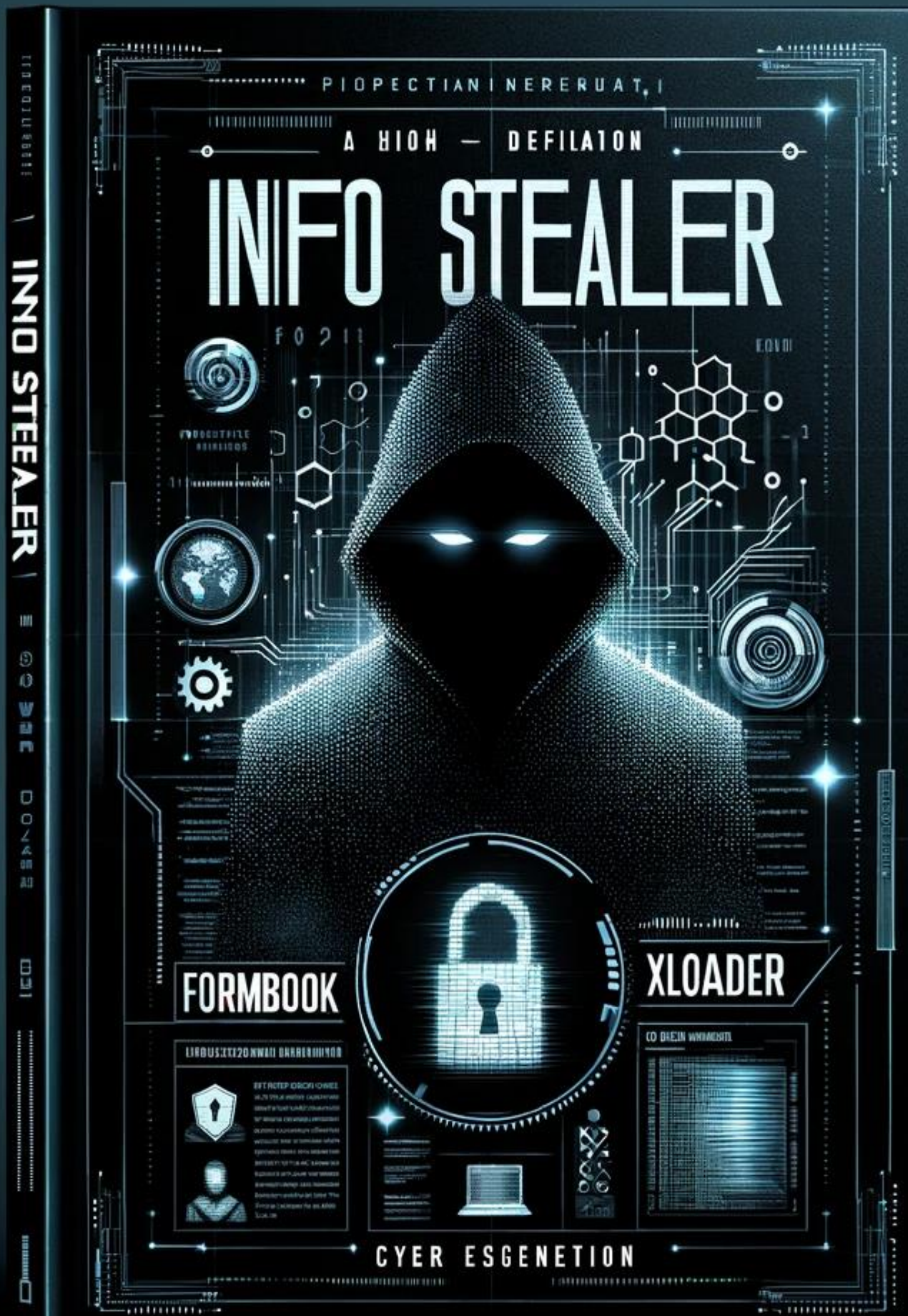
Layers of Deception:

Analyzing the Complex Stages of XLoader Malware Evolution



Shayan Ahmed Khan
THREAT RESEARCHER

Contents

Executive Summary 2

Overview 3

Threat Report: XLoader 4.3 4

Initial Detonation: 4

Stage 1: Dropper 6

Stage2: Xloader 4.3 10

Defeating Anti-Analysis: 13

Decryption/Deobfuscation Routine: 20

Process Enumeration: 32

Process Injection: 33

Stage 3: Partially Decrypted Xloader 4.3 38

Defeating Anti-Analysis: 38

Decryption/Deobfuscation: 39

Indicator Removal: 40

Process Injection: 41

System Information Discovery: 42

Dynamic Library/API resolution: 43

Process Enumeration & Injection: 43

Botnet registration: 44

Stealer: 45

Decrypted Functions: 49

Privilege Escalation: 51

Persistence: 52

Setting Inline Hooks: 52

References: 55

Executive Summary

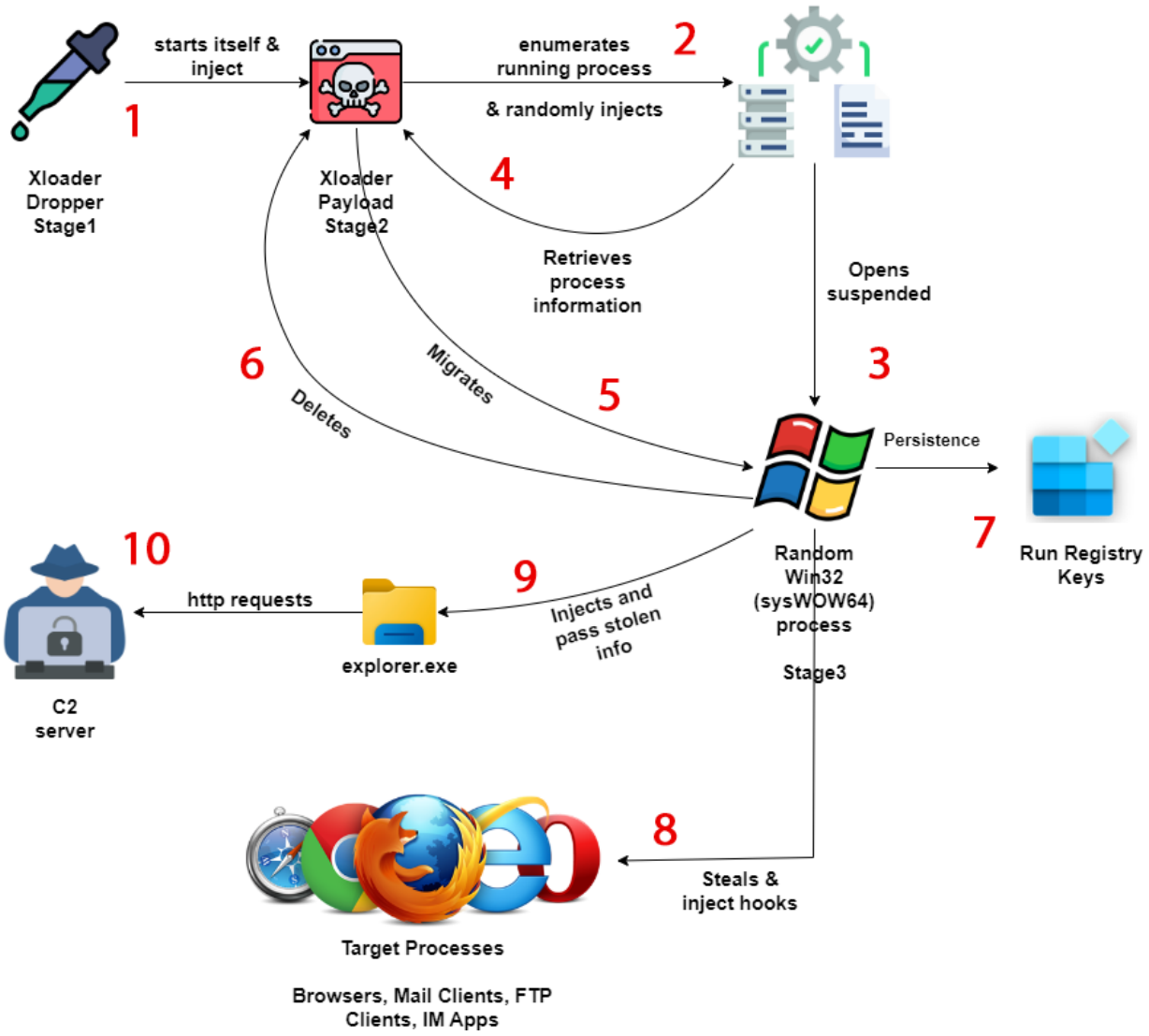
XLoader, an advanced evolution of the **FormBook** malware, stands out as a highly sophisticated cyber threat renowned for its dual functionality as an **information stealer** and a versatile downloader for malicious payloads. Noteworthy for its resilient nature, xLoader constantly adapts to the latest and most intricate **evasion techniques**, making it a formidable challenge for cybersecurity defenses. Its notoriety is heightened by its role as a commercial **Malware-as-a-service** solution, enabling cybercriminals to tailor and deploy the malware for diverse malicious activities. The malware's continuous evolution and ability to elude detection emphasize the critical need for robust cybersecurity measures to counter its intricate and multifaceted attacks, which target both individuals and organizations alike.

Key Findings:

1. **Initial Dropper:** Xloader uses a similar initial dropper as some of the other infostealers like Remcos RAT and Agent Tesla. The initial dropper is a dotnet executable file, which contains multiple embedded **DLLs** which are extracted and decrypted at run-time to launch the payload which is the actual malware. The payload is launched using **Process Hollowing** in either itself or another running process, depending upon the configuration of the initial dropper.
2. **Native Assembly Paylaod:** Xloader is written in native low level asm/c language. There are no strings, imports and libraries found in this payload. Native assembly with the combination of c language already makes it **much harder to analyze and detect** than other infostealers like Remcos, Agent Tesla, NanoCore etc.
3. **Anti-Analysis/VM Techniques:** It uses advance techniques that detects if the malware is running in an analysis environment. The usage of advanced techniques makes sure that, **anti-vm checks** are not easily bypassed as simply as patching a jump condition or return condition.
4. **Custom Encryption Algorithms:** It uses a **Custom RC4** encryption/decryption algorithm with additional subtraction operations.
5. **API/String/Libraries Hashing:** Xloader uses **CRC32/BZIP2** hashing algorithm for its strings, libraries and APIs to hide its internal working.
6. **Encrypted Core Functions:** Xloader's core malicious functions are all encrypted that are decrypted at-run time and assembly is renewed or regenerated after all anti-vm checks have been bypassed and a key has been generated.
7. **Unhooked Clean Ntdll:** It uses a clean copy of **ntdll** manually mapped into its memory which bypass all hooks for ntdll APIs. It uses Native APIs for its malicious activities which are hidden from EDR solutions.
8. **Persistence:** Xloader adds persistence using Run Registry Keys and copying itself in Program Files (x86).
9. **Privilege Escalation:** It escalates privileges only for copying itself in the Program Files (x86) and adding persistence. The privilege escalation is achieved by abusing DllHost.exe and COM objects.
10. **Process Injection:** Xloader relies heavily on process injection. It infects multiple processes in its execution and even migrate to a different process.
11. **Decoy C2s:** It uses a combination of decoy C2 servers and made significant effort to hide its real C2.
12. **Form Grabber:** Xloader is not just an infostealer. It also works as a form grabber. Inline hooks are injected into multiple victim processes to grab information before encryption is performed.

Overview

XLoader emerges as an exceptionally sophisticated infostealer and form grabber malware, distinguished by its adept use of advanced defense evasion techniques to maintain stealth and resilience. Beyond its evasive maneuvers, XLoader incorporates a myriad of anti-VM techniques, strategically avoiding execution in analysis environments. This malware's primary objective is data exfiltration, achieved through the theft and capture of sensitive information from a broad spectrum of applications, including browsers, email clients, FTP clients, and instant messaging apps. Notably, XLoader is designed to operate seamlessly across a variety of platforms, amplifying its threat level. Its multifaceted attack flow encompasses a strategic and systematic approach, making it a potent tool for cybercriminals seeking to compromise both individual users and organizational systems. The constant evolution of XLoader underscores the need for robust cybersecurity measures to counter its intricate and adaptable nature.



Xloader Attack Chain

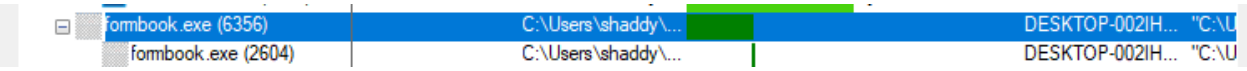
Threat Report: XLoader 4.3

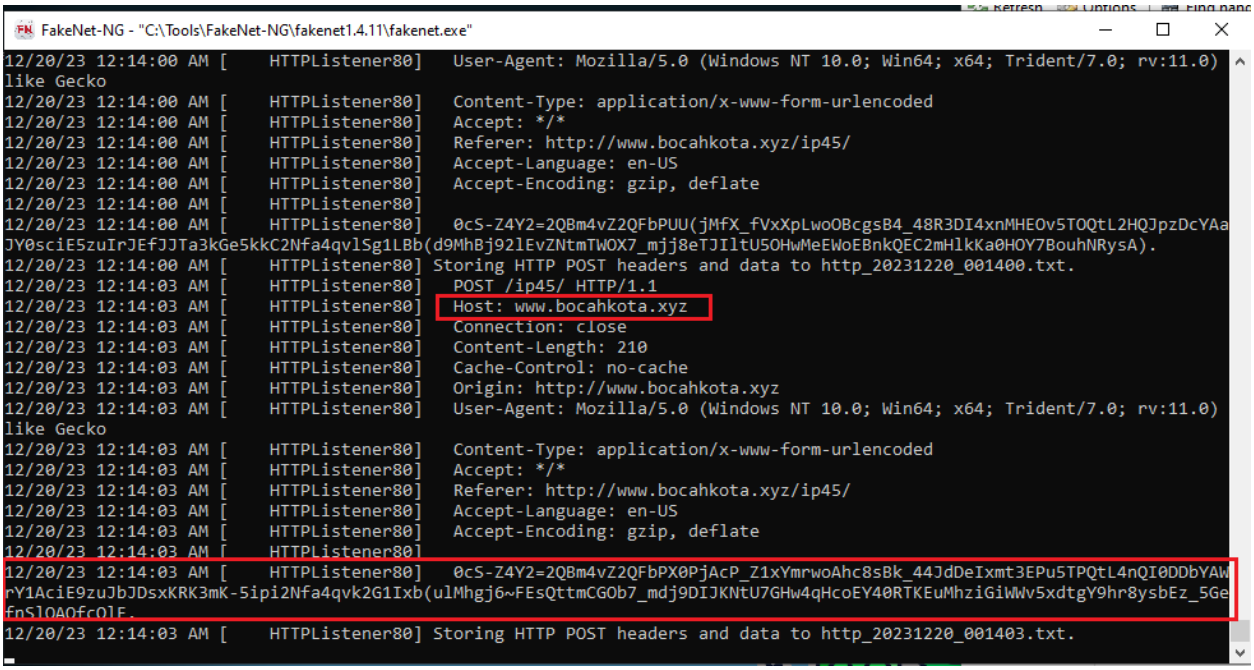
This section of the report provides a detailed technical analysis of **Xloader 4.3** malware. The flow of this report will be in order of steps that I performed during my analysis. This is one of the most complex pieces of malware that I have analyzed, and there are so many stages to its execution. I have tried to cover as much as possible in the given time, but if some things remain unanswered then I apologize beforehand. Now let us dive down into the technical details and internal workings of Xloader 4.3 previously known as **Formbook** infostealer.

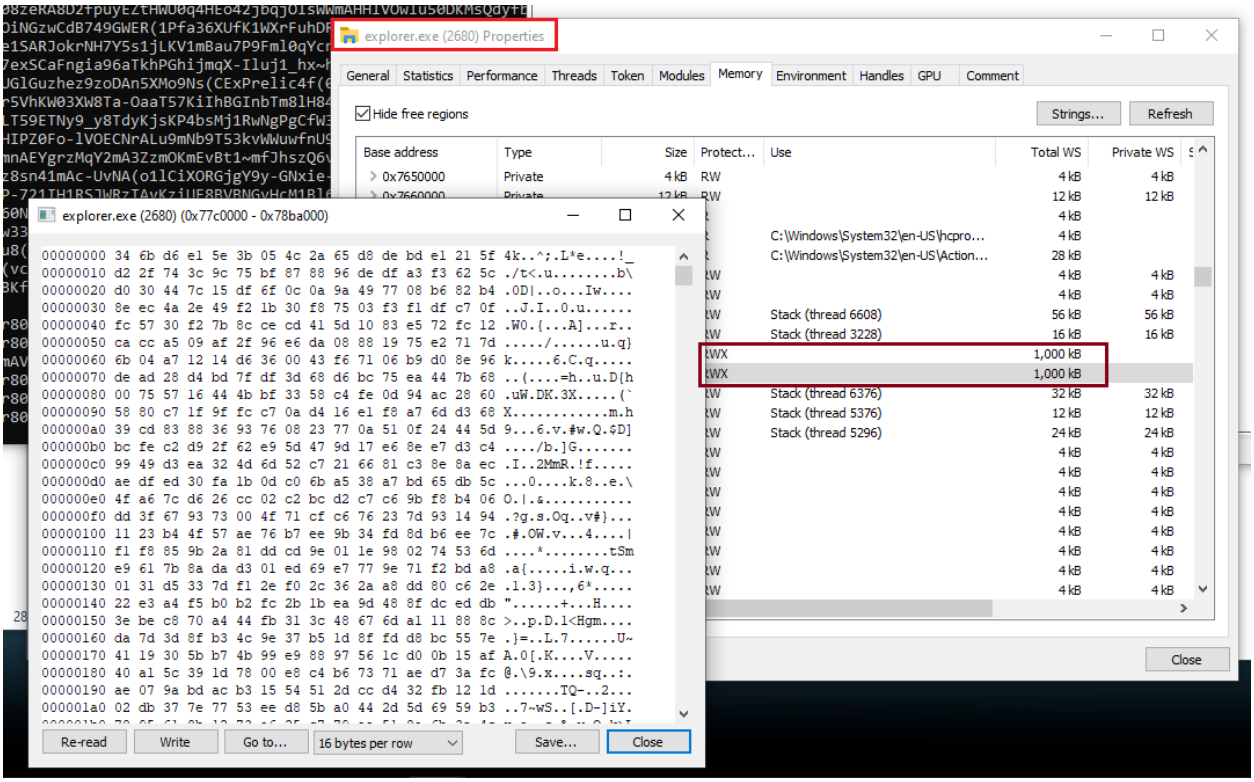
Initial Detonation:

Starting with the initial detonation of xloader. I have detonated the malware in my isolated analysis environment in the presence of procmon, wireshark and other such analysis tools. **Nothing happened!!!** Which likely suggests that there are anti-analysis techniques in the malware. I tried detonating the malware again but this time, I had **renamed** my analysis tools and the execution started.

- Process tree shows that it started another instance of itself.
- Multiple DNS & HTTP request are sent to different domains.
- Deleted itself
- Request are sent through explorer.exe







Few of the resolved domains are listed below:

- hxxp:\\www.twin68s.online
- hxxp:\\www.cicreception2023.org
- hxxp:\\www.morubixaba.com
- hxxp:\\www.gestionamostualquiler.org
- hxxp:\\www.superios.info
- hxxp:\\www.bocahkota.xyz
- hxxp:\\www.lolisex77.top
- hxxp:\\www.fhsbfjbsljsdfsd.xyz
- hxxp:\\www.mifurgoentuangar.fun
- hxxp:\\www.necessarymusthave.shop
- hxxp:\\www.abk-importexport.com
- hxxp:\\www.adoniadou.com
- hxxp:\\www.delret.tech
- hxxp:\\www.humidlandscaping.com
- hxxp:\\www.wlkwinn.net
- hxxp:\\www.8ai.ooo
- hxxp:\\www.minevisn.com
- hxxp:\\www.moheganmart.com
- hxxp:\\www.jacksonmoddy.com

Stage 1: Dropper

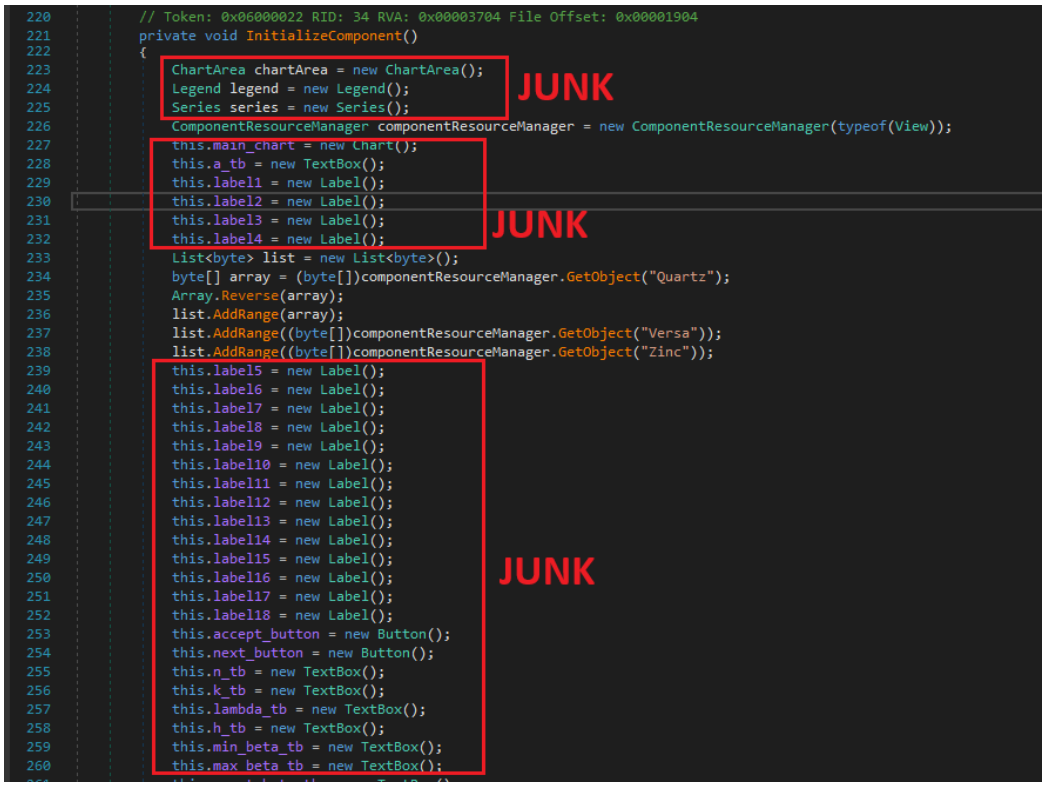
The initial dropper is a dotnet executable. It is similar to what other infostealers or RAT uses for dropping their payloads like Agent Tesla or Remcos RAT. The first step is always static analysis, which extracts suspicious strings for me and provide insight to the malware.

No	Strings	Details
1	System.Reflection	Loading assembly at run-time
2	ofnIepyTgnirtS **(StringTypeInfo)** ofnIdohteM **(MethodInfo)**	Inverted strings shows an inverted resources is embedded inside
3	.edom SOD ni nur eb tonnac margorp sihT! **(!This program cannot be run in DOS mode)**	Inverted resource is another binary
4	System.Activator	Activating assembly at run-time

The extracted strings suggest 3 main points:

- Dropper is obfuscated that loads other assemblies at run-time
- Further resources are inverted to avoid signature-based detection
- Must have more than 1 assemblies

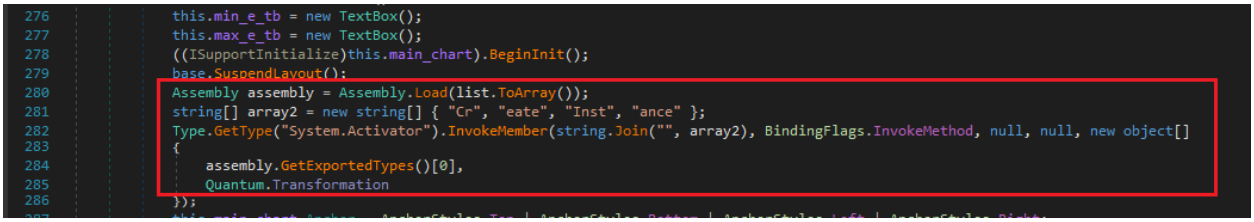
In the initial dropper, there is a lot of junk code added to divert the focus of analyst. The few lines of malicious code are spread through the whole code.



The relevant lines of code shows that malware is loading binary from 3 different resources:

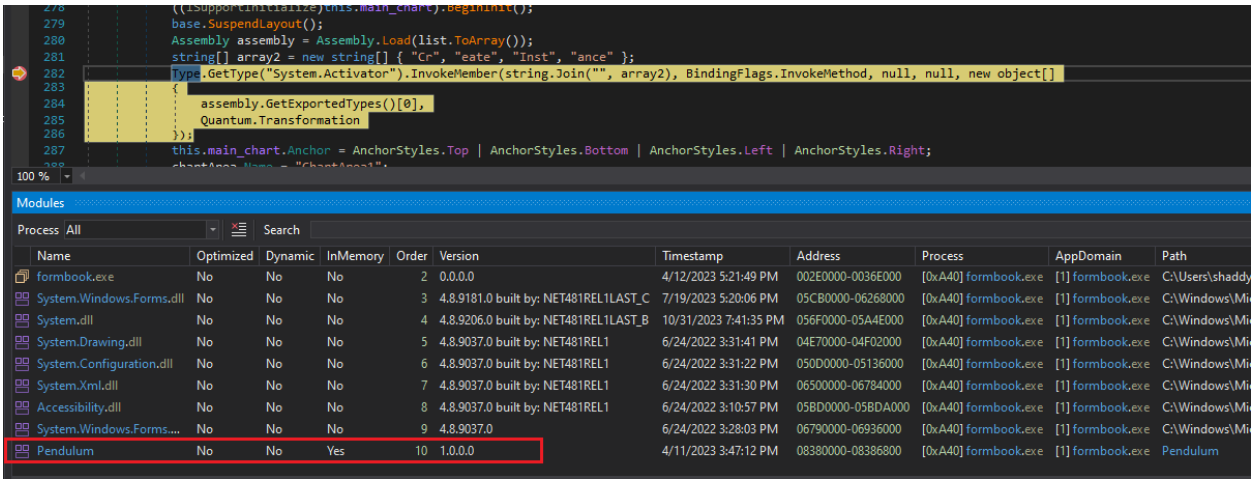
- Quartz which is also reversed
- Versa
- Zinc

These 3 are the malicious resources that are combined and loaded at run-time for further execution. After going through a lot of junk code, I came across the line of code that resolves this assembly at run-time and create instance of resource followed by loading the first method using **System.Activator** class.



Since, stage1 malware resolves assemblies at run-time and activate the method from resolved assemblies therefore static analysis is not possible ahead of this step, so I shifted to dynamic analysis.

- The runtime binary that has been loaded can be seen in the modules window.
- The name of runtime generated binary is **pendulum**. In the code, the malware is invoking the first member returned by the GetExportedTypes which means the first member of exports would be executed.
- We can locate the first function in the pendulum binary and set the breakpoint ahead to stop and debug it.



There are further binaries being resolved from the resource of first loaded DLL which is Pendulum. In the modules tab, we can trace which dlls are being added and keep following through.

