Malware Analysis Series (MAS): Article 7

by Alexandre Borges
release date: JANUARY/05/2023 | rev: A

0. Quote

"The two most important days in your life are the day you are born and the day you find out why."
(Mark Twain, Ernest T. Campbell, and others, and also mentioned in "The Equalizer" movie -- 2014)

1. Introduction

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

If readers haven't read previous 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/
- **MAS_6:** https://exploitreversing.com/2022/11/24/malware-analysis-series-mas-article-6/

This time we will be analyzing **Dridex**, which a complex banking trojan and that has been updated many times in the last few years. Similar to other malware threats, **Dridex** steals credential information (keylogger behavior) and send it to adversaries using an encrypted set of C2 servers, as usual and seen in other families. On most occasions, it's delivered by a malicious document as an attached file, but it is not the only used vector. In terms of history, Dridex came up in 2014, and it is composed by a loader, which is responsible for installing the payload and downloading additional modules such as VNC and SOCKS support, and the payloader that's able to download additional modules too.

Excellent malware analysts already analyzed it, produced good reports, and our goal here is only show few hard aspects of this binary. As usual, there isn't the purpose of dissecting any malware in this series of articles, but only to present few points of view that enable readers to proceed with their own analysis.

To keep 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, Any.Run** and other ones.

If you want, you might use **Malwoverview tool** (https://github.com/alexandreborges/malwoverview) to download and, get first information and analysis about downloaded sample from most of these services.

2. Acknowledgments

We are in a new year, and it is hard to believe that I started writing this series of articles at end of 2021, about a remarkably simple malware (Hancitor). Reading it again, I realized I included a list of concepts and foundations about code injection, hints about unpacking, and after having done a short analysis, I explained step-by-step how to write a C2 extractor for that family. Being honest, I didn't have any plans for writing a second article about malware analysis, but an unexpected reaction happened in favor of the article and a substantial number of professionals asked to write a second article. Checking my records, more than ten thousand people downloaded the article in less than one month, what also was surprising.

I have been around working with information security, either as primary or secondary work, for so much time. I guess my first serious contact with information security was in 1997! At that time, I read the famous *"Smashing The Stack For Fun And Profit"* (released in 1996 on Phrack *by Aleph One*) and I clearly remember that it took me 45 days to really understand the article. It was a different age, and we didn't have the Google yet. At that time, I knew I had found my passion, but you know…passions are difficult to be followed. Three years later, I worked as security analyst, and my primary role was executing penetration test against company who had contracted the service. Hey… it was the year 2000 and this kind of job was really unknown. Since this time, my passion for reversing engineering and vulnerability research/exploit development has made part of my daily life, even when I worked for big companies doing a different job.

I initiated my reverse engineering career almost two decades ago (even as a secondary job and, sometimes, hobby), but similar to other colleagues, I also read the famous series about exploitation from **Peter Van Eeckhoutte** (a.k.a. **@corelanc0de3r**) since 2009 (thank you for such excellent articles and friendship, Peter), and also articles from other researchers and, of course, my passion was there, equalized over a subtle balance with reverse engineering. After few years, I was focusing and planning to follow my career in vulnerability research, but life had other plans and I returned working full time with reverse engineering (out of big-techs), and for a long sequence of years. Of course, an extensive and excellent list of events happened since 2015, I spoke in many big conferences around the world, met amazing people over the years, learned a lot of stuff, and I am grateful for everything and every single moment.

In the last quarter of 2022, and after many, long years, I definitely returned (migrated) to vulnerability research because, as I said, it has been my passion since ever, and now I finally can focus a hundred percent of my energy on learning and doing something I really love to do, although I use reverse engineering and programming for everything, which have been incredibly useful and, of course, I also like it so much. Probably one of great experiences so far is that I have chance to remember myself that every single day I know less.

As I have mentioned, reverse engineering and malware analysis make part of my life, and I plan to keep speaking about it at conferences (if I have the opportunity to, of course) and writing this series of articles and other new ones , to help professionals because I have realized as much useful these articles have been for many people and, in some cases, this series (**MAS**) has helped them as an initial reference for working in reverse engineering area.

Of course, there wouldn't be this series without receiving the decisive help from **Ilfak Guilfanov (@ilfak)**, from **Hex-Rays SA (@HexRaysSA)** because I didn't have an own IDA Pro license, and he kindly provided everything I needed to write this series about malware analysis and other one that are coming. However, his help didn't stop in 2021, and he and Hex-Rays have continuously helped until the present moment by

providing immediate support for everything I need to keep these public projects. Additionally, Ilfak is always truly kind replying to me in every single time that I sent a message to him.

This section, about acknowledgments, can be translated to one word: gratitude. Personally, all messages from **Ilfak** and **Hex-Rays** expressing their trust and praises on this series of articles until now are one of most motivation to keep writing as well readers who send me even a single message thanking me.

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

I have chosen a quote to start each article to subtly show my thinking about the life and information security in general, sometimes mirroring the present days and all the challenges that have forced to reflect on everything. At the end of day, we should invest in the work that we really love doing, don't matter our age, because the life is short, and the ahed day is our future.

Finally, I leave the same message that **Steven Seely (@steventseeley)** sent me when I mentioned I was finally restarting my career in vulnerability research: *"enjoy the journey".*

3. Environment Setup

This article has a lab setup using 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/ . At time of drafting this article, **IDA Pro 8.2** has been released, and readers should read about the new features: https://hex-rays.com/blog/ida-8-2-released/ . Of course, readers might use other reverse engineering tool, but I'll 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
- **HxD editor:** https://mh-nexus.de/en/hxd/
- **Malwoverview:** https://github.com/alexandreborges/malwoverview
- **Capa: pip install -U flare-capa** | https://github.com/mandiant/capa/releases

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

4. References

Readers are able to find articles, news, references, and reports analyzing **Dridex** and, although I haven't had the opportunity to read them, I recommend readers to do it because they were written by excellent security researchers and companies, who covered and analyzed several aspects of the same family, and readers can learn what's more appropriate for their work. The list below doesn't have any preferred order:

- https://malpedia.caad.fkie.fraunhofer.de/details/win.dridex
- https://us-cert.cisa.gov/ncas/alerts/aa19-339a
- https://unit42.paloaltonetworks.com/excel-add-ins-dridex-infection-chain/
- https://blogs.vmware.com/security/2021/03/analysis-of-a-new-dridex-campaign.html
- https://www.cert.ssi.gouv.fr/uploads/CERTFR-2020-CTI-008.pdf
- https://redcanary.com/threat-detection-report/threats/dridex/

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)**
- http://0xeb.net/ **(by Elias Bachaalany: @0xeb)**
- https://www.hexacorn.com/index.html **(@Hexacorn)**
- 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://azeria-labs.com/writing-arm-assembly-part-1/ **(by Maria Markstedter: @Fox0x01)**
- https://github.com/mnrkbys **(by Minoru Kobayashi: @unkn0wnbit)**
- https://voidsec.com/member/voidsec/ **(by Paolo Stagno: @Void_Sec)**
- https://windows-internals.com/author/yarden/ (**by Yarden Shafir @yarden_shafir**)

Certainly, there're 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

We are going to be working on following Dridex sample (SHA 256):
87e2dad373f75f5c0a200821523aebe45f6f4103b51fb0155ed2bf060ec50b04.dll

Readers can gather first information about the sample from Malware Bazaar:

```
remnux@remnux:~$ malwoverview.py -b 1 -B 87e2dad373f75f5c0a200821523aebe45f6f4103b51fb
155ed2bf060ec50b04 -o 0


                                        MALWARE BAZAAR REPORT

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

sha256_hash:   87e2dad373f75f5c0a200821523aebe45f6f4103b51fb0155ed2bf060ec50b04
sha1_hash:     970fd83b12bd1919ab684723b69c8a90d1a36b9b
md5_hash:      b22a00cefca58fa81234983b81de1fee
first_seen:    2021-12-21 14:48:53
file_name:     87e2dad373f75f5c0a200821523aebe45f6f4103b51fb0155ed2bf060ec50b04
file_size:     479232 bytes
file_type:     dll
mime_type:     application/x-dosexec
imphash:       d67883ee85eede67419711a8fbd7ca0d
tlsh:          T11EA4AF3181C5528AD705123423DA8065227F5326CC957FBF9CF982732A6BAEDDE3E0D6
comments:
               Dridex distributed via Log4Shell
reporter:      blubbfiction
delivery:      web_download
tags:          dll Dridex
UnpacMe:
               https://www.unpac.me/results/a5bb94cc-796b-47db-9268-adb68cc05656/

Triage:        https://tria.ge/reports/211221-r61ksadgh5/
Triage sigs:
               Dridex
               Dridex Loader
               Program crash
               Suspicious behavior: EnumeratesProcesses
               Suspicious behavior: GetForegroundWindowSpam
               Suspicious use of AdjustPrivilegeToken
               Suspicious use of WriteProcessMemory
```

[Figure 1]: Checking sample information on Malware Bazaar

Additionally, this sample can be easily downloaded by running the following command:

- **malwoverview.py -b 5 -B 87e2dad373f75f5c0a200821523aebe45f6f4103b51fb0155ed2bf060ec50b04 -o 0**

Remember that the download is protected with a password: **infected**. Thus, you'll need to unpack it by running: **7z e 87e2dad373f75f5c0a200821523aebe45f6f4103b51fb0155ed2bf060ec50b04 -pinfected**.

From **Triage** (you should first get the job ID with: **malwoverview.py -x 1 -X <hash> -o 0**), we have:

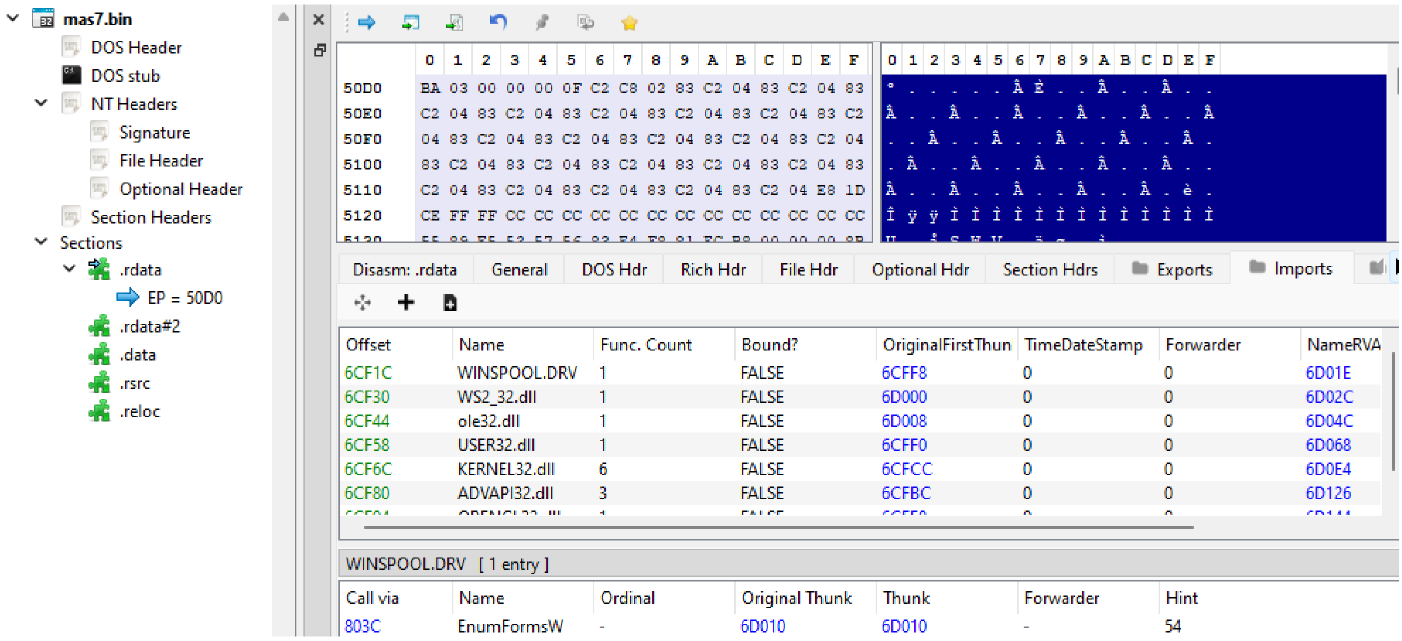
```
remnux@remnux:~/malware/mas/mas_7$ malwoverview.py -x 2 -X 211221-r61ksadgh5 -o 0

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

score:     10
extracted:
           botnet:   22206
           c2:
                     120.50.40.185:443
                     139.59.14.223:8172
                     121.40.104.209:6602
                     139.162.113.169:593
           family:   dridex
           key:      key
           value:    S9OYlNFUvY5N1RDSpi8BgH6SgS8gPIcU
           key:      key
           value:    zwTHMB1SiSgHnmlqIchyvEq6lSioc0XHE4rT4eCydGgyrIipLBzPItrelc82jktTbqgPlT4yGq
           rule:     DridexLoader
           dumped:   memory/748-56-0x00000000746F0000-0x0000000074766000-memory.dmp
           resource: behavioral1/memory/748-56-0x00000000746F0000-0x0000000074766000-memory.dmp
           tasks:    behavioral1 behavioral2

id:        211221-r61ksadgh5
target:    87e2dad373f75f5c0a200821523aebe45f6f4103b51fb0155ed2bf060ec50b04
size:      479232
md5:       b22a00cefca58fa81234983b81de1fee
sha1:      970fd83b12bd1919ab684723b69c8a90d1a36b9b
sha256:    87e2dad373f75f5c0a200821523aebe45f6f4103b51fb0155ed2bf060ec50b04
completed: 2021-12-21T14:51:45Z

signatures:
           Dridex
           Dridex Loader
           Program crash
           Suspicious behavior: EnumeratesProcesses
           Suspicious behavior: GetForegroundWindowSpam
           Suspicious use of AdjustPrivilegeToken
           Suspicious use of WriteProcessMemory

targets:
           family:   dridex
           iocs:
                     time.windows.com
                     52.109.8.20
                     8.8.8.8
                     168.61.215.74
           md5:      b22a00cefca58fa81234983b81de1fee
           score:    10
           sha1:     970fd83b12bd1919ab684723b69c8a90d1a36b9b
           sha256:   87e2dad373f75f5c0a200821523aebe45f6f4103b51fb0155ed2bf060ec50b04
           size:     479232bytes
           tags:
                     family:dridex
                     botnet:22206
                     botnet
                     loader
           target:   87e2dad373f75f5c0a200821523aebe45f6f4103b51fb0155ed2bf060ec50b04
           tasks:    behavioral1 behavioral2
```

[Figure 2]: Triage information about the sample

We already have enough information about the sample:

- **This sample is really the Dridex loader.**
- **Possibly enumerates processes and changes privileges.**
- **It could be using WriteProcessMemory( ) for injection.**
- **The associated botnet is 22206.**
- **It communicates with a series of C2 servers (to be checked later).**
- **Two keys are presented.**

Checking its imported functions (there isn't any exported function even being a DLL) using **PE Bear**:



[Figure 3]: Triage information about the sample

This sample, like other **Dridex samples**, has anti-debugging tricks to delay the malware analysis. Here you have a list of options to unpack the sample:

- Using **x64dbg** with **ScyllaHide plugin** (used against anti-debugging techniques) and setting well-known breakpoints (check the first and second articles from this series).
- Using **OllyDbg** with **StrongOD or Phantom** plugins (they are used against anti-debugging techniques) and setting up well-known breakpoints (check first and second articles from this series).
- Using **hollows_hunter**: https://github.com/hasherezade/hollows_hunter
- Using **pe-sieve**: https://github.com/hasherezade/pe-sieve
- Using **UnpacMe** service from OALabs: https://www.unpac.me/

To use **hollows_hunter** tool, one of the suggested syntax is: **hollows_hunter64.exe /pname rundll32.exe /loop**

Regardless of the process you chose to perform the unpacking process, likely you will get two binaries:

- **SHA 256**: 45feffe2ffb4ccc9be7a9f83dff63872fd2cf0f2e73294437e129049c311e6e7 (DLL)
- **SHA 256:** d5d8e409720272563108e7a665d8d7d2fa4c773efdd260b85d3424e35618b963 (DLL)

The first one is really small (about 9 KB), but the second one has about 132 KB and it will be our target.

Offset	Name	Func. Count	Bound?	OriginalFirstThun	TimeDateStamp	Forwarder	NameRVA
1CC38	KERNEL32.dll	15	FALSE	1CC90	0	0	1CDEC
1CC4C	USER32.dll	1	FALSE	1CCD0	0	0	1CE08
1CC60	ADVAPI32.dll	1	FALSE	1CC88	0	0	1CE24

KERNEL32.dll [15 entries]

Call via	Name	Ordinal	Original Thunk	Thunk	Forwarder	Hint
1A008	LCMapStringA	-	1CCFE	77E3FC10	-	394
1A00C	IsBadReadPtr	-	1CD0E	77E1BEB3	-	35E
1A010	HeapValidate	-	1CD1E	77E230AD	-	33B
1A014	GetStringTypeA	-	1CD2E	77E16D95	-	2C2
1A018	GetStartupInfoA	-	1CD40	77DE1E10	-	2BD
1A01C	GetLocaleInfoA	-	1CD52	77E1B5C3	-	252
1A020	LoadLibraryA	-	1CCEE	77E2DEA5	-	3A5
1A024	GetConsoleOutputCP	-	1CD7C	77E35343	-	1F2
1A028	FreeEnvironmentStringsA	-	1CD92	77E3CB62	-	19C
1A02C	FlushFileBuffers	-	1CDAC	77E1873F	-	192
1A030	DebugBreak	-	1CDC0	77E70385	-	FB
1A034	CreateFileA	-	1CDCE	77E2ECA1	-	BA

[Figure 4]: PE Bear: imported functions

Disasm | General | DOS Hdr | Rich Hdr | File Hdr | Optional Hdr | Section Hdrs | Exports | Imports

Offset	Name	Value	Meaning
1C9A0	Characteristics	0	
1C9A4	TimeDateStamp	FFFFFFFF	Sunday, 07.02.2106 06:28:15 UTC
1C9A8	MajorVersion	0	
1C9AA	MinorVersion	0	
1C9AC	Name	1CA2C	mshtmled.dll
1C9B0	Base	1	
1C9B4	NumberOfFunctions	5	
1C9B8	NumberOfNames	5	
1C9BC	AddressOfFunctions	1C9C8	
1C9C0	AddressOfNames	1C9F0	

Exported Functions [5 entries]

Offset	Ordinal	Function RVA	Name RVA	Name	Forwarder
1C9C8	1	12508	1CA39	DllCanUnloadN...	
1C9CC	2	18A1B	1CA49	DllEnumClassO...	
1C9D0	3	15F64	1CA5D	DllGetClassObject	
1C9D4	4	21C8	1CA6F	DllRegisterServer	
1C9D8	5	7F87	1CA81	DllUnregisterSer...	

[Figure 5] PE Bear: exported functions

Interestingly, this **DLL's name (mshtmled.dll)** has been also viewed in other **Dridex samples** and, in a general way, there're few changes among them.

Further and valuable information can also be collected by using **capa.exe** (its standalone version) as shown below (the sample was renamed to **mas_7_unpacked.bin**):

```
C:\Users\Administrador\Desktop\MAS\MAS_7>capa mas_7_unpacked.bin
loading : 100%|████████████████████████████████████████████████████████████| 76
matching: 100%|████████████████████████████████████| 272/272 [00:07<00:00, 38.20 functions/
+------------------------+------------------------------------------------------------------------------------+
| md5                    | 1d24a6c36f320b3b87c6e61a10b47432                                                   |
| sha1                   | c064b3e1b807331fba9321a69264437bcd08076f                                           |
| sha256                 | d5d8e409720272563108e7a665d8d7d2fa4c773efdd260b85d3424e35618b963                   |
| os                     | windows                                                                            |
| format                 | pe                                                                                 |
| arch                   | i386                                                                               |
| path                   | mas_7_unpacked.bin                                                                 |
+------------------------+------------------------------------------------------------------------------------+

+------------------------+------------------------------------------------------------------------------------+
| ATT&CK Tactic          | ATT&CK Technique                                                                   |
|------------------------+------------------------------------------------------------------------------------|
| DEFENSE EVASION        | Obfuscated Files or Information T1027                                              |
|                        | Virtualization/Sandbox Evasion::System Checks T1497.001                            |
| DISCOVERY              | Process Discovery T1057                                                            |
| EXECUTION              | Shared Modules T1129                                                               |
+------------------------+------------------------------------------------------------------------------------+

+-----------------------------+-------------------------------------------------------------------------------+
| MBC Objective               | MBC Behavior                                                                  |
|-----------------------------+-------------------------------------------------------------------------------|
| ANTI-BEHAVIORAL ANALYSIS    | Virtual Machine Detection [B0009]                                             |
| CRYPTOGRAPHY                | Encrypt Data::RC4 [C0027.009]                                                 |
|                             | Encryption Key::RC4 KSA [C0028.002]                                           |
|                             | Generate Pseudo-random Sequence::RC4 PRGA [C0021.004]                         |
| DATA                        | Check String [C0019]                                                          |
|                             | Encode Data::Base64 [C0026.001]                                               |
|                             | Encode Data::XOR [C0026.002]                                                  |
| DEFENSE EVASION             | Disable or Evade Security Tools::Heavens Gate [F0004.008]                     |
|                             | Obfuscated Files or Information::Encoding-Standard Algorithm [E1027.m02]      |
+-----------------------------+-------------------------------------------------------------------------------+

+------------------------------------------------------+------------------------------------------------------+
| CAPABILITY                                           | NAMESPACE                                            |
|------------------------------------------------------+------------------------------------------------------|
| 64-bit execution via heavens gate (3 matches)        | anti-analysis/anti-disasm                            |
| reference anti-VM strings                            | anti-analysis/anti-vm/vm-detection                   |
| reference Base64 string                              | data-manipulation/encoding/base64                    |
| encode data using XOR                                | data-manipulation/encoding/xor                       |
| encrypt data using RC4 KSA                           | data-manipulation/encryption/rc4                     |
| encrypt data using RC4 PRGA                          | data-manipulation/encryption/rc4                     |
| contain a resource (.rsrc) section                   | executable/pe/section/rsrc                           |
| get process heap force flags                         | host-interaction/process                             |
| access PEB ldr_data (2 matches)                      | linking/runtime-linking                              |
| parse PE header                                      | load-code/pe                                         |
| resolve function by parsing PE exports (3 matches)   | load-code/pe                                         |
+------------------------------------------------------+------------------------------------------------------+
```

[Figure 6]: Malware profiling using Capa

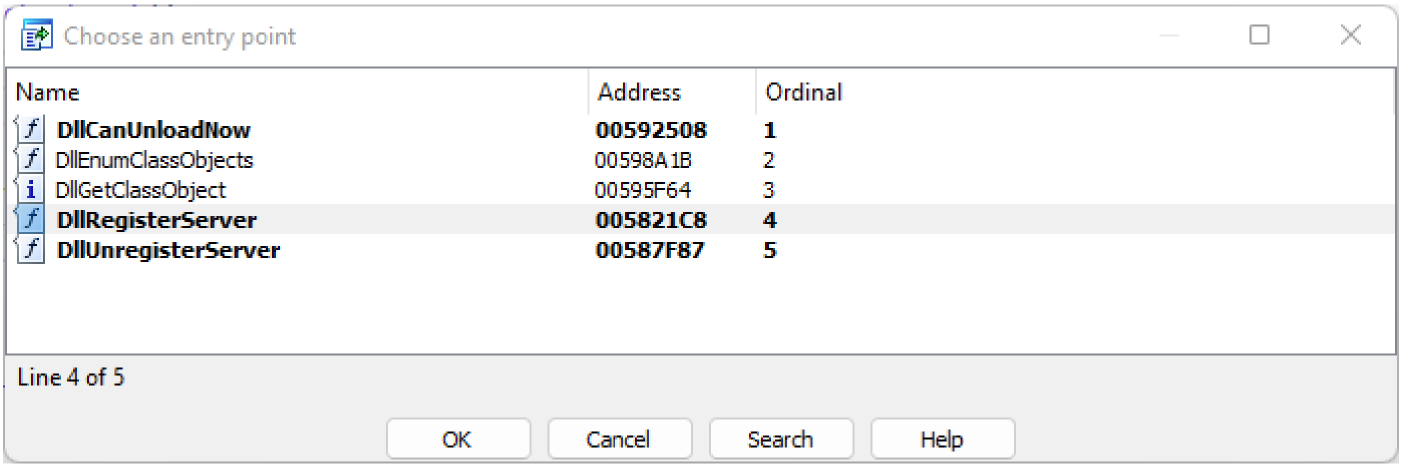
From the output above, quite important items are:

- **Detects virtual machine**
- **Uses RC4 algorithm**
- **Tries to disable security tools**
- **Information can be obfuscated**
- **Gets process heap force flags**
- **Parses PE header**

We have enough information to proceed to the reversing engineering stage using **IDA Pro**, when we will try to understand few points of this malware sample.

7. Reversing

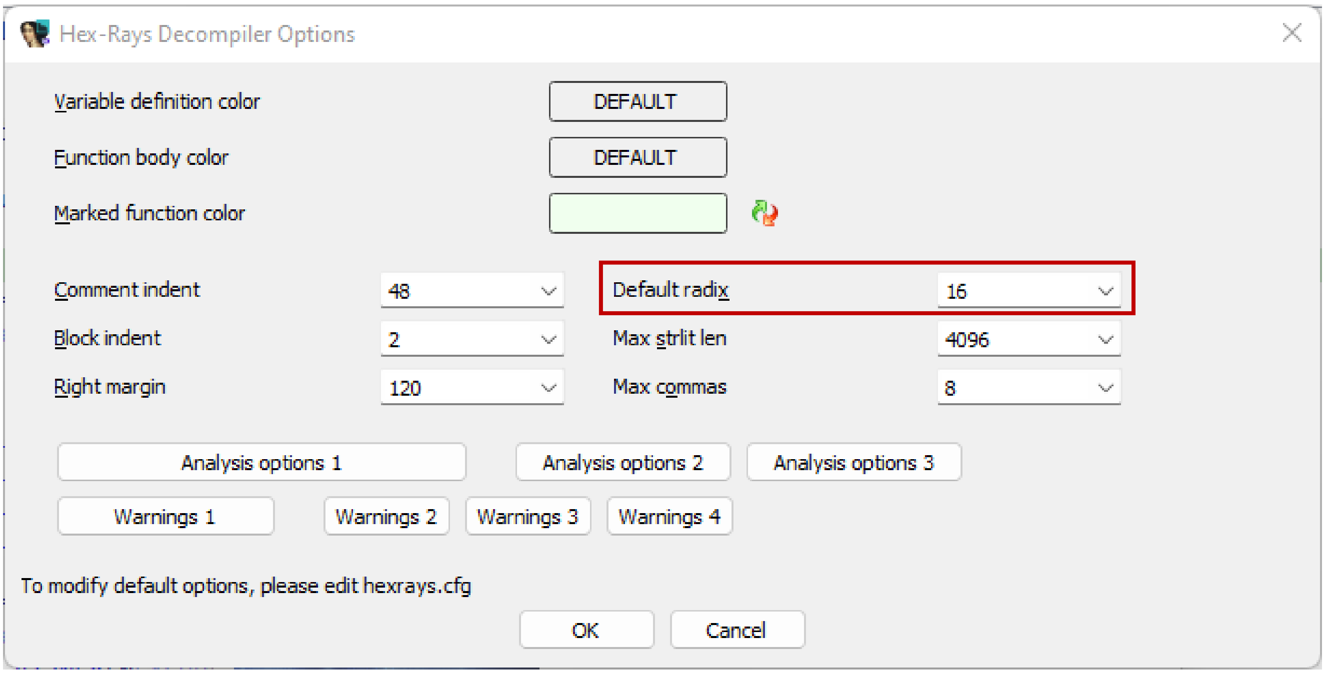
Dridex is a strange malware that demands our attention since beginning to understand what's really happening. As the unpacked sample is a DLL, so there are one or more exported functions and, of course, our first step is trying to understand which one we should or not to follow. Listing the possible entry points (**CTRL+E**), we learn that, in this case, there are five exported functions (potential entry points):



[Figure 7]: Exported functions and possible entry points

We must walk slowly here because the malware's author might be misleading us and, eventually, one or more than these functions might be fake exported functions. Additionally, names might not be what we are expecting, so we have to check up all functions and trying to find how to begin the analysis.

My first suggestion for readers is for configuring the decompiler to show values in hexadecimal instead of decimal to accelerate analysis. This task can be done by going to **Edit | Plugins | Hex-Rays Decompiler | Options** and make the change "**Default radix**" to **16**, as shown below:



[Figure 8]: Hex-Rays Decompiler

Next step, as usual, we must perform the entire decompilation of the binary to force the IDA Pro to show us the best representation of the pseudo code. To do it: **File | Produce File | Create C File**... Pay attention to this detail here: eventually, we have to do it again later.

