Malware Analysis Series (MAS): Article 5

author: Alexandre Borges
date: September/14/2022 | rev: A

0. Quote

"Things go wrong. The odds catch up. Probability is like gravity: you cannot negotiate with gravity".
(Det. James 'Sonny' Crockett | Miami Vice movie - 2006)

1. Introduction

Welcome to the **fifth article** of ***Malware Analysis*** **Series (MAS).** If readers haven't read the first four articles yet, all of them are available on the following links:

- **MAS_1**: https://exploitreversing.com/2021/12/03/malware-analysis-series-mas-article-1/
- **MAS_2:** https://exploitreversing.com/2022/02/03/malware-analysis-series-mas-article-2/
- **MAS_3:** https://exploitreversing.com/2022/05/05/malware-analysis-series-mas-article-3/
- **MAS_4:** https://exploitreversing.com/2022/05/12/malware-analysis-series-mas-article-4/

We have covered many different topics so far and, in a general way, the main goal continues being to present fundamental concepts, and applied and practical approaches on malware analysis to help readers to build up all necessary skills and move forward on their own analysis and learning path. As many readers know, it isn't my intention to propose hard samples because it'd be completely useless for an effective learning and daily work, and in my opinion, would also be an unnecessary showing off behavior.

In the first four articles we presented, explained, applied, and learned techniques related to:

- malware profiling and main tools used to get fundamental information on malware samples.
- basic obfuscation (Control Flow Flattening) and anti-forensics concepts.
- main evidence's items used to recognize a packed code and their associated challenges.
- usual and well-known unpacking tricks and how to fix IAT (Import Address Table) of extracted binaries from memory.
- code injection review and managing DLL / API hashing resolution cases.
- unpacking methods and APIs to setup breakpoint over the unpacking procedure.
- how to write C2 data configuration extractors using Python, IDC, and IDA Python.
- how to write string de-deobfuscating scripts.
- relevant IDA Pro plugins commonly used over the analysis.
- .NET reflection concepts, internals unpacking and de-obfuscation.
- parsing and recognizing PE structures using IDA Pro.
- handling C++ structures through enumeration, local types, and structures.

Therefore, we're ready to move forward and, in this article, we'll be analyzing few features of Bumblebee malware. As I've already mentioned previously, we don't have any intention to dissecting it, but only some interesting aspects of it.

2. 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 happy, thankful, and fortunate I feel myself in receiving their help. Although it's already much more than I would be able to dream in receiving, last June/2022 **Ilfak** and **Hex-Rays** once again kindly agreed in helping me by providing new licenses of IDA Pro for other platforms due to new series I've just started writing and planned to release as soon as possible. Personally, all words from Ilfak expressing his trust and praise about this series of articles until now are the most important for me.

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

3. Environment Setup

This article has a lab setup using the following environment:

- **Windows 11 running in a virtual machine.** You're able to download a **virtual machine for VMware, Hyper-V, VirtualBox or Parallels from Microsoft** on: https://developer.microsoft.com/en-us/windows/downloads/virtual-machines/. If you already have a valid license for Windows 11, so you can download the **ISO file** from: https://www.microsoft.com/software-download/windows11

- **IDA Pro or IDA Home version (@HexRaysSA):** https://hex-rays.com/ida-pro/ . I'll be using IDA Pro version 8.x and, mainly, the Hex-Rays Decompiler in this article.
- **x64dbg(@x64dbg):** https://x64dbg.com/
- **PEBear (@hasherezade):** https://github.com/hasherezade/pe-bear-releases
- **DiE (from @horsicq):** https://github.com/horsicq/DIE-engine/releases
- **CFF Explorer:** https://ntcore.com/?page_id=388
- **HxD editor:** https://mh-nexus.de/en/hxd/
- **Resource Hacker:** http://www.angusj.com/resourcehacker/
- **Malwoverview:** https://github.com/alexandreborges/malwoverview
- **Floss: pip install -U flare-floss |** https://github.com/mandiant/flare-floss/releases/tag/v2.0.0
- **Capa: pip install -U flare-capa |** https://github.com/mandiant/capa/releases

To get further information about lab configuration, I recommend readers to reserve some time to review the **first and second articles of this series**. Both articles present concepts about the unpacking topic and other details that, eventually, could be useful.

4. References

I could find several articles analyzing **Bumblebee** and, although I haven't had the opportunity to read them (my time is incredibly short), I recommend readers to do it because they were written by excellent security researchers and companies, which covered and analyzed several aspects of the same family, and readers can learn what's more appropriate for their work. The list below doesn't have any preferred order:

- https://blog.google/threat-analysis-group/exposing-initial-access-broker-ties-conti/
- https://blog.sekoia.io/bumblebee-a-new-trendy-loader-for-initial-access-brokers/
- https://blog.cyble.com/2022/06/07/bumblebee-loader-on-the-rise/
- https://symantec-enterprise-blogs.security.com/blogs/threat-intelligence/bumblebee-loader-cybercrime
- https://research.nccgroup.com/2022/04/29/adventures-in-the-land-of-bumblebee-a-new-malicious-loader/
- https://www.cynet.com/blog/orion-threat-alert-flight-of-the-bumblebee/
- https://www.proofpoint.com/us/blog/threat-insight/bumblebee-is-still-transforming

5. Recommended Blogs and Websites

There're many excellent cyber security researchers keeping blogs and writing amazing articles related to reverse might be interested in reading and following their contents. I tried googling to make a quick and sorted list in **alphabetical order** as follow below:

- https://hasherezade.github.io/articles.html **(by Aleksandra Doniec: @hasherezade)**
- https://malwareunicorn.org/#/workshops **(by Amanda Rousseau: @malwareunicorn)**
- https://captmeelo.com/ **(by Capt. Meelo: @CaptMeelo)**
- https://csandker.io/ **(by Carsten Sandker: @0xcsandker)**
- https://chuongdong.com/ **(by Chuong Dong: @cPeterr)**
- https://elis531989.medium.com/ **(by Eli Salem: @elisalem9)**
- http://0xeb.net/ **(by Elias Bachaalany: @0xeb)**
- https://cyb3rops.medium.com/ **(by Florian Roth: @cyb3rops)**
- https://hex-rays.com/blog/ **(by Hex-Rays: @HexRaysSA)**
- https://github.com/Dump-GUY/Malware-analysis-and-Reverse-engineering **(by Jiří Vinopal: @vinopaljiri)**
- https://kienmanowar.wordpress.com/ **(by Kien Tran Trung: @kienbigmummy)**
- https://www.inversecos.com/ **(by Lina Lau: @inversecos)**
- https://maldroid.github.io/ **(by Łukasz Siewierski: @maldr0id)**
- https://voidsec.com/member/voidsec/ **(by Paolo Stagno: @Void_Sec)**
- https://www.ragingrock.com/AndroidAppRE/ **(by Maddie Stone: @maddiestone)**
- https://azeria-labs.com/writing-arm-assembly-part-1/ **(by Maria Markstedter: @Fox0x01)**
- https://github.com/mnrkbys **(by Minoru Kobayashi: @unkn0wnbit)**
- https://repnz.github.io/ **(by Ori Damari: @0xrepnz)**
- https://windows-internals.com/author/yarden/ **(by Yarden Shafir: @yarden_shafir)**

Certainly, there're several other excellent blogs explaining concepts and applied techniques about mentioned topics. I'll include these references as soon as I learn about them in next articles.

6. Gathering Information

We'll be examining the following sample:
57c4bdf0a644df4fd39f3d73d4570e6c88d8b7239ab4a395dba441ab15a5024f.

As usual, we should acquire initial information about our sample, which is available to download from **Malware Bazaar**:

```
remnux@remnux:~/malware/mas/mas_5$ malwoverview.py -b 1 -B 57c4bdf0a644df4fd39f3d73d4570e6c88d8b7239ab4a395dba441ab15a5024f
 -o 0

                                       MALWARE BAZAAR REPORT
----------------------------------------------------------------------------------------------------

sha256_hash:  57c4bdf0a644df4fd39f3d73d4570e6c88d8b7239ab4a395dba441ab15a5024f
sha1_hash:    36bd693fa90b0438d6366414a202fcc9eb894c87
md5_hash:     95ca84ee56bade777c6e867a1f6cd78a
first_seen:   2022-06-15 13:44:12
file_name:    Scanned-Documents-0629.img
file_size:    3276800 bytes
file_type:    img
mime_type:    application/octet-stream
country:      US
tlsh:         T12CE518DD923656CFEC1767B71DC43E950CD2446B8F164EA854BE2208CA353F836A42AF
reporter:     k3dg3
tags:         146l Bumblebee TA579
remnux@remnux:~/malware/mas/mas_5$
remnux@remnux:~/malware/mas/mas_5$ malwoverview.py -b 5 -B 57c4bdf0a644df4fd39f3d73d4570e6c88d8b7239ab4a395dba441ab15a5024f
 -o 0

                                       MALWARE BAZAAR REPORT
----------------------------------------------------------------------------------------------------

SAMPLE SAVED!
```

[Figure 1] Check and download the binary from Malware Bazaar

Readers can unzip (using **7z e <zip file>**) the malware sample and there will be a file with **.img extension**. Afterwards, it's quite easy to use the **7z** command to "unpack" this .img file and we're going to find the following files:

```
remnux@remnux:~/malware/mas/mas_5/unzipped$ file *
inf.bat:                    ASCII text, with no line terminators
information.dll:            PE32+ executable (DLL) (GUI) x86-64, for MS Windows
ScannedDocuments-0622.lnk: MS Windows shortcut, Item id list present, Points to a file or direct
ory, Has Relative path, Has command line arguments, Icon number=153, Archive, ctime=Mon Dec 27 0
2:30:39 2021, mtime=Tue Jun 14 18:42:57 2022, atime=Mon Dec 27 02:30:39 2021, length=289792, win
dow=hide
```

[Figure 2] Unzipped files from .img file

Likely, the best approach is to examine them in the following order: the .bat file, the link file and, finally, the DLL binary.

```
remnux@remnux:~/malware/mas/mas_5/unzipped$ more inf.bat
start rundll32 information.dll,hKOgtkmCis
remnux@remnux:~/malware/mas/mas_5/unzipped$
remnux@remnux:~/malware/mas/mas_5/unzipped$ pip install LnkParse3
Defaulting to user installation because normal site-packages is not writeable
Requirement already satisfied: LnkParse3 in /home/remnux/.local/lib/python3.9/site-packages (1.2.0)
remnux@remnux:~/malware/mas/mas_5/unzipped$ lnkparse ScannedDocuments-0622.lnk
Windows Shortcut Information:
   Link CLSID: 00021401-0000-0000-C000-000000000046
   Link Flags: HasTargetIDList | HasLinkInfo | HasRelativePath | HasArguments | HasIconLocation | IsUnicode
 | EnableTargetMetadata - (524523)
   File Flags: FILE_ATTRIBUTE_ARCHIVE - (32)

   Creation Timestamp: 2021-12-26 21:30:39.886936+00:00
   Modified Timestamp: 2021-12-26 21:30:39.902558+00:00
   Accessed Timestamp: 2022-06-14 13:42:57.855901+00:00

   Icon Index: 153
   Window Style: SW_SHOWNORMAL
   HotKey: UNSET - UNSET {0x0000}

   TARGETS:
      Index: 78
      ITEMS:
         Root Folder
            Sort index: My Computer
            Guid: 20D04FE0-3AEA-1069-A2D8-08002B30309D
         Volume Item
            Flags: 0xf
            Data: None
```

[Figure 3] Checking the inf.bat content and decoding the .lnk file

```
   DATA
      Relative path: ..\Windows\System32\cmd.exe
      Command line arguments: /c inf.bat
      Icon location: %systemroot%\system32\imageres.dll

   EXTRA BLOCKS:
      SPECIAL_FOLDER_LOCATION_BLOCK
         Special folder id: 37
      KNOWN_FOLDER_LOCATION_BLOCK
         Known folder id: 1AC14E77-02E7-4E5D-B744-2EB1AE5198B7
      DISTRIBUTED_LINK_TRACKER_BLOCK
         Length: 88
         Version: 0
         Machine identifier: desktop-i8bn9qk
         Droid volume identifier: D67B2F6A-E9C7-482D-8540-9F20B8BB8171
         Droid file identifier: E089BE25-72EC-11EC-BEDD-B263FDAFB6EF
         Birth droid volume identifier: D67B2F6A-E9C7-482D-8540-9F20B8BB8171
         Birth droid file identifier: E089BE25-72EC-11EC-BEDD-B263FDAFB6EF
      METADATA_PROPERTIES_BLOCK
         Version: 0x53505331
         Format id: DABD30ED-0043-4789-A7F8-D013A4736622
```

[Figure 4] Decoding the .lnk file – truncated output for saving space

In few words, we learned from figures above that:

a. the order of execution is: **ScannedDocuments-0622.lnk → inf.bat → information.dll**
b. the **DLL is a 64-bit binary** and one of its exported functions named **hKOgtkmCis** is executed.

Thus, it's clear for us that we must analyze the DLL (this time is 64-bit) to understand what the threat does. However, before proceeding, let's collect further information that could, eventually, help us:

```
remnux@remnux:~/malware/mas/mas_5/unzipped$ malwoverview.py -v 2 -V information.dll -o 0

MD5 hash:             c9216484a6371b055705ec5f4098ab01
SHA1 hash:            a13903e50408e11996159fba5f7deab1e73e8f08
SHA256 hash:          fed9bc8df9141f8f8f7a9203bc26b5b22123c154702fcd625379f2f7ecd31cb2

Malicious:            42
Undetected:           25

AV Report:
                      Avast:          Win64:DropperX-gen [Drp]
                      Avira:          TR/Kryptik.tfgwo
                      BitDefender:    Trojan.GenericKD.39810172
                      DrWeb:          CLEAN
                      Emsisoft:       Trojan.GenericKD.39810172 (B)
                      ESET-NOD32:     a variant of Win64/Kryptik.DER
                      F-Secure:       CLEAN
                      FireEye:        Trojan.GenericKD.39810172
                      Fortinet:       W64/GenKryptik.FUZQ!tr
                      Kaspersky:      Trojan-Dropper.Win64.BumbleBee.dmv
                      McAfee:         Artemis!C9216484A637
                      Microsoft:      Trojan:Win64/BumbleBee.BE!MTB
                      Panda:          Trj/Chgt.AB
                      Sophos:         Mal/Generic-S + Troj/Bumble-D
                      Symantec:       Trojan.Gen.MBT
                      TrendMicro:     Trojan.Win64.BUMBLELOADER.YXCFOZ
                      ZoneAlarm:      CLEAN

Overlay:              NO
```

[Figure 5] Verifying the extracted DLL against Virus Total

```
remnux@remnux:~/malware/mas/mas_5/unzipped$ malwoverview.py -x 1 -X fed9bc8df9141f8f8f7a9203bc26b5b22123c
154702fcd625379f2f7ecd31cb2 -o 0

                                          TRIAGE OVERVIEW REPORT

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

id:         220616-tng2tsfhfl
status:     reported
kind:       file
filename:   information.dll
submitted:  2022-06-16T16:12:03Z
completed:  2022-06-16T16:14:40Z
------------------------------------------------------------------------------------------
id:         220615-qym42shban
status:     reported
kind:       file
filename:   TA579_20220614.zip
submitted:  2022-06-15T13:40:17Z
completed:  2022-06-15T13:43:46Z
------------------------------------------------------------------------------------------
next:       2022-06-15T13:40:17.376115Z
remnux@remnux:~/malware/mas/mas_5/unzipped$
remnux@remnux:~/malware/mas/mas_5/unzipped$ malwoverview.py -x 2 -X 220616-tng2tsfhfl -o 0

                                          TRIAGE SEARCH REPORT

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

score:      10
extracted:
            botnet:    146l
            c2:
                       242.165.212.79:339
                       162.144.249.150:239
                       63.122.120.151:268
                       144.52.138.51:193
```

```
                  102.109.16.255:445
                  137.253.55.69:235
         family:  bumblebee
         key:     key
         value:   VcFFI2Rj6t15
         rule:    BumbleBee
         dumped:  memory/1160-54-0x00000000024E0000-0x00000000025F7000-memory.dmp
         resource: behavioral1/memory/1160-54-0x00000000024E0000-0x00000000025F7000-memory.dmp
         tasks:   behavioral1 behavioral2

id:       220616-tng2tsfhfl
target:   information.dll
size:     2057728
md5:      c9216484a6371b055705ec5f4098ab01
sha1:     a13903e50408e11996159fba5f7deab1e73e8f08
sha256:   fed9bc8df9141f8f8f7a9203bc26b5b22123c154702fcd625379f2f7ecd31cb2
completed: 2022-06-16T16:14:40Z
signatures:
         BumbleBee
         Enumerates VirtualBox registry keys
         Identifies VirtualBox via ACPI registry values (likely anti-VM)
         Looks for VirtualBox Guest Additions in registry
         Checks BIOS information in registry
         Identifies Wine through registry keys
         Suspicious behavior: EnumeratesProcesses

targets:
         family:  bumblebee
         iocs:
                  93.184.220.29
                  8.238.24.126
                  51.104.15.252
                  204.79.197.203
         md5:     c9216484a6371b055705ec5f4098ab01
         score:   10
         sha1:    a13903e50408e11996159fba5f7deab1e73e8f08
         sha256:  fed9bc8df9141f8f8f7a9203bc26b5b22123c154702fcd625379f2f7ecd31cb2
         size:    2057728bytes
         tags:
                  family:bumblebee
```

[Figure 6] Checking the extracted DLL against Triage

We've learned new information:

- The malware seems to be, in fact, **Bumblebee**.
- The malware has **anti-virtual machine techniques to detect VirtualBox and uses BIOS information probably to detect virtual machines**.
- **It enumerates processes**, but it could have different goals like **anti-debugging technique and code injection**, for example.
- There're many **C2 servers** which are likely store in **encrypted format** within the malware.
- **The botnet's name is: 146l**

From this point, we have possible tasks to accomplish like:

- Collecting further **information about the binary itself**.
- Checking **whether the malware is packed or not**.
- If it's packed, so we need to **unpack it**.
- **Writing a script to extract C2 information**.
- Collecting additional information about malware's features.

Using **PEBear**, we have:

Disasm: .text | General | DOS Hdr | Rich Hdr | File Hdr | Optional Hdr | Section Hdrs | Exports | Imports | Resources | Exception

Offset	Name	Func. Count	Bound?	OriginalFirstThun	TimeDateStamp	Forwarder	NameRVA	FirstThunk
102570	KERNEL32.dll	76	FALSE	103B98	0	0	103F46	18000

KERNEL32.dll [76 entries]

Call via	Name	Ordinal	Original Thunk	Thunk	Forwarder	Hint
18000	GetStdHandle	-	103E00	103E00	-	2C7
18008	GetCurrentDirectoryA	-	103E10	103E10	-	208
18010	CreateFileA	-	103E28	103E28	-	BA
18018	GetFileInformationByHandle	-	103E36	103E36	-	23E
18020	SetFileTime	-	103E54	103E54	-	50F
18028	CloseHandle	-	103E62	103E62	-	7F
18030	HeapAlloc	-	103E70	103E70	-	338
18038	GetProcessHeap	-	103E7C	103E7C	-	2A9
18040	GetCurrentProcessId	-	103E8E	103E8E	-	210
18048	ExitProcess	-	103EA4	103EA4	-	157

Disasm: .text | General | DOS Hdr | Rich Hdr | File Hdr | Optional Hdr | Section Hdrs | Exports | Imports | Resources

Offset	Name	Value	Meaning
102520	Characteristics	0	
102524	TimeDateStamp	62A87C33	Tuesday, 14.06.2022 12:16:51 UTC
102528	MajorVersion	0	
10252A	MinorVersion	0	
10252C	Name	103B52	bficra771ix.dll
102530	Base	1	
102534	NumberOfFunc...	1	

Exported Functions [1 entry]

Offset	Ordinal	Function RVA	Name RVA	Name	Forwarder
102548	1	1550	103B62	hKOgtkmCis	

[Figure 7] Checking Imported and Exported Functions

As expected, the binary has few imports **(only one DLL: kernel32.dll), so it's likely packed, and exports one function/routine** (the same present in the **.bat file**), which is the "**real entry point**" in this case.

To unpack this DLL, we can use **x64dbg** and try to setup the breakpoint on few functions like to investigate unpacked PE file in the memory (**self-injection** or **remote injection**). However, pay attention to one detail: it's recommended to **configure these breakpoints after debugger having hit the binary's entry point**:

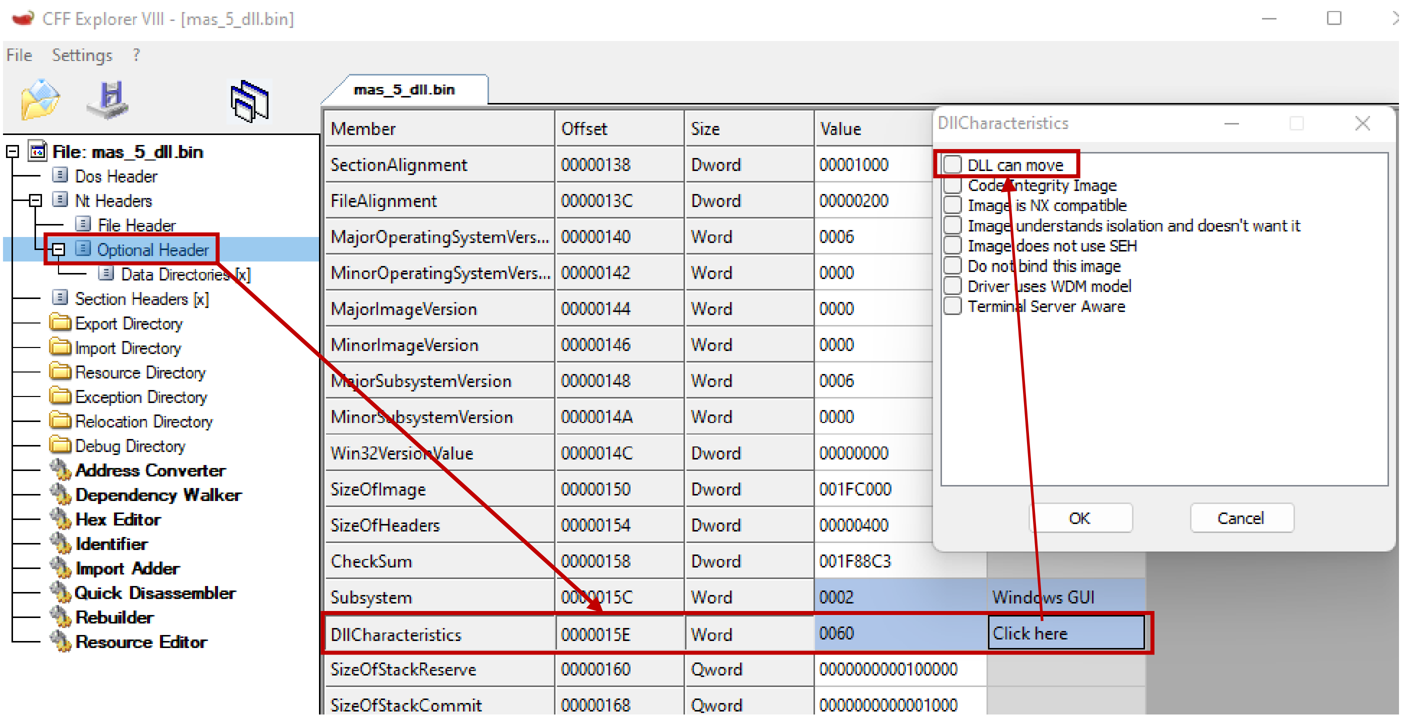
- **VirtualAlloc( )**
- **ResumeThread( )**

Before starting the unpacking procedure, there's a simple step that always helps during analysis (not necessarily in this binary), mainly when readers are analyzing a sample dynamically using debuggers:

disabling system or binary's memory randomization (also known as dynamic rebasing) to prevent the executable of being allocated in different memory addresses each time it's executed.

To accomplish this task readers could take three approaches:

- **Disabling memory randomization globally on the system:**
    - Creates a Registry's entry: **HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management → MoveImages=dword:00000000**
    - **Reboot** the system.
- **Disabling binary's memory randomization (possibility of executable in being allocated in different memory regions each time is called) using CFF Explorer:**



[Figure 8] Disabling DLL moving in the memory

- Disabling binary's **memory randomization** feature (**dynamic rebasing**) using **command line:**
    - Download and extract **setdllcharacteristics** tool (from **Didier Stevens -- @DidierStevens**): https://blog.didierstevens.com/my-software/#setdllcharacteristics
    - **setdllcharacteristics.exe -d <binary>**

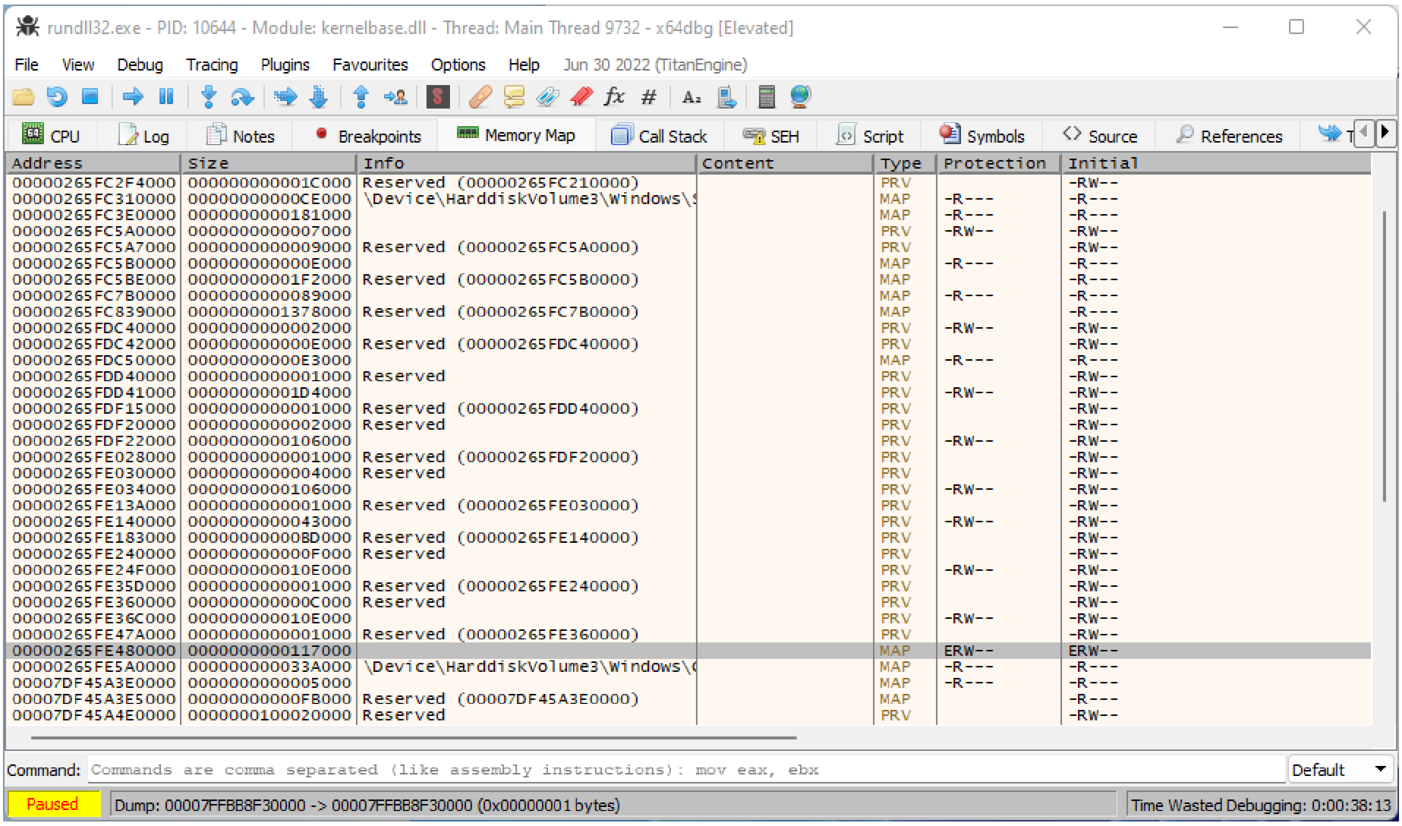
Usually, people prefer acting on the binary instead of the system, but it a personal decision of each one.

Remember from previous articles of this series, the recommended approach to **debug a DLL** is to use the **rundll32.exe (C:\Windows\System32**) and pass the DLL and the respective exported function (or ordinal number) as argument. Thus, **launch the x64dbg**, open the **rundll32.exe** into debugger and go to **File | Change Command Line** and alter its content as shown below:

- **"C:\Windows\System32\rundll32.exe" C:\Users\Administrator\Desktop\ARTICLES\MAS_5\mas_5_dll.bin,hKOgtkmCis**

Restart/reload the **x64dbg** session (**CTRL+F2**) and the debugger will stop at **System Breakpoint**. Run once (**F9**) and debugger will hit the **Entry BreakPoint**. Now you can configure breakpoints on the functions mentioned previously. If you want, after hitting the first break point, you can setup a breakpoint at beginning of exported function (**hKOgtkmCis**) **by following the standard procedure: CTRL+G** and enter **mas_5_dll.bin.hKOgtkmCis.**

Readers will realize that **they're not able to get anything by following the dump of data pointed by VirtualAlloc's returned address (RAX)**. Nonetheless, you're able to see a **ERW region** as shown below:



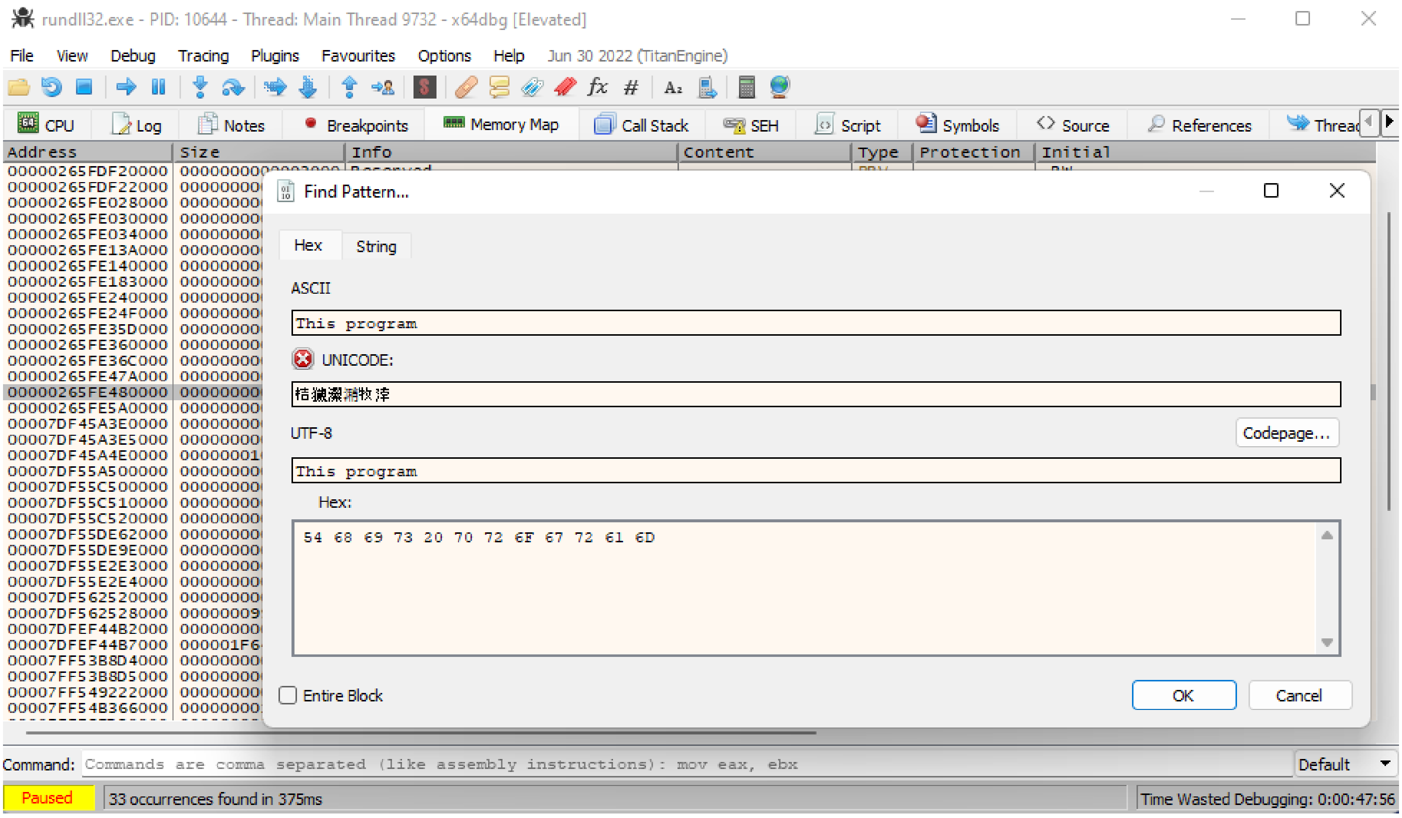
[Figure 9] x64dbg – unpacking the malware

In the figure above, I only visualized the **Memory Map view**, **searched for ERW sections** and quicky found the unpacked binary.

