Structures(data_buffer + 16, v8, data_len);
18     mw_RC4_decrypt(data_len, data_buffer + 16, (int)key_stream);
19     if ( a3 )
20       *a3 = data_len + 16;
21     return (LPVOID)data_buffer;
22   }
23   return ptr_data_buffer;
24 }
```

[Figure 95]

The function on **line 14** is related to **Mersenne Twister**, which we've already explained previously, but subroutine call on **line 15** we have a promising call to **sub_10007118** (renamed as **mw_w_SHA1**). Go into this subroutine, we an interesting piece of code:

```
1  char *__cdecl mw_w_SHA1(int a1)
2  {
3    _BYTE *v1; // ecx
4    char *strange_string; // esi
5    int var_str_length; // esi
6    char buffer_data[256]; // [esp+8h] [ebp-11Ch] BYREF
7    char sha_buffer[20]; // [esp+108h] [ebp-1Ch] BYREF
8    char *strange_string_ref; // [esp+11Ch] [ebp-8h] BYREF
9
10   mw_construct_PE_Import_Structures((int)buffer_data, v1, 16);
11   strange_string = mw_w_decode_string_table_0();// jHxastDcds)oMc=jvh7wdUhxcsdt2
12   strange_string_ref = strange_string;
13   lstrcpynA(&buffer_data[16], strange_string, 240);
14   var_str_length = mw_str_length_char(strange_string);
15   mw_ww_string_length(&strange_string_ref);
16   mw_SHA1(var_str_length + 16, buffer_data, (int)sha_buffer);
17   memset(buffer_data, 0, sizeof(buffer_data));
18   return mw_RC4_init(20, (int)sha_buffer, a1);
19 }
```

[Figure 96]

The **sub_1000F681** subroutine (named **mw_SHA1**) on **line 16** is a **SHA1 routine** and its first lines provide necessary **cryptographic constants** to confirm it (check SHA1 pseudo code here: https://en.wikipedia.org/wiki/SHA-1):

```
19  v3 = 0;
20  v10 = arg_key;
21  v4 = arg_length;
22  v12 = 0;
23  v13 = 0;
24  v5 = 0;
25  v15 = 0;
26  v6 = 0;
27  v11[0] = 0x67452301;
28  v11[1] = 0xEFCDAB89;
29  v11[2] = 0x98BADCFE;
30  v11[3] = 0x10325476;
31  v11[4] = 0xC3D2E1F0;
32  v16 = 0;
33  v17 = 0;
34  if ( arg_length )
```

[Figure 97]

Returning to **mw_w_SHA1**, there's call for **sub_1000F353** (renamed as **mw_RC4_init**), which is a slightly modified **KSA (Key-scheduling algorithm)** that the reader can check reference on https://en.wikipedia.org/wiki/RC4:

```
14  key_len_ref = key_len;
15  var_256 = 256;
16  S_array_ref = S_array;
17  LOBYTE(key_len) = 0;
18  counter = 0;
19  counter_2 = (_BYTE *)S_array;
20  do
21    *counter_2++ = counter++;
22  while ( (__int16)counter < 256 );
23  *(_WORD *)(S_array + 256) = 0;
24  LOBYTE(v7) = 0;
25  key_stream = (char *)S_array;
26  S_array_updated = (_BYTE *)S_array;
27  do
28  {
29    v9 = (unsigned __int8)key_len;
30    key_stream_temp = *key_stream;
31    v7 = (unsigned __int8)(*key_stream + *(_BYTE *)(v9 + arg_key) + v7);
32    *S_array_updated = *(_BYTE *)(v7 + S_array_ref);
33    S_array_ref = S_array;
34    *(_BYTE *)(v7 + S_array) = key_stream_temp;
35    key_len = (v9 + 1) % key_len_ref;
36    key_stream = ++S_array_updated;
37    --var_256;
38  }
39  while ( var_256 );
40  return key_stream;
41 }
```

[Figure 98]

Returning to **sub_1000719F subroutine** (**mw_SHA_RC4**) on **Figure 96**, the call to **sub_1000F3C5** on **line 18** accepts the key stream returned by **mw_w_SHA1 subroutine**, so **it's a RC4's PRGA**, as expected:

```
1  char __usercall mw_RC4_decrypt@<al>(int arg_data_length@<edx>, int arg_data@<ecx>, int arg_key_stream)
2  {
3    char decrypted_data; // al
4    unsigned __int8 v5; // bl
5    unsigned __int8 key_stream_index; // bh
6    char start_key_stream; // dl
7    int counter; // [esp+18h] [ebp+8h]
8
9    decrypted_data = arg_data_length;
10   counter = 0;
11   v5 = *(_BYTE *)(arg_key_stream + 256);
12   for ( key_stream_index = *(_BYTE *)(arg_key_stream + 257); counter < arg_data_length; ++counter )
13   {
14     start_key_stream = *(_BYTE *)(++v5 + arg_key_stream);
15     key_stream_index += start_key_stream;
16     *(_BYTE *)(v5 + arg_key_stream) = *(_BYTE *)(key_stream_index + arg_key_stream);
17     *(_BYTE *)(key_stream_index + arg_key_stream) = start_key_stream;
18     decrypted_data = *(_BYTE *)((unsigned __int8)(start_key_stream + *(_BYTE *)(v5 + arg_key_stream)) + arg_key_stream);
19     *(_BYTE *)(counter + arg_data) ^= decrypted_data;
20   }
21   *(_BYTE *)(arg_key_stream + 256) = v5;
22   *(_BYTE *)(arg_key_stream + 257) = key_stream_index;
23   return decrypted_data;
24 }
```

[Figure 99]

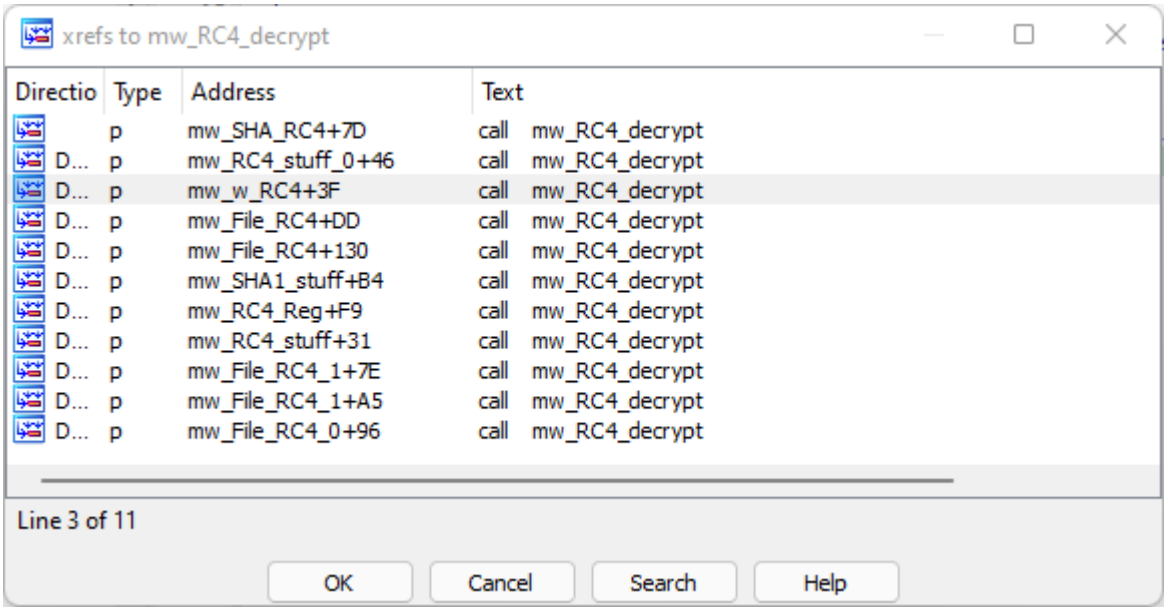
We've found three relevant pieces of our puzzle:

a. a call to a **SHA function** (**sub_1000F681** – renamed as **mw_SHA1**), which is called from **sub_10007118** (renamed as **mw_w_SHA1**) .

b. a call to a **RC4 KSA subroutine (sub_1000F353)** in **sub_10007118**.

c. a call to the **RC4 PRGA routine** (**sub_1000F3C5**, which was renamed to **mw_RC4_PRGA**) in **sub_1000179F** (renamed as **mw_SHA_RC4**).

We need to find which the code is being decrypting using RC4. From **Figure 95**, list all references to **mw_RC4_PRGA subroutine (sub_1000F3C5):**



[Figure 100]

Pick up the **third reference (sub_10008F17)** and go to there:

```
1  int __usercall mw_w_RC4@<eax>(
2          unsigned __int16 arg_key_length@<dx>,
3          int arg_key@<ecx>,
4          _BYTE *arg_data_1,
5          int arg_data_len,
6          _BYTE *arg_key_1)
7  {
8    char S_array[268]; // [esp+Ch] [ebp-120h] BYREF
9    char sha_buffer[20]; // [esp+118h] [ebp-14h] BYREF
10
11   mw_construct_PE_Import_Structures((int)arg_key_1, arg_data_1, arg_data_len);
12   mw_RC4_init(arg_key_length, arg_key, (int)S_array);
13   mw_RC4_PRGA(arg_data_len, (int)arg_key_1, (int)S_array);
14   mw_SHA1(arg_data_len - 20, arg_key_1 + 20, (int)sha_buffer);
15   if ( sub_1000891F(sha_buffer, arg_key_1, 20) )
16     return -1;
17   mw_construct_PE_Import_Structures((int)arg_key_1, arg_key_1 + 20, arg_data_len - 20);
18   return arg_data_len - 20;
19 }
```

[Figure 101]

Basically we found the same three critical subroutines, but in a different order. Listing cross references to subroutine on **Figure 101** (**mw_w_RC4 -- sub_10008F17**), we only found a couple of subroutines. Picking the first one up (**sub_100089C6** – named as **mw_ww_RC4**), we have many instructions, but three important lines of code:

- **(line 22) mw_SHA(arg_key_buffer_1, (char *)arg_key);**
- **(line 33) mw_w_RC4(0x14u, (int)key, key + 20, resource_size - 20, *arg_key_stream);**
- **(line 40) mw_w_RC4(arg_key_len, arg_key_1 + 1024, key, resource_size, *arg_key_stream);**

Of course, if you noticed the argument's names I used, so you can guess where we're going to. Once again, list all cross-references to **sub_10008F17** and you'll find only two subroutines, so choose the first one (**sub_10008AC1**).

Apparently, we didn't found anything, but it's too early to give up. Listing the cross-references to **sub_10008AC1 subroutine**, there'll be only two references again and choose the second one (**sub_10002783**).

We're inside of **sub_10002783 subroutine** (named **mw_w_resource_crypto_0**), which is really essential for our goals and there's three interesting lines here:

- The call for **sub_1000A6CA subroutine** (**mw_get_resources**) on **line 13**.

- The call to **sub_10001080 subroutine** (**mw_w_decode_string_table_0**) on **line 18**. We've already decoded the entired encoded string table and we know this specific one all bring the following string: **\System32\WindowsPowerShell\v1.0\powershell.exe**.

- The call for **sub_10008AC1** (named **mw_w_resource_crypto**) on **line 19**, which is exactly the subroutine that we came from.

The **sub_10002783 subroutine** (named **mw_w_resource_crypto_0**) is the following one:

```
1  _DWORD *__thiscall mw_w_resource_crypto_0(void *this)
2  {
3    _DWORD *v2; // edi
4    _BYTE *resource_data_ref; // esi
5    _BYTE *v4; // eax
6    _BYTE *v6; // [esp+Ch] [ebp-Ch] BYREF
7    char *key; // [esp+10h] [ebp-8h] BYREF
8    SIZE_T resource_size; // [esp+14h] [ebp-4h] BYREF
9
10   v2 = 0;
11   resource_size = 0;
12   key = mw_w_decode_string_table_0();           // 5812
13   resource_data_ref = mw_get_resources(key, (HMODULE)this, (int *)&resource_size);
14   v6 = resource_data_ref;
15   mw_ww_string_length(&key);
16   if ( resource_data_ref )
17   {
18     key = mw_w_decode_string_table_0();         // \System32\WindowsPowerShell\v1.0\powershell.exe
19     v4 = mw_w_resource_crypto(resource_size, resource_data_ref, (int)key);
20     if ( v4 )
21       v2 = sub_1000A493(v4);
22     mw_ww_string_length(&key);
23     mw_w_string_length(&v6, resource_size);
24   }
25   return v2;
26 }
```

[Figure 102]

There's an additional and short string to on **line 12**: **5812**. Write it down. Going into **sub_1000A6CA** subroutine, the mystery finally ends:

```
1  _BYTE *__usercall mw_get_resources@<eax>(const CHAR *arg_key@<edx>, HMODULE hModule@<ecx>, int *arg_resource_size)
2  {
3    int v4; // edi
4    mw_struct_iat_kernel32 *ptr_struct_iat_kernel32; // esi
5    int v6; // eax
6    HRSRC handle_res_1; // eax
7    int resource_size; // esi
8    _BYTE *handle_resource; // eax
9    _BYTE *resource_data; // ecx
10   int v12[2]; // [esp+10h] [ebp-10h]
11   HRSRC handle_res_1_ref; // [esp+18h] [ebp-8h]
12   const CHAR *v14; // [esp+1Ch] [ebp-4h]
13
14   v14 = arg_key;
15   v12[0] = 10;
16   v4 = 0;
17   v12[1] = 3;
18   while ( 1 )
19   {
20     ptr_struct_iat_kernel32 = ::ptr_struct_iat_kernel32;
21     v6 = mw_mersenne_random(&ptr_struct_sysinfo_ref->sysinfo_10.state, 30, 50);
22     handle_res_1 = ptr_struct_iat_kernel32->ptr_FindResourceA(
23                      hModule,
24                      v14,
25                      (LPCSTR)(v12[v4] + ptr_struct_sysinfo_ref->sysinfo_10.ptr_buffer_TokenInformation * v6));
26     handle_res_1_ref = handle_res_1;
27     if ( handle_res_1 )
28       break;
29     if ( (unsigned int)++v4 >= 2 )
30       return 0;
31   }
32   resource_size = ::ptr_struct_iat_kernel32->ptr_SizeofResource(hModule, handle_res_1);
33   handle_resource = ::ptr_struct_iat_kernel32->ptr_LoadResource(hModule, handle_res_1_ref);
```

[Figure 103]

Finally! This subroutine finds and loads resources data from the binary's **.rsrc section**. Return to **sub_10002783** subroutine (named **mw_w_resource_crypto_0**) and ask for cross-references to **sub_1000A6CA** subroutine (**mw_get_resources**) and pick up the first one (**sub_1000173B** – renamed to **mw_resource_decryptor**) and you're going to see the following relevant lines:

```
57  mw_w_string_length(&data_key, (int)resource_data);
58  data_key = mw_w_decode_string_table_0();      // 3719
59  resource_data = mw_get_resources(data_key, (HMODULE)ptr_struct_sysinfo_ref->sysinfo_3.hDLL, (int *)&resource_size);
60  mw_ww_string_length(&data_key);
61  if ( resource_data )
62  {
63    data_key = mw_w_decode_string_table_0();    // \System32\WindowsPowerShell\v1.0\powershell.exe
64    v25 = (unsigned int)mw_w_resource_crypto(resource_size, resource_data, (int)data_key);
65    mw_ww_string_length(&data_key);
66    if ( v25 )
67    {
68      v6 = sub_100016B1(*(_DWORD *)(v25 + 1064), *(_DWORD *)(v25 + 1060), (int *)&v24);
69      v2 = v24;
70      v30 = v6;
71      v21 = v6;
72      v26 = v24;
73    }
```

[Figure 104]

The idea is exactly the same:

- **(line 58)** provides a resource identification (**3719**).
- **(line 59)** gets the resource data.
- **(line 63)** the same "strange string".
- **(line 64)** decrypts the provided resource data.

Thus, it's feasible to understand the sequence of events from **Figures 102** and **104** and conclude that:

- The two provided **resource IDs** are: **3719** and **5812** (from **page 80**).
- The resource data is fetched.
- A key is provided: **\System32\WindowsPowerShell\v1.0\powershell.exe**
- The decrypting function (**mw_w_resource_crypto**) is called.
- Following the decrypting function, we return to **sub_100089C6** (renamed as **mw_ww_RC4**) again and have the same lines:

    - **(line 22) mw_SHA(arg_key_buffer_1, (char *)arg_key);**
    - **(line 33) mw_w_RC4(0x14u, (int)key, key + 20, resource_size - 20, *arg_key_stream);**
    - **(line 40) mw_w_RC4(arg_key_len, arg_key_1 + 1024, key, resource_size, *arg_key_stream);**

A minimum interpretation of these line of code is:

- The key is provided to a **SHA1 subroutine**.
- The result from **SHA1 subroutine** (**key_stream**) is provided to RC4 routine. In this case the SHA1 function is working as a **Key Derivation Function (KDF).**
- The RC4 subroutine returns both resource data (**lines 33** and **40**).

Is there any indicator that the stored information in the resource section are IP addresses? From **sub_1000173B** subroutine (named **mw_resource_decryptor**), which is the same of **Figure 104**, list all cross-references and pick up the first one (**sub_100019DE subroutine**) and go to there.

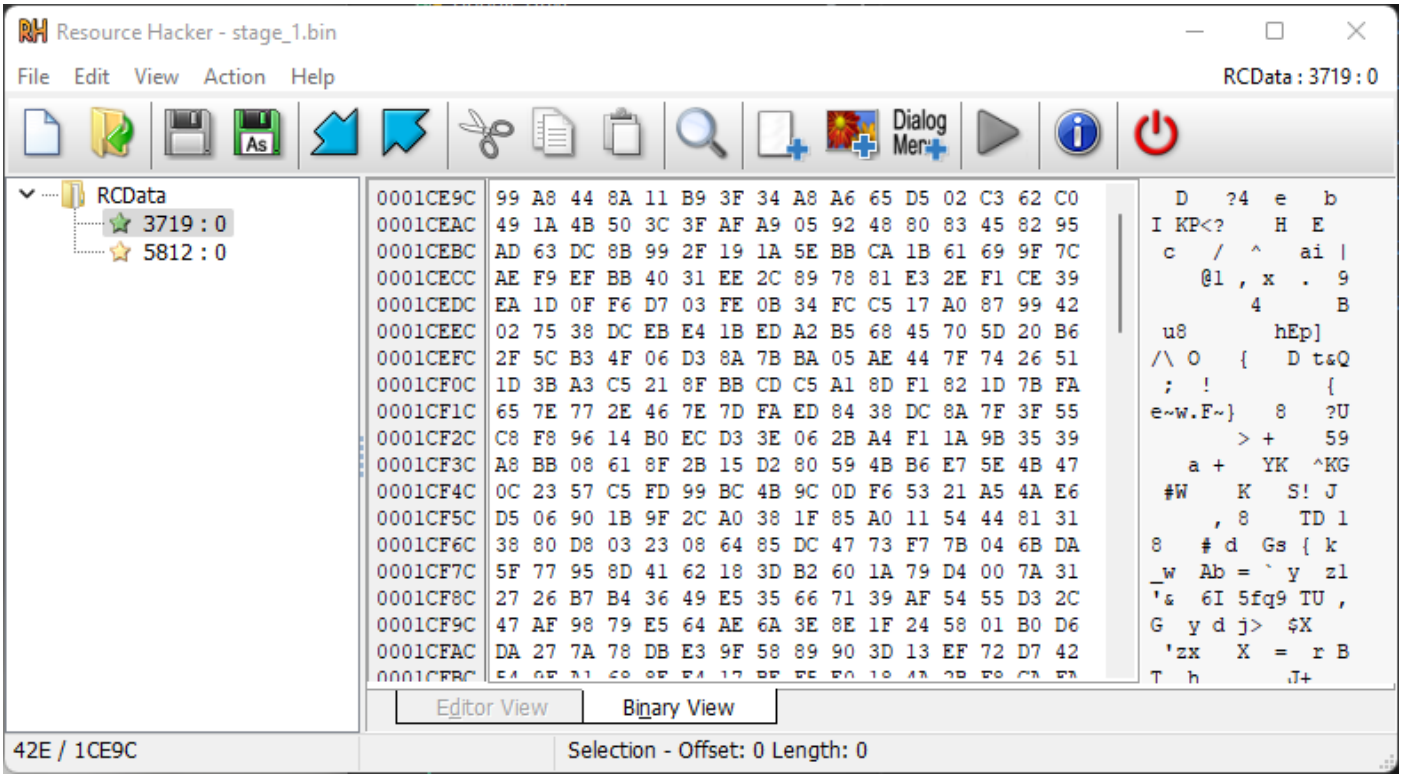
Readers are going to see the **inet_ntoa( )** function that **converts an IPv4 address into an ASCII string in dotted-decimal format**. Thus, we can confirm our hypothesis and the information stored in the **.rsrc section** is actually a set of C2's IPv4.

Nonetheless, it isn't all. If readers repeat the process of getting the cross-references of **sub_1000173B** subroutine (named **mw_resource_decryptor**) and picking up the second cross-reference (**sub_1000336E** subroutine), so they are going to find a very suggesting line of code:

- **mw_w_str_length_char(&v21[v26], v29, "%u.%u.%u.%u:%u", v30, v31, v33, v36, v39);**

As you can guess, IP addresses seem to be formatted as **X.Y.Z.W:port**, so we have to take it into account while extracting and decoding resource data.

Using **Resource Hacker tool** (http://www.angusj.com/resourcehacker/) the reader is able to confirm that there're two **resource IDs** (**3719** and **5812**) as we've learned from the code:



[Figure 105]

It's great! Our next task is writing a code to extract and decrypt the **.rsrc section information** and on **page 81** we have a good hint about what we need to do. As I showed in the previous page, the **mw_w_RC4 subroutine (sub_100089C6)** is called twice to **decrypt both resource's IDs**, but pay attention to **line 33 (page 81)**. It suggests that the **real data starts at byte 20 onward**. Curiously, 20 bytes are the same length of any hash provided by SHA1. A script must:

- **Extract the resource data.**
- **Apply SHA1 on the key, so generating a derived key.**
- **Decrypt the extracted resource data using this key.**
- **Formatting the output to IPV4 notation.**

As most malware threats, C2 IP addresses are followed by their respective ports, so the format has the following pattern: **[4 bytes for IP Address][2 bytes for Port].**

The following **Python 3 script** has been written for educational purposes and there are many lines of the code that wouldn't be necessary (few of them are commented), but they might help you:

```
import binascii
import pefile
import ipaddress
from Crypto.Cipher import ARC4
from Crypto.Hash import SHA1

# Key found in the code (doubled slashs because the interpretation)
key = b'\\System32\\WindowsPowerShell\\v1.0\\powershell.exe'

# SHA hash of the key
sha1_key = SHA1.new(data=key).digest()

# This routine extracts data from .rsrc section.
def extract_resource(filename, res_identification):

    extracted_data = b""
    pe=pefile.PE(filename)
    for resource in pe.DIRECTORY_ENTRY_RESOURCE.entries:
        if  hasattr(resource, 'directory'):
            for res_id in resource.directory.entries:
                if hasattr(res_id, 'name'):
                    if (res_id.name):
                      if (str(res_id.name) == str(res_identification)):
                         offset = res_id.directory.entries[0].data.struct.OffsetToData
                         resid_size = res_id.directory.entries[0].data.struct.Size

    extracted_data = pe.get_memory_mapped_image()[offset:offset+resid_size]
    return extracted_data