Opening the sample, we can see a bunch of routines' calls and, as expected, it attracts our attention:

```
HRESULT __fastcall sub_581000(int *a1, _DWORD *a2)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  v2 = a1;
  sub_58BF3C((int)v20, (char *)0x1D, a1[7]);
  sub_58C494((int)v22, (int)v20[0]);
  sub_58DFA4(v20);
  if ( sub_58C9C0(v22, v2, 0) )
  {
    sub_58F584(v21, 0);
    sub_58F584(v18, 0);
    sub_58F584(v19, 0);
    v5 = v2 + 3;
    if ( !v2 )
      v5 = 0;
    if ( sub_58C280(v5, (int)v2) )
      sub_58FCA4(v2, (int *)0x1FFFFF);
    v6 = v2 + 3;
    if ( !v2 )
      v6 = 0;
    v7 = *v6;
    sub_58CD04((char *)v23, 0);
    sub_5812DC(v15, v7, (int)v23, a2);
    if ( LOBYTE(v15[0]) )
    {
      v8 = sub_58FB5C(v2);
      v9 = &unk_59A6C0;
      if ( v8 == 0x20 )
        v9 = &unk_59A060;
```

[Figure 9]: sub_581000 routine

Unfortunately, things get complicated quickly. For example, go into **sub_58C9C0 → sub_592518** and you will find something like:

```
HANDLE __usercall sub_592518@<eax>(int a1@<ecx>, int *a2@<esi>)
{
  HANDLE result; // eax

  if ( !a1 )
    return NtCurrentTeb()->ClientId.UniqueThread;
  result = (HANDLE)sub_59306C(0x8E844D1E, 0x333A3BAF, a2, 0x8E844D1E, 0x8E844D1E);
  if ( !result )
    return 0;
  __debugbreak();
  __debugbreak();
  return result;
}
```

[Figure 10]: sub_592518 routine

There're interesting points here:

- **sub_59306C(0x8E844D1E, 0x333A3BAF, a2, 0x8E844D1E, 0x8E844D1E);**
- **two calls to __debugbreak();**
- **a strange return in the middle of the code, besides the second one at the final.**

Before continuing, it is recommended to load additional **Type Libraries** (**SHIFT+F11**) as we have done in previous articles: **ntapi_win7** (likely, **mssdk_win7** is already loaded). At the same way, I suggest you loading additional signatures (**SHIFT+F5**): **vc32ucrt, vc32rtf, vc32mfc, vc432_14** and **pe**. Please, readers should realize that, as this sample seems to be obfuscated and the control flow is also messed up, so probably these new signatures will not get effective. However, it does not matter because in most cases all these new signatures and libraries will help us, so it's interesting to get used to loading them.

The **assembly code** of the pseudo code shown in **Figure 10** follows below:

```
.text:00592518 ; HANDLE __usercall sub_592518@<eax>(int@<ecx>, int *@<esi>)
.text:00592518 sub_592518      proc near               ; CODE XREF: sub_58C9C0+6↑p
.text:00592518                 push    ebx
.text:00592519                 mov     ebx, ecx
.text:0059251B                 test    ebx, ebx
.text:0059251D                 jz      short loc_59253D
.text:0059251F                 mov     eax, 8E844D1Eh
.text:00592524                 mov     edx, 333A3BAFh
.text:00592529                 push    eax
.text:0059252A                 push    eax
.text:0059252B                 call    sub_59306C
.text:00592530                 test    eax, eax
.text:00592532                 jz      short loc_592539
.text:00592534                 push    ebx
.text:00592535                 int     3               ; Trap to Debugger
.text:00592536                 int     3               ; Trap to Debugger
.text:00592537                 jmp     short loc_592543
.text:00592539 ; ---------------------------------------------------------------------------
.text:00592539
.text:00592539 loc_592539:                             ; CODE XREF: sub_592518+1A↑j
.text:00592539                 xor     eax, eax
.text:0059253B                 jmp     short loc_592543
.text:0059253D ; ---------------------------------------------------------------------------
.text:0059253D
.text:0059253D loc_59253D:                             ; CODE XREF: sub_592518+5↑j
.text:0059253D                 mov     eax, large fs:24h
.text:00592543
.text:00592543 loc_592543:                             ; CODE XREF: sub_592518+1F↑j
.text:00592543                                         ; sub_592518+23↑j
.text:00592543                 pop     ebx
.text:00592544                 retn
.text:00592544 sub_592518      endp ; sp-analysis failed
.text:00592544
.text:00592544 ; ---------------------------------------------------------------------------
.text:00592545                 align 4
```

[Figure 11]: sub_592518 Assembly code

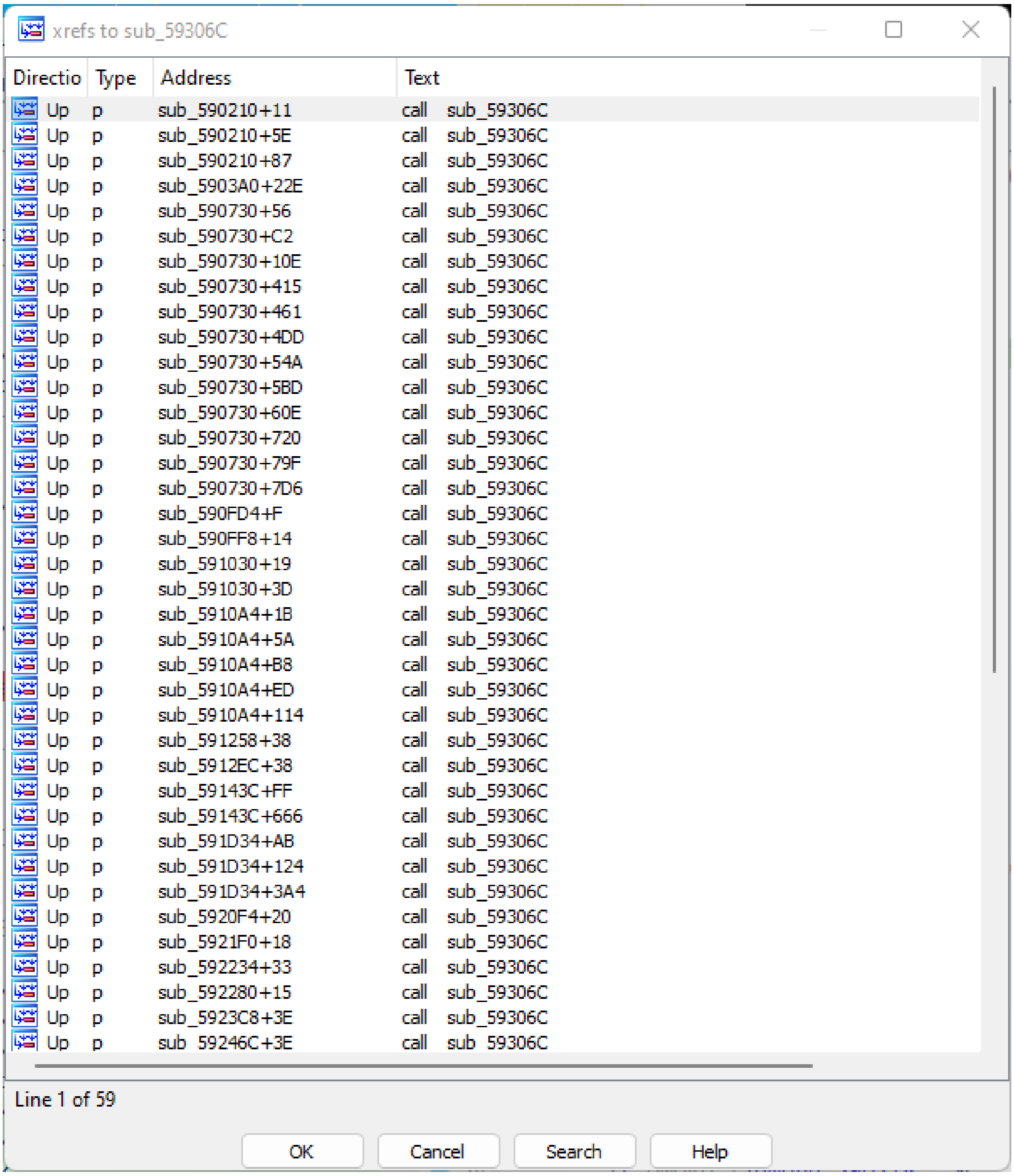
As mentioned in the previous page, the first point to comment is about the following subroutine call:

- **sub_59306C(0x8E844D1E, 0x333A3BAF, a2, 0x8E844D1E, 0x8E844D1E);**

Initially, and based on previous articles (remember Quakbot and Emotet in second and third articles of this series), we can make few guesses:

- **sub_59306C( )** is a function that is responsible for the **hashing resolving task.**
- An educated guess is that one of these hashes **is a DLL hash and the other one is an API hash**.
- There could be or not a **XOR key** involved with the hashing procedure.

If **s sub_59306C( )** is a hash resolving function, so it will be called many times and the fastest way to learn about it is list all cross-references (**X hotkey**) as shown below:



Directio	Type	Address	Text
Up	p	sub_590210+11	call sub_59306C
Up	p	sub_590210+5E	call sub_59306C
Up	p	sub_590210+87	call sub_59306C
Up	p	sub_5903A0+22E	call sub_59306C
Up	p	sub_590730+56	call sub_59306C
Up	p	sub_590730+C2	call sub_59306C
Up	p	sub_590730+10E	call sub_59306C
Up	p	sub_590730+415	call sub_59306C
Up	p	sub_590730+461	call sub_59306C
Up	p	sub_590730+4DD	call sub_59306C
Up	p	sub_590730+54A	call sub_59306C
Up	p	sub_590730+5BD	call sub_59306C
Up	p	sub_590730+60E	call sub_59306C
Up	p	sub_590730+720	call sub_59306C
Up	p	sub_590730+79F	call sub_59306C
Up	p	sub_590730+7D6	call sub_59306C
Up	p	sub_590FD4+F	call sub_59306C
Up	p	sub_590FF8+14	call sub_59306C
Up	p	sub_591030+19	call sub_59306C
Up	p	sub_591030+3D	call sub_59306C
Up	p	sub_5910A4+1B	call sub_59306C
Up	p	sub_5910A4+5A	call sub_59306C
Up	p	sub_5910A4+B8	call sub_59306C
Up	p	sub_5910A4+ED	call sub_59306C
Up	p	sub_5910A4+114	call sub_59306C
Up	p	sub_591258+38	call sub_59306C
Up	p	sub_5912EC+38	call sub_59306C
Up	p	sub_59143C+FF	call sub_59306C
Up	p	sub_59143C+666	call sub_59306C
Up	p	sub_591D34+AB	call sub_59306C
Up	p	sub_591D34+124	call sub_59306C
Up	p	sub_591D34+3A4	call sub_59306C
Up	p	sub_5920F4+20	call sub_59306C
Up	p	sub_5921F0+18	call sub_59306C
Up	p	sub_592234+33	call sub_59306C
Up	p	sub_592280+15	call sub_59306C
Up	p	sub_5923C8+3E	call sub_59306C
Up	p	sub_59246C+3E	call sub_59306C

[Figure 12]: sub_59306C being called 59 times

We seem to be in the right way, but we need to examine the function because this routine could be a wrapper. Additionally, there could be additional **hash resolving functions**.

Anyway, we can rename it temporarily to **ab_maybe_hash_resolving** and, after we have confirmed its working, so we rename it again. Going inside it, we find **sub_59306C** that shows us the following:

```
int __userpurge sub_59306C@<eax>(int a1@<eax>, int a2@<edx>, int *a3@<esi>, int a4, int a5)
{
  int result; // eax
  int v8; // eax

  result = sub_590304(a2, 1);
  if ( !result )
  {
    if ( a1 != 0x39731522
      && ((v8 = sub_591D34(a1, a1)) != 0 || (unsigned __int8)sub_5903A0(a1, (int)a3) && (v8 = sub_591D34(a1, a1)) != 0) )
    {
      return sub_59143C(v8, a2, a3, v8, v8);
    }
    else
    {
      return 0;
    }
  }
  return result;
}
```

[Figure 13]: sub_59306C routine

Moving into the **sub_590304** routine, we have the following scenario:

```
int __fastcall sub_590304(int a1, char a2)
{
  int v3; // ecx
  int v5; // edx
  int result; // eax
  int v7; // ebp
  int v8; // edx

  v3 = dword_59D208;
  if ( dword_59D208 == 0xC55649E1 )
  {
    v3 = 0;
    dword_59D208 = 0;
  }
  switch ( a1 )
  {
    case 0x1C6EF387:
      v5 = dword_59D20C;
      if ( dword_59D20C == 0xF0909E5B )
        break;
      return v5;
    case 0x45B68B68:
      v5 = dword_59D210;
      if ( dword_59D210 != 0xB85BC9BE )
        return v5;
      break;
    case 0x2EA96E2A:
      v5 = dword_59D214;
      if ( dword_59D214 != 0x4C71E88D )
        return v5;
      break;
  }
```

[Figure 14]: sub_590304 routine

Taking **line 9 (Figure 13)** and **lines 10, 19, 24, 29 (Figure 14)** comparisons with hexadecimals remind me the context of **Emotet**, where we have **control-flow flattening** (check for the third article of this series), so readers might assume that, apparently, there's a **potential obfuscation** applied to the code.

This time is not so critical to be so precise and, for now, we can say that this sample is using obfuscation and, in special, a form of control-flow flattening. Anyway, it isn't what we are looking for. Examining **sub_591D34**, which accept two equal arguments, we have as **30 first lines** the following content:

```
void *__userpurge sub_591D34@<eax>(int a1@<eax>, int a2)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  v3 = dword_59D21C;
  if ( dword_59D21C == 0xF4C64A91 )
  {
    v3 = 0;
    dword_59D21C = 0;
  }
  if ( a1 == 0x60A28C5C )
  {
    result = (void *)dword_59D220;
    if ( dword_59D220 != 0x248DF160 )
      return result;
    ProcessEnvironmentBlock = NtCurrentTeb()->ProcessEnvironmentBlock;
    if ( !v3 )
      goto LABEL_29;
  }
  else
  {
    ProcessEnvironmentBlock = NtCurrentTeb()->ProcessEnvironmentBlock;
    if ( a1 == 0x39731522 )
      return ProcessEnvironmentBlock->ImageBaseAddress;
LABEL_13:
    if ( !v3 )
    {
      if ( a1 == 0xA731522 )
      {
        for ( i = 0; ; i += v40 )
```

[Figure 15]: sub_591D34 routine

Obviously, this **sub_591D34** routine is **parsing the _PEB structure and PE structures** and, when these operations happen in a context as this one that we are analyzing (within of a call like **ab_maybe_hashing_resolving(0x8E844D1E, 0xF9D6C1FF, a2, 0x8E844D1E, 0x8E844D1E))**, so we can assume that DLL and/or API hashing resolutions are involved, as we suspected previously. If readers want to improve the presented code, so it's necessary to add the following standard structures:

- **_PEB**
- **_PEB_LDR_DATA**
- **_LDR_DATA_TABLE_ENTRY**
- **_IMAGE_DOS_HEADER**

- **_IMAGE_NT_HEADERS**
- **_IMAGE_OPTIONAL_HEADER**
- **_IMAGE_SECTION_HEADER**
- **_IMAGE_DATA_DIRECTORY**

We should not be concerned if we are going to use or not all these added structures, but if we need them, so they will be already available. Examining **sub_591D34 routine**, which it's exceptionally long, certainly we will find interesting pieces of codes. For example, take a look at the **while loop on line 175**:

```
175         while ( counter_1 <= Length );
176         v18 = Length == 0;
177         v20 = 0;
178         if ( !v18 )
179         {
180           v21 = 0;
181           do
182           {
183             v22 = *(char *)(Buffer + 2 * v20);
184             v33 = v21;
185             if ( (unsigned int)(v22 - 0x41) <= 0x19 )
186               v22 = (char)(v22 + 0x20);
187             v41[v20] = v22;
188             if ( !v22 )
189               break;
190             ++v21;
191             ++v20;
192           }
193           while ( v33 + 1 < Flink->BaseDllName.Length );
194         }
195         if ( a1 == (sub_594FFC(v41, v20) ^ 0xE462D21C) )
196           break;
197         if ( Flink == (LDR_DATA_TABLE_ENTRY *)v35[0] )
198           return (void *)DllBase;
199         Flink = Flink->InLoadOrderLinks.Flink;
200       }
201       DllBase = (int)Flink->DllBase;
202       if ( !DllBase )
203         return (void *)DllBase;
204       goto LABEL_43;
205     }
206   }
```

[Figure 16]: sub_591D34 routine: a small piece of code

The malware's author is concerned to normalize case of the DLL name to lowercase as readers can verify on **lines 185** and **186**. Additionally, there is a remarkably interesting point which we are talking about in next pages: there is an XOR operation on **line 195** and, according to past experience, we already know that this value (**0xE462D21C**) will be particularly important during the process of hash resolving. Finally, in this code, I only altered the types of Flink and Blink to **LDR_DATA_TABLE_ENTRY *** by using "**Y hotkey**", and this minor change brough us a bit more of context.

Returning to **sub_5906C** (**Figure 13**), let's examine the **sub_59143C routine (line 12)**. From this point onward and assuming you know about the **PE structure**, a minimal work of changing variable types (**Y hotkey**), renaming variables (**N hotkey**) by using these mentioned types and adding the **MACRO_IMAGE enumeration**, we have the following result:

```
1  char *__userpurge sub_59143C@<eax>(int a1@<eax>, char *a2@<edx>, int *a3@<esi>, int a4, int a5)
2  {
3    // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]
4
5    v71 = a2;
6    result = (char *)sub_590304((int)a2, 1);
7    if ( !result )
8    {
9      v64 = 0;
10     if ( dword_59D208 && !byte_59D2E4 )
11     {
12       v65 = 0;
13       byte_59D2E4 = 1;
14       sub_58F584(v66, 0);
15       sub_58F584(v94, 0x1C);
16       v72 = (signed int *)sub_58F4BC(v94, 0);
17       Ldr = NtCurrentTeb()->ProcessEnvironmentBlock->Ldr;
18       Blink = Ldr->InLoadOrderModuleList.Blink;
19       Flink = Ldr->InLoadOrderModuleList.Flink;
20       Blink_1 = Blink;
21       while ( 1 )
22       {
23         DllBase = Flink->DllBase;
24         v61 = 0;
25         ptr_nt_headers = (_IMAGE_NT_HEADERS *)((char *)DllBase + DllBase->e_lfanew);
26         sub_58F584(v62, 0);
27         SizeOfOptionalHeader = ptr_nt_headers->FileHeader.SizeOfOptionalHeader;
28         NumberOfSections = ptr_nt_headers->FileHeader.NumberOfSections;
29         NumberOfSections_1 = NumberOfSections;
30         ptr_image_section_header = (_IMAGE_SECTION_HEADER *)((char *)&ptr_nt_headers->OptionalHeader
31                                                            + SizeOfOptionalHeader);
32         if ( NumberOfSections )
33         {
34           v12 = 0;
35           do
36           {
37             if ( (ptr_image_section_header->Characteristics & IMAGE_SCN_MEM_EXECUTE) != 0 )
```

[Figure 17]: a first piece code code from sub_59143C after a minimal work

This function is huge (more than three hundred lines of pseudo code) and even without analyzing the entire routine, we can understand that the call for **sub_59306C(0x8E844D1E, 0x333A3BAF, a2, 0x8E844D1E, 0x8E844D1E)** is performing PE parsing to perform a possible API hashing task. There're other points to be checked, but that's enough for now.

At the same function (**sub_59143C**), there is a small trick: on **line 222** we seen the same **XOR operation we mentioned previously**, so we could go a little further and to examine a new piece of code. Before proceeding, return to **sub_59143C** signature:

- **char *__userpurge sub_59143C@<eax>(int a1@<eax>, char *a2@<edx>, int *a3@<esi>, int a4, int a5)**

The fourth and fifth parameters (I will rename them to *arg_4* and *arg_5*) aren't being used. Furthermore, I will also rename the first two arguments to *ptr_dll* and *ptr_api_name* respectively (names could be better). This behavior using one or more fake arguments for a routine is a well-known resource used by obfuscators and protectors in general. About the name, they might be not precise, but we can update them later just in case we need to do.

At the same way we did previously, it is necessary to re-type (**Y hotkey**) and rename (**N hotkey**) all possible variables to get a reasonable result as shown in the next figure:

```
225   if ( ptr_dll )
226   {
227     ptr_nt_headers_1 = *(ptr_dll + 0x3C);
228     ptr_export = *(&ptr_nt_headers_1->OptionalHeader.DataDirectory[0].VirtualAddress
229                  + ptr_dll);
230     size_table = (ptr_export
231                 + *(&ptr_nt_headers_1->OptionalHeader.DataDirectory[0].Size
232                   + ptr_dll));
233     AddressOfNames = (ptr_dll + *(&ptr_export->AddressOfNames + ptr_dll));
234     AddressOfNameOrdinals = (ptr_dll
235                            + *(&ptr_export->AddressOfNameOrdinals + ptr_dll));
236     if ( *(&ptr_export->NumberOfNames + ptr_dll) )
237     {
238       counter_2 = 0;
239       api_hashed = v71 ^ 0xE462D21C;
240       while ( 1 )
241       {
242         counter_3 = 0;
243         ptr_address_of_names = (ptr_dll + *AddressOfNames);
244         LOBYTE(ptr_address_of_names_1) = *ptr_address_of_names;
245         if ( ptr_address_of_names_1 )
246         {
247           do
248           {
249             ptr_names = ptr_address_of_names[++counter_3];
250             *(&ptr_address_of_names_1 + counter_3) = ptr_names;
251           }
252           while ( ptr_names );
253         }
254         target_api_hash = sub_594FFC(&ptr_address_of_names_1, counter_3);
255         if ( target_api_hash == api_hashed )
256           break;
257         ++AddressOfNames;
258         ++AddressOfNameOrdinals;
259         if ( ++counter_2 >= *(&ptr_export->NumberOfNames + ptr_dll) )
260           goto LABEL_79;
261       }
262       ptr_function = 0;
263       addr_rva_next_function = *(ptr_dll
264                                + *(&ptr_export->AddressOfFunctions + ptr_dll)
265                                + 4 * *AddressOfNameOrdinals);
266       if ( addr_rva_next_function < ptr_export
267         || addr_rva_next_function >= size_table )
268       {
269         addr_function = addr_rva_next_function + ptr_dll;
270         ptr_function = addr_function;
271       }
```

[Figure 18]: sub_59143C (second part)

Now, our point of interest is the **line 239**:

- **api_hashed = api_hash_name ^ 0xE462D21C (previously api_hashed = v71 ^ 0xE462D21C)**

Clearly the API hash value, which is a representation of the API's name, is being XOR'ed with a constant that works as a XOR key. Actually, this behavior is common and similar to other malware families, and we have to pay attention and consider this value when we will be trying to use any plugin to accelerate the API hash resolving process.

Certainly, a question that many readers might be would be: *"Do I have to follow this procedure of analysis only to find the XOR key?"*. No, you don't.

One of the quickest (and certainly dirty) way to find the XOR key (if there is one) is by using the search resource of IDA Pro to look for all XOR operations and, likely, if you find a repetitive XOR operation using the same immediate value, so probably it is the XOR key that you are looking for.

Thus, to perform the search operation for XOR instructions, jump to the **Assembly view** (**IDA View**). From there, go to **Search → Text** and type ' **xor *** ' (of course, you can try a real regular expression) and you will receive an output like the following one:

```
.text:0058FF7A    sub_58FDE0    xor    eax, 0E462D21Ch
.text:00590167    sub_590130    xor    eax, 0E462D21Ch
.text:005919E5    sub_59143C    xor    eax, 0E462D21Ch
.text:00591F71    sub_591D34    xor    eax, 0E462D21Ch
.text:00598FCE    sub_598E8C    xor    eax, 0E462D21Ch
.text:005950A6    sub_595088    xor    eax, [ecx+ebp*4]
```

[Figure 19]: searching for XOR key

The same immediate value has been used over five places, so it is probably the XOR key: **0xE462D21C.**

So far we identified that the code is using **DLL/API hash resolving**, but there are other pending questions related to this specific point: what's the algorithm used over this hashing operations? To find a possible answer, it is time to focus our analysis on any location that is involved with the XOR key that we showed above. For example, readers could try to examine the second one at **0x00590167**, as shown below:

```
int __usercall sub_590130@<eax>(int a1@<ecx>, _BYTE *a2@<esi>)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  v7[1] = a2;
  v7[0] = a2;
  result = *(a1 + 0x3C);
  if ( !result )
  {
    v4 = sub_590180(a1);
    sub_58E06C(v4, v7);
    v5 = v7[0];
    v6 = sub_58E8A8(v7[0], 0x7FFFFFFF);
    *(a1 + 0x3C) = sub_594FFC(v5, v6) ^ 0xE462D21C;
    sub_58DFA4(v7);
    return *(a1 + 0x3C);
  }
  return result;
}
```

[Figure 20]: Function involved with the found XOR key

Once again, there is a series of routines being called, but **sub_594FFC routine** seems to be more interesting because it is used and xor'ed against the XOR key. Of course, if readers have a spare time, so it would be amazing to analyze the other routines too. Examining the **sub_594FFC routine**, we have:

```
1 int __fastcall sub_594FFC(_BYTE *a1, int a2)
2 {
3   char *i; // edi
4   char v6[1036]; // [esp+8h] [ebp-40Ch] BYREF
5
6   for ( i = dword_59D248; !word_59D2F0; sub_5950B8(i) )
7   {
8     if ( i != 0x774C488D )
9       return sub_595088(i, a1, a2);
10    if ( !sub_593064(0x39731522, 0x45B68B68) )
11      break;
12    i = sub_59361C();
13    dword_59D248 = i;
14  }
15  sub_5950B8(v6);
16  return sub_595088(v6, a1, a2);
17 }
```

[Figure 21]: sub_594FFC routine

Apparently there is nothing here, but not so fast. Pay attention to the **for loop** on line 6, which is using two **DWORD's** and one routine **sub_5950B8( ).** The first **DWORD** (**dword_59D248**) is referred by two pieces of code and one of them is responsible for writing such double word. Readers can find both addresses by using **CTRL+X hotkey**. Moving into **sub_5950B8( )** we have:

```
int __thiscall sub_5950B8(char *this)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  v1 = 0;
  v2 = _mm_loadu_si128(&xmmword_59BF30);
  v3 = _mm_loadu_si128(&xmmword_59BF40);
  v4 = _mm_loadu_si128(&xmmword_59BF50);
  v5 = _mm_loadu_si128(&xmmword_59BF60);
  do
  {
    v6 = v3;
    LOBYTE(result) = 0;
    do
    {
      LOBYTE(result) = result + 1;
      v8 = _mm_cmpeq_epi32(_mm_and_si128(v4, v6), 0i64);
      v9 = _mm_srli_epi32(v6, 1u);
      v10 = _mm_xor_si128(v8, -1i64);
      v6 = _mm_or_si128(
             _mm_and_si128(_mm_xor_si128(v9, v5), v10),
             _mm_andnot_si128(v10, v9));
    }
    while ( result < 8u );
    *&this[4 * v1] = v6;
    v1 += 4;
    v3 = _mm_add_epi32(v3, v2);
  }
  while ( v1 < 0x100 );
  return result;
}
```

[Figure 22]: sub_5950B8 routine

At first, results do not seem to be so relevant, but we found constants being used and related to multiple operations. Actually, it is really a great news because constants are valuable to identify cryptography algorithms. Therefore, we need to examine what are these constants:

```
.data:0059BF30 xmmword_59BF30  xmmword 4000000040000000400000004h
.data:0059BF30                                          ; DATA XREF: sub_5950B8+2↑r
.data:0059BF40 xmmword_59BF40  xmmword 3000000020000000100000000h
.data:0059BF40                                          ; DATA XREF: sub_5950B8+E↑r
.data:0059BF50 xmmword_59BF50  xmmword 1000000010000000100000001h
.data:0059BF50                                          ; DATA XREF: sub_5950B8+16↑r
.data:0059BF60 xmmword_59BF60  xmmword 0EDB88320EDB88320EDB88320EDB88320h
.data:0059BF60                                          ; DATA XREF: sub_5950B8+1E↑r
.data:0059BF70 byte_59BF70     db 4Eh                   ; DATA XREF: sub_595B3C+69↑r
```

[Figure 23]: Crypto constants

Initially there is nothing really useful again, but pay attention to the last constant:

- 0EDB88320EDB88320EDB88320EDB88320

Let me show the same line again, but this time using colors:

- **0EDB88320 EDB88320 EDB88320 EDB88320**

The **same 4-bytes hexadecimal is repeated four times**. Searching for this hexadecimal constant on the Internet, readers are going to quickly confirm that it is related to **CRC32 algorithm** (https://en.wikipedia.org/wiki/Cyclic_redundancy_check):

CRC-32	ISO 3309 (HDLC), ANSI X3.66 (ADCCP), FIPS PUB 71, FED-STD-1003, ITU-T V.42, ISO/IEC/IEEE 802-3 (Ethernet), SATA, MPEG-2, PKZIP, Gzip, Bzip2, POSIX cksum,[53] PNG,[54] ZMODEM, many others	0x04C11DB7	$\boxed{\text{0xEDB88320}}$	0xDB710641	0x82608EDB[15]
		$x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^{8}+x^{7}+x^{5}+x^{4}+x^{2}+x+1$			

[Figure 24]: CRC32 constants (from Wikipedia)

Finally, we found the algorithm being used by the sample to generate all hash values that are used by the malware to encode its API and DLL names. Therefore, until now, we could confirm the following facts:

- The malware is using **API / DLL hashing**.
- There is a XOR key being used in the hashing procedure: **0xE462D21C**
- The algorithm being used to calculate the hash is **CRC32**.

Certainly, they won't be the only obstacles that we will have to manage in our analysis.

