

# Malware Analysis Series (MAS): Article 7

by Alexandre Borges

release date: JANUARY/05/2023 | rev: A.1

## 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 payload 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

<https://exploitreversing.com>

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 ahead 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/winsiders/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/mnrkby> (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 87e2dad373f75f5c0a200821523aebe45f6f4103b51fb0155ed2bf060ec50b04 -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: S90YLNfUvY5N1RDSpi8BgH6SgS8gPIcU
  key: key
  value: zwTHMB1SiSgHnmlqIchyvEq6lSioc0XHE4rT4eCydGgyrIipLBzPItreIc82jktTbqgP1T4yGq
  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

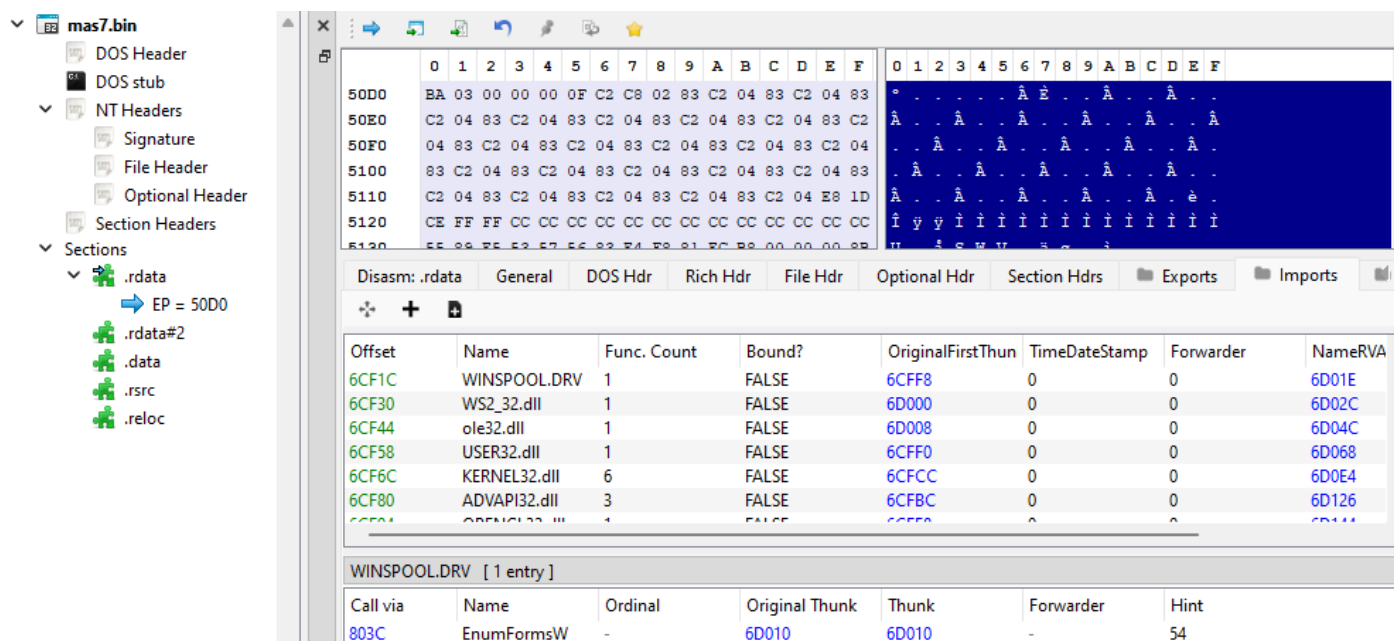
targets:
  family: dridex
  iocs:
    time.windows.com
    52.109.8.20
    8.8.8.8
    168.61.215.74
  md5: b22a00cefca58fa81234983b81de1fee
  score: 10
  sha1: 970fd83b12bd1919ab684723b69c8a90d1a36b9b
  sha256: 87e2dad373f75f5c0a200821523aebe45f6f4103b51fb0155ed2bf060ec50b04
  size: 479232bytes
  tags:
    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](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?	OriginalFirstThunk	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

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

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	mshtml.dll
1C9B0	Base	1	
1C9B4	NumberOfFunctions	5	
1C9B8	NumberOfNames	5	
1C9BC	AddressOfFunctions	1C9C8	
1C9C0	AddressOfNames	1C9F0	

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 (**mshtml.dll**) has been also viewed in other **Dridex samples** and, in a general way, there're few changes among them.

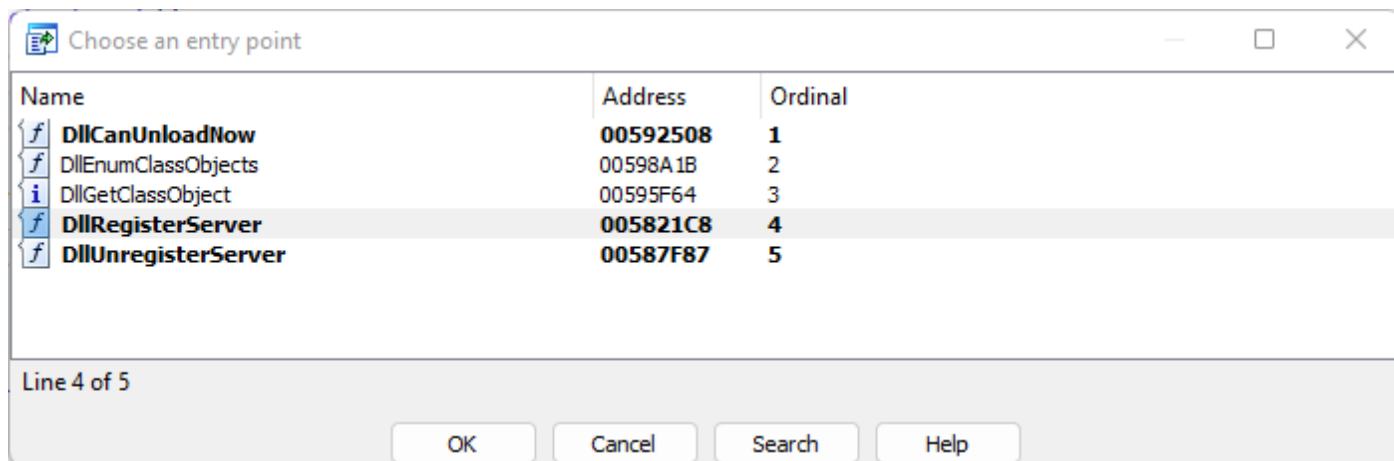
Further and valuable information can also be collected by using **capa.exe** (its standalone version) as shown below (the sample was renamed to **mas\_7\_unpacked.bin**):





## 7. Reversing

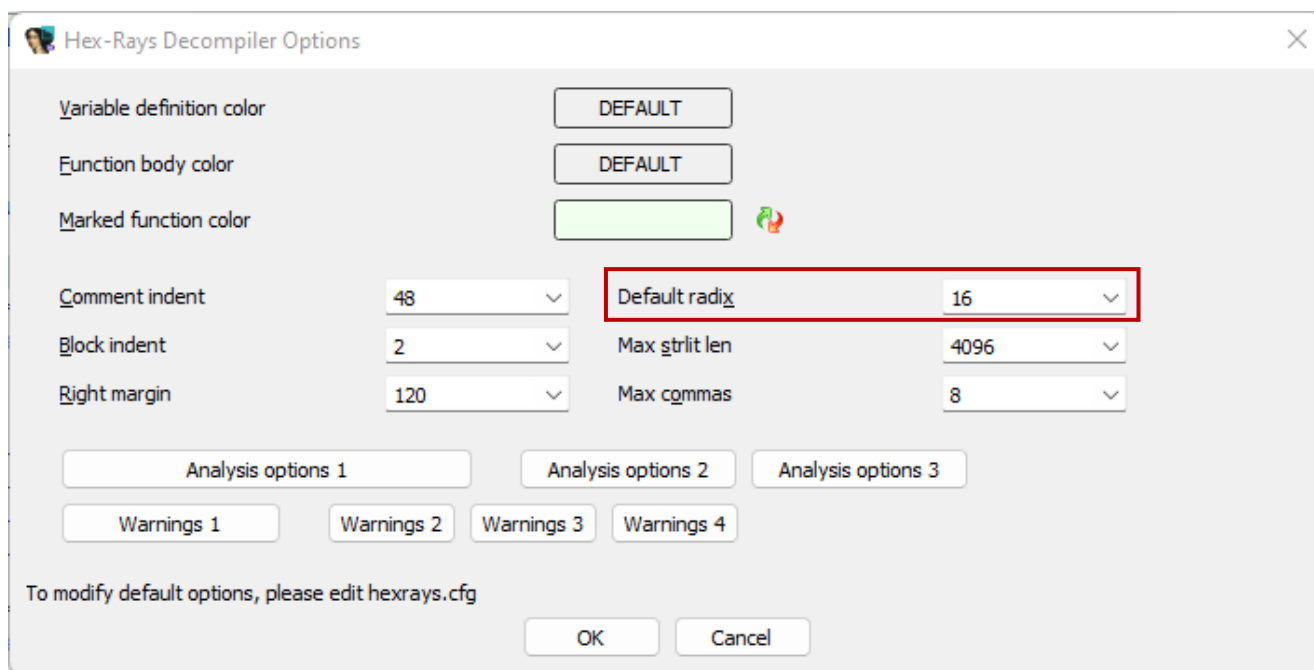
Dridex is a strange malware that demands our attention since beginning to understand what's really happening. As the unpacked sample is a DLL, so there are one or more exported functions and, of course, our first step is trying to understand which one we should or not to follow. Listing the possible 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:

```
1 HRESULT __fastcall sub_581000(int *a1, _DWORD *a2)
2 {
3     // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]
4
5     v2 = a1;
6     sub_58BF3C((int)v20, (char *)0x1D, a1[7]);
7     sub_58C494((int)v22, (int)v20[0]);
8     sub_58DFA4(v20);
9     if ( sub_58C9C0(v22, v2, 0) )
10    {
11        sub_58F584(v21, 0);
12        sub_58F584(v18, 0);
13        sub_58F584(v19, 0);
14        v5 = v2 + 3;
15        if ( !v2 )
16            v5 = 0;
17        if ( sub_58C280(v5, (int)v2) )
18            sub_58FCA4(v2, (int *)0x1FFFFFF);
19        v6 = v2 + 3;
20        if ( !v2 )
21            v6 = 0;
22        v7 = *v6;
23        sub_58CD04((char *)v23, 0);
24        sub_5812DC(v15, v7, (int)v23, a2);
25        if ( LOBYTE(v15[0]) )
26        {
27            v8 = sub_58FB5C(v2);
28            v9 = &unk_59A6C0;
29            if ( v8 == 0x20 )
30                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:

```
1 HANDLE __usercall sub_592518@eax(int a1@ecx, int *a2@esi)
2 {
3     HANDLE result; // eax
4
5     if ( !a1 )
6         return NtCurrentTeb()->ClientId.UniqueThread;
7     result = (HANDLE)sub_59306C(0x8E844D1E, 0x333A3BAF, a2, 0x8E844D1E, 0x8E844D1E);
8     if ( !result )
9         return 0;
10    __debugbreak();
11    __debugbreak();
12    return result;
13 }
```

[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+61p
.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 loc_592539:          ; CODE XREF: sub_592518+1A1j
.text:00592539          xor    eax, eax
.text:0059253B          jmp    short loc_592543
.text:0059253D ; -----
.text:0059253D loc_59253D:          ; CODE XREF: sub_592518+51j
.text:0059253D          mov    eax, large fs:24h
.text:00592543 loc_592543:          ; CODE XREF: sub_592518+1F1j
                    ; sub_592518+231j
.text:00592543          pop    ebx
.text:00592544          retn
.text:00592544 sub_592518      endp ; sp-analysis failed
.text:00592544 ; -----
.text:00592545          align 4
```