# This routine decrypt resource data using RC4 + hashed key
def data_decryptor(key_data, data):

    data_cipher = ARC4.new(key_data)
    decrypted_config = data_cipher.decrypt(data)
    return decrypted_config

# Main routine used to perform all necessary calls.
def main( ):

    # Path to the malware
    filename = r"C:\Users\Administrador\Desktop\MAS\mas_3\stage_1.bin"

    # Extracts the second resource id (smaller) and decrypt it.
    resource_data = extract_resource(filename, 5812)
    hex_resource_data = binascii.hexlify(resource_data)
```

```
46
47      # Remember that the useful data start at 21th byte onward.
48      decrypted_data = data_decryptor(sha1_key,resource_data)[20:]
49
50      # Prints extracted and decrypted resource
51      # Uncomment next three lines to show hexadecimal representation.
52      #print('\nEXTRACTED RESOURCE DATA 5812: ', end='')
53      #print(hex_resource_data)
54      #print(str(decrypted_data))
55      print('\nDECRYPTED BOTNET AND CAMPAIGN ID: ', end='')
56      print("\n" + 34*'-')
57      print(decrypted_data.decode('latin1'))
58
59      # extracts the first resource id and decrypt it.
60      resource_data = extract_resource(filename, 3719)
61      decrypted_data = data_decryptor(sha1_key,resource_data)
62
63      # Prints SHA1 key, extracted and decrypted resource
64      # Uncoment next 6 lines if you want to see hexdecimal representation.
65      #print('SHA1 KEY: ', end='')
66      #print(binascii.hexlify(sha1_key))
67      #print('\nEXTRACTED RESOURCE DATA 3719: ', end='')
68      #print(binascii.hexlify(resource_data))
69      #print('\nDECRYPTED RESOURCE 3719: ', end='')
70      #print(binascii.hexlify(decrypted_data))
71
72      # Remember that the useful data start at 21th byte onward.
73      resource_item = decrypted_data[21:]
74
75      # The format is: [4 bytes for IP address][2 bytes of port]
76      # Extracts and print the IP:port list
77      print('C2 IP ADDRESS LIST: ')
78      print(30*'-')
79
80      k = 0
81      i = 0
82      while (k < len(resource_item)):
83          ip_item = resource_item[k:k+4]
84          ip_port = resource_item[k+4:k+6]
85          print("IP[%d]: %s" % (i,ipaddress.IPv4Address(ip_item)),end=':')
86          print(int(binascii.hexlify(ip_port),16))
87          k = k + 7
88          i = i + 1
89
90  if __name__ == '__main__':
91      main( )
```

[Figure 106]

I've chosen not writing my own RC4 implementation (and there're several ones available on the Internet) because **PyCryptodome** (https://pycryptodome.readthedocs.io/en/latest/src/introduction.html) works well for this kind of task. Additionally, I've also used an existing Python IP Address library (**ipaddress:** https://docs.python.org/3/library/ipaddress.html) because it makes the code more readable. The output from our scripts can be checked against the Triage's output on **page 7** (when it was truncated):

```
DECRYPTED BOTNET AND CAMPAIGN ID:
---------------------------------
10=obama150
3=1640256791

C2 IP ADDRESS LIST:
-------------------------------
IP[0]: 96.21.251.127:2222
IP[1]: 70.51.134.181:2222
IP[2]: 69.14.172.24:443
IP[3]: 186.64.87.213:443
IP[4]: 94.62.161.77:995
IP[5]: 103.139.242.30:990
IP[6]: 114.79.148.170:443
IP[7]: 217.164.247.241:2222
IP[8]: 178.153.86.181:443
IP[9]: 136.232.34.70:443
IP[10]: 37.210.226.125:61202
IP[11]: 173.21.10.71:2222
IP[12]: 31.219.154.176:32101
IP[13]: 140.82.49.12:443
IP[14]: 32.221.229.7:443
IP[15]: 24.152.219.253:995
IP[16]: 106.51.48.170:50001
IP[17]: 114.38.161.124:995
IP[18]: 96.37.113.36:993
IP[19]: 190.39.205.165:443
IP[20]: 45.9.20.200:2211
IP[21]: 105.198.236.99:995
IP[22]: 70.163.1.219:443
IP[23]: 103.139.242.30:995
IP[24]: 24.95.61.62:443
IP[25]: 136.143.11.232:443
IP[26]: 31.215.215.152:1194
IP[27]: 103.143.8.71:6881
IP[28]: 102.65.38.67:443
IP[29]: 31.215.70.105:443
IP[30]: 86.97.9.221:443
IP[31]: 83.110.91.18:2222
IP[32]: 63.153.187.104:443
IP[33]: 74.15.2.252:2222
IP[34]: 217.165.123.47:61200
IP[35]: 41.228.22.180:443
```

```
IP[36]: 24.53.49.240:443
IP[37]: 149.135.101.20:443
IP[38]: 94.200.181.154:995
IP[39]: 67.209.195.198:443
IP[40]: 209.210.95.228:32100
IP[41]: 96.80.109.57:995
IP[42]: 80.14.196.176:2222
IP[43]: 38.70.253.226:2222
IP[44]: 24.222.20.254:443
IP[45]: 103.142.10.177:443
IP[46]: 217.128.93.27:2222
IP[47]: 103.157.122.130:21
IP[48]: 24.178.196.158:2222
IP[49]: 182.191.92.203:995
IP[50]: 76.169.147.192:32103
IP[51]: 78.180.66.163:995
IP[52]: 89.41.8.168:443
IP[53]: 190.73.3.148:2222
IP[54]: 79.173.195.234:443
IP[55]: 120.150.218.241:995
IP[56]: 182.56.56.249:443
IP[57]: 121.175.104.13:32100
IP[58]: 76.25.142.196:443
IP[59]: 79.167.192.206:995
IP[60]: 59.6.7.83:61200
IP[61]: 71.74.12.34:443
IP[62]: 83.110.98.231:995
IP[63]: 89.137.52.44:443
IP[64]: 114.143.92.41:61202
IP[65]: 67.165.206.193:993
IP[66]: 94.60.254.81:443
IP[67]: 23.233.146.92:443
IP[68]: 73.151.236.31:443
IP[69]: 96.80.109.57:995
IP[70]: 187.162.59.232:995
IP[71]: 72.252.201.34:995
IP[72]: 50.237.134.22:995
IP[73]: 201.172.31.95:80
IP[74]: 100.1.119.41:443
IP[75]: 40.134.247.125:995
IP[76]: 109.12.111.14:443
IP[77]: 89.101.97.139:443
IP[78]: 24.55.112.61:443
IP[79]: 93.48.80.198:995
IP[80]: 75.188.35.168:443
IP[81]: 83.199.144.45:2222
IP[82]: 92.154.9.41:50002
IP[83]: 111.125.245.116:995
IP[84]: 39.49.105.128:995
IP[85]: 82.152.39.39:443
IP[86]: 105.106.30.144:443
IP[87]: 31.35.28.29:443
IP[88]: 103.139.242.30:22
IP[89]: 218.101.110.3:995
IP[90]: 182.176.180.73:443
IP[91]: 121.175.104.13:443
IP[92]: 65.100.174.110:8443
IP[93]: 79.160.207.214:443
IP[94]: 70.224.68.92:443
IP[95]: 173.25.166.81:443
IP[96]: 176.205.152.44:443
IP[97]: 108.4.67.252:443
IP[98]: 189.174.46.65:995
IP[99]: 187.189.86.168:443
IP[100]: 176.24.150.197:443
IP[101]: 86.98.52.117:443
IP[102]: 200.54.14.34:80
IP[103]: 103.139.242.30:443
IP[104]: 103.139.242.30:465
IP[105]: 103.139.242.30:993
IP[106]: 103.139.242.30:993
IP[107]: 78.101.89.174:2222
IP[108]: 78.101.89.174:443
IP[109]: 73.5.119.219:443
IP[110]: 74.5.148.57:443
IP[111]: 68.186.192.69:443
IP[112]: 50.33.112.74:995
IP[113]: 70.93.80.154:995
IP[114]: 75.169.58.229:32100
IP[115]: 63.143.92.99:995
IP[116]: 217.39.100.89:443
IP[117]: 46.9.77.245:995
IP[118]: 173.71.147.134:995
IP[119]: 75.110.250.187:443
IP[120]: 194.36.28.238:443
IP[121]: 65.100.174.110:443
IP[122]: 65.100.174.110:443
IP[123]: 82.78.212.133:443
IP[124]: 83.110.107.123:443
IP[125]: 59.88.168.108:443
IP[126]: 65.128.74.102:443
IP[127]: 68.204.7.158:443
IP[128]: 78.101.82.198:995
IP[129]: 80.6.192.58:443
IP[130]: 41.97.234.150:995
IP[131]: 114.79.145.28:443
IP[132]: 188.54.96.91:443
IP[133]: 50.238.6.36:443
IP[134]: 83.110.107.123:443
IP[135]: 217.165.11.65:61200
IP[136]: 103.143.8.71:995
IP[137]: 2.178.67.97:61202
IP[138]: 86.198.237.51:2222
IP[139]: 88.253.171.236:995
IP[140]: 187.172.146.123:443
IP[141]: 92.167.4.71:2222
IP[142]: 189.30.244.252:995
IP[143]: 194.36.28.26:443
IP[144]: 84.199.230.66:443
IP[145]: 14.96.67.177:443
IP[146]: 50.238.6.36:443
IP[147]: 182.56.57.23:995
IP[148]: 87.70.118.51:443
IP[149]: 93.48.58.123:2222
```

[Figure 107]

From the output, we learned that the botnet's name is "**obama150**" and the number might be a reference of existing 150 IP addresses in the C2 configuration list. Additionally, the campaign ID is **1640256791**.

Once again, this decrypt script can be improved a lot in several areas by removing fixed inputs such as filename, encrypting key and resource IDs , but I think that it's enough to understand the general idea.

6. Further observations

The **sub_100084AF subroutine** shows us an interesting sequence of subroutine and Windows APIs calls:

```
pbEncoded = mw_HeapAlloc(0x126u);
if ( !pbEncoded )
  return -2;
do
{
  decoded_content = &pbEncoded[counter];
  decoded_content_temp = mw_encrypted_string_1[counter] ^ mw_encrypted_string_2[counter & 0xF];
  ++counter;
  *decoded_content = decoded_content_temp;
}
while ( counter < 0x126 );
ptr_memory_HeapAlloc_ref = pbEncoded;
if ( ((int (__stdcall *)(int, int, _BYTE *, int, MACRO_CRYPT_FORMAT, _DWORD, void **, int *))prt_mw_struct_iat_crypt32->ptr_CryptDecodeObjectEx)(
       1,
       8,
       pbEncoded,
       294,
       CRYPT_DECODE_ALLOC_FLAG,
       0,
       &pvStructInfo,
       &pcbStructInfo)
  && (ptr_struct_iat_advapi32->ptr_CryptAcquireContextA(&hCryptProv, 0, 0, 1, CRYPT_VERIFYCONTEXT)
   || ptr_struct_iat_advapi32->ptr_CryptAcquireContextA(&hCryptProv, 0, 0, 1, CRYPT_UPDATE_KEY|CRYPT_VERIFYCONTEXT))
  && ((int (__stdcall *)(HCRYPTPROV, int, void *, HCRYPTKEY *))prt_mw_struct_iat_crypt32->ptr_CryptImportPublicKeyInfo)(
       hCryptProv,
       0x10001,
       pvStructInfo,
       &hPubKey)
  && ptr_struct_iat_advapi32->ptr_CryptCreateHash(
       hCryptProv,
       CRYPT_FORCE_KEY_PROTECTION_HIGH|CRYPT_CREATE_SALT,
       0,
       0,
       &phKey)
  && ptr_struct_iat_advapi32->ptr_CryptHashData(phKey, pbData, dwDataLen, 0) )
{
  pbSignature = (const BYTE *)v14;
  v9 = 0;
  v10 = (BYTE *)(v14 + 255);
  do
  {
    v11 = pbSignature[v9];
    pbSignature[v9++] = *v10;
    *v10-- = v11;
  }
  while ( v9 < 0x80 );
  if ( ptr_struct_iat_advapi32->ptr_CryptVerifySignatureA(phKey, pbSignature, 256, hPubKey, 0, 0) )
    var_ret = 1;
```

[Figure 108]

Few of APIs being called are:

- **CryptDecodeObjectEx:** this function decodes a structure of the type indicated by the **lpszStructType** parameter.
- **CryptAcquireContext:** this function acquires a handle to a particular key container within a particular **cryptographic service provider (CSP).**
- **CryptImportPublicKeyInfo**: this function converts and imports the public key information into the provider and returns a handle of the public key.
- **CryptCreateHash:** this function initiates the hashing of a stream of data.
- **CryptHashData:** this function adds data to a specified hash object.
- **CryptVerifySignatureA:** this function verifies the signature of a hash object

Based on APIs being invoked, this piece of code seems to be handling a public key involved in a C2 communication. Furthermore, soon at its beginning, there's an interesting XOR operation that take us to two encoded data in different binary sections, which **encrypted data is stored in the .data section** and the **key is stored in the .rdata section.** To extract, decrypt and format the possible public key we are going to use a script very similar to used previously, but slightly changed:

```python
# Extracts and format the public key

import pefile
import binascii

data_seg_start = ''
rdata_seg_start = ''

# Decrypter routine used to decode the stored data.
def simple_decrypter(data_string, data_key):
    decoded = ''

    for i in range(0, len(data_string)):
        decoded += chr((data_string[i]) ^ (data_key[i % len(data_key)]))
    return (binascii.b2a_hex(decoded.encode('latin-1')))

# Routine responsible for extracting encoded bytes from .data section.
def extract_data(filename):
    pe=pefile.PE(filename)
    for section in pe.sections:
        if '.data' in section.Name.decode(encoding='utf-8').rstrip('x00'):
            return (section.get_data(section.VirtualAddress, section.SizeOfRawData))

# Routine responsible for extracting encoded bytes from .rdata section
def extract_rdata(filename):
    pe2=pefile.PE(filename)
    for section2 in pe2.sections:
        if '.rdata' in section2.Name.decode(encoding='utf-8').rstrip('x00'):
            return (section2.get_data((section2.VirtualAddress + 0x168), section2.SizeOfRawData))

# This routine calculates the offset from start of the section until the address of the data.
def calc_offsets(x_seg_start, x_start):