The next point is to understand the strange objective of "**int 3**" (**__debugbreak( )**), which is being used over multiple places in the malware's code. First, we have to pick up an example of pseudo code to analyze and try to have a better comprehension about what is really happening:

```
1  int sub_593628()
2  {
3    int result; // eax
4    int (__cdecl *v1)(int, _DWORD, _DWORD, _DWORD, _DWORD, _DWORD); // ebx
5
6    if ( dword_59D228 == 0xA33C83E5 )
7    {
8      v1 = sub_593064(0x60A28C5C, 0x1C6EF387);
9      dword_59D22C = sub_593064(0x60A28C5C, 0x5E0AFAA3);
10     if ( dword_59D228 == 0xA33C83E5 )
11       dword_59D228 = v1(2, 0, 0, 0, 0, 0);
12   }
13   result = sub_593064(0x60A28C5C, 0x45B68B68);
14   if ( !result )
15     return 0;
16   __debugbreak();
17   __debugbreak();
18   return result;
19 }
```

[Figure 25]: Sub_593628 routine: pseudo code

Although we have not discussed any technique to resolve hashes (it will be done in next pages), we can do a fast analysis about what is occurring in this figure:

- **line 6:** apparently a status variable is controlling the execution flow. Instructions from **line 8 to line 11** will be only executed whether the condition is **True**. However, there is a detail: if readers check the **dword_59D228**'s value in the **.data section**, it contains initially exactly the sane value used in the comparison: **0xA33C83E5**.
- **line 8 and 9:** the **hash resolving routine (sub_593064)** is being called. **On line 8**, a function pointer is returned and stored into **v1**. In addition, the result of calling the hash resolving routine on **line 9** is stored into **dword_59D22C** and not used at this time.
- **line 10:** the same status variable and value from line 6 are used one more time.
- **line 11**: the function pointer is used to invoke the function with 6 arguments.
- **line 13**: once again, the routine responsible for handling API hash resolving **(sub_593064)** is called for the third time.
- **lines 16 and 17:** the **__debugbreak( )** function is called.

Obviously, there is an evident anti-analysis trick here that was introduced by an obfuscator, but readers should pay attention to one detail: **on line 8**, the resolved API's address is returned, stored into **v1** and then called on **line 11**, when its result is stored into exactly the same variable **dword_59D228**. However, the returned function pointer from call to **sub_593064** is loaded into result and, apparently, it is not used for anything else.

Additionally, the test using "**if instruction**" on **line 14** is weird because whether the calling for **sub_593064** is successful (and we can assume it is), so **lines 16** and **17** wouldn't be executed. Finally, both **_debugbreak( )** calls, which are "**int 3**", are called and the **resolved API (the function pointer)** is returned to the caller function, but it is not directly called as we saw with **v1** on **line 11**.

Observing the Assembly of **sub_593628** routine certainly will help readers to start to understand what could be happening in this strange piece of code:

```
text:00593628 ; int sub_593628()
text:00593628 sub_593628      proc near               ; CODE XREF: sub_59361C+4↑j
text:00593628                                         ; sub_593958+3D↓p ...
text:00593628                 push    esi
text:00593629                 push    edi
text:0059362A                 push    ebx
text:0059362B                 mov     esi, offset dword_59D228
text:00593630                 mov     edi, ecx
text:00593632                 cmp     dword ptr [esi], 0A33C83E5h
text:00593638                 jz      short loc_59365C
text:0059363A
text:0059363A loc_59363A:                             ; CODE XREF: sub_593628+5F↓j
text:0059363A                                         ; sub_593628+6E↓j
text:0059363A                 push    45B68B68h
text:0059363F                 push    60A28C5Ch
text:00593644                 call    sub_593064
text:00593649                 test    eax, eax
text:0059364B                 jz      short loc_593656
text:0059364D                 push    edi
text:0059364E                 push    8
text:00593650                 push    dword ptr [esi]
text:00593652                 int     3               ; Trap to Debugger
text:00593653                 int     3               ; Trap to Debugger
text:00593654                 jmp     short loc_593658
text:00593656 ; ---------------------------------------------------------------------------
text:00593656
text:00593656 loc_593656:                             ; CODE XREF: sub_593628+23↑j
text:00593656                 xor     eax, eax
text:00593658
text:00593658 loc_593658:                             ; CODE XREF: sub_593628+2C↑j
text:00593658                 pop     ebx
text:00593659                 pop     edi
text:0059365A                 pop     esi
text:0059365B                 retn
text:0059365C ; ---------------------------------------------------------------------------
text:0059365C
text:0059365C loc_59365C:                             ; CODE XREF: sub_593628+10↑j
text:0059365C                 push    1C6EF387h
text:00593661                 push    60A28C5Ch
text:00593666                 call    sub_593064
text:0059366B                 mov     ebx, eax
text:0059366D                 push    5E0AFAA3h
text:00593672                 push    60A28C5Ch
text:00593677                 call    sub_593064
text:0059367C                 mov     dword_59D22C, eax
text:00593681                 cmp     dword ptr [esi], 0A33C83E5h
text:00593687                 jnz     short loc_59363A
text:00593689                 xor     eax, eax
text:0059368B                 push    eax
text:0059368C                 push    eax
text:0059368D                 push    eax
text:0059368E                 push    eax
text:0059368F                 push    eax
text:00593690                 push    2
text:00593692                 call    ebx
text:00593694                 mov     [esi], eax
text:00593696                 jmp     short loc_59363A
text:00593696 sub_593628      endp ; sp-analysis failed
text:00593696
text:00593698
```

[Figure 26]: sub_593628 routine: Assembly code

Clearly both "**int 3**" instructions shouldn't be here. This is a well-known anti-analysis trick used to defeat dynamic and static analysis, and I have seen it (or a variation) over other samples. During a possible dynamic analysis, the "**int 3**" instruction will be interpreted as an exception (**EXCEPTION_BREAKPOINT**), forcing us to skip it or even passing the exception to the designed exception handlers registered by system, and it will cause delays to analysts to examine the sample. Nonetheless, it is not our concern here.

About the aspect of the static analysis, both calls for API name resolution from **line 09** and **13** from **Figure 25 (pseudo code)** are not being really used, apparently. Anyway, we know that after a function's call, its result is returned (pushed) into **EAX register**. Additionally, analyzing the **Figure 26**, readers can see a kind of pattern because soon after both "**int 3**" instructions, where there is a jump to a near location, which do a quick restoration of non-volatile registers and return. This action action seems cause a severe problem in the disassembler because the block under **loc_59365C** is only executed if the instruction at **0x00593638** is true, and inversion in the execution flow cause issues during the disassembling process. Regardless of the commented side effect, remember that the pointer to the resolved function stored in **eax**. Therefore, what is the trick?

In any other place around the code the malware is registering an exception handler to manage an exception type as **EXCEPTION_BREAKPOINT**, which is related to **_EXCEPTION_RECORD** structure, and this handler likely will be executing the function on the top of stack, which is exactly the same **eax's value** that was pushed to stack previously. At the end, the resolved API will be called and executed. Therefore, we can assume one **int 3 instruction** together the exception handler produces an effect as **call eax** instruction.

To be able to find the exception-related code, it will be easier to handle the DLL and API resolution first and, afterwards, returning and searching for the code. Of course, we could use just the Assembly code to track such code if we wanted to, and it would take a bit longer.

About API hash resolving, we hold the following information so far:

- algorithm: **crc32**
- xor key: **0xE462D21C**
- function associated with hash resolving is **sub_59306C**

In the other hand, we have the following pending tasks:

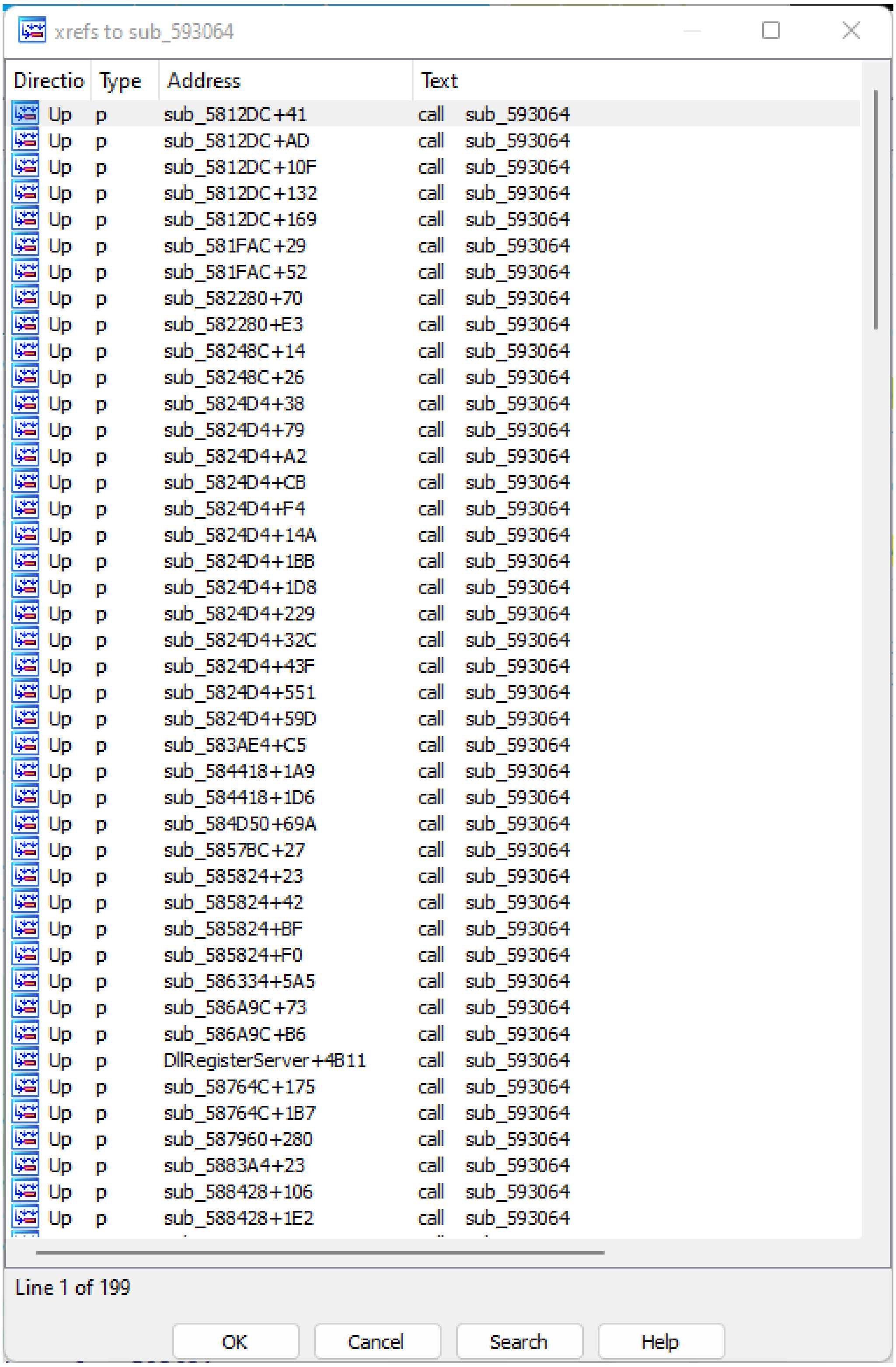
- finding where malware has registered the exception handler, and analyzing it to confirm our assumptions.
- improving the pseudo code and, at the same time, managing a better solution to handle with both **int 3** instructions.

To handle hashing resolving, there are several possibilities and all of them are excellent solutions:

- **Shellcode Hashes from flare-ida (Mandiant):** https://github.com/mandiant/flare-ida/blob/master/plugins/shellcode_hashes_search_plugin.py
- **Apihashes IDA Pro plugin (from Igor Kuznetsov):** https://github.com/KasperskyLab/Apihashes
- **HashDb IDA Pro plugin (OALabs):** https://github.com/OALabs/hashdb-ida

If readers pay attention to **Figure 25**, the routine involved with hashes is **sub_593064**, which accepts two arguments, but we also found previously the **sub_59306C** routine that accepts five arguments and also seems to be related to hash resolution.

Examining the cross-references (**X** hotkey) to **sub_593064** routine, we found that there is a lot of references to it:



[Figure 27]: sub_593064 routine: resolving hash

Wow! This routine is called 199 times! Now we can understand what is happening: this routine (**sub_593064**) is another hash resolving routine, besides the **sub_59306C routine** (that one we mentioned previously, which accepts five arguments), and both use the same internal routines.

Therefore, we can adopt the following nomenclature to make things easier for us:

- **sub_593064: ab_hash_resolving**
- **sub_59306C: ab_hash_resolving_internal**

Most certainly, readers already used **Shellcode Hashes** from **flare-ida project**, but I am going to quickly explain how to set up it and use it in our case:

1. Clone the **flare-ida** repository: https://github.com/mandiant/flare-ida. In my case, I did it under *C:\github* directory.

2. Copy the plugin file (*C:\github\flare-ida\plugins\shellcode_hashes_search_plugin.py*) to IDA Pro's plugin directory, which in my case is: *C:\Program Files\IDA Pro 8.2\plugins*

3. Copy the entire **flare folder** (*C:\github\flare-ida\python\flare)* to **IDA Pro's Python folder** (*C:\Program Files\IDA Pro 8.2\python\3).*

4. Now we have to generate the hash database, so it is a matter of choice. Readers must choose which DLLs will be used as source to generate the hash database. You can create a directory with all chosen DLLs and generate a partial database, or include all DLLs from *C:\Windows\System32* directory to generate a big database and, of course, much more complete than our partial one. I have generated both databases:

    a. **ab_hashes_partial.db** (60.960 KB – generated in a couple of minutes)
    b. **ab_hashes_full.db** (596.688 KB – takes about 30 minutes to get finished)

```
C:\github\flare-ida\shellcode_hashes>python make_sc_hash_db.py ab_hashes_partial.db partial_dll_folder
Processing file partial_dll_folder\advapi32.dll
Processed 870 export symbols in 1.75 seconds: partial_dll_folder\advapi32.dll
Processing file partial_dll_folder\advpack.dll
Processed 84 export symbols in 0.36 seconds: partial_dll_folder\advpack.dll
Processing file partial_dll_folder\comctl32.dll
Processed 119 export symbols in 0.18 seconds: partial_dll_folder\comctl32.dll
Processing file partial_dll_folder\comdlg32.dll
Processed 28 export symbols in 0.06 seconds: partial_dll_folder\comdlg32.dll
Processing file partial_dll_folder\crypt32.dll
Processed 295 export symbols in 0.85 seconds: partial_dll_folder\crypt32.dll
Processing file partial_dll_folder\dnsapi.dll
Processed 315 export symbols in 1.18 seconds: partial_dll_folder\dnsapi.dll
Processing file partial_dll_folder\gdi32.dll
```

[Figure 28]: Shellcode Hashes: generating the database

5. If readers don't have a ready list of interesting DLLs, **Hexacorn** (https://twitter.com/Hexacorn) recently authored an article and his list is good enough for handling most of the cases. The Adam's article is available on: https://www.hexacorn.com/blog/2022/12/03/using-make_sc_hash_db-py-to-create-api-hashing-dbs/

6. Check whether the installed and default Python on Windows matches exactly with the version being used by IDA Pro:

    a. on Windows: **python -V**

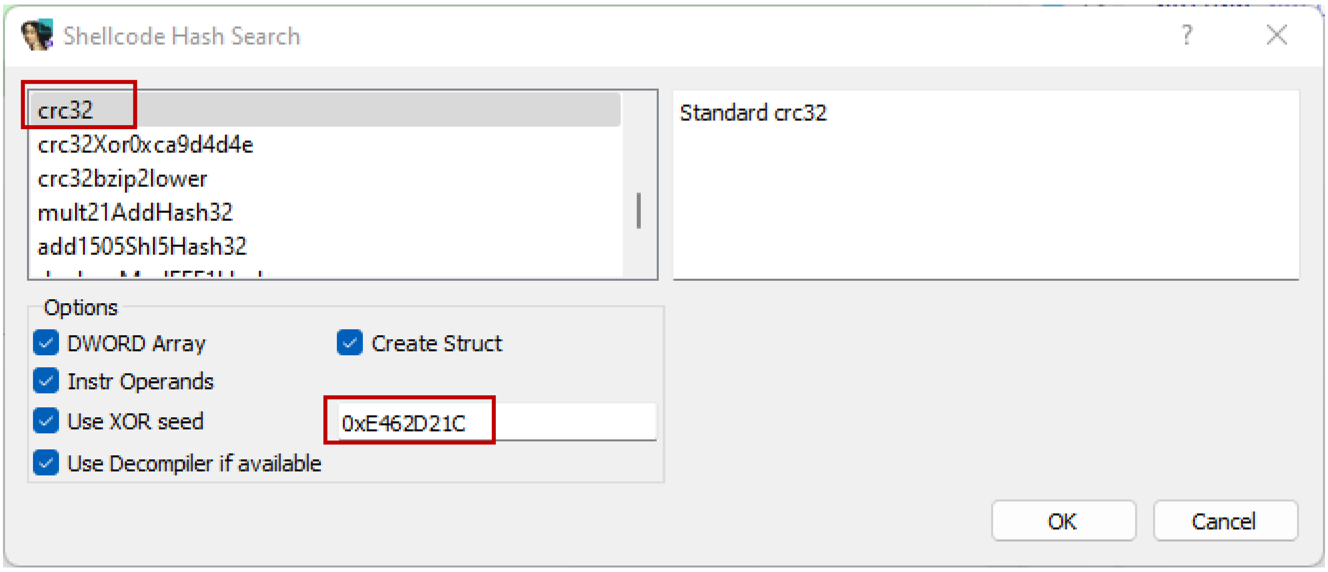
b. go to *"C:\Program Files\IDA Pro 8.2"* and run the **idapyswitch.exe** executable. Be sure of picking up exactly the same version that it is the default one on Windows.

c. Open the IDA Pro, go to the **Python command line** and type the following command, which should return the default Python version used by IDA Pro, and the same from Windows:

   i. import sys

   ii. sys.version

7. On IDA Pro, go to **Edit → Plugins → Shellcode Hashes** and pick up the generated database. A form will be shown, and readers must pick up the algorithm (**CRC32,** as we found) and enter the **XOR key** (**0xE462D21C)**, as we also discovered:



[Figure 29]: Shellcode Hashes plugin

After running the plugin, we have the following result:

```
int sub_593628()
{
  int result; // eax
  int (__cdecl *v1)(int, _DWORD, _DWORD, _DWORD, _DWORD, _DWORD); // ebx

  if ( dword_59D228 == 0xA33C83E5 )
  {
    v1 = sub_593064(0x60A28C5C, 0x1C6EF387);     // ntoskrnl.exe!b'RtlCreateHeap'
    dword_59D22C = sub_593064(0x60A28C5C, 0x5E0AFAA3);// ntoskrnl.exe!b'RtlDestroyHeap'
    if ( dword_59D228 == 0xA33C83E5 )
      dword_59D228 = v1(2, 0, 0, 0, 0, 0);
  }
  result = sub_593064(0x60A28C5C, 0x45B68B68);  // ntoskrnl.exe!b'RtlAllocateHeap'
  if ( !result )
    return 0;
  __debugbreak();
  __debugbreak();
  return result;
}
```

[Figure 30]: Shellcode Hashes: applied names

The resulting is great and, as readers can see, all DLL and API names found by **Shellcode Hashes plugin** are applied as comment. Readers shouldn't forget that, to get to this point, we worked and analyzed the code to find:

- **hash algorithm**
- **XOR key**
- **hash resolving routine**

Without having the hash algorithm and XOR key in our hands, so it would be impossible to use the plugin to get the right result. In the other hand, the **plugin** also applies comments on Assembly code:

```
.text:00593628 ; int sub_593628()
.text:00593628 sub_593628      proc near               ; CODE XREF: sub_59361C+4↑j
.text:00593628                                         ; sub_593958+3D↓p ...
.text:00593628                 push    esi
.text:00593629                 push    edi
.text:0059362A                 push    ebx
.text:0059362B                 mov     esi, offset dword_59D228
.text:00593630                 mov     edi, ecx
.text:00593632                 cmp     dword ptr [esi], 0A33C83E5h
.text:00593638                 jz      short loc_59365C
.text:0059363A
.text:0059363A loc_59363A:                             ; CODE XREF: sub_593628+5F↓j
.text:0059363A                                         ; sub_593628+6E↓j
.text:0059363A                 push    45B68B68h       ; ntoskrnl.exe!b'RtlAllocateHeap'
.text:0059363F                 push    60A28C5Ch
.text:00593644                 call    sub_593064
.text:00593649                 test    eax, eax
.text:0059364B                 jz      short loc_593656
.text:0059364D                 push    edi
.text:0059364E                 push    8
.text:00593650                 push    dword ptr [esi]
.text:00593652                 int     3               ; Trap to Debugger
.text:00593653                 int     3               ; Trap to Debugger
.text:00593654                 jmp     short loc_593658
.text:00593656 ; ---------------------------------------------------------------------------
.text:00593656
.text:00593656 loc_593656:                             ; CODE XREF: sub_593628+23↑j
.text:00593656                 xor     eax, eax
.text:00593658
.text:00593658 loc_593658:                             ; CODE XREF: sub_593628+2C↑j
.text:00593658                 pop     ebx
.text:00593659                 pop     edi
.text:0059365A                 pop     esi
.text:0059365B                 retn
.text:0059365C ; ---------------------------------------------------------------------------
.text:0059365C
.text:0059365C loc_59365C:                             ; CODE XREF: sub_593628+10↑j
.text:0059365C                 push    1C6EF387h       ; ntoskrnl.exe!b'RtlCreateHeap'
.text:00593661                 push    60A28C5Ch
.text:00593666                 call    sub_593064
.text:0059366B                 mov     ebx, eax
.text:0059366D                 push    5E0AFAA3h       ; ntoskrnl.exe!b'RtlDestroyHeap'
.text:00593672                 push    60A28C5Ch
.text:00593677                 call    sub_593064
```

[Figure 31]: Shellcode Hashes: applied names on Assembly code

There's a small catch here: if readers will apply the full database hash, which was generated using *C:\Windows\System* directory, eventually the name of DLL will be different from shown above because such a function might be exported by more than one DLL.

Personally, I like **Shellcode Hashes plugin** because it is easy to work with it since you have done the analysis correctly. Furthermore, it offers us good points:

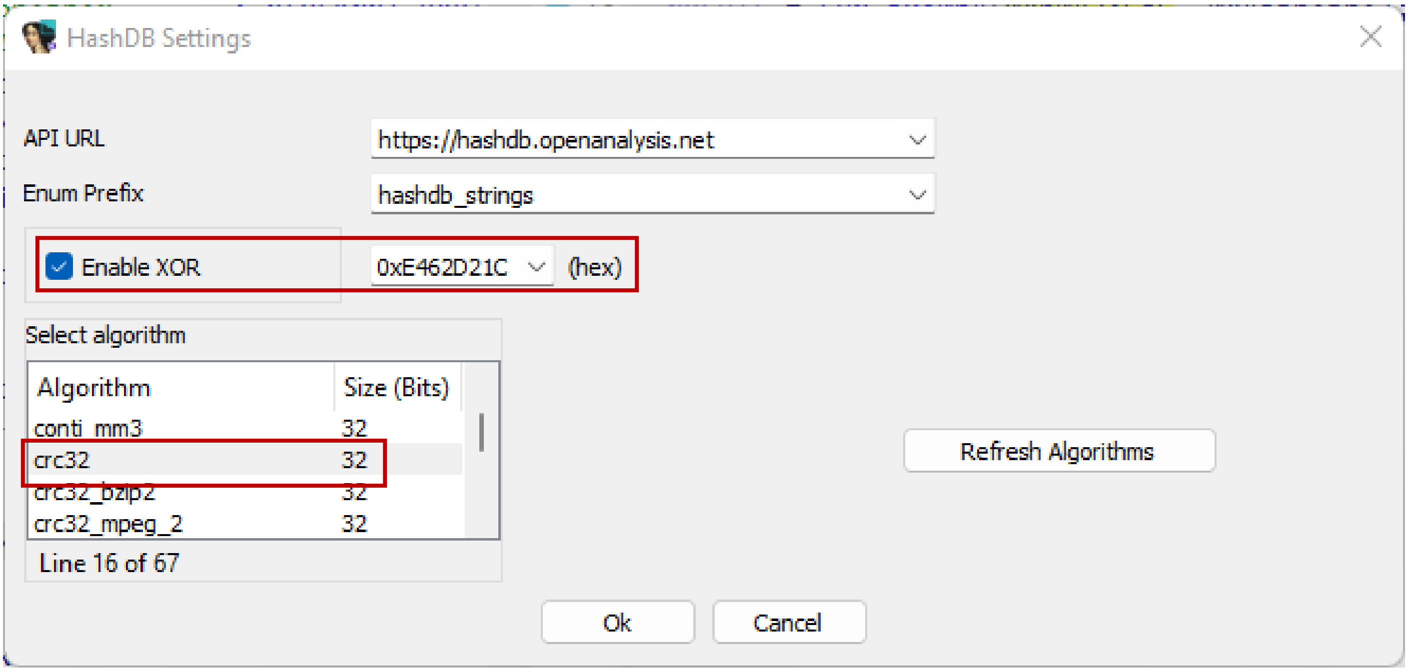
- It's excellent to be used with **analysis of shellcodes** (shellcode analysis will be a key topic covered in next versions of this series – maybe in **MAS 9** or **MAS 10** – stay tuned!)
- It makes comments on **pseudo and Assembly code**.
- It has a quite extensive list of **available hash algorithms**, although Mandiant haven't updated since then, unfortunately.
- **Keep information private** without transmitting any information to other place on Internet.

The good plugin offered by **Igor Kuznetsov** (**Apihashes:** https://github.com/KasperskyLab/Apihashes) has a similar principle to **Shellcode Hashes,** but I will not show it here**.** Readers can make tests with it and, certainly, will get the same result obtained by other plugins.

The other plugin is **HashDB plugin** from **OALabs**, which can be cloned by executing **git clone https://github.com/OALabs/hashdb-ida.** To get it working, copy **hashdb.py** to IDA's plugin directory (*C:\Program Files\IDA Pro 8.2\plugins*). **Attention:** as HashDB performs lookup on OALabs server, so you should remember to keep Internet access in your environment.

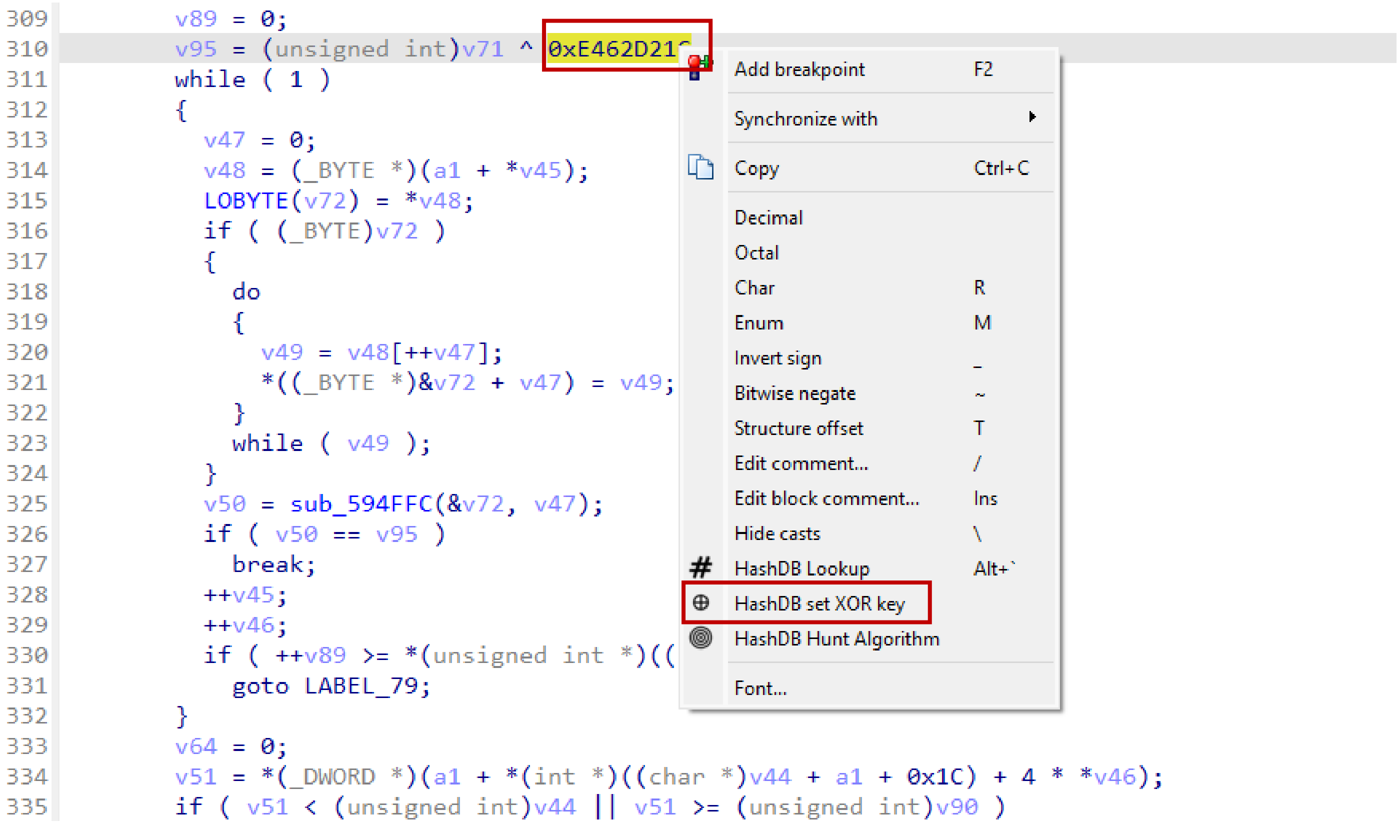
There is more than one way to proceed. The steps to get hash resolution are:

a. Go to **Edit → Plugins → HashDB**
b. Pick the algorithm up: **crc32**
c. Enter the XOR key: **0xE462D21C**
d. Click on OK button.