[Figure 11]: sub\_592518 Assembly code

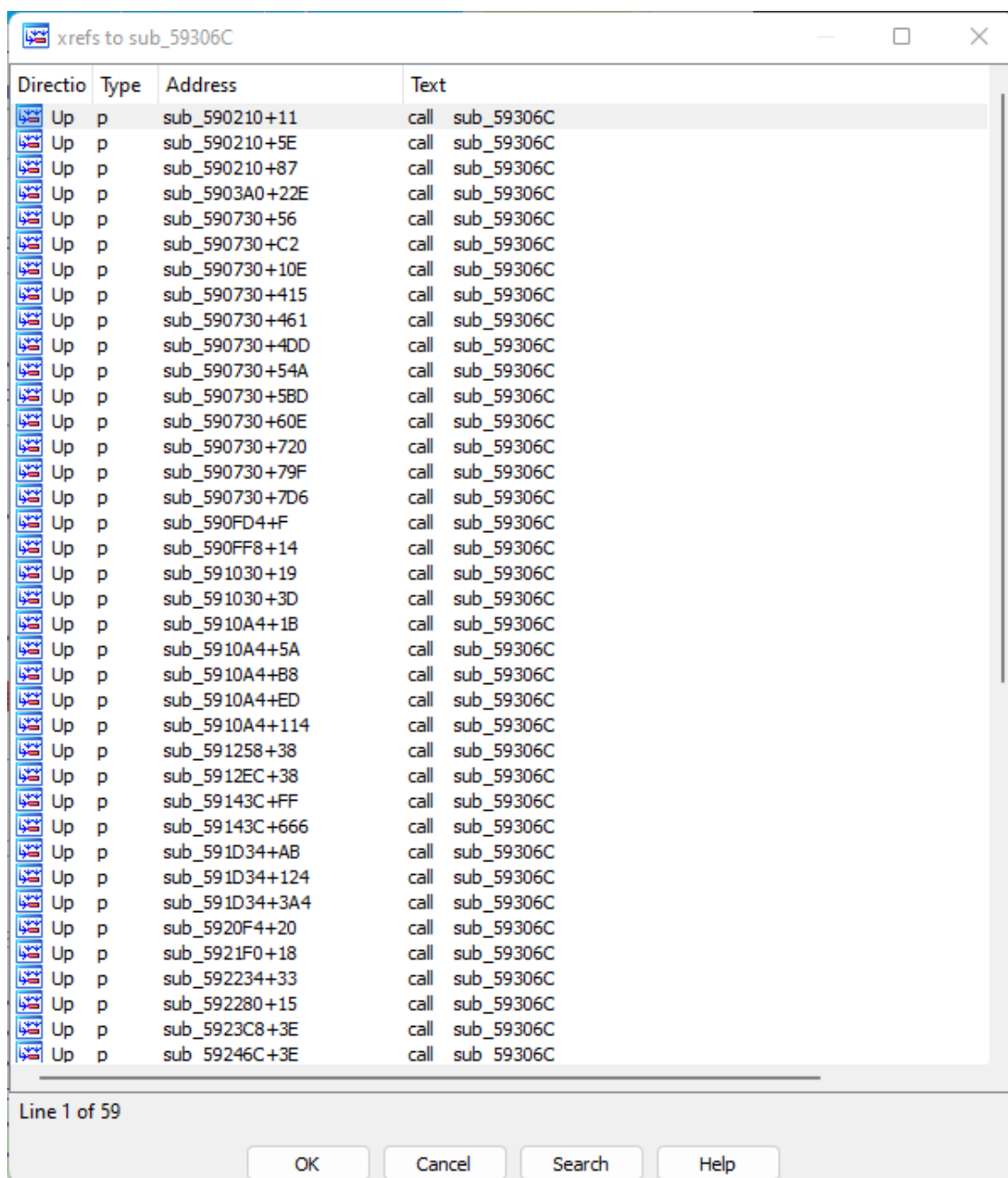
As mentioned in the previous page, the first point to comment is about the following subroutine call:

- **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 **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:



[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:

```
1 int __userpurge sub_59306C@<eax>(int a1@<eax>, int a2@<edx>, int *a3@<esi>, int a4, int a5)
2 {
3     int result; // eax
4     int v8; // eax
5
6     result = sub_590304(a2, 1);
7     if ( !result )
8     {
9         if ( a1 != 0x39731522
10            && ((v8 = sub_591D34(a1, a1)) != 0 || (unsigned __int8)sub_5903A0(a1, (int)a3) && (v8 = sub_591D34(a1, a1)) != 0 )
11            {
12             return sub_59143C(v8, a2, a3, v8, v8);
13         }
14         else
15         {
16             return 0;
17         }
18     }
19     return result;
20 }
```

[Figure 13]: sub\_59306C routine

Moving into the **sub\_590304** routine, we have the following scenario:

```
1 int __fastcall sub_590304(int a1, char a2)
2 {
3     int v3; // ecx
4     int v5; // edx
5     int result; // eax
6     int v7; // ebp
7     int v8; // edx
8
9     v3 = dword_59D208;
10    if ( dword_59D208 == 0xC55649E1 )
11    {
12        v3 = 0;
13        dword_59D208 = 0;
14    }
15    switch ( a1 )
16    {
17        case 0x1C6EF387:
18            v5 = dword_59D20C;
19            if ( dword_59D20C == 0xF0909E5B )
20                break;
21            return v5;
22        case 0x45B68B68:
23            v5 = dword_59D210;
24            if ( dword_59D210 != 0xB85BC9BE )
25                return v5;
26            break;
27        case 0x2EA96E2A:
28            v5 = dword_59D214;
29            if ( dword_59D214 != 0x4C71E88D )
30                return v5;
31            break;
32    }
```

[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:

```
1 void *__userpurge sub_591D34@<eax>(int a1@<eax>, int a2)
2 {
3     // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]
4
5     v3 = dword_59D21C;
6     if ( dword_59D21C == 0xF4C64A91 )
7     {
8         v3 = 0;
9         dword_59D21C = 0;
10    }
11    if ( a1 == 0x60A28C5C )
12    {
13        result = (void *)dword_59D220;
14        if ( dword_59D220 != 0x248DF160 )
15            return result;
16        ProcessEnvironmentBlock = NtCurrentTeb()->ProcessEnvironmentBlock;
17        if ( !v3 )
18            goto LABEL_29;
19    }
20    else
21    {
22        ProcessEnvironmentBlock = NtCurrentTeb()->ProcessEnvironmentBlock;
23        if ( a1 == 0x39731522 )
24            return ProcessEnvironmentBlock->ImageBaseAddress;
25 LABEL_13:
26        if ( !v3 )
27        {
28            if ( a1 == 0xA731522 )
29            {
30                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 brought 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:

```
1 int __usercall sub_590130@<eax>(int a1@<ecx>, _BYTE *a2@<esi>)
2 {
3     // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]
4
5     v7[1] = a2;
6     v7[0] = a2;
7     result = *(a1 + 0x3C);
8     if ( !result )
9     {
10        v4 = sub_590180(a1);
11        sub_58E06C(v4, v7);
12        v5 = v7[0];
13        v6 = sub_58E8A8(v7[0], 0x7FFFFFFF);
14        *(a1 + 0x3C) = sub_594FFC(v5, v6) ^ 0xE462D21C;
15        sub_58DFA4(v7);
16        return *(a1 + 0x3C);
17    }
18    return result;
19 }
```

[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_595088(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:

```
1 int __thiscall sub_5950B8(char *this)
2 {
3     // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]
4
5     v1 = 0;
6     v2 = _mm_loadu_si128(&xmmword_59BF30);
7     v3 = _mm_loadu_si128(&xmmword_59BF40);
8     v4 = _mm_loadu_si128(&xmmword_59BF50);
9     v5 = _mm_loadu_si128(&xmmword_59BF60);
10    do
11    {
12        v6 = v3;
13        LOBYTE(result) = 0;
14        do
15        {
16            LOBYTE(result) = result + 1;
17            v8 = _mm_cmpeq_epi32(_mm_and_si128(v4, v6), 0i64);
18            v9 = _mm_srli_epi32(v6, 1u);
19            v10 = _mm_xor_si128(v8, -1i64);
20            v6 = _mm_or_si128(
21                _mm_and_si128(_mm_xor_si128(v9, v5), v10),
22                _mm_andnot_si128(v10, v9));
23        }
24        while ( result < 8u );
25        *&this[4 * v1] = v6;
26        v1 += 4;
27        v3 = _mm_add_epi32(v3, v2);
28    }
29    while ( v1 < 0x100 );
30    return result;
31 }
```

[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+21r
·data:0059BF40 xmmword_59BF40 xmmword 3000000020000000100000000h
·data:0059BF40 ; DATA XREF: sub_5950B8+E1r
·data:0059BF50 xmmword_59BF50 xmmword 1000000010000000100000001h
·data:0059BF50 ; DATA XREF: sub_5950B8+161r
·data:0059BF60 xmmword_59BF60 xmmword 0EDB88320EDB88320EDB88320EDB88320h
·data:0059BF60 ; DATA XREF: sub_5950B8+1E1r
·data:0059BF70 byte_59BF70 db 4Eh ; DATA XREF: sub_59583C+691r
```

[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](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, <sup>[53]</sup> PNG, <sup>[54]</sup> ZMODEM, many others	0x04C11DB7	0xEDB88320	0xDB710641	0x82608EDB <sup>[15]</sup>
		$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$			

[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 same 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         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         loc_593656:                ; CODE XREF: sub_593628+23↑j
text:00593656         xor    eax, eax
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         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 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.

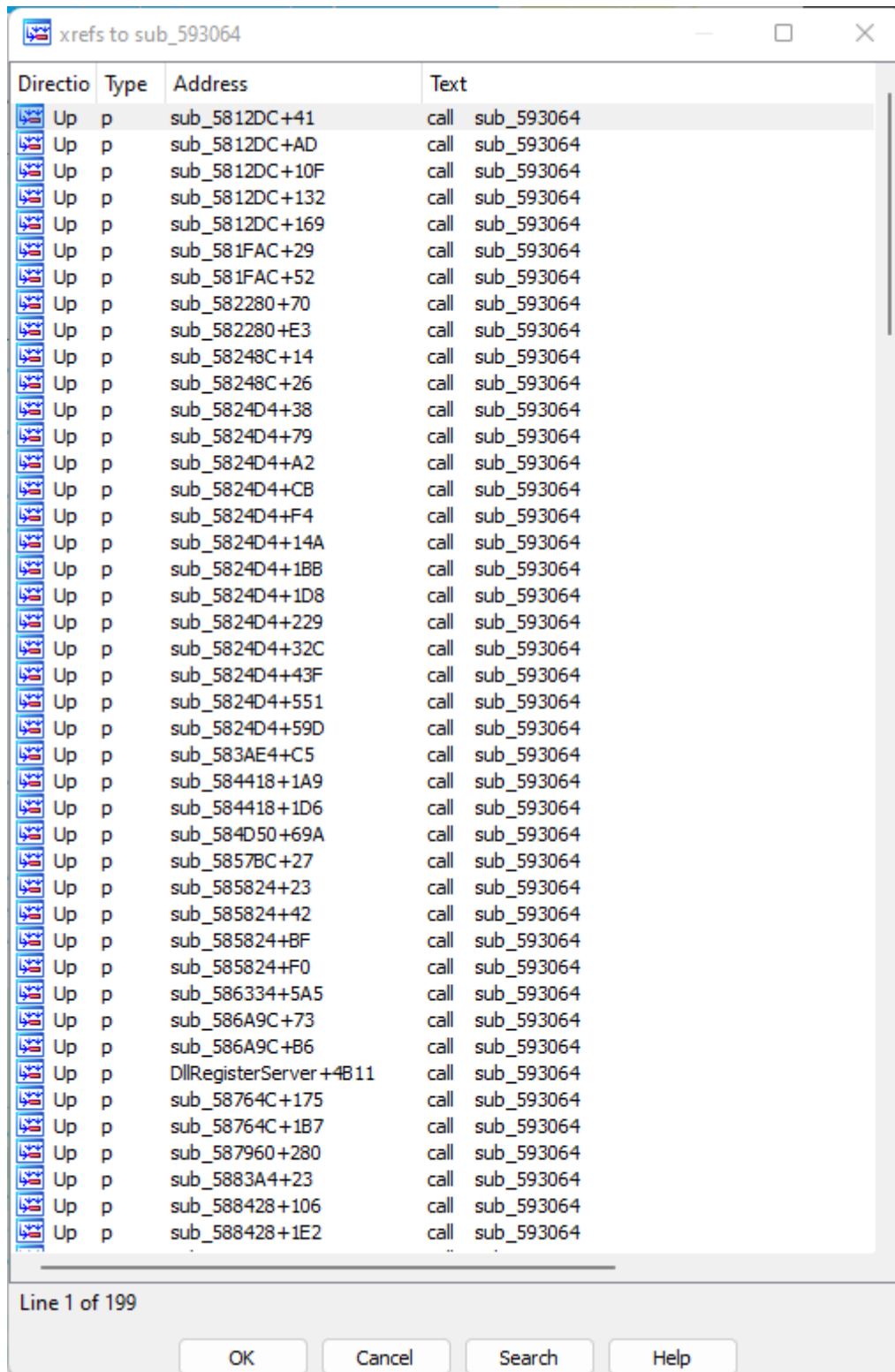
To handle hashing resolving, there are several possibilities and all of them are excellent solutions:

- **Shellcode Hashes from flare-ida (Mandiant):** [https://github.com/mandiant/flare-ida/blob/master/plugins/shellcode\\_hashes\\_search\\_plugin.py](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:

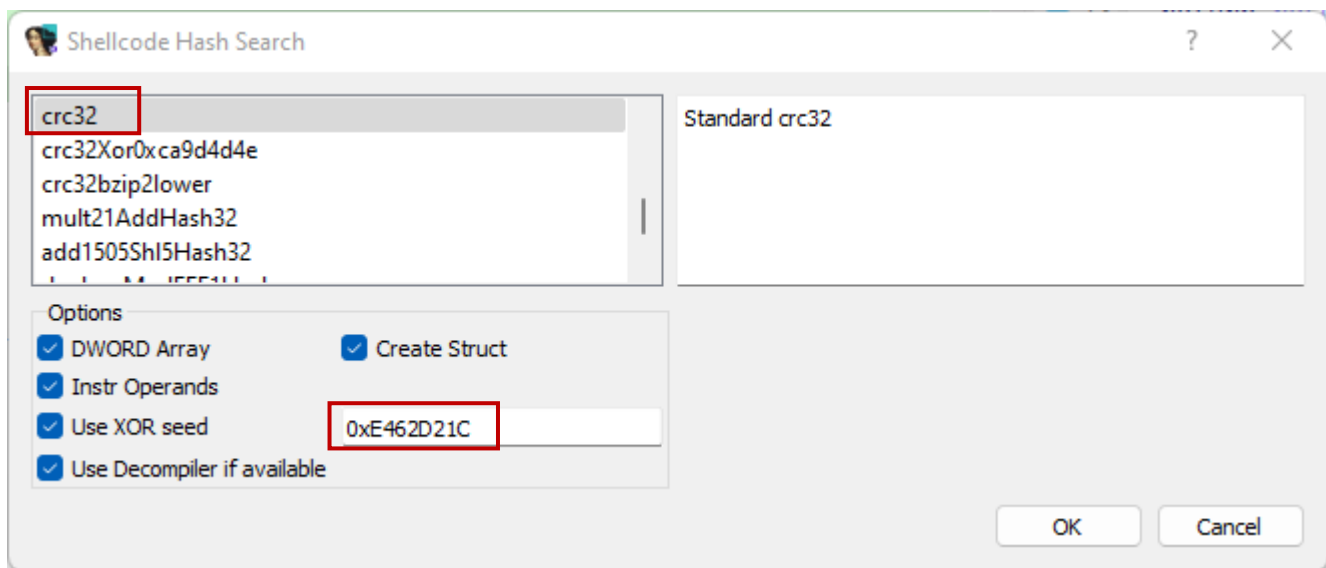
1. Clone the **flare-ida** repository: <https://github.com/mandiant/flare-ida>. In my case, I did it under *C:\github* directory.
2. Copy the plugin file (*C:\github\flare-ida\plugins\shellcode\_hashes\_search\_plugin.py*) to IDA Pro's plugin directory, which in my case is: *C:\Program Files\IDA Pro 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/](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:

```
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); // ntoskrnl.exe!b'RtlCreateHeap'
9         dword_59D22C = sub_593064(0x60A28C5C, 0x5E0AFAA3); // ntoskrnl.exe!b'RtlDestroyHeap'
10        if ( dword_59D228 == 0xA33C83E5 )
11            dword_59D228 = v1(2, 0, 0, 0, 0, 0);
12    }
13    result = sub_593064(0x60A28C5C, 0x45B68B68); // ntoskrnl.exe!b'RtlAllocateHeap'
14    if ( !result )
15        return 0;
16    __debugbreak();
17    __debugbreak();
18    return result;
19 }
```

[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      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      loc_593656:                ; CODE XREF: sub_593628+23↑j
.text:00593656      xor     eax, eax
.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      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

<https://exploitreversing.com>

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:

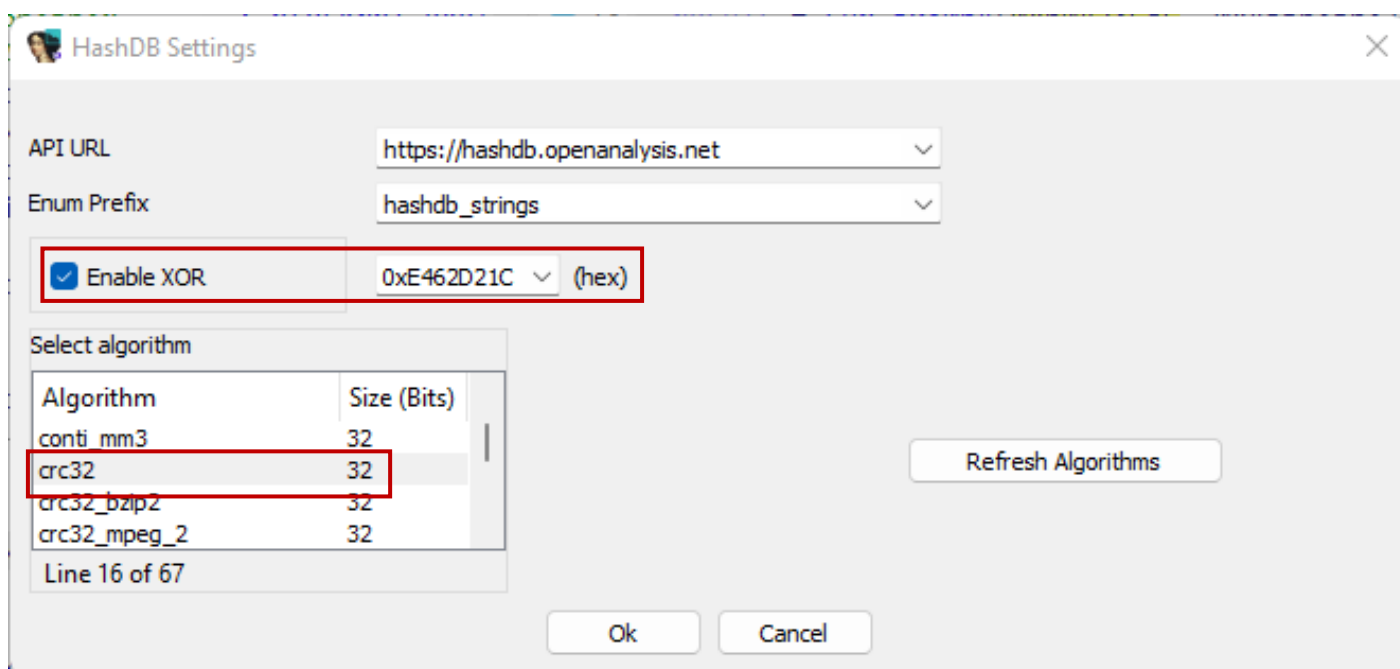
- It's excellent to be used with **analysis of shellcodes** (shellcode analysis will be a key topic covered in next versions of this series – maybe in **MAS 9** or **MAS 10** – stay tuned!)
- It makes comments on **pseudo and Assembly code**.
- It has a quite extensive list of **available hash algorithms**, 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.

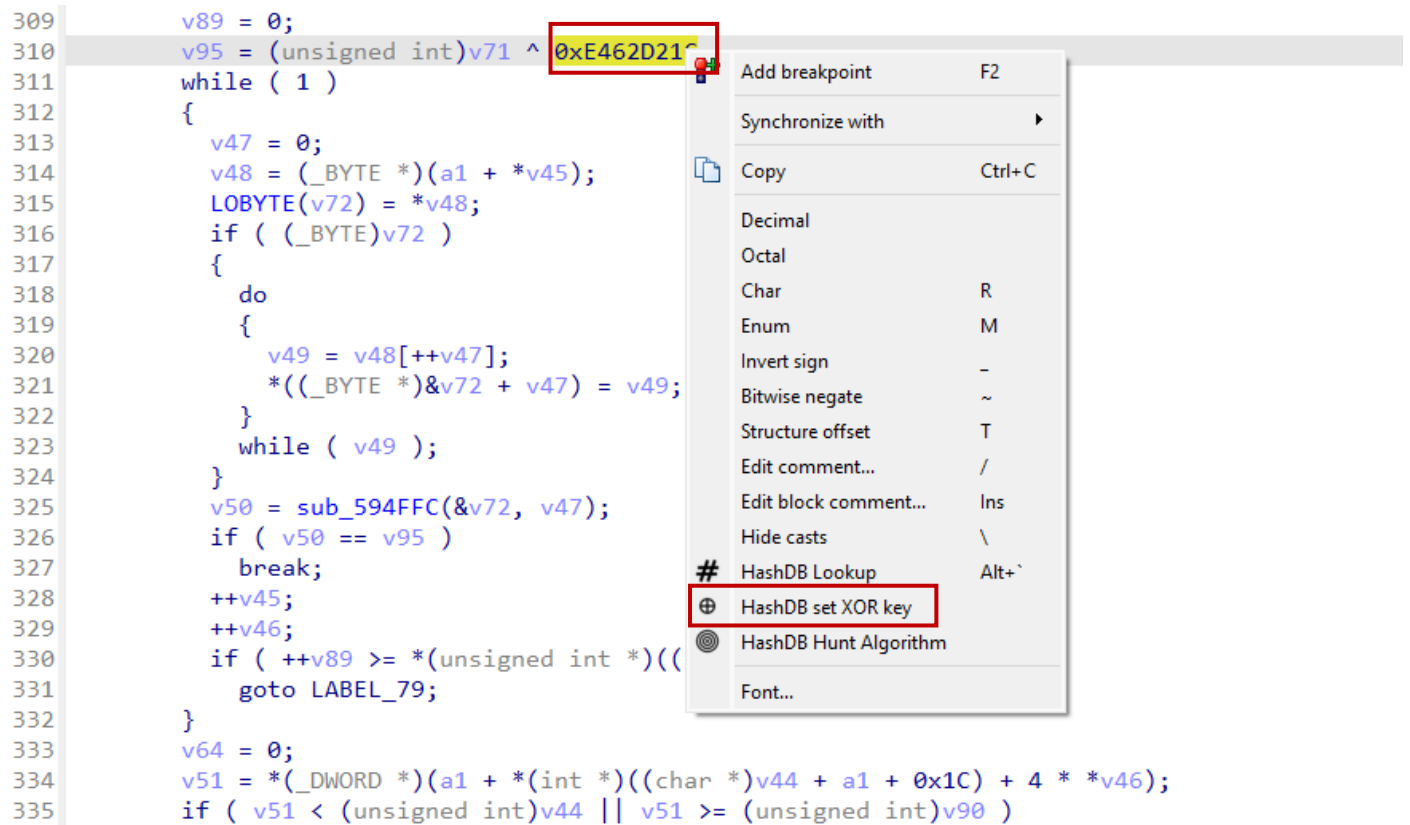
There is more than one way to proceed. The steps to get hash resolution are:

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



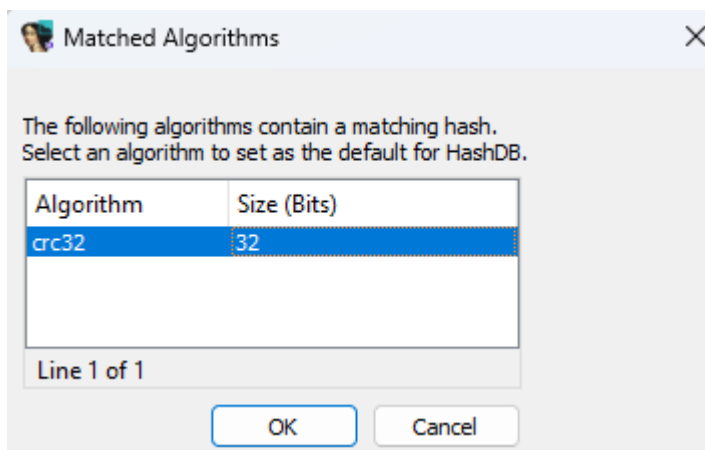
[Figure 32]: HashDB Settings

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

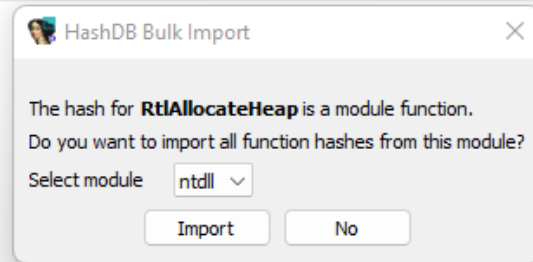
After getting this step done, right click on the hash and choose **HashDB Hunt Algorithm** (check the image above). The advantage of this option is that HashDB will try to guess the algorithm being used automatically. In other words, if you are lucky, there will not be necessary to analyze the code before 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**:

```
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); // ntoskrnl.exe!b'RtlCreateHeap'
9         dword_59D22C = sub_593064(0x60A28C5C, 0x5E0AFAA3); // ntoskrnl.exe!b'RtlDestroyHeap'
10        if ( dword_59D228 == 0xA33C83E5 )
11            dword_59D228 = v1(2, 0, 0, 0, 0, 0);
12    }
13    result = sub_593064(0x60A28C5C, 0x45B68B68); // ntoskrnl.exe!b'RtlAllocateHeap'
14    if ( !result )
15        return 0;
16    __debugbreak();
17    __debugbreak();
18    return result;
19 }
```