- Another binary that is being loaded at run-time from the resource of pendulum is the **cruiser.dll** which could be seen in the modules window. This binary undergoes gzip decompression and loaded using Activator class.
- This binary contains a few methods called "**CausalitySource** and **SearchResult**" which performs some kind of decryption of another third resource which will also be loaded on runtime.

```
1088
1089        // Token: 0x06000005 RID: 5 RVA: 0x00002238 File Offset: 0x00000438
1090        public static void Dodge(string StringTypeInfo, string InputBlockSize, string EscapedIRemotingFormatter)
1091        {
1092            Thread.Sleep(44102);
1093            Type type = Canvas.GlobalAssemblyCache(Canvas.Magnatic()).GetType("Munoz.Himentater");
1094            object obj = Activator.CreateInstance(type);
1095            StringTypeInfo = (string)type.GetMethod("CausalitySource").Invoke(obj, new object[] { StringTypeInfo });
1096            InputBlockSize = (string)type.GetMethod("CausalitySource").Invoke(obj, new object[] { InputBlockSize });
1097            Bitmap bitmap = Canvas.LowestBreakIteration(StringTypeInfo, EscapedIRemotingFormatter);
1098            byte[] array = Canvas.NamedArguments(Canvas.RestoreOriginalBitmap(bitmap, 150, 150));
1099            array = (byte[])type.GetMethod("SearchResult").Invoke(obj, new object[] { array, InputBlockSize });
1100            Assembly assembly = Canvas.GlobalAssemblyCache(array);
1101            Canvas.ParsingState(assembly);
1102            Environment.Exit(0);
1103        }
1104
1105        // Token: 0x06000006 RID: 6 RVA: 0x0000230C File Offset: 0x0000050C
```

100 %

Modules

Process All | Search

Name	Optimized	Dynamic	InMemory	Order	Version	Timestamp	Address	Process	AppDomain	Path
System.Xml.dll	No	No	No	7	4.8.9037.0 built by: NET481REL1	6/24/2022 3:31:30 PM	06500000-06784000	[0xA40] formbook.exe	[1] formbook.exe	C:\Windows\Micr
Accessibility.dll	No	No	No	8	4.8.9037.0 built by: NET481REL1	6/24/2022 3:10:57 PM	05BD0000-05BDA000	[0xA40] formbook.exe	[1] formbook.exe	C:\Windows\Micr
System.Windows.Forms....	No	No	No	9	4.8.9037.0	6/24/2022 3:28:03 PM	06790000-06936000	[0xA40] formbook.exe	[1] formbook.exe	C:\Windows\Micr
Pendulum	No	No	Yes	10	1.0.0.0	4/11/2023 3:47:12 PM	08380000-08386800	[0xA40] formbook.exe	[1] formbook.exe	Pendulum
Cruiser	No	No	Yes	11	5.0.0.0	4/10/2023 3:01:02 AM	00820000-00825C00	[0xA40] formbook.exe	[1] formbook.exe	Cruiser

- The last resource that has been decrypted and loaded is called **Discompard.dll**.
- In the method of ParsingState, it could be seen that a method from this assembly is being called for further execution of malware.

```
1100            Assembly assembly = Canvas.GlobalAssemblyCache(array);
1101            Canvas.ParsingState(assembly);
1102            Environment.Exit(0);
1103        }
1104
1105        // Token: 0x06000006 RID: 6 RVA: 0x0000230C File Offset: 0x0000050C
1106        private static void ParsingState(object TP)
1107        {
1108            Type type = ((Assembly)TP).GetTypes()[20];
1109            MethodInfo methodInfo = type.GetMethods()[29];
1110            methodInfo.Invoke(null, null);
1111        }
```

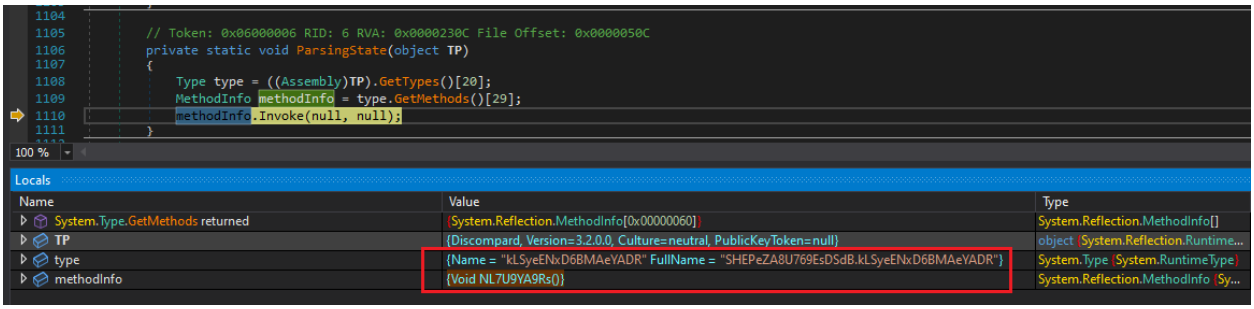
100 %

Modules

Process All | Search

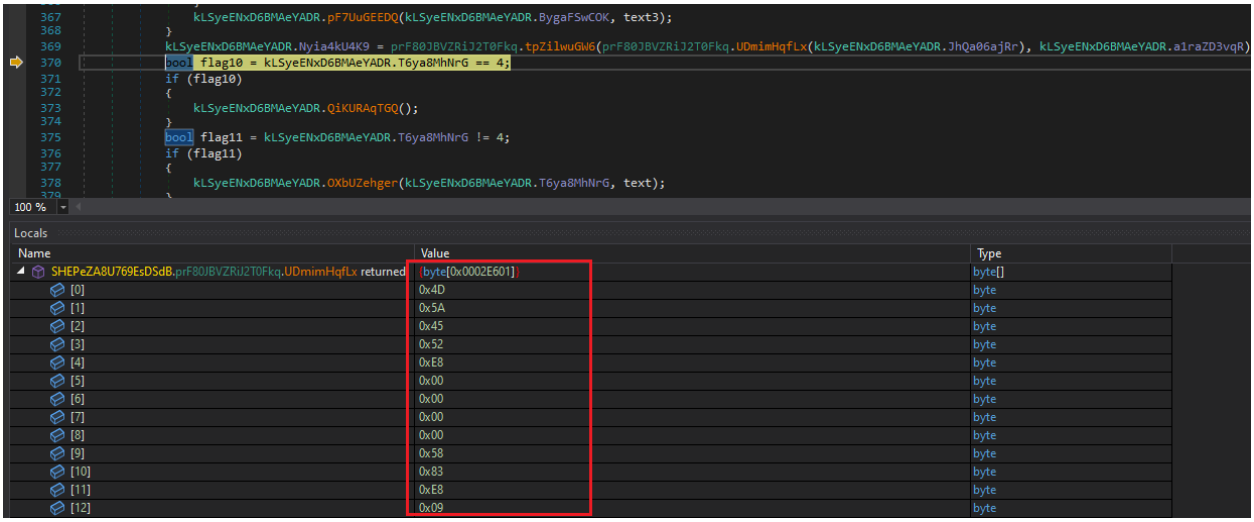
Name	Optimized	Dynamic	InMemory	Order	Version	Timestamp
System.dll	No	No	No	4	4.8.9206.0 built by: NET481REL1LAST_B	10/31/2023 7:41:35 PM
System.Drawing.dll	No	No	No	5	4.8.9037.0 built by: NET481REL1	6/24/2022 3:31:41 PM
System.Configuration.dll	No	No	No	6	4.8.9037.0 built by: NET481REL1	6/24/2022 3:31:22 PM
System.Xml.dll	No	No	No	7	4.8.9037.0 built by: NET481REL1	6/24/2022 3:31:30 PM
Accessibility.dll	No	No	No	8	4.8.9037.0 built by: NET481REL1	6/24/2022 3:10:57 PM
System.Windows.Forms....	No	No	No	9	4.8.9037.0	6/24/2022 3:28:03 PM
Pendulum	No	No	Yes	10	1.0.0.0	4/11/2023 3:47:12 PM
Cruiser	No	No	Yes	11	5.0.0.0	4/10/2023 3:01:02 AM
Discompard	No	No	Yes	12	3.3.0.0	4/12/2023 5:21:39 PM

- We can also see the names of classes and methods that are being called from this assembly in the locals. Using this information, we can then setup another breakpoint in the **Discompard.dll** method and continue debugging the 3rd resource.
- Again, we can explore the third binary and setup a breakpoint on the function that it tries to call.

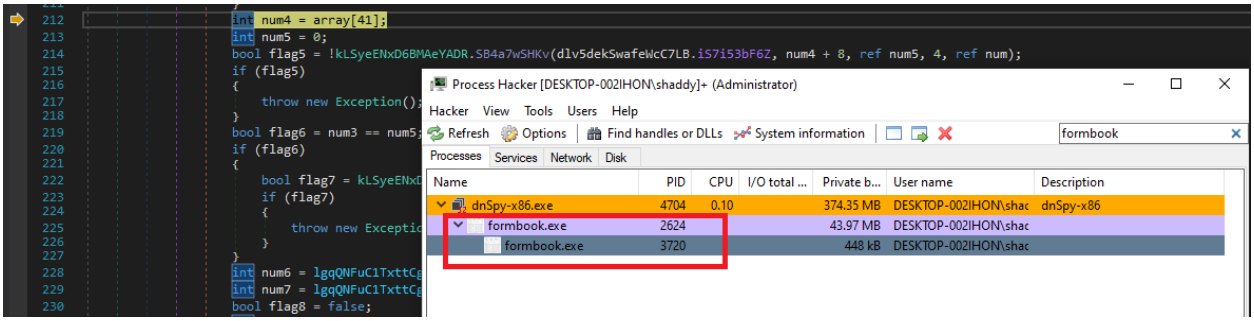


We have now entered the method called by the previous dll. This binary is highly obfuscated with random variable and class names. Normally, what I do is that I check if a deobfuscator like de4dot or some other tool is able to deobfuscate such a binary. If it is possible then I patch the resource and continue my debugging with the deobfuscated version. But in this case, it is very tricky because this resource is dependent upon two other binaries that are being called first and to patch all these will be such a headache. So, I decided to move forward with the obfuscated version and see if I could understand what it is doing from the local variables and return values.

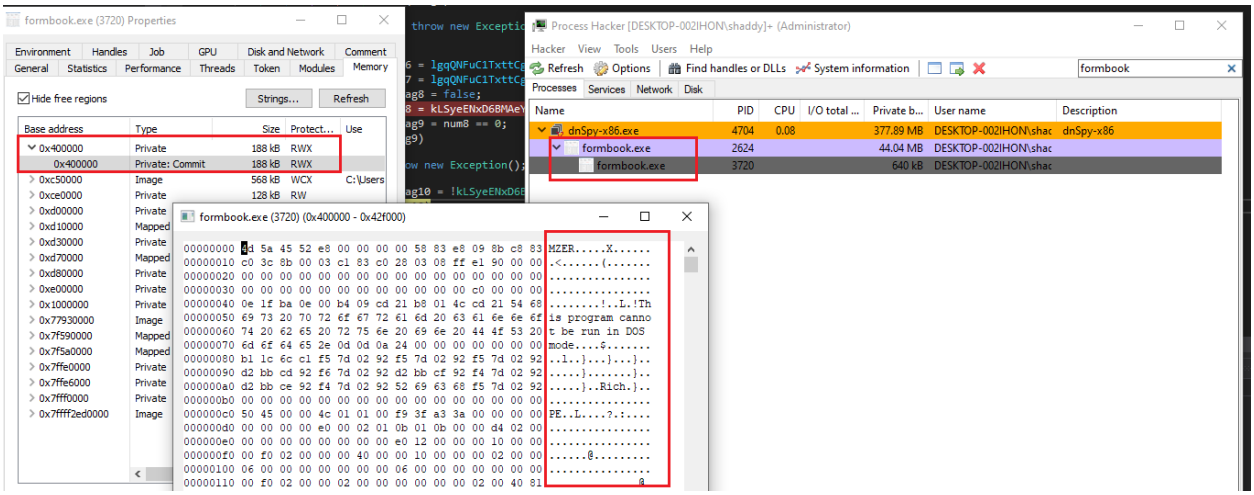
- I kept stepping over and checking the variables and function returns.
- It skipped most of the flags but then I stepped over a function and a return value shows that another binary has been returned. The MZ bytes (4D 5A) could be seen in the array.



- It confirms that this malware might perform some kind of injection or dump the binary in a file and execute it as a 2nd stage malware.
- I stepped into a function that is obfuscated but it looks like it is performing **process hollowing**, as the malware opens itself in a suspended state and ready to inject in the address space of this process.



- Stepped over few of the functions while checking RWX memory region of the process
- At one point it reserved the memory and then started writing shellcode into that memory in chunks
- It changes the execution of base image to the injected shellcode and finally resume the process using ResumeThread API.

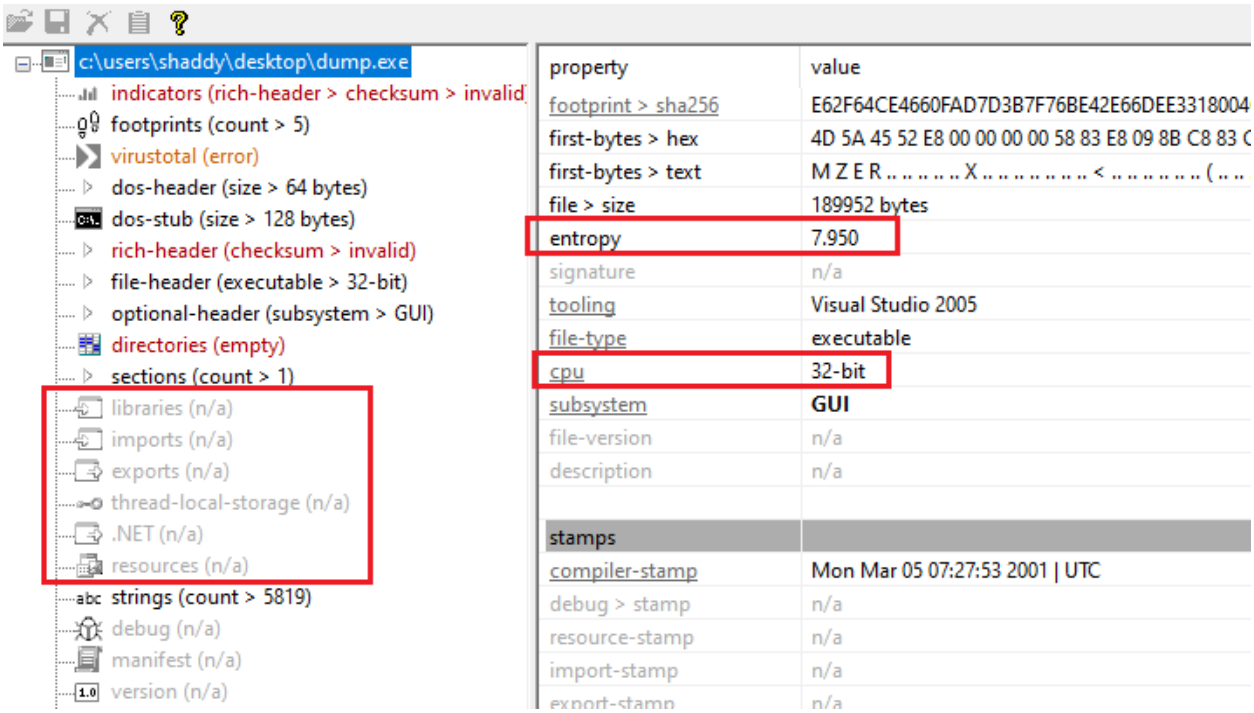


- This is the exact behavior of process hollowing.
- I dumped this shellcode to analyze the malware separately as a second stage payload.
- The stage2 malware is the real xloader payload.

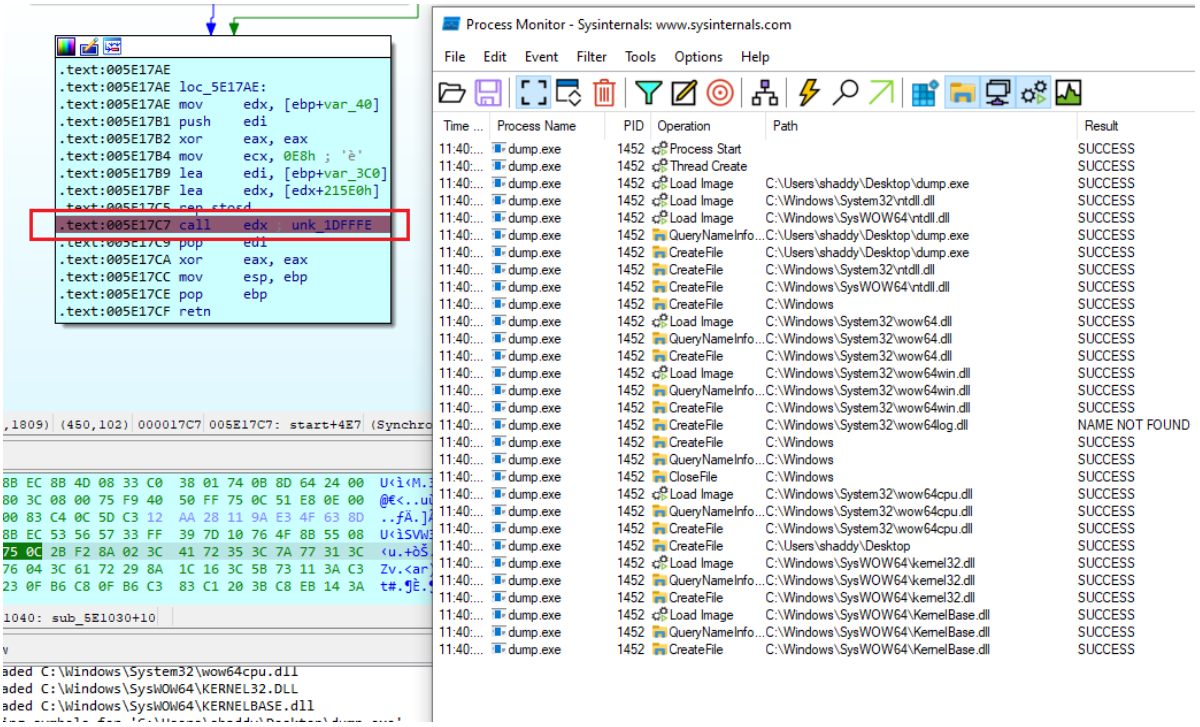
Stage2: Xloader 4.3

Xloader is an infostealer malware that is the updated version of Formbook malware. It is sold on dark web for cheap prices with a MaaS architecture (Malware-as-a-Service). The authors of this malware put great effort in adding latest defense evasion techniques.

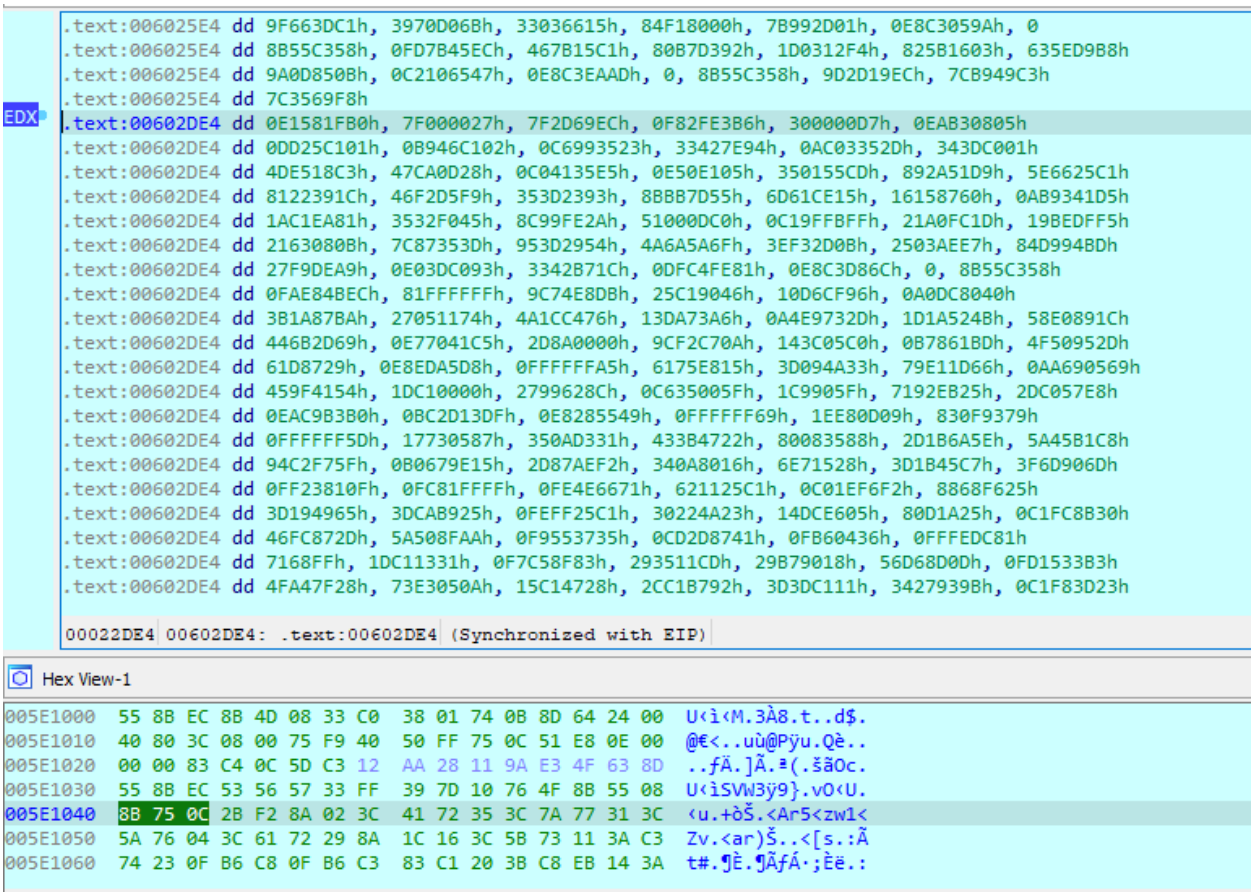
- Xloader aka Formbook is written in pure native assembly with a combination of c language
- The entropy is very high which suggests that there is embedded code or it might be packed
- There are 0 libraries, imports, strings found in this payload
- There are no valid strings other than the DOS message



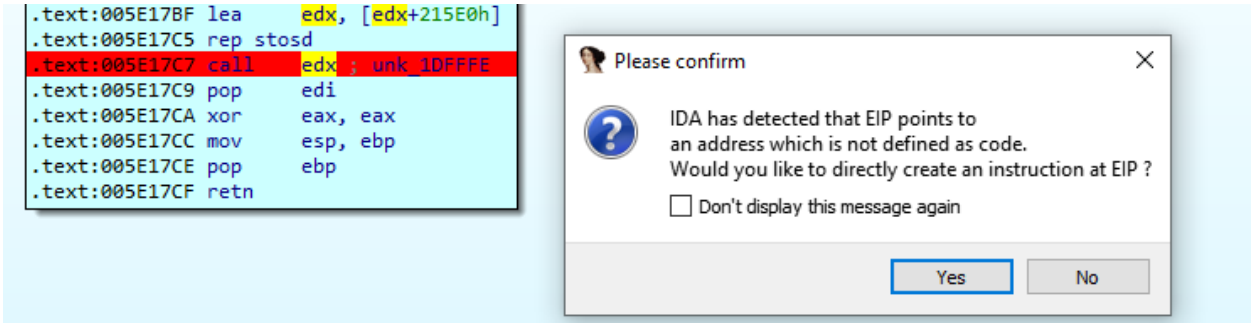
- The start of malware is fairly simple, it loads some necessary libraries before going to the malicious code
- It also performs some other kind of computations, probably decompressing some of its malicious code
- After the calculations, I came across a call to edx which leads to an unidentified code



- The **"call edx"** instruction moves the program flow to a set of native assembly which is unidentified by IDA at this moment
- This means that, the code to which edx register now points was not understood by IDA which indicates that it might be encrypted at first
- From there the execution of real formbook payload starts



- IDA resolves this chunk of assembly at run-time to continue debugging this dump.
- This is one of the many anti-analysis techniques added in the xloader payload.

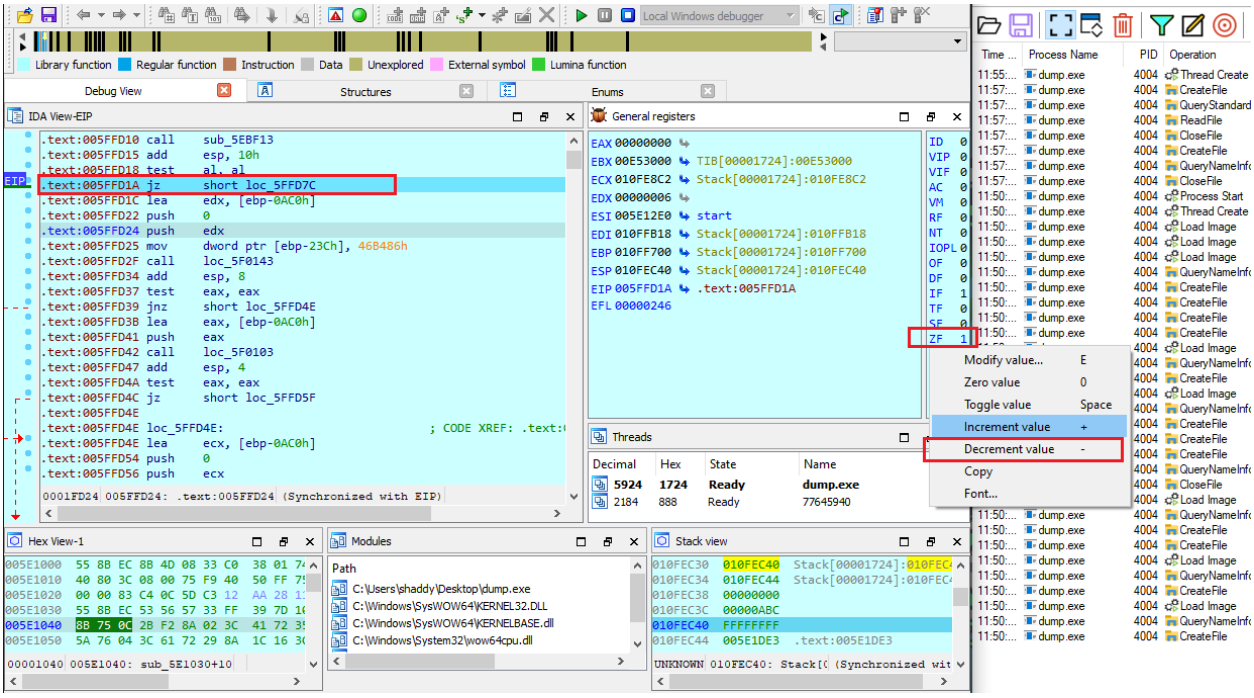


After going through the newly resolved chunk of code, my program exited without doing anything else. I understood that there are anti-analysis techniques involved in this malware. So, my battle started with defeating anti-analysis techniques provided in the section below.

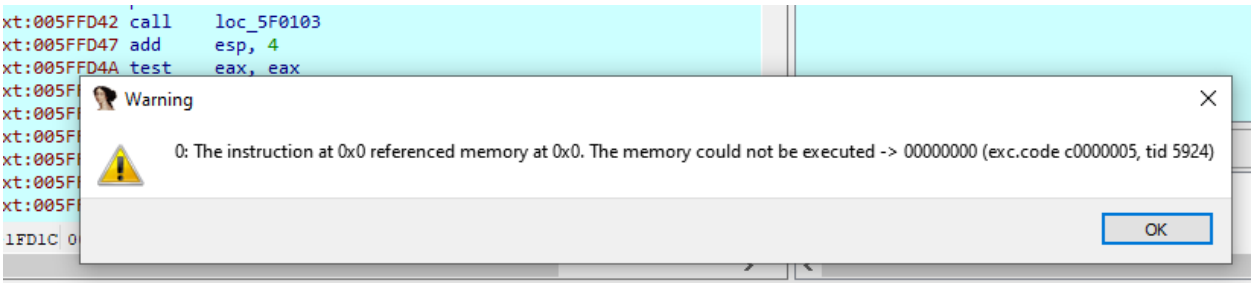
Defeating Anti-Analysis:

TAKE # 1: FAILED

- In first take, I simply changed the jump condition to divert the program from exiting the malware to continue with the actual program flow
- Changed the zero flag from 1 to 0 which sets the condition appropriately to let the program continue



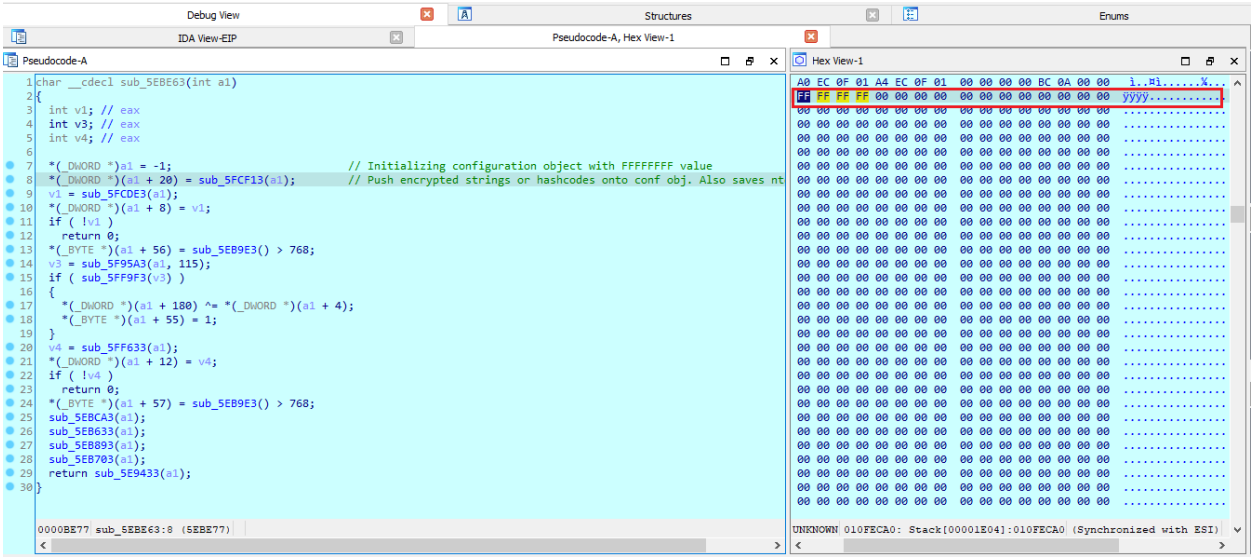
- It continues the program, however it throughs exception right after stepping over a few functions.
- This patch will not work
- The malware is dependent upon the values that this flag is setting somewhere



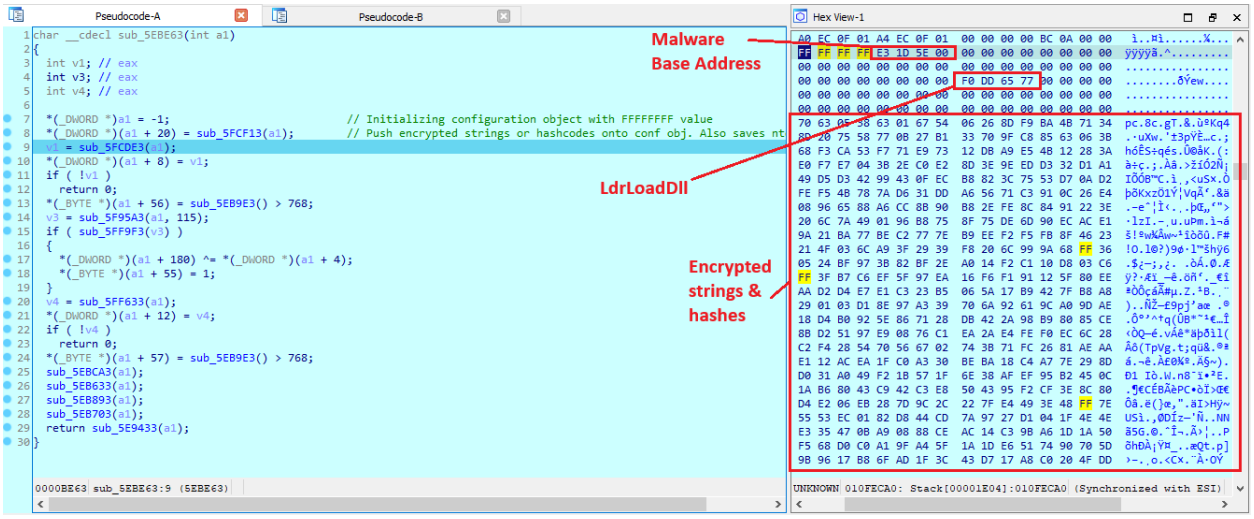
TAKE # 2: FAILED

The configuration object:

- Xloader payload initializes a configuration object on which it bases most of its execution flow
- The configuration obj is initialized with FFFFFFFF value and after that each function contributes to it.
- Some encrypted values are pushed onto this configuration object.