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

[Figure 109]

```python
from Crypto.IO import PEM

def main():

    data_2 = b''
    rdata_2 = b''

    # Defines start of each section (.data and .rdata section)
    # and encrypted data (data and key).
    data_seg_start = '0x1001D000'
    rdata_seg_start = '0x10018168'
    data_start = '0x1001E528'
    rdata_start = '0x1001B868'

    # Calculates offset of data and key related to the
    # start of each respective section.
    data_rel = calc_offsets(data_seg_start, data_start)
    rdata_rel = calc_offsets(rdata_seg_start, rdata_start)

    # Defines a variable to hold the sample's path
    filename = r"C:\Users\Administrador\Desktop\MAS\mas_3\stage_1.bin"
```

```
22
23      # Call extract routine to fetch all necessary bytes from .data and
24      # .rdata section.
25      data_1 = extract_data(filename)
26      rdata_1 = extract_rdata(filename)
27
28      # Looking for the end of data and key bytes.
29      d_off = 0x0
30      rd_off = 0x0
31      if (b'\x00\x00' in data_1[int(data_rel,16):]):
32          d_off = (data_1[int(data_rel,16):]).index(b'\x00\x00')
33      if (b'\x00\x00' in rdata_1[int(rdata_rel,16):]):
34          rd_off = (rdata_1[int(rdata_rel,16):]).index(b'\x00\x00')
35
36      # Uncomment the next 4 lines to verify extracted data and key bytes.
37      # print("\nData extracted from .data section: ", end='')
38      #print(binascii.b2a_hex(data_1[int(data_rel,16):int(data_rel,16) + d_off]))
39      #print("\nData extracted from .rdata section: ", end='')
40      #print(binascii.b2a_hex(rdata_1[int(rdata_rel,16):int(rdata_rel,16) + rd_off]))
41
42      # Collects encrypted data and key.
43      data_2 = data_1[int(data_rel,16):int(data_rel,16) + d_off]
44      rdata_2 = rdata_1[int(rdata_rel,16):int(rdata_rel,16) + rd_off]
45
46      # Calls function responsible for decoding the encrypted data.
47      decoded_data = simple_decrypter(data_2, rdata_2)
48
49      # Uncomment next three line to show the decrypted data in hexadecimal
50      # format.
51      #print("\n")
52      #print(decoded_data)
53      #print("\n")
54
55      # Format the extracted data as a public key in PEM Format.
56      marker =  "RSA PUBLIC KEY"
57      pem_key = PEM.encode(decoded_data, marker, passphrase=None, randfunc=None)
58      print(pem_key)
59
60  if __name__ == '__main__':
61      main( )
```

```
-----BEGIN RSA PUBLIC KEY-----
MzA4MjAxMjIzMDBkMDYwOTJhODY0ODg2ZjcwZDAxMDEzYjIxNWY3MmUwN2M1NDBj
NmVhZTYyNzM0MDk4M2UxNTU5ZGRjYWMzODYxZmMxZWZmNmMyMzFmMDNjYTZmN2I0
YWViMzU3OTY4NzM2YzkyZDZlNWFiNGVjYTA0ZjI3MGE2YzE3MzIwNTdlNWVkOTM4
MjI0YzQ5MjBjZjBiZjUzMmI4Y2NkNjUzNzE0MWFjYmRmMTZlZmYyOGU2ZmE0Mjg2
ZDljYzViNzI3ZTM3MzZkM2NlMWI5NDQwOTkzMDI2MWExNmZjMjI1NjA5NzhkMjU4
NTA2ODkyYTRiZjA0NjMzMDcwODdkNDFlY2U3YTBlZDc0ZjMwMzAxZGEyN2UyYmJl
NTlkMjM1Y2E2YWM3OTA4ZDcxOTkxZGFiOTY4MThiNDJmZmE3Yzk1NjZlODZkZjg0
ZDMzZTcwN2ZhNGFjYmQ0ODNhOTk5MjUyYTkyMmVmZTFhZjg1OWJjYjY0YmZiY2Rm
MDMxYmJmNzczYTI4MDVkMzYzYzE3YzUzNTFkZGJiMGNjZjc0MzdjM2I2OTcyNTAx
YjM1MzgyMjM2ZmQwMzJhZDBkMGE1N2I2OGZhMDM3OWFhZDE0YjA3M2E3MjRhYTY4
NDI0NzE2YjdhNDcyNmZjNDg0ZTk2N2NjMzNlYjc0MzcwMmU1NjIxZjVmODhjZDdh
ZDVmNjlmOTQ4YTlkNWNkNTdmMDhhMmNmNTgxYTIzN2U0ZDQ3NDAzM2YxMDc0NDdl
NjUyNjVjNzA2Mjdj
-----END RSA PUBLIC KEY-----
```

[Figure 110]

Another very interesting subroutine is **sub_1000D9B1** because there're few interesting **WMI strings** such as **'SELECT * FROM Win32_OperatingSystem'** and also **'ROOT\CIMv2'**. Where's the code referring WMI?

```
LPVOID sub_1000D9B1()
{
  LPVOID v0; // edi
  _BYTE *v1; // ebx
  OLECHAR *v3; // esi
  LPVOID v4; // eax
  VARIANTARG pvarg; // [esp+14h] [ebp-20h] BYREF
  _BYTE *v6; // [esp+28h] [ebp-Ch] BYREF
  OLECHAR *v7; // [esp+2Ch] [ebp-8h] BYREF
  OLECHAR *v8; // [esp+30h] [ebp-4h] BYREF

  v0 = 0;
  v8 = mw_w_decode_string_table_1(0xACDu);      // ROOT\CIMV2
  v1 = mw_COM_IWbemLocator(v8);
  v6 = v1;
  mw_w_string_length_0((_BYTE **)&v8);
  if ( !v1 )
    return 0;
  v3 = mw_w_decode_string_table_1(0xA99u);      // SELECT * FROM Win32_OperatingSystem
  v8 = v3;
  v7 = mw_w_decode_string_table_1(0xAD8u);      // Caption
  if ( sub_1000D8F5(v3, *(_DWORD *)v1, v7, (int)&pvarg) )
  {
    v4 = mw_w_construct_PE_Import_Structures_0(pvarg.bstrVal);
    v0 = v4;
    if ( v4 )
      ((void (__stdcall *)(LPVOID, const wchar_t *))ptr_struct_iat_shlwapi->ptr_strtrimW)(v4, L" ");
    VariantClear(&pvarg);
  }
  mw_w_string_length_0((_BYTE **)&v8);
  mw_w_string_length_0((_BYTE **)&v7);
  sub_1000D784(&v6);
  return v0;
}
```

[Figure 111]

Analyzing the **sub_1000D6D0** subroutine, we have:

```
  ppv = 0;
  pProxy = 0;
  CoInitializeEx(0, 0);
  CoInitializeSecurity(0, -1, 0, 0, 0, 3u, 0, 0, 0);
  if ( CoCreateInstance(&rclsid, 0, 1u, &riid, &ppv) >= 0
    && (v2 = SysAllocString(psz),
        (*(int (__stdcall **)(LPVOID, BSTR, _DWORD, _DWORD, _DWORD, _DWORD, _DWORD, _DWORD, IUnknown **))(*(_DWORD *)ppv + 0xC))(
          ppv,
          v2,
          0,
          0,
          0,
          0,
          0,
          0,
          &pProxy) >= 0)
    && CoSetProxyBlanket(pProxy, 0xAu, 0, 0, 3u, 3u, 0, 0) >= 0
    && (result = mw_HeapAlloc(8u)) != 0 )
  {
    result[1] = ppv;
    *result = pProxy;
  }
  else
  {
    if ( pProxy )
      pProxy->lpVtbl->Release(pProxy);
    if ( ppv )
      (*(void (__stdcall **)(LPVOID))(*(_DWORD *)ppv + 8))(ppv);
    return 0;
  }
  return result;
}
```

[Figure 112]

The **class ID** and **interface ID** values are stored in the **.rdata section** as shown below:

```
.rdata:1001B840 rclsid          dd 4590F811h            ; Data1
.rdata:1001B840                                         ; DATA XREF: sub_1000D6D0+36↑o
.rdata:1001B840                 dw 1D3Ah                ; Data2
.rdata:1001B840                 dw 11D0h                ; Data3
.rdata:1001B840                 db 89h, 1Fh, 0, 0AAh, 0, 4Bh, 2Eh, 24h; Data4
.rdata:1001B850 ; const IID riid
.rdata:1001B850 riid            dd 0DC12A687h           ; Data1
.rdata:1001B850                                         ; DATA XREF: sub_1000D6D0+2E↑o
.rdata:1001B850                 dw 737Fh                ; Data2
.rdata:1001B850                 dw 11CFh                ; Data3
.rdata:1001B850                 db 88h, 4Dh, 0, 0AAh, 0, 4Bh, 2Eh, 24h; Data4
.rdata:1001B860                 db 0F3h
.rdata:1001B861                 db    4
.rdata:1001B862                 db    0
.rdata:1001B863                 db    0
.rdata:1001B864                 db  5Ah ; Z
.rdata:1001B865                 db    0
.rdata:1001B866                 db    0
.rdata:1001B867                 db    0
```

[Figure 113]

Easily the reader is able to understand that **class ID** and **interface ID** are the following **GUIDs:**