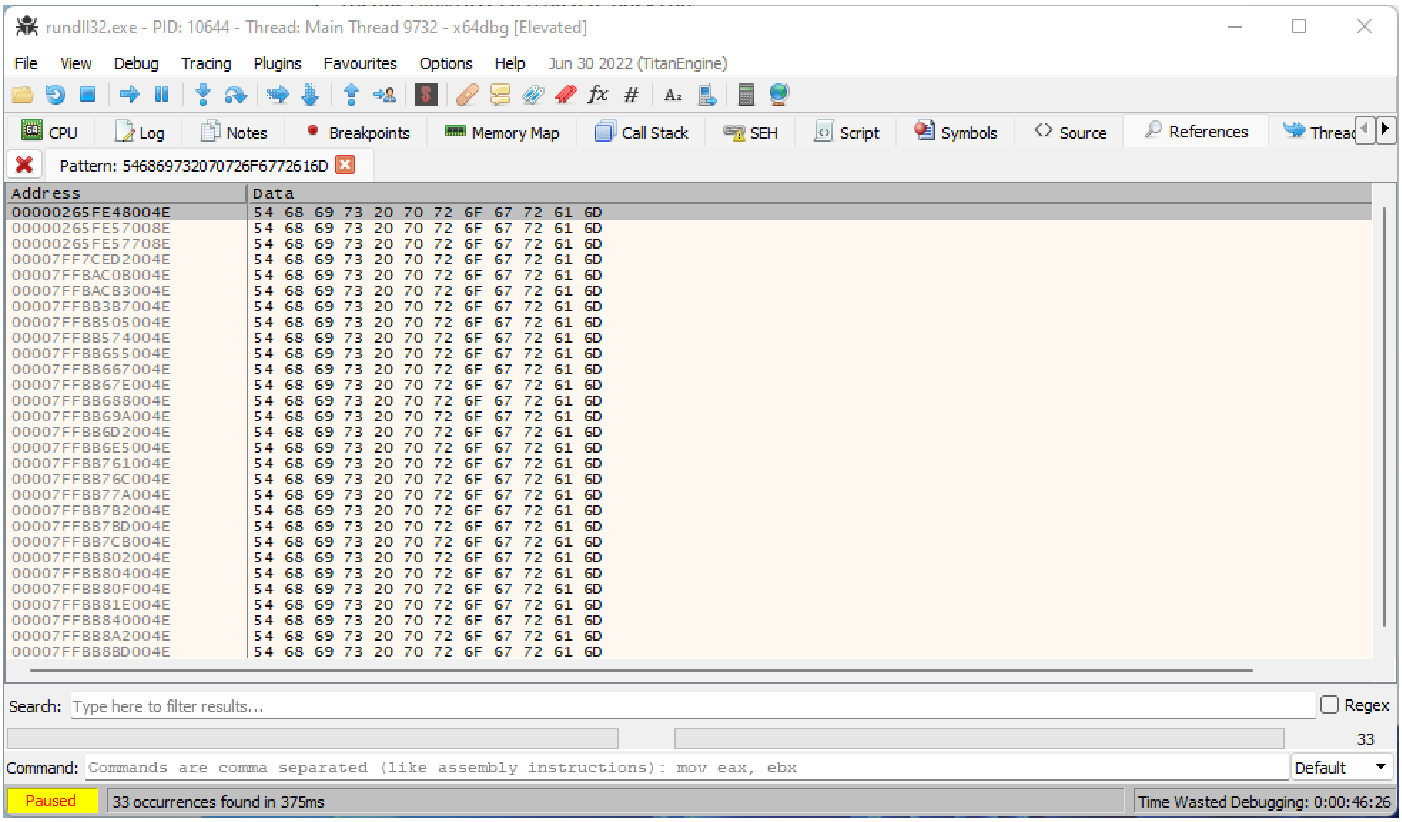
However, as explained in previous articles, there're multiples ways to find a PE executable on memory, and one of them is by **searching for a given string over all memory regions**.

To execute this operation, we can try the **"Find Pattern…"** option **(CTRL+B)** and enter a string that help us to find the PE executable. In this case I've tried using **"This program" string**.

Take care: **some malware threats wipe the PE header, so looking for this string might not work!**



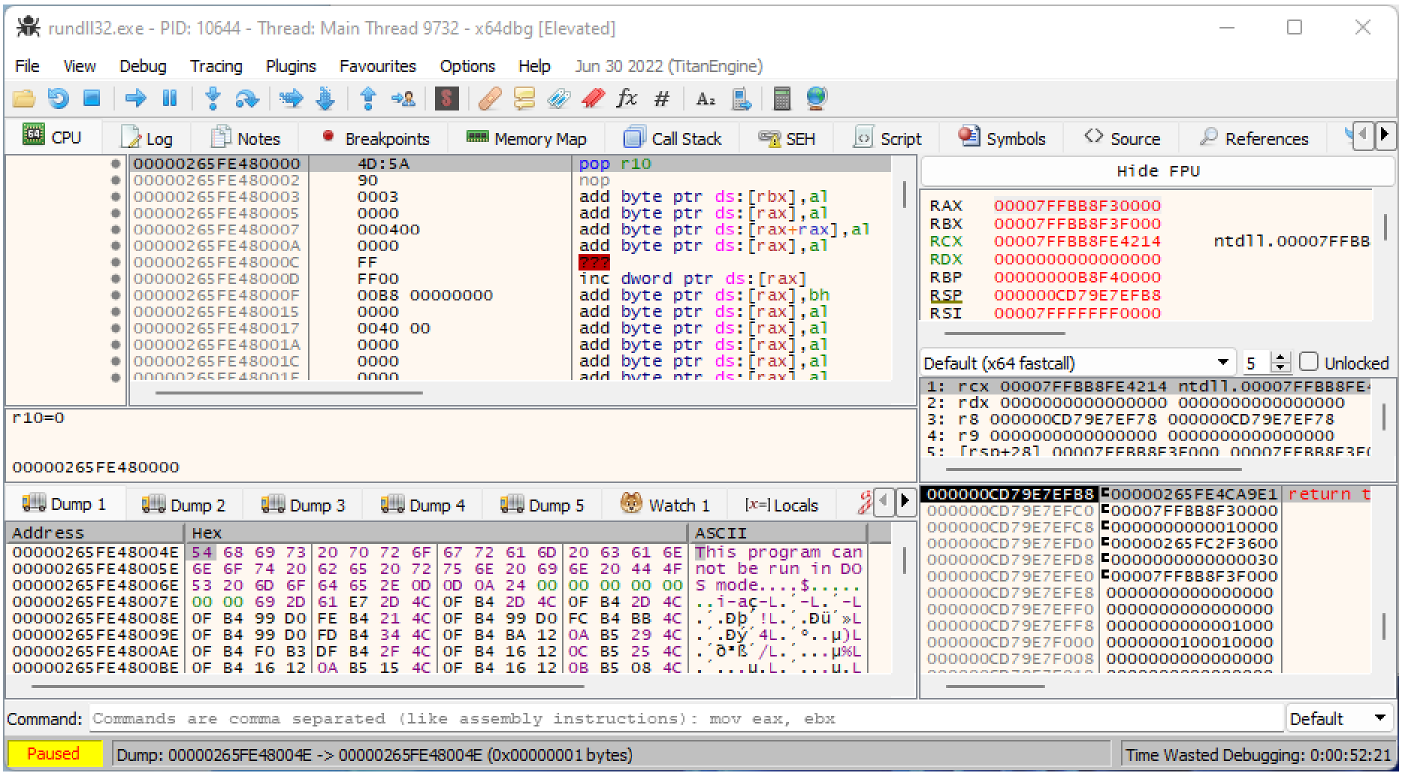
[Figure 10] x64dbg – searching strings



[Figure 11] x64dbg – results of our search

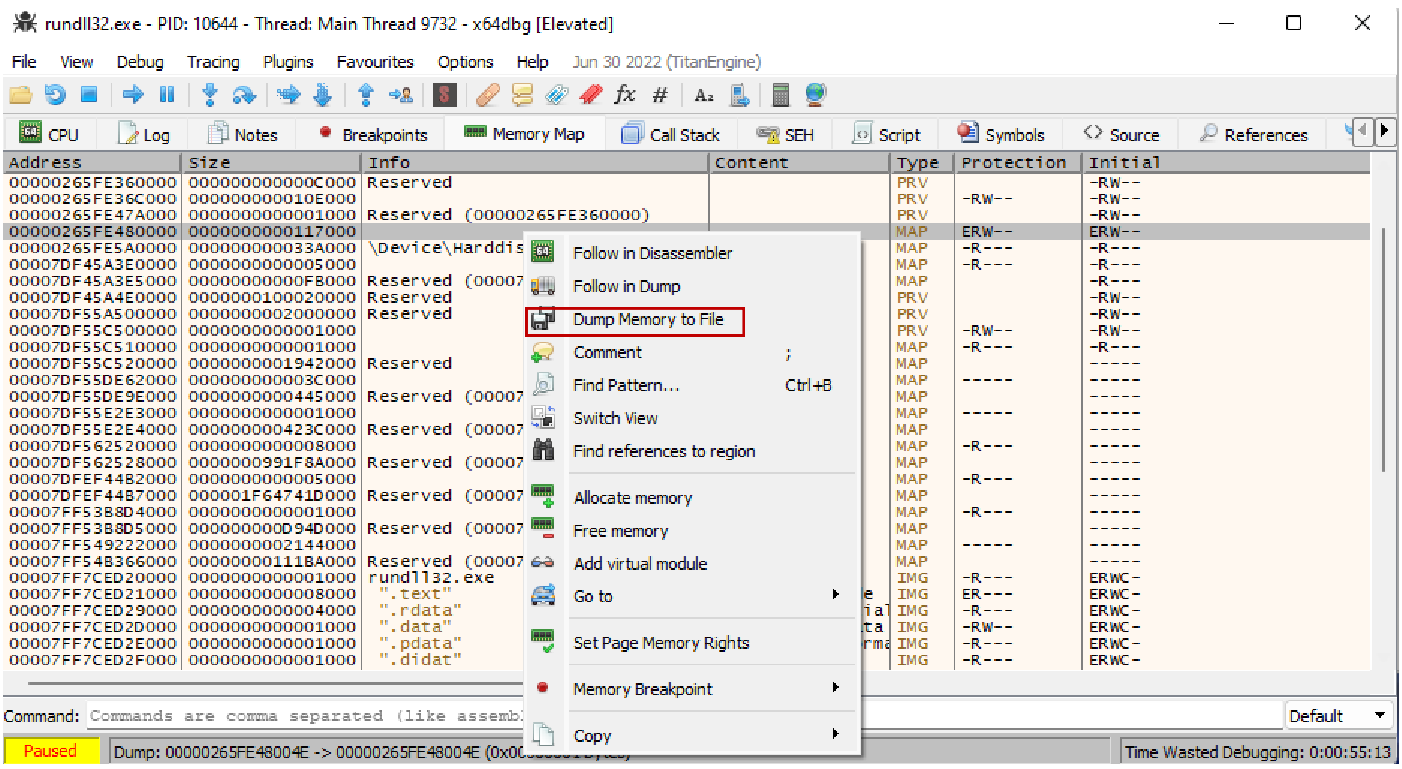
Many addresses have been found, but only the first three ones are not loaded DLLs.

We have to **right-click each one of the found addresses** and then choosing **Following in Dump**:



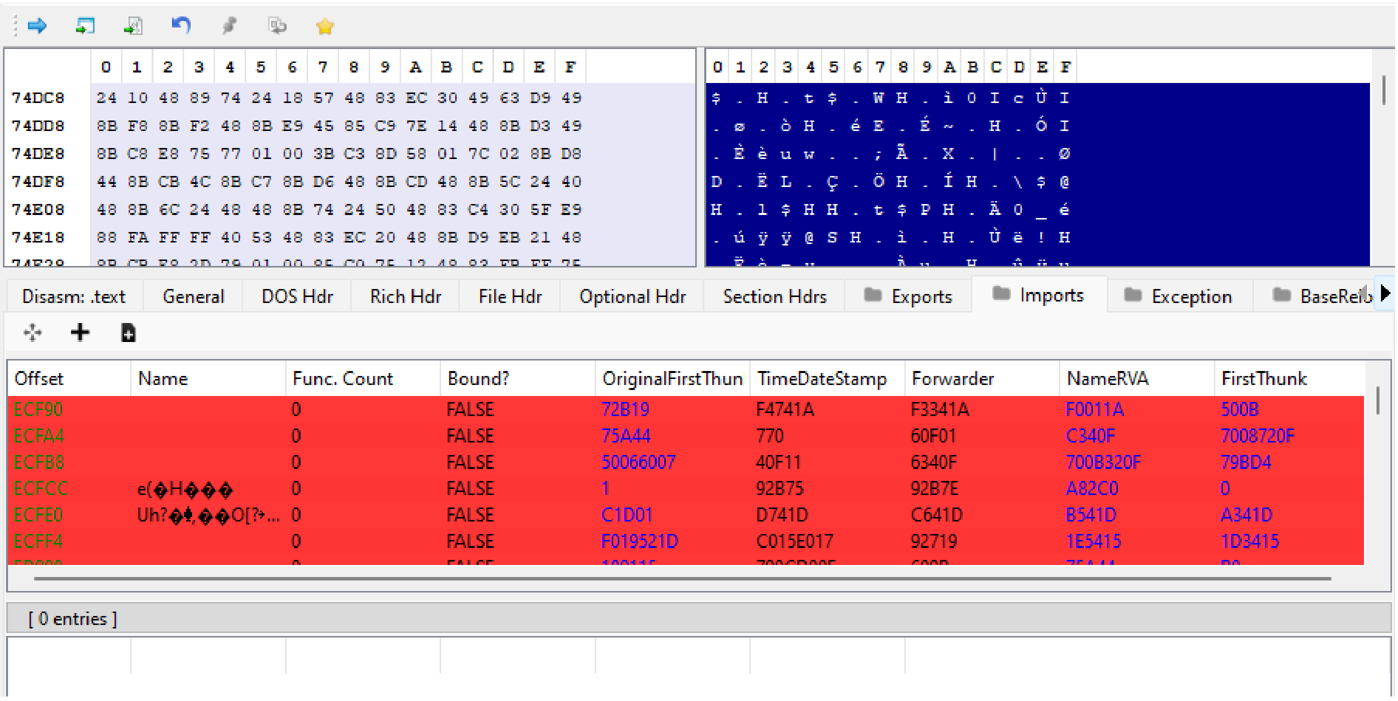
[Figure 12] x64dbg – unpacked PE binary

Once you're there, **right click on Dump area → Follow in Memory Map → right click → Dump Memory to File:**



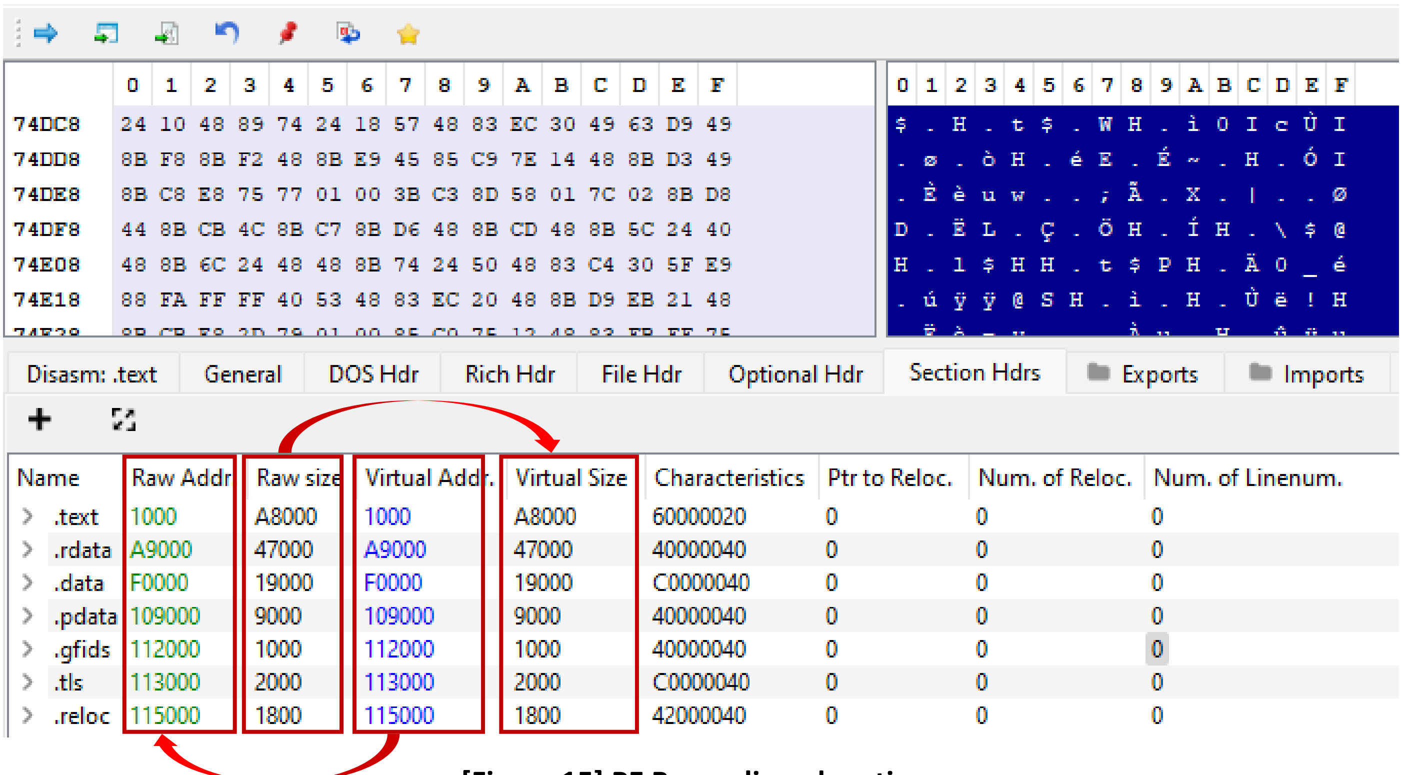
[Figure 13] x64dbg – saving unpacked PE binary

Open the extracted binary onto **PEBear** and readers will see a binary with **IAT destroyed**, but it isn't any problem. As the unpacked malware has been dumped from memory, so it's in **"mapped format"** and its **respective addresses represent the memory addresses** and **not the raw (on disk) addresses**:

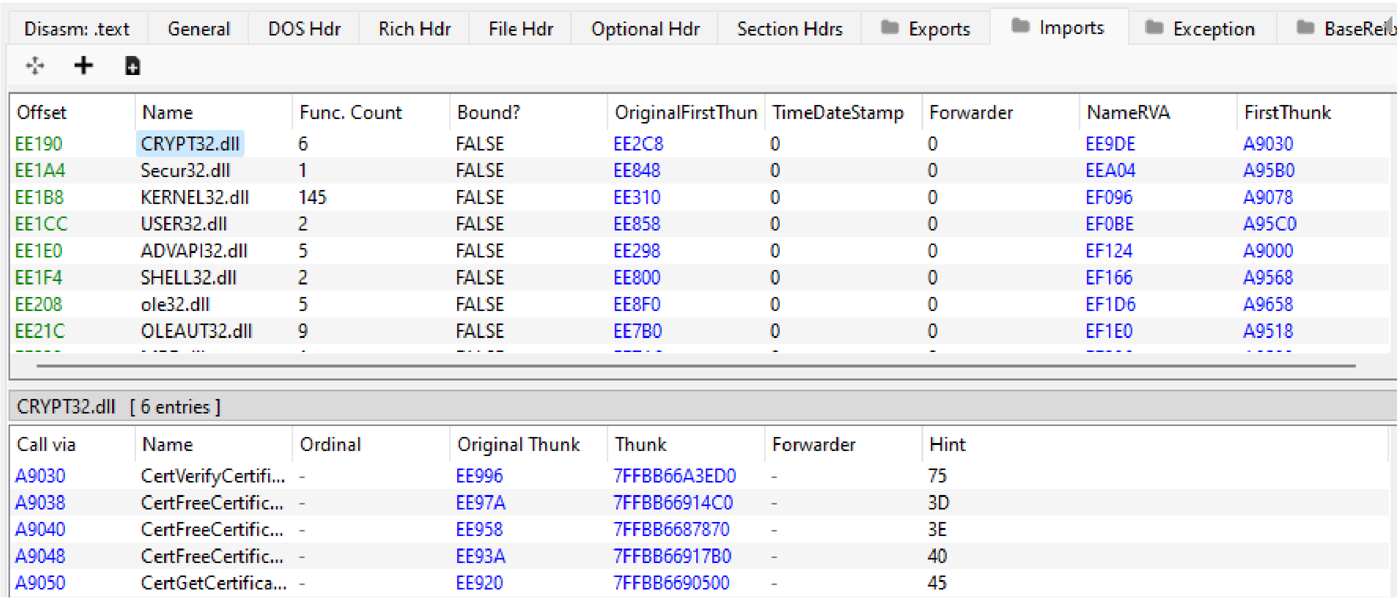


[Figure 14] PE Bear: extracted binary presenting issues with Imports

To fix it, readers must go to **"Section Hrds" tab** and perform the following operations for each section: 1. Make **Raw Addr.** equal to its respective **Virtual Address**; 2. Calculates the **Raw Size** of each section **(next section address minus current section address**) and **copy the resulting value** to **Virtual Size**:



[Figure 15] PE Bear: aligned sections



Disasm: .text | General | DOS Hdr | Rich Hdr | File Hdr | Optional Hdr | Section Hdrs | Exports | Imports | Exception | BaseReloc

Offset	Name	Func. Count	Bound?	OriginalFirstThun	TimeDateStamp	Forwarder	NameRVA	FirstThunk
EE190	CRYPT32.dll	6	FALSE	EE2C8	0	0	EE9DE	A9030
EE1A4	Secur32.dll	1	FALSE	EE848	0	0	EEA04	A95B0
EE1B8	KERNEL32.dll	145	FALSE	EE310	0	0	EF096	A9078
EE1CC	USER32.dll	2	FALSE	EE858	0	0	EF0BE	A95C0
EE1E0	ADVAPI32.dll	5	FALSE	EE298	0	0	EF124	A9000
EE1F4	SHELL32.dll	2	FALSE	EE800	0	0	EF166	A9568
EE208	ole32.dll	5	FALSE	EE8F0	0	0	EF1D6	A9658
EE21C	OLEAUT32.dll	9	FALSE	EE7B0	0	0	EF1E0	A9518

CRYPT32.dll [6 entries]

Call via	Name	Ordinal	Original Thunk	Thunk	Forwarder	Hint
A9030	CertVerifyCertifi...	-	EE996	7FFBB66A3ED0	-	75
A9038	CertFreeCertific...	-	EE97A	7FFBB66914C0	-	3D
A9040	CertFreeCertific...	-	EE958	7FFBB6687870	-	3E
A9048	CertFreeCertific...	-	EE93A	7FFBB66917B0	-	40
A9050	CertGetCertifica...	-	EE920	7FFBB6690500	-	45

[Figure 16] PE Bear: fixed Imports

Save the fixed file (*right-click "mas_5_unpacked" and pick-up "Save the executable as" option*) and keep it together with original malware sample and extracted one (before fixing). It's always recommended to save all artifacts within an only folder.

There're good indicators this binary is the final unpacked version because there're many DLLs (not only one), a suggestion of networking communication ***(WS2_32.dll – WinSock2)*** **,** strings related to virtual machine detection (we'll see them while analyzing the sample using IDA Pro later) and other clues.

Before proceeding to the analysis section, I'd like to quickly review some concepts about Assembly x64 because this sample is our first binary written for 64-bit systems.

7. Reviewing x64 assembly concepts

Likely readers already know about calling conventions and I wouldn't touch on this topic, but eventually I could prevent problems to new reverse engineers, so I'd like to leave some few words here.

While most malware samples continue being released for x86 architecture (32-bit), we have seen an increasing number of threats written for x64 architecture like the Bumblebee sample being analyzed in this article and clearly all attackers will be migrating to 64-bit in next years.

In x86 (32-bit) architecture, which **32-bit values are returned into EAX register**, **64-bit values are returned** through **EDX:EAX** and registers such **EBP, ESI, EDI** and **EBX** are **restored at the end of function**, we have:

- **__stdcall**
  - The callee is responsible for cleaning the stack.
  - All arguments are passed onto the stack by value.
  - All Win32 APIs use this standard.
  - Functions using this calling convention requires a function prototype.
  - The _stdcall is used by Microsoft compilers.

- **__cdecl**
  - The caller is responsible for cleaning the stack.
  - At same way of __stdcall, arguments are passed onto the stack.
  - This calling convention is the default one for C and C++ programs.
  - The __cdecl is used in Microsoft compilers.
  - Variadic functions use this convention because the callee don't know how many arguments were passed.

- **__fastcall**
  - This convention is only applied to x86 architecture.
  - In Microsoft compilers, first two arguments are passed by register (ECX and EDX, respectively). All remaining arguments are passed via stack. Other compilers use different scheme.
  - The callee is responsible for cleaning the stack.

- **__thiscall**
  - It's the convention used for C++ class member functions on x86 architecture.
  - The callee is responsible for cleaning the stack.
  - The "this" pointer is passed through ECX register.
  - "this" pointers are available for non-static C++ member functions.

- **__clrcall**
  - This determines that a function can only be called from a managed code.
  - It must be used for virtual functions called from managed mode.
  - This convention can't be used for functions being called from native code.

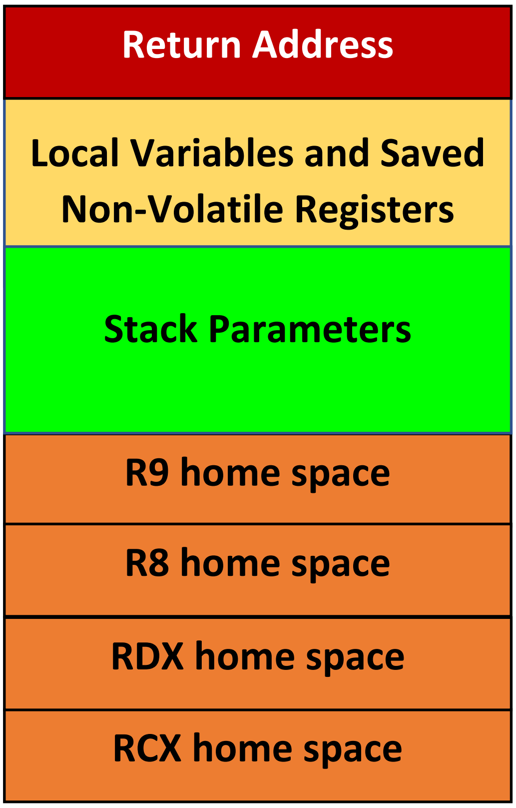
- **__vectorcall**
  - Arguments are passed through registers (when possible).
  - This convention used more registers than __fastcall .
  - Only supported in x86/x64 native processors with SSE2 support.
  - Three types of arguments can be passed by register in vectorcall: integer, vector, and homogenous vector aggregate types.
  - The _vectorcall is used in Microsoft compilers.

In x64-bit architecture, things a bit different because there's only one calling convention (**x64 __fastcall**) and many other details:

- All parameters and values are **64-bit (QWORD)**.
- The **returned value** goes to **RAX**.
- The concept of **shadow home** (as well known as **home space**) comes up. In few words, the **shadow home (allocated by the caller) aims to reserve a space (0x20 bytes) for callees to save the first 4 parameters** that are being passed **even that there isn't any parameter being passed!**
- If the callee has any local variable, so it'll allocate an additional space for them in addition to the 0x20 bytes (32 bytes).

- If the shadow home is not used for storing function's parameters (because there isn't any), so the **compiler uses this space to save non-volatile registers.**
- The **first four parameters** are passed to **RCX, RDX, R8 and R9 registers**, respectively. All remaining parameters are passed on the stack.
- The **caller (non-leaf function) usually saves volatile registers** such as **RAX, RCX, R8, R9, R10 and R11** because they can be changed by callee. Other register like **XMM4 and XMM5** are also saved.
- The **callee saves R12, R13, R14, R15, RSI, RDI, RSP, RBX (non-volatile)** and restores them later.
- The **caller (non-leaf function) is responsible for allocating space for parameters being passed to callee**, which makes the usage of variadic functions easier.
- The **stack must be aligned in 16 bytes**. Usually, it's not an issue, but functions such as **malloc( )** and **alloca( )** might broken this alignment.

A possible picture to illustrate a **non-leaf function being called** follows (we aren't considering **alloca()** function neither a frame pointer being used):



[Figure 17] Possible x64 stack organization

There're many other quite relevant details about x64 assembly:

- x64 executable follows the **PE32+ format**.
- **RxD (R8D, R9D,...): 32-bit**
- **RxW (R8W, R9W,...): 16-bit**
- **RxL (R8L, R9L,..): 8-bit**

- The **x64 code supports instruction pointer-relative data addressing (RIP relative addressing).** In other words, it's possible to access data at location that's away by an offset from the current pointer.
- In binaries containing **PIC** (**Position Independent Code**), the address is not stored "in the instruction", but only **an offset from the current instruction pointer (RIP).**
- **Functions can't allocate space at the middle of the code like may occur in x86, but only at their beginning.** Therefore, **there aren't pop and push instructions at the middle of the function.** Because of this rule, the **stack size keeps constant over the function's life**.
- Most of the time, **RSP register acts as stack pointer and frame points**, but there's an exception: **alloca( ).**
- PE32+ binaries have an additional section named **.pdata (as known as Exception Directory)**, which holds information used for handling exceptions.
- The **GS register is used to access the TEB (Thread Environment Block) from the user mode**, but it is also used to access the **KPCR (Kernel Processor Control Region) when the code is executing from kernel mode**.
- If readers don't know about **KPCR**, there's one for each logical processor and it represents a structure that contains general information about the processor and its status. Use the **WinDbg** (or, in some case, SysInternals' **livekd**) and try: a. **dt nt!_KPCR** ; b**. dt!_KPRCB**; c. **!pcr** commands.