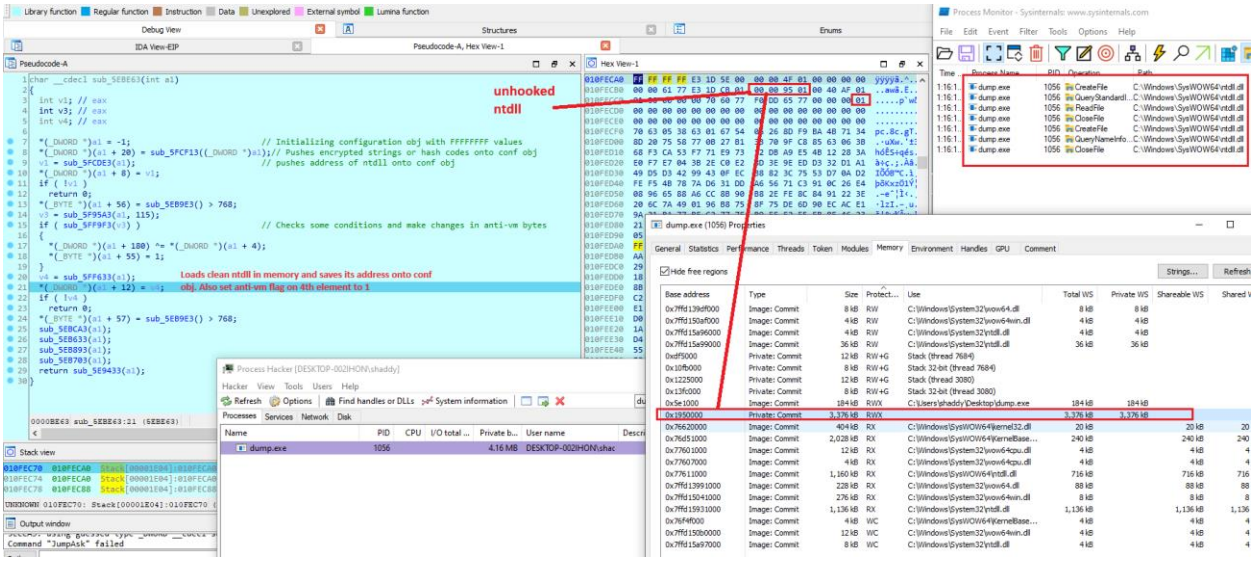


- The first function, saves lots of encrypted strings or hash codes. The purpose of these will be cleared later on in the execution
- Next to FF values, the base address of executing malware is saved
- On the third line another address is stored which is actually the address of **LdrLoadDll** function from ntdll. This will be used to laod further libraries



- I stepped over each function and monitored changes in memory side by side.
- Every function is contributing to the conf obj.

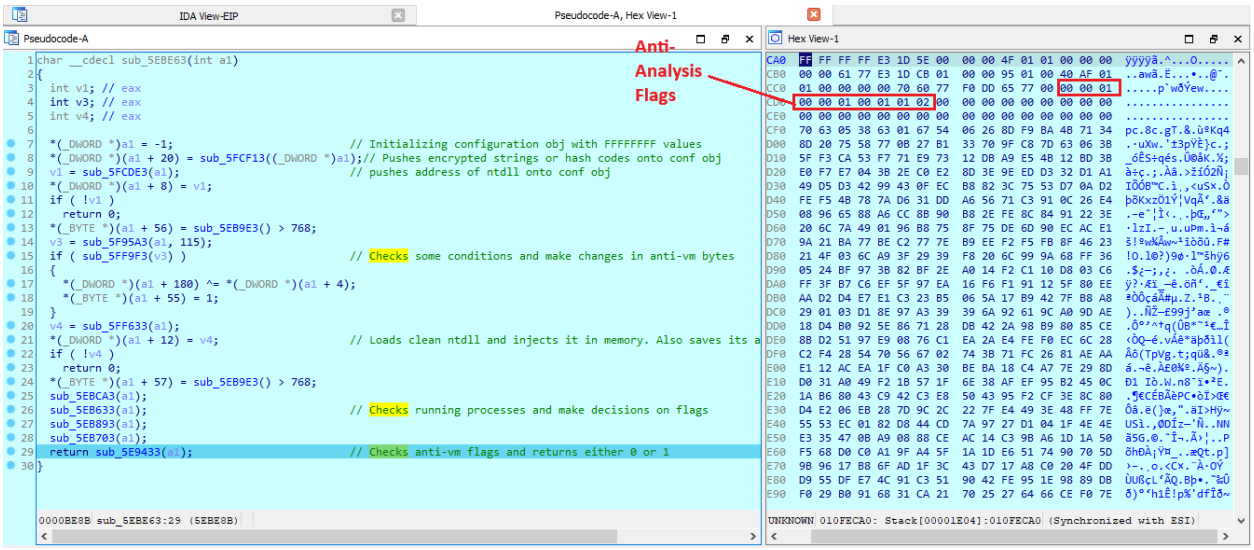
- The function in the screenshot below is loading a clean ntdll in the memory and saves it address on the conf obj
- Also, it is setting value in anti-vm flags that starts from the 45th element of the conf obj.
- The address of injected ntdll in memory starts on **0x1950000** and similarly in the 4 bytes after 24th element we have the address of injected ntdll saved.
- The flag value of 1 is also set in anti-vm flags.



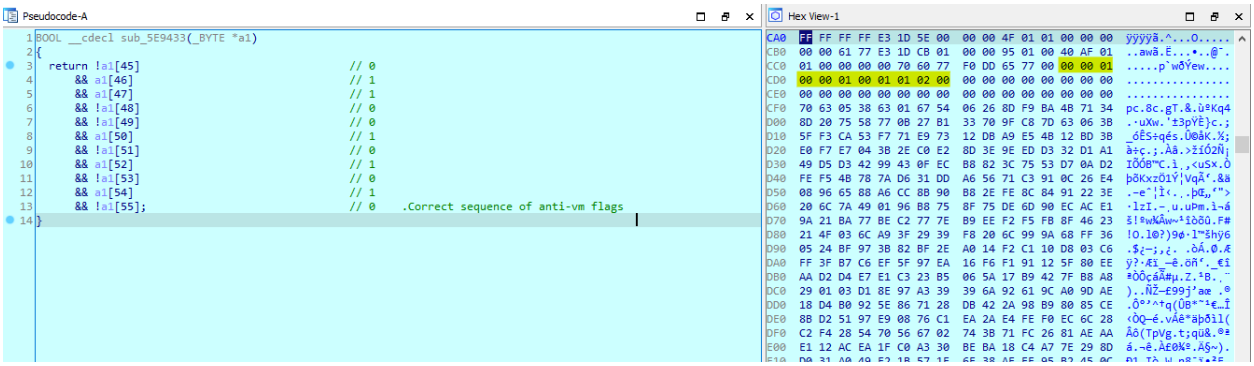
- Continuing with the execution.
- It checks other anti-vm checks
- Like taking snapshot of running processes and filtering out if any of those processes are listed by the malware
- In the screenshot, we can see that it detected **procmon** in running processes

Address	Length	Result
0xdfe744	54	\Windows\SysWOW64\ntdll.dll
0xdfeb00	58	C:\Windows\SysWOW64\ntdll.dll
0x10fe808	11	procmon.exe
0x10fe94c	22	svchost.exe
0x10feb54	11	Procmon.exe
0x10feee3	10	{3]l:JW*h
0x1110b3c	80	C:\Program Files\IDA Pro 7.5 SP3\ida.exe

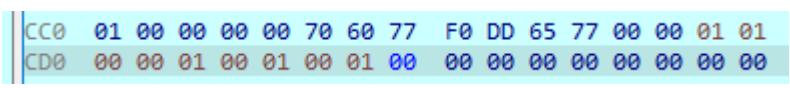
- After performing some of the anti-vm checks, it updated the flags on anti-analysis bytes as shown in screenshot below:



- The last function is matching the anti-vm flags with the sequence it requires to progress.
- As can be seen in the screenshot, my sequence doesn't match to what it should be,
- It means the malware has either **detected the debugger** or tools like **procmon** or some other parameter
- Therefore, the program exits.

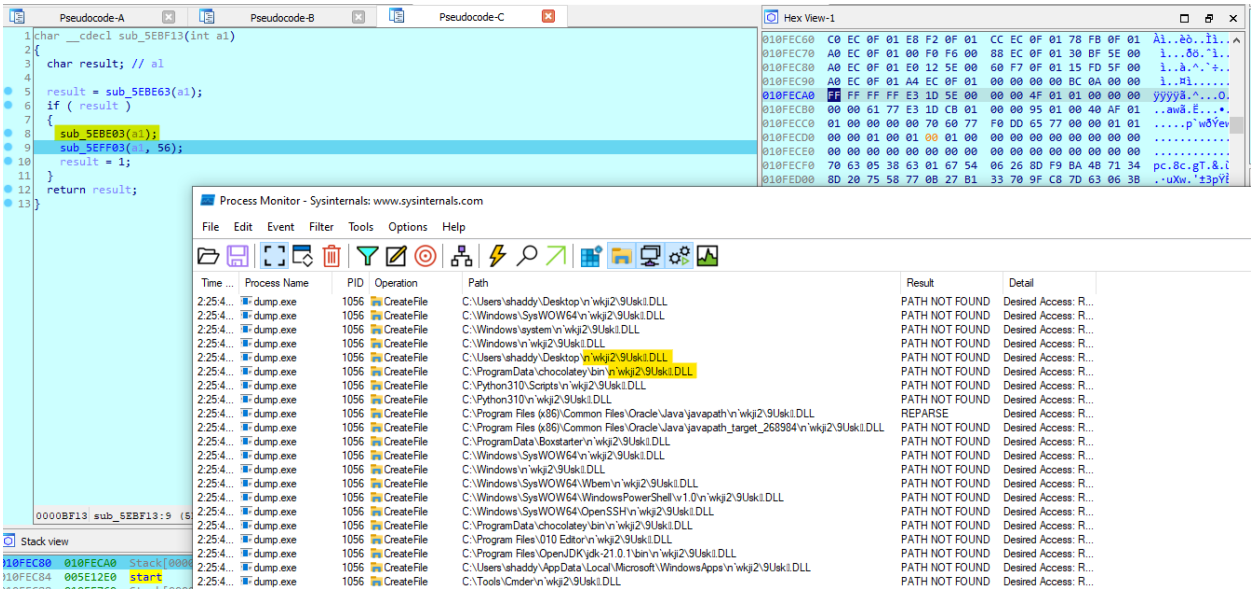


- So, in take # 2 of defeating anti-reverse engineering or anti-vm techniques, I simply patched the sequence of these flags in the memory to the required sequence.
- Patching memory, and moving onto the execution should work, because these flags are being used somewhere ahead in the program. So, simply changing the conditional jump would always crash the program.
- However, in case of memory patch, these values would be continued in the program and this issue should be fixed.

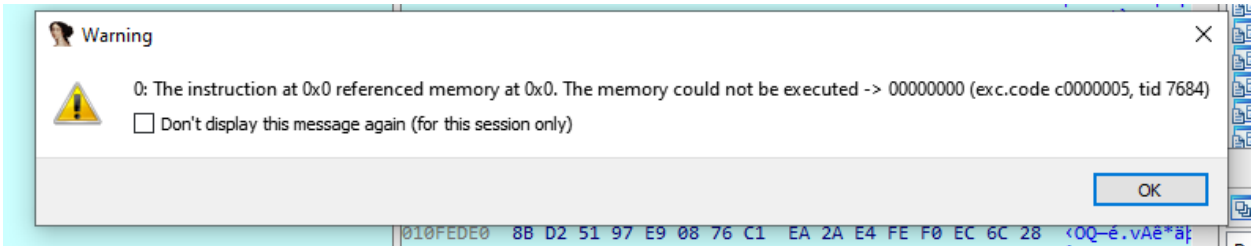


- Patched the memory and now it goes back to the condition which is true
- However, something is wrong here.
- Because the names of the dll being searched is very weird.

- Now I understand, that these sequences of bytes are being used in a decryption algorithm to decrypt the names of libraries and APIs.
- But since I patched the bytes in memory, it should have been able to decrypt accurately which it is not. That means that the sequence is used somewhere else before performing the anti-analysis check.



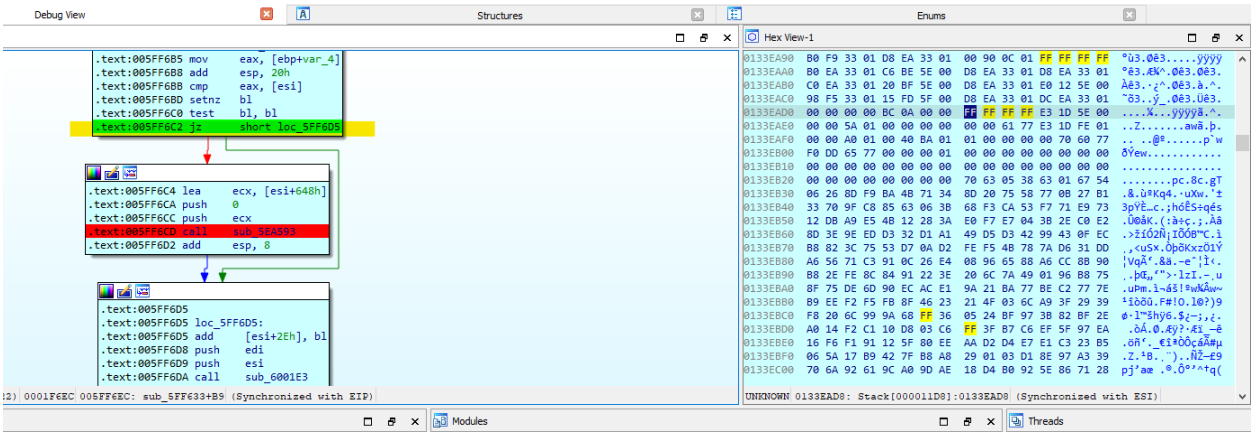
- I let the malware continue and again it crashed, because it was not able to decrypt its configuration and hence looking for encrypted dll names.
- So that means, I might be missing some important function and because it is detecting the debugger, it would be skipping some important function.



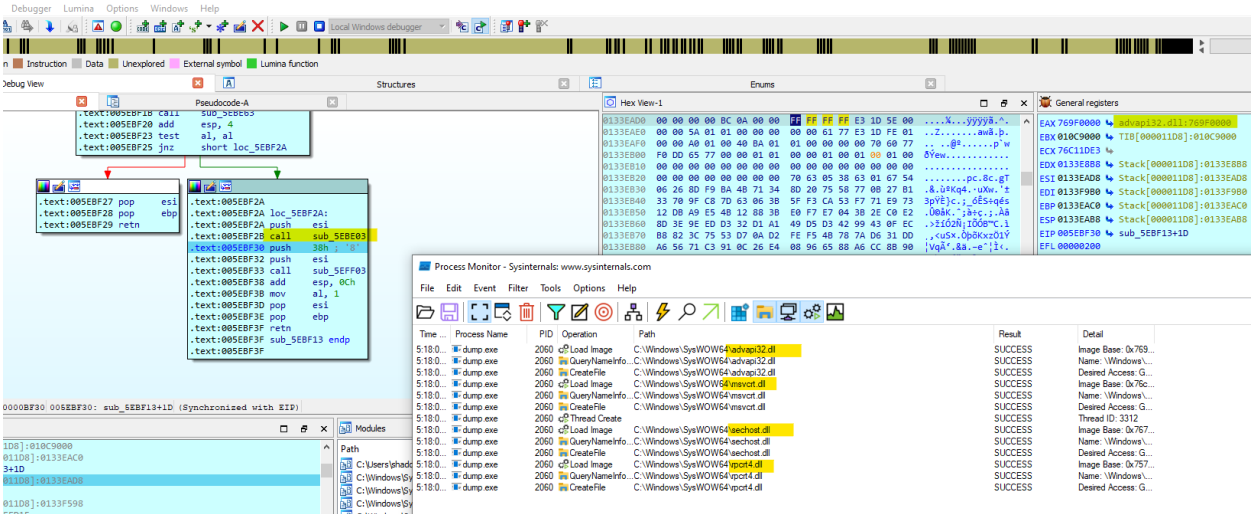
TAKE # 3: PASSED

- In third take, I have debugged a lot of the code and finally, found the function over which the program was skipping because of a single flag condition not being met.
- So, I changed the values of condition to allow it to execute as well as changed the value of register that was being pushed to the **conf obj**.
- In my environment, there were always 3 flags that were changed. The value on the third element was 0 however it should be 1, and the two elements at 11,12th position.

- I also know that those two were changed because of procmon and other such analysis tools. So, it is easier to just change the name of procmon and continue.
- Instead of applying memory patches, I have changed the values at run-time before they were pushed onto the memory stack and **voila**, the malware executed perfectly without any exceptions.



- Now this time, I stepped over the function that loads libraries and instead of encrypted names, the full names of libraries have been seen and successfully loaded as can be seen in procmon.
- I let the program continue without any other interaction and the debugger exited with status code 0, which means now there is no exception.
- However, it still hasn't performed all the functionality which indicates there are **more anti-analysis techniques** ahead.

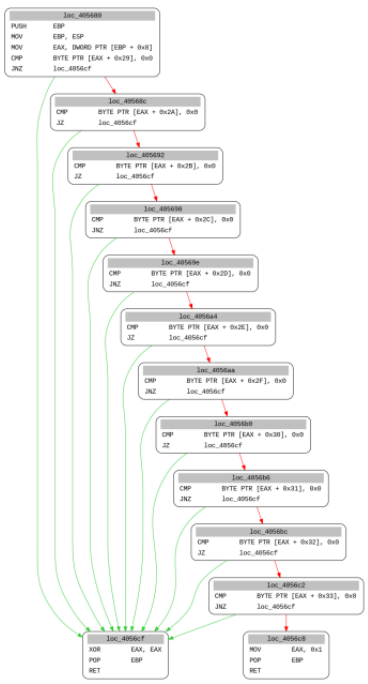


Output window

```
77645940: thread has started (tid=3172)
75700000: loaded C:\Windows\SysWOW64\RPCRT4.dll
77645940: thread has started (tid=4144)
75560000: loaded C:\Windows\SysWOW64\user32.dll
77290000: loaded C:\Windows\SysWOW64\win32u.dll
76700000: loaded C:\Windows\SysWOW64\GDI32.dll
768D0000: loaded C:\Windows\SysWOW64\gdi32full.dll
77480000: loaded C:\Windows\SysWOW64\msvcp_win.dll
767B0000: loaded C:\Windows\SysWOW64\ucrtbase.dll
76020000: loaded C:\Windows\SysWOW64\IMM32.DLL
Debugger: thread 4144 has exited (code 0)
Debugger: thread 3172 has exited (code 0)
Debugger: process has exited (exit code 0)
```

I found a very good resource, that explains all the flags that previous formbook version looked for in its analysis. Luckily in the latest xloader, it is still using a similar approach and we can map those flags easily. The following slide shows all the anti-analysis flags that the xloader uses in its configuration.

Checking anti-analysis tests results



1. WOW32 Reserved hook
2. Software debugger
3. Kernel debugger
4. Blacklisted base file name
5. Blacklisted username
6. Blacklisted username
7. Blacklisted loaded module path
8. Blacklisted loaded module path
9. Blacklisted running process
10. Blacklisted running process
11. Blacklisted loaded DLL

5 . 16

Reference: https://www.botconf.eu/botconf-presentation-or-article/in-depth-formbook-malware-analysis/

Decryption/Deobfuscation Routine:

Xloader relies heavily on encryption and obfuscation to avoid being detected from EDR solutions. There is multi-layered encryption performed on its code. The APIs are all hashes, the string and libraries are also hashes. Even the hashes are encrypted in the conf obj. The core functions of xloader are all encrypted and decrypted at run-time after anti-analysis checks are cleared.

Decrypting Library Names:

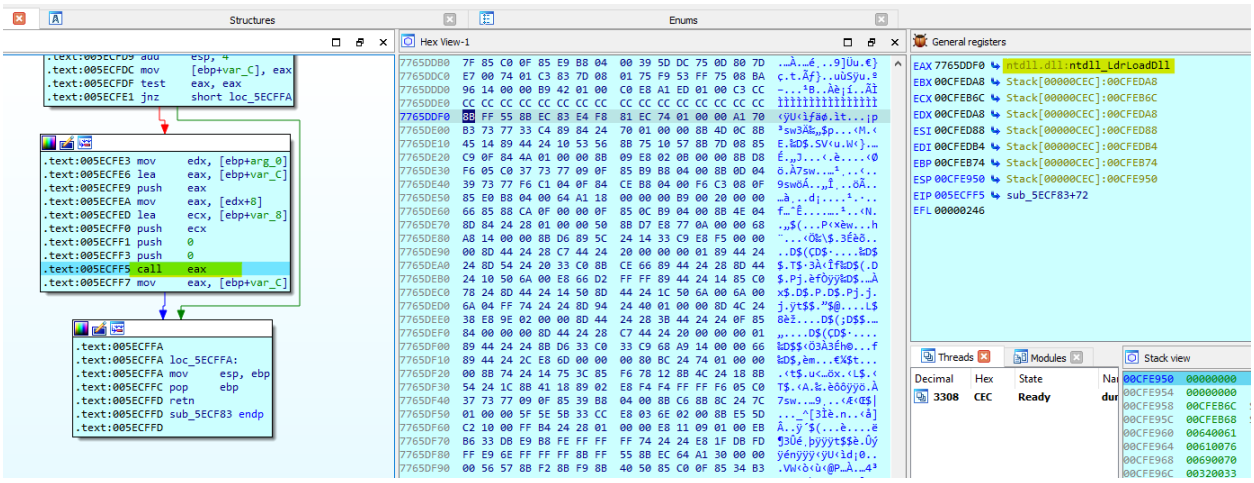
- The decryption routine starts, I stepped through the next function after anti-vm checks have been cleared and it looks like the anti-vm flag bytes are used as decryption seed value.
- The library names are being decrypted one by one.

```
00CFEBA0  ED A4 57 F9 24 64 7E EE  76 C5 B4 C9 BE 17 3A F7  íWù$d~îvÅ´É¾.:÷
00CFEBB0  A5 13 49 6D 95 31 38 D5  86 5F E3 04 C8 E5 DF 58  ¥.Im•18Õ†_ã.Èåß
00CFEBC0  EB 35 1F 8C A9 45 03 3C  9B 90 EA 8A D2 A6 E2 6B  ë5.Œ©E.<›.êŠÒ¦âk
00CFEBD0  6C 4E AF BD D7 F4 B0 FB  FA FD 56 53 42 59 FC 66  lN¯½×ô°ûúýVSBYüf
00CFEBE0  8F 7C 36 9A B8 CD EC 48  BA 91 C0 7D 05 30 47 34  .|6š¸ÍìHº'À}.0G4
00CFEBF0  2C 39 D3 3E 0A 2F 82 E1  46 D8 20 87 B6 1A FE 16  ,9Ó>./,áFØ·‡¶.þ.
00CFEC00  D6 7A 85 28 0D DE A0 10  93 D4 07 80 B5 8B 92 08  Öz…(.Þ .“Ô.€µ‹’.
00CFEC10  6A 02 E6 0C 4B 9D 65 3A  50 00 00 D1 38 EC CF 00  j.æ.K.e:P..Ñ8ìÏ.
00CFEC20  C7 CE 5E 00 78 EC CF 00  0D 00 00 00 7C ED CF 00  ÇÎ^.xìÏ.....|íÏ.
00CFEC30  58 EE CF 00 2C EE CF 00  E4 ED CF 00 8B F9 5F 00  XîÏ.,îÏ.äíÏ.‹ù_.
00CFEC40  78 EC CF 00 0D 00 00 00  7C ED CF 00 7C ED CF 00  xìÏ.....|íÏ.|íÏ.
00CFEC50  7C ED CF 00 58 EE CF 00  00 00 00 00 7C ED CF 00  |íÏ.XîÏ.....|íÏ.
00CFEC60  78 EC CF 00 74 F4 CF 00  0D 00 00 00 79 EC CF 00  xìÏ.tôÏ.....yìÏ.
00CFEC70  00 00 00 00 03 01 00 00  6B 65 72 6E 65 6C 33 32  ........kernel32
00CFEC80  2E 64 6C 6C 00 00 00 00  00 00 00 00 00 00 00 00  .dll............
00CFEC90  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00CFECA0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
```

Hex View-1

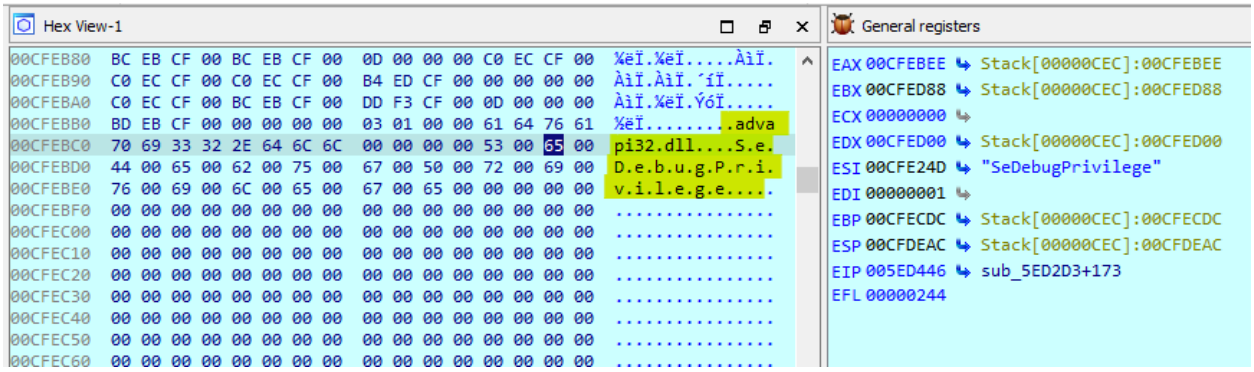
```
00CFEC20  CC ED CF 00 8B F9 5F 00  60 EC CF 00 0D 00 00 00  ÌíÏ.‹ù_.`ìÏ.....
00CFEC30  64 ED CF 00 64 ED CF 00  64 ED CF 00 58 EE CF 00  díÏ.díÏ.díÏ.XîÏ.
00CFEC40  00 00 00 00 64 ED CF 00  60 EC CF 00 81 F4 CF 00  ....díÏ.`ìÏ..ôÏ.
00CFEC50  0D 00 00 00 61 EC CF 00  00 00 00 00 03 01 00 00  ....aìÏ.........
00CFEC60  61 64 76 61 70 69 33 32  2E 64 6C 6C 00 00 00 00  advapi32.dll....
00CFEC70  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00CFEC80  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00CFEC90  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00CFECA0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
```