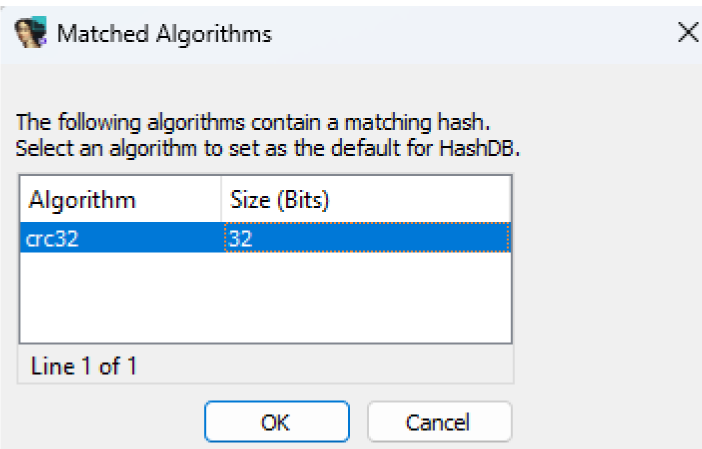
[Figure 32]: HashDB Settings

We are ready to go, but let me explain an alternative method to do this setup and with an additional advantage. Once we already found places over the code with the XOR key (for example, **sub_593064 → sub_59306C → sub_59143C**), we can set up the plugin by **right clicking the XOR key** and choosing **HashDB set XOR key** option:



[Figure 33]: HashDB: setting XOR key

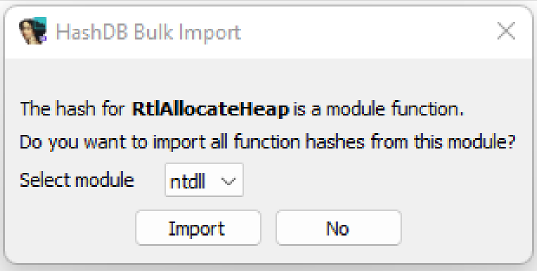
After getting this step done, right click on the hash and choose **HashDB Hunt Algorithm** (check the image above). The advantage of this option is that HashDB will try to guess the algorithm being used automatically. In other words, if you are lucky, there will not be necessary to analyze the code before resolving hashes because the plugin will be able to detect the algorithm for you (CRC32, in this case). Don't forget to mark the algorithm once the **Matched Algorithms form** is presented!



[Figure 34]: HashDB: searching and select the algorithm

Finally, we are ready to right-click on any hash, choose **HashDB Lookup** and click on **Import**:

```
int sub_593628()
{
  int result; // eax
  int (__cdecl *v1)(int, _DWORD, _DWORD, _DWORD, _DWORD, _DWORD); // ebx

  if ( dword_59D228 == 0xA33C83E5 )
  {
    v1 = sub_593064(0x60A28C5C, 0x1C6EF387);     // ntoskrnl.exe!b'RtlCreateHeap'
    dword_59D22C = sub_593064(0x60A28C5C, 0x5E0AFAA3);// ntoskrnl.exe!b'RtlDestroyHeap'
    if ( dword_59D228 == 0xA33C83E5 )
      dword_59D228 = v1(2, 0, 0, 0, 0, 0);
  }
  result = sub_593064(0x60A28C5C, 0x45B68B68);  // ntoskrnl.exe!b'RtlAllocateHeap'
  if ( !result )
    return 0;
  __debugbreak();
  __debugbreak();
  return result;
}
```



[Figure 35]: HashDB: look up for hash

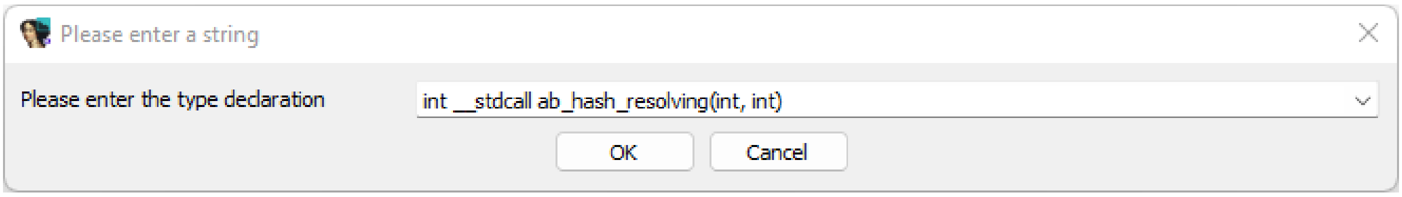
Imports might take few seconds and freeze the IDA Pro, but that is a normal behavior and let it to finish its job. **HashDB**, like **Shellcode Hashes** from Mandiant, creates an enumeration containing all hashed functions, as shown below:

```
FFFFFFFF ; enum hashdb_strings_crc32, mappedto_74, width 4 bytes
FFFFFFFF NtReplyWaitReceivePort_0  = 45B30h
FFFFFFFF RtlCaptureStackContext_0  = 5E8BFEh
FFFFFFFF RtlAreBitsClear_0  = 613663h
FFFFFFFF RtlDowncaseUnicodeChar_0  = 6905C1h
FFFFFFFF _alldiv_0          = 8EFD2Fh
FFFFFFFF NtWow64GetCurrentProcessorNumberEx_0  = 909C02h
FFFFFFFF NtWriteVirtualMemory_0  = 0E54B25h
FFFFFFFF RtlValidateUnicodeString_0  = 0FB2702h
FFFFFFFF iswdigit_0         = 1109040h
FFFFFFFF NtOpenResourceManager_0  = 150E492h
FFFFFFFF _ltoa_s_0          = 1883A4Dh
FFFFFFFF ZwWow64CsrGetProcessId_0  = 1C3DA68h
FFFFFFFF RtlUserThreadStart_0  = 1D67F7Dh
FFFFFFFF RtlCreateProcessParameters_0  = 2145DE4h
FFFFFFFF ZwQueryBootEntryOrder_0  = 21BCEC9h
FFFFFFFF RtlGenerate8dot3Name_0  = 297A893h
FFFFFFFF RtlAnsiCharToUnicodeChar_0  = 297F5F8h
FFFFFFFF RtlReportSqmEscalation_0  = 2CA88D2h
FFFFFFFF ZwCompareTokens_0  = 2DAD4B2h
FFFFFFFF RtlAbortRXact 0  = 2E7C61Eh
```

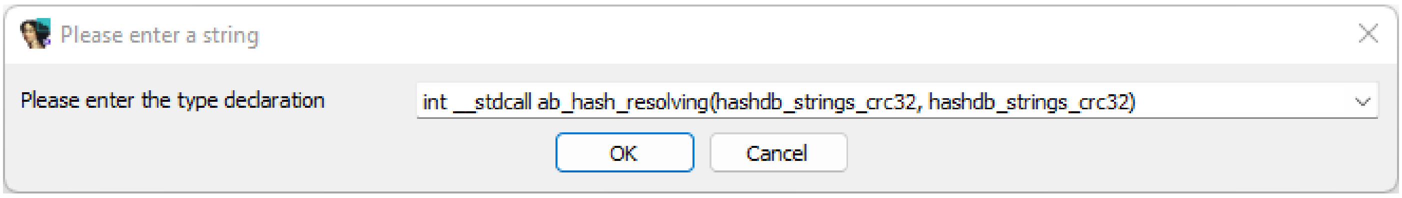
[Figure 36]: HashDB creates an enumeration for APIs

We have the enumeration created by HashDB (**hashdb_strings_crc32**), which will be useful for us.

Obviously, we have to do it for both hashes used in the routine. Now we can edit the routine signature (**sub_593064 – renamed to ab_hash_resolving**) and change its two argument's type to **hashdb_strings_crc32**, as shown below:



[Figure 37]: Before changing the sub_593064 routine's type



[Figure 38]: After changing the sub_593064 routine's type

Now it's enough to press **F5** and the result will be much better:

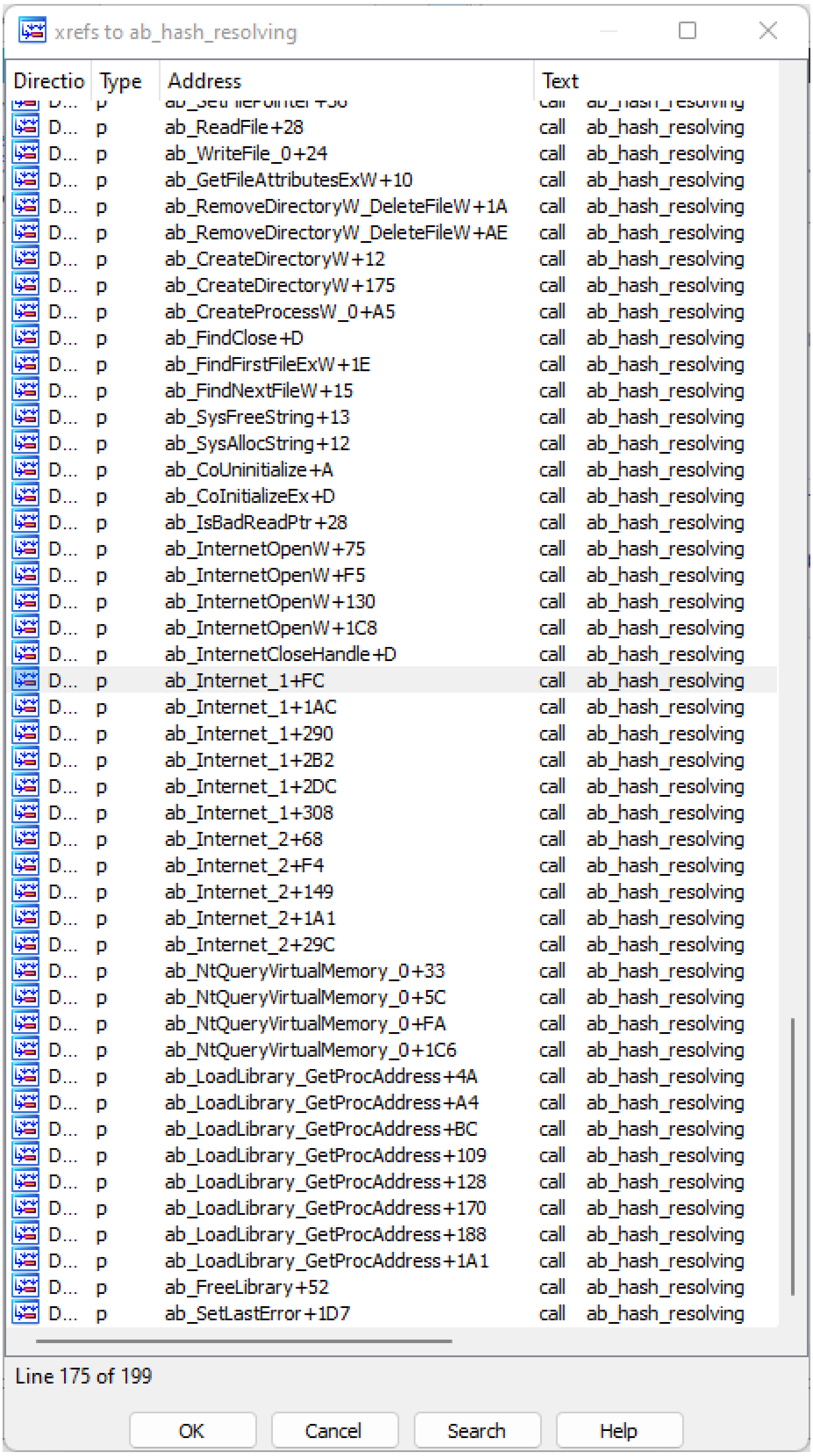
```
int sub_593628()
{
  int RtlAllocateHeap_0; // eax
  int (__cdecl *RtlCreateHeap_0)(int, _DWORD, _DWORD, _DWORD, _DWORD, _DWORD); // ebx

  if ( dword_59D228 == 0xA33C83E5 )
  {
    RtlCreateHeap_0 = ab_hash_resolving_0(ntdll_dll, RtlCreateHeap_0);// ntoskrnl.exe!b'RtlCreateHeap'
    dword_59D22C = ab_hash_resolving_0(ntdll_dll, RtlDestroyHeap_0);// ntoskrnl.exe!b'RtlDestroyHeap'
    if ( dword_59D228 == 0xA33C83E5 )
      dword_59D228 = RtlCreateHeap_0(2, 0, 0, 0, 0, 0);
  }
  RtlAllocateHeap_0 = ab_hash_resolving_0(ntdll_dll, RtlAllocateHeap_0);// ntoskrnl.exe!b'RtlAllocateHeap'
  if ( !RtlAllocateHeap_0 )
    return 0;
  __debugbreak();
  __debugbreak();
  return RtlAllocateHeap_0;
}
```

[Figure 39]: After updating the pseudo code

There is a note here: I renamed **v1** to **RtlCreateHeap_0** and the **result variable** to **RltAllocateHeap_0**. Actually, this code needs a supplemental change, but I am going to do it in the next pages after we analyze the piece of code related to the exception handler.

Additionally, I always like to rename the subroutine's name (**sub_593628**, in this case) to one of the API's name within routine (maybe the most important) or a name representing the entire goal of that routine because a better name will provide us with a guideline to analyze other parts of code later.

From this point onward, the suggestion is to repeat the same procedure: **F5** + **HashDB Lookup** (for APIs coming from different DLLs) + **renaming** for all **199 routines** that are calling **hashdb_strings_crc32**. Yes, it takes a meaningful amount of time, but the final result provides us a much better indication and is going to help us to find what we are looking for:



[Figure 40]: All API hashes resolved, and wrapper routines renamed

As I mentioned in the last page, I used HashDB plugin to resolve hashes and, in this case, there is something really interesting: eventual places where I couldn’t get a result through HashDB, I already had answers from **Shellcode Hashes plugin.** Therefore, it has worked as a double-checking.

Now we are able to return to one of pending problems. As I mentioned, the malware actor may have registered an exception handler which manages exceptions of type **EXCEPTION_BREAKPOINT,** whose type is related to an **_EXCEPTION_RECORD structure**. The expected goal is to force an exception, transfer the execution flow to the registered exception handler and, at end, execute the address stored on the top of the stack, which is exactly the same content of **eax** that holds the resolved API's address and has been pushed onto the stack.

A good starting point to search for this exception handler is at beginning of the DLL and, more specifically, one the first lines of **DllRegisterServer function**. as shown below:

```
HRESULT __stdcall DllRegisterServer()
{
  int *v0; // esi
  unsigned int i; // eax
  int v3; // eax
  struct _EXCEPTION_REGISTRATION_RECORD *ExceptionList; // [esp+24h] [ebp-8h] BYREF
  int *v5; // [esp+28h] [ebp-4h]

  v5 = sub_5821FC;
  ExceptionList = NtCurrentTeb()->NtTib.ExceptionList;
  for ( i = 0; i < 0x13512; ++i )
  {
    __debugbreak();
    __debugbreak();
    __debugbreak();
    __debugbreak();
  }
  v5 = v0;
  ExceptionList = v0;
  sub_5930E8(&ExceptionList, v0, 0);
  if ( !byte_59D028 )
  {
    ab_OutputDebugStringW_CreateProcess(v0);
    dword_59D1C4(0);
LABEL_8:
    sub_593144(&ExceptionList, v0);
    return 1;
  }
  if ( byte_59D268 )
    goto LABEL_8;
  byte_59D268 = 1;                              // kernel32.dll!b'CreateThread'
  v0 = ab_hash_resolving(kernel32_dll, CreateThread_0);
  if ( v0 )
  {
    v3 = sub_591D30(0xA731522);
    if ( (v0)(0, 0, ab_VirtualFree_ExitThread, v3, 0, 0) )
      goto LABEL_8;
  }
  sub_593144(&ExceptionList, v0);
  return 0;
}
```

[Figure 41]: DllRegisterServer( ) function

Indeed, there is a list of clues that we are close to our target because we can see a declaration of an instance of **_EXCEPTION_REGISTRATION_RECORD structure** (**ExceptionList**) **on line 6**. Also, the code is retrieving the current **ExceptionList** from the **TEB (Thread Environment Block**) **on line 10** and storing into the declared **ExceptionList variable**. Readers remember that the first members of **_TEB structure** is given by the following:

```
00000000 _TEB            struc ; (sizeof=0xFE8, align=0x4, copyof_2)
00000000 NtTib           _NT_TIB ?
0000001C EnvironmentPointer dd ?                  ; offset
00000020 ClientId        _CLIENT_ID ?
00000028 ActiveRpcHandle dd ?                     ; offset
0000002C ThreadLocalStoragePointer dd ?           ; offset
00000030 ProcessEnvironmentBlock dd ?             ; offset
00000034 LastErrorValue  dd ?
00000038 CountOfOwnedCriticalSections dd ?
0000003C CsrClientThread dd ?                     ; offset
00000040 Win32ThreadInfo dd ?                     ; offset
00000044 User32Reserved  dd 26 dup(?)
000000AC UserReserved    dd 5 dup(?)
000000C0 WOW32Reserved   dd ?                     ; offset
000000C4 CurrentLocale   dd ?
000000C8 FpSoftwareStatusRegister dd ?
000000CC SystemReserved1 dd 54 dup(?)             ; offset
000001A4 ExceptionCode   dd ?
000001A8 ActivationContextStackPointer dd ?       ; offset
```

[Figure 42]: _TEB structure: first fields

Readers can get the same structure from IDA Pro: **SHIFT+F9 (Structures View) → Insert → Add standard structure → CTRL+F → _TEB**, just in case the **_TEB** is not already added.

The first argument of **_TEB** is a member of type **_NT_TIB (TIB: Thread Information Block)**, which has the following structure:

```
00000000 _NT_TIB         struc ; (sizeof=0x1C, align=0x4, copyof_3)
00000000                                         ; XREF: _TEB/r
00000000 ExceptionList   dd ?                    ; offset
00000004 StackBase       dd ?                    ; offset
00000008 StackLimit      dd ?                    ; offset
0000000C SubSystemTib    dd ?                    ; offset
00000010 anonymous_0     _NT_TIB::$0349ADB4452EC09BEC08E2292695FBBA ?
00000014 ArbitraryUserPointer dd ?               ; offset
00000018 Self            dd ?                    ; offset
0000001C _NT_TIB         ends
```

[Figure 43]: _NT_TIB structure

Although readers are not able to see the exact type of **ExceptionList member** as well as other members in the image above, you can retrieve the full structure definition by going to **Local Types tab** (**SHIFT+F1**), searching the **_NT_TIB** and requesting to edit it (**CTRL+E**). The same information can be retrieved from https://www.vergiliusproject.com/kernels/x64/Windows%2011/22H2%20(2022%20Update)/_NT_TIB.

I prefer fetching structure definitions from the **IDA Pro** always that it is possible, as shown below:

```
struct _NT_TIB
{
  struct _EXCEPTION_REGISTRATION_RECORD *ExceptionList;
  PVOID StackBase;
  PVOID StackLimit;
  PVOID SubSystemTib;
  union
  {
    PVOID FiberData;
    DWORD Version;
  };
  PVOID ArbitraryUserPointer;
  struct _NT_TIB *Self;
};
```

[Figure 44]: _NT_TIB structure with type information

Following the same procedure, we learn that the **_EXCEPTION_REGISTRATION_RECORD** structure has the definition below:

```
struct _EXCEPTION_REGISTRATION_RECORD
{
  struct _EXCEPTION_REGISTRATION_RECORD *Next;
  PEXCEPTION_ROUTINE Handler;
};
```

[Figure 45]: _EXCEPTION_REGISTRATION_RECORD structure

Therefore, the **_PEB → _TEB → _NT_TIB structure** stores a member that is a pointer to a list of **_EXCEPTION_REGISTRATION_RECORD** structures (linked by the **Next pointer),** which holds a **field named Handler.** In other words, this **Handler member (PEXCEPTION_ROUTINE type)** represents an exception routine (actually, the **_EXCEPTION_RECORD**), which is linked to other **_EXCEPTION_RECORD** structures through its first field, as follows:

```
typedef EXCEPTION_DISPOSITION __stdcall EXCEPTION_ROUTINE(struct
_EXCEPTION_RECORD *ExceptionRecord, PVOID EstablisherFrame,
struct _CONTEXT *ContextRecord, PVOID DispatcherContext);
```

[Figure 46]: EXCEPTION_ROUTINE type

```
Offset Size  struct _EXCEPTION_RECORD
             {
  0000 0004    DWORD ExceptionCode;
  0004 0004    DWORD ExceptionFlags;
  0008 0004    struct _EXCEPTION_RECORD *ExceptionRecord;
  000C 0004    PVOID ExceptionAddress;
  0010 0004    DWORD NumberParameters;
  0014 003C    ULONG_PTR ExceptionInformation[15];
       0050  };
```

[Figure 47]: _EXCEPTION_RECORD structure

Returning to **DllRegisterServer** routine, go into **sub_5930E8** and the following function will be presented:

```
_DWORD *__userpurge sub_5930E8@<eax>(_DWORD *a1@<ecx>, int *a2@<esi>, int a3)
{
  bool v4; // zf
  int (__stdcall *v6)(int, int *); // eax

  v4 = dword_59D224 == 0xEB797E01;
  *a1 = a3;
  if ( v4 )
  {
    HIBYTE(word_59D2F0) = 1;                    // ntdll.dll!b'RtlAddVectoredExceptionHandler'
    v6 = sub_59306C(0x60A28C5C, 0x5EC9D014, a2, 0x60A28C5C, 0x60A28C5C);
    dword_59D224 = sub_593138(v6);
    HIBYTE(word_59D2F0) = 0;
  }
  return a1;
}
```

[Figure 48]: sub_5930E8 routine

Before we proceed, there is a small detail to comment. One **pages 11 and 12**, we discussed about a second routine that also is responsible for resolving hashes and that accepts five arguments. Furthermore, this routine is **referred 59 times**. Certainly, we can apply the same approach to improve and solve the API hashing issues. As readers can see on **Figure 48,** the **Shellcode Hashes plugin from Mandiant** has already solved, but we haven't done the same with **HashDB**. If readers to analyze the **sub_59306C → sub_59143C** routine, you will learn that the **XOR key** is exactly the same (**0xE462D21C**).

Therefore, we must **change the sub_59306C signature** from:

- **char *__userpurge sub_59306C@<eax>(int@<eax>, char *@<edx>, int *@<esi>, int, int)**

To:

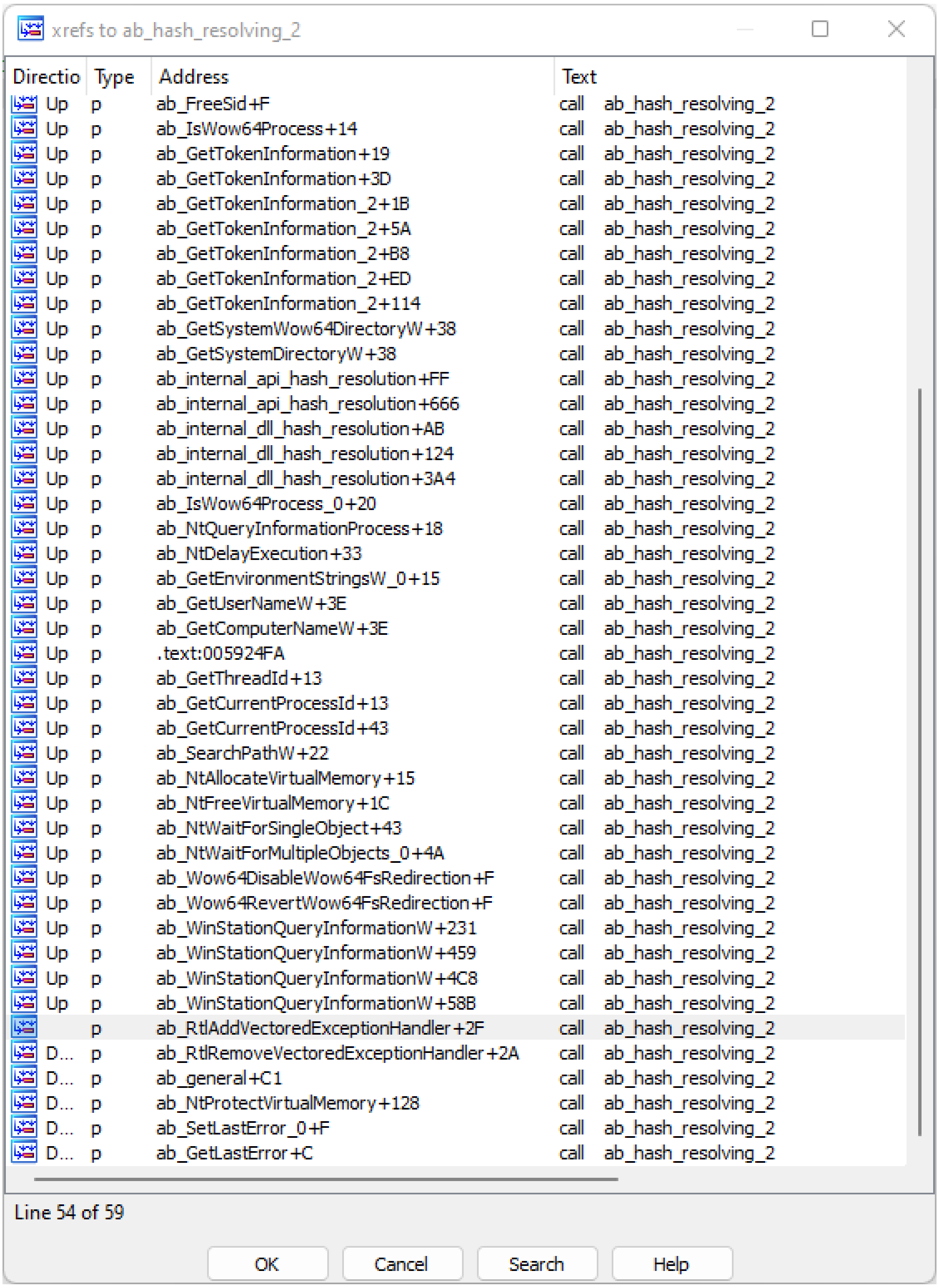
- **char *__userpurge sub_59306C@<eax>(hashdb_strings_crc32@<eax>, hashdb_strings_crc32 @<edx>, int *@<esi>, hashdb_strings_crc32, hashdb_strings_crc32)**

The same **sub_59306C** routine is now present as:

```
void __usercall sub_593144(_DWORD *a1@<ecx>, int *a2@<esi>)
{
  if ( *a1 && dword_59D224 != 0xEB797E01 )
  {
    dword_59D224 = 0xEB797E01;                  // ntdll.dll!b'RtlRemoveVectoredExceptionHandler'
    if ( sub_59306C(
           ntdll_dll,
           RtlRemoveVectoredExceptionHandler_0,
           a2,
           ntdll_dll,
           ntdll_dll) )
    {
      __debugbreak();
      __debugbreak();
    }
  }
}
```

[Figure 49]: sub_5930E8 routine: after hashing resolving with HashDB

We have found the **RtlRemoveVectoredExceptionHandler** routine, which it is responsible for unregistering a **vectored exception handler**, so it is much likely that we are close to find the **RtlAddVectoredContinueHandler** routine. If readers perform the same approach from **sub_593064 routine** that we did on **Figure 40** to this second API hashing routine (**sub_59306C routine – renamed as ab_hash_resolving_2**), resolving each one of the references using **HashDB plugin** and renaming each respective parent function, the answer will come up instantly:


xrefs to ab_hash_resolving_2

Directio	Type	Address	Text
Up	p	ab_FreeSid+F	call ab_hash_resolving_2
Up	p	ab_IsWow64Process+14	call ab_hash_resolving_2
Up	p	ab_GetTokenInformation+19	call ab_hash_resolving_2
Up	p	ab_GetTokenInformation+3D	call ab_hash_resolving_2
Up	p	ab_GetTokenInformation_2+1B	call ab_hash_resolving_2
Up	p	ab_GetTokenInformation_2+5A	call ab_hash_resolving_2
Up	p	ab_GetTokenInformation_2+B8	call ab_hash_resolving_2
Up	p	ab_GetTokenInformation_2+ED	call ab_hash_resolving_2
Up	p	ab_GetTokenInformation_2+114	call ab_hash_resolving_2
Up	p	ab_GetSystemWow64DirectoryW+38	call ab_hash_resolving_2
Up	p	ab_GetSystemDirectoryW+38	call ab_hash_resolving_2
Up	p	ab_internal_api_hash_resolution+FF	call ab_hash_resolving_2
Up	p	ab_internal_api_hash_resolution+666	call ab_hash_resolving_2
Up	p	ab_internal_dll_hash_resolution+AB	call ab_hash_resolving_2
Up	p	ab_internal_dll_hash_resolution+124	call ab_hash_resolving_2
Up	p	ab_internal_dll_hash_resolution+3A4	call ab_hash_resolving_2
Up	p	ab_IsWow64Process_0+20	call ab_hash_resolving_2
Up	p	ab_NtQueryInformationProcess+18	call ab_hash_resolving_2
Up	p	ab_NtDelayExecution+33	call ab_hash_resolving_2
Up	p	ab_GetEnvironmentStringsW_0+15	call ab_hash_resolving_2
Up	p	ab_GetUserNameW+3E	call ab_hash_resolving_2
Up	p	ab_GetComputerNameW+3E	call ab_hash_resolving_2
Up	p	.text:005924FA	call ab_hash_resolving_2
Up	p	ab_GetThreadId+13	call ab_hash_resolving_2
Up	p	ab_GetCurrentProcessId+13	call ab_hash_resolving_2
Up	p	ab_GetCurrentProcessId+43	call ab_hash_resolving_2
Up	p	ab_SearchPathW+22	call ab_hash_resolving_2
Up	p	ab_NtAllocateVirtualMemory+15	call ab_hash_resolving_2
Up	p	ab_NtFreeVirtualMemory+1C	call ab_hash_resolving_2
Up	p	ab_NtWaitForSingleObject+43	call ab_hash_resolving_2
Up	p	ab_NtWaitForMultipleObjects_0+4A	call ab_hash_resolving_2
Up	p	ab_Wow64DisableWow64FsRedirection+F	call ab_hash_resolving_2
Up	p	ab_Wow64RevertWow64FsRedirection+F	call ab_hash_resolving_2
Up	p	ab_WinStationQueryInformationW+231	call ab_hash_resolving_2
Up	p	ab_WinStationQueryInformationW+459	call ab_hash_resolving_2
Up	p	ab_WinStationQueryInformationW+4C8	call ab_hash_resolving_2
Up	p	ab_WinStationQueryInformationW+58B	call ab_hash_resolving_2
	p	ab_RtlAddVectoredExceptionHandler+2F	call ab_hash_resolving_2
D...	p	ab_RtlRemoveVectoredExceptionHandler+2A	call ab_hash_resolving_2
D...	p	ab_general+C1	call ab_hash_resolving_2
D...	p	ab_NtProtectVirtualMemory+128	call ab_hash_resolving_2
D...	p	ab_SetLastError_0+F	call ab_hash_resolving_2
D...	p	ab_GetLastError+C	call ab_hash_resolving_2

Line 54 of 59

OK Cancel Search Help

[Figure 50]: sub_59306C routine (renamed as ab_hash_resolving_2) references

Returning to **DllRegisterServer routine** for the third time, we realize that **RtlVectorExceptionHandler** was already there:

```
HRESULT __stdcall DllRegisterServer()
{
  int *v0; // esi
  unsigned int i; // eax
  int v3; // eax
  struct _EXCEPTION_REGISTRATION_RECORD *ExceptionList; // [esp+24h] [ebp-8h] BYREF
  int *v5; // [esp+28h] [ebp-4h]

  v5 = sub_5821FC;
  ExceptionList = NtCurrentTeb()->NtTib.ExceptionList;
  for ( i = 0; i < 0x13512; ++i )
  {
    __debugbreak();
    __debugbreak();
    __debugbreak();
    __debugbreak();
  }
  v5 = v0;
  ExceptionList = v0;
  ab_RtlAddVectoredExceptionHandler(&ExceptionList, v0, 0);
  if ( !byte_59D028 )
  {
    ab_OutputDebugStringW_CreateProcess(v0);
    dword_59D1C4(0);
LABEL_8:
    ab_RtlRemoveVectoredExceptionHandler(&ExceptionList, v0);
    return 1;
  }
  if ( byte_59D268 )
    goto LABEL_8;
  byte_59D268 = 1;                              // kernel32.dll!b'CreateThread'
  v0 = ab_hash_resolving(kernel32_dll, CreateThread_0);
  if ( v0 )
  {
    v3 = sub_591D30(0xA731522);
    if ( (v0)(0, 0, ab_VirtualFree_ExitThread, v3, 0, 0) )
      goto LABEL_8;
  }
  ab_RtlRemoveVectoredExceptionHandler(&ExceptionList, v0);
  return 0;
}
```

[Figure 51]: DllRegisterServer routine, after resolving API hashes.