If readers have spare time to examine the Windows system, there're many examples about details mentioned above. For example, **in the function below, non-volatile registers are being saved before function continuing**:

```
3: kd> u nt!KiExecuteAllDpcs
nt!KiExecuteAllDpcs:
fffff807`662a6f80 4053            push    rbx
fffff807`662a6f82 56              push    rsi
fffff807`662a6f83 57              push    rdi
fffff807`662a6f84 4154            push    r12
fffff807`662a6f86 4155            push    r13
fffff807`662a6f88 4156            push    r14
fffff807`662a6f8a 4157            push    r15
fffff807`662a6f8c 4881ecc0010000  sub     rsp,1C0h
```

[Figure 18] Disassembling a kernel function: saving non-volatile registers

A **DPC (Deferred Procedure Call)** is queued from **ISR (Interrupt Service Routine)**, which must be executed very quickly, to accomplish the **"most part of the task" not done by the ISR.**

In other words, **DPCs (through DpcForIsr or CustomDpc routines) are responsible for finishing tasks initiated by ISR (for example, an I/O operation)** and are called from an **arbitrary DPC context** at **DISPATCH_LEVEL (IRQL).**

Returning to our theme, readers can clearly notice that **KiExecuteAllDpc( )** (figure above) is a **callee saving all non-volatile registers before continuing for restoring them later**.

Of course, over this long function, there're many pieces of code that offer other demonstrations related to we're talking about. For example, the next piece of code shows a function being called **(EtwTraceLongDpcMitigationEvent( )**) and six arguments being passed to it: first four arguments through registers (**RCX, RDX, R8D, R9D**) and other two through the stack (**[RSP+20h] and [RSP+28]**). Pay attention to a detail: all arguments are passed using **mov instruction** (and variants), and not **push instructions**:

```
nt!KiExecuteAllDpcs+0x1e5c46:
fffff807`6648cbc6 44886c2428      mov     byte ptr [rsp+28h],r13b
fffff807`6648cbcb 44885c2420      mov     byte ptr [rsp+20h],r11b
fffff807`6648cbd0 440fb64c2432    movzx   r9d,byte ptr [rsp+32h]
fffff807`6648cbd6 450fb687c3000000 movzx   r8d,byte ptr [r15+0C3h]
fffff807`6648cbde 488bd3          mov     rdx,rbx
fffff807`6648cbe1 498bcf          mov     rcx,r15
fffff807`6648cbe4 e817d21900      call    nt!EtwTraceLongDpcMitigationEvent (fffff807`66629e00)
fffff807`6648cbe9 90              nop
fffff807`6648cbea e947a6e1ff      jmp     nt!KiExecuteAllDpcs+0x2b6 (fffff807`662a7236)  Branch
```

[Figure 19] Disassembling a kernel function: passing arguments through register and stack

Finally, that's a good example of **non-volatile parameters being restored** in a late point:

```
nt!KiExecuteAllDpcs+0x81e:
fffff807`662a779e 488b8c24b0010000 mov     rcx,qword ptr [rsp+1B0h]
fffff807`662a77a6 4833cc          xor     rcx,rsp
fffff807`662a77a9 e8a22c1300      call    nt!_security_check_cookie (fffff807`663da450)
fffff807`662a77ae 4881c4c0010000  add     rsp,1C0h
fffff807`662a77b5 415f            pop     r15
fffff807`662a77b7 415e            pop     r14
fffff807`662a77b9 415d            pop     r13
fffff807`662a77bb 415c            pop     r12
fffff807`662a77bd 5f              pop     rdi
fffff807`662a77be 5e              pop     rsi
fffff807`662a77bf 5b              pop     rbx
fffff807`662a77c0 c3              ret
```

[Figure 20] Disassembling a kernel function: restoring non-volatile registers

Of course, readers don't need to use this specific function and, eventually, could try any function called by **nt!KiExecuteAllDpc( )** whether don't want to "change" to another scope. For example, to list functions being called by **nt!KiExecuteAllDpc( )**, execute:

```
3: kd> uf /c nt!KiExecuteAllDpcs
nt!KiExecuteAllDpcs (fffff807`202a7010)
  nt!KiExecuteAllDpcs+0x48c (fffff807`202a749c):
    call to nt!guard_dispatch_icall (fffff807`20421220)
  nt!KiExecuteAllDpcs+0x726 (fffff807`202a7736):
    call to nt!KiSetVpThreadSpinLockCount (fffff807`2027e780)
  nt!KiExecuteAllDpcs+0x7f4 (fffff807`202a7804):
    call to nt!KiSelectReadyThread (fffff807`20282210)
  nt!KiExecuteAllDpcs+0x816 (fffff807`202a7826):
    call to nt!KiSetVpThreadSpinLockCount (fffff807`2027e780)
  nt!KiExecuteAllDpcs+0x829 (fffff807`202a7839):
    call to nt!_security_check_cookie (fffff807`203da470)
  nt!KiExecuteAllDpcs+0x9d7 (fffff807`202a79e7):
    call to nt!EtwpLogKernelEvent (fffff807`20279780)
  nt!KiExecuteAllDpcs+0xacb (fffff807`202a7adb):
    call to nt!KiSetVpThreadSpinLockCount (fffff807`2027e780)
  nt!KiExecuteAllDpcs+0xb4f (fffff807`202a7b5f):
    call to nt!KiSetVpThreadSpinLockCount (fffff807`2027e780)
  nt!KiExecuteAllDpcs+0xc80 (fffff807`202a7c90):
    call to nt!KiEnterDeferredReadyState (fffff807`20294540)
  nt!KiExecuteAllDpcs+0xcdb (fffff807`202a7ceb):
    call to nt!KiDeferredReadySingleThread (fffff807`202a9450)
  nt!KiExecuteAllDpcs+0xd1f (fffff807`202a7d2f):
    call to nt!KiFlushSoftwareInterruptBatch (fffff807`202ac510)
  nt!KiExecuteAllDpcs+0xe03 (fffff807`202a7e13):
    call to nt!KiGetThreadEffectiveRankNonZero (fffff807`202a8e90)
```

[Figure 21] Listing functions called by a given function

Click on any called function and try to perform a similar and very quick analysis. Additionally, if reader want to examine and learn a bit more about structures related to the function table entry for the given function and even exceptions, it's recommended to execute:

```
3: kd> .fnent nt!KiExecuteAllDpcs
Debugger function entry 0000018d`507e59d0 for:
(fffff807`202a7010)   nt!KiExecuteAllDpcs   |  (fffff807`202a80e0)   nt!KiSelectActiveTimerTable
Exact matches:
    nt!KiExecuteAllDpcs (void)

BeginAddress      = 00000000`002a7010
EndAddress        = 00000000`002a80cb
UnwindInfoAddress = 00000000`000680b4

Unwind info at fffff807`200680b4, 24 bytes
  version 2, flags 3, prolog 25, codes b
  handler routine: nt!_GSHandlerCheck_SEH (fffff807`20413820), data 1
  00: offs c, unwind op 6, op info 0  UWOP_EPILOG Length: c. Flags: 0
  01: offs 86, unwind op 6, op info 8 UWOP_EPILOG Offset from end: 886 (FFFFF807202A7845)
  02: offs 13, unwind op 1, op info 0 UWOP_ALLOC_LARGE FrameOffset: 1c0.
  04: offs c, unwind op 0, op info f  UWOP_PUSH_NONVOL reg: r15.
  05: offs a, unwind op 0, op info e  UWOP_PUSH_NONVOL reg: r14.
  06: offs 8, unwind op 0, op info d  UWOP_PUSH_NONVOL reg: r13.
  07: offs 6, unwind op 0, op info c  UWOP_PUSH_NONVOL reg: r12.
  08: offs 4, unwind op 0, op info 7  UWOP_PUSH_NONVOL reg: rdi.
  09: offs 3, unwind op 0, op info 6  UWOP_PUSH_NONVOL reg: rsi.
  0a: offs 2, unwind op 0, op info 3  UWOP_PUSH_NONVOL reg: rbx.
```

[Figure 22] Examining the function table structures and fields related to exception

I'm not sure whether readers already touched this stuff previous, but some few notes could be useful:

- **BeginAddress**: offset to the start point of the function. By adding this offset to the base of the module, we get the address of the function.

- **EndAddress:** offset to the end point of the function.

- **UnwindInfoAddress:** it is a pointer to the unrolling information structure, which describes the correct way that stack should be unrolled whether an exception occurred.

- All three fields (**BeginAddress, EndAddress** and **UnwindInfoAddress**) make part of the **_RUNTIME_FUNCTION structure**, which is located inside **.pdata section**.

- When an exception occurs, the first step is to use the RIP to search for an entry from this structure that describes the current function.

- We should note that several registers are listed as non-volatile, so they must be saved before the function overwriting them and, later, they will be recovered when it's appropriated. Furthermore, the remaining flags tell use a bit more about the context:

    - **UWOP_ALLOC_LARGE:** Allocates a large area on the stack. If the **operation info** is equal to 1, so part of the allocation (512K up to 4GB – 8 bytes) is recorded in the next two slots. If the **operation info** is zero, then the size of the allocation (136 to 512K – 8 bytes) would be recorded in the next slot.

- **UWOP_ALLOC_SMALL** (not shown): it represents the size of the allocation (allocations between 8 and 128 bytes).
- **UWOP_PUSH_NONVOL**: this unwind operation code means a **push** of a non-volatile, which decrements RSP by 8.
- **UWOP_PUSH_NONVOL:** this unwind operation code indicates that function saves a nonvolatile integer register on the stack **using a MOV instead of a PUSH**.

Picking up a function from stack we have:

```
3: kd> knf
 #   Memory  Child-SP          RetAddr               Call Site
00           ffffb981`c7d729b8 fffff807`204b2e7c     nt!DbgBreakPointWithStatus
01         8 ffffb981`c7d729c0 fffff807`202b7168     nt!KdCheckForDebugBreak+0x1986d4
02        30 ffffb981`c7d729f0 fffff807`202b6e01     nt!KeAccumulateTicks+0x188
03        70 ffffb981`c7d72a60 fffff807`202b3e36     nt!KiUpdateRunTime+0x61
04        60 ffffb981`c7d72ac0 fffff807`202b5142     nt!KiUpdateTime+0x686
05       3f0 ffffb981`c7d72eb0 fffff807`202b2a72     nt!KeClockInterruptNotify+0x272
06        90 ffffb981`c7d72f40 fffff807`202913d0     nt!HalpTimerClockInterrupt+0xe2
07        30 ffffb981`c7d72f70 fffff807`2041994a     nt!KiCallInterruptServiceRoutine+0xa0
08        40 ffffb981`c7d72fb0 fffff807`20419f17     nt!KiInterruptSubDispatchNoLockNoEtw+0xfa
09           ffffa301`3de45ab0 fffff807`2041bc7a     nt!KiInterruptDispatchNoLockNoEtw+0x37
0a       190 ffffa301`3de45c40 00000000`00000000     nt!KiIdleLoop+0x5a
3: kd> .fnent nt!KiUpdateTime
Debugger function entry 0000018d`507e59d0 for:
(fffff807`202b37b0)   nt!KiUpdateTime    |  (fffff807`202b4170)   nt!PpmCheckSnapAllDeliveredPerformance
Exact matches:
    nt!KiUpdateTime (void)

BeginAddress      = 00000000`002b37b0
EndAddress        = 00000000`002b4166
UnwindInfoAddress = 00000000`0006a444

Unwind info at fffff807`2006a444, 30 bytes
  version 2, flags 3, prolog 37, codes 11
  handler routine: nt!_GSHandlerCheck (fffff807`203dfee4), data 3b0
  00: offs a, unwind op 6, op info 0   UWOP_EPILOG Length: a. Flags: 0
  01: offs 5, unwind op 6, op info 3   UWOP_EPILOG Offset from end: 305 (FFFFF807202B3E61)
  02: offs 25, unwind op 4, op info 7  UWOP_SAVE_NONVOL FrameOffset: 408 reg: rdi.
  04: offs 25, unwind op 4, op info 6  UWOP_SAVE_NONVOL FrameOffset: 400 reg: rsi.
  06: offs 25, unwind op 4, op info 5  UWOP_SAVE_NONVOL FrameOffset: 3f8 reg: rbp.
  08: offs 25, unwind op 4, op info 3  UWOP_SAVE_NONVOL FrameOffset: 3f0 reg: rbx.
  0a: offs 25, unwind op 1, op info 0  UWOP_ALLOC_LARGE FrameOffset: 3c0
  0c: offs 1e, unwind op 0, op info f  UWOP_PUSH_NONVOL reg: r15.
  0d: offs 1c, unwind op 0, op info e  UWOP_PUSH_NONVOL reg: r14.
  0e: offs 1a, unwind op 0, op info d  UWOP_PUSH_NONVOL reg: r13.
  0f: offs 18, unwind op 0, op info c  UWOP_PUSH_NONVOL reg: r12.
  10: offs 16, unwind op 2, op info 0  UWOP_ALLOC_SMALL.

3: kd> u nt!KiUpdateTime
nt!KiUpdateTime:
fffff807`202b37b0 48895c2408        mov     qword ptr [rsp+8],rbx
fffff807`202b37b5 48896c2410        mov     qword ptr [rsp+10h],rbp
fffff807`202b37ba 4889742418        mov     qword ptr [rsp+18h],rsi
fffff807`202b37bf 48897c2420        mov     qword ptr [rsp+20h],rdi
fffff807`202b37c4 489c              pushfq
fffff807`202b37c6 4154              push    r12
fffff807`202b37c8 4155              push    r13
fffff807`202b37ca 4156              push    r14
3: kd> u
nt!KiUpdateTime+0x1c:
fffff807`202b37cc 4157              push    r15
fffff807`202b37ce 4881ecc0030000    sub     rsp,3C0h
fffff807`202b37d5 488b05a4919500    mov     rax,qword ptr [nt!_security_cookie (fffff807`20c0c980)]
fffff807`202b37dc 4833c4            xor     rax,rsp
fffff807`202b37df 48898424b0030000 mov      qword ptr [rsp+3B0h],rax
fffff807`202b37e7 88542430          mov     byte ptr [rsp+30h],dl
fffff807`202b37eb 41b808010000      mov     r8d,108h
fffff807`202b37f1 884c2431          mov     byte ptr [rsp+31h],cl
```

3f0 = 3c0 + (4 * 8) + 8 + 8 , where:

- **3c0: ALLOC_LARGE**
- **(4 * 8): five registers pushed.**
- **8: return address's size**
- **8: UWOP_ALLOC_SMALL**

UWOP_ALLOC_SMALL = (op info) * 8 + 8 = 0 * 8 + 8 = 8

[Figure 23] Correlating details

There're other quite relevant details that, sometimes, are very useful for the daily job in reverse engineering and programming. For example, functions can be declared with **naked attribute** and, as a side effect, **these functions don't have prolog and neither epilog**. We can use **naked function** in a **trampoline function, which is responsible for restoring the overwritten instructions, while writing a hooking program**.

As probably readers already know, **hooking** is a legal mechanism used for **monitoring, instrumentation, and extension of a target function**. In the malware world, it's also a technique used by the malicious binary to **modify the system aiming to hide multiple artifacts and activities**.

Hooking can be performed inline or at the IAT (Import Address Table), for example. At the end, **the general idea is having the following execution flow**:

- Original function is called.
- The **inline hooking** at its beginning takes effect and redirects the execution to the malicious function.
- At end of the malicious function, the execution flow is redirected to the **trampoline function**.
- The **trampoline function** restores the overwritten instructions.
- The **trampoline function** jumps to the original (hooked) function to executing the remaining instructions.

Anyway, I don't have intention to enter in detail about hooking in this article and, probably, I will return to this topic in future texts.

8. Reversing: first part

Let's proceed to the reversing phase. Of course, I don't have any plan or intention to be very detailed in this analysis because it isn't necessary and because we'll have several opportunities to do it over this long series of articles. Furthermore, based on my experience, the best approach to teach something for professionals is by exposing topics gradually without a big changing of the difficult level to allow readers to get used to the terms and techniques and, intuitively, learn all necessary concepts.

I'll be using **IDA Pro 8.x** and, mainly, the **Hex-Rays Decompiler** to get further understand of each explained details and then from this point onward is done assuming readers are using the same tools. Just in case you aren't, so try to adapt explanations to your scenario and context.

I'll be analyzing the **mas_5_unpacked.bin** file that's resultant from our unpacking procedure. As usual, it's recommended to accomplish few steps before proceeding the analysis:

- Decompile the entire binary: **File | Produce File | Create C File**. Save the C file in the same directory of the unpacked binary.

- Load important libraries (*remember: the sample is 64-bit*) for analysis: **View | Open Subviews | Type Libraries (SHIFT+F11 hotkey)** and insert libraries (**INS hotkey**) such as:

  - **ntapi64_win7**
  - **ntddk64_win7 (it's usually necessary while analyzing kernel drivers)**

- **mssdk64_win7** (usually inserted automatically).

It's also advisable to add some signatures (*once again, remember: the sample is 64-bit*), which will help us in most of reversing cases. Thus, go to **View | Open Subview | Signatures (SHIFT+F5 hotkey)** and insert (**INS key**) the following library modules:

- **vc64rtf**
- **vc64ucrt**
- **vc64seh**

Once reader already have the decompiled binary and loaded the main libraries and signatures, so open a **Pseudo Code window** and configure it side-by-side with the **Assembly View** window and synchronize it with the **IDA View** (**right click → Synchronize with**). Additionally, it's interesting to check strings (**SHIFT+F12**) to help us as an extra guidance for our reversing work.

Readers will see several functions involving C++ over the code, so it's also recommended to demangle C++ names by going to **Options | Demangled Names** and mark **Names**.

There're so many interesting aspects over the code that's hard to choose a point to starting the analysis. For example, by visualizing strings (**SHIFT+F12 hotkey**), we find several clues about what could being done by the code. Thus, part of these strings follows below:

- **powershell**
- **VirtualBox**
- **LogonUserA**
- **LogonUserW**
- **cmd.exe /c**
- **User name:**
- **In-Reply-To**
- **FindWindowW**
- **QueueUserAPC**
- **tasks**
- **BOCHS**
- **Domain**
- **WinExec**
- **beast.http**
- **LdrUnloadDll**
- **VcFFI2Rj6t15**
- **VMWare**
- **MapViewOfFile**
- **FindNextFileA**
- **Injection-Date**
- **IsWow64Process**
- **GetProductInfo**
- **CryptImportKey**
- **FindFirstFileA**
- **FindFirstFileW**
- **CreateProcessA**
- **CreateProcessW**
- **qemu-ga**
- **CreateMailslotA**
- **CreateSemaphoreW**
- **OpenProcessToken**
- **SeDebugPrivilege**
- **NtQueueApcThread**
- **CreateNamedPipeA**
- **NtCreateThreadEx**
- **NtQueueApcThread**
- **idaq.exe**
- **VBoxWddm**
- **FileName**
- **DeviceId**
- **VEN_VBOX**
- **NtCreateProcessEx**
- **IsDebuggerPresent**
- **WriteProcessMemory**
- **LdrHotPatchRoutine**
- **vboxvideo**
- **ZwReadVirtualMemory**
- **GetNativeSystemInfo**
- **NtCreateDebugObject**
- **RtlDecompressBuffer**
- **ZwWriteVirtualMemory**
- **EnumProcessModulesEx**
- **CreateRemoteThreadEx**
- **\\Windows Mail\\wab.exe**
- **ROOT\\CIMV2**
- **ShowWindow**
- **regmon.exe**
- **idaq64.exe**
- **LordPE.exe**
- **windbg.exe**
- **x32dbg.exe**
- **x64dbg.exe**
- **Identifier**
- **VIRTUALBOX**
- **MACAddress**
- **VirtualBox**
- **prl_cc.exe**
- **VMSrvc.exe**
- **mscoree.dll**
- **wscript.exe**
- **ollydbg.exe**
- **tcpview.exe**
- **procmon.exe**
- **filemon.exe**
- **procexp.exe**
- **dumpcap.exe**
- **PETools.exe**
- **Fiddler.exe**
- **VBoxVideoW8**
- **vdagent.exe**
- **NtAdjustPrivilegesToken**
- **NtAllocateVirtualMemory**
- **wscript.exe /E:vbscript**
- **NtQueryInformationThread**
- **CreateToolhelp32Snapshot**
- **ZwQueryInformationProcess**
- **autoruns.exe**
- **vboxtray.exe**
- **PostQueuedCompletionStatus**
- **Win32_Process**
- **autorunsc.exe**
- **Wireshark.exe**
- **ImportREC.exe**
- **\\\\.\\VBoxGuest**
- **ACPIBus_BUS_0**
- **vdservice.exe**
- **SOFTWARE\\Wine**
- **procexp64.exe**

[Figure 24] Main strings found in the unpacked sample

- **prl_tools.exe**
- **HookExplorer.exe**
- **SysInspector.exe**
- **joeboxserver.exe**
- **httpdebugger.exe**
- **VideoBiosVersion**
- **SELECT * FROM Win32_ComputerSystem**
- **Win32_ProcessStartup**
- **SELECT * FROM Win32_ComputerSystemProduct**
- **ImmunityDebugger.exe**
- **Checking reg key %s**
- **System32\\vboxogl.dll**
- **\\\\.\\pipe\\VBoxTrayIPC**
- **VBoxTrayToolWndClass**
- **System32\\vboxdisp.dll**
- **System32\\vboxhook.dll**
- **System32\\vboxtray.exe**
- **System32\\vboxmrxnp.dll**
- **SELECT * FROM Win32_Bus**
- **Z:\\hooker2\\Common\\md5.cpp**
- **ProcessStartupInformation**
- **HARDWARE\\ACPI\\DSDT\\VBOX__**
- **HARDWARE\\ACPI\\RSDT\\VBOX__**
- **HARDWARE\\ACPI\\FADT\\VBOX__**
- **VirtualBox Shared Folders**
- **System32\\vboxoglcrutil.dll**
- **System32\\drivers\\viofs.sys**
- **D:\\Sources\\boost_1_78_0\\boost/beast/http/impl/verb.ipp**
- **D:\\Sources\\boost_1_78_0\\boost/beast/http/impl/read.hpp**
- **D:\\Sources\\boost_1_78_0\\boost/beast/http/impl/write.hpp**
- **System32\\drivers\\VBoxSF.sys**
- **System32\\vboxoglpackspu.dll**
- **System32\\drivers\\balloon.sys**
- **System32\\drivers\\pvpanic.sys**
- **System32\\drivers\\vioscsi.sys**
- **System32\\drivers\\viostor.sys**
- **%WINDIR%\\System32\\wscript.exe**
- **SELECT * FROM Win32_PnPEntity**
- **SELECT * FROM Win32_BaseBoard**
- **SELECT * FROM Win32_PnPDevice**
- **System32\\drivers\\VBoxMouse.sys**
- **System32\\drivers\\VBoxGuest.sys**
- **System32\\drivers\\VBoxVideo.sys**
- **SELECT * FROM Win32_NTEventlogFiles**
- **SYSTEM\\ControlSet001\\Services\\VBoxSF**
- **SYSTEM\\ControlSet001\\Services\\netkvm**
- **Copyright (c) by P.J. Plauger, licensed by Dinkumware, Ltd. ALL RIGHTS RESERVED.**
- **SELECT * FROM Win32_NetworkAdapterConfiguration**
- **void __cdecl boost::beast::http::message<1,struct boost::beast::http::basic_string_body<char,struct std::char_traits<char>,class std::allocator<char> >,class boost::beast::http::basic_fields<class std::allocator<char> > >::prepare_payload(struct std::integral_constant<bool,1>)**

[Figure 25] Main strings found in the unpacked sample (second part)

Based on the list from **Figure 24 and 25**, few considerations follow below:

- Clearly the malware threat detects a series of debuggers and tools like **Ollydbg, IDA Pro (idaq.exe), Immunity Debugger, WinDbg, x64/x32dbg, Process Hacker, Process Monitor, Bochs** and so on. The array containing all tool's names was renamed to **searched_tools[ ].**

- The **sub_18004D60 subroutine (renamed to ab_DetectRunningTools())**, which is used to detect these tools above, is called three times (check cross references through **X hotkey**). In addition, it's a called within a loop (with a **Sleep( )** call) and if a process named with any one of these strings is found, so the malware execution is terminated (**TerminateProcess( )**)

- The **sub_18004D60( )** itself calls **sub_180050380( )**, which contains a typical sequence of calls such as **CreateToolhelp32Snapshot( ) + Process32FirstW( ) + Process32NextW( )**. In few words, these last two calls are used to parse the snapshot result.

- A relevant point as this routine **sub_180050380( )** (we renamed it to **ab_SearchProcesses**) that searches for processes is well-used by the malware and is called **six times** from other parts of the code, so maybe is a good point to do follow-up.

```
__int64 sub_18004D060()
{
  __int64 v0; // rbx
  __int64 result; // rax
  PCWSTR psz2[32]; // [rsp+20h] [rbp-E0h]

  v0 = 0i64;
  psz2[0] = L"ollydbg.exe";
  psz2[1] = L"ProcessHacker.exe";
  psz2[2] = L"tcpview.exe";
  psz2[3] = L"autoruns.exe";
  psz2[4] = L"autorunsc.exe";
  psz2[5] = L"filemon.exe";
  psz2[6] = L"procmon.exe";
  psz2[7] = L"regmon.exe";
  psz2[8] = L"procexp.exe";
  psz2[9] = L"idaq.exe";
  psz2[10] = L"idaq64.exe";
  psz2[11] = L"ImmunityDebugger.exe";
  psz2[12] = L"Wireshark.exe";
  psz2[13] = L"dumpcap.exe";
  psz2[14] = L"HookExplorer.exe";
  psz2[15] = L"ImportREC.exe";
  psz2[16] = L"PETools.exe";
  psz2[17] = L"LordPE.exe";
  psz2[18] = L"SysInspector.exe";
  psz2[19] = L"proc_analyzer.exe";
  psz2[20] = L"sysAnalyzer.exe";
  psz2[21] = L"sniff_hit.exe";
  psz2[22] = L"windbg.exe";
  psz2[23] = L"joeboxcontrol.exe";
  psz2[24] = L"joeboxserver.exe";
  psz2[25] = L"joeboxserver.exe";
  psz2[26] = L"ResourceHacker.exe";
  psz2[27] = L"x32dbg.exe";
  psz2[28] = L"x64dbg.exe";
  psz2[29] = L"Fiddler.exe";
  psz2[30] = L"httpdebugger.exe";
```

[Figure 26] Debuggers and Tools verified by malware

```
__int64 __fastcall sub_180050380(PCWSTR psz2)
{
  HANDLE Toolhelp32Snapshot; // rax
  void *v3; // rdi
  int v4; // eax
  void *v5; // rcx
  int v7; // eax
  PROCESSENTRY32W pe; // [rsp+20h] [rbp-258h] BYREF

  memset(&pe, 0, sizeof(pe));
  Toolhelp32Snapshot = CreateToolhelp32Snapshot(2u, 0)
  v3 = Toolhelp32Snapshot;
  if ( Toolhelp32Snapshot == (HANDLE)-1i64 )
    return 0i64;
  pe.dwSize = 568;
  if ( Process32FirstW(Toolhelp32Snapshot, &pe) )
  {
    v4 = StrCmpIW(pe.szExeFile, psz2);
    v5 = v3;
    if ( !v4 )
    {
LABEL_4:
      CloseHandle(v5);
      return pe.th32ProcessID;
    }
    while ( Process32NextW(v3, &pe) )
    {
      v7 = StrCmpIW(pe.szExeFile, psz2);
      v5 = v3;
      if ( !v7 )
        goto LABEL_4;
    }
  }
  CloseHandle(v3);
  return 0i64;
}
```

[Figure 27] Common subroutine searching for specific running processes

- The malware threat also detects through **sub_18004FAB0** (renamed to **ab_DetectVirtualMachines**) whether the malware is running on virtual machines like **VMWare, VirtualBox** or **Xen**. Additionally, it also uses **VMI queries (WQL)** to detect the virtual environment.

- Indeed, WMI will be used several times to help on detecting virtual environments artifacts and, as readers are going to notice, **COM (Component Object Model)** will be involved in this context. In few pages we'll return to this subject.

- Another anti-analysis approach used by the sample is detecting sandboxes and testing virtual machines artifacts and, in special, common usernames used by test/sandboxes environments. The subroutine **sub_18004F860** (renamed to **ab_CheckUserNames**) is responsible for it and readers can see that there's a list of them shown in **Figure 28** (next page).

```
__int64 sub_18004F860()
{
  WCHAR *v0; // rax
  WCHAR *v1; // rsi
  unsigned int v3; // ebp
  __int64 v4; // rbx
  const wchar_t *v5; // rdi
  __int64 pcbBuffer; // [rsp+20h] [rbp-2B8h] BYREF
  wchar_t *String1[18]; // [rsp+30h] [rbp-2A8h]
  char Buffer[512]; // [rsp+C0h] [rbp-218h] BYREF

  LODWORD(pcbBuffer) = 257;
  String1[0] = L"CurrentUser";
  String1[1] = L"Sandbox";
  String1[2] = L"Emily";
  String1[3] = L"HAPUBWS";
  String1[4] = L"Hong Lee";
  String1[5] = L"IT-ADMIN";
  String1[6] = L"Johnson";
  String1[7] = L"Miller";
  String1[8] = L"milozs";
  String1[9] = L"Peter Wilson";
  String1[10] = L"timmy";
  String1[11] = L"sand box";
  String1[12] = L"malware";
  String1[13] = L"maltest";
  String1[14] = L"test user";
  String1[15] = L"virus";
  String1[16] = L"John Doe";
  v0 = (WCHAR *)j__malloc_base(0x202ui64);
  v1 = v0;
  if ( !v0 )
    return 1i64;
  if ( !GetUserNameW(v0, (LPDWORD)&pcbBuffer) )
  {
    j__free_base(v1);
    return 1i64;
  }
```

[Figure 28] Usernames verified by the malware threat

```
_BOOL8 ab_CheckMemory()
{
  struct _MEMORYSTATUSEX v1; // [rsp+20h] [rbp-58h] BYRE

  memset(&v1.dwMemoryLoad, 0, 60);
  v1.dwLength = 64;
  GlobalMemoryStatusEx(&v1);
  return v1.ullTotalPhys < 0x100000000i64;
}
```

[Figure 29] Checking if the system's memory has less than 4GB.

- The malware, through **sub_18004FA40( )** (renamed to **ab_CheckMemory**), uses a well-known trick for testing whether the system has 4GB at least because it's usual public sandboxes or even virtual environments using less than it. To get the task done, the **GlobalMemoryStatusEx( )** API is used, which returns both physical and virtual memory.