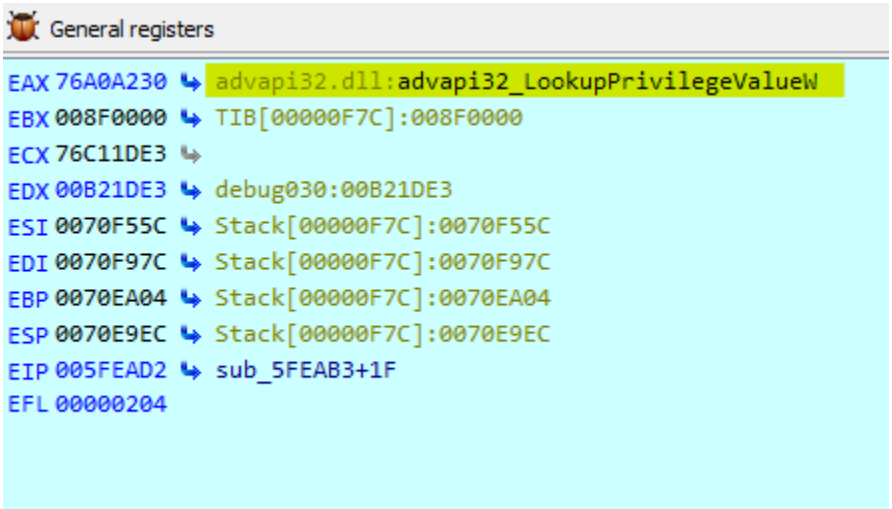
- These libraries are then loaded by the native function **LdrLoadDll**



Decrypting API Names:

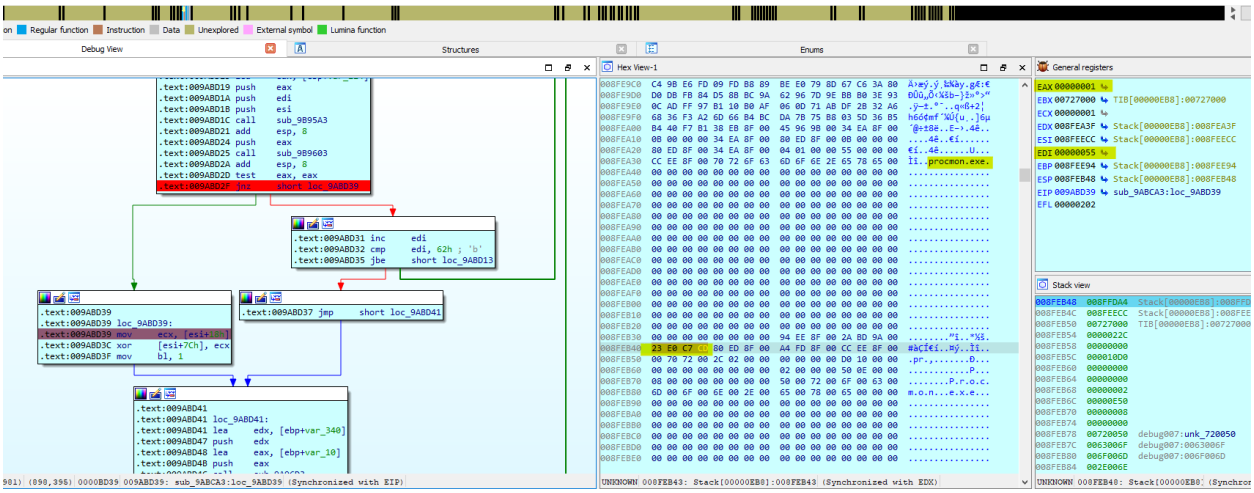
- Some of the APIs that are being decrypted suggests that it looks for further **Process Injection**
  - ❖ LookupPrivilegeValueW
  - ❖ SeDebugPrivilege
  - ❖ AdjustPrivilegeToken



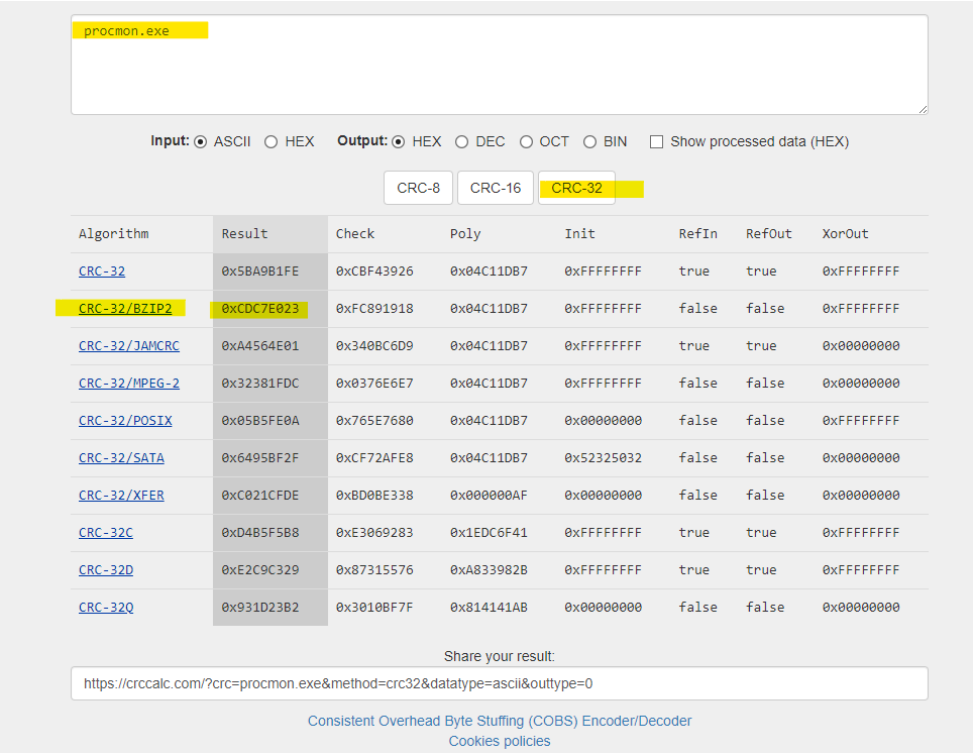


Computing String Hashes:

- There is a hashing algorithm used for strings, apis etc.
- It loads all the string hashes and compare the running processes with each hash value, if it finds any such process, it adds desired value on the anti-vm flag on conf obj.
- In the screenshot below, it is checking the process name hash with the value of pre-defined set of hashes that it stored.



- The hash value that is it is comparing to is **23 E0 C7 CD** which in hex is (0xCDC7E023).
- I have checked 32-bit hashing algorithms by calculating the hash of procmon and found the hashing algorithm that it uses.
- It uses **CRC-32/BZIP2** hashing for its strings

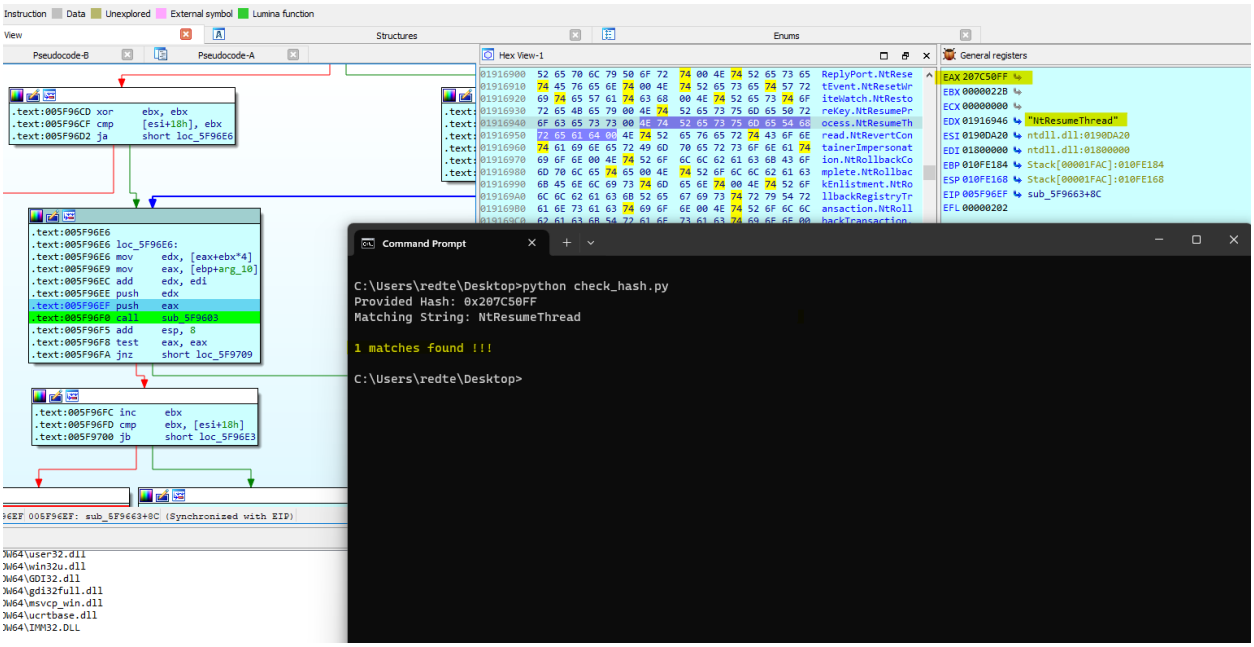


All the hashes that it checks are listed below:

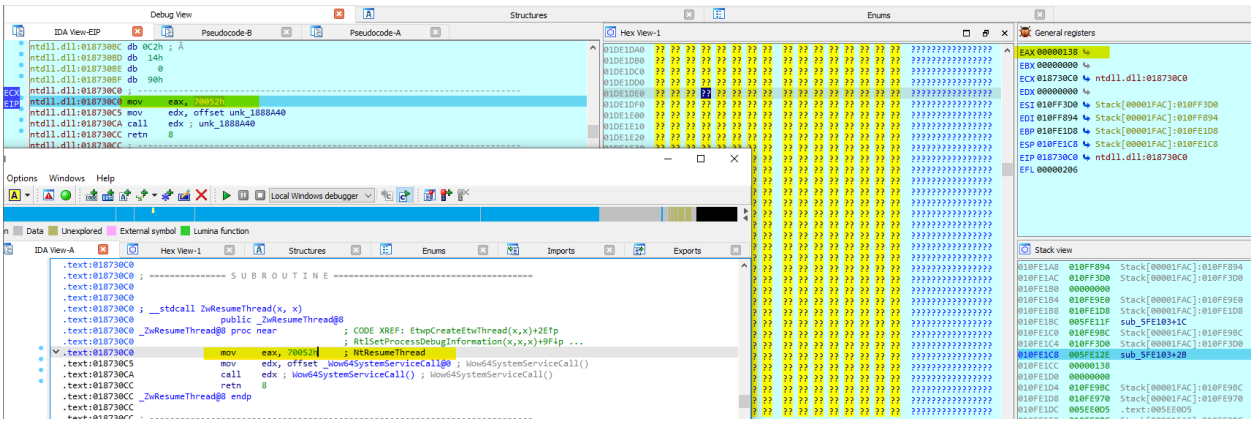
1	86 90 BE 3E	0x3EBE9086	vmwareuser.exe
2	B5 DD 6F 4C	0x4C6FDDB5	vmwareservice.exe
3	3E B1 6D 27	0x276DB13E	vboxservice.exe
4	8E 0A 0F E0	0xE00F0A8E	vboxtray.exe
5	04 94 CF 85	0x85CF9404	sandboxiedcomlaunch.exe
6	84 87 24 B2	0xB2248784	sandboxierpcss.exe
7	23 E0 C7 CD	0xCDC7E023	procmon.exe
8	50 5F 1F 01	0x011F5F50	filemon.exe
9	1C BC D4 1D	0x1DD4BC1C	wireshark.exe
10	E2 FC 35 82	0x8235FCE2	netmon.exe
11	D5 E2 2C C7	0xC72CE2D5	--
12	8B 17 63 02	0x0263178B	--
13	56 53 58 57	0x57585356	--
14	40 52 B9 9C	0x9CB95240	sharedintapp.exe
15	EF 9F C3 0C	0x0CC39FEF	--
16	57 AC 47 93	0x9347AC57	vmsrvc.exe
17	DC 22 95 9D	0x9D9522DC	vmusrvc.exe
18	0E C7 1B 91	0x911BC70E	python.exe
19	B9 3D 44 74	0x74443DB9	perl.exe
20	A9 1A 4C F0	0xF04C1AA9	regmon.exe

Computing API Hashes:

- Similar to strings hashes
- The APIs that are being loaded from injected **ntdll** are also called by hashes instead of names
- This method makes detection very hard even for manually analyzing the malware.
- The malware loads all exports of ntdll one by one and computes the CRC-32/BZIP2 hash of those apis then compares it with its decrypted hashes.
- If a match is found, then it retrieves the address and call the function, **hence bypassing all API hooks.**



- I wrote a little script that does the same, I provide the hash and it searches in a list of commonly used strings,apis,paths etc, computes their hashes and then compares with the provided hash to check weather a match has been found or not.
- Here in this case, the hash matched on **NtResumeThread** API call, so malware will exit the loop and continues to retrieve the address and then call the api.
- It manually searches for the address of desired API and calls it, this way the debugger is also not able to detect which API is being called.
- In the screenshot below, I have opened another instance of same dll in IDA with symbols and we can see the hex value that is being pushed onto eax register is the same.

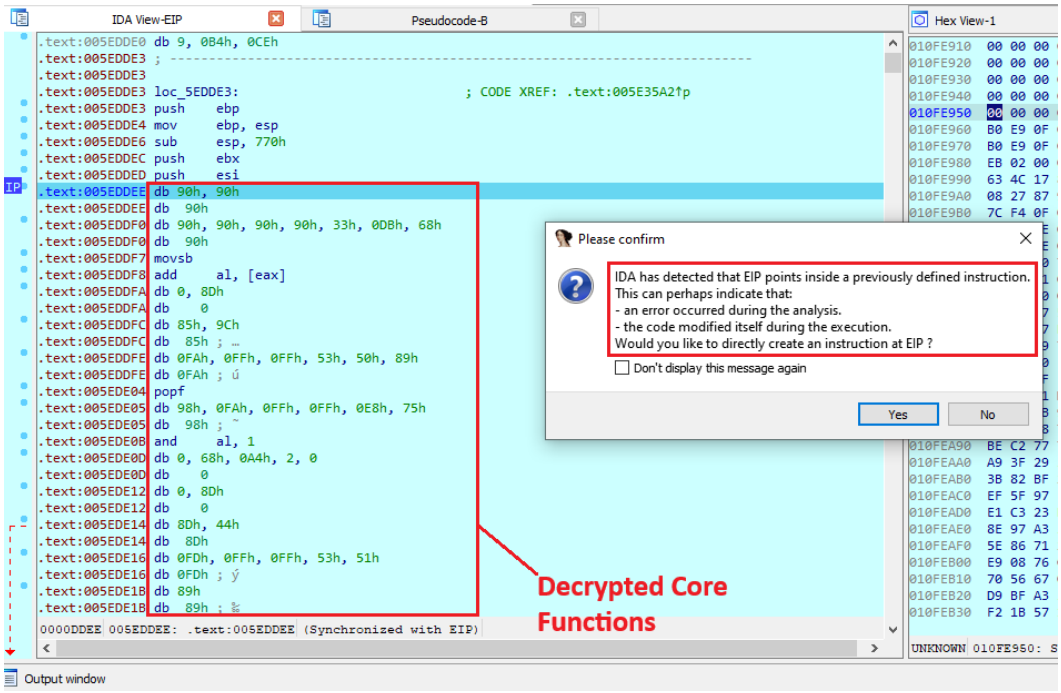


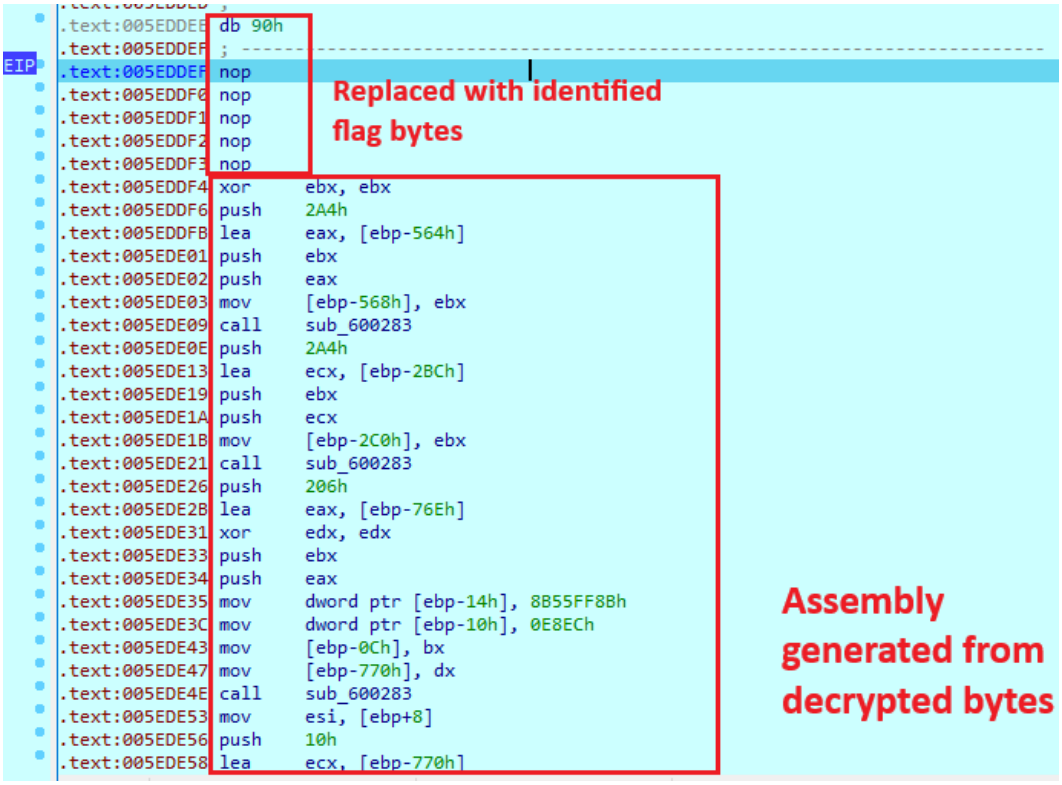
- I know the hashing function, so instead of stepping through this native assembly of hundreds of functions in a loop, I have just setup the breakpoint on that function by writing IDA python script and just continuing again and again to see the decrypted APIs
- The List of APIs that I found are listed below:

1	NtOpenDirectoryObject	
2	NtCreateMutant	
3	RtlSetEnvironmentVariable	
4	NtCreateSection	
5	NtMapViewOfSection	
6	NtOpenProcess	
7	RtlAllocHeap	
8	NtQueryInformationToken	
9	NtProtectVirtualMemory	
10	NtCreateFile	
11	NtDelayExecution	
12	NtReadVirtualMemory	
13	NtOpenThread	
14	NtReadFile	
15	NtUnmapViewOfSection	
16	NtResumeThread	
17	ExitProcess	
18	NtQuerySystemInformation	
19	NtOpenProcessToken	
20	NtAdjustPrivilegesToke	
21	NtReadVirtualMemory	
22	RtlQueryEnvironmentVariable	
23	RtlDosPathNameToNtPathName_U	
24	NtSuspendThread	
25	NtGetContextThread	
26	NtSetContextThread	

Decrypting Core Malicious Functions:

- The malware decrypts its core functions at run-time and then jumps to those functions continuing the execution flow.
- Xloader sets up a function by **push ebp** and **mov ebp, esp** and other starting instructions but below these all bytes are encrypted.
- In previous versions of formbook, the core malicious functions could be identified by the magic bytes of 48909090, 49909090 etc.
- However, in the latest xloader 4.3 these starting bytes are random.
- After the anti-vm checks and establishing the RC4 decryption key. These functions are decrypted at run-time and the execution flow jumped to the decrypted assembly.
- IDA resolves the decrypted bytes and recreates assembly instructions to continue.

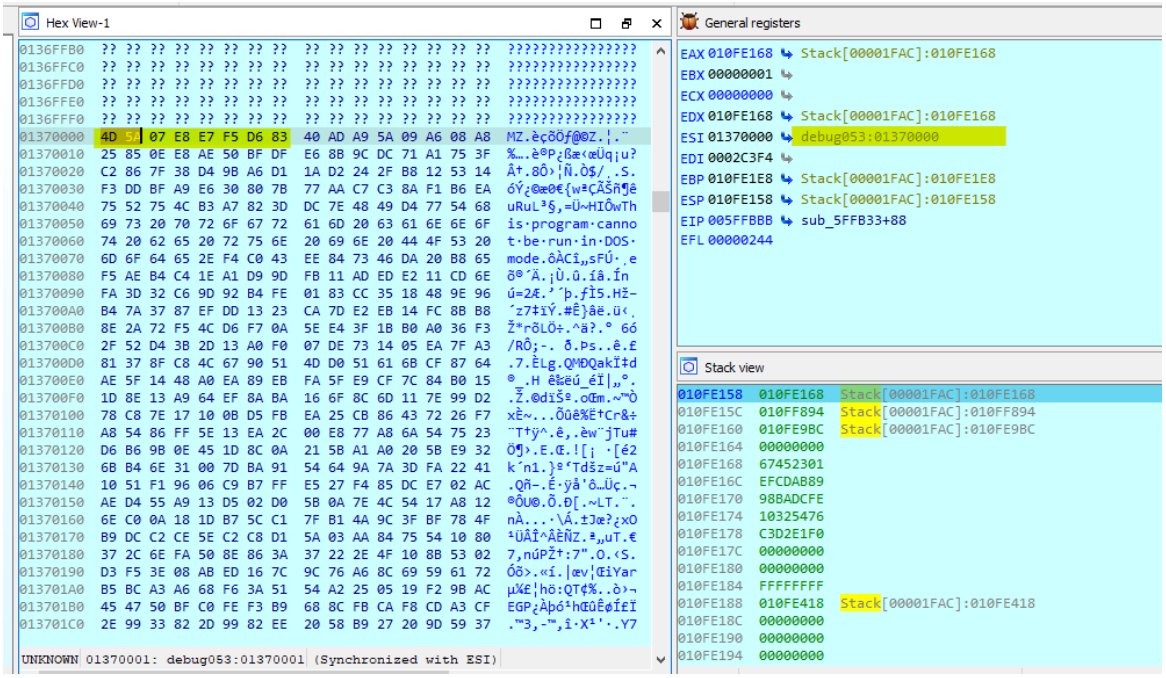


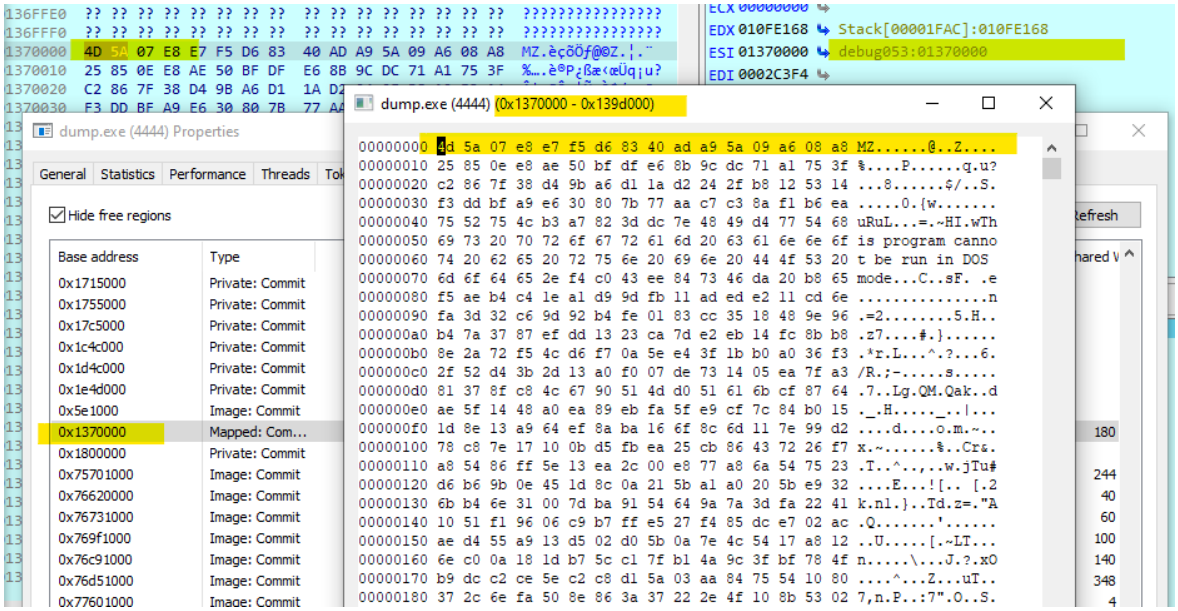


Understanding the detailed technical methodology of decrypting these encryption and obfuscation techniques. This following blog by **zscaler** is an excellent resource.
https://www.zscaler.com/blogs/security-research/technical-analysis-xloader-s-code-obfuscation-version-4-3

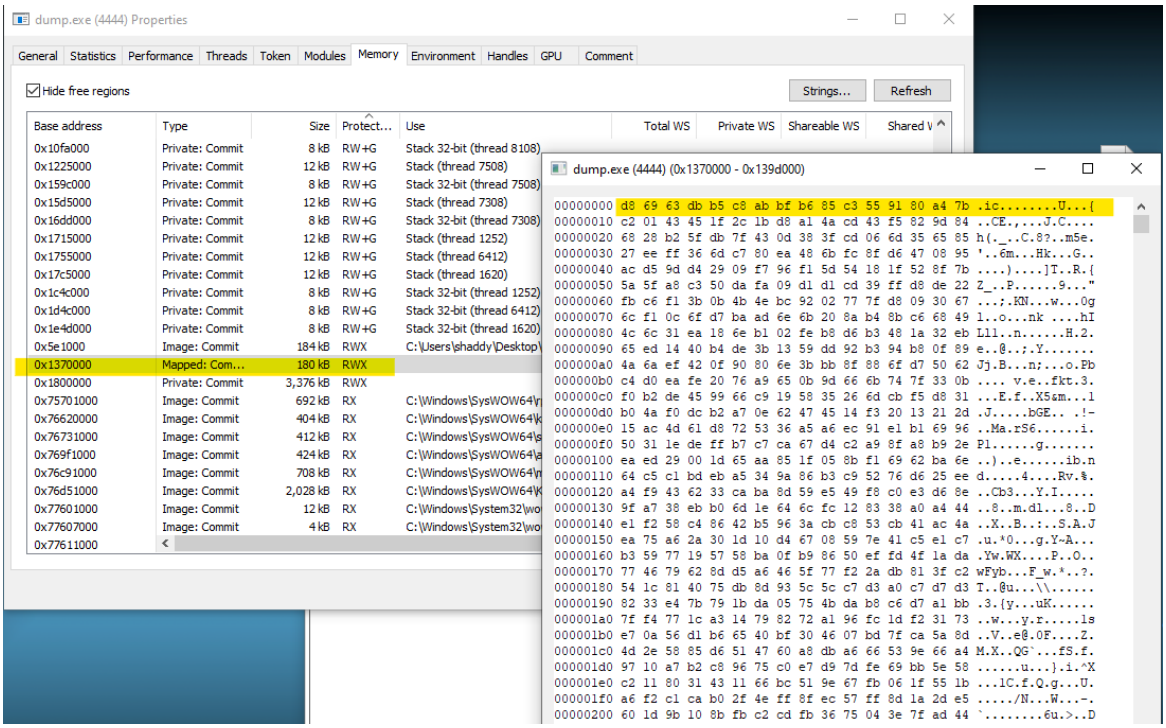
Partially Decrypted Shellcode:

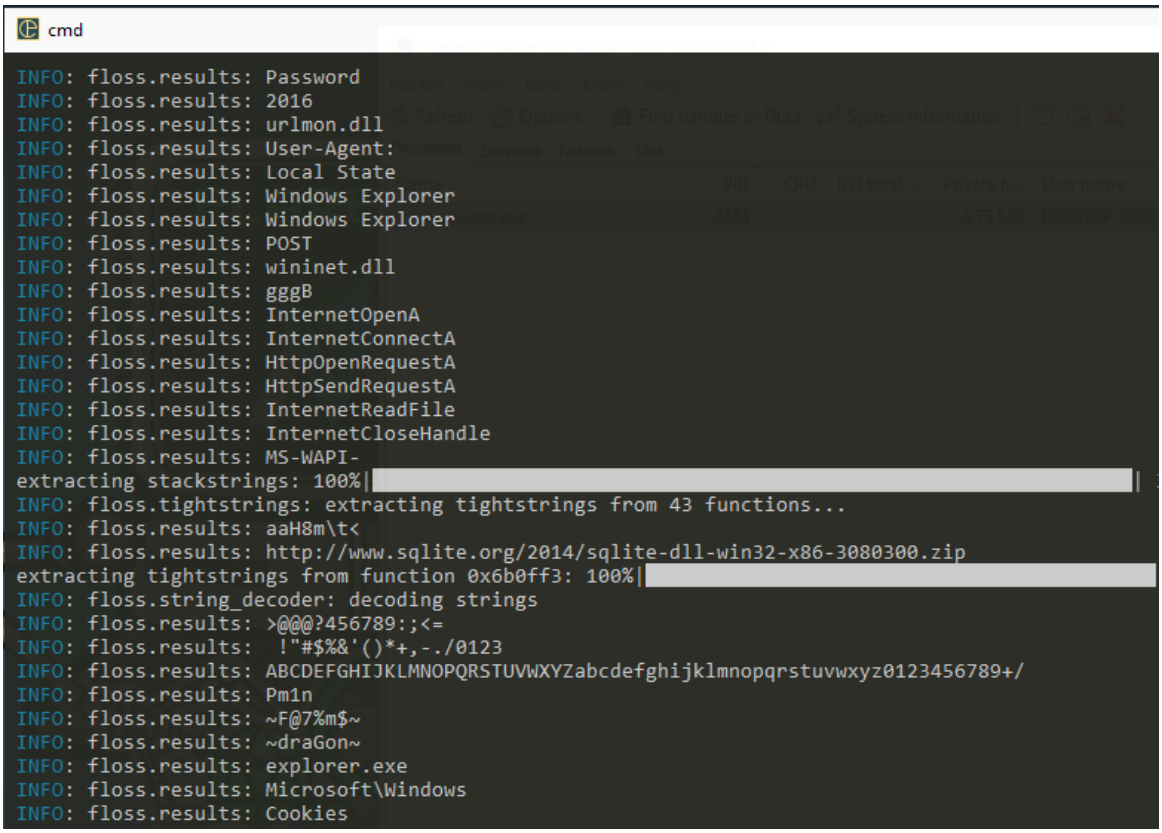
- Stepped over a few functions and it looks like it reads itself and most likely trying to inject itself in some other process
- The malware is now preparing for another binary to inject further. As can be seen in the screenshot of the dump that I found in the memory
- This memory dump is **RWX** memory region in itself as can be seen in the process hacker





- I stepped over a few functions while monitoring the memory region.
- The malware is decrypting the shellcode from the binary
- Only plain shellcode is left without MZ headers
- This is the 3rd stage xloader which is partially decrypted
- I dumped the binary from memory and run a FLOSS string search on it which provides some useful insights


dump.exe (4444) Properties
General
Statistics
Performance
Threads
Token
Modules
Memory
Environment
Handles
GPU
Comment
Hide free regions
Strings...
Refresh
Base address
Type
Size
Protect...
Use
Total WS
Private WS
Shareable WS
Shared V
0x10fa000 Private: Commit 8 kB RW+G Stack 32-bit (thread 8108)
0x1225000 Private: Commit 12 kB RW+G Stack (thread 7508)
0x159c000 Private: Commit 8 kB RW+G Stack 32-bit (thread 7508)
0x15d5000 Private: Commit 12 kB RW+G Stack (thread 7308)
0x16dd000 Private: Commit 8 kB RW+G Stack 32-bit (thread 7308)
0x1715000 Private: Commit 12 kB RW+G Stack (thread 1252)
0x1755000 Private: Commit 12 kB RW+G Stack (thread 6412)
0x17c5000 Private: Commit 12 kB RW+G Stack (thread 1620)
0x1c4c000 Private: Commit 8 kB RW+G Stack 32-bit (thread 1252)
0x1d4c000 Private: Commit 8 kB RW+G Stack 32-bit (thread 6412)
0x1e4d000 Private: Commit 8 kB RW+G Stack 32-bit (thread 1620)
0x5e1000 Image: Commit 184 kB RWX C:\Users\shaddy\Desktop\
0x1370000 Mapped: Com... 180 kB RWX
0x1800000 Private: Commit 3,376 kB RWX
0x75701000 Image: Commit 692 kB RX C:\Windows\SysWOW64\
0x76620000 Image: Commit 404 kB RX C:\Windows\SysWOW64\
0x76731000 Image: Commit 412 kB RX C:\Windows\SysWOW64\
0x769f1000 Image: Commit 424 kB RX C:\Windows\SysWOW64\
0x76c91000 Image: Commit 708 kB RX C:\Windows\SysWOW64\
0x76d51000 Image: Commit 2,028 kB RX C:\Windows\SysWOW64\
0x77601000 Image: Commit 12 kB RX C:\Windows\System32\
0x77607000 Image: Commit 4 kB RX C:\Windows\System32\
0x77611000
dump.exe (4444) (0x1370000 - 0x139d000)
00000000 d8 69 63 db b5 c8 ab bf b6 85 c3 55 91 80 a4 7b .ic.....U...{
00000010 c2 01 43 45 1f 2c 1b d8 a1 4a cd 43 f5 82 9d 84 ..CE.,...J.C....
00000020 68 28 b2 5f db 7f 43 0d 38 3f cd 06 6d 35 65 85 h(._..C.8?..m5e.
00000030 27 ee ff 36 6d c7 80 ea 48 6b fc 8f d6 47 08 95 '..6m...Hk...G..
00000040 ac d5 9d d4 29 09 f7 96 f1 5d 54 18 1f 52 8f 7b ....)....]T..R.{
00000050 5a 5f a8 c3 50 da fa 09 d1 d1 cd 39 ff d8 de 22 Z_..P......9..."
00000060 fb c6 f1 3b 0b 4b 4e bc 92 02 77 7f d8 09 30 67 ...;.KN...w...0g
00000070 6c f1 0c 6f d7 ba ad 6e 6b 20 8a b4 8b c6 68 49 l..o...nk ....hI
00000080 4c 6c 31 ea 18 6e b1 02 fe b8 d6 b3 48 1a 32 eb Ll1..n......H.2.
00000090 65 ed 14 40 b4 de 3b 13 59 dd 92 b3 94 b8 0f 89 e..@..;.Y.......
000000a0 4a 6a ef 42 0f 90 80 6e 3b bb 8f 88 6f d7 50 62 Jj.B...n;...o.Pb
000000b0 c4 d0 ea fe 20 76 a9 65 0b 9d 66 6b 74 7f 33 0b .... v.e..fkt.3.
000000c0 f0 b2 de 45 99 66 c9 19 58 35 26 6d cb f5 d8 31 ...E.f..X5&m...1
000000d0 b0 4a f0 dc b2 a7 0e 62 47 45 14 f3 20 13 21 2d .J.....bGE.. .!-
000000e0 15 ac 4d 61 d8 72 53 36 a5 a6 ec 91 e1 b1 69 96 ..Ma.rS6......i.
000000f0 50 31 1e de ff b7 c7 ca 67 d4 c2 a9 8f a8 b9 2e P1......g.......
00000100 ea ed 29 00 1d 65 aa 85 1f 05 8b f1 69 62 ba 6e ..)..e......ib.n
00000110 64 c5 c1 bd eb a5 34 9a 86 b3 c9 52 76 d6 25 ee d.....4....Rv.%.
00000120 a4 f9 43 62 33 ca ba 8d 59 e5 49 f8 c0 e3 d6 8e ..Cb3...Y.I.....
00000130 9f a7 38 eb b0 6d 1e 64 6c fc 12 83 38 a0 a4 44 ..8..m.dl...8..D
00000140 e1 f2 58 c4 86 42 b5 96 3a cb c8 53 cb 41 ac 4a ..X..B..:..S.A.J
00000150 ea 75 a6 2a 30 1d 10 d4 67 08 59 7e 41 c5 e1 c7 .u.*0...g.Y~A...
00000160 b3 59 77 19 57 58 ba 0f b9 86 50 ef fd 4f 1a da .Yw.WX....P..O..
00000170 77 46 79 62 8d d5 a6 46 5f 77 f2 2a db 81 3f c2 wFyb...F_w.*..?.
00000180 54 1c 81 40 75 db 8d 93 5c 5c c7 d3 a0 c7 d7 d3 T..@u...\\......
00000190 82 33 e4 7b 79 1b da 05 75 4b da b8 c6 d7 a1 bb .3.{y...uK......
000001a0 7f f4 77 1c a3 14 79 82 72 a1 96 fc 1d f2 31 73 ..w...y.r.....1s
000001b0 e7 0a 56 d1 b6 65 40 bf 30 46 07 bd 7f ca 5a 8d ..V..e@.0F....Z.
000001c0 4d 2e 58 85 d6 51 47 60 a8 db a6 66 53 9e 66 a4 M.X..QG`...fS.f.
000001d0 97 10 a7 b2 c8 96 75 c0 e7 d9 7d fe 69 bb 5e 58 ......u...}.i.^X
000001e0 c2 11 80 31 43 11 66 bc 51 9e 67 fb 06 1f 55 1b ...1C.f.Q.g...U.
000001f0 a6 f2 c1 ca b0 2f 4e ff 8f ec 57 ff 8d 1a 2d e5 ...../N...W...-.
00000200 60 1d 9b 10 8b fb c2 cd fb 36 75 04 3e 7f ad 44 `........6u.>..D


cmd
INFO: floss.results: Password
INFO: floss.results: 2016
INFO: floss.results: urlmon.dll
INFO: floss.results: User-Agent:
INFO: floss.results: Local State
INFO: floss.results: Windows Explorer
INFO: floss.results: Windows Explorer
INFO: floss.results: POST
INFO: floss.results: wininet.dll
INFO: floss.results: gggB
INFO: floss.results: InternetOpenA
INFO: floss.results: InternetConnectA
INFO: floss.results: HttpOpenRequestA
INFO: floss.results: HttpSendRequestA
INFO: floss.results: InternetReadFile
INFO: floss.results: InternetCloseHandle
INFO: floss.results: MS-WAPI-
extracting stackstrings: 100%
INFO: floss.tightstrings: extracting tightstrings from 43 functions...
INFO: floss.results: aaH8m\t<
INFO: floss.results: http://www.sqlite.org/2014/sqlite-dll-win32-x86-3080300.zip
extracting tightstrings from function 0x6b0ff3: 100%
INFO: floss.string_decoder: decoding strings
INFO: floss.results: >@@@?456789:;<=
INFO: floss.results: !"#$%&'()*+,-./0123
INFO: floss.results: ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/
INFO: floss.results: Pm1n
INFO: floss.results: ~F@7%m$~
INFO: floss.results: ~draGon~
INFO: floss.results: explorer.exe
INFO: floss.results: Microsoft\Windows
INFO: floss.results: Cookies

Extracted Stings from Xloader 4.3 Stage3 shellcode		
1	Extracted	2016 2012 2008 open \explorer.exe windir .exe \rundll32.exe \System32 \SysWOW64 windir .exe .dll \Current Session \INetCookies \Microsoft\Windows .sqlite \Cookies \explorer.exe windir Clipboard Unknown [System] USERNAME .dll log.ini sog.ini ProgramFiles SysWOW64\ SELECT name, value FROM autofill name value: datetime SELECT host_key, path, is_secure, expires_utc, name, value, encrypted_value FROM cookies FALSE TRUE Cookies Autofill Chrome PATH Firefox\ .exe Firefox Program Files

		\Firefox CurrentVersion Main Install Directory guid httpRealm hostname profiles.ini PATH Thunderbird\ Firefox\ null Account Password POP3Account POP3Password Account.stg Fox Recovery \Program Files Opera Chrome \3r9Pk-75_ Recovery \Opera Software\Opera Stable \Opera Software\Opera Stable !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~ encrypted_key Local State Pass User Internet Explorer\IntelliForms\Storage2 Pass Name __Vault Iexplor Outlook Recovery Password 2016 urlmon.dll User-Agent: Local State Windows Explorer Windows Explorer POST wininet.dll gggB InternetOpenA

		InternetConnectA HttpOpenRequestA HttpSendRequestA InternetReadFile InternetCloseHandle MS-WAPI-
2	Floss decoded & tight strings	aaH8m\t< hxxp://www.sqlite.org/2014/sqlite-dll-win32-x86-3080300.zip >@@@?456789:;<= !"#$%&'()*+,-./0123 ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/ Pm1n ~F@7%m$~ ~draGon~ explorer.exe Microsoft\Windows Cookies encrypted_key Local State TRUE 381F ppwEw czX5 .dll 7CbI sqlite3 sqlite3.dll

Process Enumeration:

- XLoader uses **NtQuerySystemInformation** to get information of all running processes in the system and then enumerates one-by-one checking and matching hashes with its own hash values stored in conf obj.

```
77 00 69 00 6E 00 6C 00  6F 00 67 00 6F 00 6E 00  w.i.n.l.o.g.o.n.
2E 00 65 00 78 00 65 00  00 00 00 00 00 00 00 00  ..e.x.e.........
```

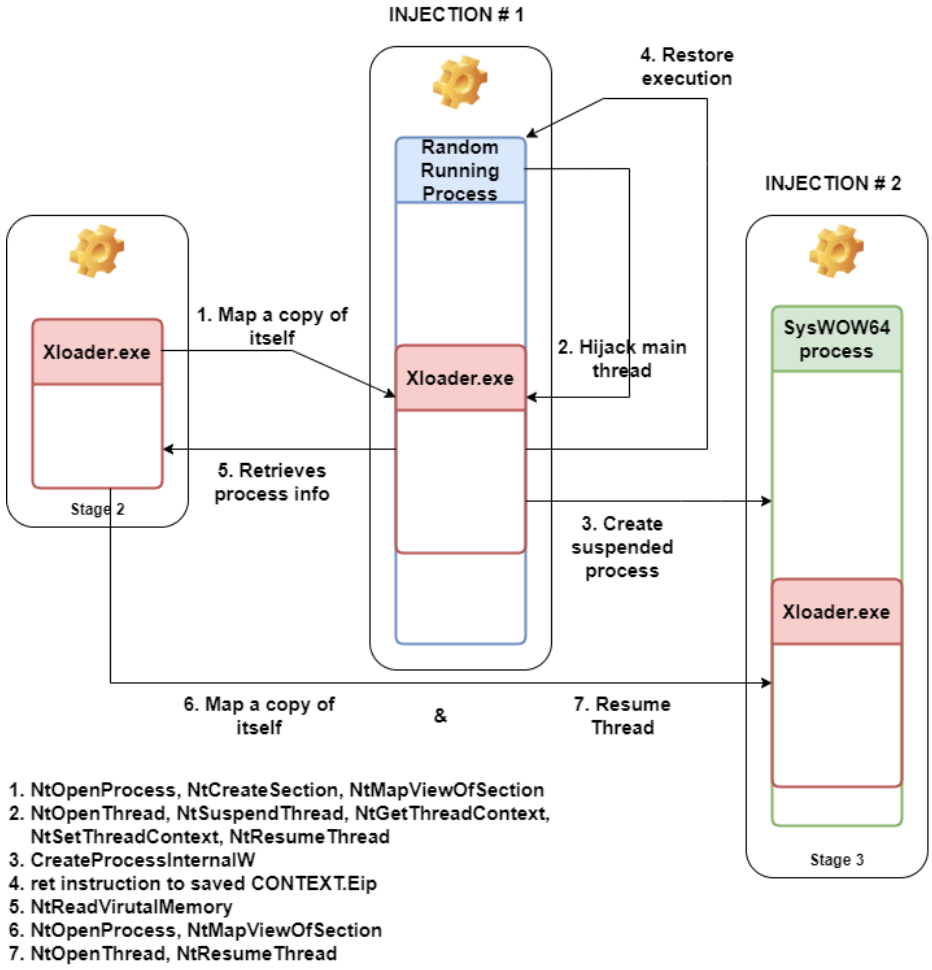
```
73 00 65 00 72 00 76 00  69 00 63 00 65 00 73 00  s.e.r.v.i.c.e.s.
2E 00 65 00 78 00 65 00  00 00 00 00 00 00 00 00  ..e.x.e.........
```

Process Injection:

Xloader Injection Overview:

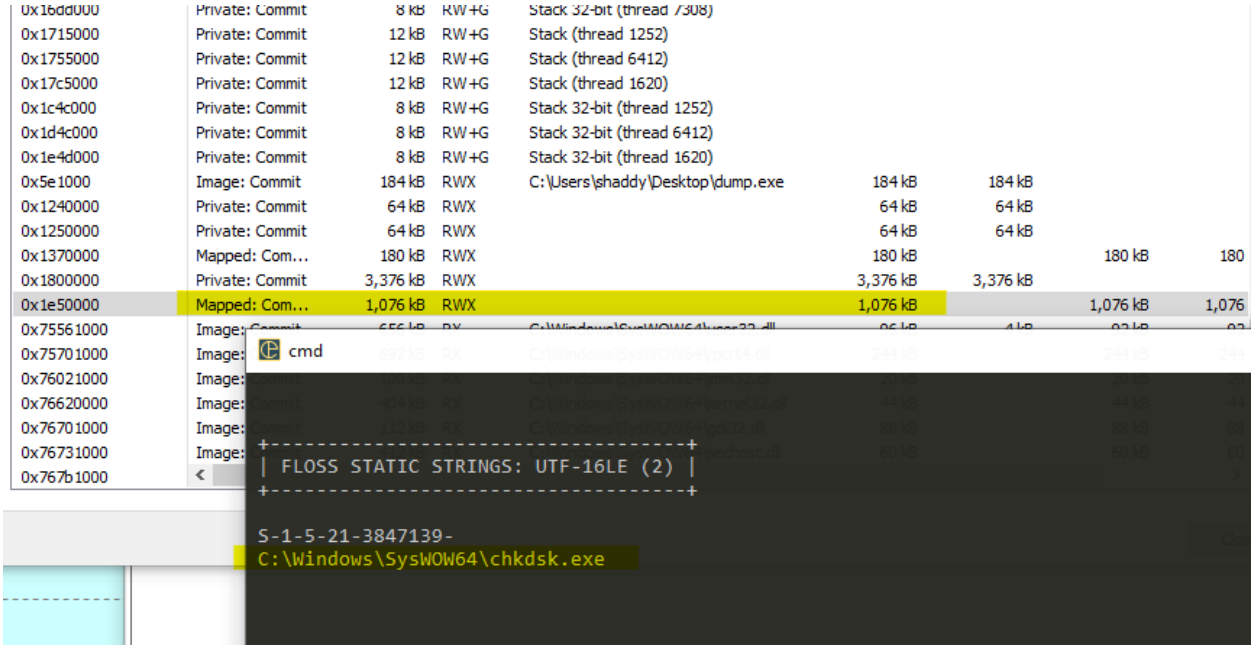
Xloader stage2 performs two process injections:

- Injection#1: in a random running process to start the win32 victim process in suspended state
- Injection#2: migrate itself into win32 suspended process and resume

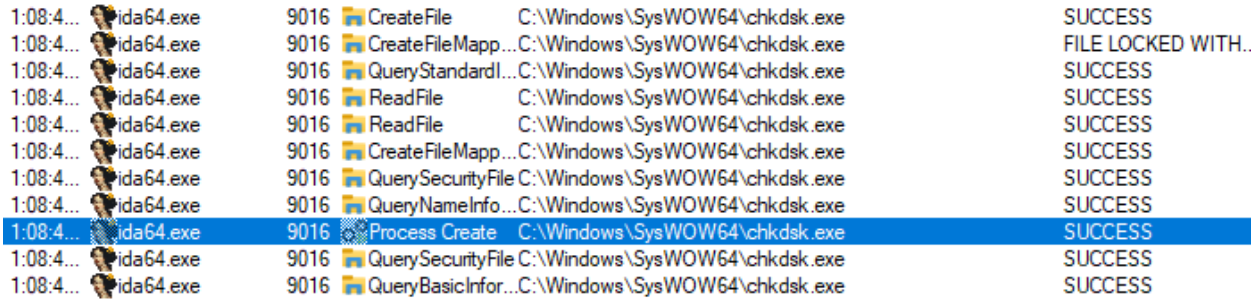


Injection # 1

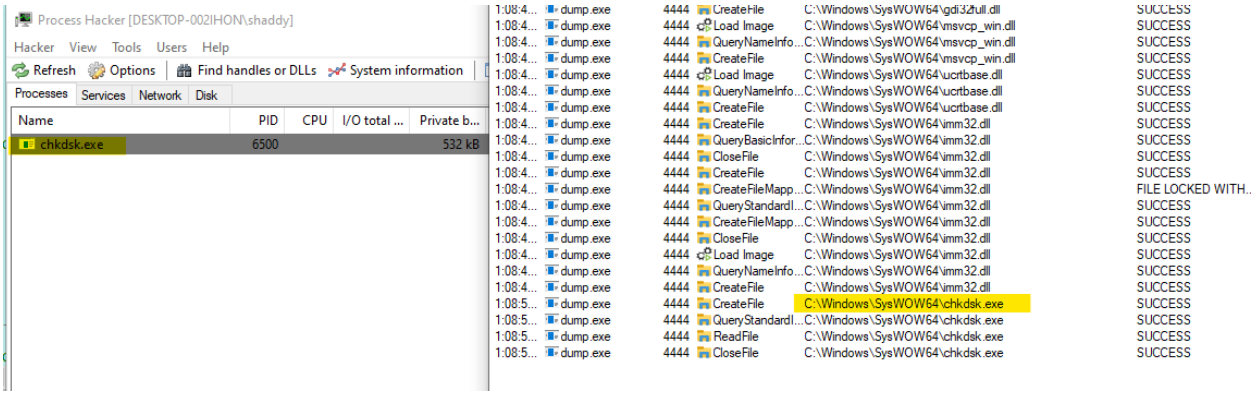
- Another memory has been reserved in the malware with RWX memory region.
- I have dumped this new region and extracted the strings
- It has a single static string which contains the name of the target process



- It means that this shellcode is used for starting the process **chkdsk.exe** which is randomized on every execution.
- Xloader selects these binaries from SysWOW64 directory, which are 32-bit processes
- It injects this shellcode in one of the above enumerated running processes, which in my case is a 64-bit IDA that I had opened along with my debugger.



- This is also one of the anti-analysis techniques used by xloader. It doesn't directly open the process itself but injects shellcode in some random process which in turn opens the SysWOW64 randomized binary in a suspended state and then retrieves its process information and continue with the execution.

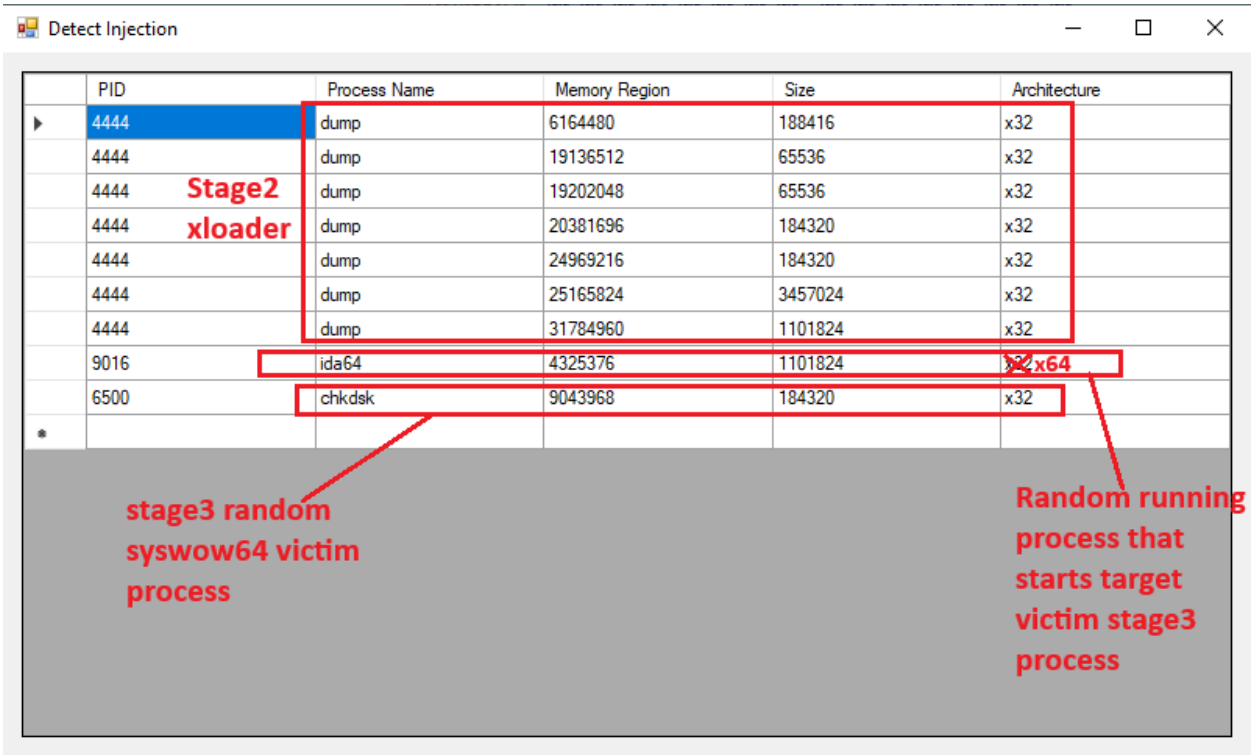


- In Ida64, the shellcode is injected which starts the process and return the process information back to stage2 malware of xloader.
- The RWX memory region could be seen in IDA64.
- This is just a **dead code** after opening the target process in suspended state.

0xfbbcdfc000	Private: Commit	12 kB	RW+G	Stack (thread 6012)			
0x420000	Mapped: Com...	1,076 kB	RWX		1,076 kB	1,076 kB	1,076
0x2d464710000	Private: Commit	64 kB	RWX		8 kB	8 kB	

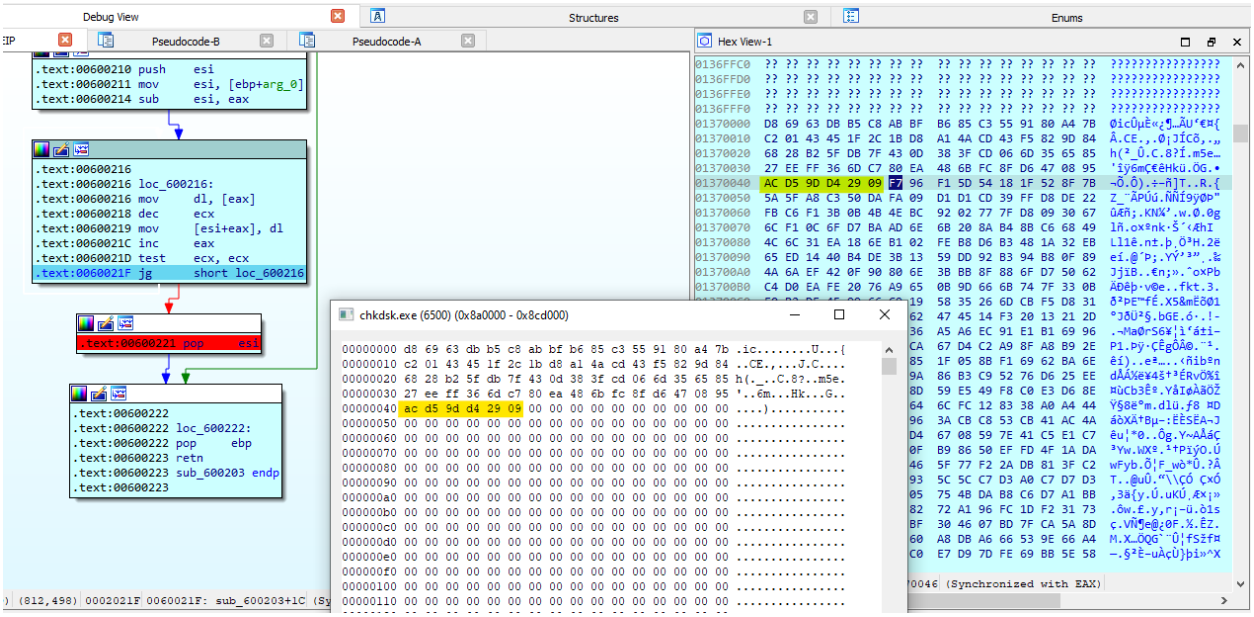
Injection # 2:

- The second injection is performed in the chkdsk.exe (randomized SysWOW64 binary)
- There are two buffers injected in the chkdsk.exe.
- 1 buffer of 180KB and other of 40KB
- Since this malware is performing so many injections, it is very difficult to keep track of everything so we got an idea of creating a tool for detecting process injections.
- I would like to give special thanks to **Osama Ellahi**, for creating this tool in short period of time which is very useful in detecting injections of such malware.