The **RtlAddVectoredContinueHandler** routine has the following prototype (check it up on: https://github.com/winsiderss/systeminformer/blob/master/phnt/include/ntrtl.h):

```
RtlAddVectoredContinueHandler(
  _In_ ULONG First,
  _In_ PVECTORED_EXCEPTION_HANDLER Handler
  );
```

Although almost certainly readers already know about this topic, let me write few words about the exceptions. So far, we are referring to **Structure Exception Handlers (SEH)** as in malware analysis as in exploit development. Do you remember about old stack exploitation techniques to avoid cookies through SEH (pop pop ret)? The fundamental idea of **SEH** is based on exception and termination handling, and it is highly likely that readers already have seen C constructions **__try** + **__finally** or **__try** + **__except**. As same way, **try** + **catch** constructions in C++ should be common to readers. Part of the explanation mentioned on **pages 34** and **35** has SEH as reference.

Nonetheless, Windows provides a supplemental exception mechanism named **Vectored Exception Handling (VEH)**, which is a sort of extension to **structured exception handling (SEH)** and works together with SEH. **VEH** allows an application to register a function (callback function) to watch or even handling exceptions from a thread. When the exception happens, so this callback function triggers a notification and send it to the application. As a rule**, VEH handlers** are called before **SEH handlers**, but they are called in the order that they are added unless you specify a specific order. These **VEH handlers** are registered by calling **AddVectoredExceptionHandler( )**:

```
PVOID AddVectoredExceptionHandler(
  ULONG                       First,
  PVECTORED_EXCEPTION_HANDLER Handler
);
```

[Figure 52]: AddVectoredExceptionHandler( )

The most valuable information here is the **Handler**, which is a pointer to a callback function, whose respective type is **PVECTORED_EXCEPTION_HANDLER**. The callback has the following prototype:

```
PVECTORED_EXCEPTION_HANDLER PvectoredExceptionHandler;

LONG PvectoredExceptionHandler(
  _EXCEPTION_POINTERS *ExceptionInfo
)
{...}
```

[Figure 53]: PvectoredExceptionHandler callback function

The **ExceptionInfo parameter** is a pointer to **EXCEPTION_POINTERS structure**, which receives the exception record, and the **EXCEPTION_POINTERS structure** is defined as:

```
typedef struct _EXCEPTION_POINTERS {
  PEXCEPTION_RECORD ExceptionRecord;
  PCONTEXT          ContextRecord;
} EXCEPTION_POINTERS, *PEXCEPTION_POINTERS;
```

[Figure 54]: _EXCEPTION_POINTERS structure

As we can see above, there is a pointer (**ExceptionRecord**) to **EXCEPTION_RECORD structure** and another pointer (**ContextRecord**) to **CONTEXT structure**. The **EXCEPTION_RECORD structure** describes an exception that is independent of the machine, and the **CONTEXT structure** holds a series of information bound to processor's registers, so its composition changes from Intel x64 processor to ARM64 processors, for example.

As we already showed the **EXCEPTION_RECORD structure** (**page 35**), so maybe it is relevant to show parts of the **_CONTEXT structure** (defined in **WinNT.h**):

```
typedef struct _CONTEXT {
  DWORD64 P1Home;
  DWORD64 P2Home;
  DWORD64 P3Home;
  DWORD64 P4Home;
  DWORD64 P5Home;
  DWORD64 P6Home;
  DWORD   ContextFlags;
  DWORD   MxCsr;
  WORD    SegCs;
  WORD    SegDs;
  WORD    SegEs;
  WORD    SegFs;
  WORD    SegGs;
  WORD    SegSs;
  DWORD   EFlags;
  DWORD64 Dr0;
```

[Figure 55]: _CONTEXT structure: first lines

```
      M128A Xmm11;
      M128A Xmm12;
      M128A Xmm13;
      M128A Xmm14;
      M128A Xmm15;
    } DUMMYSTRUCTNAME;
    DWORD           S[32];
  } DUMMYUNIONNAME;
  M128A   VectorRegister[26];
  DWORD64 VectorControl;
  DWORD64 DebugControl;
  DWORD64 LastBranchToRip;
  DWORD64 LastBranchFromRip;
  DWORD64 LastExceptionToRip;
  DWORD64 LastExceptionFromRip;
} CONTEXT, *PCONTEXT;
```

[Figure 56]: _CONTEXT structure: last lines

Now that we quickly refreshed few facts about exceptions, it is time to return to our code in **DllRegisterServer( )**, which as we saw in **Figure 51**. As we learned, **AddVectoredExceptionHandler( )** registers a vectored exception handler that, actually, is a callback method.

However, the malware is not using **AddVectoredExceptionHandler( )**, but **RtlAddVectoredExceptionHandler( )** that, fortunately, has identical arguments (check it on: https://processhacker.sourceforge.io/doc/ntrtl_8h.html#aa9f0aa2c4497322dc3c16e509967baea).

The **RtlAddVectoredExceptionHandler( )** returns a pointer to the exception handlers, but you should pay attention to the fact that is not the real **RtlAddVectoredExceptionHandler( )**, but a wrapper to it. Thus, moving into **RtlAddVectoredExceptionHandler( )**, we have:

```
_DWORD *__userpurge ab_RtlAddVectoredExceptionHandler@<eax>(
        _DWORD *a1@<ecx>,
        int *a2@<esi>,
        int a3)
{
  bool v4; // zf
  char *RtlAddVectoredExceptionHandler; // eax

  v4 = dword_59D224 == 0xEB797E01;
  *a1 = a3;
  if ( v4 )
  {
    HIBYTE(word_59D2F0) = 1;                    // ntdll.dll!b'RtlAddVectoredExceptionHandler'
    RtlAddVectoredExceptionHandler = ab_hash_resolving_2(
                                       ntdll_dll,
                                       RtlAddVectoredExceptionHandler_0,
                                       a2,
                                       ntdll_dll,
                                       ntdll_dll);
    dword_59D224 = sub_593138(RtlAddVectoredExceptionHandler);
    HIBYTE(word_59D2F0) = 0;
  }
  return a1;
}
```

[Figure 57]: ab_RtlAddVectoredExceptionHandler routine

This routine is pretty identical to other ones. Checking its respective Assembly code and also **sub_593138** routine, we have:

```
.text:00593104 loc_593104:                             ; CODE XREF: ab_RtlAddVectoredExceptionHandler+14↑j
.text:00593104                 mov     eax, 60A28C5Ch  ; hashdb_strings_crc32
.text:00593109                 mov     edx, 5EC9D014h  ; ntdll.dll!b'RtlAddVectoredExceptionHandler'
.text:0059310E                 push    eax             ; hashdb_strings_crc32
.text:0059310F                 push    eax             ; hashdb_strings_crc32
.text:00593110                 mov     byte ptr word_59D2F0+1, 1
.text:00593117                 call    ab_hash_resolving_2
.text:0059311C                 mov     ecx, eax
.text:0059311E                 mov     edx, offset dword_5934B0
.text:00593123                 call    sub_593138
.text:00593128                 mov     dword_59D224, eax
.text:0059312D                 mov     byte ptr word_59D2F0+1, 0
.text:00593134                 jmp     short loc_5930FE
.text:00593134 ab_RtlAddVectoredExceptionHandler endp
.text:00593134
.text:00593134 ; ---------------------------------------------------------------------------
.text:00593136                 align 4
.text:00593138
.text:00593138 ; =============== S U B R O U T I N E =======================================
.text:00593138
.text:00593138
.text:00593138 ; int __thiscall sub_593138(int (__stdcall *this)(int, int *))
.text:00593138 sub_593138      proc near               ; CODE XREF: ab_RtlAddVectoredExceptionHandler+3B↑p
.text:00593138                 push    offset dword_5934B0
.text:0059313D                 push    1
.text:0059313F                 call    ecx
.text:00593141                 retn
.text:00593141 sub_593138      endp
```

RtlAddVectoredExceptionHandle

[Figure 58]: ab_RtlAddVectoredExceptionHandler routine: Assembly code

The **Figure 58** shows us that the **sub_593138** (**Figure 57, line 20**) is actually calling **RtlAddVectoredExceptionHandler( )**, which was just resolved in the previous assembly instruction (**Figure 20, line 14**). At this point, the handler is registered and ready to be called.

Therefore, the malware forces the handler to be executed as an exception handler. Once the handler is called, it will do its job and, after having finished, it will execute a return to the next value on the top of the stack, which is exactly the eax's value (returned by **ab_hash_resolving( )**) and that is the resolved API address. In this case, the code is using two **int 3** instructions (**0xCC,0xCC**) as equivalent to **call eax** (**0xFF,0xD0**). Just in case readers want to check these opcodes, a valuable resource is the online assembler and disassembler on: https://shell-storm.org/online/Online-Assembler-and-Disassembler/.

```
.text:005936B0                 jz      short loc_5936D6
.text:005936B2                 test    ebp, ebp
.text:005936B4                 jz      short loc_5936D6
.text:005936B6                 push    60014416h       ; ntoskrnl.exe!b'memset'
.text:005936BB                 push    60A28C5Ch       ; hashdb_strings_crc32
.text:005936C0                 call    ab_hash_resolving
.text:005936C5                 test    eax, eax
.text:005936C7                 jz      short loc_5936D6
.text:005936C9                 push    ebp
.text:005936CA                 movzx   ebx, bl
.text:005936CD                 push    ebx
.text:005936CE                 push    edi
.text:005936CF                 int     3               ; Trap to Debugger
.text:005936D0                 int     3               ; Trap to Debugger
.text:005936D1                 add     esp, 0Ch
.text:005936D4                 jmp     short loc_5936D8
.text:005936D6 ; ---------------------------------------------------------------
.text:005936D6
.text:005936D6 loc_5936D6:                             ; CODE XREF: ab_memset+18
.text:005936D6                                         ; ab_memset+1C↑j ...
.text:005936D6                 xor     eax, eax
.text:005936D8
.text:005936D8 loc_5936D8:                             ; CODE XREF: ab_memset+3C
.text:005936D8                 pop     ebp
.text:005936D9                 pop     ebx
.text:005936DA                 pop     edi
.text:005936DB                 retn
.text:005936DB ab_memset       endp
```

```
00593620  E9 03 00 00 00 90 90 90  56 57 53 BE 28 D2 59 00
00593630  8B F9 81 3E E5 83 3C A3  74 22 68 68 8B B6 45 68
00593640  5C 8C A2 60 E8 1B FA FF  FF 85 C0 74 09 57 6A 08
00593650  FF 36 CC CC EB 02 33 C0  5B 5F 5E C3 68 87 F3 6E
00593660  1C 68 5C 8C A2 60 E8 F9  F9 FF FF 8B D8 68 A3 FA
00593670  0A 5E 68 5C 8C A2 60 E8  E8 F9 FF FF A3 2C D2 59
00593680  00 81 3E E5 83 3C A3 75  B1 33 C0 50 50 50 50 50
00593690  6A 02 FF D3 89 06 EB A2  8B 44 24 04 0F B6 54 24
005936A0  08 8B 4C 24 0C 57 53 55  8B F8 8B E9 8B DA 85 FF
005936B0  74 24 85 ED 74 20 68 16  44 01 60 68 5C 8C A2 60
005936C0  E8 9F F9 FF FF 85 C0 74  0D 55 0F B6 DB 53 57 CC
005936D0  CC 83 C4 0C EB 02 33 C0  5D 5B 5F C3 8B 44 24 04
005936E0  8B 54 24 08 8B 4C 24 0C  56 53 55 8B EA 8B F1 8B
005936F0  D8 85 ED 74 25 85 DB 74  21 85 F6 74 1D 68 CF 7D
00593700  23 35 68 5C 8C A2 60 E8  58 F9 FF FF 85 C0 74 0A
00593710  56 55 53 CC CC 83 C4 0C  EB 02 33 C0 5D 5B 5E C3
00593720  56 57 53 55 8B 7C 24 14  85 FF 74 34 8B 74 24 18
00593730  85 F6 74 2C 8B 6C 24 1C  85 ED 74 24 8B 5C 24 20
00593740  85 DB 74 1C 68 40 C1 92  96 68 5C 8C A2 60 E8 11
00593750  F9 FF FF 85 C0 74 09 53  55 56 57 CC CC 83 C4 10
00593760  5D 5B 5F 5E C3 90 90 90  56 57 53 55 8B 7C 24 14
00593770  85 FF 74 41 8B 74 24 18  85 F6 74 39 8B 6C 24 1C
00593780  85 ED 74 31 8B 5C 24 20  85 DB 74 29 83 7C 24 24
00593790  00 74 22 68 AA 2A 15 60  68 5C 8C A2 60 E8 C2 F8
005937A0  FF FF 85 C0 74 0F FF 74  24 24 53 55 56 57 CC CC
005937B0  83 C4 14 EB 02 33 C0 5D  5B 5F 5E C3 56 57 53 55
005937C0  83 EC 0C 8B 54 24 24 8B  74 24 2C 8B 44 24 20 3B
005937D0  F2 77 2A 8B 5C 24 28 8B  FA 2B FE 8D 2C 02 2B EE
005937E0  0F B6 1B EB 08 8D 47 01  F7 DF 03 FD 4F 47 8B D3
005937F0  8B CF E8 E6 00 00 00 8B  F8 85 FF 75 04 33 C0 EB
00593800  13 8B C7 8B CE 8B 54 24  28 E8 1A 00 00 00 85 C0
00593810  75 D3 8B C7 83 C4 0C 5D  5B 5F 5E C3 8B 44 24 04
00593820  8B 54 24 08 8B 4C 24 0C  56 53 55 8B D8 8B F1 8B
```

[Figure 59]: sub_593698 routine, and the synchronized HexView that shows two CC opcodes.

The next suggested step is to **make a backup of the IDA .idb file and the unpacked sample** to avoid corrupting them. I will be using, only as reference, the **sub_593698 routine** (renamed to **ab_memset**):

```
int __cdecl ab_memset(int a1, char a2, int a3)
{
  int result; // eax

  if ( !a1 )
    return 0;
  if ( !a3 )
    return 0;
  result = ab_hash_resolving(ntdll_dll, memset_0);// ntoskrnl.exe!b'memset'
  if ( !result )
    return 0;
  __debugbreak();
  __debugbreak();
  return result;
}
```

[Figure 60]: sub_593698 routine, which will be used as reference for changes

To confirm whether our theory that the two **int 3 instructions** (**\xCC\xCC**) is equivalent to **call eax** (**\xFF\xD0**), we are alter the hexadecimal directly in the **Hex View tab**. To do it, click on **Hex View tab**, press **F2 hotkey** and make the change:

```
00593620  E9 03 00 00 00 90 90 90  56 57 53 BE 28 D2 59 00
00593630  8B F9 81 3E E5 83 3C A3  74 22 68 68 8B B6 45 68
00593640  5C 8C A2 60 E8 1B FA FF  FF 85 C0 74 09 57 6A 08
00593650  FF 36 CC CC EB 02 33 C0  5B 5F 5E C3 68 87 F3 6E
00593660  1C 68 5C 8C A2 60 E8 F9  F9 FF FF 8B D8 68 A3 FA
00593670  0A 5E 68 5C 8C A2 60 E8  E8 F9 FF FF A3 2C D2 59
00593680  00 81 3E E5 83 3C A3 75  B1 33 C0 50 50 50 50 50
00593690  6A 02 FF D3 89 06 EB A2  8B 44 24 04 0F B6 54 24
005936A0  08 8B 4C 24 0C 57 53 55  8B F8 8B E9 8B DA 85 FF
005936B0  74 24 85 ED 74 20 68 16  44 01 60 68 5C 8C A2 60
005936C0  E8 9F F9 FF FF 85 C0 74  0D 55 0F B6 DB 53 57 FF
005936D0  D0 83 C4 0C EB 02 33 C0  5D 5B 5F C3 8B 44 24 04
005936E0  8B 54 24 08 8B 4C 24 0C  56 53 55 8B EA 8B F1 8B
005936F0  D8 85 ED 74 25 85 DB 74  21 85 F6 74 1D 68 CF 7D
00593700  23 35 68 5C 8C A2 60 E8  58 F9 FF FF 85 C0 74 0A
00593710  56 55 53 CC CC 83 C4 0C  EB 02 33 C0 5D 5B 5E C3
00593720  56 57 53 55 8B 7C 24 14  85 FF 74 34 8B 74 24 18
00593730  85 F6 74 2C 8B 6C 24 1C  85 ED 74 24 8B 5C 24 20
```

```
int __cdecl ab_memset(int a1, char a2, int a3)
{
  int result; // eax

  if ( !a1 )
    return 0;
  if ( !a3 )
    return 0;
  result = ab_hash_resolving(ntdll_dll, memset_0);
  if ( !result )
    return 0;
  __debugbreak();
  __debugbreak();
  return result;
}
```

[Figure 61]: sub_593698 routine: hexadecimal bytes changed

Press **F2 hotkey** again to commit changes and we will see the following content:

```
int __cdecl ab_memset(int a1, unsigned __int8 a2, int a3)
{
  int (__cdecl *memset_0)(int, _DWORD, int); // eax

  if ( a1
    && a3
    && (memset_0 = (int (__cdecl *)(int, _DWORD, int))ab_hash_resolving(
                                                         ntdll_dll,
                                                         memset_0)) != 0 )
  {
    return memset_0(a1, a2, a3);
  }
  else
  {
    return 0;
  }
}
```

[Figure 62]: sub_593698 routine: after changes

We have gotten a much better result because:

- there are not both **__debugbreak( )** instructions anymore.
- we can see the **memset( ) function** being explicitly called with its three parameters, which it was not possible previously.
- the **IF condition** has been completely fixed and we can see what's really happening.
- the **function pointer** to **memset** appeared and confirms that the function accepts three arguments.
- the **Assembly view (IDA View-A)** has been fixed too and there isn't any analysis issue (red line) marked on the code.

To save space here, I will show only one more example with effects from this change to illustrate that we will have a much clearer pseudo and Assembly code after doing this manipulation over the code.

```
int ab_RtlAllocateHeap()
{
  int RtlAllocateHeap_0; // eax
  int (__cdecl *RtlCreateHeap_0)(int, _DWORD, _DWORD, _DWORD, _DWORD, _DWORD); // ebx

  if ( dword_59D228 == 0xA33C83E5 )
  {
    RtlCreateHeap_0 = ab_hash_resolving(ntdll_dll, RtlCreateHeap_0);// ntoskrnl.exe!b'RtlCreateHeap'
    dword_59D22C = ab_hash_resolving(ntdll_dll, RtlDestroyHeap_0);// ntoskrnl.exe!b'RtlDestroyHeap'
    if ( dword_59D228 == 0xA33C83E5 )
      dword_59D228 = RtlCreateHeap_0(2, 0, 0, 0, 0, 0);
  }
  RtlAllocateHeap_0 = ab_hash_resolving(ntdll_dll, RtlAllocateHeap_0);// ntoskrnl.exe!b'RtlAllocateHeap'
  if ( !RtlAllocateHeap_0 )
    return 0;
  __debugbreak();
  __debugbreak();
  return RtlAllocateHeap_0;
}
```

[Figure 63]: sub_593628 routine

After we have followed the same procedure and applied changes, we got the following result:

```
int __thiscall ab_RtlAllocateHeap(void *this)
{
  int (__stdcall *RtlAllocateHeap_0)(int, int, void *); // eax
  int (__stdcall *RtlCreateHeap_0)(int, _DWORD, _DWORD, _DWORD, _DWORD, _DWORD); // ebx

  if ( dword_59D228 == 0xA33C83E5 )
  {
    RtlCreateHeap_0 = ab_hash_resolving(ntdll_dll, RtlCreateHeap_0);// ntoskrnl.exe!b'RtlCreateHeap'
    RtlDestroyHeap_0_1 = ab_hash_resolving(ntdll_dll, RtlDestroyHeap_0);// ntoskrnl.exe!b'RtlDestroyHeap'
    if ( dword_59D228 == 0xA33C83E5 )
      dword_59D228 = RtlCreateHeap_0(2, 0, 0, 0, 0, 0);
  }
  RtlAllocateHeap_0 = ab_hash_resolving(ntdll_dll, RtlAllocateHeap_0);// ntoskrnl.exe!b'RtlAllocateHeap'
  if ( RtlAllocateHeap_0 )
    return RtlAllocateHeap_0(dword_59D228, 8, this);
  else
    return 0;
}
```

[Figure 64]: sub_593628 routine: after changes

Once again, the final result is clearer, and we can see both **RtlCreateHeap_0( )** and **RtlAllocateHeap_0( )** being invoked with all arguments. Another beneficial effect of this change is that we also can perform a supplemental marking-up on the code due to fact that new lines were revealed to us.

The next step is composed by the following tasks:

- evaluating **the number of occurrences of this hexadecimal sequence exist** on the idb file.
- performing **replacements on the IDA .idb file** or directly on the **unpacked binary file**.

The IDA Pro provides an efficient and effortless way to search for binaries sequences and text, which will be especially useful for us to accomplish the first task.

Clicking on any line on **IDA View-A**, press **ALT+B hotkey** to activate the **Binary search** form. In the **String field**, type **CC CC** and make sure that **Hex** format is selected as well as **Find all occurrences** too, and press **OK**:

Address	Function	Instruction
.text:0058795D		align 10h
.text:00599A28		align 10h
.text:00599A34		align 10h
.text:00595F79		dd 0FFD0F3E8h, 74C085FFh, 75FF560Eh, 85CCCC00h, 330A74C0h
.text:005934F2		dd 247C8B57h, 8B078B08h, 53D00h, 1974C000h, 0FD3Dh, 3D1274C0h
.text:00596BA0		dd 7C8B5756h, 778D0C24h, 0B2176814h, 0B768A738h, 0C7E62A48h
.text:0058133D	ab_NtMapViewOfSection	int 3; Trap to Debugger
.text:005813FA	ab_NtMapViewOfSection	int 3; Trap to Debugger
.text:0058142A	ab_NtMapViewOfSection	int 3; Trap to Debugger
.text:00581463	ab_NtMapViewOfSection	int 3; Trap to Debugger
.text:0058200B	ab_NtDuplicateObject	int 3; Trap to Debugger
.text:005821E8	DllRegisterServer	int 3; Trap to Debugger
.text:005821EB	DllRegisterServer	int 3; Trap to Debugger
.text:00582303	ab_RegLoadKeyW	int 3; Trap to Debugger
.text:00582374	ab_RegLoadKeyW	int 3; Trap to Debugger
.text:0058255D	ab_OutputDebugStringW_C...	int 3; Trap to Debugger
.text:00582586	ab_OutputDebugStringW_C...	int 3; Trap to Debugger
.text:005825AF	ab_OutputDebugStringW_C...	int 3; Trap to Debugger
.text:005825D8	ab_OutputDebugStringW_C...	int 3; Trap to Debugger
.text:00582698	ab_OutputDebugStringW_C...	int 3; Trap to Debugger
.text:005826BB	ab_OutputDebugStringW_C...	int 3; Trap to Debugger
.text:00582A81	ab_OutputDebugStringW_C...	int 3; Trap to Debugger
.text:00583C3B	ab_CoCreateInstance	int 3; Trap to Debugger
.text:00583FEE	ab_CoCreateInstance	int 3; Trap to Debugger
.text:005845FB	ab_NtOpenMutant_explorer...	int 3; Trap to Debugger
.text:005853F3	ab_GetEnvironmentStringsW	int 3; Trap to Debugger
.text:005857EF	ab_OutputDebugStringW	int 3; Trap to Debugger
.text:005868EB	ab_RegUnLoadKeyW	int 3; Trap to Debugger
.text:00587812	ab_CreateProcessW	int 3; Trap to Debugger
.text:005888D2	ab_NtSetEvent	int 3; Trap to Debugger
.text:00588B7F	ab_NtQueueApcThread	int 3; Trap to Debugger
.text:00588CFD	ab_code_injection_1	int 3; Trap to Debugger
.text:00589021	ab_code_injection_1	int 3; Trap to Debugger
.text:0058B4BC	ab_CreateFileMappingW_Nt...	int 3; Trap to Debugger
.text:0058B4E4	ab_CreateFileMappingW_Nt...	int 3; Trap to Debugger
.text:0058B545	ab_CreateFileMappingW_Nt...	int 3; Trap to Debugger
.text:0058B5BB	ab_NtDuplicateObject_0	int 3; Trap to Debugger
.text:0058B649	ab_NtDuplicateObject_1	int 3; Trap to Debugger
.text:0058B677	ab_thread_searching	int 3; Trap to Debugger
.text:0058B6B1	ab_thread_searching	int 3; Trap to Debugger
.text:0058B6F6	ab_thread_searching	int 3; Trap to Debugger
.text:0058B760	ab_thread_searching	int 3; Trap to Debugger
.text:0058B9AF	ab_GlobalAddAtomW_Global...	int 3; Trap to Debugger
.text:0058B9EA	ab_GlobalDeleteAtom	int 3; Trap to Debugger
.text:0058BB65	ab_NtClose	int 3; Trap to Debugger
.text:0058C062	ab_GetVolumeInformationW	int 3; Trap to Debugger
.text:0058C761	ab_NtCreateMutant	int 3; Trap to Debugger
.text:0058C818	ab_ConvertStringSecurityDe...	int 3; Trap to Debugger
.text:0058C84C	ab_ConvertStringSecurityDe...	int 3; Trap to Debugger
.text:0058C876	ab_ConvertStringSecurityDe...	int 3; Trap to Debugger

[Figure 65]: Partial results from the search for \xCC\xCC sequence

This result shows us that:

- Most of occurrences are exactly the **same trick** used to make our reversing task more complex.
- **Not all occurrences are related to Trap to Debugger**, and some of them are related to **hexadecimal data**.
- IDA Pro found **132 occurrences**, and **126 hits** are suitable for our context.

Once we have decided to write a script, we should be careful in not change all occurrences because few of them are not related to "**trap to debugger**" trick. In the other side, as these **0xCC** sequences are used as data or even as stack offset, so this few inappropriate changes would be really little impacting and would not cause any visible effect on the reversing task. Anyway, we will avoid doing it.