- **rclsid: 4590F811-1D3A-11D0-891F-00AA004B2E24**
- **riid: DC12A687-737F-11CF-884D-00AA004B2E24**

Looking for the **interface ID (riid)** on the Internet, the reader will discover that it refers to the **IWbemLocator** interface. Therefore, you change the type (**'Y' hotkey**) of **ppv** on **line 8 and 12** from **'LPVOID'** to **'IWbemLocator*'** and set **ptr_buffer**'s type to '**IWbemServices ***' (check the allocation on **line 16**), you'll get a **better pseudo code than the Figure 112**:

```
1  IWbemServices *__thiscall mw_COM_IWbemLocator(OLECHAR *arg_network_strings)
2  {
3    OLECHAR *ptr_network_strings; // eax
4    IWbemServices *ptr_buffer; // eax
5    IWbemLocator *ppv; // [esp+8h] [ebp-8h] BYREF
6    IUnknown *pProxy; // [esp+Ch] [ebp-4h] BYREF
7
8    ppv = 0;
9    pProxy = 0;
10   CoInitializeEx(0, 0);
11   CoInitializeSecurity(0, -1, 0, 0, 0, 3u, 0, 0, 0);
12   if ( CoCreateInstance(&rclsid, 0, 1u, &riid, &ppv) >= 0
13     && (ptr_network_strings = SysAllocString(arg_network_strings),
14         ppv->lpVtbl->ConnectServer(ppv, ptr_network_strings, 0, 0, 0, 0, 0, 0, (IWbemServices **)&pProxy) >= 0)
15     && CoSetProxyBlanket(pProxy, 0xAu, 0, 0, 3u, 3u, 0, 0) >= 0
16     && (ptr_buffer = (IWbemServices *)mw_HeapAlloc(8u)) != 0 )
17   {
18     ptr_buffer[1].lpVtbl = (struct IWbemServicesVtbl *)ppv;
19     ptr_buffer->lpVtbl = (struct IWbemServicesVtbl *)pProxy;
20   }
21   else
22   {
23     if ( pProxy )
24       pProxy->lpVtbl->Release(pProxy);
25     if ( ppv )
26       ppv->lpVtbl->Release(ppv);
27     return 0;
28   }
29   return ptr_buffer;
30 }
```

[Figure 114]

As you probably already know, **IWbemLocator interface** offers **Windows Management** through **IWbemServices**, which is returned by the **ConnectServer( ) method**. In other words, **IWbemLocator::ConnectServer creates a connection to a WMI namespace**, so that's the reason for we have seen a WMI query previously. Furthermore, the strings **"ROOT\CIMv2"** is passed within the **strNetworkResource parameter** to **ConnectServer function**, and it makes sense because this parameter must contain the object path of the WMI namespace.

Returning to **sub_1000DCE9 routine** (renamed as **mw_w_COM_IWbemLocator**), we should change the result's type on **line 49** to **IWbemServices *** (same from **ptr_buffer** on **Figure 114**). Afterwards, change the type of v7 variable to **IWbemServices **** on **line 80** that **ExecQuery( )** will come up. Remember that **IWbemServices::ExecQuery** method executes a query to retrieve object and whether the reader observe **lines 63, 64 and 76**, that's exactly what's happening (**Figure 115**):

```
49   result = mw_COM_IWbemLocator(a1);
50   p_lpVtbl = &result->lpVtbl;
51   v23 = result;
52   if ( result )
53   {
54     v4 = mw_HeapAlloc(0x10u);
55     v36 = v4;
56     if ( !v4 )
57     {
58 LABEL_52:
59       mw_w_string_length((_BYTE **)&v30, -2);
60       sub_1000D784((_BYTE **)&v23);
61       return (IWbemServices *)v4;
62     }
63     v32 = mw_w_decode_string_table_1(0x592u);   // select
64     v40 = mw_w_decode_string_table_1(0xBDBu);   // from
65     v5 = mw_lstrcatW(v32);
66     v30 = v5;
67     mw_w_string_length_0((_BYTE **)&v32);
68     mw_w_string_length_0((_BYTE **)&v40);
69     if ( !v5 )
70     {
71       mw_w_string_length(&v36, 0);
72       v4 = v36;
73       goto LABEL_52;
74     }
75     bstrString = SysAllocString(v5);
76     v40 = mw_w_decode_string_table_1(0x893u);   // WQL
77     v32 = SysAllocString(v40);
78     mw_w_string_length_0((_BYTE **)&v40);
79     v6 = bstrString;
80     v7 = (struct IWbemServicesVtbl **)*p_lpVtbl;
```

[Figure 115]

In addition, the return of **ExecQuery** is its **fifth argument** that has a **IEnumWbemClassObject **** type. Thus, you need to set it to **'IEnumWbemClassObject *'** (a reference already exists) and do the same for variable **v9** (**line 84**).

Soon the reader do it, the **IEnumWbemClassObject::Next** method also will appear in the code, which has the goal of getting one or more objects starting at the current position in the enumeration. Likewise, its **third argument** is type **IWbemClassObject **** , but as there's already an existing reference, so change its type to **'IWbemClassObject *'**.

Likely the **GetNames method**, which **retrieves the same of the properties in the object**, will **appear on line 99**. Remember that the **GetNames method** makes possible each property to be accessed by **using IWbemClassObject::Get method**. The **Figure 116** shows the result of this manipulation:

```
81    v8 = 0;
82    if ( !(*v7)->ExecQuery((IWbemServices *)v7, v32, bstrString, 0, 0, &ppEnum) )
83    {
84      v9 = ppEnum;
85      v38 = 0;
86      v40 = 0;
87      if ( ppEnum )
88      {
89        while ( 1 )
90        {
91          if ( v9->lpVtbl->Next(v9, 60000, 1, &v38, (ULONG *)&puReturned) )
92            goto LABEL_46;
93          plLbound = 0;
94          plUbound = 0;
95          rgIndices[0] = 0;
96          psa = 0;
97          if ( !puReturned )
98            goto LABEL_46;
99          if ( v38->lpVtbl->GetNames(v38, 0, 64, 0, &psa) < 0 )
100         {
101           v38->lpVtbl->Release(v38);
102           goto LABEL_41;
103         }
```

[Figure 116]

I'm going to return to **COM (Component Object Model)** topic in next articles, so even you didn't understand details of this subroutine here, we have a better opportunity to talk about it in a near future.

Changing the focus to another piece of code, Qakbot's authors seems having used the same scheme (**key → SHA1 → derived key → RC4**) to encrypt/decrypt Registry entries within **sub_1000A23A subroutine**, which is called several times (check its cross-references):

```
43  mw_w_mersenne_random(v15, (int)&data_content[v8 + 6], (unsigned int)v24[0]);
44  v23 = a1;
45  v12 = v21;
46  v24[0] = *(_BYTE **)(v21 + 264);
47  mw_SHA1(8, (char *)&v23, (int)sha_buffer);
48  mw_RC4_init(20, (int)sha_buffer, (int)key_stream);
49  mw_RC4_PRGA((int)data_length, (int)data_content, (int)key_stream);
50  v13 = 0;
51  v24[0] = sub_10009CBA((unsigned int *)v12);
52  if ( v24[0] )
53  {
54    sub_1000994C(v17, crc32_value, 0x10u);
55    if ( ptr_struct_iat_advapi32->ptr_RegOpenKeyExA(*(HKEY *)(v12 + 268), v24[0], 0, 2, (PHKEY)&a3) )
56    {
57      v13 = -3;
58    }
59    else
60    {
61      if ( ptr_struct_iat_advapi32->ptr_RegSetValueExA((HKEY)a3, v17, 0, 3, data_content, (DWORD)data_length) )
62        v13 = -4;
63      ptr_struct_iat_advapi32->ptr_RegCloseKey((HKEY)a3);
64    }
65    mw_w_string_length(v24, -1);
66  }
67  mw_w_string_length(&v19, 0);
68  return v13;
69 }
```

[Figure 117]