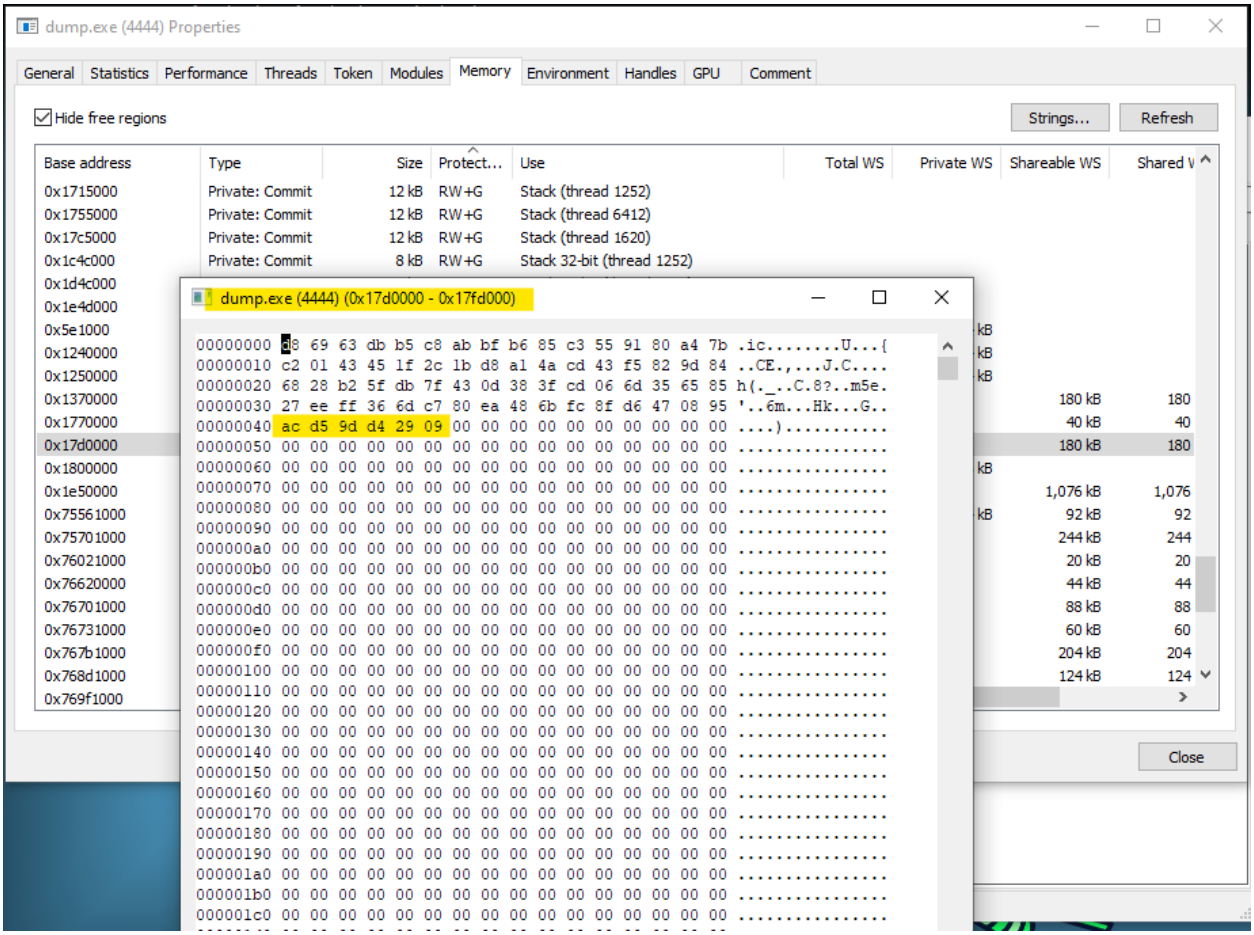


Tool link: https://github.com/Jhangju/injectionview

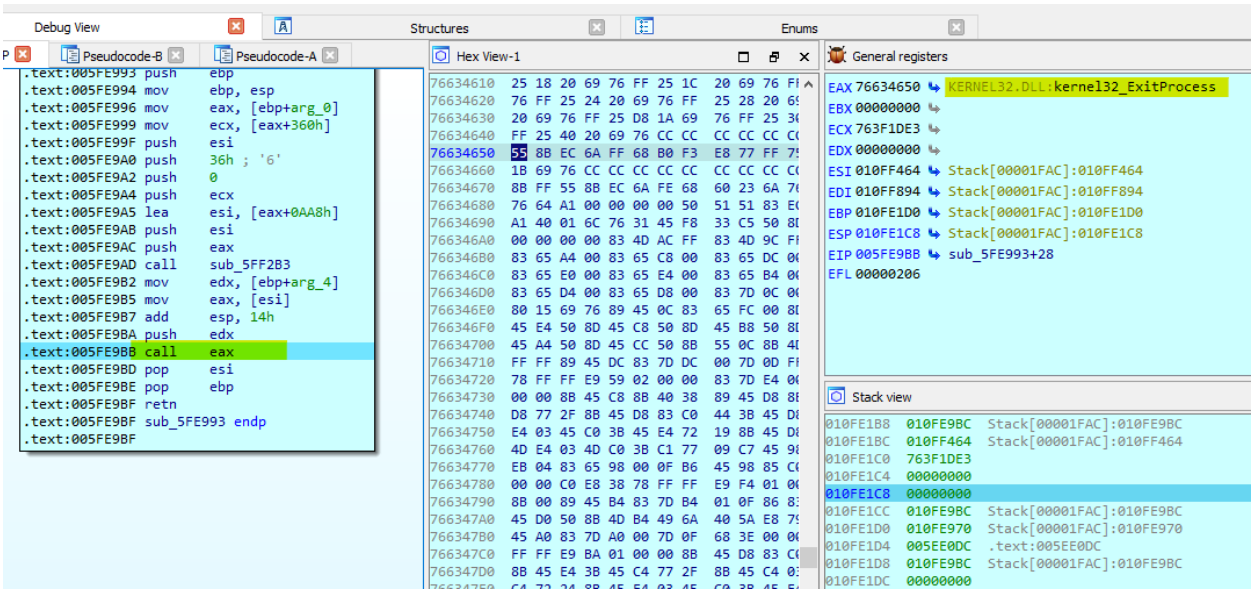
- The smaller buffer contains the original chkdsk.exe bytes.
- I also found the function that writes shellcode in the **180KB** empty buffer.
- This is also a shared memory region between the formbook payload and victim process of chkdsk.exe
- Because the buffer is simultaneously being written in both processes.



- Here in xloader payload, the memory region is also being written simultaneously
- This is the same partially decrypted shellcode that I have displayed above, with most of the decrypted strings.
- From here onwards, the stage3 of formbook will be executed.



- Finally, after resuming the suspended process in chkdsk.exe
- It exits using ExitProcess API

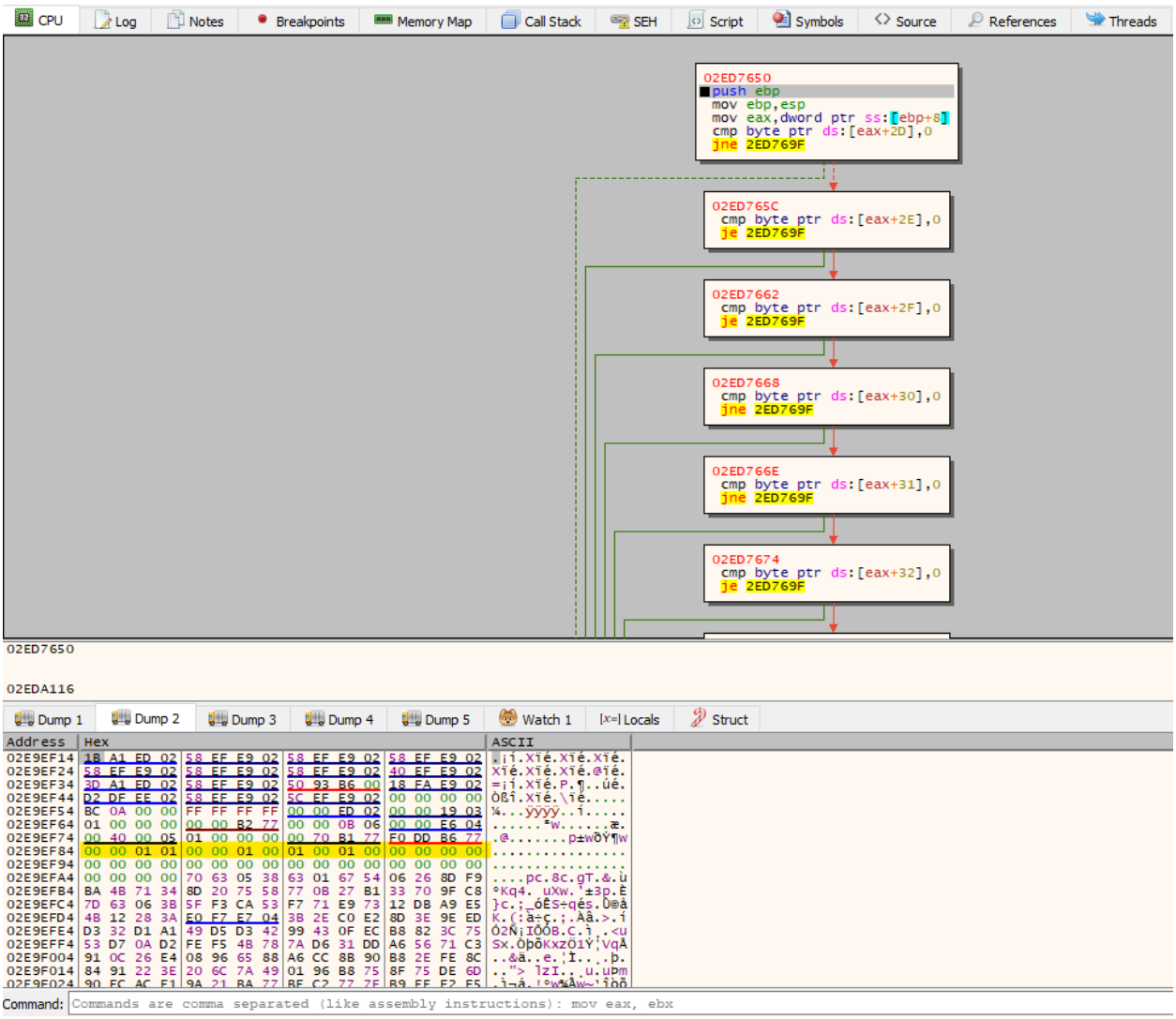


Stage 3: Partially Decrypted Xloader 4.3

Before resuming the thread on injected process. I have attached x32dbg to the victim process to continue debugging further. In the EAX register, the address of xloader injected code is already set by stage2 malware. So, I just jumped to address in disassembly and added breakpoint on it. Then from the stage2 malware I allowed the malware to continue hence resuming the thread on stage3. Stage2 malware has exited and we have debugger attached to the entry point of stage3 malware which I will continue from here. This whole execution flow is very similar to stage2 malware. So, I will move forward with only key details in this section:

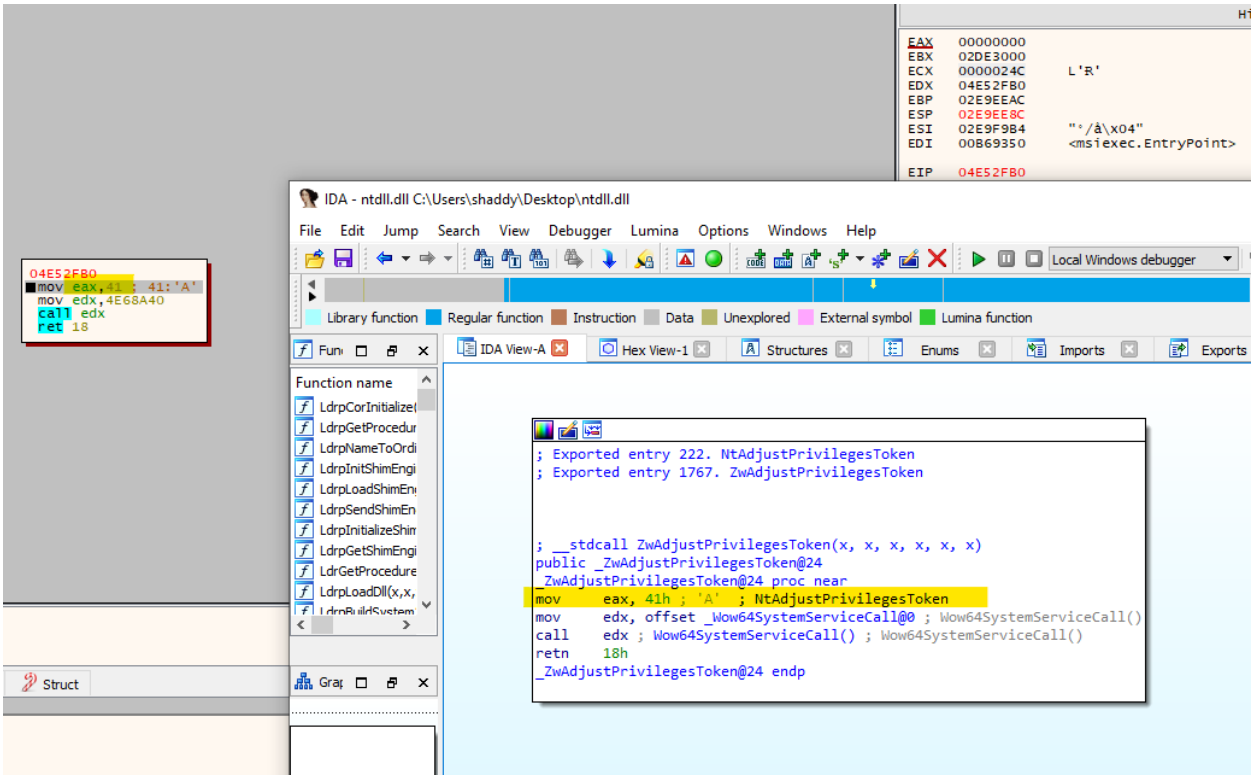
Defeating Anti-Analysis:

- Xloader has decrypted some of its functions and now migrated to the process **msiexec.exe** (which was **chkdsk.exe** in previous examples)
- Before resuming the thread, I've attached debugger to the injected process and continued my analysis from there.
- This is the same cycle being repeated first.
- I have to defeat anti-analysis techniques again
- Similar to stage2 I have bypassed anti-analysis techniques again and correct sequence of bytes have been generated as highlighted below



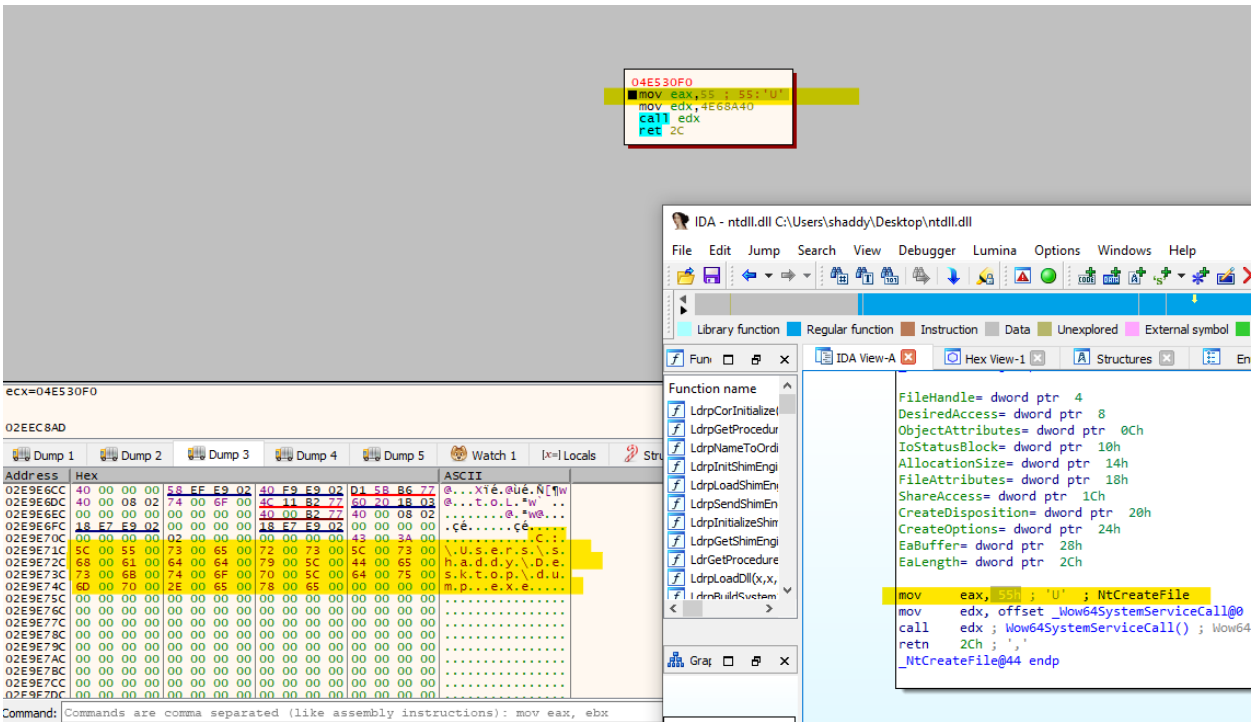
Decryption/Deobfuscation:

- This injected stage3 payload performs the same initial steps.
- It performs anti-vm techniques and checks
- Decrypt further library names and load using LdrLoadDll
- Decrypt API names and match hashes. Finally load those APIs from the injected fresh copy of **ntdll**
- A few of the APIs that it uses for Process Injection are resolved:
    - ❖ LookupPrivilegeValue
    - ❖ SeDebugPrivilege
    - ❖ NtAdjustPrivilegeToken



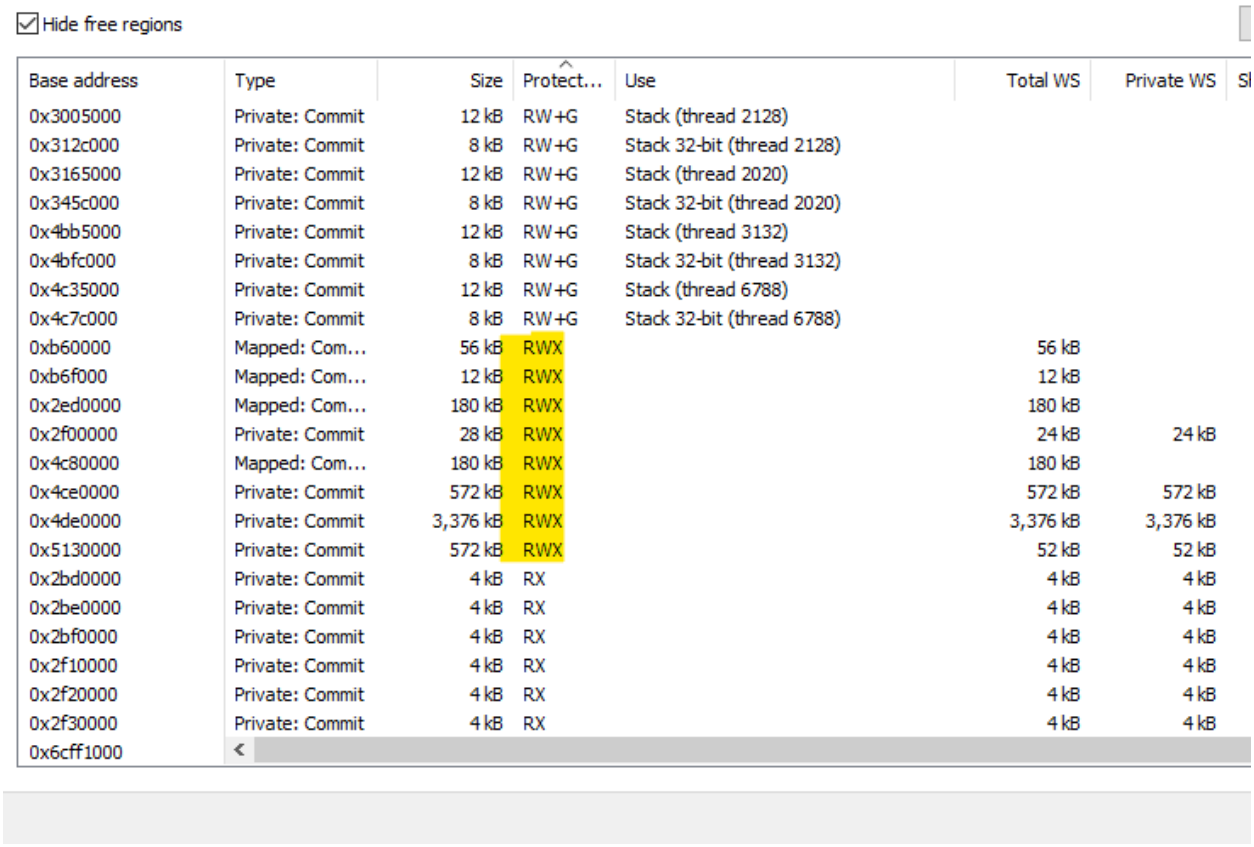
Indicator Removal:

- It will delete the stage2 malware with following sequence of APIs
  - ❖ NtCreateFile
  - ❖ NtQueryInformationFile
  - ❖ NtReadFile
  - ❖ NtClose
  - ❖ ZwDeleteFile



Process Injection:

- The next series of APIs being used are:
  - ❖ NtCreateSection
  - ❖ NtMapViewOfSection
  - ❖ NtAllocateVirtualMemory
  - ❖ NtOpenProcessToken
  - ❖ NtQueryInformationToken
  - ❖ ConvertSidToStringW
  - ❖ NtAllocateVirtualMemory
- It is preparing another shellcode to inject further in some process. There are a few more RWX sections created in the memory of infected process

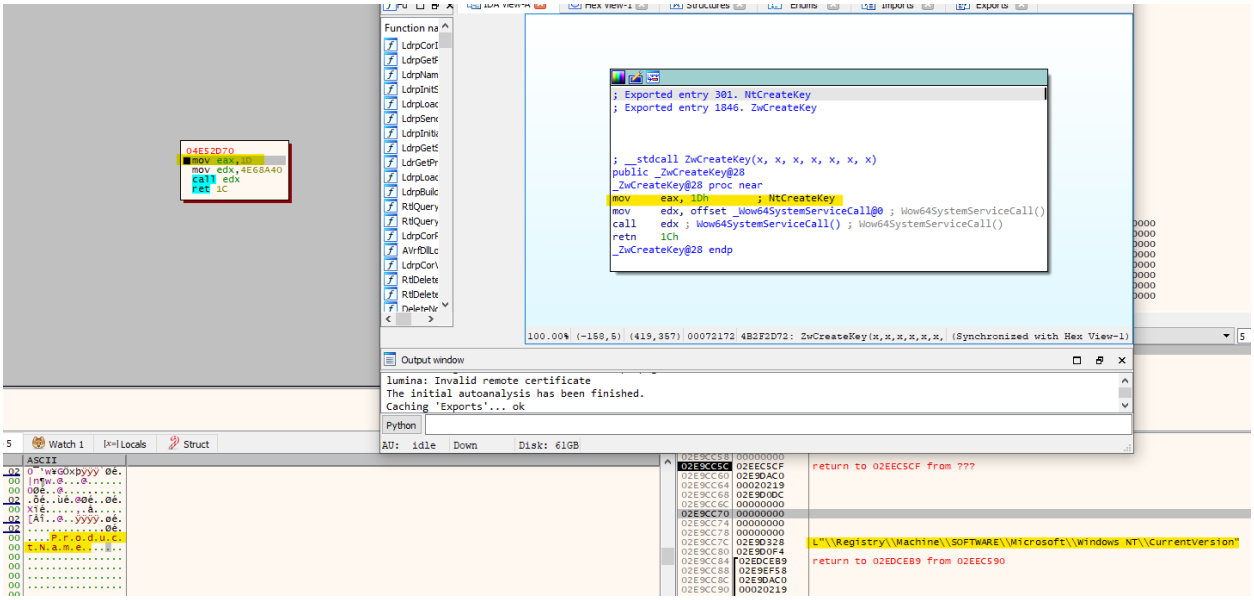


☑ Hide free regions

Base address	Type	Size	Protect...	Use	Total WS	Private WS	Sl
0x3005000	Private: Commit	12 kB	RW+G	Stack (thread 2128)			
0x312c000	Private: Commit	8 kB	RW+G	Stack 32-bit (thread 2128)			
0x3165000	Private: Commit	12 kB	RW+G	Stack (thread 2020)			
0x345c000	Private: Commit	8 kB	RW+G	Stack 32-bit (thread 2020)			
0x4bb5000	Private: Commit	12 kB	RW+G	Stack (thread 3132)			
0x4bfc000	Private: Commit	8 kB	RW+G	Stack 32-bit (thread 3132)			
0x4c35000	Private: Commit	12 kB	RW+G	Stack (thread 6788)			
0x4c7c000	Private: Commit	8 kB	RW+G	Stack 32-bit (thread 6788)			
0xb60000	Mapped: Com...	56 kB	RWX		56 kB		
0xb6f000	Mapped: Com...	12 kB	RWX		12 kB		
0x2ed0000	Mapped: Com...	180 kB	RWX		180 kB		
0x2f00000	Private: Commit	28 kB	RWX		24 kB	24 kB	
0x4c80000	Mapped: Com...	180 kB	RWX		180 kB		
0x4ce0000	Private: Commit	572 kB	RWX		572 kB	572 kB	
0x4de0000	Private: Commit	3,376 kB	RWX		3,376 kB	3,376 kB	
0x5130000	Private: Commit	572 kB	RWX		52 kB	52 kB	
0x2bd0000	Private: Commit	4 kB	RX		4 kB	4 kB	
0x2be0000	Private: Commit	4 kB	RX		4 kB	4 kB	
0x2bf0000	Private: Commit	4 kB	RX		4 kB	4 kB	
0x2f10000	Private: Commit	4 kB	RX		4 kB	4 kB	
0x2f20000	Private: Commit	4 kB	RX		4 kB	4 kB	
0x2f30000	Private: Commit	4 kB	RX		4 kB	4 kB	
0x6cff1000							

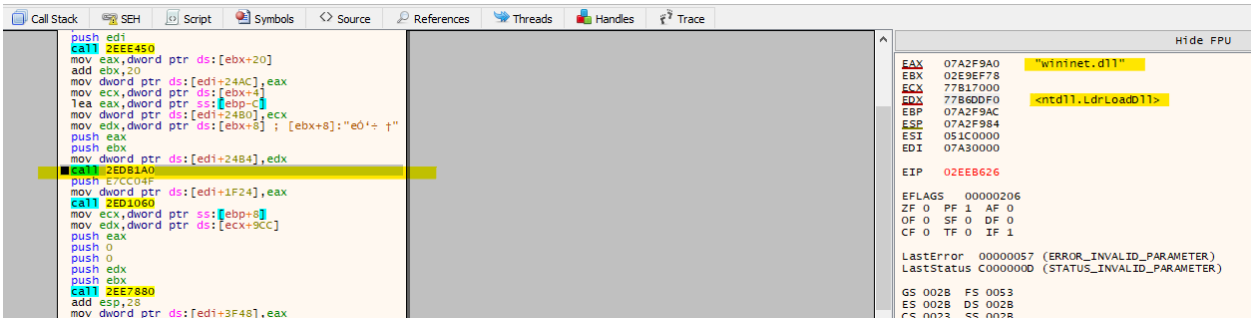
System Information Discovery:

- It retrieves the system information from the Registry like the **"Product Name", "CurrentBuild"** of OS etc
  - ❖ NtCreateKey
  - ❖ NtQueryValueKey



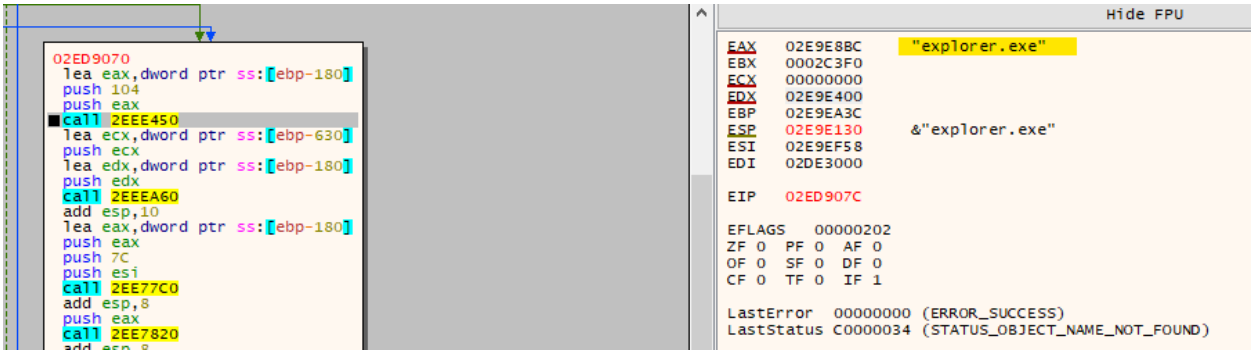
Dynamic Library/API resolution:

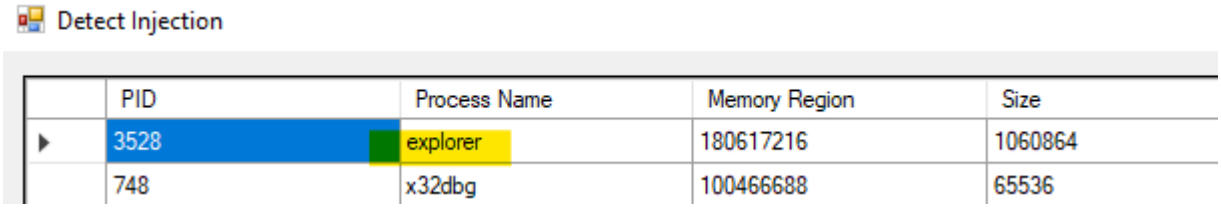
- Loading libraries **wininet.dll** using **LdrLoadDll**