Another possible decision would be writing a pure Python script to change the sequence **\xCC\xCC** to **\xFF\D0** inside the binary and certainly it would work, but we would have the same side effect of changing data (instead of instructions), and we also would be changing the binary that is something I do not like. Eventually, I would have to re-analyze (and marking up) the new binary.

I have chosen writing a script using **IDA Python/IDC** and change only the **IDA Pro .idb** file to perform all necessary operations. Therefore, go to **File → Script Command**... and write the following script:

```
1  import idautils
2  import idc
3  import ida_allins
4
5  target_functions = ["ab_hash_resolving", "ab_hash_resolving_2"]
6  patch1 = 0xFF
7  patch2 = 0xD0
8
9  for t_func in target_functions:
10     target_addr = idc.get_name_ea_simple(t_func)
11     for addr_item in idautils.CodeRefsTo(target_addr, 0):
12         func_ref = ida_funcs.get_func(addr_item)
13         if(func_ref):
14             for ea in Heads(func_ref.start_ea, func_ref.end_ea):
15                 insn = idaapi.insn_t()
16                 length = idaapi.decode_insn(insn, ea)
17                 if insn.itype == ida_allins.NN_int3:
18                     idc.patch_byte(ea, patch1)
19                     idc.patch_byte(ea + 1, patch2)
20                     break
21
22 print("\n\n[*]Patch applied!\n")
```

[Figure 66]: IDA Python/IDC script for patching \xCC bytes

The script itself is quite simple, but there are few details that I would like to comment:

- If it is necessary, you can also import **ida_funcs** and **idaapi** modules explicitly.

- I used **both hash resolving functions** as reference to find the name of wrapper functions where they are being called and, having the name and start address of each wrapper function, the script lists all Assembly instructions for each wrapper function and compare with them with **int 3** instruction. This was the motivation for creating an array of function names on **line 5**, and new functions could be added to this list if it were necessary.

- The final goal is to replace **\xCC\xCC** (**int 3; int 3**) by **\xFF\xD0** (**call eax**). Therefore, I didn't want to replace both **\xCC** byte for the same provided byte, but the first **\xCC** byte should be replaced by

\xFF and the second **\xCC** byte should be replaced by **\xD0**. That is the reason for using the **break** instruction on **line 20**. Indeed, the goal was searching for the first **int 3** instruction, applying the first patch over the first **\xCC** byte and, afterwards, incrementing **ea** in **1** to get the next **int 3** address, and apply the second patch over it too.

- I could have written a script to ensure that **there would be two subsequent int 3 instructions** before applying the patch, but we already had verified previously that there was not any **int 3** out of this context.

- On **line 10**, the function **idc.get_name_ea_simple( )** retrieves the address of a function given by the **target_functions** array. Information about the function available on: https://www.hex-rays.com/products/ida/support/idadoc/255.shtml.

- On **line 11**, the **CodeRefsTo( )** gets all references to the the provided hash function and, as we already had learned previously, there are many ones. Information about the function available on: https://www.hex-rays.com/products/ida/support/idapython_docs/idautils.html#idautils.CodeRefsTo

- On **line 12**, **get_func( )** retrieves the reference (**address**) to the function object (structure), given the address of the function. Further information on: https://www.hex-rays.com/products/ida/support/idapython_docs/ida_funcs.html#ida_funcs.get_func.

- On **line 13**, the script checks whether the **reference (address) is valid (not null).** Invalid references are not a common occurrence, but it might happen, and, without this line, the script might stop.

- On **line 14**, **Heads( )** gets a list of heads (instructions or data items) given the start and end addresses. More information available on: https://www.hex-rays.com/products/ida/support/idapython_docs/idautils.html#idautils.Heads.

- On **line 15**, the **insn_t constructor**, from **insn_t class**, is called and returns an object of this class. Information on: https://www.hex-rays.com/products/ida/support/sdkdoc/classinsn__t.html.

- On **line 16**, the **decode_insn( ) function**, which interprets the specified address as an instruction and fills the **insn_t structure** provided as first parameter. The return is the length of the instruction or zero. Further information on: https://www.hex-rays.com/products/ida/support/sdkdoc/ua_8hpp.html#af83aad26f4b3e39e7fbda441100f15cf.

- On **line 17**, the **itype field** (member of **insn_t class**), which contains the internal code of the instruction, is used to check whether the provided instruction is an **int 3 instruction**. In additional, readers might be interested in the fact that it is possible to verify any instruction using **ida_allins module**. Further information on https://hex-rays.com/products/ida/support/idapython_docs/ida_allins.html#ida_allins.NN_int3.

- One **lines 18 and 19**, once we are sure that we found an **int 3 instruction**, so we can patch its respective opcode using our own opcode Please, pay attention to the fact that I used **ea variable** as argument for the first **patch_byte( )** call on the **line 18**, but I used **ea + 1** as argument for the second **patch_byte( )** call to **fix the second int 3 instruction**. Information about the **patch_byte function** can be found on: https://www.hex-rays.com/products/ida/support/idadoc/713.shtml

- The **break instruction** on **line 20** is a little trick: once the script finds the first **int 3** instruction, it leaves the interaction within the provided function, and starts to list instructions of the next one.

I have run the script once and, using the IDA Pro binary mechanism (**Search → Sequence of Bytes** – or **ALT+B**), I got the following result:

Address	Function	Instruction
.text:005821EB	DllRegisterServer	int 3; Trap to Debugger
.text:00583FEE	ab_CoCreateInstance	int 3; Trap to Debugger
.text:0058795D		align 10h
.text:00592505		int 3; Trap to Debugger
.text:005934F2		dd 247C8B57h, 8B078B08h, 53D00h, 1974C000h, 0FD3Dh, 3D1274C0h
.text:00595F79		dd 0FFD0F3E8h, 74C085FFh, 75FF560Eh, 85CCCC00h, 330A74C0h
.text:00596BA0		dd 7C8B5756h, 778D0C24h, 0B2176814h, 0B768A738h, 0C7E62A48h
.text:00599A28		align 10h
.text:00599A34		align 10h

[Figure 67]: Results for new \xCC search after running the script

The fourth result indicates a potential issue with function because the name is not appearing. Jumping to there, we can easily notice that there isn't any indication for the end of function, as shown below:

```
5924E8 ; ---------------------------------------------------------------------------
5924E8
5924E8 loc_5924E8:                             ; CODE XREF: ab_LoadLibrary_GetProcAddress-
5924E8                 push    esi
5924E9                 push    ebp
5924EA                 push    esi
5924EB                 push    esi
5924EC                 mov     esi, edx
5924EE                 mov     eax, 8E844D1Eh
5924F3                 mov     edx, 0B5CA9B57h ; kernel32.dll!b'IsBadReadPtr'
5924F8                 mov     ebp, ecx
5924FA                 call    ab_hash_resolving_2
5924FF                 test    eax, eax
592501                 jz      short loc_59250F
592503                 push    esi
592504                 push    ebp
592505                 int     3               ; Trap to Debugger
592506                 int     3               ; Trap to Debugger
592506 ; ---------------------------------------------------------------------------
592507                 db 85h
```

[Figure 68]: Results of searching for \xCC byte after running the script

Fortunately, we can fix this issue easily by putting the cursor on its first address and pressing **E hotkey**, which will solve the problem. Now, running the script a second time and repeating the search, we have:

Address	Function	Instruction
.text:0058795D		align 10h
.text:005934F2		dd 247C8B57h, 8B078B08h, 53D00h, 1974C000h, 0FD3Dh, 3D1274C0h
.text:00595F79		dd 0FFD0F3E8h, 74C085FFh, 75FF560Eh, 85CCCC00h, 330A74C0h
.text:00596BA0		dd 7C8B5756h, 778D0C24h, 0B2176814h, 0B768A738h, 0C7E62A48h
.text:00599A28		align 10h
.text:00599A34		align 10h

[Figure 69]: Results of searching for \xCC byte after running the script for the second time.

That's perfect! We got replacing all **int 3 instruction pairs** in the **.idb database** by our bytes representing **call eax**, but without changing any of data information which also was in the **.text section**. Additionally, we didn't need to create a new patched binary.

Once again, we can check the pseudo code of any of routines that contained **int 3; int 3** trick to be sure that they are correct and fortunately we realized there is a better and cleaner code, as shown below:

```
int (__stdcall *__fastcall ab_CryptGenRandom(int *a1, int a2))(int, int, int *)
{
  int (__stdcall *CryptAcquireContextW_0)(int *, _DWORD, _DWORD, int, unsigned int); // ecx
  int v5; // ebp
  int (__stdcall *CryptGenRandom_0)(int, int, int *); // eax
  int v8[4]; // [esp+1Ch] [ebp-10h] BYREF

  v8[0] = 0;
  CryptAcquireContextW_0 = ab_hash_resolving(
                             advapi32_dll,
                             CryptAcquireContextW_0);// advapi32.dll!b'CryptAcquireContextW'
  if ( !CryptAcquireContextW_0
    || !CryptAcquireContextW_0(v8, 0, 0, 0x18, 0xF0000000)
    && ab_GetLastError(a1) )
  {
    return ab_memset(a1, 0, a2);
  }
  v5 = v8[0];
  CryptGenRandom_0 = ab_hash_resolving(advapi32_dll, CryptGenRandom_0);// advapi32.dll!b'CryptGenRandom'
  if ( CryptGenRandom_0 )
    CryptGenRandom_0 = CryptGenRandom_0(v5, a2, a1);
  if ( v5 && v5 != 0xFFFFFFFF )
    return ab_CryptReleaseContext(v5);
  return CryptGenRandom_0;
}
```

[Figure 70]: sub_59494C routine: general aspect after running the script

Now, we finally have a bit better binary, which we can complete the markup process, and to be able to interpret new findings and pieces of code.

There are other aspects and portions of the **Dridex code** to be analyzed and this is a time-consuming task, obviously. Furthermore, we need to establish and focus on more objective goals because we have enough functions to spend several days in trying to analyze them.

Basically, we don't have vital information until now: **strings** and **IP addresses** used to connect to command-and-control channel (**C2**) from malware's authors.

Anyway, I have adopted the same guideline of past article in visualizing **.data** and **.rdata** sections (**CTRL+S hotkey**) and, from there, finding important routines manipulating and decrypting these data blocks.

Choose segment to jump

Name	Start	End	R	W	X	D	L	Align	Base	Type
.text	00581000	0059A000	R	.	X	.	L	para	0001	public
.idata	0059A000	0059A060	R	.	.	.	L	para	0002	public
.rdata	0059A060	0059D000	R	.	.	.	L	para	0002	public
.data	0059D000	0059E000	R	W	.	.	L	para	0003	public
.CRT	0059E000	0059F000	R	.	.	.	L	para	0004	public

Line 1 of 5

OK Cancel Search Help

[Figure 71]: Binary's sections

Choosing **.rdata section** and jumping to it, we have the following content:

```
.rdata:0059A060 unk_59A060      db 0D4h                 ; DATA XREF: sub_581000+108↑o
.rdata:0059A061                 db  67h ; g
.rdata:0059A062                 db  3Bh ; ;
.rdata:0059A063                 db  2Eh ; .
.rdata:0059A064                 db  9Ah
.rdata:0059A065                 db 0E3h
.rdata:0059A066                 db  32h ; 2
.rdata:0059A067                 db 0ACh
.rdata:0059A068                 db  71h ; q
.rdata:0059A069                 db  4Ah ; J
.rdata:0059A06A                 db 0AAh
.rdata:0059A06B                 db  68h ; h
.rdata:0059A06C                 db 0F6h
.rdata:0059A06D                 db    5
.rdata:0059A06E                 db  1Dh
.rdata:0059A06F                 db    9
.rdata:0059A070                 db 0D9h
.rdata:0059A071                 db 0FCh
.rdata:0059A072                 db 0E4h
.rdata:0059A073                 db  7Eh ; ~
.rdata:0059A074                 db  99h
.rdata:0059A075                 db  9Eh
.rdata:0059A076                 db 0C6h
.rdata:0059A077                 db  9Bh
.rdata:0059A078                 db  68h ; h
.rdata:0059A079                 db  5Fh ; _
.rdata:0059A07A                 db  2Bh ; +
.rdata:0059A07B                 db 0AAh
.rdata:0059A07C                 db  55h ; U
.rdata:0059A07D                 db  64h ; d
.rdata:0059A07E                 db  70h ; p
.rdata:0059A07F                 db  83h
.rdata:0059A080                 db  7Fh ;
```

[Figure 72]: Start of .rdata section

As readers can realize, there one reference soon at the beginning of the section. Checking the reference (**X hotkey**) and jumping to it, we have:

```
if ( LOBYTE(v15[0]) )
{
  FullProcessImageNameW = ab_w_QueryFullProcessImageNameW(v2);
  ptr_ab_encoded_data_2 = &ab_encoded_data_2;
  if ( FullProcessImageNameW == 0x20 )
    ptr_ab_encoded_data_2 = &ab_encoded_data_1;
  ab_w_rc4_0(v20, ptr_ab_encoded_data_2, 0);
  sub_58F6C0(v21, v20[0]);
  ab_ww_RtlFreeHeap_0(v20);
  v10 = sub_58F6A8(v21);
  sub_58F828(v18, v10);
  v11 = sub_58F4BC(v21, 0);
  v12 = sub_58F4BC(v18, 0);
  sub_5878B4(a2, v11, v12);
  if ( ab_w_QueryFullProcessImageNameW(v2) == 0x20 )
  {
    ab_memset(v29, 0, 0x47C);
    v29[0x103] = sub_58F4CC(a2);
    v29[0x101] = 0x56473829;
    sub_58201C(v29);
    v29[0x109] = v17;
    v29[0x10A] = v16;
    sub_58F4DC(v19, v29, 0x47C);
    v13 = ab_IAT_1(v18, v2, a2, 0x4BC, v19);
  }
```

[Figure 73]: sub_581000 routine

I've already renamed few data references and variables and, much more important, I renamed the **sub_59214C** routine to **ab_w_rc4_0** because within it there is a **call instruction** to the real **RC 4 routine**:

```
_DWORD *__stdcall ab_w_rc4_0(_DWORD *a1, int data_to_be_decrypted, int a3)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  sub_58DF4C(a1, 0x2800);
  v3 = *a1;
  *v3 = 0;
  sub_58F37C(ptr_raw_key, data_to_be_decrypted, 0x30);
  ptr_key = sub_58F4BC(ptr_raw_key, 0);
  ptr_key_1 = sub_58F4CC(ptr_raw_key);
  ab_reverse_bytes(ptr_key, ptr_key_1);
  v10 = v3;
  v12 = a3;
  v11 = 0;
  v9[0] = a3 == 0;
  key = sub_58F4BC(ptr_raw_key, 0);
  ab_rc4(key, 48, data_to_be_decrypted + 0x30, 0x7FFFFFFF, 0, sub_59341C, v9);
  ab_ww_RtlFreeHeap_1();
  return a1;
}
```

[Figure 74]: sub_59214C routine renamed to ab_w_rc4_0

The routine **ab_rc4** is the new name of **sub_594B38**, which clearly it's a **RC 4 routine (we learned about RC4 in past articles of this series)**, and it is partially shown below:

```
void __fastcall ab_rc4(
        int a1,
        int a2,
        int a3,
        int a4,
        int a5,
        int (__stdcall *a6)(int, int),
        int a7)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  if ( a1 && a2 && a3 && a4 && (a5 || a6) )
  {
    for ( i = 0; i < 0x100; ++i )
    {
      v10 = *(i % a2 + a1);
      v21[i] = i;
      *(v22 + i) = v10;
    }
    v11 = 0;
    v12 = 0;
    LOBYTE(v13) = 0;
    do
    {
      v14 = v21[v12];
      ++v11;
      v13 = (v13 + v14 + *(v22 + v12));
      v21[v12++] = v21[v13];
      v21[v13] = v14;
    }
    while ( v11 < 0x100 );
    if ( a4 > 0 )
    {
      v15 = 0;
      v16 = 1;
      LOBYTE(v17) = 0;
      do
      {
        while ( 1 )
```

[Figure 75]: sub_594B38 rename to ab_rc4

Returning to **sub_59214C (ab_w_rc4_0)**, from **Figure 74**, readers might be wondering how I got such conclusions, but they are quite easy to understand the decisions. First, look at **lines 8** and **17 (Figure 74)**, as shown below:

- **sub_58F37C(ptr_raw_key, data_to_be_decrypted, 0x30);**
- **ab_rc4(key, 48, data_to_be_decrypted + 0x30, 0x7FFFFFFF, 0, sub_59341C, v9);**

There are few points:

- **data_to_be_decrypted** argument is the second argument from **sub_59214C (ab_w_rc3_0)** and it comes from the call on **line 31** from **sub_581000 (Figure 73).**

- I know that the **first 0x30 bytes (48 bytes)** is the decryption **key** because the **second argument** of the **call instruction** for **ab_rc4 routine** (**line 17, Figure 74**) is **data_to_be_decrypted+0x30**. Additionally, **on line 8**, the **sub_58DF4C** routine using the same block of data (**data_to_be_decrypted** parameter), and the third argument is exactly the same **0x30**. If readers examine the **sub_58F37C routine**, you will confirm that it is a wrapper to **memcpy( )** function. Thus, the first argument of **sub_58F37C routine** is a pointer to the **key**, which I renamed to **ptr_raw_key**.

- This pointer is used as argument of **ab_reverse_bytes** routine (**sub_594928**), which readers can check its content and confirm that **it takes the passed array of bytes and simply invert them**:

```
char __fastcall ab_reverse_bytes(int raw_key_member, int ptr_raw_key_1)
{
  int counter_2; // ebx
  int counter_1; // ecx
  char result; // al

  counter_2 = ptr_raw_key_1 - 1;
  if ( ptr_raw_key_1 - 1 > 0 )
  {
    counter_1 = 0;
    do
    {
      result = *(counter_1 + raw_key_member);
      *(counter_1 + raw_key_member) = *(counter_2 + raw_key_member);
      ++counter_1;
      *(counter_2 + raw_key_member) = result;
      --counter_2;
    }
    while ( counter_1 < counter_2 );
  }
  return result;
}
```

[Figure 76]: sub_594928

- As a reference has been passed to the **sub_594928 routine**, so the result is also the **same content of ptr_raw_key**, which it was already inverted.

Examining references (**X hotkey)** to **ab_w_rc4_0 routine (sub_59214C)**, we realize it called seven times:

xrefs to ab_w_rc4_0

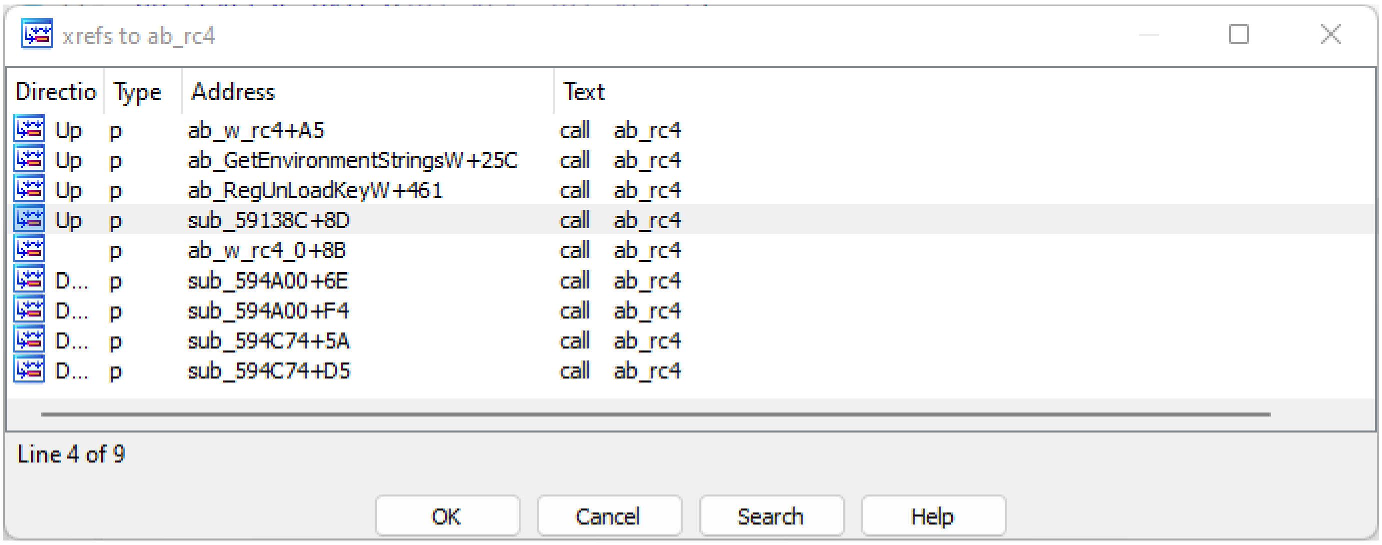
Directio	Type	Address	Text
Up	p	sub_581000+11D	call ab_w_rc4_0
Up	p	sub_58788C+A	call ab_w_rc4_0
Up	p	ab_IAT_1+11E	call ab_w_rc4_0
Up	p	ab_code_injection_2+4E5	call ab_w_rc4_0
D...	p	ab_InternetOpenW+225	call ab_w_rc4_0
D...	p	ab_Internet_2+234	call ab_w_rc4_0
D...	p	sub_59913C+24	call ab_w_rc4_0

Line 1 of 7

OK Cancel Search Help

[Figure 77]: references to ab_w_rc4_0 (sub_59214C)

However, there is a further detail. **The ab_rc4** routine (**sub_594B38**), which is the real RC4 function (and not a wrapper) is **called 9 times** (once again, check it using **X hotkey**) and one of them is by our **ab_w_rc4** wrapper function (**sub_59214C)**, as shown below:



[Figure 78]: references to ab_rc4 (sub_594B38)

Therefore, we know that this malware sample is massively using **RC4 algorithm** to encrypt its data blocks (hopefully strings and/or IP addresses), and in all cases using the same scheme:

- **[48 bytes key] [encrypted data]**

We should remember that key was originally inverted, so we have to fix it before using it.

Eventually, there can be new layers of obfuscation and encryption, but for now we don't have any further clue. That's what we know so far:

- **RC4** is being used by other routines, and readers can get information about the RC4 algorithm from https://en.wikipedia.org/wiki/RC4.
- Relevant encrypted information is stored on **.rdata section**.
- The information is stored and organized as: **[48 bytes key] [encrypted data]**
- The key is stored with its bytes **inverted**.
- We need to extract the information, and **separate it between key and data**.
- It is necessary to **invert** the extracted key before using it.
- We have to use the key to decrypt the encrypted information using **RC4 routine from a library**.

We will be focused on strings and eventual IP addresses that might appear during of the process. If we find other type of information (configuration files, binaries, shellcode and so on), we will only save them to an eventual and future analysis to not make this article bigger than it is.

There are two ways to proceed:

- Writing a pure Python script.
- Writing an IDA Python/IDC script.

As I have already used IDA Python/IDC previously, I will write pure Python scripts (using Jupyter notebook as environment), which makes easier to readers to adapt it and debug any issues.

Anyway, I will be showing a first version of script, but it is not the final one. Why? Because the presented results will demand further attention of us, but I always like to show the true order of issues during the analysis and reproduce what I did to move forward. I have commented the script for helping the reader, but I still need to explain decisions of few lines:

```
import binascii
import pefile
import base64
from Crypto.Cipher import ARC4

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

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

# This routine extracts and returns data from .rdata section,
# .rdata section address and file image base.
def extract_data(filename):
    pe=pefile.PE(filename)
    for section in pe.sections:
        if '.rdata' 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 .rdata section section.
def calc_offsets(end_addr, start_addr):
    data_offset = int(end_addr,16) - int(start_addr,16)
    return data_offset

# Print decrypted data.
def print_data(data):
    for item in data.split(b'\x00\x00'):
        final_data = item.replace(b'\x00', b'').decode('utf-8')
        print(final_data)

# encrypted_string_addr: start address of the encrypted strings
def show_data(encrypted_string_addr):

    # Next two lines extracts .rdata section's information.
    filename = r"C:\Users\Administrador\Desktop\MAS\MAS_7\mas_7_unpacked.bin"
    data_encoded_extracted, sect_address, file_image_base = extract_data(filename)

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

```

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

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

    # Splits key and encrypted data, and reverse the extracted key
    key_orig = encrypted_data[0:48]
    key_reversed = key_orig[0:48][::-1]
    data_orig = encrypted_data[48:]

    # These commented lines were initially added to
    # confirm whether the script was really working as the
    # expected.
    #key_hex = binascii.b2a_hex(key_orig)
    #print(key_orig)
    #print(key_hex)

    # Finally, it calls the routine for decrypting data.
    decrypted = data_decryptor(key_reversed,data_orig)

    # Print the decoded string.
    print_data(decrypted)
```

```python
def main():

    print("\nDecrypted Data:")
    print(16 * "-" + "\n")

    data_location = ['0x59C560','0x59C5C0','0x59AF80','0x59C660','0x59A060',\
                     '0x59A6C0','0x59B840','0x59B980']
    for addr in data_location:
        print("\n[*] Data at: %s\n" % addr)
        show_data(addr)

if __name__ == '__main__':

    main( )
```

[Figure 79]: first version of the decryption script

As readers can realize, the script is basically composed by two parts, where the first one is a series of support routines, and the second part is the main routine. Based on collected references to **ab_w_rc4_0** routine, I searched for such referred addresses in the **.rdata section** that points to encrypted data block and created an array with all these addresses. For while that is an appropriate solution, but we will change it soon. There are other references to encrypted data blocks, but I am not concerned with it. The output of the script is shown in the next page:

```
Decrypted Data:
----------------


[*] Data at: 0x59C560

Connection: CloseTransfer-Encoding


[*] Data at: 0x59C5C0

GET
POST



[*] Data at: 0x59AF80

Starting path: ShellFolderS9OYlNFUvY5N1RDSpi8BgH6SgS8gPIcU;zwTHMB1SiSgHnmlqIchyvE
q6lSioc0XHE4rT4eCydGgyrIipLBzPItrelc82jktTbqgPlT4yGq<autoElevate>truetruefalse<Ta
sk xmlns="http://schemas.microsoft.com/windows/2004/02/mit/task" version="1.3"><R
egistrationInfo></RegistrationInfo><Triggers><LogonTrigger><Enabled>true</Enabled
><UserId></UserId></LogonTrigger><TimeTrigger><Repetition><Interval>PT30M</Interv
al><StopAtDurationEnd>false</StopAtDurationEnd></Repetition><StartBoundary>2020-0
1-01T00:00:00</StartBoundary><Enabled>true</Enabled></TimeTrigger></Triggers><Pri
ncipals><Principal id="Author"><LogonType>InteractiveToken</LogonType><RunLevel>L
eastPrivilege</RunLevel><UserId></UserId></Principal></Principals><Settings><Mult
ipleInstancesPolicy>IgnoreNew</MultipleInstancesPolicy><DisallowStartIfOnBatterie
s>false</DisallowStartIfOnBatteries><StopIfGoingOnBatteries>false</StopIfGoingOnB
atteries><AllowHardTerminate>false</AllowHardTerminate><StartWhenAvailable>false
</StartWhenAvailable><RunOnlyIfNetworkAvailable>false</RunOnlyIfNetworkAvailable>
<IdleSettings><StopOnIdleEnd>true</StopOnIdleEnd><RestartOnIdle>false</RestartOnI
dle></IdleSettings><AllowStartOnDemand>true</AllowStartOnDemand><Enabled></Enable
d><Hidden>true</Hidden><RunOnlyIfIdle>false</RunOnlyIfIdle><DisallowStartOnRemote
AppSession>false</DisallowStartOnRemoteAppSession><UseUnifiedSchedulingEngine>fal
se</UseUnifiedSchedulingEngine><WakeToRun>false</WakeToRun><ExecutionTimeLimit>PT
0S</ExecutionTimeLimit><Priority>7</Priority></Settings><Actions Context="Autho
r"><Exec><Command></Command></Exec></Actions></Task><Author>$(@%systemroot%\syste
m32\wininet.dll,-16000)</Author>

[*] Data at: 0x59C660
```


```
o6ijqKOoo6ijqKOoo6ijqKqqqqqqqqqrKqyqp+fn5+1o6Okqqq6qpaqqKrDw5aWt67WvaPFo6Ofw5yqq
qyqvwN/EX8BfwE/AQGQgn2XWVlZWVl/WVlgfX9/WVlZWVlZWVlZWVlZmoh9WVBQUFBZWVlZYZRhnllZhV
mSo2BgWVlZWVlZWVlZWVmfAQMBBAPVA8wBvAPwEBAQEFBQUFAUICAgIAEBAQHEAhAAAAAAAQHAwhARAgM
RAwMEAAAUAAIAAMbIAgICAgAA/////wAAAP/KAQEBAAYABADAwgEBAwH//wEAA8TExgMBAQH/AwMDyEAA
CgAEAAAAAH8AMwEAAAAAAAAD/v///AAAAAAcAAP8AAAAAAAAAAAAAAAAAAP//AAAAvwAAAAAAAAAAAAfwAA/
0pKSkpLUkpKSkpPTEpKSkpKSkpKVUVASkpKRVlNRkpdSkpKSkpKSkpKSkpKSkphY2dOSkprbUpKRW1KSk
RFSkoAAAACDQYGBgYOAAAAAAYGBgAGBgIGAAoKBwcGAgUFAgIAAAQEBAQAAAAOBQYGBgEGAAAIABAAGAA
gACgAMACAAYIBhgD2z/4/qwCwALEAswC6+LsAwADBAMe/Yv8Ajf8AxP8Axf8A///rAf8OEggAEwkAFggA
FwkAKwkArv8Hsv8AtP8Atf8AwwEAx/+/5wgA8AIA
```