- An interesting subroutine is **sub_180050270** (renamed to **ab_checkMACAddress**) that is responsible for retrieving the MAC address from the network adapter using **GetAdaptersInfo( )**, which gets adapter information (only for IPv4) for the local system. In general, its behavior is like other cases on Windows system programming, where an API is called twice: the first one to get the correct size of the necessary structure and the second one to allocate and use it. Readers notice that the **sub_180050270** is parsing all retrieved MAC addresses and verifying whether any of them matches to the given argument. The function **GetAdaptersInfo( )** has the following signature:

  - **IPHLPAPI_DLL_LINKAGE ULONG GetAdaptersInfo(PIP_ADAPTER_INFO AdapterInfo, PULONG SizePointer)**

```
__int64 __fastcall ab_checkMACAddress(_BYTE *arg_1_byte)
{
  unsigned int var_return_value; // esi
  HANDLE ProcessHeap; // rax
  struct _IP_ADAPTER_INFO *var_struct_IP_ADAPTER_INFO; // rax
  struct _IP_ADAPTER_INFO *ref_var_struct_IP_ADAPTER_INFO; // rbx
  ULONG AdaptersInfo; // eax
  HANDLE ptr_Heap; // rax
  ULONG var_SizePointer; // ebx
  HANDLE ptr_Heap_1; // rax
  struct _IP_ADAPTER_INFO *struc_1_IP_ADAPTER_INFO; // rax
  char v12; // cl
  struct _IP_ADAPTER_INFO *struc_2_IP_ADAPTER_INFO; // rax
  HANDLE v14; // rax
  ULONG SizePointer; // [rsp+38h] [rbp+10h] BYREF
  __int16 v16; // [rsp+40h] [rbp+18h]

  SizePointer = 0x2C0;
  var_return_value = 0;
  ProcessHeap = GetProcessHeap();
  var_struct_IP_ADAPTER_INFO = (struct _IP_ADAPTER_INFO *)HeapAlloc(ProcessHeap, 0, 0x2C0ui64);
  ref_var_struct_IP_ADAPTER_INFO = var_struct_IP_ADAPTER_INFO;
  if ( !var_struct_IP_ADAPTER_INFO )
    return 0i64;
  AdaptersInfo = GetAdaptersInfo(var_struct_IP_ADAPTER_INFO, &SizePointer);
  if ( AdaptersInfo == ERROR_BUFFER_OVERFLOW )
  {
    ptr_Heap = GetProcessHeap();
    HeapFree(ptr_Heap, 0, ref_var_struct_IP_ADAPTER_INFO);
    var_SizePointer = SizePointer;
    ptr_Heap_1 = GetProcessHeap();
    struc_1_IP_ADAPTER_INFO = (struct _IP_ADAPTER_INFO *)HeapAlloc(ptr_Heap_1, 0, var_SizePointer);
    ref_var_struct_IP_ADAPTER_INFO = struc_1_IP_ADAPTER_INFO;
    if ( !struc_1_IP_ADAPTER_INFO )
      return 0i64;
    AdaptersInfo = GetAdaptersInfo(struc_1_IP_ADAPTER_INFO, &SizePointer);
    ref_var_struct_IP_ADAPTER_INFO = struc_1_IP_ADAPTER_INFO;
    if ( !struc_1_IP_ADAPTER_INFO )
      return 0i64;
    AdaptersInfo = GetAdaptersInfo(struc_1_IP_ADAPTER_INFO, &SizePointer);
  }
  if ( !AdaptersInfo )
  {
    v12 = arg_1_byte[4];
    LOBYTE(v16) = *arg_1_byte;
    HIBYTE(v16) = arg_1_byte[2];
    struc_2_IP_ADAPTER_INFO = ref_var_struct_IP_ADAPTER_INFO;
    while ( struc_2_IP_ADAPTER_INFO->AddressLength != 6
         || v16 != *(_WORD *)struc_2_IP_ADAPTER_INFO->Address
         || v12 != struc_2_IP_ADAPTER_INFO->Address[2] )
    {
      struc_2_IP_ADAPTER_INFO = struc_2_IP_ADAPTER_INFO->Next;
      if ( !struc_2_IP_ADAPTER_INFO )
        goto LABEL_14;
    }
    var_return_value = 1;
  }
LABEL_14:
  v14 = GetProcessHeap();
  HeapFree(v14, 0, ref_var_struct_IP_ADAPTER_INFO);
  return var_return_value;
}
```

[Figure 30] Checking MAC Addresses

```
typedef struct _IP_ADAPTER_INFO {
  struct _IP_ADAPTER_INFO *Next;
  DWORD                   ComboIndex;
  char                    AdapterName[MAX_ADAPTER_NAME_LENGTH + 4];
  char                    Description[MAX_ADAPTER_DESCRIPTION_LENGTH + 4];
  UINT                    AddressLength;
  BYTE                    Address[MAX_ADAPTER_ADDRESS_LENGTH];
  DWORD                   Index;
  UINT                    Type;
  UINT                    DhcpEnabled;
  PIP_ADDR_STRING         CurrentIpAddress;
  IP_ADDR_STRING          IpAddressList;
  IP_ADDR_STRING          GatewayList;
  IP_ADDR_STRING          DhcpServer;
  BOOL                    HaveWins;
  IP_ADDR_STRING          PrimaryWinsServer;
  IP_ADDR_STRING          SecondaryWinsServer;
  time_t                  LeaseObtained;
  time_t                  LeaseExpires;
} IP_ADAPTER_INFO, *PIP_ADAPTER_INFO;
```

[Figure 31] _IP_ADAPTER_INFO structure used by GetAdaptersInfo( )

- The subroutine **sub_18004F280( )** (renamed to **ab_checkWineRegistryEntry**) checks for Wine Registry's entry:

```
__int64 ab_checkWineRegistryEntry()
{
  HKEY hKey; // [rsp+30h] [rbp-228h] BYREF
  char Buffer[512]; // [rsp+40h] [rbp-218h] BYREF

  memset(Buffer, 0, sizeof(Buffer));
  sprintf_s_0(Buffer, 0x100ui64, L"Checking reg key %s ", L"SOFTWARE\\Wine");
  hKey = 0i64;
  if ( RegOpenKeyExW(HKEY_CURRENT_USER, L"SOFTWARE\\Wine", 0, 0x20019u, &hKey) )
    return 0i64;
  RegCloseKey(hKey);
  return 1i64;
}
```

[Figure 32] Checking for Wine within Registry

- The subroutine **sub_18004D280( )** (renamed to **ab_CheckVBoxHW**) searches for several different **VirtualBox hardware artifacts**. Please, check the whole subroutine on **Figure 33 (**next page).

- The string "***Checking reg key HARDWARE\\Description\\System - %s is set to %s***" refers directly to the code of a well-known project to detect virtual machines, which readers can easily search for on the Internet.

- There're other similar subroutines such as **sub_18004D3D0** (checks for VirtualBox Services), **sub_18004D520** (checks for VirtualBox processes), **sub_18004D520** (checks for VirtualBox executables, DLLs, and drivers), **sub_18004D790** (checks for VirtualBox Guest Additions), **sub_18004D8C0** (checks for VirtualBox devices and pipes used for IPC and debugging) and **sub_18004D9E0( )** that checks for VirtualBox shared folders.

- Additionally, there're two calls for **FindWindowW( )** to check for windows related to **VirtualBox**:

```
__int64 ab_CheckVBoxHW()
{
  __int64 v0; // rbx
  const WCHAR **v1; // r14
  const WCHAR *v2; // rdi
  __int64 v3; // rcx
  __int64 result; // rax
  const wchar_t *v5; // [rsp+30h] [rbp-278h]
  __int64 array_VBoxArtifacts[11]; // [rsp+38h] [rbp-270h] BYREF
  char Buffer[512]; // [rsp+90h] [rbp-218h] BYREF

  v0 = 0i64;
  array_VBoxArtifacts[2] = (__int64)L"HARDWARE\\Description\\System";
  v5 = L"HARDWARE\\DEVICEMAP\\Scsi\\Scsi Port 0\\Scsi Bus 0\\Target Id 0\\Logical Unit Id 0";
  array_VBoxArtifacts[1] = (__int64)L"VBOX";
  array_VBoxArtifacts[0] = (__int64)L"Identifier";
  v1 = (const WCHAR **)array_VBoxArtifacts;
  array_VBoxArtifacts[4] = (__int64)L"VBOX";
  array_VBoxArtifacts[3] = (__int64)L"SystemBiosVersion";
  array_VBoxArtifacts[6] = (__int64)L"VideoBiosVersion";
  array_VBoxArtifacts[7] = (__int64)L"VIRTUALBOX";
  array_VBoxArtifacts[9] = (__int64)L"SystemBiosDate";
  array_VBoxArtifacts[10] = (__int64)L"06/23/99";
  array_VBoxArtifacts[5] = (__int64)L"HARDWARE\\Description\\System";
  array_VBoxArtifacts[8] = (__int64)L"HARDWARE\\Description\\System";
  while ( 1 )
  {
    memset(Buffer, 0, sizeof(Buffer));
    v2 = v1[1];
    sprintf_s_0(Buffer, 0x100ui64, L"Checking reg key HARDWARE\\Description\\System - %s is set to %s", *v1, v2);
    result = w_RegQueryValueEx(v3, *(v1 - 1), *v1, v2);
    if ( (_DWORD)result )
      break;
    ++v0;
    v1 += 3;
    if ( v0 >= 4 )
      return result;
  }
  return 1i64;
}
```

[Figure 33] Subroutine used to check for the VirtualBox hardware artifacts.

- There're other many functions checking VirtualBox, VirtualPC, BOCHS, QEMU and other artifacts:

  - **sub_18004DA80( ):** checks for running VirtualBox processes.
  - **sub_18004DB30( ):** checks for VirtualBox MAC addresses through WMI and COM APIs.
  - **sub_18004DD10( ):** checks for VirtualBox Video Adapter using COM + WMI.
  - **sub_180004E3F0( ):** checks for VirtualBox Device.
  - **sub_18004EAB0( ):** checks for VirtualBox Device.
  - **sub_18004E820( ):** checks for VirtualBox manufacturer (Oracle).
  - **sub_18004E610( ):** checks for VirtualBox buses.
  - **sub_18004FF10( ):** checks for VirtualPC processes.
  - **sub_18004ED40( ):** checks for Qemu Registry's entries.
  - **sub_18004EE30( ):** checks for Qemu Processes.
  - **sub_18004EEF0( ):** checks for Qemu and SPICE Guest Tools.
  - **sub_18004F160( ):** checks for BOCHS.
  - **sub_18004F340( ):** checks for VirtIO services.

- **sub_18004F680( ):** checks for VirtIO directory.
- **sub_18004F460( ):** checks for VirtIO drivers.
- **sub_18004FE60( ):** checks for Parallels processes.

Finally, an overview of **sub_18004CD50** (renamed **ab_checkVirtualMachinesAndTools**), which invokes all **anti-analysis routines** mentioned above, follows:

```
bool ab_checkVirtualMachinesAndTools()
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  if ( (unsigned int)ab_checkMACAddress(&str_XenMAC) )
    return 1;
  v1 = 0;
  if ( (unsigned int)ab_SearchProcesses(L"procexp64.exe") )
    return 1;
  v2 = ab_DetectRunningTools() == 0;
  if ( !v2 )
    return 1;
  ModuleHandleW = GetModuleHandleW(L"kernel32.dll");
  if ( ModuleHandleW )
    v4 = GetProcAddress(ModuleHandleW, "wine_get_unix_file_name") != 0i64;
  else
    v4 = 0;
  v5 = v2 & !v4 & ((unsigned int)ab_checkWineRegistryEntry() == 0);
  if ( !v5 )
    return 1;
  v6 = v5 & ((unsigned int)ab_CheckVBoxHW() == 0);
  v7 = ((unsigned int)ab_CheckVBoxServices() == 0) & v6;
  v8 = ((unsigned int)ab_CheckVBoxFiles() == 0) & v7;
  v9 = v8 & !ab_CheckVBoxGuestAdditions();
  if ( !v9 )
    return 1;
  v10 = v9 & ((unsigned int)ab_checkMACAddress(L"\b") == 0);
  v11 = ((unsigned int)ab_checkVBoxDevPipes() == 0) & v10;
  WindowW = FindWindowW(L"VBoxTrayToolWndClass", 0i64);
  v13 = FindWindowW(0i64, L"VBoxTrayToolWnd");
  if ( WindowW || (v14 = 0, v13) )
    v14 = 1;
  v15 = (v14 ^ 1) & !ab_checkVBoxSharedFolders() & v11;
  v16 = ((unsigned int)ab_checkVBoxProcesses() == 0) & v15;
  v17 = ((unsigned int)ab_checkVBoxMacAddress() == 0) & v16;
  v18 = ((unsigned int)ab_checkVBoxVideoAdapter() == 0) & v17;
  v19 = ((unsigned int)ab_checkVirtualBox_1() == 0) & v18;
  v20 = ((unsigned int)ab_checkVirtualBox_2() == 0) & v19;
  v21 = ((unsigned int)ab_checkVBoxBuses() == 0) & v20;
  v22 = ((unsigned int)ab_checkVBoxManufacturer() == 0) & v21;
  v23 = ((unsigned int)ab_checkVBoxDevice_1() == 0) & v22;
  v24 = ((unsigned int)ab_checkVBoxDevice_2() == 0) & v23;
  v25 = v24 & ((unsigned int)ab_checkVBoxDevice_3() == 0);
```

```
44   if ( !v25 )
45     return 1;
46   v26 = (unsigned __int8)v25 & ((unsigned int)ab_checkVirtualPCProcesses() == 0);
47   if ( !v26 )
48     return 1;
49   v27 = v26 & ((unsigned int)ab_checkQEMURegistryEntries() == 0);
50   v28 = ((unsigned int)ab_checkQEMUProcesses() == 0) & v27;
51   v29 = ((unsigned int)ab_checkQEMUGuestTools() == 0) & v28;
52   v30 = ((unsigned int)ab_checkBOCHS() == 0) & v29;
53   v31 = v30 & ((unsigned int)ab_checkQEMU() == 0);
54   if ( !v31 )
55     return 1;
56   v32 = v31 & ((unsigned int)ab_checkVirtIOServices() == 0);
57   v33 = ((unsigned int)ab_checkVirtIODrivers() == 0) & v32;
58   v34 = v33 & !ab_checkVirtIODirectory();
59   if ( !v34 )
60     return 1;
61   v35 = v34 & ((unsigned int)ab_checkParallelsProcesses() == 0);
62   v36 = v35 & ((unsigned int)ab_checkMACAddress(byte_1800C4598) == 0);
63   if ( v36
64     && (v37 = (unsigned __int8)v36 & ((unsigned int)ab_DetectVirtualMachines() == 0)) != 0
65     && (v38 = (unsigned __int8)v37 & ((unsigned int)ab_CheckUserNames() == 0)) != 0
66     && (LOBYTE(v1) = !ab_CheckMemory(), (v1 & v38) != 0) )
67   {
68     return sub_18004FCB0(v40, v39);
69   }
70   else
71   {
72     return 1;
73   }
74 }
```

[Figure 34] Subroutine invoking different virtual machines checks.

Other few details I haven't commented about:

- **Process Explorer 64-bit** process (**procexp64.exe** -- from SysInternals) is checked whether it is running or not on **line 8**.

- wine_get_unix_file_name's address is retrieved on line 15 to be used with Wine detection function (**ab_checkWineRegistryEntry( )**).

As we've mentioned on **pages 24 and 28**, there's the usage of COM and WMI functions over several anti-analysis routines that are responsible for detecting tools and different hypervisor frameworks. Therefore, eventually it would be interesting to analyze one of these pieces of code involving such COM APIs.

At beginning of the **ab_DetectVirtualMachine** subroutine (**sub_18004FAB0**), the subroutine **sub_180050460** is called and, if readers step into the subroutine, they will find several **APIs related to COM** such as:

- **CoInitializeEx**
- **CoInitializeSecurity**
- **CoCreateInstance**
- **CoUninitialize**

- **CoSetProxyBlanket**

The respective code is shown below:

```
.text:0000000180050460                 mov     [rsp+arg_10], rsi
.text:0000000180050465                 push    rdi
.text:0000000180050466                 sub     rsp, 50h
.text:000000018005046A                 mov     rdi, rdx
.text:000000018005046D                 mov     rsi, rcx
.text:0000000180050470                 xor     edx, edx        ; dwCoInit
.text:0000000180050472                 xor     ecx, ecx        ; pvReserved
.text:0000000180050474                 call    cs:CoInitializeEx
.text:000000018005047A                 test    eax, eax
.text:000000018005047C                 jns     short loc_18005048B
.text:000000018005047E                 xor     eax, eax
.text:0000000180050480                 mov     rsi, [rsp+58h+arg_10]
.text:0000000180050485                 add     rsp, 50h
.text:0000000180050489                 pop     rdi
.text:000000018005048A                 retn
.text:000000018005048B ; ---------------------------------------------------------------------------
.text:000000018005048B
.text:000000018005048B loc_18005048B:                          ; CODE XREF: sub_180050460+1C↑j
.text:000000018005048B                                         ; DATA XREF: .rdata:00000001800E704C↓o ...
.text:000000018005048B                 mov     [rsp+58h+arg_8], rbp
.text:0000000180050490                 xor     r9d, r9d        ; pReserved1
.text:0000000180050493                 xor     ebp, ebp
.text:0000000180050495                 xor     r8d, r8d        ; asAuthSvc
.text:0000000180050498                 mov     [rsp+58h+pReserved3], rbp ; pReserved3
.text:000000018005049D                 or      edx, 0FFFFFFFFh ; cAuthSvc
.text:00000001800504A0                 mov     [rsp+58h+dwCapabilities], ebp ; dwCapabilities
.text:00000001800504A4                 xor     ecx, ecx        ; pSecDesc
.text:00000001800504A6                 mov     [rsp+58h+pAuthList], rbp ; pAuthList
.text:00000001800504AB                 mov     [rsp+58h+dwImpLevel], 3 ; dwImpLevel
.text:00000001800504B3                 mov     [rsp+58h+dwAuthnLevel], ebp ; dwAuthnLevel
.text:00000001800504B7                 call    cs:CoInitializeSecurity
.text:00000001800504BD                 test    eax, eax
.text:00000001800504BF                 js      short loc_1800504E4
.text:00000001800504C1                 lea     r9, stru_1800C4670 ; riid
.text:00000001800504C8                 mov     qword ptr [rsp+58h+dwAuthnLevel], rdi ; ppv
.text:00000001800504CD                 xor     edx, edx        ; pUnkOuter
.text:00000001800504CF                 lea     r8d, [rbp+1]    ; dwClsContext
.text:00000001800504D3                 lea     rcx, rclsid     ; rclsid
.text:00000001800504DA                 call    cs:CoCreateInstance
.text:00000001800504E0                 test    eax, eax
.text:00000001800504E2                 jns     short loc_1800504FC
```

[Figure 35] Subroutine including many calls using COM APIs.

We already mentioned and managed code involving COM in previous articles, but it might be useful to remember about few concepts related to the technology.

9. Few words about COM

In general words, **COM (Component Object Model)** was designed to work in a distributed computing model (Client/Server, RPC, distributed objects) and one of its goals was extending the communication concepts at that time, by offering advantages to developers for making easier to write COM objects in any language and exporting their respective functionalities to be consumed by programs in general.

A **COM object** exposes its interfaces (well-defined interfaces) to make easy and possible any client to use its services. In this case, it isn't important whether the client is or not at the same machine/system. Indeed, **an interface is only a set of functions**, and objects and clients can communicate to each other through them.

However, let's make it clear**: a client interacts with pointers to interfaces and don't have access to anything else inside the interface.** A **COM object** (**an instance of a class**) can implement **multiple interfaces, but it's also important to highlight that the class must implement all functions defined by an interface (even doing nothing).**

Server and clients can communicate to each other through a pair of **proxy** and **stub objects**, which the **proxy** works as a **server representation of a remote server running in the client address space** and the **stub** is a sort of **listener running on the server side and waiting for client requests**. In other words, a possible scheme to illustrate it is the following one:

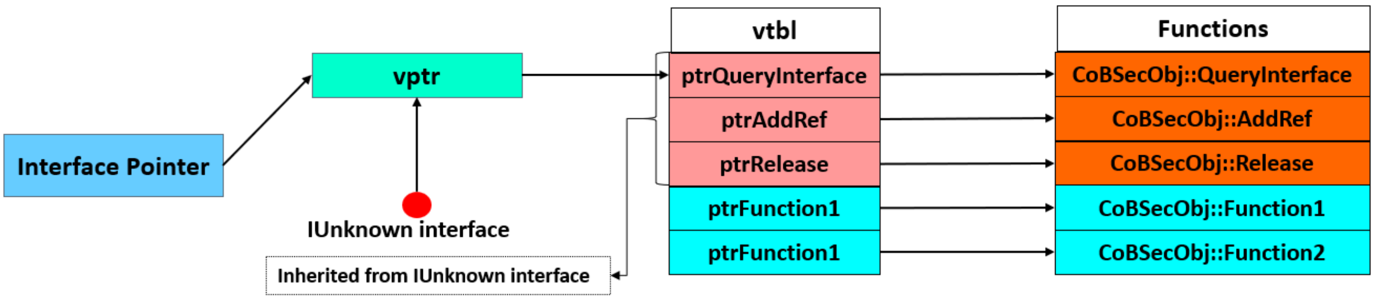
- **invoke method -> interface -> local proxy -> remote stub -> interface -> remote method**

Once again, clients consume services through interfaces offered by COM object, but don't have any idea of services' implementation, which is a well-known encapsulation example (as already mentioned earlier).

A **COM object** is an instance of a **COM class (classes can be instantiated, but interfaces can't)**. It represents an object definition and implements the **IUnknown interface, which is the base interface of all interfaces in COM** and that, at least, supports **QueryInterface** method (used for service discovering). In other words, the **QueryInterface( )** returns a pointer to the requested interface back to the client.

A **COM component**, which is a binary module represented by an **executable** (**the server is implemented as a standalone executable module**) or **DLL (the server is implemented as a module to be loaded and executed within the address space of the client)**, also supports concepts of **inheritance** and **polymorphism** besides other security features, offered by interfaces, such as access control and impersonation. The **COM's lifetime** is managed by the usage of reference count through **IUnknown::AddRef** and **IUnknown::Release** methods.

The general idea on how a COM binary is built up is shown below:



[Figure 36] Representation of part of a COM object (adapted, but based on COM Specification and "Learning DCOM" book by L. Thai Thuan, which was released in 1999)

Few points of the image above:

- **vtbl:** virtual table (table of function pointers).
- **vptr:** virtual table pointer to the **vtbl**.

- **IUnknown interface:** as explained, it's the parent of all COM interfaces and it's used, through the **IUnknown::QueryInterface** method, to find dynamically other interfaces and perform lifetime management through **IUnknown::AddRef** and **IUnknown::Release** methods.

Therefore, **each instantiated object has an own associated vptr**, although **there's only one vtbl per class**.

To recap:

- An **interface** is composed by one or more functions.
- A **class** is the implementation of one or more interfaces.
- An **object** is the instance of a COM class.
- A **binary** is composed by one or more COM classes.
- **Classes** can be instantiated, but **interfaces** can't be instantiated.
- Each instantiated object has an own **vptr**.
- There's only one **vtbl** per class.
- A **COM component** can be represented by an **executable** (standalone process) or **DLL** (loaded into the client's process).
- **Proxy** runs in the client side and **stub** runs on the server side as a listener.

There's a different kind of COM object named **Factory** that is responsible for creating and/or instantiating other **COM objects** associated with a determined **COM class**. In this case, standard factories are represented by **IClassFactory interface**, which is derived from **IUnknown interface.** Requests received by **IClassFactory** to instantiate a COM object triggers **IClassFactory::CreateInstance**, which is responsible for accomplishing the task**.** Thus, we have that: **COM Factory -> COM Object**.

Class factories and **COM objects** can be packed into either an executable or DLL file. If they are packed as an executable, so they can run as a service or local/remote server in a different execution context from COM client. Furthermore, executable **COM classes** can be registered using **CoRegisterClassObject( ).** If they are packed as a DLL, so it's an **in-process server** and are usually loaded by the client.

In the COM world, a function can be invoked using two different approaches:

- **Static Invocation:**

  - That's the invocation of a method (and respective signature) in compile-time.
  - All methods are called through a vtbl: **interface pointer -> vptr --> vtbl**
  - There's an alternative technique to invoke methods through type libraries (from an interface repository) and dispatch interface as well known as **IDispatch**.
  - All methods from a dispatch interface are called through a dispatch identifier (**dispid**).

- **Dynamic Invocation:**

  - This method is also known as **late binding**.
  - It's supported by **IDispatch interface**, which can be used to discover and invoke methods. However, to use this method demands a **lookup operation for** the **dispid**.

In other words, an **application interface is derived from IUnknown interface** and, of course, **inherits pure virtual functions that need to be implemented**.

There're many COM functions and, among them, we have **CoCreateInstance( )**, which creates a **COM object**, is one of most important, for sure:

```
HRESULT CoCreateInstance(
  REFCLSID  rclsid,
  LPUNKNOWN pUnkOuter,
  DWORD     dwClsContext,
  REFIID    riid,
  LPVOID    *ppv
);
```

[Figure 37] CoCreateInstance function

Its parameters, as described on MSDN, are:

- **rclsid:** The CLSID associated with the data and code that will be used to create the object.
- **pUnkOuter:** If NULL, indicates that the object is not being created as part of an aggregate. If non-NULL, pointer to the aggregate object's **IUnknown interface** (the controlling **IUnknown**).
- **dwClsContext:** Context in which the code that manages the newly created object will run.
- **riid:** A reference to the identifier of the interface to be used to communicate with the object.
- **ppv:** Address of pointer variable that receives the interface pointer requested in riid**.**

We have strict interest on three these parameters: **rclsid, riid and ppv.** The **clsid** and **riid** are referenced by a respective **GUID (128-bit hexadecimal**), which each one of them are unique (eliminates any chance of name collision), and that classes and interfaces can be referred . One key aspect of interfaces is that they are immutable then they are not versioned.

COM classes are registered into the operating system and identified by these GUIDs, which are used as a representation of the class within Registry:

- **HKLM\Software\Classes\CLSID**
- **HKCU\Software\Classes\CLSID**

As readers likely already learned from other articles, during a COM hijacking attack, malware and adversaries could establish persistence by replacing a legit COM entry in the Registry or even enumerating CLSID subkeys such as **LocalServer32** and **InProcServer32** to discover abandoned binary references, which is not so rare due to failed uninstallation processes.

In terms of nomenclature associated to Registry:

- **Server:** it's a binary that's referred by the CLSID inside of Registry.
- **LocalServer32 key:** it's the path to an executable (.exe) implementation.
- **InProcServer32 key:** it's the path to a dynamic linked library (.dll) implementation.

The fundamental COM APIs used to write COM clients are:

- **CoInitialize( )/CoInitializeEx( )**

- **CoCreateInstance( )**
- **CoUninitialize( )**

The **CoCreateInstance( )** calls internally the **CoGetClassObject( )** to get a class factory that through **IClassFactory::CreateInstance** to create the requested COM object**.** Moreover, **CoGetClassObject( )** is commonly used to create multiple objects for a given class object .

10. Reversing: second part

Now we can resume from where we stopped (the subroutine **sub_180050460** and, more precisely, in the code referred by **Figure 35**), and handle details related to COM functions.

Right before the calling for **CoCreateInstance( )**, there're five arguments passed to the function, which **riid, rclsid** and **ppv** are the most important ones:

```
01800504C1                lea     r9, stru_1800C4670 ; riid
01800504C8                mov     qword ptr [rsp+58h+dwAuthnLevel], rdi ; ppv
01800504CD                xor     edx, edx          ; pUnkOuter
01800504CF                lea     r8d, [rbp+1]      ; dwClsContext
01800504D3                lea     rcx, rclsid       ; rclsid
01800504DA                call    cs:CoCreateInstance
```

[Figure 38] Arguments passed to CoCreateInstance function

Examining **rclsid** and **riid** we have the following:

```
  rdata:00000001800B1418 ; const IID rclsid
> rdata:00000001800B1418 rclsid          dd 4590F811h            ; Data1
  rdata:00000001800B1418                                         ; DATA XREF: sub_18004B560+54↑o
  rdata:00000001800B1418                                         ; sub_18004C1D0+53↑o ...
  rdata:00000001800B1418                 dw 1D3Ah                ; Data2
  rdata:00000001800B1418                 dw 11D0h                ; Data3
  rdata:00000001800B1418                 db 89h, 1Fh, 0, 0AAh, 0, 4Bh, 2Eh, 24h; Data4
  rdata:00000001800B1428 ; const IID riid
> rdata:00000001800B1428 riid            dd 0DC12A687h           ; Data1
  rdata:00000001800B1428                                         ; DATA XREF: sub_18004B560+47↑o
  rdata:00000001800B1428                                         ; sub_18004C1D0+46↑o ...
  rdata:00000001800B1428                 dw 737Fh                ; Data2
  rdata:00000001800B1428                 dw 11CFh                ; Data3
  rdata:00000001800B1428                 db 88h, 4Dh, 0, 0AAh, 0, 4Bh, 2Eh, 24h; Data4
```

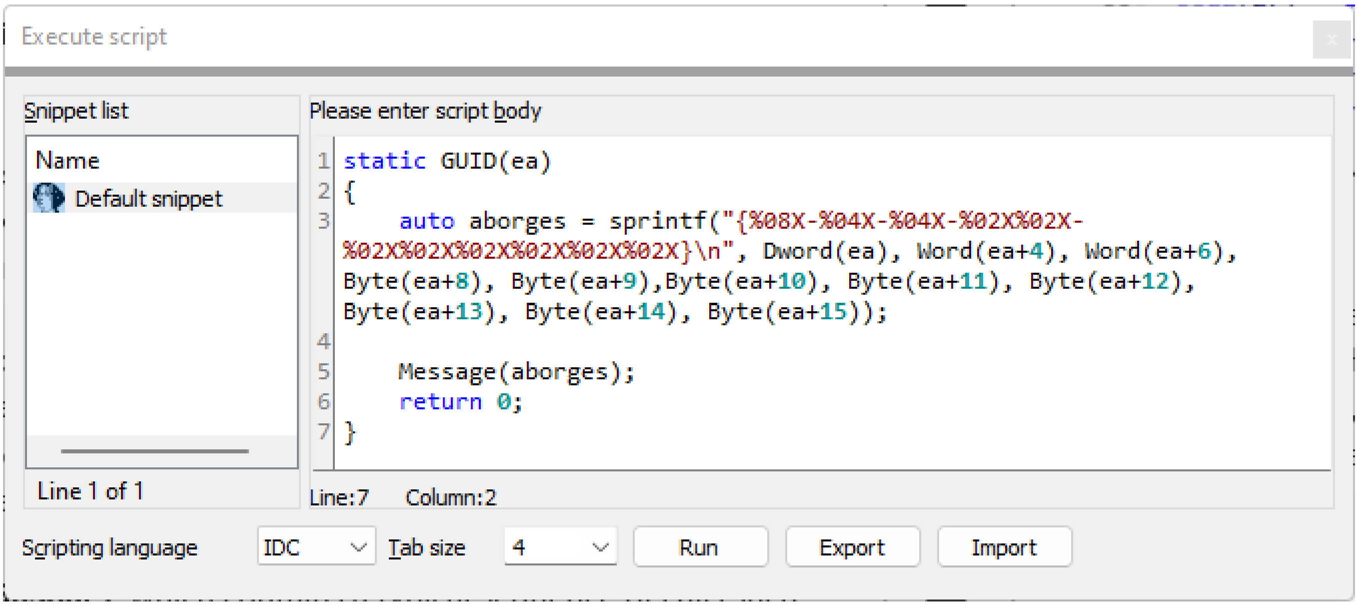
[Figure 39] Content of arguments passed to CoCreateInstance function

Both **rclsid** and **riid** are represented by a **GUID struct**, which has the following composition:

```
typedef struct _GUID {
  unsigned long  Data1;
  unsigned short Data2;
  unsigned short Data3;
  unsigned char  Data4[8];
} GUID;
```

[Figure 40] GUID structure

To decode **GUID**s (as known as **UUID – Universally Unique Identifier**) we can use a simple script written in **IDC** as shown below:

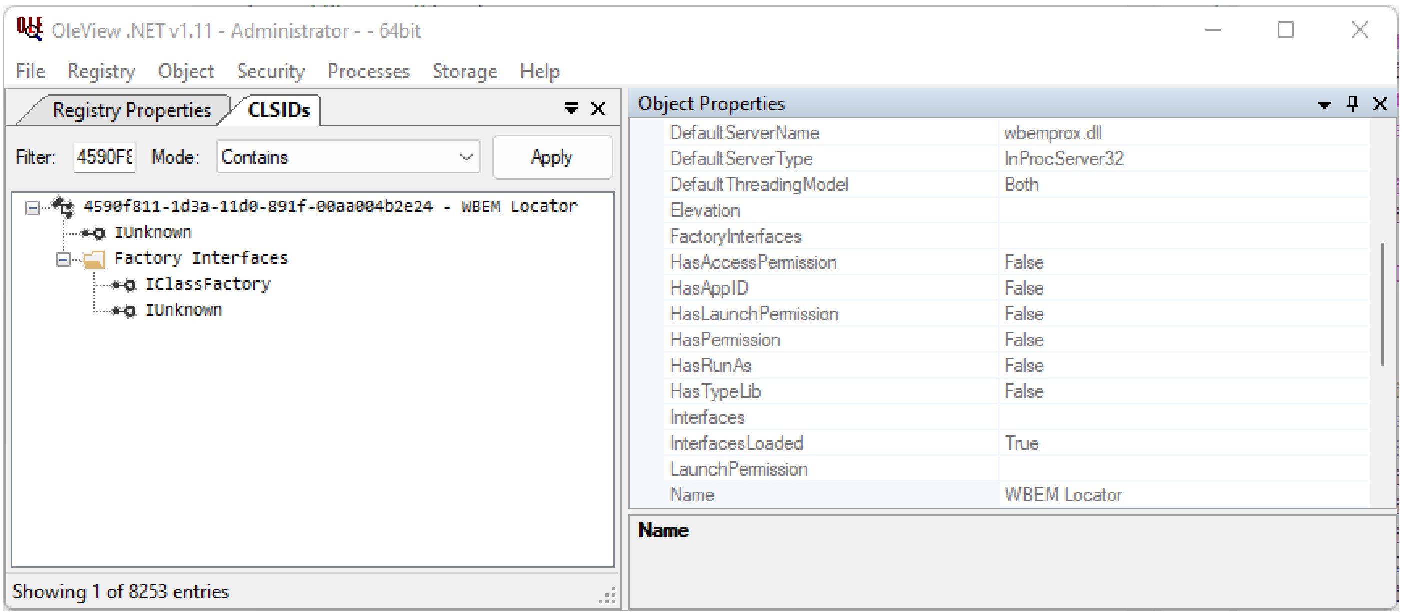


[Figure 41] IDC script to decode GUID Structure

Once the script has been executed then we can use the **GUID( ) function**. To decode the CLSID and IID we must pass the address of the start of their structures as shown below:

- **GUID(<rclsid address>) = GUID(0x00000001800B1418): 4590F811-1D3A-11D0-891F-00AA004B2E24**
- **GUID(<riid address>) = GUID(0x00000001800C4670): DC12A687-737F-11CF-884D-00AA004B2E24**

There're many ways to get the appropriate information and meaning of found CLSID and IID, and one of them is using the **OleView .NET** (https://github.com/tyranid/oleviewdotnet) or **COMView** (https://www.japheth.de/COMView.html). For example, searching for the CLSID on **OleView .NET** we have:



[Figure 42] OleView .NET: searching for Class ID details

Of course, we always have the option (and, in many times, it's an easier way) to google it on the Internet and soon we can find some important and related information:

- https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-wmi/46710c5c-d7ab-4e4c-b4a5-ebff311fdcd1.