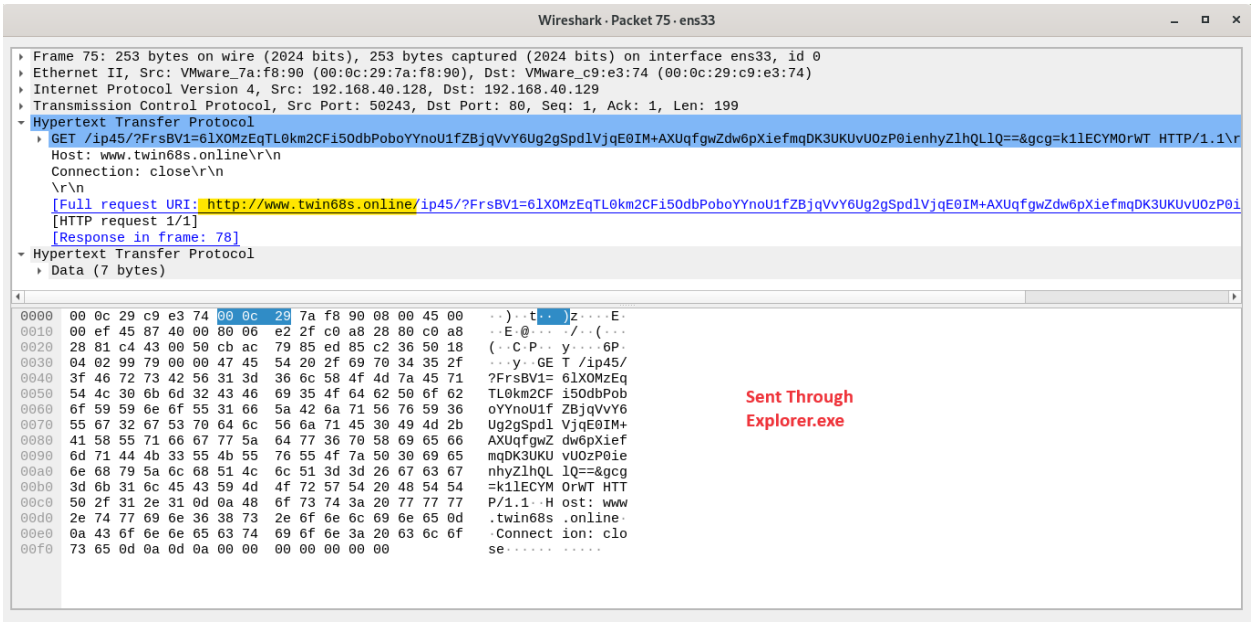


Process Enumeration & Injection:

- Looks like the next injection will be in **"explorer.exe"**.
- It enumerates all the process by looping through the list of processes returned by **"NtQuerySystemInformation"**
- NtCreateMutant
- NtCreateSection
- NtMapViewOfSection
- NtDelayExecution
- NtAllocateVirtualMemory

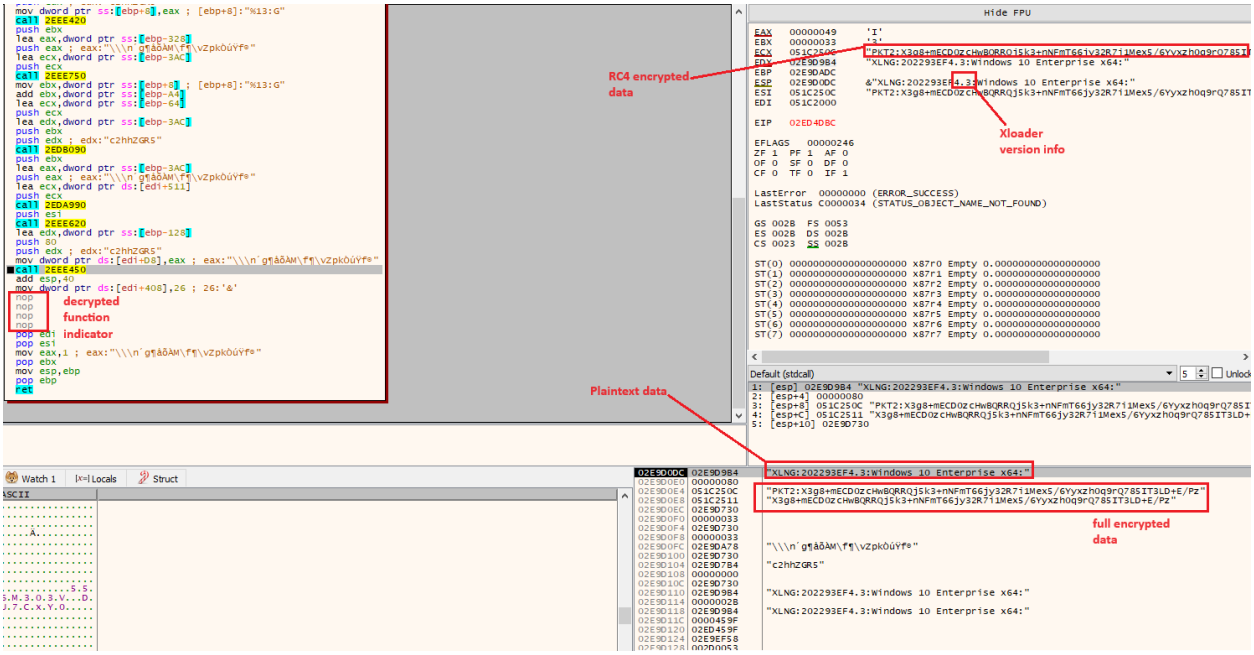






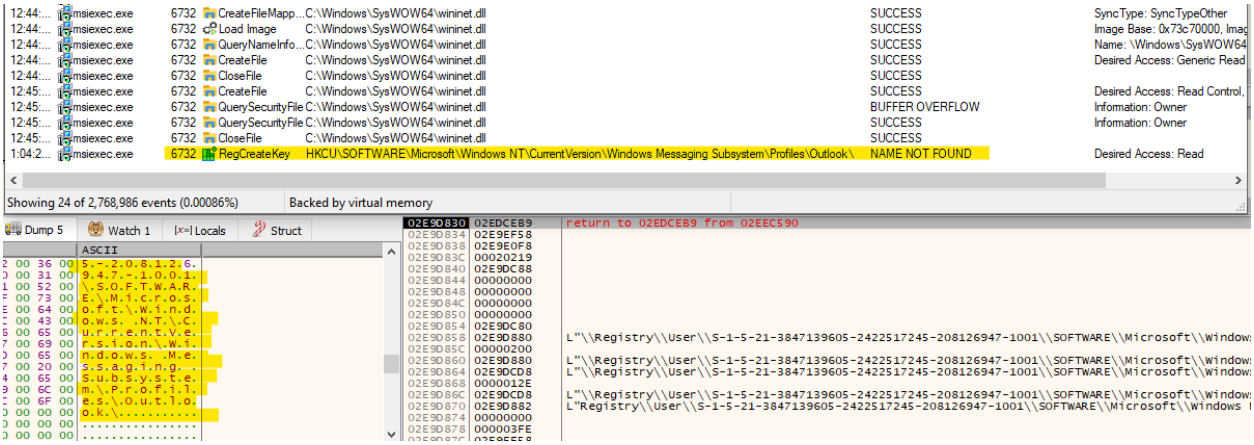
Botnet registration:

- The data it collects and sends in the first request is provided below:
- The Magic word: XLNG
- Bot ID: 202293EF
- Xloader Version: 4.3
- OS: Windows 10 Enterprise x64
- Username: base64_encoded

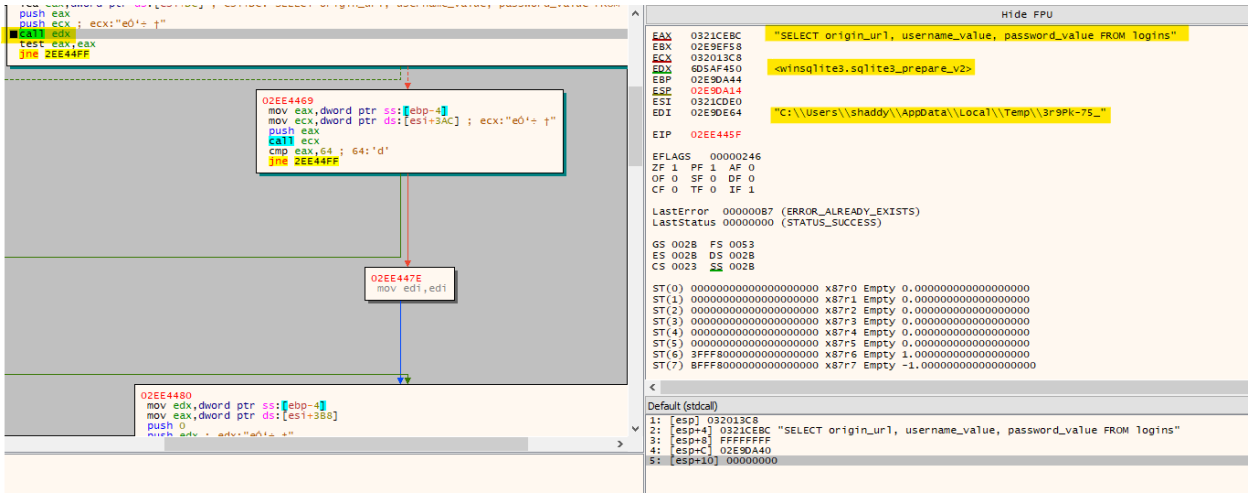


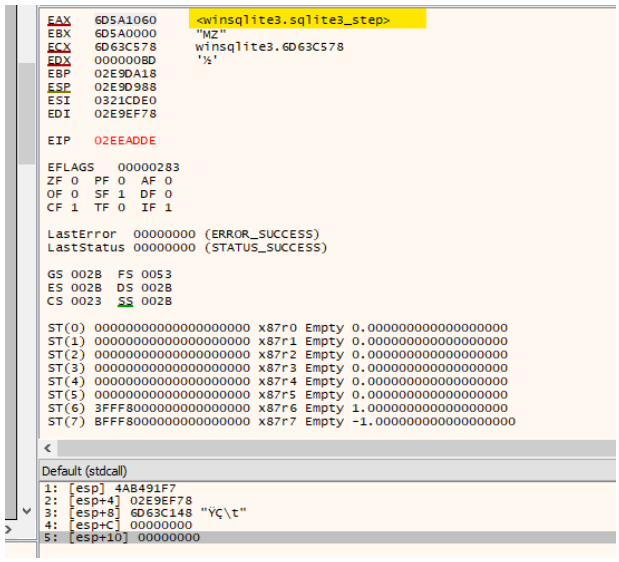
Stealer:

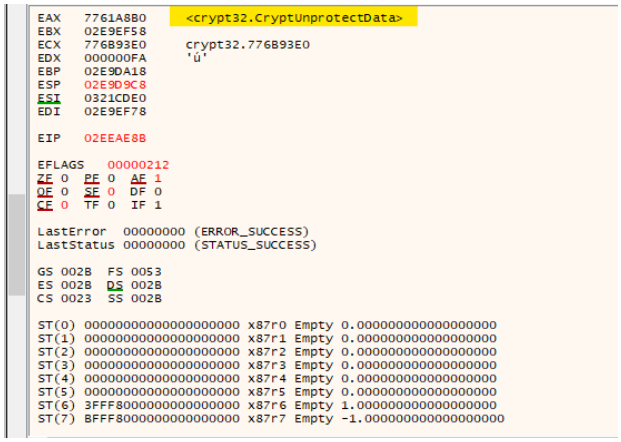
- Xloader is an infostealer and form grabber.
- After registering the device, it looks for all the things it could steal from the victim
- There are a large number of email clients, browsers, ftp clients, messaging apps that it tries to look for in different paths to fetch and steal the data



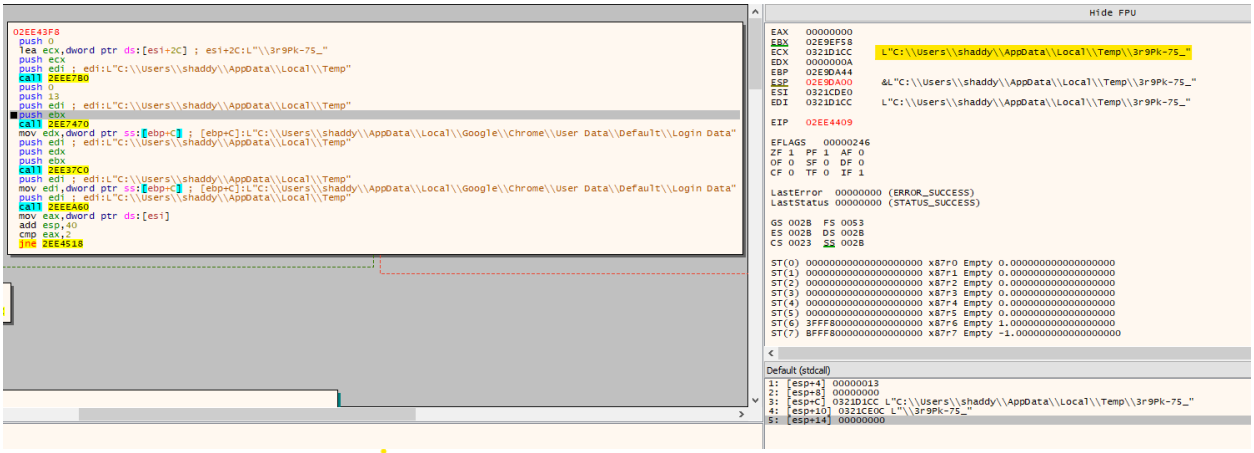
- If it finds anything, it then tries to steal that data
- Like in case of chrome, it founds login data and it will fetch the data using sqlite3 queries
- It uses winsqlite3.dll to extract passwords
- The query is **"SELECT origin_url, username_value, password_value FROM logins"**
- It decrypts that data using **crypt32.CryptUnprotectedData** from the key found in local state

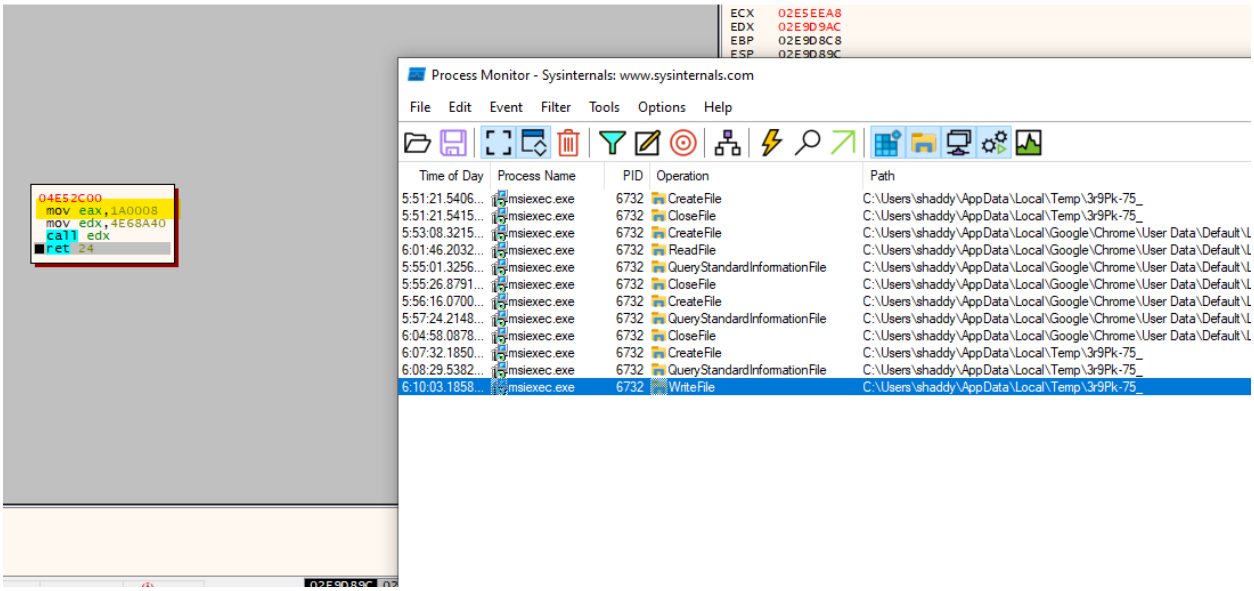




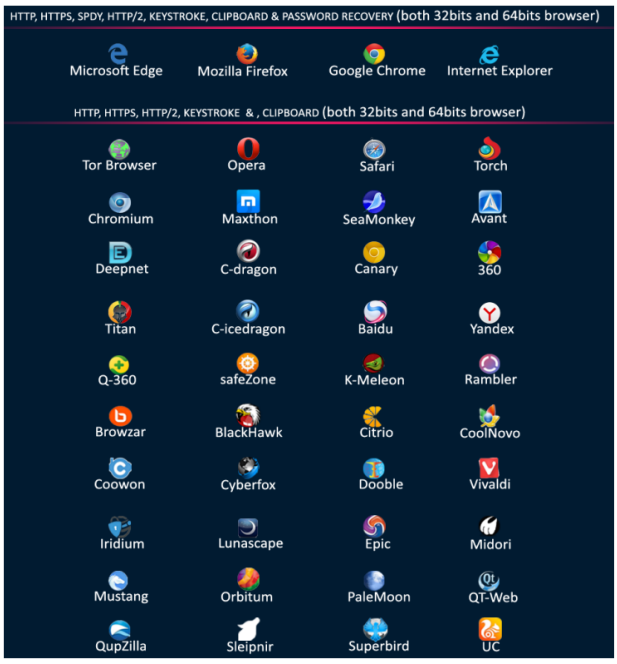


- If it finds anything, it creates a file in temp folder with the static name of "**3r9Pk-75**"
- If the file exists already, it first deletes the previous one and then write new with the updated date.
- Reads the file by the following API sequence
    - ❖ NtCreateFile
    - ❖ NtQueryInformationFile
    - ❖ NtReadFile
    - ❖ NtWriteFile

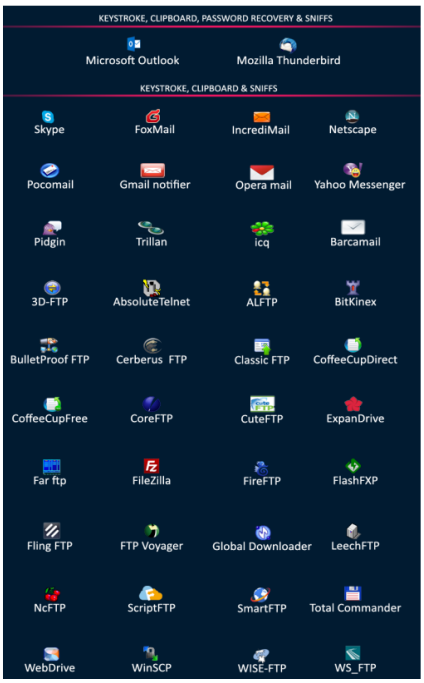

02EE43F8
push 0
lea ecx,dword ptr ds:[esi+2C] ; esi+2C:L"\\3r9Pk-75_"
push ecx
push edi ; edi:L"C:\\Users\\shaddy\\AppData\\Local\\Temp"
call 2EEE7B0
push 0
push 13
push edi ; edi:L"C:\\Users\\shaddy\\AppData\\Local\\Temp"
push ebx
call 2EE7470
mov edx,dword ptr ss:[ebp+C] ; [ebp+C]:L"C:\\Users\\shaddy\\AppData\\Local\\Google\\Chrome\\User Data\\Default\\Login Data"
push edi ; edi:L"C:\\Users\\shaddy\\AppData\\Local\\Temp"
push edx
push ebx
call 2EE37C0
push edi ; edi:L"C:\\Users\\shaddy\\AppData\\Local\\Temp"
mov edi,dword ptr ss:[ebp+C] ; [ebp+C]:L"C:\\Users\\shaddy\\AppData\\Local\\Google\\Chrome\\User Data\\Default\\Login Data"
push edi ; edi:L"C:\\Users\\shaddy\\AppData\\Local\\Temp"
call 2EEEA60
mov eax,dword ptr ds:[esi]
add esp,40
cmp eax,7
jne 2EE4518
Hide FPU
EAX 00000000
EBX 02E9EF58
ECX 0321D1CC L"C:\\Users\\shaddy\\AppData\\Local\\Temp\\3r9Pk-75_"
EDX 0000000A
EBP 02E9DA44
ESP 02E9DA00 &L"C:\\Users\\shaddy\\AppData\\Local\\Temp\\3r9Pk-75_"
ESI 0321CDE0
EDI 0321D1CC L"C:\\Users\\shaddy\\AppData\\Local\\Temp\\3r9Pk-75_"
EIP 02EE4409
EFLAGS 00000246
ZF 1 PF 1 AF 0
OF 0 SF 0 DF 0
CF 0 TF 0 IF 1
LastError 00000000 (ERROR_SUCCESS)
LastStatus 00000000 (STATUS_SUCCESS)
GS 002B FS 0053
ES 002B DS 002B
CS 0023 SS 002B
ST(0) 00000000000000000000 x87r0 Empty 0.00000000000000000000
ST(1) 00000000000000000000 x87r1 Empty 0.00000000000000000000
ST(2) 00000000000000000000 x87r2 Empty 0.00000000000000000000
ST(3) 00000000000000000000 x87r3 Empty 0.00000000000000000000
ST(4) 00000000000000000000 x87r4 Empty 0.00000000000000000000
ST(5) 00000000000000000000 x87r5 Empty 0.00000000000000000000
ST(6) 3FFF8000000000000000 x87r6 Empty 1.00000000000000000000
ST(7) BFFF8000000000000000 x87r7 Empty -1.00000000000000000000
Default (stdcall)
1: [esp+4] 00000013
2: [esp+8] 00000000
3: [esp+C] 0321D1CC L"C:\\Users\\shaddy\\AppData\\Local\\Temp\\3r9Pk-75_"
4: [esp+10] 0321CE0C L"\\3r9Pk-75_"
5: [esp+14] 00000000


ECX 02E5EEA8
EDX 02E9D9AC
EBP 02E9D8C8
ESP 02E9D89C
04E52C00
mov eax,1A0008
mov edx,4E68A40
call edx
ret 24
Process Monitor - Sysinternals: www.sysinternals.com
File Edit Event Filter Tools Options Help
Time of Day Process Name PID Operation Path
5:51:21.5406... msiexec.exe 6732 CreateFile C:\Users\shaddy\AppData\Local\Temp\3r9Pk-75_
5:51:21.5415... msiexec.exe 6732 CloseFile C:\Users\shaddy\AppData\Local\Temp\3r9Pk-75_
5:53:08.3215... msiexec.exe 6732 CreateFile C:\Users\shaddy\AppData\Local\Google\Chrome\User Data\Default\L
6:01:46.2032... msiexec.exe 6732 ReadFile C:\Users\shaddy\AppData\Local\Google\Chrome\User Data\Default\L
5:55:01.3256... msiexec.exe 6732 QueryStandardInformationFile C:\Users\shaddy\AppData\Local\Google\Chrome\User Data\Default\L
5:55:26.8791... msiexec.exe 6732 CloseFile C:\Users\shaddy\AppData\Local\Google\Chrome\User Data\Default\L
5:56:16.0700... msiexec.exe 6732 CreateFile C:\Users\shaddy\AppData\Local\Google\Chrome\User Data\Default\L
5:57:24.2148... msiexec.exe 6732 QueryStandardInformationFile C:\Users\shaddy\AppData\Local\Google\Chrome\User Data\Default\L
6:04:58.0878... msiexec.exe 6732 CloseFile C:\Users\shaddy\AppData\Local\Google\Chrome\User Data\Default\L
6:07:32.1850... msiexec.exe 6732 CreateFile C:\Users\shaddy\AppData\Local\Temp\3r9Pk-75_
6:08:29.5382... msiexec.exe 6732 QueryStandardInformationFile C:\Users\shaddy\AppData\Local\Temp\3r9Pk-75_
6:10:03.1858... msiexec.exe 6732 WriteFile C:\Users\shaddy\AppData\Local\Temp\3r9Pk-75_

Web-Browsers



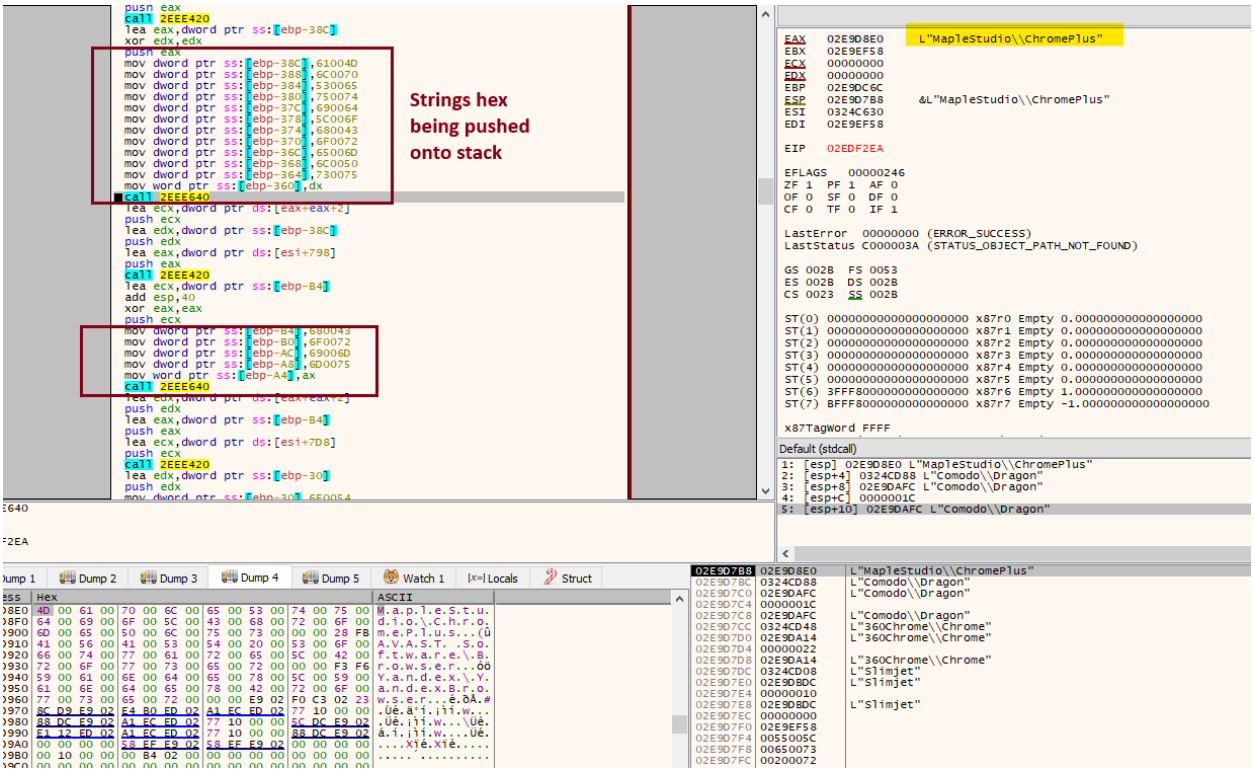
Mail clients, FTP clients, IM apps



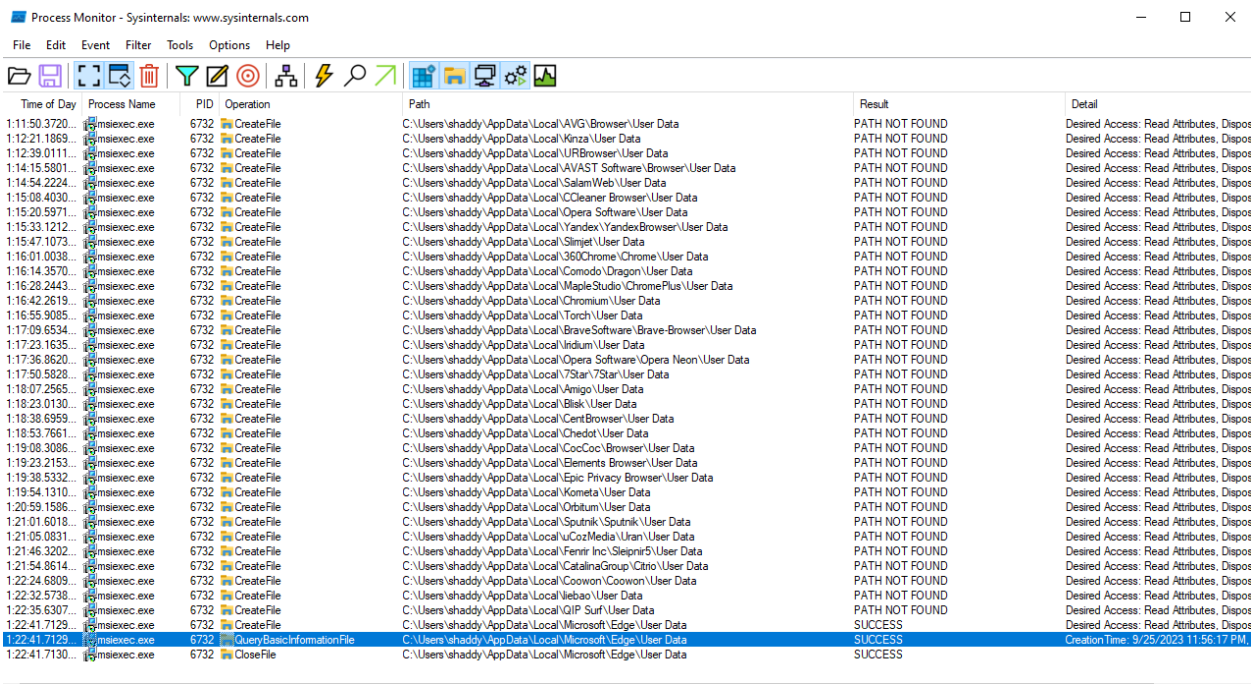
Targeted processes

Decrypted Functions:

- A lot of data is hidden at first because of encrypted functions
- Similar to stage2 malware, the stage3 version also have encrypted functions in it
- Those are decrypted at run-time
- Those functions also contain encrypted hex-based strings for targeted processes
- The strings for targeted applications and paths are pushed onto stack at run-time.