```
[*] Data at: 0x59A060
```


DQYAAFUci+xRTlNWwfIz21fs+TsD83R6OF4QqxLbDulGBNRJKHFTA1ADyP/RiDkoOYcIdC6LgQzrE700A
40EmIM4g3TB/zDIl3D5w4BLeer2dvxq4fi3HNkdKGQWosCHO8N0G4kcRfhoGYAijeD8UN8IOgdq/4ldrE
g4iFZTpzIBX15bycNTkYOQJLhNY1ogV4nq8ICUZjkCdAMHM8Dp/gN/YTtyPIPygT5QRQoYiXX0GJHlYNQ
0g2Xs4QPoMBRQDOBTamQE5jAnvd6bDejrIsg0PotdYPzkhcB0bxNiWHFQ/2dF4H02AmsBfMqF2w+Eo46f
ahRKCJlRXBGL8J3QOwx1IErkrsrxEuwbtGHpa79xGEYMBgiI7BAfi0X0AV4E/3BUzHXwoOOXTItAJsxAU
jymBwPDiQYiWDRKHq4a3sAPt0gUM9IAg8QMjUwBGGYHO1AGc2OmWYRyCxxD+InBDIXJdCCMFPwDVFHuS3
cEM03waNzSULNT3kQbCOsd3dlIOIhUfhbo8xtQM3ZKi0CzZEDpSOnDMSg5DHyjIAXk5k3faIYrQTSeoUZ
8h0KDuKQNA+VWQJRV+HR4azCAoB0DwutQW7HwgMqJBk3cjUgIDGHs0ASD6eT3wTH+/wJ2O5mK7LQJ2NGB
cOJu8CDJ+hFzdTA6VXv0f+FsD4FsAWwRZfTwZoPuAnNz0VA5wnJhyET4A0AxX/6Cs3WfhHyEME4ExhlF/
wHK9CCFMe+cIYBphBLBFY3SKNzNRXRDuA+ZFQFM+oP2EJE/MQdA/IovhPQtIM2pG2wWNlDsFc2hLEao0O
9qHL5hNgQ34SE/Tt4iYCbrEC4r8Mg7PNOTUDxVSezTcx0UiP++UG1YTPAlFHQKuwoDXfTfOZ7rB5sYFfv
Lv9zJR6C88Cn9vUIJeQclDA+zBqqRBagCCZR1pCZ4SYsuwO4ZwcGDPBhgPguQj/lAOHXQ61QNInCKqgOz
M9AUQdn5kHjwQ/lVsgmVtFACbC2AbH0IMuSqrkxLpLGQT5JMQjw/D7STNRSY8AwR98IVAj11bnhlJROAQ
w9AdAzA2BvAFYPgAqcU6wgMhAdAlD5BHyB0BsH9BFkO5sHwDamCXIqLUfQJSfiTFp/Pdl4LMgn0xA+i+w
SaPMyb5D6DYGQOyMp8SAIVwSiHMDkSD4xwq8ZQMLgogHQnFldqARMRCSTQid6K2kcFEYHqiNz3Hdob0jk
GI9GJFkcsC8ZRDgGLolteX+Meu9bIz+hIafuAgpHr7uYrXRi9HnUIjYYM7bqYmTP/CgJElsYECFHommBC
RVwId5NE/rYoX999Mgb4BOwB6P+WLKwlKUQQIwh1WnqezCVTW5ZFNKBmVKFt/LLaIkzMS9AdHyIwZFdAJ
CfVRB8NnmgUQAds+iJYiImGURxevPwoFB9uUCxxG4sTiajO6Beqwmh6O8c/dB9VIYoVFVcmFUB5IRQ+KE
gyo1IQYVb6DDm+GCF1ZKw8HDJoEIZUoAaOOJnzrGEWA1o+ECPkCLuCgKJWDVOS27kE0UsNFIl1J3fQvsh
0SvBNesKIgAA=


```
[*] Data at: 0x59A6C0
```


jQcAAEgHhdIPhMUNAuSJXDMkCAp0qSBX7IPs6oAxehwmi9qGBvl0GgpgAgZKCEUzP8BE/HYoANJMA8lB/
9HGYUNFg3sQwHQ3e4sfGP/IBmPweFcxSxAqPPEDCjsWDIb/l9gEt+7OG3nmTEp0NDaPMCGEyDDATSae5I
WMdCVkZDgkOA1MjUQM76tU2EBBueCAwj+Dyf+NkSj/IVxooj3NkMOAlBiHQRr/0BIGtjAKg7bOXAXEIF/
DzDDHVVZXOUFUdunftQXMi+xBz0C4TVpDO+oBBuFmOQJ0B0HL6bJAVGNyPOgDBvKBPlBFGjF16lpGMEbm
6/xQ3BTfQZHfKFiXdE2hllX+qP+SRccNkSh1wzgRICAwoQqU5WBMmHV0UHSJVPDQEsoQfUEK/8NAAIP7A
ny7TQeF9nSLSSKMJJDDugjBgUSNQhhCPLCZ2/gkCHVnJ9ZFuJLwOQAGUp33QipoI+lDz6wrcDoIRBt4GB
CjEMOIwxzm2GZUuNUGUM4nkIvwY0U8obDb3APGoYIHql8j74MGVwgPt8sUPyRVAGZEO1gGc3LC4HQBKIP
nRvijiaAMRTl+HmMiGl78z0QGgxTaGRRWBIDVlWzLVZhKXggk6yAzCkY4rAx+H0BOUfz05yRUyo83NMUI
CiaYzG2+tvfjJAbHSYPGKOlhBt75fJckIvB0TXfGSCtlSHMhhSGVFF/LDDmYtB10f3UokLAWCgPT62TTH
8iKQgSgE1IHIYPoSYdxy3Kp/qH6dkJBmJ4COMHoHwyB4f3gWoP4A3RiDgoKdWBcY8EUTgEEoTIHE0YpD7
G0gSDDMsIChOmPnv77u9H6DTvBcr6xlcbQo0jWVHSWqoUAmX8IQLZzAX6I+1KMlCigH3vgfItjkBm6FFA
LCt+OReCze3YPm1dASaNrSBA6GV38ZDAWznpNylbPKBPQMRTwSIhThBpPgkcQTWkdI+BPGJkBgf/BTGPJ
MhOePpThAzYiuCvrGhY/TCnUZX9M0Cqk13312FRUK8WJR134DxhNAzTD/xMOQTkFO3QQDoRd2Qh1SbOGB
vfrCiGMv27zBvl3dDDyK84WuRQBiYAGUNAqTcE8C3UFSgJ8OAKEaug2vO/WQDLCdEgyQRTDCDyBdcdAit
P8gVXFFCYWTBsk8BTwhPCSpybrKIvl2FYYBC2K8cCnTUjrU8cXyNsC80CE9ki6i9fBMszocvuGdOlk/Aq
RKCQbo6M/6APCGPMoaDDoI0zpwJI3vJODrkw7LAJcATxBv5DH9wPJDQIGlIIPgKtBI1THEUA8dA8K2EUb
ggqD4dk6OsEV6wwPkl+e6iNYIEpAm3SYIaGBBEQKXhSJdAmDaLrprRYRC6T0JJfwQfhJC8V2lH/3vyx77
wnv9cijIKTXRnBialv0or//xko3Rv07O/EVjF3HaSsBX1godNPDQK2ACMS6AZlYksZWFRPICLFWkMko1u
3sgVw6j4FID0TKurjIVYjag8ZHHAEwhMcGnCSApPuw7xxAQV+dXh1dnVz7dX3DLlVT9W63Y/FYsIEQBKA
mZTihBVDDIghM92R02SS/BDtiTVBDyAn0iXwkSX5AwI13Sv4M/TjkTPow3jsoCW4oXD8iqQE4SZ751lL1
QJjx/5NUUBlN3R49qRj6lcDKZJ8oohogEMhGYKS/H4MuF8Y4nYMqKTCQEaH7PqpYDaIUURm/oYAiklUoB
oKSQbmWQK1eOgIuqBCJg0MwYvaLICuGSxA0s30hjKarhOjz+ZRyzYAoHXirKD4YySoYSshceGgXshqIGd
ZBN+z4AYO7KOKGdZ5DQzAhyImDGDFTQJIZu8yTs1OgkFAfQUq8JrPdSRAuVRUH3mmI1eFF11EyFaZkel3
UMlJ01OiIJFiGKcOSWxiFAA==

```
[*] Data at: 0x59B840

9QAAAFUci+yDhyxTVnN1CNBGOFcAagdZUYlN/I07Tjy8UABFCP9WFIPEfgzy7PpqZyAM/GwICMwU/ywYd
TPk2w+LRN4krARmAgyNTfBRCNzfZDZmdub4oCN03n0gMn3cBuAE8Aj4BvT/FlpwaHAQ4UNJVyoK9EpuCH
JhO991A5gO6Gkki90u/uJcAVCTekAZJQR8Qwr7AnyPiF7OYOh5gomVjdSDIOSAxTADyjtXASAiegzqdmY
0DhCUYOQMHF9eWxzJwgQwAA==


[*] Data at: 0x59B980

kAEAAEADVVNWV0FUl1BIjWwkddEcgeyYNwLDi9kCBklwQbgHgA2NU3hMHolFdzoPTW//bSge7GdJPYPOP
ExvJ4dVb8eLwUG5IEAoiUQk8YZCMDP2Ln17dEUBZgOLB4Nl58M2COs4CkL3IX8c7xvHVOsEfGd4ZFBADT
hUAmhNZ/erfyMzyXmWMyQwIecoutYM3uoNT/gSoeJkGRNOTUN87zZWwP+DUShOoBGIaBAQhDrjYE3v9v7
FDlV/UbCsB11vhfZgD0TYSSmJC8Q2MXPPFAiqxpEPxxCeg8wPhU7+AoTdg7op4iDe6NhRTO1dv0CyRSaJ
g9lDYD87D3nZHDxURswbUAcJQJAoUVzNBjFcUCHcKKEFIFTQXahV0nExGIRLaLIYEiADTQdiDlqBxERyO
UFeHVxfBltdwwA=
```

[Figure 80]: output from the first version of the script

As readers can realize, the first three outputs are in clear text, but the next ones are not. Of course, these output are encoded in **Base 64** and our first measure would be decode it, but there is another issue to be handled, which I didn't show you yet.

There is an important detail to comment: one **lines 29 to 32**, I defined a routine named **print_data( )**. This routine was necessary because without including it we would see the following output for the second address of the list within the **main( )** routine:

```
Decrypted Data:
----------------

b'G\x00E\x00T\x00\x00\x00P\x00O\x00S\x00T\x00\x00\x00\x00'
```

[Figure 81]: output for the second address without any manipulation.

We see that:

- The output is an array of bytes.
- There is a **\x00 (blue)** prefixing each letter (I only marked once to not pollute the image)
- There is a **\x00\x00 (red)** separating each word

Thus, on **lines 30, 31** and **32** we:

- separated words
- removed all **\x00** prefixes.
- converted to string.

Although readers might not remember, we already saw similar manipulations (not equal) in the third article of this series (**MAS 3**), which showed details about the Emotet reversing. About the **Base64** strings, we can decode them now, but before doing it, we have to pick up one of address of the array (for example, the last one), and examine code around the call to **ab_w_rc4_0 routine** (**sub_59214C**), which is using such address:

```
129  LODWORD(v135) = v35;
130  sub_58F584(v111, 0);
131  sub_58F584(v110, 0);
132  ab_w_rc4_0(v142, &unk_59B980, 0);
133  sub_58F6C0(v111, v142[0]);
134  ab_ww_RtlFreeHeap_0(v142);
135  v36 = sub_58F6A8(v111);
136  sub_58F828(v110, v36);
137  v37 = sub_58F4BC(v111, 0);
138  v38 = sub_58F4BC(v110, 0);
139  sub_5878B4(v5, v37, v38);
140  v128 = sub_58F4CC(v110);
141  v134 = sub_58AFE8(v95, 0x20000000, v128 + 2);
142  v127 = v39;
143  LODWORD(v40) = sub_58AFE8(v95, 0x80000000, 0x82);
```

[Figure 82]: piece of code within sub_589088

The **ab_w_rc4_0** (**sub_59214C**) routine is called on **line 133**. On the next line the **sub_58F6C0( )** is called, and part of its content is the following:

```
      do
      {
        ++v8;
        if ( v2 >= v17 )
          break;
        v9 = v20[v8 - 1];
        v2 = &v20[v8];
        v10 = v9 - 0x41;
        if ( (v9 - 0x41) <= 0x19 )
          goto LABEL_12;
        if ( (v9 - 0x61) <= 0x19 )
        {
          v10 = v9 - 0x47;
          goto LABEL_12;
        }
        if ( (v9 - 0x30) <= 9 )
        {
          v10 = v9 + 4;
LABEL_12:
          if ( v10 == 0xFFFFFFFF )
            goto LABEL_19;
          goto LABEL_18;
        }
        if ( v9 == '+' )
        {
          v10 = '>';
        }
        else
        {
          if ( v9 != '/' )
          {
```

[Figure 83]: part of sub_59214C subroutine

As we already had discovered by inference of decrypted strings, this **sub_59214C routine** handle the **Base64** decoding and there is a well-known library to decode such strings, so it is not a problem.

Returning to **sub_589088 routine**, on **line 139** there is a call to **sub_5878B4** routine. Moving inside this routine, we have:

```
1 void __usercall sub_5878B4(int a1@<ebp>, _BYTE *a2, _BYTE *a3)
2 {
3   // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]
4
5   BYTE4(v5) = 0x80;
6   v6 = 0;
7 LABEL_2:
8   *a3++ = *a2++;
9   LOBYTE(v6) = 2;
10  while ( 1 )
11  {
12    v5 = sub_587937(SBYTE4(v5), a2);
13    if ( !v14 )
14      goto LABEL_2;
15    v7 = sub_587937(SBYTE4(v5), a2);
16    if ( !v14 )
17    {
18      v11 = sub_587943(v7, a2);
19      v13 = v12 - v6;
20      if ( !v13 )
21      {
22        v5 = sub_587941(v11, a2);
23 LABEL_20:
24        LODWORD(v5) = a1;
25        LOBYTE(v6) = 1;
26        goto LABEL_21;
27      }
28      LODWORD(v11) = (v13 - 1) << 8;
29      LOBYTE(v11) = *a2++;
30      v5 = sub_587941(v11, a2);
31      if ( v5 < 0x7D00 )
32      {
33        if ( BYTE1(v5) >= 5u )
34        {
```

[Figure 84]: part of sub_5878B4 subroutine

This routine doesn't provide us many clues about what is really happening, but there is a subtle evidence that readers can use: the constant **0x7D00** on **line 31**.

This is a well-known constant used by **APLib decompression method** and, even readers didn't know about, a quick search on Google would confirm that my statement is correct.

Now we understand the sequence of events:

- **encrypted code → RC4 → Base64 → APLib**

There are two interesting projects that are enabled to handle APLib code:

- https://github.com/CERT-Polska/malduck **(by CERT Poland)**
- https://github.com/snemes/aplib **(by Sandor Nemes)**

Therefore, we must change the first version of this script to manage the following points:

- Decoding **Base64** (when necessary)
- Decompressing the result from Base64 using **APLib** (when necessary)

I prefer using **malduck package** because it offers support to a series of algorithms such as *AES, Blowfish, Camellia, ChaCha20, DES/DES3, Salsa20, Serpent, Rabbit, RC4, XOR, RSA, aPLib, gzip, lznt1, SHA1, MD5, SHA256*, and other useful features.

To install malduck, run: **pip install malduck**

The next version of our script **covers three possibilities after decrypting data blocks using RC4 algorithm: plain text information, decoded Base64 information, and decoded Base64 followed by decompressed aPLib information**. I am going to comment about few lines to make sure that everything is clear to readers:

```
import binascii
import pefile
import base64
import struct
import os
from Crypto.Cipher import ARC4
from malduck import aplib

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

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

# This routine extracts and returns data from .rdata section,
# .rdata section address and file image base.
def extract_data(filename):
    pe=pefile.PE(filename)
    for section in pe.sections:
        if '.rdata' 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 .rdata section section.
def calc_offsets(end_addr, start_addr):
    data_offset = int(end_addr,16) - int(start_addr,16)
    return data_offset

```

```python
# Print decrypted data.
def print_data(data):
    for item in data.split(b'\x00\x00'):
        final_data = item.replace(b'\x00', b'').decode('utf-8')
        print(final_data)

# encrypted_string_addr: start address of the encrypted strings
def show_data(encrypted_string_addr, base = 0):

    # Next two lines extracts .rdata section's information.
    filename = r"C:\Users\Administrador\Desktop\MAS\MAS_7\mas_7_unpacked.bin"
    data_encoded_extracted, sect_address, file_image_base = extract_data(filename)

    # Next three lines find the RVA of the .rdata section, the absolute address
    # of the .rdata 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' in data_encoded_extracted[data_offset:]):
        d_off = (data_encoded_extracted[int(data_offset):]).index(b'\x00\x00')

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

    # Splits key and encrypted data, and invert the extracted key
    key_orig = encrypted_data[0:48]
    key_reversed = key_orig[0:48][::-1]
    data_orig = encrypted_data[48:]

    # These commented lines were initially added to
    # confirm whether the script was really working as the
    # expected.
    #key_hex = binascii.b2a_hex(key_orig)
    #print(key_orig)
    #print(key_hex)

    # Finally, it calls the routine for decrypting data.
    decrypted = data_decryptor(key_reversed,data_orig)

    # Print the decoded string.
    if (base == 1):
        decrypted_base64 = base64.b64decode(decrypted)
        print(decrypted_base64)
        data_len = struct.unpack('<I', decrypted_base64[:4])
        data_corpus = aplib(decrypted_base64[4:])
        print("\n[+] Decompressed Data Length: %d" % data_len)
```

```
        #print(data_corpus)
        return(data_corpus)
    elif (base == 2):
        decrypted_base64 = base64.b64decode(decrypted)
        print(decrypted_base64)
    else:
        print_data(decrypted)
```

```
def main():

    counter = 1
    dir_path = r"C:\Users\Administrador\Desktop\MAS\MAS_7\Saved_Files"
    returned_corpus = b''

    print("\nDecrypted Data:")
    print(16 * "-" + "\n")

    data_clear_text = ['0x59C560','0x59C5C0','0x59AF80']

    data_base64_aplib = ['0x59A060','0x59A6C0',\
                         '0x59B840','0x59B980']

    data_pure_base64 = ['0x59C660']

    for addr in data_clear_text:
        print("\n[*] Clear data at: %s\n" % addr)
        show_data(addr)

    for addr in data_base64_aplib:
        print("\n[*] Compressed aPLib data at: %s\n" % addr)
        returned_data = show_data(addr, 1)
        bin_filename = ("filedata_%s.bin" % counter)
        filename = os.path.join(dir_path, bin_filename)
        if (not os.path.exists(filename)):
            open(filename, 'wb').write(returned_data)
            print("[+] Decompressed aPLib data saved as %s\n" % filename)
            counter = counter + 1
        else:
            counter = counter + 1

    for addr in data_pure_base64:
        print("\n\n[*] Decoded base64 data at: %s\n" % addr)
        show_data(addr, 2)

if __name__ == '__main__':

    main( )
```

[Figure 85]: Second version of the decryption and decoding script

```
Decrypted Data:
----------------


[*] Clear data at: 0x59C560

Connection: CloseTransfer-Encoding


[*] Clear data at: 0x59C5C0

GET
POST



[*] Clear data at: 0x59AF80

Starting path: ShellFolderS9OYlNFUvY5N1RDSpi8BgH6SgS8gPIcU;zwTHMB1SiSgHnmlqIchyvEq6lSioc0XHE4rT4eCydGgyr
IipLBzPItrelc82jktTbqgPlT4yGq<autoElevate>truetruefalse<Task xmlns="http://schemas.microsoft.com/window
s/2004/02/mit/task" version="1.3"><RegistrationInfo></RegistrationInfo><Triggers><LogonTrigger><Enabled>
true</Enabled><UserId></UserId></LogonTrigger><TimeTrigger><Repetition><Interval>PT30M</Interval><StopAt
DurationEnd>false</StopAtDurationEnd></Repetition><StartBoundary>2020-01-01T00:00:00</StartBoundary><Ena
bled>true</Enabled></TimeTrigger></Triggers><Principals><Principal id="Author"><LogonType>InteractiveTok
en</LogonType><RunLevel>LeastPrivilege</RunLevel><UserId></UserId></Principal></Principals><Settings><Mu
ltipleInstancesPolicy>IgnoreNew</MultipleInstancesPolicy><DisallowStartIfOnBatteries>false</DisallowStar
tIfOnBatteries><StopIfGoingOnBatteries>false</StopIfGoingOnBatteries><AllowHardTerminate>false</AllowHar
dTerminate><StartWhenAvailable>false</StartWhenAvailable><RunOnlyIfNetworkAvailable>false</RunOnlyIfNetw
orkAvailable><IdleSettings><StopOnIdleEnd>true</StopOnIdleEnd><RestartOnIdle>false</RestartOnIdle></Idle
Settings><AllowStartOnDemand>true</AllowStartOnDemand><Enabled></Enabled><Hidden>true</Hidden><RunOnlyIf
Idle>false</RunOnlyIfIdle><DisallowStartOnRemoteAppSession>false</DisallowStartOnRemoteAppSession><UseUn
ifiedSchedulingEngine>false</UseUnifiedSchedulingEngine><WakeToRun>false</WakeToRun><ExecutionTimeLimit>
PT0S</ExecutionTimeLimit><Priority>7</Priority></Settings><Actions Context="Author"><Exec><Command></Com
mand></Exec></Actions></Task><Author>$(@%systemroot%\system32\wininet.dll,-16000)</Author>
[*] Compressed aPLib data at: 0x59A060

b'\r\x06\x00\x00U\x1c\x8b\xecQNSV\xc1\xf23\xdbW\xec\xf9;\x03\xf3tz8^\x10\xab\x12\xdb\x0e\xe9F\x04\xd4I(q
S\x03P\x03\xc8\xff\xd1\x889(9\x87\x08t.\x8b\x81\x0c\xeb\x13\xbd4\x03\x8d\x04\x98\x838\x83t\xc1\xff0\xc8
\x97p\xf9\xc3\x80Ky\xea\xf6v\xfcj\xe1\xf8\xb7\x1c\xd9\x1d(d\x16\xa2\xc0\x87;\xc3t\x1b\x89\x1cE\xf8h\x19
\x80"\x8d\xe0\xfcP\xdf\x08:\x07j\xff\x89]\xacH8\x88VS\xa72\x01_^[\xc9\xc3S\x91\x83\x90$\xb8McZ W\x89\xea
\xf0\x80\x94f9\x02t\x03\x073\xc0\xe9\xfe\x03\x7fa;r<\x83\xf2\x81>PE\n\x18\x89u\xf4\x18\x91\xe5`\xd44\x83
e\xec\xe1\x03\xe80\x14P\x0c\xe0Sjd\x04\xe60\'\xbd\xde\x9b\r\xe8\xeb"\xc84>\x8b]`\xfc\xe4\x85\xc0to\x13bX
qP\xffgE\xe0}6\x02k\x01|\xca\x85\xdb\x0f\x84\xa3\x8e\x9fj\x14J\x08\x99Q\\\x11\x8b\xf0\x9d\xd0;\x0cu J\xe
4\xae\xca\xf1\x12\xec\x1b\xb4a\xe9k\xbfq\x18F\x0c\x06\x08\x88\xec\x10\x1f\x8bE\xf4\x01^\x04\xffpT\xccu\x
f0\xa0\xe3\x97L\x8b@&\xcc@R<\xa6\x07\x03\xc3\x89\x06"X4J\x1e\xae\x1a\xde\xc0\x0f\xb7H\x143\xd2\x00\x83\x
c4\x0c\x8dL\x01\x18f\x07;P\x06sc\xa6Y\x84r\x0b\x1cC\xf8\x89\xc1\x0c\x85\xc9t \x8c\x14\xfc\x03TQ\xeeKw\x0
43M\xf0h\xdc\xd2P\xb3S\xdeD\x1b\x08\xeb\x1d\xdd\xd9H8\x88T~\x16\xe8\xf3\x1bP3vJ\x8b@\xb3d@\xe9H\xe9\xc31
(9\x0c|\xa3 \x05\xe4\xe6M\xdfh\x86+A4\x9e\xa1F|\x87B\x83\xb8\xa4\r\x03\xe5V@\x94U\xf8txk0\x80\xa0\x1d\x0
3\xc2\xebP[\xb1\xf0\x80\xca\x89\x06M\xdc\x8dH\x08\x0ca\xec\xd0\x04\x83\xe9\xe4\xf7\xc11\xfe\xff\x02v;\x9
9\x8a\xec\xb4\t\xd8\xd1\x81p\xe2n\xf0 \xc9\xfa\x11su0:U{\xf4\x7f\xe1l\x0f\x81l\x01l\x11e\xf4\xf0f\x83\xe
e\x02ss\xd1P9\xc2ra\xc8D\xf8\x03@1_\xfe\x82\xb3u\x9f\x84|\x840N\x04\xc6\x19E\xff\x01\xca\xf4 \x851\xef\x
9c!\x80i\x84\x12\xc1\x15\x8d\xd2(\xdc\xcdEtC\xb8\x0f\x99\x15\x01L\xfa\x83\xf6\x10\x91?1\x07@\xfc\x8a/\x8
4\xf4- \xcd\xa9\x1bl\x166P\xec\x15\xcd\xa1,F\xa8\xd0\xefj\x1c\xbea6\x047\xe1!?N\xde"`&\xeb\x10.+\xf0\xc
8;<\xd3\x93P<UI\xec\xd3s\x1d\x14\x88\xff\xbePmXL\xf0%\x14t\n\xbb\n\x03]\xf4\xdf9\x9e\xeb\x07\x9b\x18\x15
\xfb\xcb\xbf\xdc\xc9G\xa0\xbc\xf0)\xfd\xbdB\ty\x07%\x0c\x0f\xb3\x06\xaa\x91\x05\xa8\x02\t\x94u\xa4&xI\x8
b.\xc0\xee\x19\xc1\xc1\x83<\x18`>\x0b\x90\x8f\xf9@8u\xd0\xebT\r"p\x8a\xaa\x03\xb33\xd0\x14A\xd9\xf9\x90x
\xf0C\xf9U\xb2\t\x95\xb4P\x02l-\x80l}\x082\xe4\xaa\xaeLK\xa4\xb1\x90O\x92LB<?\x0f\xb4\x935\x14\x98\xf0\x
0c\x11\xf7\xc2\x15\x02=unxe%\x13\x80C\x0f@t\x0c\xc0\xd8\x1b\xc0\x15\x83\xe0\x02\xa7\x14\xeb\x08\x0c\x84
\x07@\x94>A\x1f t\x06\xc1\xfd\x04Y\x0e\xe6\xc1\xf0\r\xa9\x82\\\x8a\x8bQ\xf4\tI\xf8\x93\x16\x9f\xcfv^\x0b
```

```
\x07@\x94>A\x1f t\x06\xc1\xfd\x04Y\x0e\xe6\xc1\xf0\r\xa9\x82\\\x8a\x8bQ\xf4\tI\xf8\x93\x16\x9f\xcfv^\x0b
2\t\xf4\xc4\x0f\xa2\xfb\x04\x9a<\xcc\x9b\xe4>\x83`d\x0e\xc8\xca|H\x02\x15\xc1(\x8709\x12\x0f\x8cp\xab\xc
6P0\xb8(\x80t\'\x16Wj\x01\x13\x11\t$\xd0\x89\xde\x8a\xdaG\x05\x11\x81\xea\x88\xdc\xf7\x1d\xda\x1b\xd29\x
06#\xd1\x89\x16G,\x0b\xc6Q\x0e\x01\x8b\xa2[^_\xe3\x1e\xbb\xd6\xc8\xcf\xe8Hi\xfb\x80\x82\x91\xeb\xee\xe6
+]\x18\xbd\x1eu\x08\x8d\x86\x0c\xed\xba\x98\x993\xff\n\x02D\x96\xc6\x04\x08Q\xe8\x9a`BE\\\x08w\x93D\xfe
\xb6(_\xdf}2\x06\xf8\x04\xec\x01\xe8\xff\x96,\xac%)D\x10#\x08uZz\x9e\xcc%S[\x96E4\xa0fT\xa1m\xfc\xb2\xd
a"L\xccK\xd0\x1d\x1f"0dW@$\'\xd5D\x1f\r\x9eh\x14@\x07l\xfa"X\x88\x89\x86Q\x1c^\xbc\xfc(\x14\x1fnP,q\x1b
\x8b\x13\x89\xa8\xce\xe8\x17\xaa\xc2hz;\xc7?t\x1fU!\x8a\x15\x15W&\x15@y!\x14>(H2\xa3R\x10aV\xfa\x0c9\xbe
\x18!ud\xac<\x1c2h\x10\x86T\xa0\x06\x8e8\x99\xf3\xaca\x16\x03Z>\x10#\xe4\x08\xbb\x82\x80\xa2V\rS\x92\xdb
\xb9\x04\xd1K\r\x14\x89u\'w\xd0\xbe\xc8tJ\xf0Mz\xc2\x88\x80\x00'
```


```
[+] Decompressed Data Length: 1549
[+] Decompressed aPLib data saved as C:\Users\Administrador\Desktop\MAS\MAS_7\Saved_Files\filedata_1.bin