[Figure 35]: HashDB: look up for hash

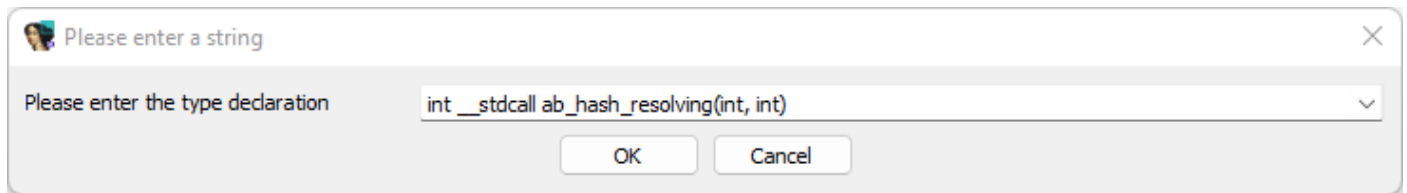
Imports might take few seconds and freeze the IDA Pro, but that is a normal behavior and let it to finish its job. **HashDB**, like **Shellcode Hashes** from Mandiant, creates an enumeration containing all hashed functions, as shown below:

```
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
```

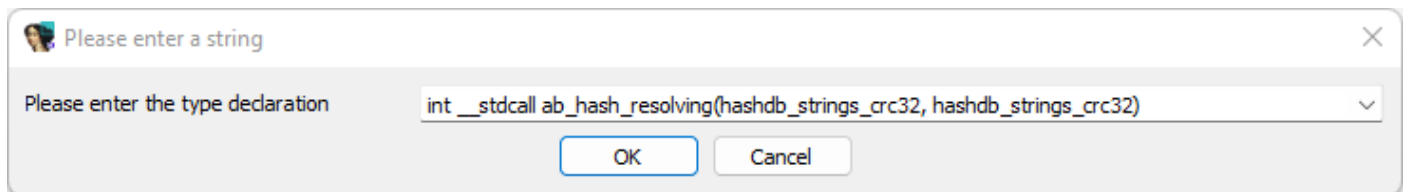
[Figure 36]: HashDB creates an enumeration for APIs

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:

```
1 int sub_593628()
2 {
3     int RtlAllocateHeap_0; // eax
4     int (__cdecl *RtlCreateHeap_0)(int, _DWORD, _DWORD, _DWORD, _DWORD, _DWORD); // ebx
5
6     if ( dword_59D228 == 0xA33C83E5 )
7     {
8         RtlCreateHeap_0 = ab_hash_resolving_0(ntdll_dll, RtlCreateHeap_0); // ntoskrnl.exe!b'RtlCreateHeap'
9         dword_59D22C = ab_hash_resolving_0(ntdll_dll, RtlDestroyHeap_0); // ntoskrnl.exe!b'RtlDestroyHeap'
10        if ( dword_59D228 == 0xA33C83E5 )
11            dword_59D228 = RtlCreateHeap_0(2, 0, 0, 0, 0, 0);
12    }
13    RtlAllocateHeap_0 = ab_hash_resolving_0(ntdll_dll, RtlAllocateHeap_0); // ntoskrnl.exe!b'RtlAllocateHeap'
14    if ( !RtlAllocateHeap_0 )
15        return 0;
16    __debugbreak();
17    __debugbreak();
18    return RtlAllocateHeap_0;
19 }
```

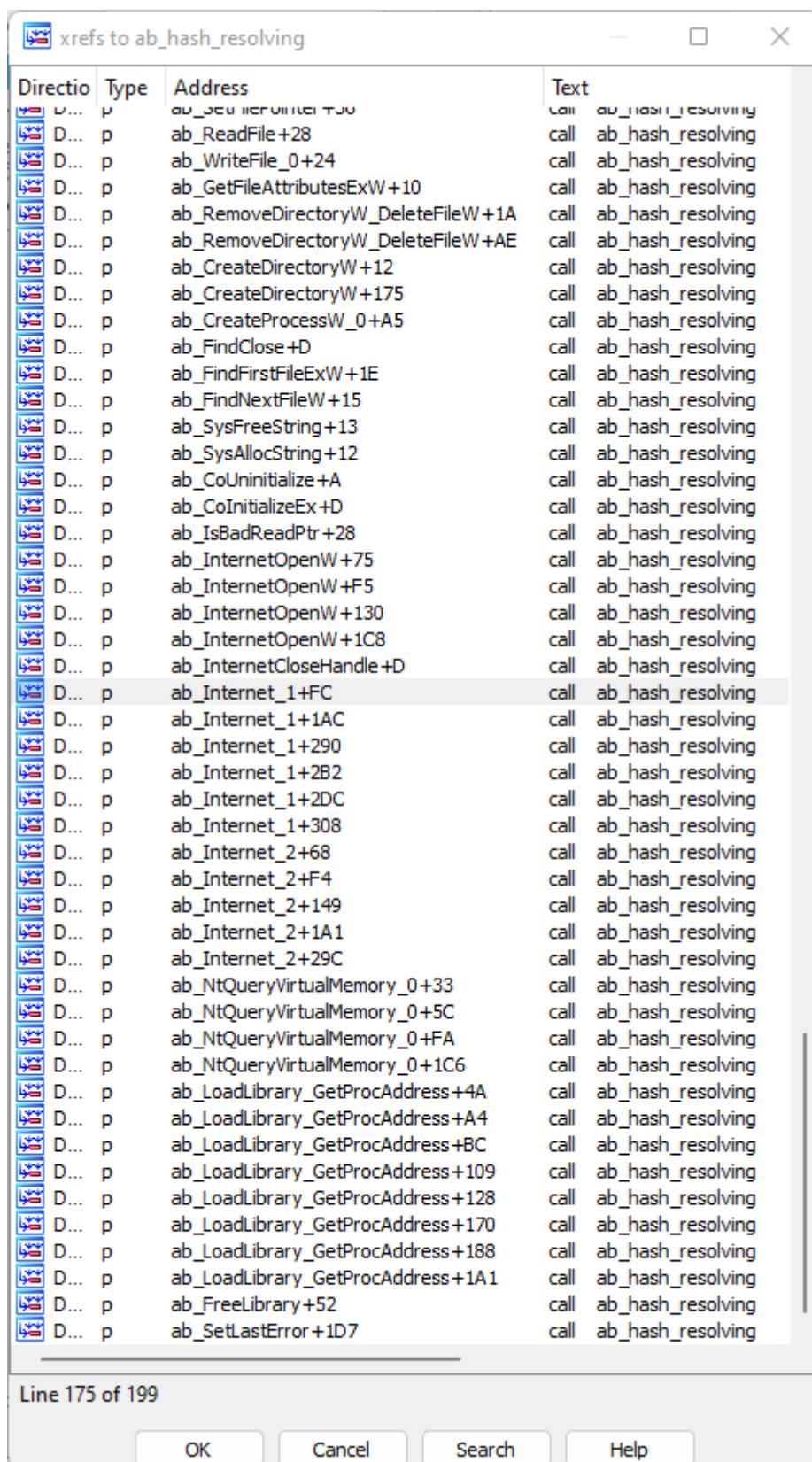
[Figure 39]: After updating the pseudo code

There is a note here: I renamed **v1** to **RtlCreateHeap\_0** and the **result variable** to **RtlAllocateHeap\_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:

```
1 HRESULT __stdcall DllRegisterServer()
2 {
3     int *v0; // esi
4     unsigned int i; // eax
5     int v3; // eax
6     struct _EXCEPTION_REGISTRATION_RECORD *ExceptionList; // [esp+24h] [ebp-8h] BYREF
7     int *v5; // [esp+28h] [ebp-4h]
8
9     v5 = sub_5821FC;
10    ExceptionList = NtCurrentTeb()->NtTib.ExceptionList;
11    for ( i = 0; i < 0x13512; ++i )
12    {
13        __debugbreak();
14        __debugbreak();
15        __debugbreak();
16        __debugbreak();
17    }
18    v5 = v0;
19    ExceptionList = v0;
20    sub_5930E8(&ExceptionList, v0, 0);
21    if ( !byte_59D028 )
22    {
23        ab_OutputDebugStringW_CreateProcess(v0);
24        dword_59D1C4(0);
25    LABEL_8:
26        sub_593144(&ExceptionList, v0);
27        return 1;
28    }
29    if ( byte_59D268 )
30        goto LABEL_8;
31    byte_59D268 = 1; // kernel32.dll!b'CreateThread'
32    v0 = ab_hash_resolving(kernel32_dll, CreateThread_0);
33    if ( v0 )
34    {
35        v3 = sub_591D30(0xA731522);
36        if ( (v0)(0, 0, ab_VirtualFree_ExitThread, v3, 0, 0) )
37            goto LABEL_8;
38    }
39    sub_593144(&ExceptionList, v0);
40    return 0;
41 }
```

[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](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:

```
1 _DWORD *__userpurge sub_5930E8@<eax>(_DWORD *a1@<ecx>, int *a2@<esi>, int a3)
2 {
3     bool v4; // zf
4     int (__stdcall *v6)(int, int *); // eax
5
6     v4 = dword_59D224 == 0xEB797E01;
7     *a1 = a3;
8     if ( v4 )
9     {
10        HIBYTE(word_59D2F0) = 1; // ntdll.dll!b'RtlAddVectoredExceptionHandler'
11        v6 = sub_59306C(0x60A28C5C, 0x5EC9D014, a2, 0x60A28C5C, 0x60A28C5C);
12        dword_59D224 = sub_593138(v6);
13        HIBYTE(word_59D2F0) = 0;
14    }
15    return a1;
16 }
```

[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:

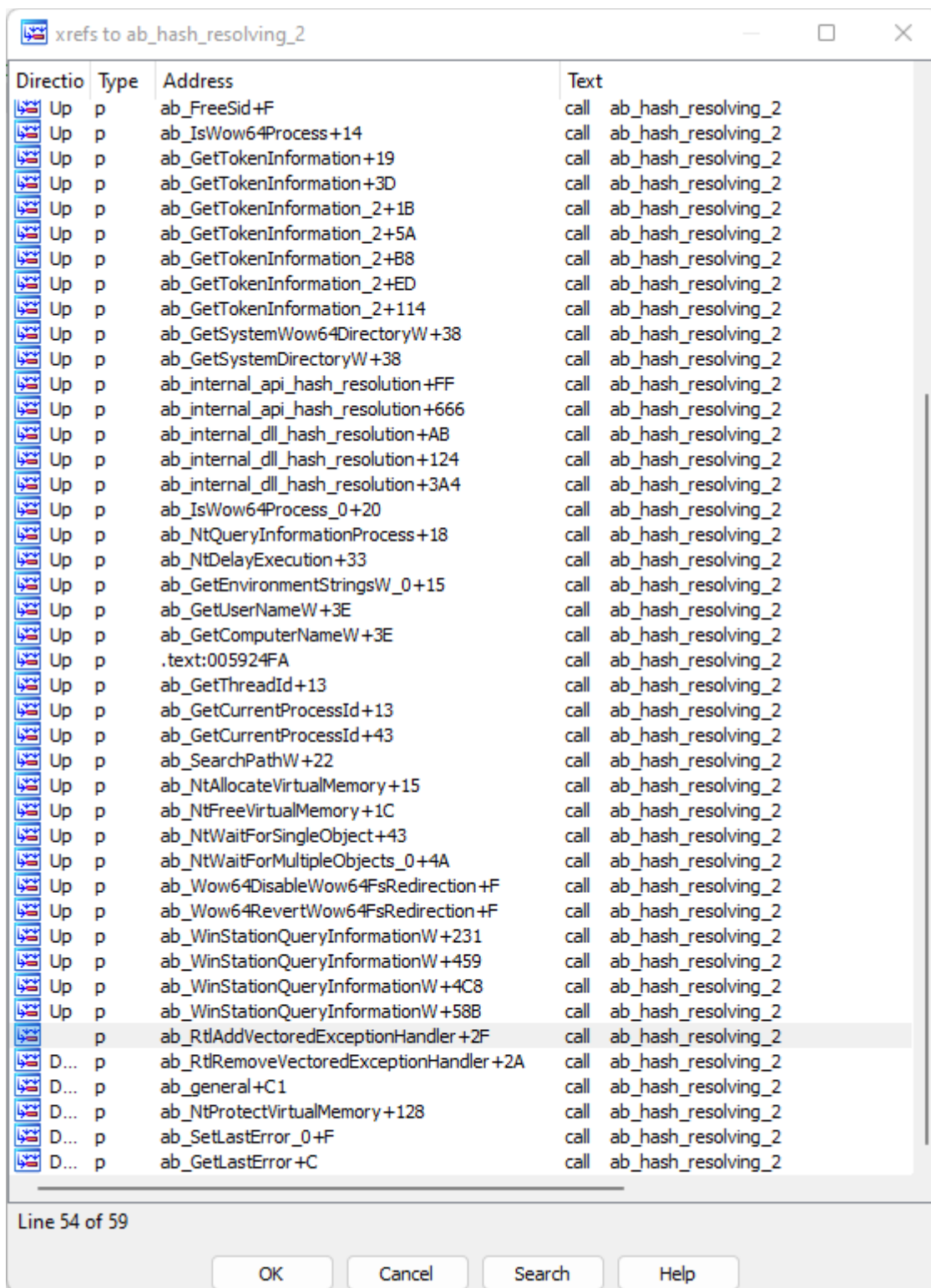
- **char \*\_\_userpurge sub\_59306C@<eax>( hashdb\_strings\_crc32@<eax>, hashdb\_strings\_crc32 @<edx>, int \*@<esi>, hashdb\_strings\_crc32, hashdb\_strings\_crc32)**