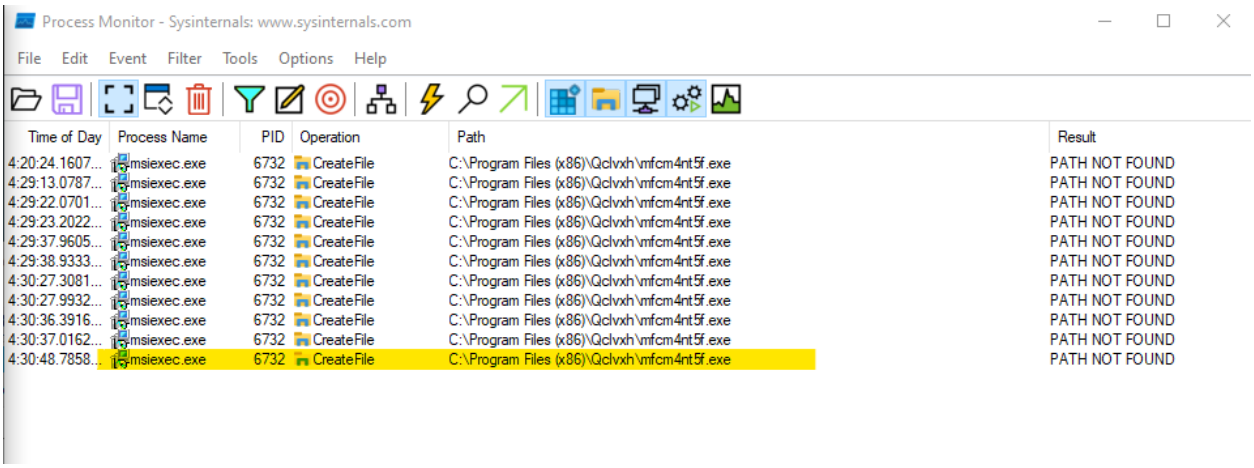


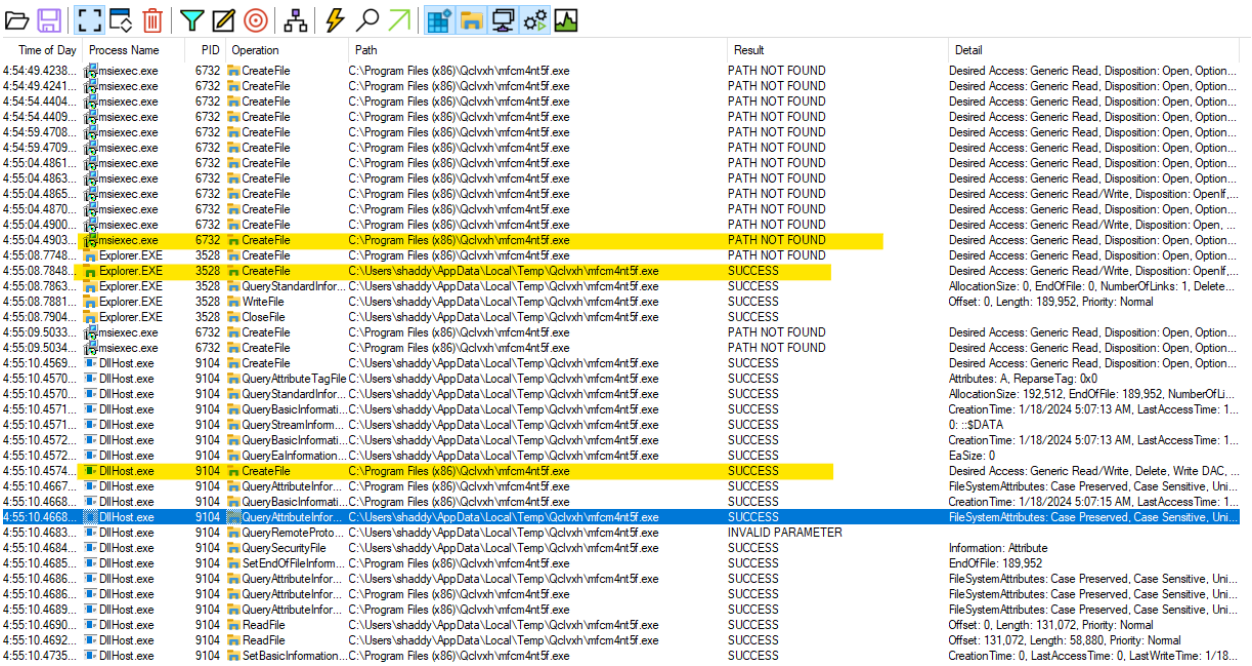
Address	Length	Result
0x2e9c4c1	16	><PProgramFiles
0x2e9c99a	13	LOCALAPPDATA
0x2e9d48c	58	C:\Users\shaddy\AppData\Local
0x2e9d694	24	LOCALAPPDATA
0x2e9d7f4	20	\User Data
0x2e9d834	42	Chromium Recovery
0x2e9d874	54	BraveSoftware\Brave-Browser
0x2e9d8ac	50	Opera Software\Opera Neon
0x2e9d8e0	44	MapleStudio\ChromePlus
0x2e9d910	44	(VAST Software\Browser
0x2e9d940	40	Yandex\YandexBrowser
0x2e9d96c	40	CatalinaGroup\Citrio
0x2e9d998	40	Fenrir Inc\Sleipnir5
0x2e9d9c4	40	Epic Privacy Browser
0x2e9d9f0	32	Elements Browser
0x2e9da14	30	360Chrome\ChroX
0x2e9e070	12	vaultcli.dll
0x2e9e30c	182	/c copy "C:\Users\shaddy\Desktop\dump.exe" "C:\Program Files (x86)\Qclvxh\mfcm4nt5f.exe" /V
0x2e9e40c	22	dllhost.exe
0x2e9e8bc	11	dllhost.exe
0x2e9f19b	10	{3]I:JW*h
0x2e9f2f8	86	C:\Program Files (x86)\Qclvxh\mfcm4nt5f.exe



Privilege Escalation:

- Privileges are escalated by abusing the dllhost.exe and COM objects
- It keeps trying to copy the stage2 malware in Program Files
- If proper privileges are not provided, it then uses explorer to write stage2 malware in temp and by abusing dllhost, it copies the malware to Program Files

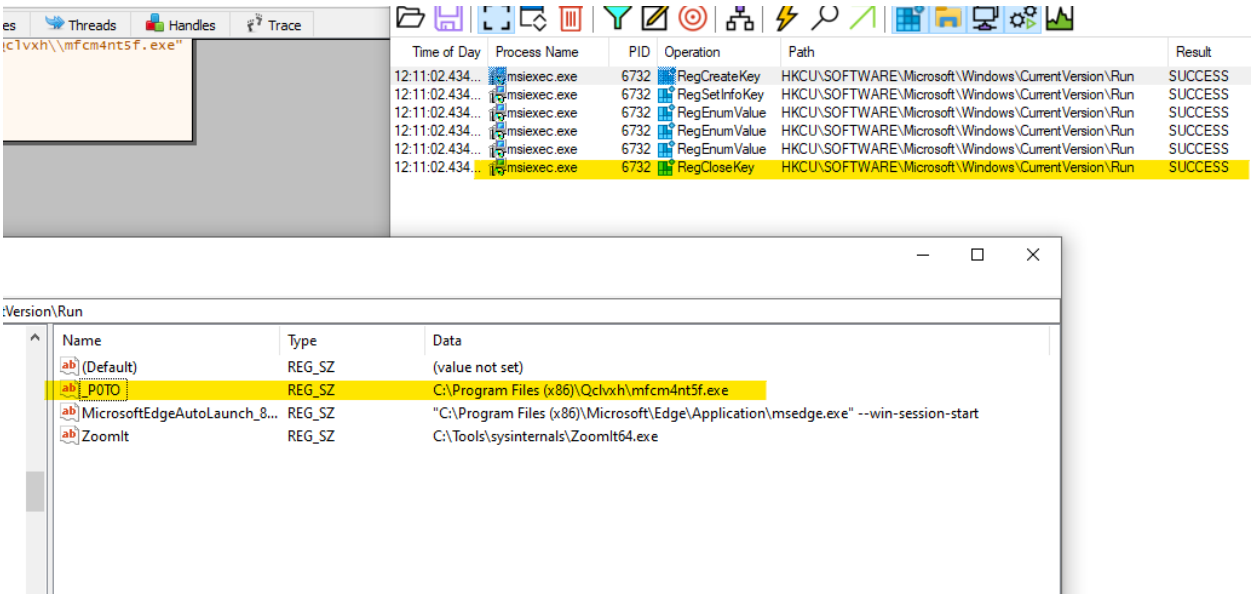




Time of Day	Process Name	PID	Operation	Path	Result	Detail
4:54:49.4238...	msiexec.exe	6732	CreateFile	C:\Program Files (x86)\Qclvxh\mfcm4nt5f.exe	PATH NOT FOUND	Desired Access: Generic Read, Disposition: Open, Option...
4:54:49.4241...	msiexec.exe	6732	CreateFile	C:\Program Files (x86)\Qclvxh\mfcm4nt5f.exe	PATH NOT FOUND	Desired Access: Generic Read, Disposition: Open, Option...
4:54:54.4404...	msiexec.exe	6732	CreateFile	C:\Program Files (x86)\Qclvxh\mfcm4nt5f.exe	PATH NOT FOUND	Desired Access: Generic Read, Disposition: Open, Option...
4:54:54.4409...	msiexec.exe	6732	CreateFile	C:\Program Files (x86)\Qclvxh\mfcm4nt5f.exe	PATH NOT FOUND	Desired Access: Generic Read, Disposition: Open, Option...
4:54:59.4708...	msiexec.exe	6732	CreateFile	C:\Program Files (x86)\Qclvxh\mfcm4nt5f.exe	PATH NOT FOUND	Desired Access: Generic Read, Disposition: Open, Option...
4:54:59.4709...	msiexec.exe	6732	CreateFile	C:\Program Files (x86)\Qclvxh\mfcm4nt5f.exe	PATH NOT FOUND	Desired Access: Generic Read, Disposition: Open, Option...
4:55:04.4861...	msiexec.exe	6732	CreateFile	C:\Program Files (x86)\Qclvxh\mfcm4nt5f.exe	PATH NOT FOUND	Desired Access: Generic Read, Disposition: Open, Option...
4:55:04.4863...	msiexec.exe	6732	CreateFile	C:\Program Files (x86)\Qclvxh\mfcm4nt5f.exe	PATH NOT FOUND	Desired Access: Generic Read, Disposition: Open, Option...
4:55:04.4865...	msiexec.exe	6732	CreateFile	C:\Program Files (x86)\Qclvxh\mfcm4nt5f.exe	PATH NOT FOUND	Desired Access: Generic Read/Write, Disposition: OpenIf,...
4:55:04.4870...	msiexec.exe	6732	CreateFile	C:\Program Files (x86)\Qclvxh\mfcm4nt5f.exe	PATH NOT FOUND	Desired Access: Generic Read, Disposition: Open, Option...
4:55:04.4900...	msiexec.exe	6732	CreateFile	C:\Program Files (x86)\Qclvxh\mfcm4nt5f.exe	PATH NOT FOUND	Desired Access: Generic Read/Write, Disposition: Open, ...
4:55:04.4903...	msiexec.exe	6732	CreateFile	C:\Program Files (x86)\Qclvxh\mfcm4nt5f.exe	PATH NOT FOUND	Desired Access: Generic Read, Disposition: Open, Option...
4:55:08.7748...	Explorer.EXE	3528	CreateFile	C:\Program Files (x86)\Qclvxh\mfcm4nt5f.exe	PATH NOT FOUND	Desired Access: Generic Read, Disposition: Open, Option...
4:55:08.7848...	Explorer.EXE	3528	CreateFile	C:\Users\shaddy\AppData\Local\Temp\Qclvxh\mfcm4nt5f.exe	SUCCESS	Desired Access: Generic Read/Write, Disposition: OpenIf,...
4:55:08.7863...	Explorer.EXE	3528	QueryStandardInfor...	C:\Users\shaddy\AppData\Local\Temp\Qclvxh\mfcm4nt5f.exe	SUCCESS	AllocationSize: 0, EndOfFile: 0, NumberOfLinks: 1, Delete...
4:55:08.7881...	Explorer.EXE	3528	WriteFile	C:\Users\shaddy\AppData\Local\Temp\Qclvxh\mfcm4nt5f.exe	SUCCESS	Offset: 0, Length: 189,952, Priority: Normal
4:55:08.7904...	Explorer.EXE	3528	CloseFile	C:\Users\shaddy\AppData\Local\Temp\Qclvxh\mfcm4nt5f.exe	SUCCESS	
4:55:09.5033...	msiexec.exe	6732	CreateFile	C:\Program Files (x86)\Qclvxh\mfcm4nt5f.exe	PATH NOT FOUND	Desired Access: Generic Read, Disposition: Open, Option...
4:55:09.5034...	msiexec.exe	6732	CreateFile	C:\Program Files (x86)\Qclvxh\mfcm4nt5f.exe	PATH NOT FOUND	Desired Access: Generic Read, Disposition: Open, Option...
4:55:10.4569...	DllHost.exe	9104	CreateFile	C:\Users\shaddy\AppData\Local\Temp\Qclvxh\mfcm4nt5f.exe	SUCCESS	Desired Access: Generic Read, Disposition: Open, Option...
4:55:10.4570...	DllHost.exe	9104	QueryAttributeTagFile	C:\Users\shaddy\AppData\Local\Temp\Qclvxh\mfcm4nt5f.exe	SUCCESS	Attributes: A, ReparseTag: 0x0
4:55:10.4570...	DllHost.exe	9104	QueryStandardInfor...	C:\Users\shaddy\AppData\Local\Temp\Qclvxh\mfcm4nt5f.exe	SUCCESS	AllocationSize: 192,512, EndOfFile: 189,952, NumberOfLi...
4:55:10.4571...	DllHost.exe	9104	QueryBasicInformati...	C:\Users\shaddy\AppData\Local\Temp\Qclvxh\mfcm4nt5f.exe	SUCCESS	CreationTime: 1/18/2024 5:07:13 AM, LastAccessTime: 1...
4:55:10.4571...	DllHost.exe	9104	QueryStreamInform...	C:\Users\shaddy\AppData\Local\Temp\Qclvxh\mfcm4nt5f.exe	SUCCESS	0: ::$DATA
4:55:10.4572...	DllHost.exe	9104	QueryBasicInformati...	C:\Users\shaddy\AppData\Local\Temp\Qclvxh\mfcm4nt5f.exe	SUCCESS	CreationTime: 1/18/2024 5:07:13 AM, LastAccessTime: 1...
4:55:10.4572...	DllHost.exe	9104	QueryEaInformation...	C:\Users\shaddy\AppData\Local\Temp\Qclvxh\mfcm4nt5f.exe	SUCCESS	EaSize: 0
4:55:10.4574...	DllHost.exe	9104	CreateFile	C:\Program Files (x86)\Qclvxh\mfcm4nt5f.exe	SUCCESS	Desired Access: Generic Read/Write, Delete, Write DAC, ...
4:55:10.4667...	DllHost.exe	9104	QueryAttributeInfor...	C:\Program Files (x86)\Qclvxh\mfcm4nt5f.exe	SUCCESS	FileSystemAttributes: Case Preserved, Case Sensitive, Uni...
4:55:10.4668...	DllHost.exe	9104	QueryBasicInformati...	C:\Program Files (x86)\Qclvxh\mfcm4nt5f.exe	SUCCESS	CreationTime: 1/18/2024 5:07:15 AM, LastAccessTime: 1...
4:55:10.4668...	DllHost.exe	9104	QueryAttributeInfor...	C:\Users\shaddy\AppData\Local\Temp\Qclvxh\mfcm4nt5f.exe	SUCCESS	FileSystemAttributes: Case Preserved, Case Sensitive, Uni...
4:55:10.4683...	DllHost.exe	9104	QueryRemoteProto...	C:\Users\shaddy\AppData\Local\Temp\Qclvxh\mfcm4nt5f.exe	INVALID PARAMETER	
4:55:10.4684...	DllHost.exe	9104	QuerySecurityFile	C:\Users\shaddy\AppData\Local\Temp\Qclvxh\mfcm4nt5f.exe	SUCCESS	Information: Attribute
4:55:10.4685...	DllHost.exe	9104	SetEndOfFileInform...	C:\Program Files (x86)\Qclvxh\mfcm4nt5f.exe	SUCCESS	EndOfFile: 189,952
4:55:10.4686...	DllHost.exe	9104	QueryAttributeInfor...	C:\Users\shaddy\AppData\Local\Temp\Qclvxh\mfcm4nt5f.exe	SUCCESS	FileSystemAttributes: Case Preserved, Case Sensitive, Uni...
4:55:10.4686...	DllHost.exe	9104	QueryAttributeInfor...	C:\Program Files (x86)\Qclvxh\mfcm4nt5f.exe	SUCCESS	FileSystemAttributes: Case Preserved, Case Sensitive, Uni...
4:55:10.4689...	DllHost.exe	9104	QueryAttributeInfor...	C:\Program Files (x86)\Qclvxh\mfcm4nt5f.exe	SUCCESS	FileSystemAttributes: Case Preserved, Case Sensitive, Uni...
4:55:10.4690...	DllHost.exe	9104	ReadFile	C:\Users\shaddy\AppData\Local\Temp\Qclvxh\mfcm4nt5f.exe	SUCCESS	Offset: 0, Length: 131,072, Priority: Normal
4:55:10.4692...	DllHost.exe	9104	ReadFile	C:\Users\shaddy\AppData\Local\Temp\Qclvxh\mfcm4nt5f.exe	SUCCESS	Offset: 131,072, Length: 58,880, Priority: Normal
4:55:10.4735...	DllHost.exe	9104	SetBasicInformation...	C:\Program Files (x86)\Qclvxh\mfcm4nt5f.exe	SUCCESS	CreationTime: 0, LastAccessTime: 0, LastWriteTime: 1/18...

Persistence:

- After the malware is copied in Program Files
- It achieves persistence by adding Run Registry Keys
- It uses the API **NtCreateKey**



Threads | Handles | Trace

Time of Day	Process Name	PID	Operation	Path	Result
12:11:02.434...	msiexec.exe	6732	RegCreateKey	HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run	SUCCESS
12:11:02.434...	msiexec.exe	6732	RegSetInfoKey	HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run	SUCCESS
12:11:02.434...	msiexec.exe	6732	RegEnumValue	HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run	SUCCESS
12:11:02.434...	msiexec.exe	6732	RegEnumValue	HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run	SUCCESS
12:11:02.434...	msiexec.exe	6732	RegEnumValue	HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run	SUCCESS
12:11:02.434...	msiexec.exe	6732	RegCloseKey	HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run	SUCCESS

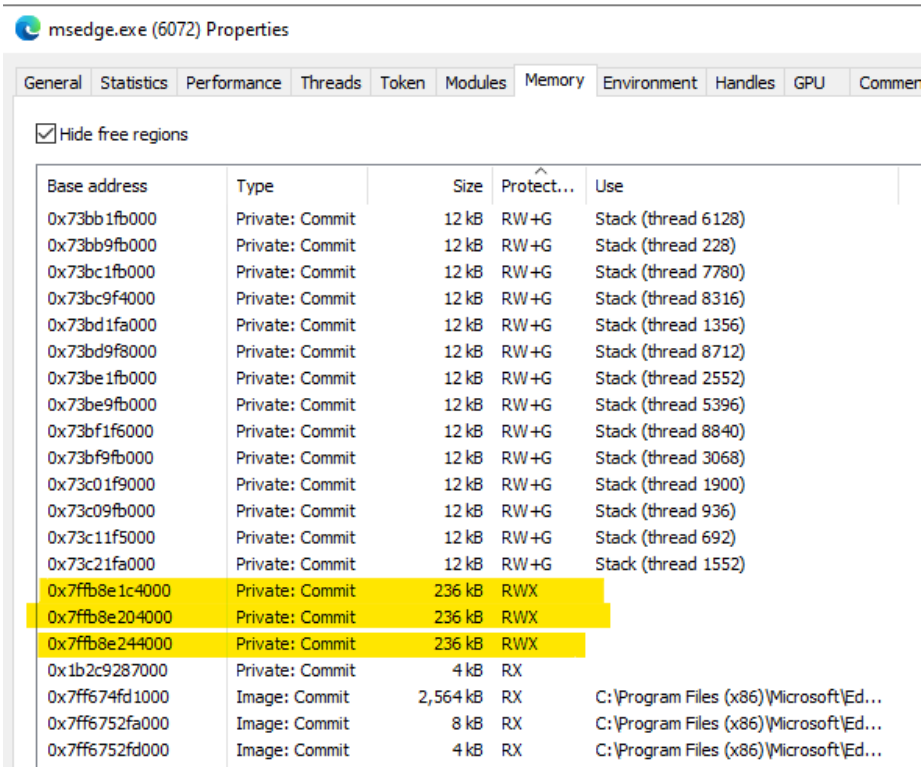
:Version\Run

Name	Type	Data
(Default)	REG_SZ	(value not set)
P0TO	REG_SZ	C:\Program Files (x86)\Qclvxh\mfcm4nt5f.exe
MicrosoftEdgeAutoLaunch_8...	REG_SZ	"C:\Program Files (x86)\Microsoft\Edge\Application\msedge.exe" --win-session-start
ZoomIt	REG_SZ	C:\Tools\sysinternals\ZoomIt64.exe

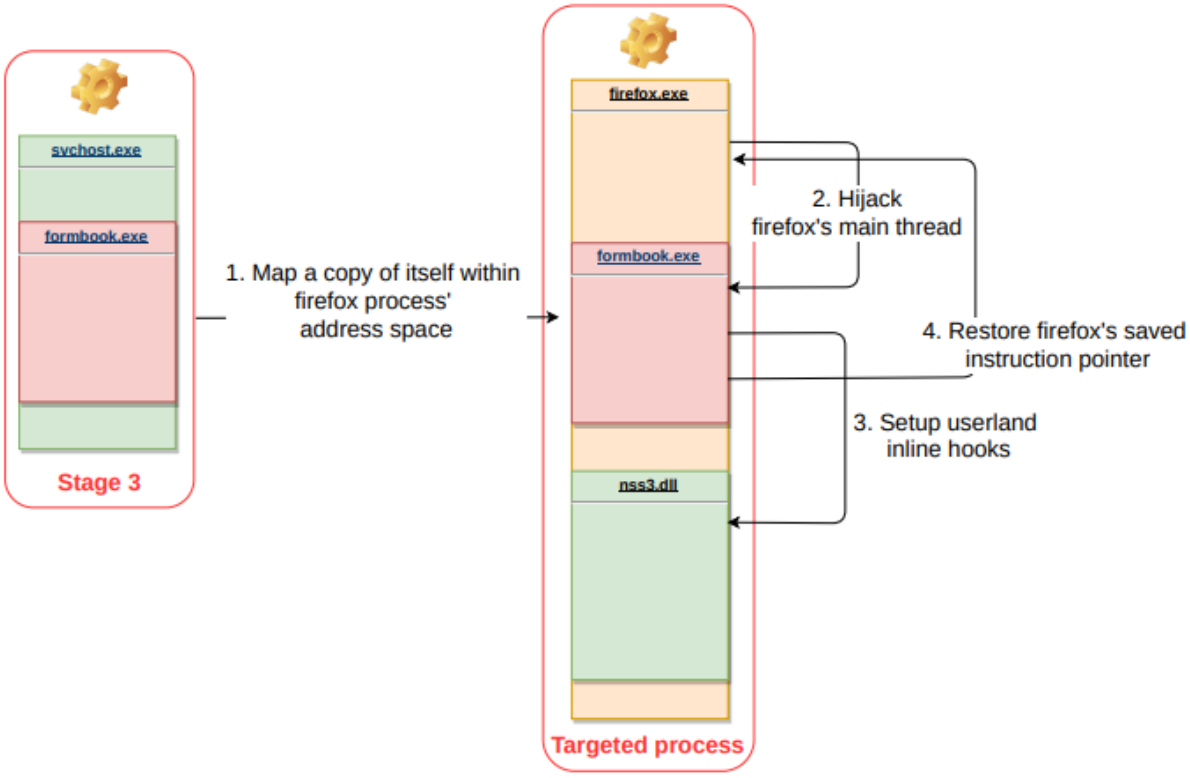
Setting Inline Hooks:

- Xloader also works as a form grabber

- It sets inline hooks in targeted processes for stealing plaintext data from the parameters of the functions
- The data stolen from victim processes is saved in a shared memory between 3 processes
    - ❖ Victim Process
    - ❖ Stage3 Malware
    - ❖ Explorer
- The xloader is stuck in a loop here
- On every loop, it does the following:
    - ❖ Enumerates all running processes
    - ❖ Set inline hooks in targeted processes if found (by injecting code)
    - ❖ Steal clipboard data
    - ❖ Tries to create a file in program files
    - ❖ Adds registry in RunKeys
    - ❖ Send a POST & GET request on one of the resolved c2 servers through **explorer.exe.** It has an injected payload in explorer.exe that it uses for exfiltrating stolen data.



Base address	Type	Size	Protect...	Use
0x73bb1fb000	Private: Commit	12 kB	RW+G	Stack (thread 6128)
0x73bb9fb000	Private: Commit	12 kB	RW+G	Stack (thread 228)
0x73bc1fb000	Private: Commit	12 kB	RW+G	Stack (thread 7780)
0x73bc9f4000	Private: Commit	12 kB	RW+G	Stack (thread 8316)
0x73bd1fa000	Private: Commit	12 kB	RW+G	Stack (thread 1356)
0x73bd9f8000	Private: Commit	12 kB	RW+G	Stack (thread 8712)
0x73be1fb000	Private: Commit	12 kB	RW+G	Stack (thread 2552)
0x73be9fb000	Private: Commit	12 kB	RW+G	Stack (thread 5396)
0x73bf1f6000	Private: Commit	12 kB	RW+G	Stack (thread 8840)
0x73bf9fb000	Private: Commit	12 kB	RW+G	Stack (thread 3068)
0x73c01f9000	Private: Commit	12 kB	RW+G	Stack (thread 1900)
0x73c09fb000	Private: Commit	12 kB	RW+G	Stack (thread 936)
0x73c11f5000	Private: Commit	12 kB	RW+G	Stack (thread 692)
0x73c21fa000	Private: Commit	12 kB	RW+G	Stack (thread 1552)
0x7ffb8e1c4000	Private: Commit	236 kB	RWX	
0x7ffb8e204000	Private: Commit	236 kB	RWX	
0x7ffb8e244000	Private: Commit	236 kB	RWX	
0x1b2c9287000	Private: Commit	4 kB	RX	
0x7ff674fd1000	Image: Commit	2,564 kB	RX	C:\Program Files (x86)\Microsoft\Ed...
0x7ff6752fa000	Image: Commit	8 kB	RX	C:\Program Files (x86)\Microsoft\Ed...
0x7ff6752fd000	Image: Commit	4 kB	RX	C:\Program Files (x86)\Microsoft\Ed...



1. NtOpenProcess(), NtCreateSection(), NtMapViewOfSection()
2. NtOpenThread(), NtSuspendThread(), NtGetThreadContext(), NtSetThreadContext(), NtResumeThread()
3. NtProtectVirtualMemory()
4. ret instruction to saved *CONTEXT.Eip*

Web-browsers targeted functions

DLL	Function	Browser	Pre-encryption
secur32.dll	EncryptMessage		Yes
wininet.dll	HttpSendRequestA HttpSendRequestW InternetQueryOptionW		Yes
nspr4.dll	PR_Write		Yes
nss3.dll	PR_Write		Yes
ws2_32.dll	WSASend		No

References:

- https://www.fortinet.com/blog/threat-research/deep-analysis-formbook-new-variant-delivered-phishing-campaign-part-ii
- https://www.zscaler.com/blogs/security-research/technical-analysis-xloader-s-code-obfuscation-version-4-3
- https://www.zscaler.com/blogs/security-research/analysis-xloader-s-c2-network-encryption
- https://www.botconf.eu/botconf-presentation-or-article/in-depth-formbook-malware-analysis/