Personally, I like to search about **COM information** and definitions on the following Microsoft website:

- https://referencesource.microsoft.com

Querying by the **interface ID** (**DC12A687-737F-11CF-884D-00AA004B2E24**) we got:

```
[InterfaceTypeAttribute(0x0001)]
[TypeLibTypeAttribute(0x0200)]
[GuidAttribute("DC12A687-737F-11CF-884D-00AA004B2E24")]
[ComImport]
interface IWbemLocator
{
    [PreserveSig] int ConnectServer_([In][MarshalAs(UnmanagedType.BStr)]  string   strNetworkResource,
[In][MarshalAs(UnmanagedType.BStr)]  string   strUser, [In]IntPtr   strPassword,
[In][MarshalAs(UnmanagedType.BStr)]  string   strLocale, [In] Int32 lSecurityFlags,
[In][MarshalAs(UnmanagedType.BStr)]  string   strAuthority, [In][MarshalAs(UnmanagedType.Interface)]
IWbemContext   pCtx, [Out][MarshalAs(UnmanagedType.Interface)]  out IWbemServices   ppNamespace);
}
```

As a summary, we have:

- Class: **WbemLocator**
- InProcServer32: **C:\Windows\system32\wbem\wbemprox.dll**
- Interface: **IWbemLocator**
- Explanation: **IWbemLocator interface** is used to get a namespace pointer to **IWbemServices interface for WMI** and, getting this pointer, we can access **Windows Management** by calling **IWbemLocator::ConnectServer**.

Next step is to use this information to make the reversed code by **IDA Pro** easier to understand. There're two paths to proceed and the composition them can help us a lot:

a. The first method consists in changing the type (**"Y" hotkey**) of **ppv** parameter. It's used as the return of **CoCreateInstance function** and contains the requested interface point, and as its type is **LPVOID**, so it's necessary to make a cast. Thus, change it from from **"LPVOID"** to "**IWbemLocator***" as well any variable receiving its value. No doubts, performing this task on pseudo code produced by the **IDA Decompiler** is always recommended, but it isn't enough. Why? Because new functions will come up and their arguments will need to be adjusted too.

b. The second path that could help us to get a better result is done by executing the following steps:

    a. If the structure isn't already added, so insert it (**SHIFT+F9** and then **INS**) using the following nomenclature: **<interface name>.vbtl**

b. use the **“T” hotkey** to apply the structure over the disassembled code (mainly for calls).

Before applying changes, it’s relevant to show the pseudo code from IDA Decompiler:

```
__int64 __fastcall sub_180050460(IUnknown **a1, LPVOID *a2)
{
  BSTR v5; // rax
  OLECHAR *v6; // rbx

  if ( CoInitializeEx(0i64, 0) < 0 )
    return 0i64;
  if ( CoInitializeSecurity(0i64, -1, 0i64, 0i64, 0, 3u, 0i64, 0, 0i64) < 0
    || CoCreateInstance(&rclsid, 0i64, 1u, &stru_1800C4670, a2) < 0 )
  {
    CoUninitialize();
    return 0i64;
  }
  v5 = SysAllocString(L"ROOT\\CIMV2");
  v6 = v5;
  if ( v5 )
  {
    if ( (*(int (__fastcall **)(_QWORD, BSTR, _QWORD, _QWORD, _QWORD, int, _QWORD, _QWORD, IUnknown **))(*(_QWORD *)*a2 + 24i64))(
           *a2,
           v5,
           0i64,
           0i64,
           0i64,
           128,
           0i64,
           0i64,
           a1) < 0 )
    {
      SysFreeString(v6);
LABEL_12:
      (*(void (__fastcall **)(_QWORD))(*(_QWORD *)*a2 + 16i64))(*a2);
      CoUninitialize();
      return 0i64;
    }
    SysFreeString(v6);
  }
  if ( CoSetProxyBlanket(*a1, 0xAu, 0, 0i64, 3u, 3u, 0i64, 0) >= 0 )
    return 1i64;
  ((void (__fastcall *)(_QWORD))(*a1)->lpVtbl->Release)(*a1);
  goto LABEL_12;
}
```

[Figure 43] Decompiled sub_180050460 before applying any change

According to the previous page (based on query from https://resourcesource.microsoft.com website), **IWbemLocator interface** has only one method: **IWbemLocator::ConnectServer.** This method creates a connection to a WMI namespace that is named **strNetworkResource**. The returned value is only a **HRESULT** value that indicates the status of the method call. The prototype of **IWbemLocator::ConnectServer** method follows below:

```
HRESULT ConnectServer(
  const BSTR    strNetworkResource,
  const BSTR    strUser,
  const BSTR    strPassword,
  const BSTR    strLocale,
  long          lSecurityFlags,
  const BSTR    strAuthority,
  IWbemContext  *pCtx,
  IWbemServices **ppNamespace
);
```

[Figure 44] IWbemLocator::ConnectServer

After applying a list of changes, which are explained in the next paragraphs, we get the following result:

```
__int64 __fastcall sub_180050460(IWbemServices **ppNamespace, IWbemLocator **ppv)
{
  OLECHAR *strNetworkResource; // rax
  OLECHAR *strNetworkResource2; // rbx

  if ( CoInitializeEx(0i64, 0) < 0 )
    return 0i64;
  if ( CoInitializeSecurity(0i64, -1, 0i64, 0i64, 0, 3u, 0i64, 0, 0i64) < 0
    || CoCreateInstance(&rclsid, 0i64, 1u, &riid_0, ppv) < 0 )
  {
    CoUninitialize();
    return 0i64;
  }
  strNetworkResource = SysAllocString(L"ROOT\\CIMV2");
  strNetworkResource2 = strNetworkResource;
  if ( strNetworkResource )
  {
    if ( (*ppv)->lpVtbl->ConnectServer(
           (IWbemLocator **)*ppv,
           strNetworkResource,
           0i64,
           0i64,
           0i64,
           128,
           0i64,
           0i64,
           ppNamespace) < 0 )
    {
      SysFreeString(strNetworkResource2);
LABEL_12:
      ((void (__fastcall *)(IWbemLocator *))(*ppv)->lpVtbl->Release)(*ppv);
      CoUninitialize();
      return 0i64;
    }
    SysFreeString(strNetworkResource2);
  }
  if ( CoSetProxyBlanket((IUnknown *)*ppNamespace, 0xAu, 0, 0i64, 3u, 3u, 0i64, 0) >= 0 )
    return 1i64;
  (*ppNamespace)->lpVtbl->Release((IWbemServices **)*ppNamespace);
  goto LABEL_12;
```

[Figure 45] sub180050460: parameters renamed and re-typed

I've changed both pseudo and assembly code. On pseudo-code, the most important changes have done by:

- Renaming (**"N" hotkey**) **a2** to **ppv** because the names of **CoCreateInstance( )** on MSDN.
- Changing ppv's type (**"Y" hotkey**) to **IWbemLocator **** (check the interface type that we found and described on page **37**).
- Editing the **CoCreateInstance( )** signature (**"Y" hotkey**) for changing the type of **ppv** to **IWbemLocator ****.
- Renaming (**"N" hotkey**) **a1** to **ppNamespace** (following the same name of the last argument for **IWbemLocator::ConnectServer( )** from MSDN).

- Changing the type (**"Y" hotkey**) of **ppNamespace** argument to **IWbemServices **** according to the expected name of the function **IWbemLocator::ConnectServer** described on MSDN.

- Editing the signature (**"Y" hotkey**) of the **IWbemLocator::ConnectServer** and changing type of the **ppNamespace argument** to **IWbemServices **** according to the expected name of the function described on MSDN.

On the assembly side, I've used the **T hotkey** to apply the correct type for any call instruction using i**ndirection** within **this subroutine (sub_180050460)** and according to the **pseudo code.**

If readers compare **Figures 43 and 45**, so you'll notice a better result and, this time, it's possible to interpreter the code, although it would be possible to improve it.

There're some necessary explanations about what's happening so far:

- **CoCreateInstance( )** creates one object that is an instance of a given class (**WbemLocator** , given by the **CLSID**) on the local system .
- The **class is the implementation of one or more COM interfaces** and the interface is given by the **iid** parameter (**IWbemLocator**).
- Remember that an **interface is really a class containing functions defined as pure virtual**, which must be implemented by an **implementation class.**
- The **clients access the virtual pointer (vptr)** that is a reference to the **virtual table** containing functions' pointers to real functions.
- The final **COM binary (DLL / exe)** can be composed **by one or more classes**.
- Clients **don't communicate with the class directly, but through interfaces**. At same way, **clients don't need to know where a COM object is located** nor its **respective implementation** because, as stated, the **interface is the main point of contact**.
- All interfaces inherit from the **IUnknow interface**, which has three methods: **AddRef( ), Release( ) and QueryInterface( ).**
- The **QueryInterface( )** aims to query an object for a given interface.
- The **CoCreateInstance( )** returns, as output in the **ppv parameter**, an interface pointer to the **IWbemLocator**.
- The **IWbemLocator interface** has only one method that's **ConnectServer( )**, which is responsible for creating a connection to a WMI namespace using its first parameter (**strNetworkResource**).
- The output from **IWbemLocator::ConnectServer( )** is its **seventh parameter** (**Figure 44**), which receives a pointer to an **IWbemServices** object. That's the reason by which we changed its type.
- So far, the malware code can connect to WMI and, in specific, to **ROOT\CIMv2 namespace** and get a pointer to **IWbemServices interface** and make next calls.
- The **CoSetProxyBlanket( ) function** is used to set the authentication information that will be used to make calls through a proxy. Its **fourth** and **fifth** parameters with **value 3** are saying that it's authenticating only at beginning of each RPC when the server receives the request, and the server process can impersonate the client's security context while acting on behalf of the client.

As readers can realize, a simple piece of code can contain many subtle information and concepts that might help us to have a better understanding of the what's happening. Of course, analyzing each piece of code in details might be time-consuming, but in several opportunities there isn't other alternative.

Continuing our analysis, go up and return to **ab_DetectVirtualMachines( ) subroutine** (**sub_18004FAB0**) and realize that there're other many functions related to COM. The key starting point is on the **line 9**, where the subroutine **sub_180050460**, which just analyzed, is called. As we learned, the **sub_180050460's parameters** were a pointer to **IWbemService interface** and a pointer to **IWbemLocator interface**. Thus, we can use them and perform the same job we did previously using MSDN to learn about associated types and outputs, and we obtained the following result:

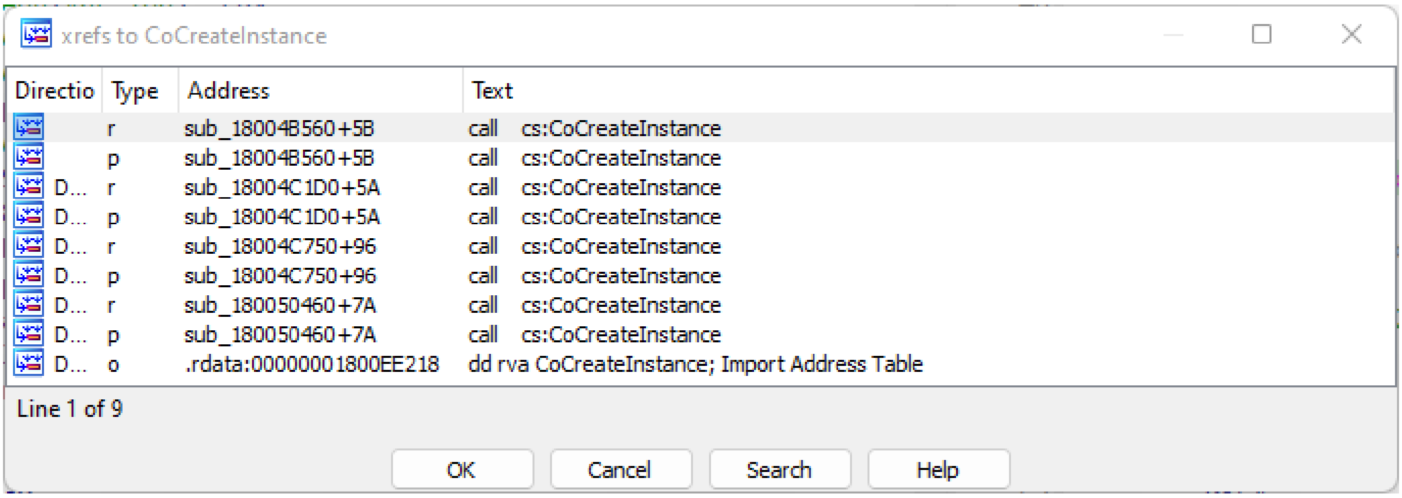
```
1  __int64 ab_DetectVirtualMachines()
2  {
3    // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]
4
5    result = 0;
6    ptr_IWbemServices = 0i64;
7    ptr_IWbemLocator = 0i64;
8    ppEnum = 0i64;
9    if ( !(unsigned int)sub_180050460(&ptr_IWbemServices, &ptr_IWbemLocator) )
10     return result;
11   var_wql = SysAllocString(L"WQL");
12   v_query = SysAllocString(L"SELECT * FROM Win32_ComputerSystem");
13   status = 1;
14   v_query_1 = v_query;
15   if ( var_wql )
16   {
17     if ( v_query && ptr_IWbemServices->lpVtbl->ExecQuery(ptr_IWbemServices, var_wql, v_query, 48, 0i64, &ppEnum) < 0 )
18     {
19       status = 0;
20       ptr_IWbemServices->lpVtbl->Release((IWbemServices **)ptr_IWbemServices);
21       ptr_IWbemLocator->lpVtbl->Release(ptr_IWbemLocator);
22       CoUninitialize();
23     }
24     SysFreeString(var_wql);
25   }
26   if ( v_query_1 )
27     SysFreeString(v_query_1);
28   if ( !status )
29     return result;
30   IEnumWbemClassObject = ppEnum;
31   This = 0i64;
32   for ( puReturned = 0; ppEnum; IEnumWbemClassObject = ppEnum )
33   {
34     IEnumWbemClassObject->lpVtbl->Next(IEnumWbemClassObject, -1, 1u, &This, &puReturned);
35     if ( !puReturned )
36       break;
37
38     if ( This->lpVtbl->Get(This, L"Model", 0, &pVal, 0i64, 0i64) >= 0 )
39     {
40       if ( pVal.vt == 8
41         && (StrStrIW(pVal.bstrVal, L"VirtualBox")
42          || StrStrIW(pVal.bstrVal, L"HVM domU")
43          || StrStrIW(pVal.bstrVal, L"VMWare")) )
44       {
45         VariantClear(&pVal);
46         ((void (__fastcall *)(IWbemClassObject *))This->lpVtbl->Release)(This);
47         result = 1;
48         break;
49       }
50       VariantClear(&pVal);
51     }
52     ((void (__fastcall *)(IWbemClassObject *))This->lpVtbl->Release)(This);
53   }
54   ((void (__fastcall *)(IWbemServices *))ptr_IWbemServices->lpVtbl->Release)(ptr_IWbemServices);
55   ((void (__fastcall *)(IWbemLocator *))ptr_IWbemLocator->lpVtbl->Release)(ptr_IWbemLocator);
56   ((void (__fastcall *)(IEnumWbemClassObject *))ppEnum->lpVtbl->Release)(ppEnum);
57   CoUninitialize();
58   return result;
59 }
```

[Figure 46] sub_18004FAB0 (ab_DetectVirtualMachines): after re-typing and renaming operations

Please, readers should observe I used a feature of IDA Pro named "*Collapsing Local Declarations*" before taking a snapshot, so readers should do the same to match the same lines which I'm referring to. Afterwards, few comments follow:

- **On line 9**, the type declaration of the first parameter of the call for **sub_180050460** is **IWbemServices *ptr_IWbemServices**.
- **On line 9**, the type declaration of the second parameter of the for **sub_180050460** is **IWbemLocator *ptr_IWbemLocator**.
- **On line 12**, string containing the **WQL query** is formed ("**SELECT * FROM Win32_ComputerSystem**"). The **Win32_ComputerSystem class** holds a series of members (properties) representing information from the local system.
- The **IWbemServices::ExecQuery( )** on **line 17** executes the given query above to retrieve possible objects. The output of this method in stored into the **fifth (and last) parameter (ppEnum)**, with has the following type's declaration: **IEnumWbemClassObject *ppEnum**. In general words, this last parameter (**ppEnum**) holds an **enumerator** that will be used to access query results. Readers should notice that this parameter is **copied to a variable on line 30**.
- Using the obtained enumerator through **ExecQuery( )**, the code parses each available property by using **IWbemClassObject::Next( )** on **line 34**.
- **On line 38**, it's possible to get properties values using **IWbemClassObject::Get method**. In this case, the code is checking **"Model" property** and looking for strings such as "**VirtualBox", "HVM domU"** and **"VMware"**.
- After having all checking's, the code releases all objects by using the inherited **Release( ) method** from **IUnknown interface** and closes the COM library on the current thread by calling **CoUninitialize( ).**

Readers will find COM code over several different malware code, and I hope it can help you to get a better understanding about how to analyze COM functions. If the reader to check other parts of the code that are using **CoCreateInstance( ),** so you will be able apply a similar approach:



[Figure 47] Finding new subroutines calling CoCreateInstance

Let's proceed with our analysis and continuing our investigation. The **ab_DetectVirtualMachines( ) subroutine** (**sub_18004FAB0** -- check **Figure 43**) is being called (list **cross-references to** the function using **X hotkey**) by **ab_checkVirtualMachinesAndTools( )** (**sub_18004CD50** --- check **Figure 31**).

According to the experience, this kind of checking for virtual environments is usually performed at the beginning of the malware execution and, most of times, right before something useful being done by the malicious code. Therefore, by listing which subroutines are calling **sub_18004CD50,** we reach the subroutine **sub_18000A120** and, apparently, new findings are possible:

```
__int64 __fastcall sub_18000A120(void *a1)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  v208[4] = -2i64;
  v162[3] = (void *)15;
  v162[2] = 0i64;
  LOBYTE(v162[0]) = 0;
  if ( hHandle )
    WaitForSingleObject(hHandle, 0xFFFFFFFF);
  if ( qword_180104980 )
    sub_180006F80(v162, (void **)&byte_180104970, 0i64, 0xFFFFFFFFFFFFFFFFui64);
  if ( ab_checkVirtualMachinesAndTools() )
    goto LABEL_391;
  v1 = time64(0i64);
  srand(v1);
  qword_1801048D8 = (HANDLE)beginthreadex(0i64, 0, (_beginthreadex_proc_type)w_DetectRunningTools, 0i64, 0, &ThrdAddr);
  *(_QWORD *)v121 = 15i64;
  *(_QWORD *)v120 = 0i64;
  LOBYTE(v119[0]) = 0;
  if ( aVcffi2rj6t15[0] )
  {
    v2 = -1i64;
    do
      ++v2;
    while ( aVcffi2rj6t15[v2] );
  }
  else
  {
    v2 = 0i64;
  }
  sub_180006E50((unsigned __int64 *)v119, aVcffi2rj6t15, v2);
  if ( *(_QWORD *)v120 )
  {
    v3 = v119;
    if ( *(_QWORD *)v121 >= 0x10ui64 )
      v3 = (__int64 *)v119[0];
    sub_1800012D0(v224, (__int64)v3, v120[0]);
    sub_180001670(v224, (__int64)asc_1800FF830, 79);
    v4 = v119;
    if ( *(_QWORD *)v121 >= 0x10ui64 )
      v4 = (__int64 *)v119[0];
    sub_1800012D0(v226, (__int64)v4, v120[0]);
    sub_180001670(v226, (__int64)byte_1800FF790, 79);
    v5 = v119;
    if ( *(_QWORD *)v121 >= 0x10ui64 )
      v5 = (__int64 *)v119[0];
    sub_1800012D0(v225, (__int64)v5, v120[0]);
    sub_180001670(v225, (__int64)byte_1800FF7E0, 79);
    v6 = v119;
    if ( *(_QWORD *)v121 >= 0x10ui64 )
      v6 = (__int64 *)v119[0];
    sub_1800012D0(v195, (__int64)v6, v120[0]);
    sub_180001670((unsigned __int8 *)v195, (__int64)byte_1800FF340, 1023);
    sub_1800014D0((__int64)v195);
    sub_1800014D0((__int64)v225);
    sub_1800014D0((__int64)v226);
    sub_1800014D0((__int64)v224);
  }
```

[Figure 48] A piece of sub_18000A120 subroutine

11. Reversing: difficulties during the analysis

Soon at the beginning of **sub_18000A120** subroutine, we have a call for the **sub_180006F80** subroutine and, for this specific function, there're four arguments. Once we move inside it, we find the following:

```
void **__fastcall sub_180006F80(void **a1, void **a2, unsigned __int64 a3, unsigned __int64 a4)
{
  char *v4; // rax
  unsigned __int64 v5; // rdi
  void **v7; // rsi
  void **v8; // rbx
  char *v9; // rax
  void *v10; // rax
  void *v11; // rcx
  bool v12; // cf
  _BYTE *v13; // rax

  v4 = (char *)a2[2];
  v5 = a4;
  v7 = a2;
  v8 = a1;
  if ( (unsigned __int64)v4 < a3 )
    sub_180050974((__int64)"invalid string position");
  v9 = &v4[-a3];
  if ( a4 > (unsigned __int64)v9 )
    v5 = (unsigned __int64)v9;
  if ( a1 == a2 )
  {
    v10 = (void *)(a3 + v5);
    if ( (unsigned __int64)a1[2] < a3 + v5 )
      sub_180050974((__int64)"invalid string position");
    a1[2] = v10;
    if ( (unsigned __int64)a1[3] >= 0x10 )
      a1 = (void **)*a1;
    *((_BYTE *)v10 + (_QWORD)a1) = 0;
    sub_1800072A0(v8, 0i64, a3);
  }
```

[Figure 49] First pseudo instructions of sub_180006F80 subroutine

As readers notice, there's a repeated call for subroutine **sub_180050974** with the string *"invalid string position"* as argument, but IDA Pro didn't identify the correct function for us. Additionally, there're other functions within **sub_180006F80 subroutine** that accepts strings arguments and also hasn't been identified.

If the malware's author had used standard libraries from **Microsoft / Visual C++,** almost certainly IDA Pro would have identified successfully. Therefore, we can make a hypothesis that external libraries were used to generate the final malicious code.

In next few pages I'm going to scratch the surface of this topic and, certainly, there're much more to be explained and demonstrated, but I'll restrict the focus on few approaches in this article.

One good resource to help us is the usage of **Lumina**, from IDA Pro, to populate and enrich the database with further information. In few words, Lumina server provides function name, prototypes, comments, operand types and other information about functions shared by Hex-Rays and other researchers. No doubts, it's possible alternative for us and it can always test it.

Using Lumina is quite easy and simple. To apply Lumina definitions, go to **Lumina menu** and pick up **pull all metadata** option (**F12 hotkey**) and Lumina's metadata will enrich the **idb database** (most of them marked in green). For example, at first lines of **sub_18000A120** we have as result:

```
1 __int64 __fastcall sub_18000A120(void *a1)
2 {
3   // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]
4 
5   v208[4] = -2i64;
6   v162[3] = (void *)15;
7   v162[2] = 0i64;
8   LOBYTE(v162[0]) = 0;
9   if ( hHandle )
10    WaitForSingleObject(hHandle, 0xFFFFFFFF);
11  if ( qword_180104980 )
12    std::string::assign(v162, (void **)&byte_180104970, 0i64, 0xFFFFFFFFFFFFFFFFui64);
13  if ( ab_checkVirtualMachinesAndTools() )
```

[Figure 50] First lines of sub_18000A120 including Lumina changes

I've highlighted the **line 12** because it's exactly one of additions provided by Lumina. Examining **std::string::assign** function (**sub_18000A120 subroutine**) we have:

```
std::string *__fastcall std::string::assign(
        std::string *this,
        const std::string *_Right,
        unsigned __int64 _Roff,
        unsigned __int64 _Count)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  v4 = *((_QWORD *)_Right + 2);
  v5 = _Count;
  v7 = _Right;
  v8 = this;
  if ( v4 < _Roff )
    Catch::throw_exception<std::domain_error>("invalid string position");
  v9 = v4 - _Roff;
  if ( _Count > v9 )
    v5 = v9;
  if ( this == _Right )
  {
    v10 = _Roff + v5;
    if ( *((_QWORD *)this + 2) < _Roff + v5 )
      Catch::throw_exception<std::domain_error>("invalid string position");
    *((_QWORD *)this + 2) = v10;
    if ( *((_QWORD *)this + 3) >= 0x10ui64 )
      this = *(std::string **)this;
    *((_BYTE *)this + v10) = 0;
    std::string::erase(v8, 0i64, _Roff);
  }
  else
  {
    if ( v5 == -1i64 )
      Catch::throw_exception<std::domain_error>("string too long");
```

[Figure 51] First lines of std::string::assign function

If metadata fetched my Lumina is correct (never believe it blindly), our first finding is that the malware have used the **Catch unit testing framework for C++**. If readers don't know about **Catch**, there're good references to it:

- https://catch2.docsforge.com/
- https://github.com/catchorg/Catch2

Initially, examining all functions from database, it seems that there's only one Catch routine. Of course, Lumina brought much more metadata for many functions and all of them can help us during our analysis:

Function name	Segment	Start	Length	Loc
SQEX::Luminous::Core::IO::Path::GetHostPCPath(cha...	.text	00000001800387B0	0000005C	000
windows_file_codecvt::do_in(_Mbstatet &,char const *...	.text	0000000180038A90	0000008E	000
windows_file_codecvt::do_out(_Mbstatet &,wchar_t c...	.text	0000000180038B20	000000A5	000
Black::Actor::Actor::PlayVoice(int,uint,bool,bool,bool,fl...	.text	0000000180038EB0	0000005F	000
std::vector<std::_List_unchecked_iterator<std::_List_...	.text	000000018003B720	0000003E	000
Scallion::Detail::String::~String(void)	.text	000000018003B7C0	0000002D	000
std::_Yarn<char>::operator=(char const *)	.text	000000018003B7F0	0000008C	000
std::wstring::_Assign_rv(std::wstring &&)	.text	000000018003BAE0	0000008C	000
std::string::insert(unsigned __int64,unsigned __int64,...	.text	000000018003BC10	00000167	000
std::_Wrap_alloc<std::_Wrap_alloc<std::allocator<st...	.text	000000018003BF10	0000006D	000
std::_Wrap_alloc<std::allocator<v8::CpuProfileDeoptI...	.text	000000018003BF80	0000006C	000
std::string::_Swap_bx(std::string &)	.text	000000018003BFF0	000000E4	000
std::vector<node::inspector::ServerSocket *>::_Rese...	.text	000000018003CF00	00000083	000
asio::detail::service_registry::init_key<asio::detail::de...	.text	000000018003D840	00000022	000
CBarbarianPayOff::Clone(void)	.text	000000018003D870	00000050	000
SQEX::Luminous::Core::IO::Path::GetHostPCPath(cha...	.text	000000018003E6F0	0000005C	000
SQEX::Luminous::Core::IO::Path::GetHostPCPath(cha...	.text	000000018003E730	0000005C	000
std::deque<antlr3::BitsetList<antlr3::TraitsBase<antlr...	.text	000000018003E870	0000003E	000
std::_Func_impl<_lambda_9f5981b46551e382ca72c81...	.text	000000018003E8B0	00000044	000
GEO::expansion_nt::~expansion_nt(void)	**.text**	**000000018003EB70**	**00000028**	**000**
std::deque<v8::internal::Page *>::_Tidy(void)	.text	000000018003EBA0	000000E1	000
std::function<bool (void)>::function<bool (voi...	**.text**	**000000018003F7A0**	**00000086**	**000**
std::string::assign(char const *,char const *)	.text	000000018003F8C0	0000009F	000
deque1_iterator__operator_plus_equals	.text	00000001800402E0	00000042	000
std::vector<std::string>::_Reserve(unsigned __int64)	.text	0000000180040F60	00000083	000
std::_Rotate_unchecked<CString *>(CString *,CStrin...	.text	00000001800415B0	0000007E	000
PdxReverse<CString *>(CString *,CString *)	.text	0000000180041A90	00000076	000
Unity::rapidjson::internal::WriteExponent(int,c...	**.text**	**0000000180043980**	**00000086**	**000**
std::mersenne_twister<uint,32,624,397,31,25...	**.text**	**0000000180047400**	**000000A5**	**000**
std::mersenne_twister<uint,32,624,397,31,25...	**.text**	**00000001800474B0**	**000000D3**	**000**
SQEX::Luminous::SceneDB::LmISceneDB<SQEX::...	**.text**	**00000001800478A0**	**0000003E**	**000**
??_Graw_pwrite_stream@llvm@@UEAAPEAXI...	**.text**	**0000000180047D94**	**00000034**	**000**
PLH::VTableSwapHook::VTableSwapHook(char ...	**.text**	**0000000180047DD0**	**00000021**	**000**
std::basic_stringstream<char,std::char_traits<char>,...	.text	0000000180048560	000000D9	000
std::wostream::`scalar deleting destructor'(uint)	.text	000000018004941C	00000078	000
std::wistream::`scalar deleting destructor'(uint)	.text	000000018004948C	00000078	000
std::streambuf::`vector deleting destructor'(uint)	.text	00000001800494FC	00000094	000
std::istream::_Sentry_base::~_Sentry_base(void)	.text	000000018004982C	0000002D	000
std::streambuf::snextc(void)	.text	000000018004985C	000000B6	000
std::ostream::sentry::~sentry(void)	.text	0000000180049A54	00000050	000
IsWindowsVersionOrGreater(ushort,ushort,us...	**.text**	**000000018004F780**	**000000CE**	**000**
printf	**.text**	**000000018004FFC0**	**00000053**	**000**
??0bad_typeid@std@@QEAA@AEBV01@@Z_0_0	**.text**	**0000000180050878**	**0000003F**	**000**
??0bad_typeid@std@@QEAA@AEBV01@@Z_0_...	**.text**	**00000001800508D8**	**0000003F**	**000**

[Figure 52] Functions populated by Lumina server

As readers can realize, Lumina is able help us with **C++ Template Libraries** a lot and general **C++ functions** , for example.