Observe the **sub_10004C5A subroutine** shown below:

```
99  ptr_heap_RegKey_manipulation = mw_RegKey_manipulation(
100                                (const CHAR *)ptr_addr_SID_struct,
101                                (const WCHAR *)ptr_memory_memset,
102                                (int)var_ComputerVolumeInformation,
103                                &arg_lpData,
104                                &var_string_from_table);// SOFTWARE\Microsoft\Microsoft Antimalware\Exclusions\Paths
105  ptr_heap_RegKey_manipulation_ref = (int)ptr_heap_RegKey_manipulation;
106  if ( ptr_heap_RegKey_manipulation )
107  {
108    mw_w_RC4_Reg((_BYTE *)0xE, (int)ptr_heap_RegKey_manipulation, byte_1001B9D2);
109    mw_w_RC4_Reg_0(var_lstrcatW_1);
110    v29 = 0;
111    var_string_from_table = *(_DWORD *)a3;
112    mw_RC4_Reg((_BYTE *)0xB, ptr_heap_RegKey_manipulation_ref, &var_string_from_table, 8, 2);
113    v17 = *(unsigned int **)(a3 + 16);
114    if ( v17 )
115      mw_SHA1_RC4_0(ptr_heap_RegKey_manipulation_ref, v17);
116    v18 = *(unsigned int **)(a3 + 12);
117    if ( v18 )
118      mw_SHA1_RC4_0(ptr_heap_RegKey_manipulation_ref, v18);
```

[Figure 118]

We can realize that the initial key that's derived by the **SHA1 hash function** comes from a composition of the computer name, account and volume information. Furthermore, it seems that malware is concerned in manipulating the key "***SOFTWARE\Microsoft\Microsoft Antimalware\Exclusions\Paths***" to, eventually, include itself as an authorized anti-malware tool and stay out the radar.

The **sub_10002E98 subroutine** (renamed as **mw_GetKeyBoardLayoutList**) checks for input locale identifiers to check whether the installed keyboard is Russian, Belarusian, Tajik, Ukrainian, and so on:

```
1  int mw_GetKeyboardLayoutList()
2  {
3    int layout_present; // esi
4    unsigned int number_input_locales; // ebx
5    unsigned int counter; // ecx
6    unsigned int limit_predefined_list; // edx
7    __int16 lpList[128]; // [esp+8h] [ebp-118h] BYREF
8    __int16 v6[12]; // [esp+108h] [ebp-18h]
9
10   v6[0] = 0x19;
11   layout_present = 0;
12   v6[1] = 0x23;
13   v6[2] = 0x3F;
14   v6[3] = 0x2C;
15   v6[4] = 0x2B;
16   v6[5] = 0x37;
17   v6[7] = 0x43;
18   v6[8] = 0x28;
19   v6[9] = 0x42;
20   v6[10] = 0x22;
21   v6[11] = 0x1A;
22   v6[6] = 0x40;
23   number_input_locales = ptr_struct_iat_user32->ptr_GetKeyboardLayoutList(0x40, (HKL *)lpList);
24   for ( counter = 0; counter < number_input_locales; ++counter )
25   {
26     for ( limit_predefined_list = 0; limit_predefined_list < 12; ++limit_predefined_list )
27     {
28       if ( (lpList[2 * counter] & 0x3FF) == v6[limit_predefined_list] )
29         layout_present = 1;
30     }
31   }
32   return layout_present;
33 }
```

[Figure 119]

Another curious point are the references done by getting a handle for two Avast protection's DLLs (**aswhooka.dll** and **aswhookx.dll**) inside the **sub_10004FB9**:

```
35  if ( (ptr_struct_sysinfo_ref->sysinfo_42.field_20 & 0x82) != 0 )
36  {
37    ptr_string_1 = mw_w_decode_string_table();  // aswhooka.dll
38    ptr_string_1_ref = ptr_string_1;
39    ptr_string_2 = mw_w_decode_string_table();  // aswhookx.dll
40    ptr_string_2_ref = ptr_string_2;
41    v20 = (unsigned int)ptr_string_2;
42    if ( ptr_string_1 )
43    {
44      if ( ptr_string_2 )
45      {
46        if ( GetModuleHandleA(ptr_string_1) || GetModuleHandleA(ptr_string_2_ref) )
47          Format = (wchar_t *)1;
48        mw_ww_string_length(&ptr_string_1_ref);
49        mw_ww_string_length((_BYTE **)&v20);
50        if ( Format )
51          return -1;
```

[Figure 120]

At the same subroutine, it establishes persistence by using the string ‘ *“C:\Windows\system32\schtasks.exe” /Create /RU “NT AUTHORITY\SYSTEM /tn <random name> /tr “regsvr32.exe -s …” ’* and soon afterwards it creates the respective process (**sub_1000AAC1**), as shown in the **Figure 121** and **Figure 122** respectively:

```
128       ptr_buffer = (WCHAR *)mw_HeapAlloc(0x1000u);
129       ptr_buffer_ref = ptr_buffer;
130       if ( ptr_buffer )
131       {
132         Format = mw_w_decode_string_table_2(0x13Eu);// "%s\system32\schtasks.exe" /Create /RU
133                                                     // "NT AUTHORITY\SYSTEM" /tn %s /tr "%s" /SC ONCE /Z /ST %02u:%02u /ET %02u:%02u
134         mw_mt_manipulation((int)Buffer, 2, 7, 10, &ptr_struct_sysinfo_ref->sysinfo_10.state);
135         ptr_string_1_ref = sub_1000628A(      // regsvr32.exe -s
136                              (int)&ptr_struct_sysinfo_ref->sysinfo_3.lpFilename,
137                              1,
138                              ptr_struct_sysinfo_ref->sysinfo_1.field_3);
139         if ( ptr_string_1_ref )
140         {
141           mw_w_str_length_wchar(
142             ptr_buffer,
143             0x1000u,
144             Format,
145             &ptr_struct_sysinfo_ref->sysinfo_27.lpBuffer,
146             Buffer,
147             ptr_string_1_ref,
148             (unsigned __int16)v22,
149             v27 % 0x3C,
150             (unsigned __int16)v20,
151             v21 % 0x3C);
152           mw_w_string_length_0((_BYTE **)&Format);
153           mw_CreateProcess((int)ptr_buffer, 0, 3000, 1);
```

[Figure 121]

```
22  if ( !ptr_struct_iat_kernel32->ptr_CreateProcessW(
23          0,
24          (LPWSTR)ptr_buffer_HeapAlloc,
25          0,
26          0,
27          0,
28          dwCreationFlags != 0 ? CREATE_NO_WINDOW : 0,
29          0,
30          0,
31          (LPSTARTUPINFOW)memory_startupinfow,
32          (LPPROCESS_INFORMATION)&hProcess) )
33    return 0;
34  if ( lpExitCode && (ptr_struct_iat_kernel32->ptr_WaitForSingleObject(hProcess, dwMilliseconds) & 0x80000000) == 0 )
35    GetExitCodeProcess(hProcess, lpExitCode);
36  ptr_struct_iat_kernel32->ptr_CloseHandle(var_handle);
37  ptr_struct_iat_kernel32->ptr_CloseHandle(hProcess);
38  return v4;
39 }
```

[Figure 122]

7. Conclusion

There're many other quite fascinating lines of codes to be analyzed (there're 510 functions!) and It would be possible to extend this article in dozens of further pages. For example, I could have analyzed C2 communication, other code injection techniques and lot of other aspects of the code, but I think it's enough for now.

My goal, as I had already mentioned in the first article, is offering a kind of review for reverse engineers to learn something new and have a kind of guideline to follow and search when it's necessary. Of course, next articles might not be so extensive as this one, but I will try to cover different aspects and topics as possible.

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

8. 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 (without 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 continue writing this series of articles this year and beyond.

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

Just in case you want to keep in touch:

- **Twitter: @ale_sp_brazil**
- **LinkedIn:** https://www.linkedin.com/in/aleborges
- **Blog:** https://exploitreversing.com

Keep reversing and I see you at next time!

Alexandre Borges