[*] Compressed aPLib data at: 0x59A6C0
```


```
b'\x8d\x07\x00\x00H\x07\x85\xd2\x0f\x84\xc5\r\x02\xe4\x89\\3$\x08\nt\xa9 W\xec\x83\xec\xea\x801z\x1c&\x8
b\xda\x86\x06\xf9t\x1a\n`\x02\x06J\x08E3?\xc0D\xfcv(\x00\xd2L\x03\xc9A\xff\xd1\xc6aCE\x83{\x10\xc0t7{\x8
b\x1f\x18\xff\xc8\x06c\xf0xW1K\x10*<\xf1\x03\n;\x16\x0c\x86\xff\x97\xd8\x04\xb7\xee\xce\x1by\xe6LJt46\x8
f0!\x84\xc80\xc0M&\x9e\xe4\x85\x8ct%dd8$8\rL\x8dD\x0c\xef\xabT\xd8@A\xb9\xe0\x80\xc2?\x83\xc9\xff\x8d\x9
1(\xff!\\h\xa2=\xcd\x90\xc3\x80\x94\x18\x87A\x1a\xff\xd0\x12\x06\xb60\n\x83\xb6\xce\\\x05\xc4 _\xc3\xcc0
\xc7UVW9ATv\xe9\xdf\xb5\x05\xcc\x8b\xecA\xcf@\xb8MZC;\xea\x01\x06\xe1f9\x02t\x07A\xcb\xe9\xb2@Tcr<\xe8\x
03\x06\xf2\x81>PE\x1a1u\xeaZF0F\xe6\xeb\xfcP\xdc\x14\xdfA\x91\xdf(X\x97tM\xa1\x96U\xfe\xa8\xff\x92E\xc7
\r\x91(u\xc38\x11  0\xa1\n\x94\xe5`L\x98utPt\x89T\xf0\xd0\x12\xca\x10}A\n\xff\xc3@\x00\x83\xfb\x02|\xbbM
\x07\x85\xf6t\x8bI"\x8c$\x90\xc3\xba\x08\xc1\x81D\x8dB\x18B<\xb0\x99\xdb\xf8$\x08ug\'\xd6E\xb8\x92\xf09
\x00\x06R\x9d\xf7B*h#\xe9C\xcf\xac+p:\x08D\x1bx\x18\x10\xa3\x10\xc3\x88\xc3\x1c\xe6\xd8fT\xb8\xd5\x06P\x
ce\'\x90\x8b\xf0cE<\xa1\xb0\xdb\xdc\x03\xc6\xa1\x82\x07\xaa_#\xef\x83\x06W\x08\x0f\xb7\xcb\x14?$U\x00fD;
X\x06sr\xc2\xe0t\x01(\x83\xe7F\xf8\xa3\x89\xa0\x0cE9~\x1ec"\x1a^\xfc\xcfD\x06\x83\x14\xda\x19\x14V\x04\x
80\xd5\x95l\xcbU\x98J^\x08$\xeb 3\nF8\xac\x0c~\x1f@NQ\xfc\xf4\xe7$T\xca\x8f74\xc5\x08\n&\x98\xccm\xbe\xb
6\xf7\xe3$\x06\xc7I\x83\xc6(\xe9a\x06\xde\xf9|\x97$"\xf0tMw\xc6H+eHs!\x85!\x95\x14_\xcb\x0c9\x98\xb4\x1d
t\x7fu(\x90\xb0\x16\n\x03\xd3\xebd\xd3\x1f\xc8\x8aB\x04\xa0\x13R\x07!\x83\xe8I\x87q\xcbr\xa9\xfe\xa1\xfa
vBA\x98\x9e\x028\xc1\xe8\x1f\x0c\x81\xe1\xfd\xe0Z\x83\xf8\x03tb\x0e\n\nu`\\c\xc1\x14N\x01\x04\xa12\x07\x
13F)\x0f\xb1\xb4\x81 \xc32\xc2\x02\x84\xe9\x8f\x9e\xfe\xfb\xbb\xd1\xfa\r;\xc1r\xbe\xb1\x95\xc6\xd0\xa3H
\xd6Tt\x96\xaa\x85\x00\x99\x7f\x08@\xb6s\x01~\x88\xfbR\x8c\x94(\xa0\x1f{\xe0|\x8bc\x90\x19\xba\x14P\x0b
\n\xdf\x8eE\xe0\xb3{v\x0f\x9bW@I\xa3kH\x10:\x19]\xfcd0\x16\xcezM\xcaV\xcf(\x13\xd01\x14\xf0H\x88S\x84\x1
aO\x82G\x10Mi\x1d#\xe0O\x18\x99\x01\x81\xff\xc1Lc\xc92\x13\x9e>\x94\xe1\x036"\xb8+\xeb\x1a\x16?L)\xd4e\x
7fL\xd0*\xa4\xd7}\xf5\xd8TT+\xc5\x89G]\xf8\x0f\x18M\x034\xc3\xff\x13\x0eA9\x05;t\x10\x0e\x84]\xd9\x08uI
\xb3\x86\x06\xf7\xeb\n!\x8c\xbfn\xf3\x06\xf9wt0\xf2+\xce\x16\xb9\x14\x01\x89\x80\x06P\xd0*M\xc1<\x0bu\x0
5J\x02|8\x02\x84j\xe86\xbc\xef\xd6@2\xc2tH2A\x14\xc3\x08<\x81u\xc7@\x8a\xd3\xfc\x81U\xc5\x14&\x16L\x1b
$\xf0\x14\xf0\x84\xf0\x92\xa7&\xeb(\x8b\xe5\xd8V\x18\x04-\x8a\xf1\xc0\xa7MH\xebS\xc7\x17\xc8\xdb\x02\xf3
@\x84\xf6H\xba\x8b\xd7\xc12\xcc\xe8r\xfb\x86t\xe9d\xfc\n\x91($\x1b\xa3\xa3?\xe8\x03\xc2\x18\xf3(h0\xe8#L
\xe9\xc0\x927\xbc\x93\x83\xaeL;,\x02\\\x01<A\xbf\x90\xc7\xf7\x03\xc9\r\x02\x06\x94\x82\x0f\x80\xabA#T\xc
7\x11@<t\x0f\n\xd8E\x1b\x82\n\x83\xe1\xd9::\xc1\x15\xeb\x0c\x0f\x92_\x9e\xea#X J@\x9bt\x98!\xa1\x81\x04D
\n^\x14\x89t\t\x83h\xba\xe9\xad\x16\x11\x0b\xa4\xf4$\x97\xf0A\xf8I\x0b\xc5v\x94\x7f\xf7\xbf,{\xef\t\xef
\xf5\xc8\xa3 \xa4\xd7Fpbj[\xf4\xa2\xbf\xff\xc6J7F\xfd;;\xf1\x15\x8c]\xc7i+\x01_X(t\xd3\xc3@\xad\x80\x08
\xc4\xba\x01\x99X\x92\xc6V\x15\x13\xc8\x08\xb1V\x90\xc9(\xd6\xed\xec\x81\\:\x8f\x81H\x0fD\xca\xba\xb8\xc
8U\x88\xda\x83\xc6G\x1c\x010\x84\xc7\x06\x9c$\x80\xa4\xfb\xb0\xef\x1c@A_\x9d^\x1d]\x9d\\\xfbu}\xc3.US\xf
5n\xb7c\xf1X\xb0\x81\x10\x04\xa0&e8\xa1\x05P\xc3"\x08L\xf7dt\xd9$\xbf\x04;bMPC\xc8\t\xf4\x89|$I~@\xc0\x8
dwJ\xfe\x0c\xfd8\xe4L\xfa0\xde;(\tn(\\?"\xa9\x018I\x9e\xf9\xd6R\xf5@\x98\xf1\xff\x93TP\x19M\xdd\x1e=\xa9
\x18\xfa\x95\xc0\xcad\x9f(\xa2\x1a \x10\xc8F`\xa4\xbf\x1f\x83.\x17\xc68\x9d\x83*)0\x90\x11\xa1\xfb>\xaaX
\r\xa2\x14Q\x19\xbf\xa1\x80"\x92U(\x06\x82\x92A\xb9\x96@\xad^:\x02.\xa8\x10\x89\x83C0b\xf6\x8b +\x86K\x1
04\xb3}!\x8c\xa6\xab\x84\xe8\xf3\xf9\x94r\xcd\x80(\x1dx\xab(>\x18\xc9*\x18J\xc8\\xh\x17\xb2\x1a\x88\x19
\xd6A7\xec\xf8\x01\x83\xbb(\xe2\x86u\x9eCC0!\xc8\x89\x83\x181S@\x92\x19\xbb\xcc\x93\xb3S\xa0\x90P\x1fAJ
\xbc&\xb3\xddI\x10.U\x15\x07\xdei\x88\xd5\xe1E\xd7Q2\x15\xa6dz]\xd42Rt\xd4\xe8\xc8$X\xc6)\xc4\x12[\x18\x
c5\x00'
```

```
[+] Decompressed Data Length: 1933
[+] Decompressed aPLib data saved as C:\Users\Administrador\Desktop\MAS\MAS_7\Saved_Files\filedata_2.bin


[*] Compressed aPLib data at: 0x59B840

b'\xf5\x00\x00\x00U\x1c\x8b\xec\x83\x87,SVsu\x08\xd0F8W\x00j\x07YQ\x89M\xfc\x8d;N<\xbcP\x00E\x08\xffV\x1
4\x83\xc4~\x0c\xf2\xec\xfajg \x0c\xfcl\x08\x08\xcc\x14\xff,\x18u3\xe4\xdb\x0f\x8bD\xde$\xac\x04f\x02\x0c
\x8dM\xf0Q\x08\xdc\xdfd6fv\xe6\xf8\xa0#t\xde} 2}\xdc\x06\xe0\x04\xf0\x08\xf8\x06\xf4\xff\x16Zphp\x10\xe1
CIW*\n\xf4Jn\x08ra;\xdfu\x03\x98\x0e\xe8i$\x8b\xdd.\xfe\xe2\\\x01P\x93z@\x19%\x04|C\n\xfb\x02|\x8f\x88^
\xce`\xe8y\x82\x89\x95\x8d\xd4\x83 \xe4\x80\xc50\x03\xca;W\x01 "z\x0c\xeavf4\x0e\x10\x94`\xe4\x0c\x1c_^
[\x1c\xc9\xc2\x040\x00'

[+] Decompressed Data Length: 245
[+] Decompressed aPLib data saved as C:\Users\Administrador\Desktop\MAS\MAS_7\Saved_Files\filedata_3.bin


[*] Compressed aPLib data at: 0x59B980

b"\x90\x01\x00\x00@\x03USVWAT\x97PH\x8dl$u\xd1\x1c\x81\xec\x987\x02\xc3\x8b\xd9\x02\x06IpA\xb8\x07\x80\r
\x8dSxL\x1e\x89Ew:\x0fMo\xffm(\x1e\xecgI=\x83\xce<Lo'\x87Uo\xc7\x8b\xc1A\xb9 @(\x89D$\xf1\x86B03\xf6.}{t
E\x01f\x03\x8b\x07\x83e\xe7\xc36\x08\xeb8\nB\xf7!\x7f\x1c\xef\x1b\xc7T\xeb\x04|gxdP@\r8T\x02hMg\xf7\xab
\x7f#3\xc9y\x963$0!\xe7(\xba\xd6\x0c\xde\xea\rO\xf8\x12\xa1\xe2d\x19\x13NMC|\xef6V\xc0\xff\x83Q(N\xa0\x1
1\x88h\x10\x10\x84:\xe3`M\xef\xf6\xfe\xc5\x0eU\x7fQ\xb0\xac\x07]o\x85\xf6`\x0fD\xd8I)\x89\x0b\xc461s\xcf
\x14\x08\xaa\xc6\x91\x0f\xc7\x10\x9e\x83\xcc\x0f\x85N\xfe\x02\x84\xdd\x83\xba)\xe2 \xde\xe8\xd8QL\xed]\x
bf@\xb2E&\x89\x83\xd9C`?;\x0fy\xd9\x1c<TF\xcc\x1bP\x07\t@\x90(Q\\\xcd\x061\\P!\xdc(\xa1\x05 T\xd0]\xa8U
\xd2q1\x18\x84Kh\xb2\x18\x12 \x03M\x07b\x0eZ\x81\xc4Dr9A^\x1d\\_\x06[]\xc3\x00"

[+] Decompressed Data Length: 400
[+] Decompressed aPLib data saved as C:\Users\Administrador\Desktop\MAS\MAS_7\Saved_Files\filedata_4.bin



[*] Decoded base64 data at: 0x59C660

b'\xa3\xa8\xa3\xa8\xa3\xa8\xa3\xa8\xa3\xa8\xa3\xa8\xa3\xa8\xa3\xa8\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xaa\xac\x
aa\xb2\xaa\x9f\x9f\x9f\x9f\xb5\xa3\xa3\xa4\xaa\xaa\xba\xaa\x96\xaa\xa8\xaa\xc3\xc3\x96\x96\xb7\xae\xd6\x
bd\xa3\xc5\xa3\xa3\x9f\xc3\x9c\xaa\xaa\xac\xaa\xbf\x03\x7f\x11\x7f\x01\x7f\x01?\x01\x01\x90\x82}\x97YYYY
Y\x7fYY`}\x7f\x7fYYYYYYYYYYYY\x9a\x88}YPPPPYYYYa\x94a\x9eYY\x85Y\x92\xa3``YYYYYYYYYYY\x9f\x01\x03\x01\x0
4\x03\xd5\x03\xcc\x01\xbc\x03\xf0\x10\x10\x10\x10PPPP\x14    \x01\x01\x01\x01\xc4\x02\x10\x00\x00\x00\x0
0\x01\x01\xc0\xc2\x10\x11\x02\x03\x11\x03\x03\x04\x00\x00\x14\x00\x02\x00\x00\xc6\xc8\x02\x02\x02\x02\x0
0\x00\xff\xff\xff\xff\x00\x00\x00\xff\xca\x01\x01\x01\x00\x06\x00\x04\x00\xc0\xc2\x01\x01\x03\x01\xff\xf
f\x01\x00\x03\xc4\xc4\xc6\x03\x01\x01\x01\xff\x03\x03\x03\xc8@\x00\n\x00\x04\x00\x00\x00\x00\x7f\x003\x0
1\x00\x00\x00\x00\x00\x00\xff\xbf\xff\xff\x00\x00\x00\x00\x07\x00\x00\xff\x00\x00\x00\x00\x00\x00\x00\x0
0\x00\x00\x00\x00\x00\x00\xff\xff\x00\x00\x00\xbf\x00\x00\x00\x00\x00\x00\x00\x00\x7f\x00\x00\xffJJJJKRJ
JJJOLJJJJJJJJUE@JJJEYMFJ]JJJJJJJJJJJJJacgNJJkmJJEmJJDEJJ\x00\x00\x00\x02\r\x06\x06\x06\x06\x0e\x00\x00
\x00\x00\x06\x06\x06\x00\x06\x06\x02\x06\x00\n\n\x07\x07\x06\x02\x05\x05\x02\x02\x00\x00\x04\x04\x04\x04
\x00\x00\x00\x0e\x05\x06\x06\x06\x01\x06\x00\x00\x08\x00\x10\x00\x18\x00 \x00(\x000\x00\x80\x01\x82\x01
\x86\x00\xf6\xcf\xfe?\xab\x00\xb0\x00\xb1\x00\xb3\x00\xba\xf8\xbb\x00\xc0\x00\xc1\x00\xc7\xbfb\xff\x00\x
8d\xff\x00\xc4\xff\x00\xc5\xff\x00\xff\xff\xeb\x01\xff\x0e\x12\x08\x00\x13\t\x00\x16\x08\x00\x17\t\x00+
\t\x00\xae\xff\x07\xb2\xff\x00\xb4\xff\x00\xb5\xff\x00\xc3\x01\x00\xc7\xff\xbf\xe7\x08\x00\xf0\x02\x00'
```

[Figure 86]: Output from the second version of the decryption and decoding script

The script presents relevant changes when compared to the previous version, but basically these new lines of code continue our work from where we stopped by decoding **Base64** data and, mainly, decompressing blocks of **aPLib compressed data**.

Therefore, necessary comments follow below:

- I kept **commented code from lines 66 to 68** just in case readers need to check the extracted key in bytes and hexadecimal.
- From **line 74 to 86**, I structured the script to take in account three scenarios, as already commented: plain text information, decoded Base64 data, and decode Base64 data followed by decompressed aPLib data.
- On **line 75**, the script decodes the Base64 data that comes from the RC4 decryption.
- On **line 80**, I kept commented **print** instruction to show the block of data after having executed the aPLib decompression.
- The compressed aPLib data has the following format: **[uncompressed data size] [compressed data]**. It is not only valid in this malware sample, but other families using aPLib present the same pattern.
- On **lines 78 and 79,** I extracted the size and uncompressed data into two different variables, **data_len** and **data_corpus**, respectively. The used **aplib( ) function** comes from **malduck package**.
- On the **main( )** routine, I separated addresses in three different lists (**lines 10 to 15**) according to the respective scenario.
- The script saves the decompressed aPLib data into different files on disk. To avoid issues, a quick checking is performed before performing each write operation to the file system (**lines 26 to 29**).
- As I already had mentioned previously, I will not analyze any of four dumped files, but if readers have interest in doing it, there might be a shellcode there. ;)

As a confirmation of fact that the first four bytes of the compressed aPLib data are really the size of the uncompressed aPLib data, the output of a file listing after extraction has finished follows below:

```
C:\Users\Administrador\Desktop\MAS\MAS_7\Saved_Files>dir
 Volume in drive C is Windows
 Volume Serial Number is D45E-7379

 Directory of C:\Users\Administrador\Desktop\MAS\MAS_7\Saved_Files

01/03/2023  09:45 PM    <DIR>          .
01/03/2023  10:38 PM    <DIR>          ..
01/03/2023  09:45 PM             1,549 filedata_1.bin
01/03/2023  09:45 PM             1,933 filedata_2.bin
01/03/2023  09:45 PM               245 filedata_3.bin
01/03/2023  09:45 PM               400 filedata_4.bin
               4 File(s)          4,127 bytes
```

[Figure 87]: List of saved uncompressed files

This output confirms exactly what has been presented as output by our script in **Figure 86**.

The last goal is to **retrieve the C2 IP address list** used by this sample. Initially, readers might consider it would be a painful step , similar to other malware families that we already learned in this series, but you will realize that this is not the case, fortunately.

A good step for finding a possible list of IP addresses is by starting analysis from functions related to Internet and, as we saw when we managed API hash resolution, there are good candidates. Using one of these network related APIs, we can find the caller routine and, from there, getting close our objective.

The **sub_584AC0 routine** is an interesting initial point:

```
1 int *__fastcall sub_584AC0(int *a1, char *a2, char a3, char a4)
2 {
3   // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]
4 
5   v17 = a1;
6   var_0FFFFFFFF_1 = var_0FFFFFFFF;
7   v6 = ab_Token_Manipulation(0, var_0FFFFFFFF);
8   ab_GetEnvironmentStringsW(v19, a2, *(v6 + 0xB));
9   sub_58F584(v18, 0);
10  sub_58F584(v17, 0);
11  v7 = ab_handling_data_3() + 1;
12  ab_NtDelayExecution(0x3E8u, var_0FFFFFFFF_1);
13  v8 = 0;
14  if ( var_0FFFFFFFF_1 == 0xFFFFFFFF )
15    goto LABEL_10;
16  while ( 1 )
17  {
18    ab_InternetOpenW(v14, 0, 1, 0);
19    v14[7] = 1;
20    v16 = 0x12C;
21    v9 = sub_58EC3C(v7, var_0FFFFFFFF_1);
22    if ( ab_Internet_1(v14, v9) )
23    {
24      ab_Internet_2(v14, v20, v19);
25      sub_58F434(v18, v20);
26      ab_ww_RtlFreeHeap_1();
27      if ( !v14[0] && (v15 == 0xC8 || v15 == 0x194) )
28      {
29        if ( !a3 )
30          break;
31        ab_w_rc4(v21, v18, a4);
32        sub_58F434(v17, v21);
33        ab_ww_RtlFreeHeap_1();
34        if ( !sub_58F4D0(v17) )
35          break;
36      }
```

[Figure 88]: sub_584AC0 routine: partial listing

As shown above, I already had renamed few routines while I managed API hash resolving issues (check **Figure 40**), so it is clear that this **sub_584AC0** routine is invoking routines related to Internet such as **ab_InternetOpenW**, **ab_Internet_W**, **ab_Internet_1** and **ab_Internet_2**.

On **line 11** I initially renamed the **sub_585708** to **ab_handling_data_3** because I didn't want spend time analyzing it at that moment, but now it gets my attention due the fact that **sub_584AC0** routine invokes other routines that are associated to the Internet communication, and we know that these routines will need IP addresses, which will need to be retrieved from somewhere.

Moving inside the **ab_handling_data_3** routine, we have the following:

```
1 _DWORD *ab_handling_data_3()
2 {
3   // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]
4 
5   result = unk_59D264;
6   if ( !unk_59D264 )
7   {
8     result = ab_w_RtlAllocateHeap();
9     if ( result )
10    {
11      v1 = 0;
12      result[1] = 0;
13      result[2] = 0;
14      result[3] = 0;
15    }
16    else
17    {
18      result = 0;
19      v1 = 0;
20    }
21    unk_59D264 = result;
22    *result = *(&byte_59D020 + 2);
23    if ( *(&byte_59D020 + 0xB) )
24    {
25      do
26      {
27        v5 = 0;
28        v2 = &ab_encoded_data_3 + 6 * v1;
29        v3 = *v2;
30        v4 = *(v2 + 2);
31        sub_59871C(&v3, v6);
32        sub_58E644((unk_59D264 + 4), v6[0], *(unk_59D264 + 4));
33        ab_ww_RtlFreeHeap_0(v6);
34        ++v1;
35      }
36      while ( v1 < *(&byte_59D020 + 0xB) );
37      return unk_59D264;
38    }
39  }
40  return result;
41 }
```

[Figure 89]: sub_585708 routine

Apparently there isn't anything really useful here, but there are clues:

- On **line 28**, there is an arithmetic operation involving an address (named **ab_encoded_data_3**).
- At the same **line 28**, there is a different multiplication operation: **6 * v1**
- On **line 36**, another arithmetic operation is explicit: **[address + offset]**.
- References as: **[ptr_var + 4]**
- The same address reference being used twice: **&byte_59D020 + 2** and **&byte_59D020 + 0xB.**

The content of **ab_encoded_data_3** follows:

```
.data:0059D000 ; Segment type: Pure data
.data:0059D000 ; Segment permissions: Read/Write
.data:0059D000 _data           segment para public 'DATA' use32
.data:0059D000                 assume cs:_data
.data:0059D000                 ;org 59D000h
.data:0059D000 var_0FFFFFFFF   dd 0FFFFFFFFh           ; DATA XREF:
.data:0059D000                                         ; sub_584AC0+
.data:0059D004                 align 20h
.data:0059D020 byte_59D020     db 2Bh                  ; DATA XREF:
.data:0059D021                 db  22h ; "
.data:0059D022                 db 0D1h
.data:0059D023                 db  70h ; p
.data:0059D024 word_59D024     dw 56BEh                ; DATA XREF:
.data:0059D026 byte_59D026     db 49h                  ; DATA XREF:
.data:0059D027 byte_59D027     db 0                    ; DATA XREF:
.data:0059D028 byte_59D028     db 0                    ; DATA XREF:
.data:0059D028                                         ; DllRegister
.data:0059D029 byte_59D029     db 0B2h                 ; DATA XREF:
.data:0059D029                                         ; ab_OutputDe
.data:0059D02A byte_59D02A     db 1                    ; DATA XREF:
.data:0059D02A                                         ; ab_OutputDe
.data:0059D02B unk_59D02B      db    4                 ; DATA XREF:
.data:0059D02C ab_encoded_data_3 db 78h                ; DATA XREF:
.data:0059D02D                 db  32h ; 2
.data:0059D02E                 db  28h ; (
.data:0059D02F                 db 0B9h
.data:0059D030                 dw 1BBh
.data:0059D032                 db 8Bh
.data:0059D033                 db  3Bh ; ;
.data:0059D034                 db  0Eh
.data:0059D035                 db 0DFh
.data:0059D036                 db 0ECh
.data:0059D037                 db  1Fh
.data:0059D038                 db  79h ; y
.data:0059D039                 db  28h ; (
.data:0059D03A                 db  68h ; h
.data:0059D03B                 db 0D1h
.data:0059D03C                 db 0CAh
.data:0059D03D                 db  19h
.data:0059D03E                 db  8Bh
.data:0059D03F                 db 0A2h
.data:0059D040                 db  71h ; q
.data:0059D041                 db 0A9h
.data:0059D042                 db 51h
.data:0059D043                 db    2
```

[Figure 90]: first bytes from .data section

Observing the code in **Figure 89** and matching data being referenced on **Figure 90**, many points become clear and, eventually, they help us to conduct short renaming task on variables from **sub_585708.**

Readers should realize that:

- the address **0x0059D020** (**byte_59D020**) is used as reference on the code.
- one **line 23**, ***(&byte_59D020 + 0xB**) take us to the content stored at **0x0059D02B**, which seems to be the number of IP addresses (**0x4**) being contacted by the malware. This conclusion is also enforced by the condition used by **while instruction** on **line 36** (**v1 < *(byte_59D020 + 0xB)**) .
- The address **0x59D024** stores the **botnet id (22206).**
- one **line 28, &ab _encode_data_3 + 6 * v1** is clearly passing through each **IP:port** combination.
- As there are **four IP:port combinations** in a supposed C2 list, and each one takes 6 bytes, so from **0x0059D02C** address (**ab_encode_data_3**) the **next 24 bytes** represent all four combinations.

The sub_585708 routine containing few renamed variables (and incomplete yet) follows below:

```
1  _DWORD *ab_handling_data_3()
2  {
3    // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]
4
5    result = unk_59D264;
6    if ( !unk_59D264 )
7    {
8      result = ab_w_RtlAllocateHeap();
9      if ( result )
10     {
11       counter = 0;
12       result[1] = 0;
13       result[2] = 0;
14       result[3] = 0;
15     }
16     else
17     {
18       result = 0;
19       counter = 0;
20     }
21     unk_59D264 = result;
22     *result = *(&initial_data_offset + 2);
23     if ( *(&initial_data_offset + 0xB) )
24     {
25       do
26       {
27         v5 = 0;
28         next_ip = &ip_list_offset + 6 * counter;
29         ip_address = *next_ip;
30         port = *(next_ip + 2);
31         sub_59871C(&ip_address, v6);
32         sub_58E644((unk_59D264 + 4), v6[0], *(unk_59D264 + 4));
33         ab_ww_RtlFreeHeap_0(v6);
34         ++counter;
35       }
36       while ( counter < *(&initial_data_offset + 0xB) );
37       return unk_59D264;
38     }
39   }
40   return result;
41 }
```

[Figure 91]: sub_585708 routine with few comments

Based on conclusions so far, I wrote a Python script to extract the **botnet** and **IP:port** combinations:

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

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

# Extract bytes from .data section, and format IP addresses and ports.
def extract_C2():

    # Next two lines extracts .data section's information.
    filename = r"C:\Users\Administrador\Desktop\MAS\MAS_7\mas_7_unpacked.bin"
    data_encoded_extracted = extract_data(filename)

    # Initialize important offsets used to extract bytes associated with
    # IP addresses and ports.
    initial_data_offset = 0x20
    botnet_offset = initial_data_offset + 4
    ip_size_offset = 0x2b
    ip_list_offset = ip_size_offset + 1

    # Extract the number of IP addresses/ports and calculate the total size.
    num_ips = ord(data_encoded_extracted[ip_size_offset:(ip_size_offset)+1])
    ip_list_bytes = 6 * num_ips

    # This line extracts the encoded IP:port data bytes.
    ip_bytes = data_encoded_extracted[ip_list_offset:ip_list_offset \
                                      + ip_list_bytes]

    # This line extracts the botnet
    extracted_botnet = struct.unpack('<h', data_encoded_extracted \
                                     [botnet_offset:botnet_offset+2])[0]
    print("\n[*] BOTNET: %s" % extracted_botnet)

    # Extract IP addresses and respective ports, and format them.
    print("\n[+] C2 IP ADDRESS LIST:")
    print(24 * '-' + "\n")

    k = 0
    i = 0
```

```
    while (k < len(ip_bytes)):
        ip_item = ip_bytes [k:k+4]
        ip_port = ip_bytes [k+4:k+6][::-1]
        print("IP[%d]: %s" % (i,ipaddress.IPv4Address(ip_item)),end=':')
        print(int(binascii.hexlify(ip_port),16))
        k = k + 6
        i = i + 1

```

```
def main():

    extract_C2()

if __name__ == '__main__':

    main( )
```

```
[*] BOTNET: 22206

[+] C2 IP ADDRESS LIST:
------------------------

IP[0]: 120.50.40.185:443
IP[1]: 139.59.14.223:8172
IP[2]: 121.40.104.209:6602
IP[3]: 139.162.113.169:593
```

[Figure 92]: script to extract botnet, and C2 IP addresses

Comparing this output above against Triage's output (**Figure 2**) we have a perfect match!

8. Conclusion

This article presented new challenges when compared to previous articles of this series, but hopefully readers have learned and enjoyed the reading. Recently a professional (*Twitter: @bushuo12*) translated the three first articles of this series to **Chinese language** if you are interested in reading them:

- **(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 stay connected:

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

Keep reversing and I see you at next time!

Alexandre Borges