Before proceeding, we have a very simple modifications in our database through the addition of new library modules to **Signatures tab** (**SHIFT+F5** hot key and **INSERT key**):

- **vs64mfc**
- **msmfc64**

The contribution of these modules is over 200 recognized functions. Fair enough.

The next important clue is that the sample has lots of strings associated to the **Boost C++ library**, as shown below:

```
C:\Users\Administrador\Desktop\MAS\MAS_5>strings -a mas_5_unpacked.bin | findstr -i boost | more
boost::filesystem::path codecvt to wstring
boost::filesystem::path codecvt to string
class boost::basic_string_view<char,struct std::char_traits<char> > __cdecl boost::beast::http::to_strin
g(enum boost::beast::http::verb)
D:\Sources\boost_1_78_0\boost/beast/http/impl/verb.ipp
void __cdecl boost::beast::http::message<1,struct boost::beast::http::basic_string_body<char,struct std:
:char_traits<char>,class std::allocator<char> >,class boost::beast::http::basic_fields<class std::alloca
tor<char> > >::prepare_payload(struct std::integral_constant<bool,1>)
D:\Sources\boost_1_78_0\boost/beast/http/impl/message.hpp
unsigned __int64 __cdecl boost::beast::http::write<class boost::wintls::stream<class boost::beast::basic
_stream<class boost::asio::ip::tcp,class boost::asio::any_io_executor,class boost::beast::unlimited_rate
_policy> >,true,struct boost::beast::http::basic_string_body<char,struct std::char_traits<char>,class st
d::allocator<char> >,class boost::beast::http::basic_fields<class std::allocator<char> >>(class boost::w
intls::stream<class boost::beast::basic_stream<class boost::asio::ip::tcp,class boost::asio::any_io_exec
utor,class boost::beast::unlimited_rate_policy> > &,const class boost::beast::http::message<1,struct boo
st::beast::http::basic_string_body<char,struct std::char_traits<char>,class std::allocator<char> >,class
 boost::beast::http::basic_fields<class std::allocator<char> > > &)
D:\Sources\boost_1_78_0\boost/beast/http/impl/write.hpp
void __cdecl boost::beast::http::header<0,class boost::beast::http::basic_fields<class std::allocator<ch
ar> > >::result(unsigned int)
unsigned __int64 __cdecl boost::beast::http::read<class boost::wintls::stream<class boost::beast::basic_
stream<class boost::asio::ip::tcp,class boost::asio::any_io_executor,class boost::beast::unlimited_rate_
policy> >,class boost::beast::basic_flat_buffer<class std::allocator<char> >,false>(class boost::wintls:
:stream<class boost::beast::basic_stream<class boost::asio::ip::tcp,class boost::asio::any_io_executor,c
lass boost::beast::unlimited_rate_policy> > &,class boost::beast::basic_flat_buffer<class std::allocator
<char> > &,class boost::beast::http::basic_parser<0> &)
D:\Sources\boost_1_78_0\boost/beast/http/impl/read.hpp
```

[Figure 53] Several strings indicating the presence of Boost C++ Library

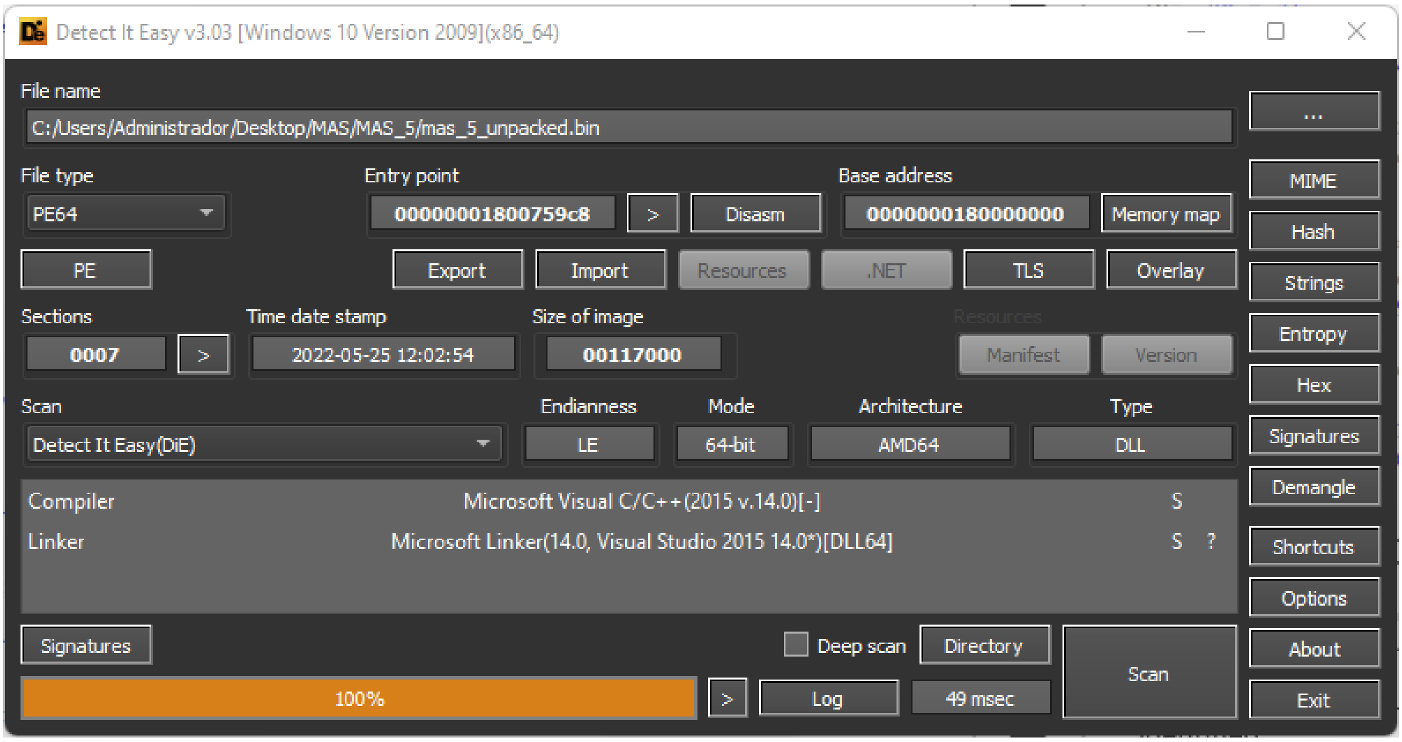
There's important information from the image above such as:

a. The malware is using **boost C++ library**: https://www.boost.org/
b. The author is using **boost version 1.78** that's available here: https://boostorg.jfrog.io/artifactory/main/release/1.78.0/source/
c. There are clues that **beast**, which is a header-only library that serves as foundation **to write network interoperable libraries** supporting **HTTP, WebSocket** and other networking skills, is also present: https://github.com/boostorg/beast

Identify libraries used over the malware program is only the first step to try to tackle reversing problems as, for example, unknown functions. As readers can realize, C++ functions and templates represents a serious issue during the reversing tasks and managing this problem demands having further details in our hands such as:

- **Identifying potential libraries and respective versions (strings command help in this task).**
- **Identifying the compiler and its version used to compile the malware code (DiE might helps us).**
- **Identifying the operating system and version used to compile the malware.**

As I mentioned above, using **DiE** is useful for identifying the compiler and its version, as shown below:



[Figure 54] DiE helps us to identify the compiler and respective version.

As the figure above shows us, it seems the malware author used **Microsoft Visual C++ 2015**, what's awesome information. However, what version of Windows was used? It isn't an easy task and, eventually, we could try to find some evidence on the code using strings command.

I'll be using **Windows 8.1 and 11,** and **Visual Studio 2015** to simulating a possible environment and compile the boost library. At the end, several individual libraries will be generated and, luckily, we could help IDA Pro to recognize few functions. Of course, there's a catch here: the malware author also used the beast header-only library, so we should try something about it, but let's move a step at time.

To compile the **boost C++ library**:

- Download and unpack the **boost C++ version 1.78**: https://boostorg.jfrog.io/artifactory/main/release/1.78.0/source/

- Download the **Visual Studio 2015** and install it and its **respective SDK**:
  - **(web installer)** https://go.microsoft.com/fwlink/?LinkId=532606&clcid=0x409
  - **(iso image)** https://go.microsoft.com/fwlink/?LinkId=615448&clcid=0x409

- Compile the boost library by going into its unpacked directory and executing the following:
  - **bootstrap.bat**
  - **.\b2 --toolset=msvc-14.0**

```
C:\Users\Administrador\Desktop\MAS\MAS_5\Research\boost_1_78_0\boost_1_78_0>.\b2 link=static
--toolset=msvc-14.0
Performing configuration checks

    - default address-model    : 64-bit [1]
    - default architecture     : x86 [1]

Building the Boost C++ Libraries.


    - compiler supports SSE2   : yes [2]
    - compiler supports SSE4.1 : yes [2]
    - has synchronization.lib  : yes [2]
    - has std::atomic_ref      : no [2]
    - has statx                : no [2]
    - has statx syscall        : no [2]
    - has BCrypt API           : yes [2]
    - has init_priority attribute : no [2]
    - has stat::st_blksize     : no [2]
    - has stat::st_mtim        : no [2]
    - has stat::st_mtimensec   : no [2]
    - has stat::st_mtimespec   : no [2]
    - has stat::st_birthtim    : no [2]
    - has stat::st_birthtimensec : no [2]
    - has stat::st_birthtimespec : no [2]
```

[Figure 55] Generating Boost Libraries

The resulting libraries are saved into **stage/lib folder,** and we're interested in the x64 version:

```
C:\Users\Administrador\Desktop\MAS\MAS_5\Research\boost_1_78_0\boost_1_78_0\stage\lib>ls *x64*
libboost_atomic-vc140-mt-gd-x64-1_78.lib
libboost_atomic-vc140-mt-x64-1_78.lib
libboost_chrono-vc140-mt-gd-x64-1_78.lib
libboost_chrono-vc140-mt-x64-1_78.lib
libboost_container-vc140-mt-gd-x64-1_78.lib
libboost_container-vc140-mt-x64-1_78.lib
libboost_context-vc140-mt-gd-x64-1_78.lib
libboost_context-vc140-mt-x64-1_78.lib
libboost_contract-vc140-mt-gd-x64-1_78.lib
libboost_contract-vc140-mt-x64-1_78.lib
libboost_coroutine-vc140-mt-gd-x64-1_78.lib
libboost_coroutine-vc140-mt-x64-1_78.lib
libboost_date_time-vc140-mt-gd-x64-1_78.lib
libboost_date_time-vc140-mt-x64-1_78.lib
libboost_exception-vc140-mt-gd-x64-1_78.lib
libboost_exception-vc140-mt-x64-1_78.lib
libboost_filesystem-vc140-mt-gd-x64-1_78.lib
libboost_filesystem-vc140-mt-x64-1_78.lib
libboost_graph-vc140-mt-gd-x64-1_78.lib
libboost_graph-vc140-mt-x64-1_78.lib
libboost_iostreams-vc140-mt-gd-x64-1_78.lib
```

[Figure 56] Few x64 Boost libraries

Now we have all x64 libraries, readers can try to generate a signature file from each one of these libraries using commands as **pcf.exe** and **sigmake.exe**, which comes from an IDA Pro version 8.0 package named "flair80.zip" and needs to be downloaded from Hex-Rays website.

The process to generate a signature file has the following steps:

- **pcf libboost_filesystem-vc140-mt-x64-1_78.lib**
- **sigmake libboost_filesystem-vc140-mt-x64-1_78.pat libboost_filesystem-vc140-mt-x64-1_78.sig**

Likely, an exclusion file would be generated and it's your decision to decide which signatures will be kept. The exclusion file has the following appearance:

```
;--------- (delete these lines to allow sigmake to read this file)
; add '+' at the start of a line to select a module
; add '-' if you are not sure about the selection
; do nothing if you want to exclude all modules

??_G_Generic_error_category@std@@UEAAPEAXI@Z        00 0000
40534883EC20488D05........488BD9488901F6C201740ABA10000000E8....
??_G_System_error_category@std@@UEAAPEAXI@Z         00 0000
40534883EC20488D05........488BD9488901F6C201740ABA10000000E8....
??_Gerror_category@std@@UEAAPEAXI@Z                 00 0000
40534883EC20488D05........488BD9488901F6C201740ABA10000000E8....

?replace@?$basic_string@_WU?$char_traits@_W@std@@V?$allocator@_W@2@@st
d@@QEAAAEAV12@V?$_String_const_iterator@V?$_String_val@U?$_Simple_type
s@_W@std@@@std@@@2@0PEA_W1@Z     0A 9535
40534883EC30488B4424604C2BC249D1F8488BD94C3BC87535488379180872l7
?replace@?$basic_string@_WU?$char_traits@_W@std@@V?$allocator@_W@2@@st
d@@QEAAAEAV12@V?$_String_const_iterator@V?$_String_val@U?$_Simple_type
s@_W@std@@@std@@@2@0PEB_W1@Z     0A 9535
40534883EC30488B4424604C2BC249D1F8488BD94C3BC87535488379180872l7
```

[Figure 57] List of some x64 Boost libraries

In this specific case, there're only 20 collisions and that's our call to decide what signatures must or not be kept. There're some rules here (I remember that I learned them from **Hex-Rays' website** and "**The IDA Pro Book**" from **Chris Eagle**) :

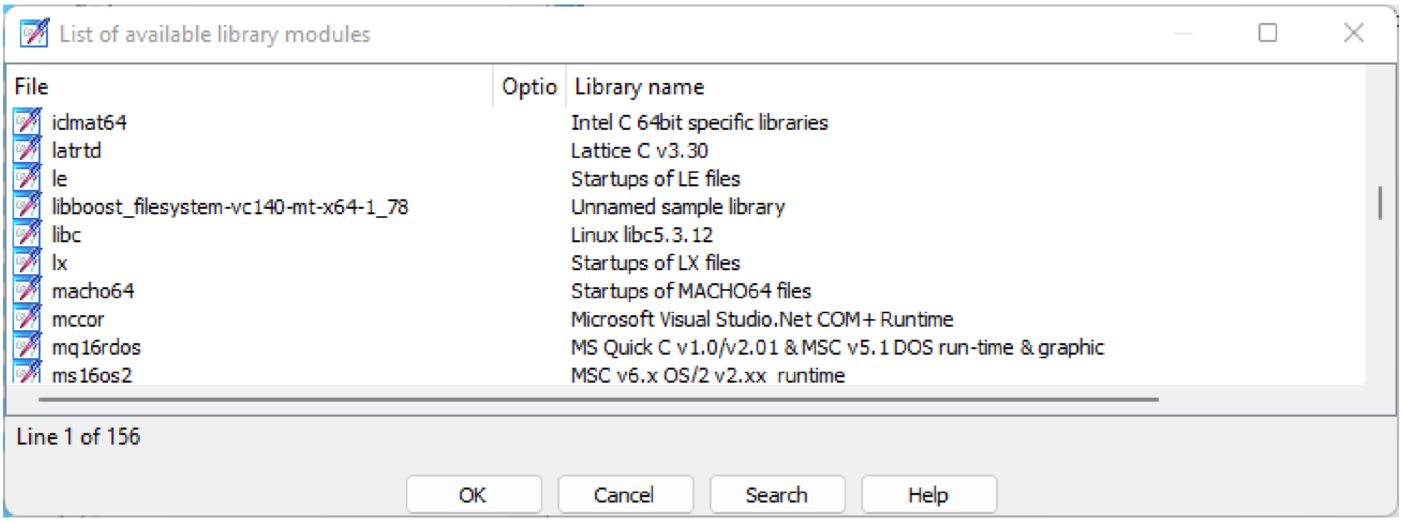
- We must remove the first four lines to proceed. However, if we only remove these four lines and save the file, so all respective signatures will be excluded from the final signature file (.sig file).
- Functions are organized by groups, so we must decide for one function from each group by prefixing the chosen one with a '+' (plus) signal or minus '-' (minus) to show a commentary in the database always that a match happens.
- Never insert a plus ('+') or minus ('-') to more than one function of a specific group.
- If the group has only one function, so don't do nothing.

To generate the signature file, repeat the command:

- **sigmake libboost_filesystem-vc140-mt-x64-1_78.pat libboost_filesystem-vc140-mt-x64-1_78.sig**

Once the signature is generated, copy the file (**libboost_filesystem-vc140-mt-x64-1_78.sig**) to **<IDA Pro installation directory>/sig/pc folder** (for example: **C:\Program Files\IDA Pro 8.0\sig\pc**). Once you've done it, you can open the IDA Pro, go to **Signatures view** (**SHIFT+F5 hotkey**) and insert the new library module (**INS key**).

For example, in this case, 175 functions were identified including only this library module:



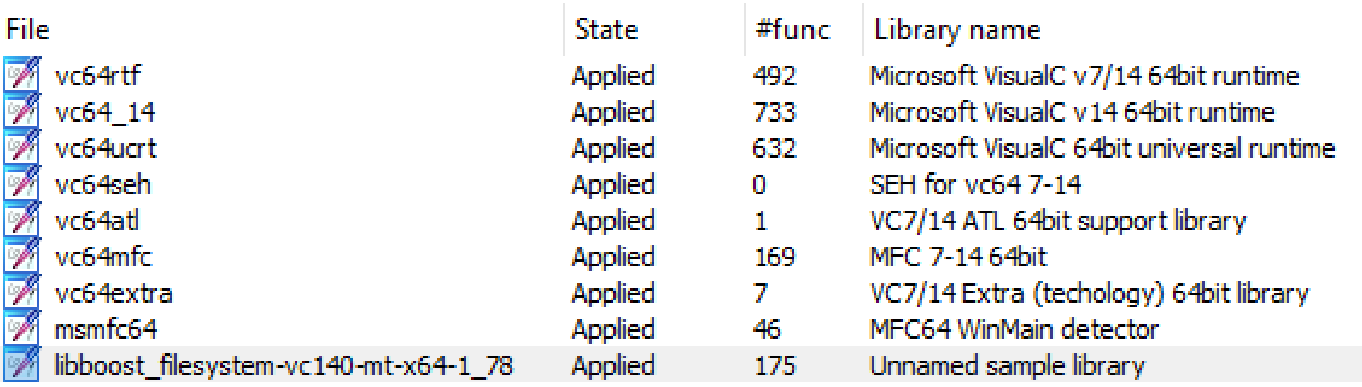
List of available library modules

File	Optio	Library name
idmat64		Intel C 64bit specific libraries
latrtd		Lattice C v3.30
le		Startups of LE files
libboost_filesystem-vc140-mt-x64-1_78		Unnamed sample library
libc		Linux libc5.3.12
lx		Startups of LX files
macho64		Startups of MACHO64 files
mccor		Microsoft Visual Studio.Net COM+ Runtime
mq16rdos		MS Quick C v1.0/v2.01 & MSC v5.1 DOS run-time & graphic
ms16os2		MSC v6.x OS/2 v2.xx runtime

Line 1 of 156

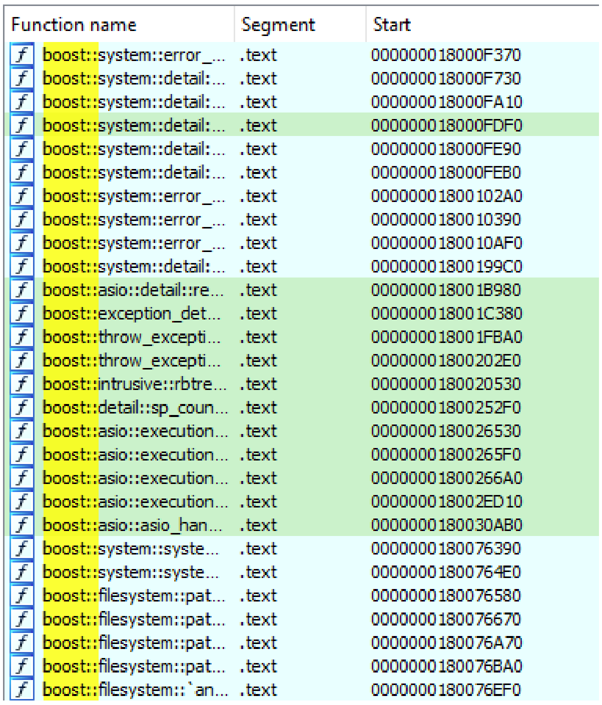
OK Cancel Search Help

[Figure 58] Listing library modules and, in specific, our library.



File	State	#func	Library name
vc64rtf	Applied	492	Microsoft VisualC v7/14 64bit runtime
vc64_14	Applied	733	Microsoft VisualC v14 64bit runtime
vc64ucrt	Applied	632	Microsoft VisualC 64bit universal runtime
vc64seh	Applied	0	SEH for vc64 7-14
vc64atl	Applied	1	VC7/14 ATL 64bit support library
vc64mfc	Applied	169	MFC 7-14 64bit
vc64extra	Applied	7	VC7/14 Extra (techology) 64bit library
msmfc64	Applied	46	MFC64 WinMain detector
libboost_filesystem-vc140-mt-x64-1_78	Applied	175	Unnamed sample library

[Figure 59] Applied 175 signatures from our library module



Function name	Segment	Start
boost::system::error_...	.text	000000018000F370
boost::system::detail:...	.text	000000018000F730
boost::system::detail:...	.text	000000018000FA10
boost::system::detail:...	.text	000000018000FDF0
boost::system::detail:...	.text	000000018000FE90
boost::system::detail:...	.text	000000018000FEB0
boost::system::error_...	.text	00000001800102A0
boost::system::error_...	.text	0000000180010390
boost::system::error_...	.text	00000001800010AF0
boost::system::detail:...	.text	00000001800199C0
boost::asio::detail::re...	.text	000000018001B980
boost::exception_det...	.text	000000018001C380
boost::throw_excepti...	.text	000000018001FBA0
boost::throw_excepti...	.text	00000001800202E0
boost::intrusive::rbtre...	.text	0000000180020530
boost::detail::sp_coun...	.text	00000001800252F0
boost::asio::execution...	.text	0000000180026530
boost::asio::execution...	.text	00000001800265F0
boost::asio::execution...	.text	00000001800266A0
boost::asio::execution...	.text	000000018002ED10
boost::asio::asio_han...	.text	0000000180030AB0
boost::system::syste...	.text	0000000180076390
boost::system::syste...	.text	00000001800764E0
boost::filesystem::pat...	.text	0000000180076580
boost::filesystem::pat...	.text	0000000180076670
boost::filesystem::pat...	.text	0000000180076A70
boost::filesystem::pat...	.text	0000000180076BA0
boost::filesystem::`an...	.text	0000000180076EF0

[Figure 60] Some lines of the list of recognized boost functions (this image also contains boost functions populated by Lumina in light green)

Of course, we should repeat the same process for each static library that we believed having been used in the code. As readers can notice, it's a time-consuming task.

There're many other procedures that, eventually, might help you, but I suggest you try them using a separated database. The success rate varies, but such procedures already helped in some cases.

Another scenario is that **most of external libraries introduces many structure types that we don't know anything about them** and, worse, we don't have enough time to learn about them.

One of possible alternatives (actually, it's a hack) to manage the **lack of applied external structure data types** given by this sort of library follows below (there're many ways to do the same steps here):

a. Write a program using the same library that you want to extract the structures' definitions**. Include headers from this library to force them to be included in the final executable**. It could seem hard, but it isn't because all these external libraries provide tutorial pages including many examples, so you don't need to learn everything about the library itself. **Indeed, the program might be very simple or even blank with a single main function since you include all headers you want to extract definitions.**

b. Configure the environment to use **same compiler version that binary has been compiled**. For example, in this case, I'm using **Visual Studio 2015** because we've identified it through the **DiE (Detect It Easy)**, and the same architecture (for example x64).

c. Don't forget to **include the headers to compile the program**. In Visual Studio 2015: **Properties | VC++ Directories | Include Directories.** For example: *C:\Users\Administrador\Desktop\MAS\MAS_5\boost_1_78_0\boost_1_78_0*.

d. Don't forget to **include the compiled libraries to link the program**. In Visual Studio 2015: **Properties | VC++ Directories | Library Directories.** For example: *C:\Users\Administrador\Desktop\MAS\MAS_5\boost_1_78_0\boost_1_78_0\stage\lib*

e. **Disable any optimization**. On Visual Studio 2015: **Project Properties | C/C++ | Optimization | Optimization: Disabled**. At the same window, set **"Whole Optimization"** to **No**.

f. Don't forget to setup to generate a **Debug version** (*and not Release version*). We need this setting because the Visual Studio is going to generate **a static library (or executable) and a respective pdb file automatically**.

g. This step is not valid for this VS 2015, but it's recommended to **Visual Studio 2022**: **Project Properties | Configuration Properties | C/C++ | General** and change **"Scan Sources for Modules Dependencies"** to **"Yes"** and **"Translate Includes to Imports"** to **"Yes (/translateInclude)"**. This setting will **force the compiler to scan the code for dependencies to be included into header units**.

h. Visual Studio allows you produce an **executable (.exe)**, **a DLL and a static library (.lib**). My best results were using **static library** because it forces everything to be included into a single file. You can configure it going to **Project Properties | Configuration Type | Static Library (.lib).**

i. Navigate to **Project Properties | Configuration Properties | C/C++ | Precompiled Headers | Precompiled Header** and alter the setting to *"Not Using Precompiled Headers".*

j. Compile the program. On Visual Studio: **Build | Build Solution (CTRL+SHIFT+B).** In the **Output window** it's going to be shown the folder where the **static library file** and its respective **.pdb file** are saved.

k. After the program having been compiled, **open the resulting .lib file onto the IDA Pro**.

l. Now we want to **generate an IDC file containing all structure type-definitions**. **Go to File | Produce file | Dump typeinfo to IDC File**. The recommendation is to use an output name following the syntax: **<program name>.idc**. Avoid using something like *<program name>.exe.idc* on Windows.

m. Now comes the final part. Using the IDA Pro, **apply this script into the malware's idb you're analyzing**. Go to **File | Script File** and execute the generated IDC script.

n. Many structure's types will be created, and you can check for them on the **Structure Tab (SHIFT+F9).**

o. Go to the the **Decompiler tab** (**pseudo code tab**) and press **F5**. Likely **several structure type definitions**, created from the **external library (headers),** will be applied automatically.

Note: it is impossible to claim this trick (a hack) will work for you, but it helped previously. Of course, this procedure might mess up your **idb file** (again: do a copy first), but trying doesn't cost anything for you and, if you're lucky, you can get a reasonable result. Additionally, it presents a good advantage: it's appliable to any library used by malware authors and, in case you already know which libraries were used, so it's worth to try it once. Let's me to provide a simple and concrete example about what I described here:

1. Create a new C++ project (**Console Application**), give it a name, and choose directory to save all files from solution. Usually, I create an apart folder to each project. I could have added more headers, but this example is only a demonstration. Note: the code below is NOT mine and, usually, you won't have enough time to learn about the library, but you should remember: the code could be very short, only including headers because we are concerned with headers and nothing more.