The same **sub\_59306C** routine is now present as:

```
1 void __usercall sub_593144(_DWORD *a1@<ecx>, int *a2@<esi>)
2 {
3     if ( *a1 && dword_59D224 != 0xEB797E01 )
4     {
5         dword_59D224 = 0xEB797E01; // ntdll.dll!b'RtlRemoveVectoredExceptionHandler'
6         if ( sub_59306C(
7             ntdll_dll,
8             RtlRemoveVectoredExceptionHandler_0,
9             a2,
10            ntdll_dll,
11            ntdll_dll) )
12         {
13             __debugbreak();
14             __debugbreak();
15         }
16     }
17 }
```

[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:



[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:

```
1 HRESULT __stdcall DllRegisterServer()
2 {
3     int *v0; // esi
4     unsigned int i; // eax
5     int v3; // eax
6     struct _EXCEPTION_REGISTRATION_RECORD *ExceptionList; // [esp+24h] [ebp-8h] BYREF
7     int *v5; // [esp+28h] [ebp-4h]
8
9     v5 = sub_5821FC;
10    ExceptionList = NtCurrentTeb()->NtTib.ExceptionList;
11    for ( i = 0; i < 0x13512; ++i )
12    {
13        __debugbreak();
14        __debugbreak();
15        __debugbreak();
16        __debugbreak();
17    }
18    v5 = v0;
19    ExceptionList = v0;
20    ab_RtlAddVectoredExceptionHandler(&ExceptionList, v0, 0);
21    if ( !byte_59D028 )
22    {
23        ab_OutputDebugStringW_CreateProcess(v0);
24        dword_59D1C4(0);
25 LABEL_8:
26        ab_RtlRemoveVectoredExceptionHandler(&ExceptionList, v0);
27        return 1;
28    }
29    if ( byte_59D268 )
30        goto LABEL_8;
31    byte_59D268 = 1; // kernel32.dll!b'CreateThread'
32    v0 = ab_hash_resolving(kernel32_dll, CreateThread_0);
33    if ( v0 )
34    {
35        v3 = sub_591D30(0xA731522);
36        if ( (v0)(0, 0, ab_VirtualFree_ExitThread, v3, 0, 0) )
37            goto LABEL_8;
38    }
39    ab_RtlRemoveVectoredExceptionHandler(&ExceptionList, v0);
40    return 0;
41 }
```

[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](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:

```
1 _DWORD *__userpurge ab_RtlAddVectoredExceptionHandler@<eax>(
2     _DWORD *a1@<ecx>,
3     int *a2@<esi>,
4     int a3)
5 {
6     bool v4; // zf
7     char *RtlAddVectoredExceptionHandler; // eax
8
9     v4 = dword_59D224 == 0xEB797E01;
10    *a1 = a3;
11    if ( v4 )
12    {
13        HIBYTE(word_59D2F0) = 1; // ntdll.dll!b'RtlAddVectoredExceptionHandler'
14        RtlAddVectoredExceptionHandler = ab_hash_resolving_2(
15            ntdll_dll,
16            RtlAddVectoredExceptionHandler_0,
17            a2,
18            ntdll_dll,
19            ntdll_dll);
20        dword_59D224 = sub_593138(RtlAddVectoredExceptionHandler);
21        HIBYTE(word_59D2F0) = 0;
22    }
23    return a1;
24 }
```

[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:00593136     align 4
.text:00593138 ; ===== SUBROUTINE =====
.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
```

The diagram highlights several key elements in the assembly code:   
1. The `call ab_hash_resolving_2` instruction at address `00593117` is boxed in red.   
2. The `call sub_593138` instruction at address `00593123` is boxed in red.   
3. The `sub_593138` label at address `00593138` is boxed in red.   
4. The `call ecx` instruction at address `0059313F` is boxed in red.   
5. Red arrows indicate the flow of control: one from `call ab_hash_resolving_2` to `mov ecx, eax`, and another from `call sub_593138` to `call ecx`.   
6. A dashed box labeled `RtlAddVectoredExceptionHandle` is shown on the right side of the code block.

[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      loc_5936D6:          ; CODE XREF: ab_memset+18
.text:005936D6      ; ab_memset+1C↑j ...
.text:005936D6      xor   eax, eax
.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 88 86 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 88 44 24 04 0F B6 54 24
005936A0  08 88 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 D8 53 57 CC
005936D0  CC 83 C4 0C EB 02 33 C0 5D 5B 5F C3 88 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 D8 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 88 74 24 18
00593730  85 F6 74 2C 8B 6C 24 1C 85 ED 74 24 8B 5C 24 20
00593740  85 D8 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 D8 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 28 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 88 44 24 04
00593820  8B 54 24 08 8B 4C 24 0C 56 53 55 8B D8 8B F1 8B
00593830  FA 85 D8 74 2E 85 ED 74 21 85 F6 74 1D 68 77 AD
```

[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**):

```
1 int __cdecl ab_memset(int a1, char a2, int a3)
2 {
3     int result; // eax
4
5     if ( !a1 )
6         return 0;
7     if ( !a3 )
8         return 0;
9     result = ab_hash_resolving(ntdll_dll, memset_0); // ntoskrnl.exe!b'memset'
10    if ( !result )
11        return 0;
12    __debugbreak();
13    __debugbreak();
14    return result;
15 }
```

[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 08 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
```

```
1 int __cdecl ab_memset(int a1, char a2, int a3)
2 {
3     int result; // eax
4
5     if ( !a1 )
6         return 0;
7     if ( !a3 )
8         return 0;
9     result = ab_hash_resolving(ntdll_dll, memset_0);
10    if ( !result )
11        return 0;
12    __debugbreak();
13    __debugbreak();
14    return result;
15 }
```

[Figure 61]: sub\_593698 routine: hexadecimal bytes changed

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

```
1 int __cdecl ab_memset(int a1, unsigned __int8 a2, int a3)
2 {
3     int (__cdecl *memset_0)(int, _DWORD, int); // eax
4
5     if ( a1
6         && a3
7         && (memset_0 = (int (__cdecl *) (int, _DWORD, int))ab_hash_resolving(
8             ntdll_dll,
9             memset_0)) != 0 )
10    {
11        return memset_0(a1, a2, a3);
12    }
13    else
14    {
15        return 0;
16    }
17 }
```

[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.

```
1 int ab_RtlAllocateHeap()
2 {
3     int RtlAllocateHeap_0; // eax
4     int (__cdecl *RtlCreateHeap_0)(int, _DWORD, _DWORD, _DWORD, _DWORD, _DWORD); // ebx
5
6     if ( dword_59D228 == 0xA33C83E5 )
7     {
8         RtlCreateHeap_0 = ab_hash_resolving(ntdll_dll, RtlCreateHeap_0); // ntoskrnl.exe!b'RtlCreateHeap'
9         dword_59D22C = ab_hash_resolving(ntdll_dll, RtlDestroyHeap_0); // ntoskrnl.exe!b'RtlDestroyHeap'
10        if ( dword_59D228 == 0xA33C83E5 )
11            dword_59D228 = RtlCreateHeap_0(2, 0, 0, 0, 0, 0);
12    }
13    RtlAllocateHeap_0 = ab_hash_resolving(ntdll_dll, RtlAllocateHeap_0); // ntoskrnl.exe!b'RtlAllocateHeap'
14    if ( !RtlAllocateHeap_0 )
15        return 0;
16    __debugbreak();
17    __debugbreak();
18    return RtlAllocateHeap_0;
19 }
```

[Figure 63]: sub\_593628 routine

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

```
1 int __thiscall ab_RtlAllocateHeap(void *this)
2 {
3     int (__stdcall *RtlAllocateHeap_0)(int, int, void *); // eax
4     int (__stdcall *RtlCreateHeap_0)(int, _DWORD, _DWORD, _DWORD, _DWORD, _DWORD); // ebx
5
6     if ( dword_59D228 == 0xA33C83E5 )
7     {
8         RtlCreateHeap_0 = ab_hash_resolving(ntdll_dll, RtlCreateHeap_0); // ntoskrnl.exe!b'RtlCreateHeap'
9         RtlDestroyHeap_0_1 = ab_hash_resolving(ntdll_dll, RtlDestroyHeap_0); // ntoskrnl.exe!b'RtlDestroyHeap'
10        if ( dword_59D228 == 0xA33C83E5 )
11            dword_59D228 = RtlCreateHeap_0(2, 0, 0, 0, 0, 0);
12    }
13    RtlAllocateHeap_0 = ab_hash_resolving(ntdll_dll, RtlAllocateHeap_0); // ntoskrnl.exe!b'RtlAllocateHeap'
14    if ( RtlAllocateHeap_0 )
15        return RtlAllocateHeap_0(dword_59D228, 8, this);
16    else
17        return 0;
18 }
```

[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 idutils
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 idutils.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](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](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](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](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](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](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:

```
1 int (__stdcall *__fastcall ab_CryptGenRandom(int *a1, int a2))(int, int, int *)
2 {
3     int (__stdcall *CryptAcquireContextW_0)(int *, _DWORD, _DWORD, int, unsigned int); // ecx
4     int v5; // ebp
5     int (__stdcall *CryptGenRandom_0)(int, int, int *); // eax
6     int v8[4]; // [esp+1Ch] [ebp-10h] BYREF
7
8     v8[0] = 0;
9     CryptAcquireContextW_0 = ab_hash_resolving(
10         advapi32_dll,
11         CryptAcquireContextW_0); // advapi32.dll!b'CryptAcquireContextW'
12     if ( !CryptAcquireContextW_0
13         || !CryptAcquireContextW_0(v8, 0, 0, 0x18, 0xF0000000)
14         && ab_GetLastError(a1) )
15     {
16         return ab_memset(a1, 0, a2);
17     }
18     v5 = v8[0];
19     CryptGenRandom_0 = ab_hash_resolving(advapi32_dll, CryptGenRandom_0); // advapi32.dll!b'CryptGenRandom'
20     if ( CryptGenRandom_0 )
21         CryptGenRandom_0 = CryptGenRandom_0(v5, a2, a1);
22     if ( v5 && v5 != 0xFFFFFFFF )
23         return ab_CryptReleaseContext(v5);
24     return CryptGenRandom_0;
25 }
```

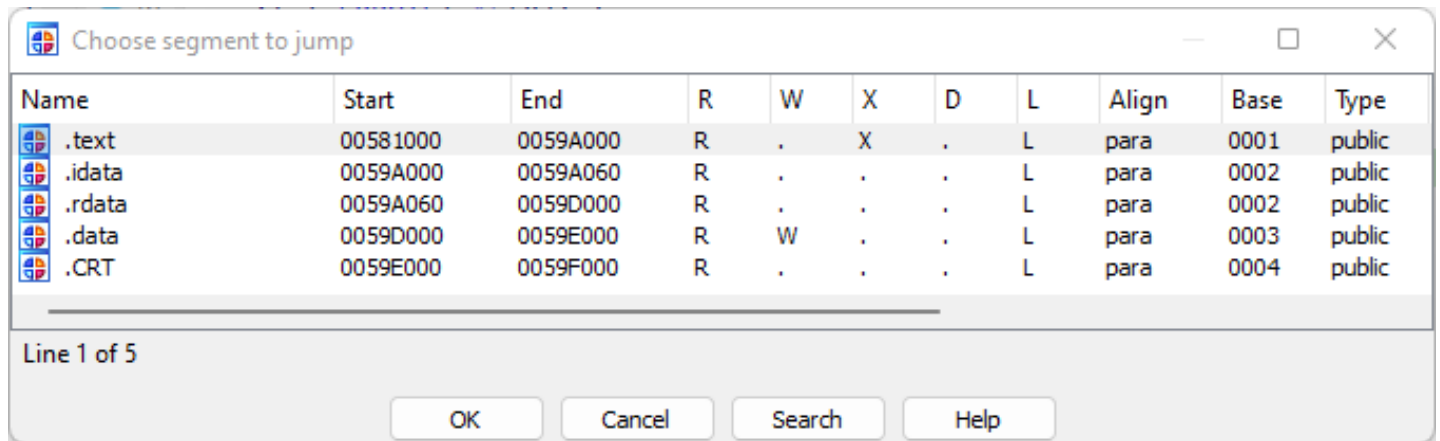
**[Figure 70]: sub\_59494C routine: general aspect after running the script**

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.



[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+108to
.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:

```
25     if ( LOBYTE(v15[0]) )
26     {
27         FullProcessImageNameW = ab_w_QueryFullProcessImageNameW(v2);
28         ptr_ab_encoded_data_2 = &ab_encoded_data_2;
29         if ( FullProcessImageNameW == 0x20 )
30             ptr_ab_encoded_data_2 = &ab_encoded_data_1;
31         ab_w_rc4_0(v20, ptr_ab_encoded_data_2, 0);
32         sub_58F6C0(v21, v20[0]);
33         ab_ww_RtlFreeHeap_0(v20);
34         v10 = sub_58F6A8(v21);
35         sub_58F828(v18, v10);
36         v11 = sub_58F4BC(v21, 0);
37         v12 = sub_58F4BC(v18, 0);
38         sub_5878B4(a2, v11, v12);
39         if ( ab_w_QueryFullProcessImageNameW(v2) == 0x20 )
40         {
41             ab_memset(v29, 0, 0x47C);
42             v29[0x103] = sub_58F4CC(a2);
43             v29[0x101] = 0x56473829;
44             sub_58201C(v29);
45             v29[0x109] = v17;
46             v29[0x10A] = v16;
47             sub_58F4DC(v19, v29, 0x47C);
48             v13 = ab_IAT_1(v18, v2, a2, 0x4BC, v19);
49         }
```

[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**:

```
1  _DWORD * __stdcall ab_w_rc4_0(_DWORD *a1, int data_to_be_decrypted, int a3)
2  {
3  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]
4
5  sub_58DF4C(a1, 0x2800);
6  v3 = *a1;
7  *v3 = 0;
8  sub_58F37C(ptr_raw_key, data_to_be_decrypted, 0x30);
9  ptr_key = sub_58F4BC(ptr_raw_key, 0);
10 ptr_key_1 = sub_58F4CC(ptr_raw_key);
11 ab_reverse_bytes(ptr_key, ptr_key_1);
12 v10 = v3;
13 v12 = a3;
14 v11 = 0;
15 v9[0] = a3 == 0;
16 key = sub_58F4BC(ptr_raw_key, 0);
17 ab_rc4(key, 48, data_to_be_decrypted + 0x30, 0x7FFFFFFF, 0, sub_59341C, v9);
18 ab_ww_RtlFreeHeap_1();
19 return a1;
20 }
```

[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:

```
1 void __fastcall ab_rc4(
2     int a1,
3     int a2,
4     int a3,
5     int a4,
6     int a5,
7     int (__stdcall *a6)(int, int),
8     int a7)
9 {
10 // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]
11
12 if ( a1 && a2 && a3 && a4 && (a5 || a6) )
13 {
14     for ( i = 0; i < 0x100; ++i )
15     {
16         v10 = *(i % a2 + a1);
17         v21[i] = i;
18         *(v22 + i) = v10;
19     }
20     v11 = 0;
21     v12 = 0;
22     LOBYTE(v13) = 0;
23     do
24     {
25         v14 = v21[v12];
26         ++v11;
27         v13 = (v13 + v14 + *(v22 + v12));
28         v21[v12++] = v21[v13];
29         v21[v13] = v14;
30     }
31     while ( v11 < 0x100 );
32     if ( a4 > 0 )
33     {
34         v15 = 0;
35         v16 = 1;
36         LOBYTE(v17) = 0;
37         do
38         {
39             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\_rc4\_0)** and it comes from the call on **line 31** from **sub\_581000 (Figure 73)**.

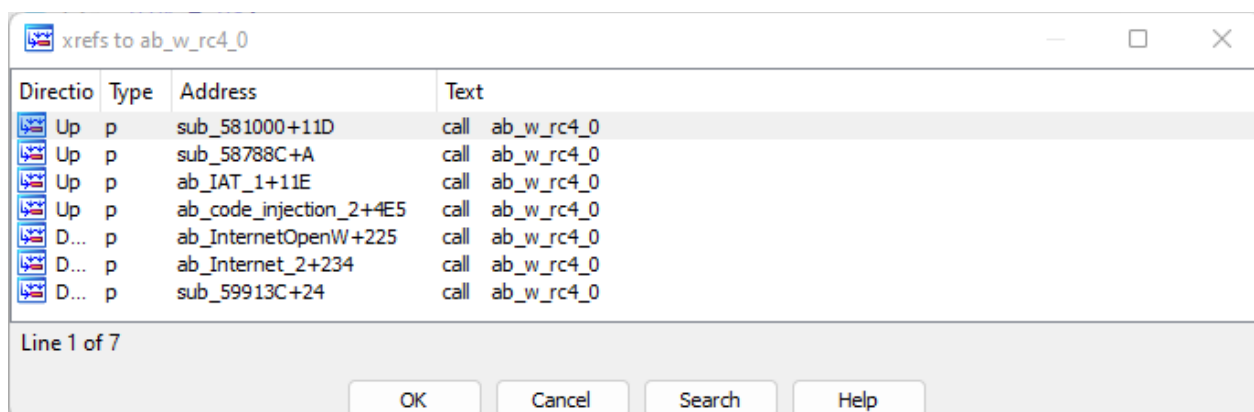
- I know that the **first 0x30 bytes (48 bytes)** is the decryption **key** because the **second argument** of the **call instruction** for **ab\_rc4** routine (line 17, Figure 74) is **data\_to\_be\_decrypted+0x30**. Additionally, on line 8, the **sub\_58DF4C** routine using the same block of data (**data\_to\_be\_decrypted** 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**:

```
1 char __fastcall ab_reverse_bytes(int raw_key_member, int ptr_raw_key_1)
2 {
3     int counter_2; // ebx
4     int counter_1; // ecx
5     char result; // al
6
7     counter_2 = ptr_raw_key_1 - 1;
8     if ( ptr_raw_key_1 - 1 > 0 )
9     {
10        counter_1 = 0;
11        do
12        {
13            result = *(counter_1 + raw_key_member);
14            *(counter_1 + raw_key_member) = *(counter_2 + raw_key_member);
15            ++counter_1;
16            *(counter_2 + raw_key_member) = result;
17            --counter_2;
18        }
19        while ( counter_1 < counter_2 );
20    }
21    return result;
22 }
```

[Figure 76]: sub\_594928

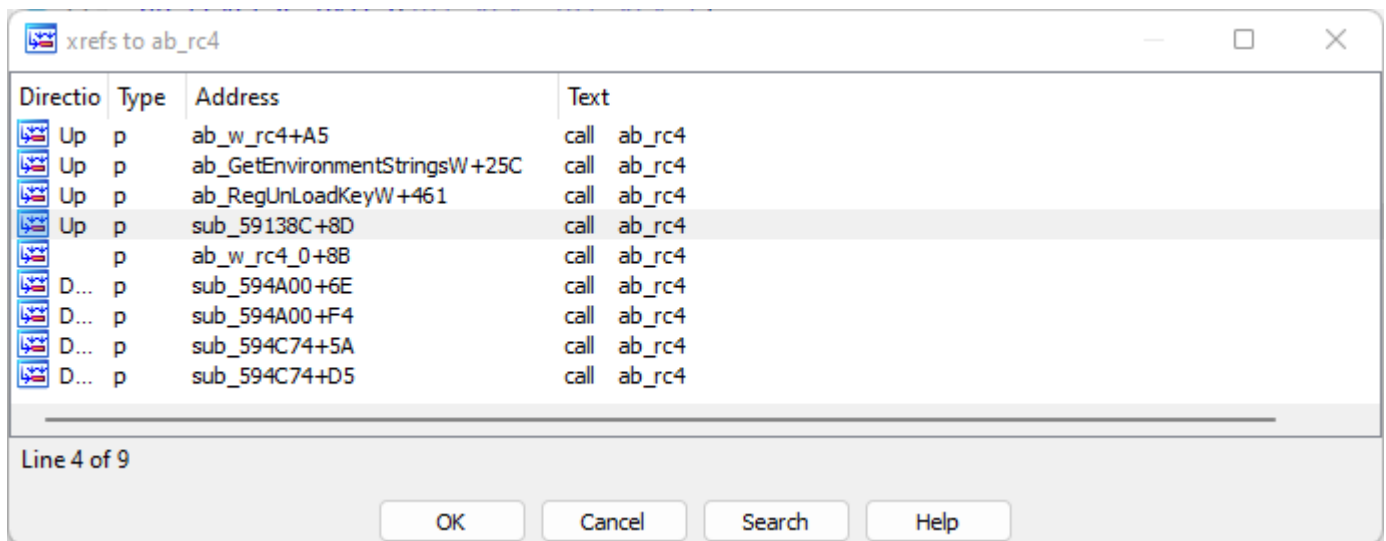
- As a reference has been passed to the **sub\_594928** routine, so the result is also the **same content of ptr\_raw\_key**, which it was already inverted.

Examining references (**X hotkey**) to **ab\_w\_rc4\_0** routine (**sub\_59214C**), we realize it called seven times:



[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:

```
1 import binascii
2 import pefile
3 import base64
4 from Crypto.Cipher import ARC4
5
6 # This routine decrypts RC4 encrypted data.
7 def data_decryptor(key_data, data):
8
9     data_cipher = ARC4.new(key_data)
10    decrypted_config = data_cipher.decrypt(data)
11    return decrypted_config
12
13 # This routine extracts and returns data from .rdata section,
14 # .rdata section address and file image base.
15 def extract_data(filename):
16     pe=pefile.PE(filename)
17     for section in pe.sections:
18         if '.rdata' in section.Name.decode(encoding='utf-8').rstrip('x00'):
19             return (section.get_data(section.VirtualAddress, section.SizeOfRawData),\
20                     section.VirtualAddress, hex(pe.OPTIONAL_HEADER.ImageBase))
21
22 # This routine calculates the offset between the current address of the targeted
23 # data and the start address of the .rdata section section.
24 def calc_offsets(end_addr, start_addr):
25     data_offset = int(end_addr,16) - int(start_addr,16)
26     return data_offset
27
28 # Print decrypted data.
29 def print_data(data):
30     for item in data.split(b'\x00\x00'):
31         final_data = item.replace(b'\x00', b'').decode('utf-8')
32         print(final_data)
33
34 # encrypted_string_addr: start address of the encrypted strings
35 def show_data(encrypted_string_addr):
36
37     # Next two lines extracts .rdata section's information.
38     filename = r"C:\Users\Administrador\Desktop\MAS\MAS_7\mas_7_unpacked.bin"
39     data_encoded_extracted, sect_address, file_image_base = extract_data(filename)
40
41     # Next three lines find the RVA of the .rdata section, the absolute address
42     # of the .rdata section and the offset of encrypted data respectively.
43     data_seg_rva_addr = hex(sect_address)
44     data_seg_real_addr = hex(int(data_seg_rva_addr,16) + int(file_image_base,16))
45     data_offset = calc_offsets(encrypted_string_addr, data_seg_real_addr)
46
```



```
47 # Looking for the end of data and key bytes.
48 d_off = 0x0
49 if (b'\x00\x00' in data_encoded_extracted[data_offset:]):
50     d_off = (data_encoded_extracted[int(data_offset:)]).index(b'\x00\x00')
51
52 # This Line extract the encrypted data
53 encrypted_data = data_encoded_extracted[data_offset:data_offset + d_off]
54
55 # Splits key and encrypted data, and reverse the extracted key
56 key_orig = encrypted_data[0:48]
57 key_reversed = key_orig[0:48][::-1]
58 data_orig = encrypted_data[48:]
59
60 # These commented lines were initially added to
61 # confirm whether the script was really working as the
62 # expected.
63 #key_hex = binascii.b2a_hex(key_orig)
64 #print(key_orig)
65 #print(key_hex)
66
67 # Finally, it calls the routine for decrypting data.
68 decrypted = data_decryptor(key_reversed,data_orig)
69
70 # Print the decoded string.
71 print_data(decrypted)
```

```
1 def main():
2
3     print("\nDecrypted Data:")
4     print(16 * "-" + "\n")
5
6     data_location = ['0x59C560', '0x59C5C0', '0x59AF80', '0x59C660', '0x59A060', \
7                     '0x59A6C0', '0x59B840', '0x59B980']
8     for addr in data_location:
9         print("\n[*] Data at: %s\n" % addr)
10        show_data(addr)
11
12 if __name__ == '__main__':
13
14     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: ShellFolderS90Y1NFUvY5N1RDSpi8BgH6SgS8gPIcU;zwTHMB1SiSgHnmlqIchyvE
q6lSioc0XHE4rT4eCydGgyrIipLbZPitrelc82jktTbqgPlT4yGq<autoElevate>truefalse<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