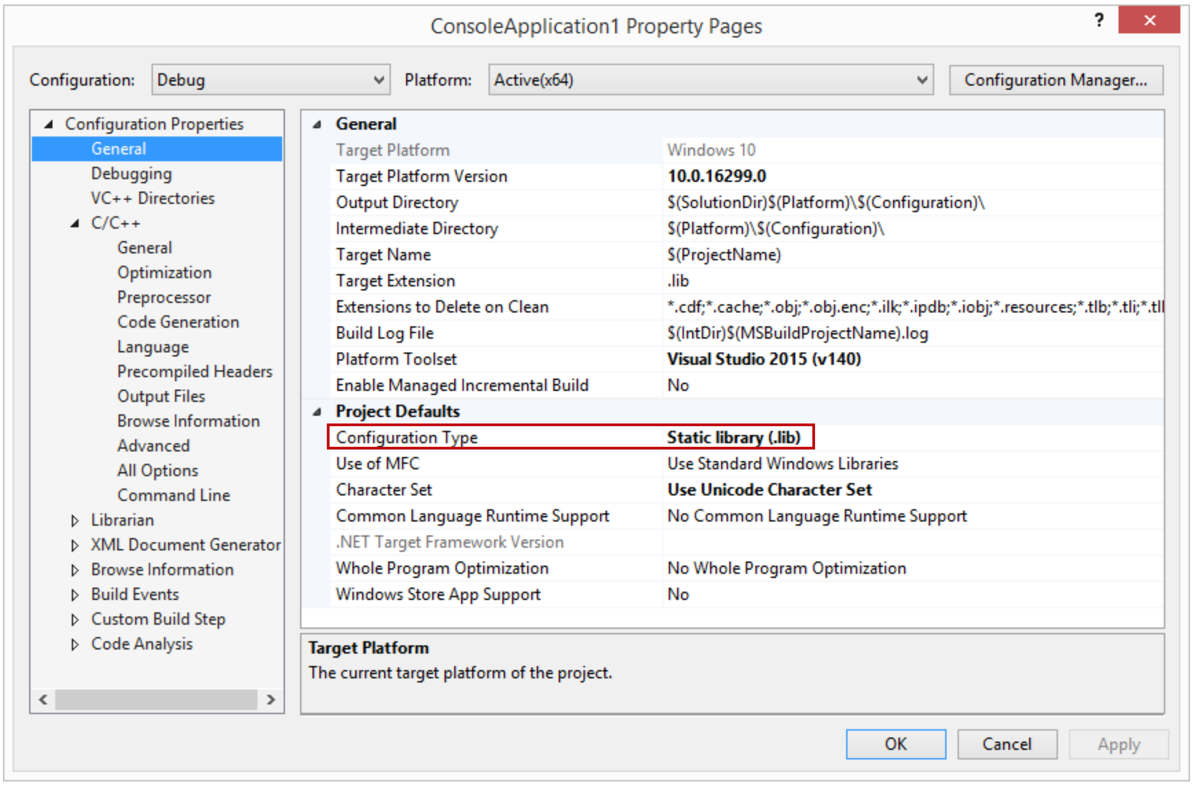
2. Enter the following code from: https://www.boost.org/doc/libs/1_80_0/libs/filesystem/example/tut2.cpp. I added few headers to increase the number of structure types in my final library, but it should have much more headers:

```
#include <iostream>
#include <boost/filesystem.hpp>
#include <boost/asio/connect.hpp>
#include <boost/asio/ip/tcp.hpp>
#include <boost/beast/core.hpp>
#include <boost/beast/websocket.hpp>
#include <boost/container/allocator.hpp>

using namespace std;
using namespace boost::filesystem;
```

```cpp

using namespace std;
using namespace boost::filesystem;

int main(int argc, char* argv[])
{
    if (argc < 2)
    {
        cout << "Usage: tut2 path\n";
        return 1;
    }

    path p(argv[1]);  // avoid repeated path construction below
    if (exists(p))    // does path p actually exist?
    {
        if (is_regular_file(p))        // is path p a regular file?
            cout << p << " size is " << file_size(p) << '\n';
        else if (is_directory(p))      // is path p a directory?
            cout << p << " is a directory\n";
        else
            cout << p << " exists, but is not a regular file or directory\n";
    }
    else
        cout << p << " does not exist\n";

    return 0;
}
```

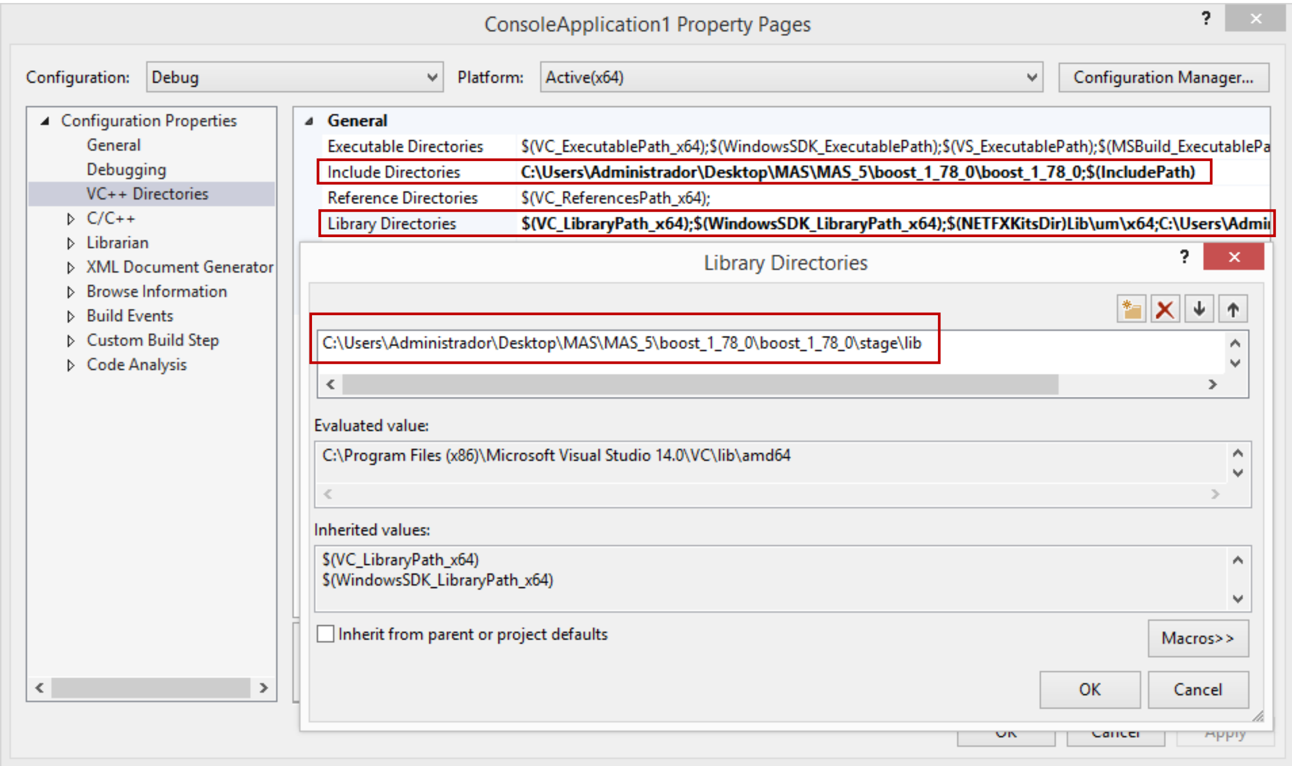
[Figure 61] Test code for type library extraction

3. Change the configuration type to "**Static Library (lib)**":



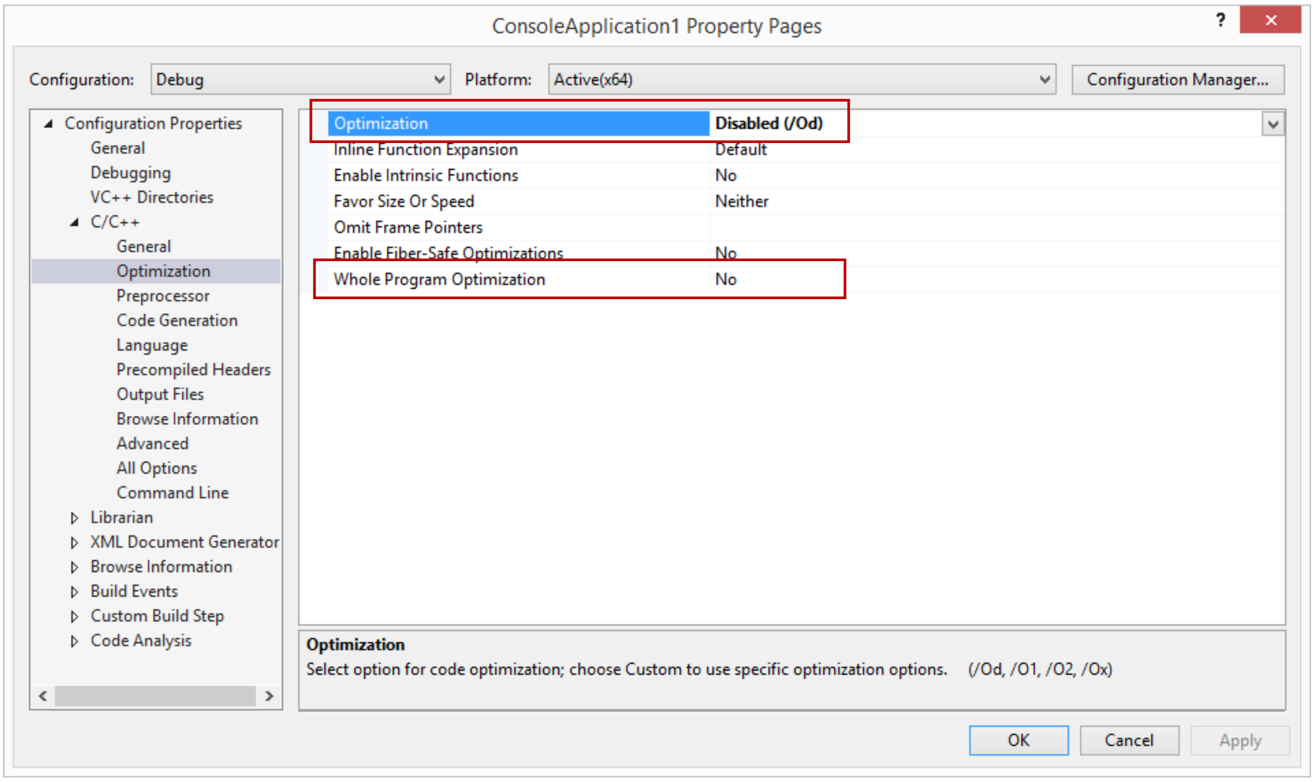
[Figure 62] VS 2015: Configuration type

4. Change "**Include Directories"** and "**Reference Directories**" settings as explained previous and as shown below:



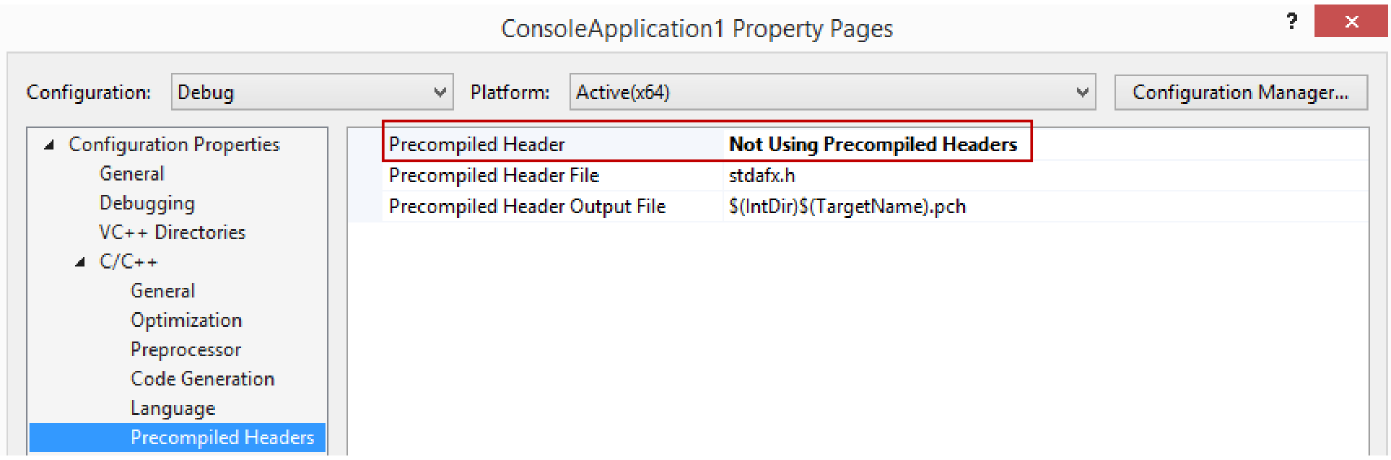
[Figure 63] VS 2015: VC++ Directories settings: Includes and Libraries Directories

5. Turn off any kind of **optimization**:



[Figure 64] VS 2015: Turning off optimizations

6. Disable **Precompiled headers:**

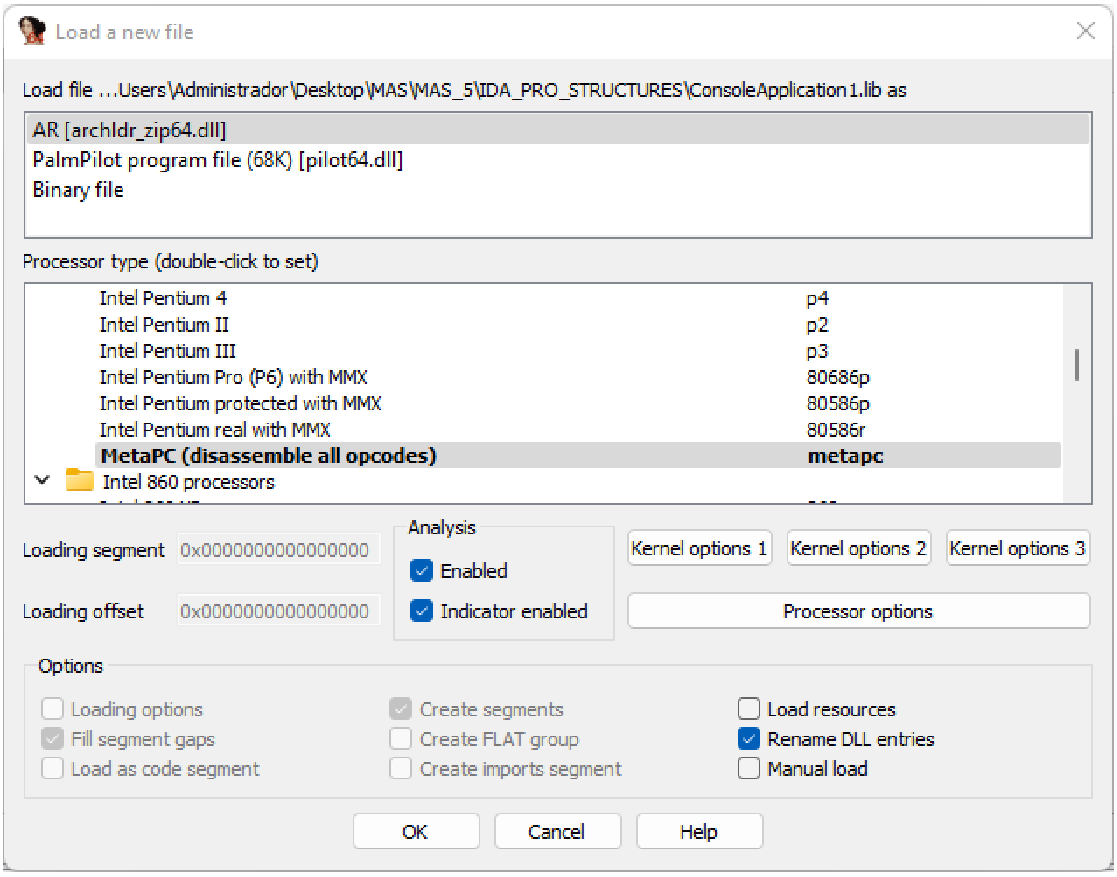


[Figure 65] VS 2015: Turn off precompiled headers

7. Compile the project: **Build | Build Solution (CTRL+SHIFT+B).** Two main files will be generated:

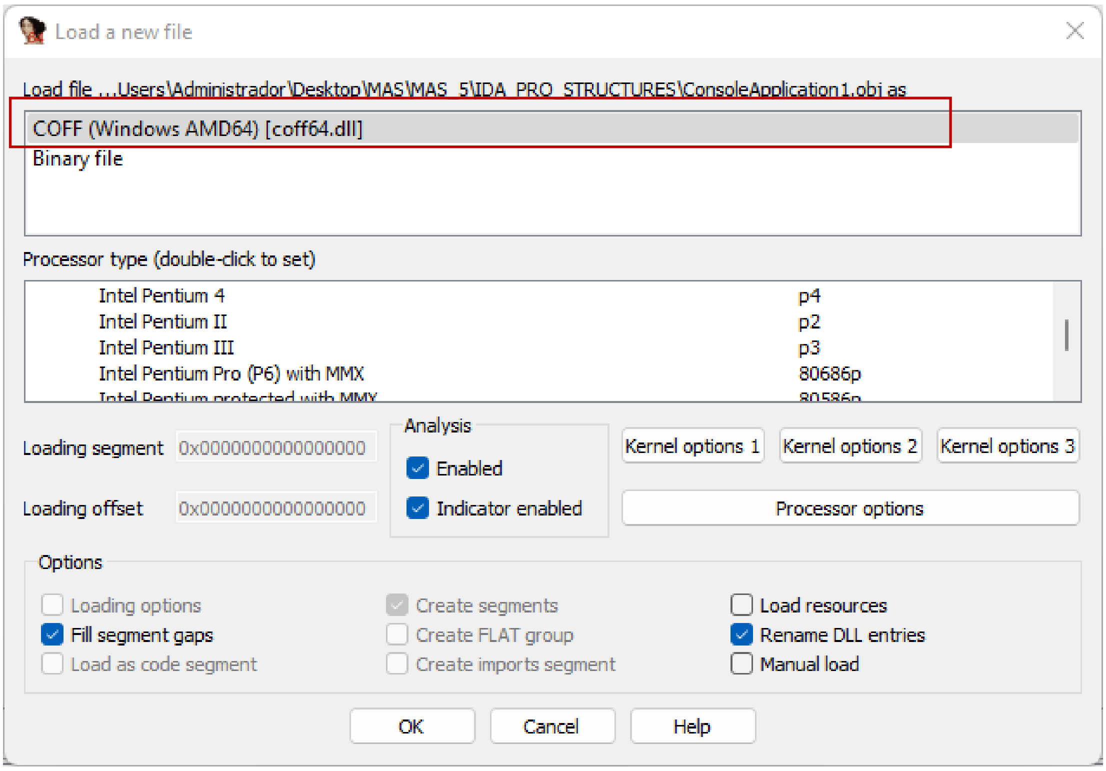
   a. **<program name>.lib**
   b. **<program name>.pdb**

8. Open up the **<programname>. lib** on IDA Pro and choose <**application name>.obj**:



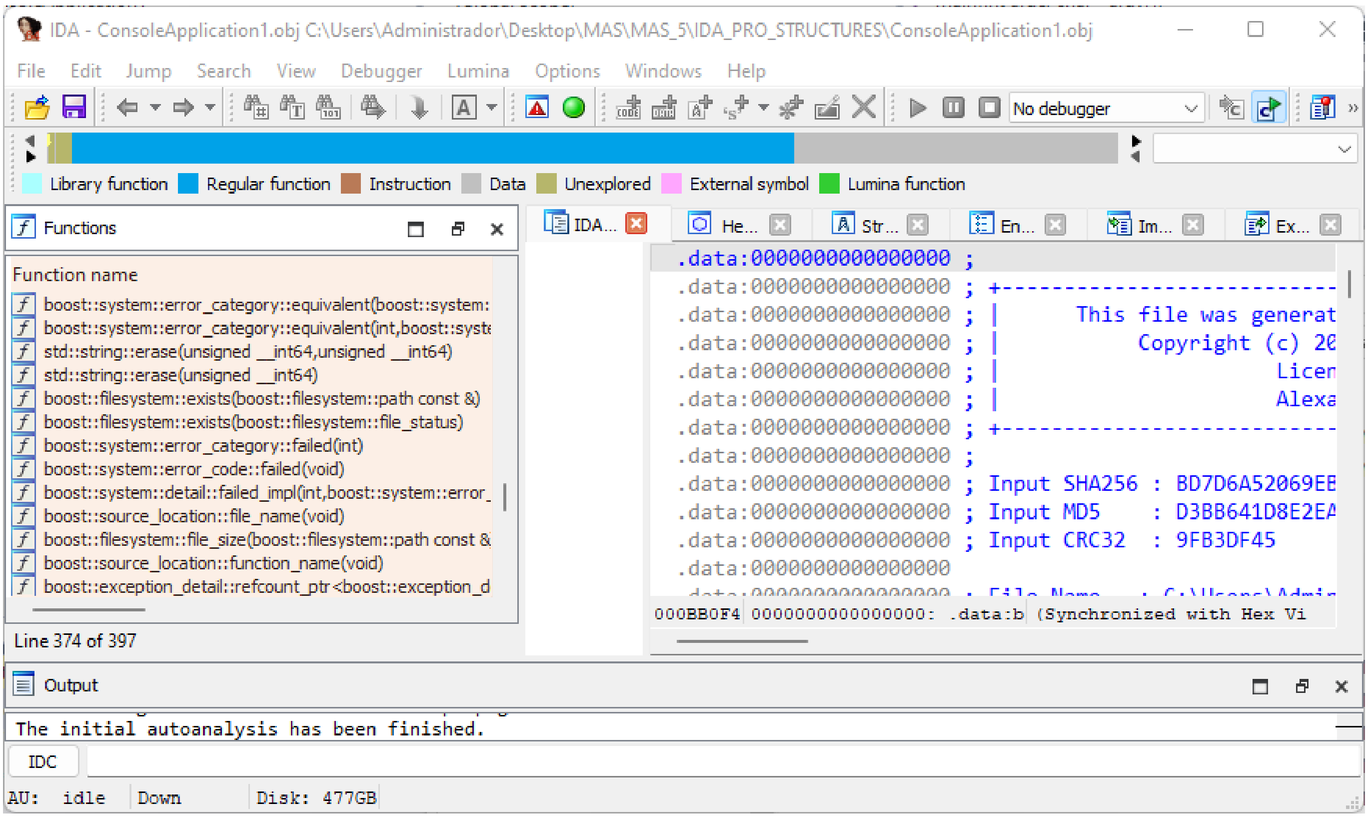
[Figure 66] IDA Pro: opening the .lib file

9. Choose **COFF (Windows AMD64)** and, after this screen, choose **loading the debugging symbols**:



[Figure 67] IDA Pro: opening the .lib file

10. Confirm that there're many **functions coming from Boost C++ Library**:



[Figure 68] IDA Pro: listing many Boost C++ functions

11. Go to **File | Produce file | Dump typeinfo to IDC File**.
12. Open the **target idb database** on IDA Pro, go to **File | Script File** and **load the saved IDC**.
13. Go to the **Structure types (SHIFT+F9)** and verify that several **new types** have been created.
14. Go the pseudo code and **press F5**. All new type definitions will be applied to the code.

Check the instructions from line 146 to line 195 before the script being executed:

```
146   v125 = 15i64;
147   v124 = 0i64;
148   LOBYTE(v123[0]) = 0;
149   if ( *(_QWORD *)v121 )
150   {
151     v15 = v120;
152     if ( *(_QWORD *)v122 >= 0x10ui64 )
153       v15 = (__int64 *)v120[0];
154     sub_1800012D0(v198, (__int64)v15, v121[0]);
155     sub_1800014E0((unsigned __int8 *)v198, (__int64)asc_1800FF830, 80);
156     v16 = v120;
157     if ( *(_QWORD *)v122 >= 0x10ui64 )
158       v16 = (__int64 *)v120[0];
159     sub_1800012D0(v228, (__int64)v16, v121[0]);
160     sub_1800014E0(v228, (__int64)byte_1800FF790, 80);
161     v17 = v120;
162     if ( *(_QWORD *)v122 >= 0x10ui64 )
163       v17 = (__int64 *)v120[0];
164     sub_1800012D0(v229, (__int64)v17, v121[0]);
165     sub_1800014E0(v229, (__int64)byte_1800FF7E0, 80);
166     v18 = v120;
167     if ( *(_QWORD *)v122 >= 0x10ui64 )
168       v18 = (__int64 *)v120[0];
169     sub_1800012D0(v227, (__int64)v18, v121[0]);
170     sub_1800014E0(v227, (__int64)byte_1800FF340, 1024);
171     sub_1800014D0((__int64)v227);
172     sub_1800014D0((__int64)v229);
173     sub_1800014D0((__int64)v228);
174     sub_1800014D0((__int64)v198);
175   }
176   v169[3] = (void *)15;
177   v169[2] = 0i64;
178   LOBYTE(v169[0]) = 0;
179   CoInitializeEx(0i64, 0);
180   v19 = 3;
181   CoInitializeSecurity(0i64, -1, 0i64, 0i64, 0, 3u, 0i64, 0, 0i64);
182   sub_18004A4EC();
183   if ( !(unsigned __int8)sub_180001000() )
184   {
185     CoUninitialize();
186     TerminateThread(qword_1801048D8, 0);
187     goto LABEL_382;
188   }
189   sub_18004BED0(Parameter);
190   v20 = sub_18004BBD0(v134);
191   sub_1800098F0(v188, v20, Parameter);
192   if ( v136 >= 0x10 )
193   {
194     v21 = *(void **)v134;
195     if ( v136 + 1 >= 0x1000 )
```

[Figure 69] IDA Pro: pseudo code before executing the IDC file

Now readers should check the same instructions from **line 146 to line 195** after the script has being executed (it's pretty different, isn't?!):

```
146   v122 = 15i64;
147   v121 = 0i64;
148   LOBYTE(v120.baseclass_0._Mypair._Myval2._Myres) = 0;
149   if ( *(_QWORD *)&v120.baseclass_0._Mypair._Myval2._Bx._Alias[8] )
150   {
151     v15 = (std::_Container_proxy *)&v120;
152     if ( v120.baseclass_0._Mypair._Myval2._Mysize >= 0x10 )
153       v15 = v120.baseclass_0._Mypair._Myval2.baseclass_0._Myproxy;
154     sub_1800012D0(&v193, (__int64)v15, *(int *)&v120.baseclass_0._Mypair._Myval2._Bx._Alias[8]);
155     sub_1800014E0((unsigned __int8 *)&v193, (__int64)asc_1800FF830, 80);
156     v16 = (std::_Container_proxy *)&v120;
157     if ( v120.baseclass_0._Mypair._Myval2._Mysize >= 0x10 )
158       v16 = v120.baseclass_0._Mypair._Myval2.baseclass_0._Myproxy;
159     sub_1800012D0(v223, (__int64)v16, *(int *)&v120.baseclass_0._Mypair._Myval2._Bx._Alias[8]);
160     sub_1800014E0(v223, (__int64)byte_1800FF790, 80);
161     v17 = (std::_Container_proxy *)&v120;
162     if ( v120.baseclass_0._Mypair._Myval2._Mysize >= 0x10 )
163       v17 = v120.baseclass_0._Mypair._Myval2.baseclass_0._Myproxy;
164     sub_1800012D0(v224, (__int64)v17, *(int *)&v120.baseclass_0._Mypair._Myval2._Bx._Alias[8]);
165     sub_1800014E0(v224, (__int64)byte_1800FF7E0, 80);
166     v18 = (std::_Container_proxy *)&v120;
167     if ( v120.baseclass_0._Mypair._Myval2._Mysize >= 0x10 )
168       v18 = v120.baseclass_0._Mypair._Myval2.baseclass_0._Myproxy;
169     sub_1800012D0(v222, (__int64)v18, *(int *)&v120.baseclass_0._Mypair._Myval2._Bx._Alias[8]);
170     sub_1800014E0(v222, (__int64)byte_1800FF340, 1024);
171     sub_1800014D0((__int64)v222);
172     sub_1800014D0((__int64)v224);
173     sub_1800014D0((__int64)v223);
174     sub_1800014D0((__int64)&v193);
175   }
176   v165 = 15i64;
177   v164 = 0i64;
178   LOBYTE(v163.baseclass_0._Mypair._Myval2._Myres) = 0;
179   CoInitializeEx(0i64, 0);
180   v19 = 3;
181   CoInitializeSecurity(0i64, -1, 0i64, 0i64, 0, 3u, 0i64, 0, 0i64);
182   sub_18004A4EC();
183   if ( !(unsigned __int8)sub_180001000() )
184   {
185     CoUninitialize();
186     TerminateThread(qword_1801048D8, 0);
187     goto LABEL_382;
188   }
189   sub_18004BED0(Parameter);
190   v20 = sub_18004BBD0(v131);
191   sub_1800098F0(v183, v20, Parameter);
192   if ( v133 >= 0x10 )
193   {
194     v21 = *(void **)v131;
195     if ( v133 + 1 >= 0x1000 )
```

[Figure 70] IDA Pro: pseudo code AFTER executing the IDC file

If you need to know any new type definition, double click on it:

```
00000000 std::string     struc ; (sizeof=0x28, align=0x8, mappedto_313)
00000000 baseclass_0     std::_String_alloc<std::_String_base_types<char> > ?
00000028 std::string     ends
00000028
00000000 ; ---------------------------------------------------------------------------
00000000
00000000 std::_String_alloc<std::_String_base_types<char> > struc ; (sizeof=0x28, align=0x8, mappedto_312)
00000000                                         ; XREF: std::string/r
00000000 _Mypair         std::_Compressed_pair<std::_Wrap_alloc<std::allocator<char> >,std::_String_val<std::_Simple_types<char> >,1> ?
00000028 std::_String_alloc<std::_String_base_types<char> > ends
00000028
00000000 ; ---------------------------------------------------------------------------
00000000
00000000 std::_Compressed_pair<std::_Wrap_alloc<std::allocator<char> >,std::_String_val<std::_Simple_types<char> >,1> struc ; (sizeof=0
00000000                                         ; XREF: std::_String_alloc<std::_String_base_types<char> >/r
00000000 _Myval2         std::_String_val<std::_Simple_types<char> > ?
00000028 std::_Compressed_pair<std::_Wrap_alloc<std::allocator<char> >,std::_String_val<std::_Simple_types<char> >,1> ends
00000028
00000000 ; ---------------------------------------------------------------------------
00000000
00000000 std::_String_val<std::_Simple_types<char> > struc ; (sizeof=0x28, align=0x8, mappedto_310)
00000000                                         ; XREF: std::_Compressed_pair<std::_Wrap_alloc<std::allocator<char> >,std::_String_val
00000000 baseclass_0     std::_Container_base12 ?
00000008 _Bx             std::_String_val<std::_Simple_types<char> >::_Bxty ?
00000018 _Mysize         dq ?
00000020 _Myres          dq ?
00000028 std::_String_val<std::_Simple_types<char> > ends
00000028
00000000 ; ---------------------------------------------------------------------------
00000000
00000000 std::_Container_base12 struc ; (sizeof=0x8, align=0x8, mappedto_244)
00000000                                         ; XREF: std::_String_val<std::_Simple_types<char> >/r
00000000                                         ; std::_String_val<std::_Simple_types<wchar_t> >/r
```

[Figure 71] New type definitions created by IDC

There's a very important point here: my result might be **completely sub-optimal**. Why? Because I only added few header files and Boost C++ Library has many header files. Therefore, if the readers have spare time, so I invite you to **add other header files** (**strings command can help you and provide a guideline of headers to be included),** and maybe your results will be a bit better than mine, which was shown on **Figure 70**. Virtually, many of these additional headers wouldn't be necessary because Visual Studio brings them into the compiled the program because dependencies, but it's valid to make a test.

Anyway, my central idea here is explaining the technique and, afterwards, according to your case, you can evaluate whether it's worths or not.

If you're looking for possible and available Boost C++ headers (version 1.78, according to our malware sample), one starting point could be this one: https://www.boost.org/doc/libs/1_78_0/libs/libraries.htm .

If readers want to do a test including new headers, so the own Visual Studio can help you to add them because while you're typing includes it opens a list of header possible options, so it's much easy to add new ones. A suggestion of additional headers follows bellow just in case you want to try it:

- **#include <boost/beast.hpp>**
- **#include <boost/beast/http.hpp>**
- **#include <boost/beast/core/basic_stream.hpp>**
- **#include <boost/beast/core/static_string.hpp>**
- **#include <boost/beast/core/basic_stream.hpp>**
- **#include <boost/beast/core/buffer_traits.hpp>**
- **#include <boost/beast/core/error.hpp>**
- **#include <boost/beast/core/file.hpp>**
- **#include <boost/beast/core/buffers_range.hpp>**
- **#include <boost/beast/core/buffers_range.hpp>**

- **#include <boost/beast/core/stream_traits.hpp>**
- **#include <boost/beast/core/ostream.hpp>**
- **#include <boost/beast/core/buffers_to_string.hpp>**
- **#include <boost/beast/core/file_win32.hpp>**
- **#include <boost/beast/http/fields.hpp>**
- **#include <boost/beast/http/type_traits.hpp>**
- **#include <boost/beast/http/type_traits.hpp>**
- **#include <boost/beast/http/message.hpp>**
- **#include <boost/beast/http/status.hpp>**
- **#include <boost/beast/http/basic_parser.hpp>**
- **#include <boost/beast/http/impl/basic_parser.hpp>**
- **#include <boost/beast/http/impl/error.hpp>**
- **#include <boost/static_string.hpp>**

There're many other hacks that, eventually, could be helpful. One of these approaches to **add external type definitions to another idb database** (in our case, the malware idb database) that I learned through a comment from **Igor Skochinsky (Hex-Rays's developer -- @IgorSkochinsky**) on **Stack Exchange | Reverse Engineering** website is the following one:

a. Using IDA Pro, open the **ConsoleApplication1.idb**, which is the same application that we've created using Visual Studio and included all headers from **Boost C++ Library**.

b. Keeping the **.idb file open**, **copy <program name>.til** to another folder (I've copied it into the same folder of IDA Pro because there I have the help's support).

c. Using **tilib64.exe** (available from https://hex-rays.com/download-center/), execute**: tilib64 -#- ConsoleApplication1.til**

d. Copy the resulting **.til file** to **/til/pc folder** (**C:\Program Files\IDA Pro 8.0\til\pc**).

e. **Open the second .idb** (in this case, the idb database of the malware we're executing) and add this new type library in the list (**SHIFT + F11** and then **INS**).

f. Although the new type definitions won't be shown in the **Local Types tab**, fortunately they will be available through **Structures tab (SHIFT+F9)** by inserting a new structure (**INS**) and choosing **"Add standard structure"**. Additionally, these structures are available for decompiler too.

According to **Igor**, **this procedure is not officially** supported and, as any other hacks, **it could cause many issues and conflicts in the idb file**, so the same recommendation is valid: **make a backup of the .idb file** before testing this approach.

Once again: these last two approaches are experimental and they can work or not. However, when you're analyzing a malware that include libraries and new types, so you could spend a considerable amount of time to understand the existing new types and, eventually (maybe...maybe....), it could be useful for your analysis. According to my experience, I never (ever) disregard any approach, suggestion or trick while reversing code because they always can be useful at some moment, so any new knowledge is always valuable and welcome.

12. Reversing: third part

Let's return to our normal path and continue analyzing the code. In special, we're analying the subroutine **sub_1800A120**. To education purposes, I'll be using the **idb version with Lumina functions applied** (that's the best approach here), but without including results from last two procedures used to apply new definition types from external C++ libraries because I'm not sure whether readers will have time to test them. Anyway, this decision doesn't change our analysis over this section.

We have the following code from **sub_1800A120**:

```
LODWORD(v2) = beginthreadex(0i64, 0, (_beginthreadex_proc_type)w_DetectRunningTools, 0i64, 0, &ThrdAddr);
qword_1801048D8 = v2;
*(_QWORD *)v122 = 15i64;
*(_QWORD *)v121 = 0i64;
LOBYTE(v120[0]) = 0;
if ( aVcffi2rj6t15[0] )
{
  v3 = -1i64;
  do
    ++v3;
  while ( aVcffi2rj6t15[v3] );
}
else
{
  v3 = 0i64;
}
std::string::assign((unsigned __int64 ****)v120, aVcffi2rj6t15, (unsigned __int64 ***)v3);
if ( *(_QWORD *)v121 )
{
  v4 = v120;
  if ( *(_QWORD *)v122 >= 0x10ui64 )
    v4 = (__int64 *)v120[0];
  sub_1800012D0(v227, (__int64)v4, v121[0]);
  sub_180001670(v227, (__int64)asc_1800FF830, 79);
  v5 = v120;
  if ( *(_QWORD *)v122 >= 0x10ui64 )
    v5 = (__int64 *)v120[0];
  sub_1800012D0(v229, (__int64)v5, v121[0]);
  sub_180001670(v229, (__int64)byte_1800FF790, 79);
  v6 = v120;
  if ( *(_QWORD *)v122 >= 0x10ui64 )
    v6 = (__int64 *)v120[0];
  sub_1800012D0(v228, (__int64)v6, v121[0]);
  sub_180001670(v228, (__int64)byte_1800FF7E0, 79);
  v7 = v120;
  if ( *(_QWORD *)v122 >= 0x10ui64 )
    v7 = (__int64 *)v120[0];
  sub_1800012D0(v198, (__int64)v7, v121[0]);
  sub_180001670((unsigned __int8 *)v198, (__int64)byte_1800FF340, 1023);
  sub_1800014D0((__int64)v198);
  sub_1800014D0((__int64)v228);
```

[Figure 72] sub_1800A120 function

There're three interesting details on this figure:

- An explicit usage of a string "**Vcffi2rj6t15**" as argument of a function.
- The repeated appearance (four times) of two subroutines: **sub_1800012D0** and **sub_180001670**.
- There're fours references to different data-related addresses: **asc_1800FF830**, **byte_1800FF790**, **byte_1800FF7E0** and **byte_1800FF340**.

As the string *"Vcffi2rj6t15",* which could be a possible key, is being assigned to **v120 local variable** and, afterwards, this **v120 variable** is being assigned to **v4, v5, v6** and **v7 local variables**. Therefore, I renamed:

- **v120 to var_key**
- **v4 to var_key_1**
- **v5 to var_key_2**
- **v6 to var_key_3**
- **v7 to var_key_4**

If readers closely pay attention, the same string (**Vcffi2rj6t15**) is being passed to **sub_1800012D0** over the four times it's called:

```
std::string::assign((std::string *)var_key, aVcffi2rj6t15, v3);
if ( *(_QWORD *)v121 )
{
  var_key_1 = var_key;
  if ( *(_QWORD *)v122 >= 0x10ui64 )
    var_key_1 = (__int64 *)var_key[0];
  sub_1800012D0(v225, (__int64)var_key_1, v121[0]);
  sub_180001670(v225, (__int64)asc_1800FF830, 79);
  var_key_2 = var_key;
  if ( *(_QWORD *)v122 >= 0x10ui64 )
    var_key_2 = (__int64 *)var_key[0];
  sub_1800012D0(v227, (__int64)var_key_2, v121[0]);
  sub_180001670(v227, (__int64)byte_1800FF790, 79);
  var_key_3 = var_key;
  if ( *(_QWORD *)v122 >= 0x10ui64 )
    var_key_3 = (__int64 *)var_key[0];
  sub_1800012D0(v226, (__int64)var_key_3, v121[0]);
  sub_180001670(v226, (__int64)byte_1800FF7E0, 79);
  var_key_4 = var_key;
  if ( *(_QWORD *)v122 >= 0x10ui64 )
    var_key_4 = (__int64 *)var_key[0];
  sub_1800012D0(v196, (__int64)var_key_4, v121[0]);
  sub_180001670((unsigned __int8 *)v196, (__int64)byte_1800FF340, 1023);
  sub_1800014D0((__int64)v196);
  sub_1800014D0((__int64)v226);
  sub_1800014D0((__int64)v227);
  sub_1800014D0((__int64)v225);
}
```

[Figure 73] sub_1800A120 function including few renamed variables

Entering into **sub_1800012D0** we have:

```
_BYTE *__fastcall sub_1800012D0(_BYTE *a1, __int64 var_key_1, int a3)
{
  _BYTE *v4; // [rsp+20h] [rbp-38h]

  *a1 = 0;
  a1[1] = 0;
  a1[2] = 0;
  v4 = a1 + 3;
  do
    *v4++ = 0;
  while ( v4 != a1 + 258 );
  sub_180001360(a1, var_key_1, a3);
  return a1;
}
```

[Figure 74] sub_180012D0 routine

Examining the content of **sub_180001360 routine** we have:

```
1 _BYTE *__fastcall sub_180001360(_BYTE *a1, __int64 a2, int a3)
2 {
3   // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]
4 
5   *a1 = 0;
6   a1[1] = 0;
7   for ( i = 0; i < 256; ++i )
8     a1[i + 2] = i;
9   if ( a3 <= 0 )
10  {
11    result = a1;
12    a1[258] = 0;
13  }
14  else
15  {
16    v6 = 0;
17    v5 = 0;
18    v4 = 0;
19    while ( v5 < 256 )
20    {
21      if ( v4 >= a3 )
22        v4 = 0;
23      v6 += a1[v5 + 2] + *(_BYTE *)(a2 + v4);
24      v7 = a1[v5 + 2];
25      a1[v5 + 2] = a1[v6 + 2];
26      a1[v6 + 2] = v7;
27      ++v5;
28      ++v4;
29    }
30    result = a1;
31    a1[258] = 1;
32  }
33  return result;
34 }
```

[Figure 75] sub_18001360 routine

Although it can be a bit difficult to recognize at first view, it's the initialization code of **RC4 algorithm.** The **"modulo 256" operation**, which is present in usual **RC4 initialization**, has been replaced by the arithmetic composed by code between lines 16 to 29.

As readers could remember, all line of code from this subroutine make part of the **KSA (Key-Scheduling Algorithm)**, which is used to initialize the permutation of starting array.

Every time you're analyzing a potential encrypted data, once you determined the algorithm used, usually they have requested inputs (given through local variables), which varies according to the algorithm. For example, while analyzing RC4, we must determine:

a. **Three arguments for KSA phase (RC4):**
    i. context
    ii. key buffer
    iii. key length

b. **Three arguments for Encrypt/Decrypt phase (RC4):**
    i. context
    ii. data buffer
    iii. data length

Of course, this “scheme” might change or having variants, but most of the time it happens this way. Therefore, it seems that on **Figure 75** the **sub_180001360** has the following parameters’ meanings:

- **a1 is the context**
- **a2 is the key**
- **a3 is the key length**

Proceeding with the analysis, let’s examine the **sub_180001670 -> sub1800014E0**:

```
1  __int64 __fastcall sub_1800014E0(unsigned __int8 *a1, __int64 a2, int a3)
2  {
3    int i; // [rsp+8h] [rbp-30h]
4    unsigned __int8 v5; // [rsp+Ch] [rbp-2Ch]
5    unsigned __int8 v6; // [rsp+Dh] [rbp-2Bh]
6    unsigned __int8 v7; // [rsp+Eh] [rbp-2Ah]
7    unsigned __int8 v8; // [rsp+Fh] [rbp-29h]
8
9    if ( (a1[258] & 1) == 0 )
10     return a1[258] & 1;
11   v8 = *a1;
12   v7 = a1[1];
13   for ( i = 0; i < a3; ++i )
14   {
15     v6 = a1[++v8 + 2];
16     v7 += v6;
17     v5 = a1[v7 + 2];
18     a1[v8 + 2] = v5;
19     a1[v7 + 2] = v6;
20     *(_BYTE *)(a2 + i) = (a1[(unsigned __int8)(v5 + v6) + 2] & 0x71 | ~a1[(unsigned __int8)(v5 + v6) + 2] & 0x8E) ^ (*(_BYTE *)(a2 + i) & 0x71 | ~*(_BYTE *)(a2 + i) & 0x8E);
21   }
22   *a1 = v8;
23   a1[1] = v7;
24   return a1[258] & 1;
25 }
```

[Figure 76] sub_1800014E0 routine

Although the figure is a bit small, the **line 20** is the following one:

- ***(_BYTE *)(a2 + i) = (a1[(unsigned __int8)(v5 + v6) + 2] & 0x71 | ~a1[(unsigned __int8)(v5 + v6) + 2] & 0x8E) ^ (*(_BYTE *)(a2 + i) & 0x71 | ~*(_BYTE *)(a2 + i) & 0x8E);**

Once again, it could not be so like the well-known RC4 encrypt algorithm representation, but it’s actually a variation its form.

At the same way, in this decryption subroutine (**sub_1800014E0**) we have:

- **a1 is the context**
- **a2 is the data buffer**
- **a3 is the data length**

Therefore, I renamed the following subroutines to better names:

- **sub_1800012D0 -> ab_w_RC4_INIT**
- **sub_180001360 -> ab_RC4_INIT**
- **sub_1800014E0 -> ab_RC4_DECRYPTION**
- **sub_180001670 -> ab_w_RC4_DECRYPTION**

Thus, we’re already know the likely key and it’s time to follow the encrypted data, which show us the following scenario:

- **asc_1800FF830:** **db '127.0.0.1',0** renamed to: **enc_data_1**
- **byte_1800FF790:** **char byte_1800FF790[80]** renamed to: **enc_data_2**
- **byte_1800FF7E0:** **char byte_1800FF7E0[80]** renamed to: **enc_data_3**
- **byte_1800FF340:** **char byte_1800FF340[1024]** renamed to: **enc_data_4**

After all renaming actions, we have:

```
std::string::assign((std::string *)var_key, aVcffi2rj6t15, v3);
if ( *(_QWORD *)v121 )
{
  var_key_1 = var_key;
  if ( *(_QWORD *)v122 >= 0x10ui64 )
    var_key_1 = (__int64 *)var_key[0];
  ab_w_RC4_INIT(context_1, (__int64)var_key_1, v121[0]);
  ab_w_RC4_DECRYPTION(context_1, (__int64)enc_data_1, 79);
  var_key_2 = var_key;
  if ( *(_QWORD *)v122 >= 0x10ui64 )
    var_key_2 = (__int64 *)var_key[0];
  ab_w_RC4_INIT(context_2, (__int64)var_key_2, v121[0]);
  ab_w_RC4_DECRYPTION(context_2, (__int64)enc_data_2, 79);
  var_key_3 = var_key;
  if ( *(_QWORD *)v122 >= 0x10ui64 )
    var_key_3 = (__int64 *)var_key[0];
  ab_w_RC4_INIT(context_3, (__int64)var_key_3, v121[0]);
  ab_w_RC4_DECRYPTION(context_3, (__int64)enc_data_3, 79);
  var_key_4 = var_key;
  if ( *(_QWORD *)v122 >= 0x10ui64 )
    var_key_4 = (__int64 *)var_key[0];
  ab_w_RC4_INIT(context_4, (__int64)var_key_4, v121[0]);
  ab_w_RC4_DECRYPTION((unsigned __int8 *)context_4, (__int64)enc_data_4, 1023);
  ab_return_context((__int64)context_4);
  ab_return_context((__int64)context_3);
  ab_return_context((__int64)context_2);
  ab_return_context((__int64)context_1);
}
```

[Figure 77] Part of sub_1800A120 subroutine after renaming

There're three encrypted data because the first one is a plain text string. About the remaining three blobs, one of them has a size of 1024 bytes and the other two ones have size of 80 bytes each, and all of them are using the same decryption key.

We must write a script to extract and decrypt the data. As readers will see, **enc_data_4** holds **C2 IP's/domains list**, **enc_data_3** holds the **botnet** and **enc_data_2** apparently holds the **campaign ID**.

The initial information is about our target:

- **key:** VcFFI2Rj6t15
- **section:** .data
- **name:** enc_data_2 **size:** 80 **start:** 0x1800FF790
- **name:** enc_data_3 **size:** 80 **start:** 0x1800FF7E0
- **name:** enc_data_4 **size:** 1024 **start:** 0x1800FF340

The Python script will be written and tested using **Jupyter Notebook** (https://jupyter.org/) because it's much easier to debug the script. The final version follows below:

```python
import binascii
import pefile
import ipaddress
from Crypto.Cipher import ARC4

# This routine extracts and returns data from .data section,
# .data section address and file image base.
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)),\
                section.VirtualAddress, hex(pe.OPTIONAL_HEADER.ImageBase)

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

    data_offset = int(end_addr,16) - int(start_addr,16)
    return data_offset

# This routine decrypts RC4 encrypted data.
def data_decryptor(key_data, data):

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

# encrypted_string_addr: start address of the encrypted strings
# data_size: it represents the size of the encrypted_data
def show_data(encrypted_string_addr, data_size):

    # Next two lines extracts .data section's information.
    filename = r"C:\Users\Administrador\Desktop\MAS\MAS_5\mas_5_unpacked.bin"
    data_encoded_extracted, sect_address, file_image_base = extract_data(filename)

    # Next three lines find the RVA of the .data section, the absolute address
    # of the .data section and the offset of encrypted data respectively.
    data_seg_rva_addr = hex(sect_address)
    data_seg_real_addr = hex(int(data_seg_rva_addr,16) + int(file_image_base,16))
    data_offset = calc_offsets(encrypted_string_addr, data_seg_real_addr)

    # This line extract the encrypted data
    encrypted_data = data_encoded_extracted[data_offset:data_offset + data_size]

    # This line defines the RC4 key
    data_key = b'VcFFI2Rj6t15'

    # Next two lines calls the RC4 data decryptor and returns the result
    decrypted_data = data_decryptor(data_key, encrypted_data)
    return decrypted_data
```

[Figure 78] First part of script to extract and decrypt C2 List and botnet

```python
def main():

    print("\nC2 IPv4 ADDRESS LIST: ")
    print(28*'-')

    enc_data_4 = show_data('0x1800FF340',1024)
    counter = 1
    for k in ((enc_data_4.decode('utf-8')).split('\00')[0].split(',')):
        print("IP[%d]: %s" % (counter, k))
        counter += 1

    counter2 = 1
    enc_data_3 = show_data('1800FF7E0',80)
    for k in ((enc_data_3.decode('utf-8')).split('\00')[0].split(',')):
        print("\nBOTNET[%d]: %s" % (counter2, k))
        counter2 += 1

    counter3 = 1
    enc_data_2 = show_data('0x1800FF790',80)
    for k in ((enc_data_2.decode('utf-8')).split('\00')[0].split(',')):
        print("\nDATA[%d]: %s" % (counter3, k))
        counter3 += 1

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

[Figure 79] Second part of script to extract and decrypt C2 List and botnet

```
C2 IPv4 ADDRESS LIST:
----------------------------
IP[1]: 242.165.212.79:339
IP[2]: 162.144.249.150:239
IP[3]: 63.122.120.151:268
IP[4]: 144.52.138.51:193
IP[5]: 18.215.29.142:436
IP[6]: 115.239.67.202:380
IP[7]: 255.11.235.99:426
IP[8]: 213.203.201.199:307
IP[9]: 143.117.20.123:425
IP[10]: 141.98.168.70:443
IP[11]: 174.150.214.40:426
IP[12]: 133.133.249.24:204
IP[13]: 126.68.7.249:422
IP[14]: 103.175.16.107:443
IP[15]: 146.70.124.77:443
IP[16]: 154.56.0.100:443
IP[17]: 180.184.129.160:223
IP[18]: 28.78.74.145:427
IP[19]: 108.28.254.44:399
IP[20]: 115.103.22.1:153
IP[21]: 149.57.112.159:122
IP[22]: 229.139.73.188:287
IP[23]: 112.110.146.153:349
IP[24]: 249.222.51.70:286
IP[25]: 180.23.251.29:230
IP[26]: 244.234.60.83:386
IP[27]: 79.133.212.60:211
IP[28]: 192.21.12.118:231
IP[29]: 31.215.170.180:431
IP[30]: 140.208.107.161:360
IP[31]: 119.177.224.146:124
IP[32]: 58.10.55.201:382
IP[33]: 57.156.134.113:446
IP[34]: 83.142.26.147:465
IP[35]: 194.135.33.16:443
IP[36]: 35.17.203.69:268
IP[37]: 104.135.8.250:417
IP[38]: 210.251.188.194:228
IP[39]: 53.96.32.99:333
IP[40]: 70.77.209.88:224
IP[41]: 65.254.82.66:498
IP[42]: 65.95.20.151:232
IP[43]: 165.158.204.41:469
IP[44]: 185.62.58.209:443
IP[45]: 102.109.16.255:445
IP[46]: 137.253.55.69:235

BOTNET[1]: 146l

DATA[1]: 444
```

[Figure 80] Extracted and decrypted C2 List

If readers compare the output above against the list offered by **Triage** given by **malwoverview** tool or even through the online report (https://tria.ge/220616-tng2tsfhfl), so you will notice that there're a perfect match for each IP address and botnet.

Actually, many quite interesting aspects are present in this binary and, undoubtedly, we could extend this analysis over many pages. Anyway, there's nothing special and, at end of the day, it's only a work of reading code, renaming variable and functions, re-typing, and interpreting APIs along the analysis.

Some functions that could be interesting:

- **sub_180039CC0:** read files
- **sub_18004A32C:** read files
- **sub_180039B90:** write file
- **sub_18004AA18:** write file
- **sub_18003DF44:** create process
- **sub_180050380**: enumerate processes
- **sub_1800124B0:** socket (receive data)
- **sub_180012AA0:** socket (send data)
- **sub_180012030:** socket configuration
- **sub_180015360:** socket configuration
- **.data section:** there's two other PE executables embedded.

About the last statement, it's quite easy to check it by running a simple **strings.exe command**:

```
C:\Users\Administrador\Desktop\MAS\MAS_5>strings -a mas_5_unpacked.bin | grep -i "This program"
!This program cannot be run in DOS mode.
!This program cannot be run in DOS mode.
!This program cannot be run in DOS mode.
```

[Figure 81] Confirming that there're two other two executable within extracted payload

One of many way to extract PE file and other types of objects from a given binary is by using **Binary Refinery** (https://github.com/binref/refinery), which is a sort of command line version of CyberChef.

To install Binary Refinery:

- **pip install -U binary-refinery**

or

- **python -m venv test**
- **. ./test/bin/activate**
- (test) $ **pip install -U git+git://github.com/binref/refinery.git**

To extract both embedded executable, we're going to extract them into a file (**payload_1**) and, afterwards, the second one to another file (**payload_2**), as shown below:

- **emit mas_5_unpacked.bin | carve-pe -r | dump payload_1**
- **emit payload_1 | carve-pe | dump payload_2**

Using the **Binary Refinery,** we're able to visualize the necessary information about both files:

```
C:\Users\Administrador\Desktop\MAS\MAS_5\binary_refinery_out>emit payload_1| peek -mml8 | pemeta -t
--------------------------------------------------------------------------------------------------------------
    crc32 = 0592556f
  entropy = 75.95%
    magic = PE32 executable (DLL) (GUI) Intel 80386, for MS Windows
   sha256 = cce5b5b2668d542e5c3d2f267647da8cf9b182e8e842fe9eb2f1ebf35bd7ba11
     size = 54.785 kB
--------------------------------------------------------------------------------------------------------------
00000: 4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 B8 00 00 00 00 00 00 00 40 00  MZ......................@.
0001A: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..........................
00034: 00 00 00 00 00 00 00 00 E0 00 00 00 0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21  ....................!..L.!
0004E: 54 68 69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F 74 20 62 65 20 72 75 6E  This.program.cannot.be.run
00068: 20 69 6E 20 44 4F 53 20 6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00 39 9F  .in.DOS.mode....$.......9.
00082: D1 2B 7D FE BF 78 7D FE BF 78 7D FE BF 78 74 86 2C 78 7A FE BF 78 7D FE BE 78  .+}..x}..x}..xt.,xz..x}..x
0009C: 50 FE BF 78 7D FE BF 78 7E FE BF 78 EA A0 BA 79 7A FE BF 78 EA A0 BF 79 7C FE  P..x}..x~..x...yz..x...y|.
000B6: BF 78 EA A0 BD 79 7C FE BF 78 52 69 63 68 7D FE BF 78 00 00 00 00 00 00 00 00  .x...y|..xRich}..x........
--------------------------------------------------------------------------------------------------------------
Header.Machine   : I386
Header.Subsystem : Windows GUI
Header.MinimumOS : Windows XP
Header.RICH[0x0] : [00937809] [import] VS2008 SP1 build 30729
Header.RICH[0x1] : [01055e97] [object] VS2015 UPD3.1 build 24215
Header.RICH[0x2] : [01005e97] [export] VS2015 UPD3.1 build 24215
Header.RICH[0x3] : [01025e97] [linker] VS2015 UPD3.1 build 24215
Header.Type      : DLL
Header.Bits      : 32
Header.ImageBase : 0x10000000
Header.ImageSize : 28672
TimeStamp.Linker : 2022-05-25 14:59:16
TimeStamp.DbgDir : 2022-05-25 14:59:16
TimeStamp.Export : 2022-05-25 14:59:16
```

[Figure 82] First DLL extracted using Binary Refinery

```
C:\Users\Administrador\Desktop\MAS\MAS_5\binary_refinery_out>emit payload_2| peek -mml8 | pemeta -t
--------------------------------------------------------------------------------------------------------------
    crc32 = ac8e4649
  entropy = 68.08%
    magic = PE32+ executable (DLL) (GUI) x86-64, for MS Windows
   sha256 = 6ca444e0d6a8a2f1675479d46b666bc33e5e85ec9257be7b6eabb553c92c4008
     size = 26.112 kB
--------------------------------------------------------------------------------------------------------------
00000: 4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 B8 00 00 00 00 00 00 00 40 00  MZ......................@.
0001A: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..........................
00034: 00 00 00 00 00 00 00 00 D8 00 00 00 0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21  ....................!..L.!
0004E: 54 68 69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F 74 20 62 65 20 72 75 6E  This.program.cannot.be.run
00068: 20 69 6E 20 44 4F 53 20 6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00 61 AD  .in.DOS.mode....$.......a.
00082: E7 A2 25 CC 89 F1 25 CC 89 F1 25 CC 89 F1 2C B4 1A F1 22 CC 89 F1 25 CC 88 F1  ..%...%...%...,..."...%...
0009C: 0F CC 89 F1 B2 92 8C F0 2F CC 89 F1 B2 92 89 F0 24 CC 89 F1 B7 92 76 F1 24 CC  ......../.......$.....v.$.
000B6: 89 F1 B2 92 8B F0 24 CC 89 F1 52 69 63 68 25 CC 89 F1 00 00 00 00 00 00 00 00  ......$...Rich%...........
--------------------------------------------------------------------------------------------------------------
Header.Machine   : AMD64
Header.Subsystem : Windows GUI
Header.MinimumOS : Windows Server 2003
Header.RICH[0x0] : [00937809] [import] VS2008 SP1 build 30729
Header.RICH[0x1] : [01055e97] [object] VS2015 UPD3.1 build 24215
Header.RICH[0x2] : [01005e97] [export] VS2015 UPD3.1 build 24215
Header.RICH[0x3] : [00ff5e92] [object] VS2015 UPD3 build 24210
Header.RICH[0x4] : [01025e97] [linker] VS2015 UPD3.1 build 24215
Header.Type      : DLL
Header.Bits      : 64
Header.ImageBase : 0x0000000180000000
Header.ImageSize : 26112
TimeStamp.Linker : 2022-05-25 14:59:21
TimeStamp.DbgDir : 2022-05-25 14:59:21
TimeStamp.Export : 2022-05-25 14:59:21
```

[Figure 83] Second DLL extracted using Binary Refinery

As readers were able to notice, we extracted a 32-DLL and a 64-DLL payload. Don't worry about **payload_1 (32-bit DLL)** having the second DLL inside it because, during an execution, only the first PE is regarded.

Do readers want to extract and decrypt the C2 List? It's trivial because we already have the virtual address, respective size and, most important, we also have the RC4 key (**read page 66**). Therefore, execute:

```
C:\Users\Administrador\Desktop\MAS\MAS_5\binary_refinery_out>emit mas_5_unpacked.bin | vsnip 0x1800FF
340:1024 | peek
----------------------------------------------------------------------------------------------------
01.024 kB; 97.82% entropy; data
----------------------------------------------------------------------------------------------------
00000: 7D 09 37 62 A6 42 88 35 54 D5 46 4D 11 F2 BD 33 94 B4 4F E4 99 80 6D  }.7b.B.5T.FM...3..O...m
00017: C8 55 30 FB 15 31 07 7F 4A 58 7F 22 5E AB D7 B3 B0 65 4B 8D 65 AA 9D  .U0..1..JX."^....eK.e..
0002E: 31 0E 87 4F A9 B5 79 BB AD 67 D3 4C E2 3C BB FB 12 D1 0A E9 8E C3 2F  1..O..y..g.L.<......../
00045: 3C 1E DF E6 00 0B 19 21 1F C0 D1 C2 10 3C 6B 0D 4E 78 B1 FA 3A 9D 44  <......!.....<k.Nx..:.D
0005C: EE DB 22 0D D8 94 E7 F9 B9 BE 4B 55 A6 AB 1B A5 BA 42 67 58 B5 0F 02  ..".......KU.....BgX...
00073: 22 A7 6B 13 A1 93 05 A0 71 F3 B4 48 C5 36 37 90 92 F1 1B 74 7C 68 10  ".k.....q..H.67....t|h.
0008A: DE A0 0D B8 99 CC DB 24 8D 66 DD 20 FD 5B 95 09 E8 DD C0 0D 0C 7D 20  .......$.f...[.......}.
000A1: 1D 7F 4B 32 16 DD EE 8D 5E 5D 4D AC B4 F2 8B A3 27 2E 2C D7 27 08 EA  ..K2....^]M.....'.,.'..
000B8: ED 9B EE AE 15 38 5E 35 F0 F9 47 CF CE 32 BB 49 BF 40 1B 62 FF 5C C5  .....8^5..G..2.I.@.b.\.
000CF: 89 52 28 4F 67 CB 8C 96 19 38 48 30 CE EA 8D 93 C5 2A A0 8C A7 70 10  .R(Og....8H0.....*...p.
----------------------------------------------------------------------------------------------------
```

```
C:\Users\Administrador\Desktop\MAS\MAS_5\binary_refinery_out>emit mas_5_unpacked.bin | vsnip 0x1800FF
340:1024 | rc4 VcFFI2Rj6t15
242.165.212.79:339,162.144.249.150:239,63.122.120.151:268,144.52.138.51:193,18.215.29.142:436,115.239
.67.202:380,255.11.235.99:426,213.203.201.199:307,143.117.20.123:425,141.98.168.70:443,174.150.214.40
:426,133.133.249.24:204,126.68.7.249:422,103.175.16.107:443,146.70.124.77:443,154.56.0.100:443,180.18
4.129.160:223,28.78.74.145:427,108.28.254.44:399,115.103.22.1:153,149.57.112.159:122,229.139.73.188:2
87,112.110.146.153:349,249.222.51.70:286,180.23.251.29:230,244.234.60.83:386,79.133.212.60:211,192.21
.12.118:231,31.215.170.180:431,140.208.107.161:360,119.177.224.146:124,58.10.55.201:382,57.156.134.11
3:446,83.142.26.147:465,194.135.33.16:443,35.17.203.69:268,104.135.8.250:417,210.251.188.194:228,53.9
6.32.99:333,70.77.209.88:224,65.254.82.66:498,65.95.20.151:232,165.158.204.41:469,185.62.58.209:443,1
02.109.16.255:445,137.253.55.69:235
```

[Figure 84] C2 List extracted and decrypted by Binary Refinery

13. Conclusion

In this article, I presented the first 64-bit sample of the MAS Series as well as concepts related to x64 Assembly, COM (Component Object Model), managed difficulties in reversing code that use external libraries, and extracted and decode the C2 List and its respective botnet.

Recently a professional (*Twitter: @bushuo12*) translated the three first articles of this series to Chinese and, just in case you're able to understand the language, **Chinese versions** follow below:

- **(MAS): Article 1** -- https://www.yuque.com/docs/share/619f03dc-1bc9-42f7-828e-fc17d82786e7
- **(MAS) : Article 2** -- https://www.yuque.com/docs/share/d16efbd6-e2e6-4325-9b9e-23c613bd2280
- **(MAS) : Article 3** -- https://www.yuque.com/docs/share/7dca2583-8456-4ca5-8862-0524fc6faaf9

Just in case you want to keep in touch:

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

Keep reversing and I see you at next time!

Alexandre Borges