```
o6ijqK0oo6ijqK0oo6ijqKqqqqqqqqqrKqyqp+fn5+1o60kqqq6qpaqqKrDw5aWt67WvaPFo60fw5yqq
qyqvW/EX8BfwE/AQGQgn2XwV1ZWV1/WVlgfX9/WV1ZWV1ZWV1ZWV1Zmoh9WVBQUFBZwV1ZYRhnl1ZhV
mSo2BgwV1ZWV1ZWV1ZWV1ZwVmfAQMBBAPVA8wBvAPwEBAQE FBQUFAUICAgIAEBAQHEAhAAAAAAQHAWHARAgM
RAwMEAAAUAIAAMBIAgICAgAA/////wAAAP/KAQEBAAYABADAwgEBAwH//wEAA8TEgMBAQH/AwMDyEAA
CgAEAAAAAH8AMwEAAAAAAD/v/////AAAAAACAAP8AAAAAAAAAAAAAAAAAAP//AAAAvwAAAAAAAAAAfwAA/
0pKSkpLUkpKSkpPTEpKSkpKSkpKVUVASkpKRV1NRkpdSkpKSkpKSkpKSkpKSkphY2d0SkprbUpKRW1KSk
RF5koAAAACDQYGBgY0AAAAAYGBgAGBgIGAAoKBwcGAgUFagIAAAQEBAQAAAA0BQYGBgEGAAAIBAAGAA
gACgAMACAAYIBhgD2z/4/qwCwALEAswC6+LsAwADBAME/Yv8Ajf8AxP8Axf8A//rAf80EggAEwkAFggA
FwkAKwkArv8Hsv8AtP8Atf8AwwEAX/+5wgA8AIA
```

[\*] Data at: 0x59A060

DQYAAFUci+xRTlNwWfIz21fs+TsD83R60F4QqxLbDu1GBNRJKHFTA1ADyP/RiDko0YcIdC6LgQzrE700A  
40EmIM4g3TB/zDil3D5w4BLeer2dvxq4fi3HNkdKGQWosCH08N0G4kcRfhoGYAijeD8UN8IOgdq/4ldrE  
g4iFZTpzIBX15bycNTkYOQJLhNY1ogV4nq8ICUZjkCdAMHM8Dp/gN/YTtyPIPygT5QRQoYiXX0GJHLYNQ  
0g2Xs4QPOMBRQDOBTamQE5jAnvd6bDejrIsg0PotdYPzkhcB0bxNiWHFQ/2dF4H02AmsBfMqF2w+EO46f  
ahRKCJLRXBGL8J3Q0wx1IErkrsrxEuwbTGHpa79xGEYMBgiI7Bafi0X0AV4E/3BUzHXwo00XTItAJsxAU  
jymBwPDiQYiWDRKHq4a3sApt0gUM9IAG8QMjUwBGGYHO1AGc20mWYRyCxxD+InBDIXJdCCMFPwDVFHuS3  
cEM03wanZSULNT3kQbCOsd3d1IOIhUfhbo8xtQM3ZKi0CzZEDpS0nDMSg5DHyjIAXk5k3faIYrQTSeoUZ  
8h0KDuKQNA+VWQJRV+HR4azCAoB0DwutQW7HwgMqJBk3cjUgIDGHs0ASD6eT3wTH+/wJ205mK7LQJ2NGB  
c0Ju8CDJ+hFzdTA6VXv0f+FSD4FsAwwRZfTwZoPuAnNz0VA5wnJhyET4A0AxX/6Cs3WfhHyEME4Exh1F/  
wHK9CCFMe+cIYBphLBFY3SKNzNRXRDUa+ZfQFM+oP2EJE/MQdA/IovhPQtIM2pG2wWN1DsFc2hLEao00  
9qHL5hNgQ34SE/Tt4iYcbrEC4r8Mg7PN0TUDxVSezTcx0UiP++UG1YTPAlFHQKuwoDXfTfOZ7rB5sYFfv  
Lv9zJR6C88Cn9vUIJeQc1DA+zBqqRBAGCCZR1pCZ45Ysuw04ZwcGDPBhgPguQj/1A0HXQ61QNIInCKqgOz  
M9AUQdn5khjwQ/lVsgmVtFACbC2AbH0IMuSqrkxLpLQGt5JMqjw/D7STNRSY8AwR98IVAj11bnh1JROAQ  
w9AdAzA2BvAFYPgAqcU6wgMhAdAlD5BHyB0Bsh9BFk05shwDamCXIqLUfQJSfiTFp/Pd14LMgn0xA+i+w  
SaPMyb5D6DYGQOyMp8SAIVwSiHMDkSD4xwq8ZQMLgogHQnFlDqARMRCSTQid6K2kcFEYHqiNz3Hdob0jk  
GI9GJfKcsC8ZRDgGLolteX+Meu9bIz+hIafuAgpHr7uYrXRi9HnUIjYYM7bqYmTP/CgJElsYECFHomBC  
RVwId5NE/rYoX999Mgb4B0wB6P+WlKw1KUQQIwh1WnqezCVTW5ZFNBmVKFt/LLaIkzMS9AdHyIwZfDAJ  
CfVRB8NnmguQAds+iJYiImGURxevPwoFB9uUCxxG4sTiaj06Beqwmh608c/db9VIYoVfVcmFUB5IRQ+KE  
gyo1IQYVb6DDm+GCF1ZKw8HDJoEIZUoAa00JnZrGEWA1o+ECpkCLuCGKJWDVOS27kE0UsNFil1J3fQvsh  
0SvBNesKIgAA=

[\*] Data at: 0x59A6C0

jQcAAEgHhdIPhMUNAUSJXDMkCAp0qSBX7IPs6oAxehwmi9qGBv10GgpgAgZKCEUzP8BE/HYoANJMA81B/  
9HGyUNFg3sQwHQ3e4sFGP/IBmPweFcxSxAqPPEDCjsWDIb/l9gEt+70G3nmTEp0NDaPMCGEyDDATSae5I  
WMdCvkZDgkOA1mJUQM76tU2EBBueCAWj+Dyf+NkSj/IVxooj3NkMOAlBiHQRr/0BIGtjAKg7bOXAXEIF/  
DzDDHVZxOUFUdunftQXMi+xBz0C4TVpDO+oBBuFmOQJ0B0HL6bJAVGNyPOgDBvKBP1BFGjF16lpGMEbm  
6/xQ3BTfQZHfKfiXde2h1lX+qP+SRccNkSh1wzgrICAWoQqU5WBMmHV0UHSJVPDQEsOqFUEK/8NAAIP7A  
ny7TQeF9nSLSSKMJJDDugjBgUSNQhhCPLCZ2/gkCHVnJ9ZFuJLwOQAGUp33QipoI+1Dz6wrcDoIRBt4GB  
CjEMOIwxzm2GZUuNUGUM4nkIvwY0U8obDb3APGoYIHq18j74MGVwgPt8sUPyRVAGZE01gGc3LC4HQBKIP  
nRvijiaAMRTl+HmMiG178z0QGgxTaGRRWBIDVlWzLVZHKXgk6yAzCkY4rAx+H0B0Ufz05yRUyo83NMUI  
CiaYzG2+tvfjJAbHSYPGK0lhBt75fJckIvB0TXFGSct1SHMhhSGVFF/LDDmYtB10f3UokLAWCGPT62TTH  
8iKQgSgE1IHIYPOsYdxy3Kp/qH6dkJBMJ4COMHoHwyB4f3gWoP4A3RiDgoKdWBcY8EUTgEEoTIHE0YpD7  
G0gSDDMsIChOmPnv77u9H6DTvBcr6x1cbQo0jWHSWqoUAMx8IQLZzAX6I+1KMLCigH3vgfItjkBm6FFA  
LCt+0ReCze3YPm1dASaNR5BA6GV38ZDAWznpNy1bPKBPQMRTwSiHThBpPgkcQTWkdI+BPGjKBgf/BTGPJ  
MhOePpThAzYiuCvrGhY/TCnUZx9M0Cqk13312FRUK8WJR134DxhNAzTD/xMOQTkF03QQDoRd2Qh1Sb0GB  
vfrCiGMv27zBv13dDDyK84WuRQBiyAGUNAqTCe8C3UF5gJ80AKEaug2v0/WQDLCDegYQRTDcDyBdcdAit  
P8gVXFfCYWTBsk8BTwhPCSpvbrKIv12FYYBC2K8cCnTUirU8cXvNsC80CE9ki6i9fBMszocvuGd0lk/Aa  
RKCQbo6M/6APCGPMoaDDoI0zpwJI3vJ0Drkw7LAJcATxBv5DH9wPJDQIGLIIPgKtBI1THEUA8dA8K2EUB  
ggqD4dk60sEV6wwPk1+e6iNYIEpAm3SYIaGBBEQKXhSjDAmDaLrprRYRC6T0JJfwQfHJC8V21H/3vyx77  
wnv9cijIKTXRnBialv0or//xko3Rv070/EVjF3HaSsBX1godNPDQK2ACMS6AZ1YksZWFRPICLFwMko1u  
3sgVw6j4FID0TKurjIVYjag8ZHHAeWhMcGnCSApUw7xxAQV+dXh1dnVz7dX3DL1VT9W63Y/FYsIEQBKA  
mZTihBVDDIghM92R02SS/BDtiTVBDyAn0iXwkSX5AwI13Sv4M/TjkTPow3jsoCw4oXD8iqQE4S27511L1  
QJjx/5NUUB1N3R49qRj61cDKZJ8oohogEMhGYKS/H4MuF8Y4nYmQKTCQEAH7PqpYDaIUURm/oYAik1UoB  
oKSQbMwQK1eOgIuqBCjg0MwYvaLICuGSxA0s30hjKarh0jz+ZryzYAOHXirKD4YySoYSshceGgXshqIGd  
ZBN+z4AY07KOKGdZ5DQzAhyImDGDFtQJIZu8yTs10gkFAfQUq8JrPdSRAuVRUH3mmI1eFF11EyFaZkel3  
UMLJ010iIJFiGKcOSWxiFAA==

```
[*] Data at: 0x59B840
```

```
9QAAAFUci+yDhyxTVnN1CNBGOFcAagdZUY1N/I07Tjy8UABFCP9WFIPEfgzy7PpqZyAM/GwICMwU/ywYd  
TPk2w+LRN4krARmAgYNTfBRCNzfZDZmdub4oCN03n0gMn3cBuAE8Aj4BvT/F1pwaHAQ4UNJVyoK9EpuCH  
Jh0991A5g06Gkki90u/uJcAVCTekAZJQR8Qwr7AnyPiF70Y0h5gomVjdSDIOSAxTADYjtXASAiegzqdmY  
0DhCUY0QMhF9eWxzJwgQwAA==
```

```
[*] Data at: 0x59B980
```

```
kAEAAEADVNVW0FU11BIjWwkdEcgeyYNwLDi9kCBklwQbgHgA2NU3hMHolFdzoPTW//bSge7GdJPYPOP  
ExvJ4dVb8eLwUG5IEAoiUQk8YZCNDP2Ln17dEUBZg0LB4N158M2COs4CkL3IX8c7xvHV0sEfGd4ZFBADT  
hUAmhNZ/erfyMzyXmWMyQwIecoutYM3uoNT/gSoeJkGRNOTUN87zZWwP+DUSHOoBGIaBAQhDrjYE3v9v7  
FD1V/UbCsB11vhfZgD0TYS5mJC8Q2MXPPFAiqxpEPxxCeg8wPhU7+AoTdg7op4iDe6NhRT01dv0CyRSaJ  
g91DYD87D3nZHDxURswbUAcJQJAoUVzNBjFcUCHcKKEFIQTQXahV0nExGIRLaLIYEiADTQdiDlqBxERYO  
UFeHVxfB1tdwwA=
```

[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\x000\x005\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 132**. On the next line the **sub\_58F6C0( )** is called, and part of its content is the following:

```
26 do
27 {
28 ++v8;
29 if ( v2 >= v17 )
30 break;
31 v9 = v20[v8 - 1];
32 v2 = &v20[v8];
33 v10 = v9 - 0x41;
34 if ( (v9 - 0x41) <= 0x19 )
35 goto LABEL_12;
36 if ( (v9 - 0x61) <= 0x19 )
37 {
38 v10 = v9 - 0x47;
39 goto LABEL_12;
40 }
41 if ( (v9 - 0x30) <= 9 )
42 {
43 v10 = v9 + 4;
44 LABEL_12:
45 if ( v10 == 0xFFFFFFFF )
46 goto LABEL_19;
47 goto LABEL_18;
48 }
49 if ( v9 == '+' )
50 {
51 v10 = '>';
52 }
53 else
54 {
55 if ( v9 != '/' )
56 {
```

[Figure 83]: part of sub\_58F6C0 routine

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**

<https://exploitreversing.com>

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:

```
1 import binascii
2 import pefile
3 import base64
4 import struct
5 import os
6 from Crypto.Cipher import ARC4
7 from malduck import aplib
8
9 # This routine decrypts RC4 encrypted data.
10 def data_decryptor(key_data, data):
11
12     data_cipher = ARC4.new(key_data)
13     decrypted_config = data_cipher.decrypt(data)
14     return decrypted_config
15
16 # This routine extracts and returns data from .rdata section,
17 # .rdata section address and file image base.
18 def extract_data(filename):
19     pe=pefile.PE(filename)
20     for section in pe.sections:
21         if '.rdata' in section.Name.decode(encoding='utf-8').rstrip('x00'):
22             return (section.get_data(section.VirtualAddress, section.SizeOfRawData)),\
23                 section.VirtualAddress, hex(pe.OPTIONAL_HEADER.ImageBase)
24
25 # This routine calculates the offset between the current address of the targeted
26 # data and the start address of the .rdata section section.
27 def calc_offsets(end_addr, start_addr):
28     data_offset = int(end_addr,16) - int(start_addr,16)
29     return data_offset
30
```

```
31 # Print decrypted data.
32 def print_data(data):
33     for item in data.split(b'\x00\x00'):
34         final_data = item.replace(b'\x00', b'').decode('utf-8')
35         print(final_data)
36
37 # encrypted_string_addr: start address of the encrypted strings
38 def show_data(encrypted_string_addr, base = 0):
39
40     # Next two lines extracts .rdata section's information.
41     filename = r"C:\Users\Administrador\Desktop\MAS\MAS_7\mas_7_unpacked.bin"
42     data_encoded_extracted, sect_address, file_image_base = extract_data(filename)
43
44     # Next three lines find the RVA of the .rdata section, the absolute address
45     # of the .rdata section and the offset of encrypted data respectively.
46     data_seg_rva_addr = hex(sect_address)
47     data_seg_real_addr = hex(int(data_seg_rva_addr,16) + int(file_image_base,16))
48     data_offset = calc_offsets(encrypted_string_addr, data_seg_real_addr)
49
50     # Looking for the end of data and key bytes.
51     d_off = 0x0
52     if (b'\x00\x00' in data_encoded_extracted[data_offset:]):
53         d_off = (data_encoded_extracted[int(data_offset):]).index(b'\x00\x00')
54
55     # This line extract the encrypted data
56     encrypted_data = data_encoded_extracted[data_offset:data_offset + d_off]
57
58     # Splits key and encrypted data, and invert the extracted key
59     key_orig = encrypted_data[0:48]
60     key_reversed = key_orig[0:48][::-1]
61     data_orig = encrypted_data[48:]
62
63     # These commented lines were initially added to
64     # confirm whether the script was really working as the
65     # expected.
66     #key_hex = binascii.b2a_hex(key_orig)
67     #print(key_orig)
68     #print(key_hex)
69
70     # Finally, it calls the routine for decrypting data.
71     decrypted = data_decryptor(key_reversed,data_orig)
72
73     # Print the decoded string.
74     if (base == 1):
75         decrypted_base64 = base64.b64decode(decrypted)
76         print(decrypted_base64)
77         data_len = struct.unpack('<I', decrypted_base64[:4])
78         data_corpus = aplib(decrypted_base64[4:])
79         print("\n[+] Decompressed Data Length: %d" % data_len)
```



```
80     #print(data_corpus)
81     return(data_corpus)
82 elif (base == 2):
83     decrypted_base64 = base64.b64decode(decrypted)
84     print(decrypted_base64)
85 else:
86     print_data(decrypted)
```

```
1 def main():
2
3     counter = 1
4     dir_path = r"C:\Users\Administrador\Desktop\MAS\MAS_7\Saved_Files"
5     returned_corpus = b''
6
7     print("\nDecrypted Data:")
8     print(16 * "-" + "\n")
9
10    data_clear_text = ['0x59C560', '0x59C5C0', '0x59AF80']
11
12    data_base64_aplib = ['0x59A060', '0x59A6C0', \
13                        '0x59B840', '0x59B980']
14
15    data_pure_base64 = ['0x59C660']
16
17    for addr in data_clear_text:
18        print("\n[*] Clear data at: %s\n" % addr)
19        show_data(addr)
20
21    for addr in data_base64_aplib:
22        print("\n[*] Compressed aPLib data at: %s\n" % addr)
23        returned_data = show_data(addr, 1)
24        bin_filename = ("filedata_%s.bin" % counter)
25        filename = os.path.join(dir_path, bin_filename)
26        if (not os.path.exists(filename)):
27            open(filename, 'wb').write(returned_data)
28            print("[+] Decompressed aPLib data saved as %s\n" % filename)
29            counter = counter + 1
30        else:
31            counter = counter + 1
32
33    for addr in data_pure_base64:
34        print("\n\n[*] Decoded base64 data at: %s\n" % addr)
35        show_data(addr, 2)
36
37 if __name__ == '__main__':
38
39     main( )
```

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



```
\x07@\x94>\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\xcc  
6P0\xb8(\x80t'\x16Wj\x01\x13\x11\t$\xd0\x89\xde\x8a\xdaG\x05\x11\x81\xea\x88\xdc\xf7\x1d\xda\x1b\xd29\x  
06#\xd1\x89\x16G,\x0b\xc6Q\x0e\x01\x8b\xa2[^_xe3\x1e\xbb\xd6\xc8\xcf\xe8Hi\xfb\x80\x82\x91\xeb\xee\xe6  
+]\x18\xbd\x1eu\x08\x8d\x86\x0c\xed\xba\x98\x993\xff\n\x02D\x96\xc6\x04\x08Q\xe8\x9a`BE\\\x08w\x93D\xfe  
\xb6(\xdf)2\x06\xf8\x04\xec\x01\xe8\xff\x96,\xac%)D\x10#\x08uZz\x9e\xcc%5[\x96E4\xa0fT\xa1m\xfc\xb2\xd  
a"L\xccK\xd0\x1d\x1f"0dw@$'\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\r5\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\x0c0t7{\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$8rL\x8dD\x0c\xef\xabT\xd8@A\xb9\xe0\x80\xc2?\x83\xc9\xff\x8d\x9  
1(\xff!\\h\xa2=\xcd\x90\xc3\x80\x94\x18\x87A\x1a\xff\xd0\x12\x06\xb60\n\x83\xb6\xce\\\x05\xc4 _\xc3\xcc0  
\xc7UWV9ATv\xe9\xdf\xb5\x05\xcc\x8b\xecA\xcf@\xb8MZC;\xea\x01\x06\xe1f9\x02t\x07A\xcb\xe9\xb2@Tcr<\xe8\x  
03\x06\xf2\x81>PE\x1a1u\xeaZF0F\xe6\xeb\xfcP\xdc\x14\xdfA\x91\xdf(X\x97tM\xa1\x96U\xfe\xa8\xff\x92E\xc7  
\r\x91(u\xc38\x11 0\xa1\n\x94\xe5`L\x98utPt\x89T\xf0\xd0\x12\xca\x10)A\n\xff\xc3@\x00\x83\xfb\x02|\xbbM  
\x07\x85\xf6t\x8b1"\x8c$\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\x1b\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? $U\x00fD;  
X\x06sr\xc2\xe0t\x01(\x83\xe7F\xf8\xa3\x89\xa0\x0cE9~\x1ec"\x1a^\xfc\xfd\x06\x83\x14\xda\x19\x14V\x04\x  
80\xd5\x951\xcbU\x98J^\x08$\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\xcb\r\xa9\xfe\xa1\xfa  
vBA\x98\x9e\x028\xc1\xe8\x1f\x0c\x81\xe1\xfd\xe0Z\x83\xf8\x03tb\x0e\n\nu` \lc\x1\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\cav\xcf(\x13\xd01\x14\xf0H\x885\x84\x1  
a0\x82G\x10Mi\x1d#\xe00\x18\x99\x01\x81\xff\xc1L\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\xbf\n\xf3\x06\xf9wt0\xf2+\xce\x16\xb9\x14\x01\x89\x80\x06P\xd0*M\xc1<\x0bu\x0  
5J\x02|8\x02\x84i\xe86\xbc\xef\xd6@2\xc2tH2A\x14\xc3\x08<\x81u\xc7@\x8a\xd3\xfc\x81U\xc5\x14&\x16L\x1b  
$\xf0\x14\xf0\x84\xf0\x92\xa7&\xeb(\x8b\xe5\xd8V\x18\x04-\x8a\xf1\xc0\xa7MH\xeb5\xc7\x17\xc8\xdb\x02\xf3  
@\x84\xf6H\xba\x8b\xd7\xc12\xcc\xe8r\xfb\x86t\xe9d\xfc\n\x91($\x1b\xa3\xa3?\xe8\x03\xc2\x18\xf3(h0\xe8#L  
\xe9\xc0\x927\xbc\x93\x83\xae!;\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$\x97\xf0A\xf8I\x0b\xc5v\x94\x7f\xf7\xbf,{\xef\t\xef  
\xf5\xc8\xa3 \xa4\xd7Fpbj[\xf4\xa2\xbf\xff\xc6J7F\xfd; \xf1\x15\x8c]\xc7i+\x01_X(t\xd3\xc3@\xad\x80\x08  
\xc4\xba\x01\x99X\x92\xc6V\x15\x13\xc8\x08\xb1V\x90\x9c9(\xd6\xed\xec\x81\\\:\x8f\x81H\x0fD\xca\xba\xb8\x  
8U\x88\xda\x83\xc6G\x1c\x010\x84\xc7\x06\x9c$\x80\xa4\xfb\x0b\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\x9f9\xd6R\xf5@\x98\x91\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\x9c*\x18J\xc8\|h\x17\xb2\x1a\x88\x19  
\xd6A7\xec\xf8\x01\x83\xbb(\xe2\x86u\x9eCC0!\xc8\x89\x83\x1815@\x92\x19\xbb\xcc\x93\xb35\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'
```



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_0FFFFFFF_1 = var_0FFFFFFF;
7     v6 = ab_Token_Manipulation(0, var_0FFFFFFF);
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_0FFFFFFF_1);
13    v8 = 0;
14    if ( var_0FFFFFFF_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_0FFFFFFF_1);
22        if ( ab_Internet_1(v14, v9) )
23        {
24            ab_Internet_2(v14, v20, v19);
25            sub_58F434(v18, v20);
26            ab_wv_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_wv_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_0FFFFFFF dd 0FFFFFFFh          ; 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:

```
1 import binascii
2 import pefile
3 import struct
4 import os
5 import ipaddress
6
7 # This routine extracts and returns data from .data section,
8 # .data section address and file image base.
9 def extract_data(filename):
10     pe=pefile.PE(filename)
11     for section in pe.sections:
12         if '.data' in section.Name.decode(encoding='utf-8').rstrip('x00'):
13             return (section.get_data(section.VirtualAddress, \
14                                     section.SizeOfRawData))
15
16 # Extract bytes from .data section, and format IP addresses and ports.
17 def extract_C2():
18
19     # Next two lines extracts .data section's information.
20     filename = r"C:\Users\Administrador\Desktop\MAS\MAS_7\mas_7_unpacked.bin"
21     data_encoded_extracted = extract_data(filename)
22
23     # Initialize important offsets used to extract bytes associated with
24     # IP addresses and ports.
25     initial_data_offset = 0x20
26     botnet_offset = initial_data_offset + 4
27     ip_size_offset = 0x2b
28     ip_list_offset = ip_size_offset + 1
29
30     # Extract the number of IP addresses/ports and calculate the total size.
31     num_ips = ord(data_encoded_extracted[ip_size_offset:(ip_size_offset)+1])
32     ip_list_bytes = 6 * num_ips
33
34     # This line extracts the encoded IP:port data bytes.
35     ip_bytes = data_encoded_extracted[ip_list_offset:ip_list_offset \
36                                       + ip_list_bytes]
37
38     # This line extracts the botnet
39     extracted_botnet = struct.unpack('<h', data_encoded_extracted \
40                                     [botnet_offset:botnet_offset+2])[0]
41     print("\n[*] BOTNET: %s" % extracted_botnet)
42
43     # Extract IP addresses and respective ports, and format them.
44     print("\n[+] C2 IP ADDRESS LIST:")
45     print(24 * '-' + "\n")
46
47     k = 0
48     i = 0
```

```
49     while (k < len(ip_bytes)):
50         ip_item = ip_bytes [k:k+4]
51         ip_port = ip_bytes [k+4:k+6][::-1]
52         print("IP[%d]: %s" % (i,ipaddress.IPv4Address(ip_item)),end=':')
53         print(int(binascii.hexlify(ip_port),16))
54         k = k + 6
55         i = i + 1
56
```

```
1 def main():
2
3     extract_C2()
4
5 if __name__ == '__main__':
6
7     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!