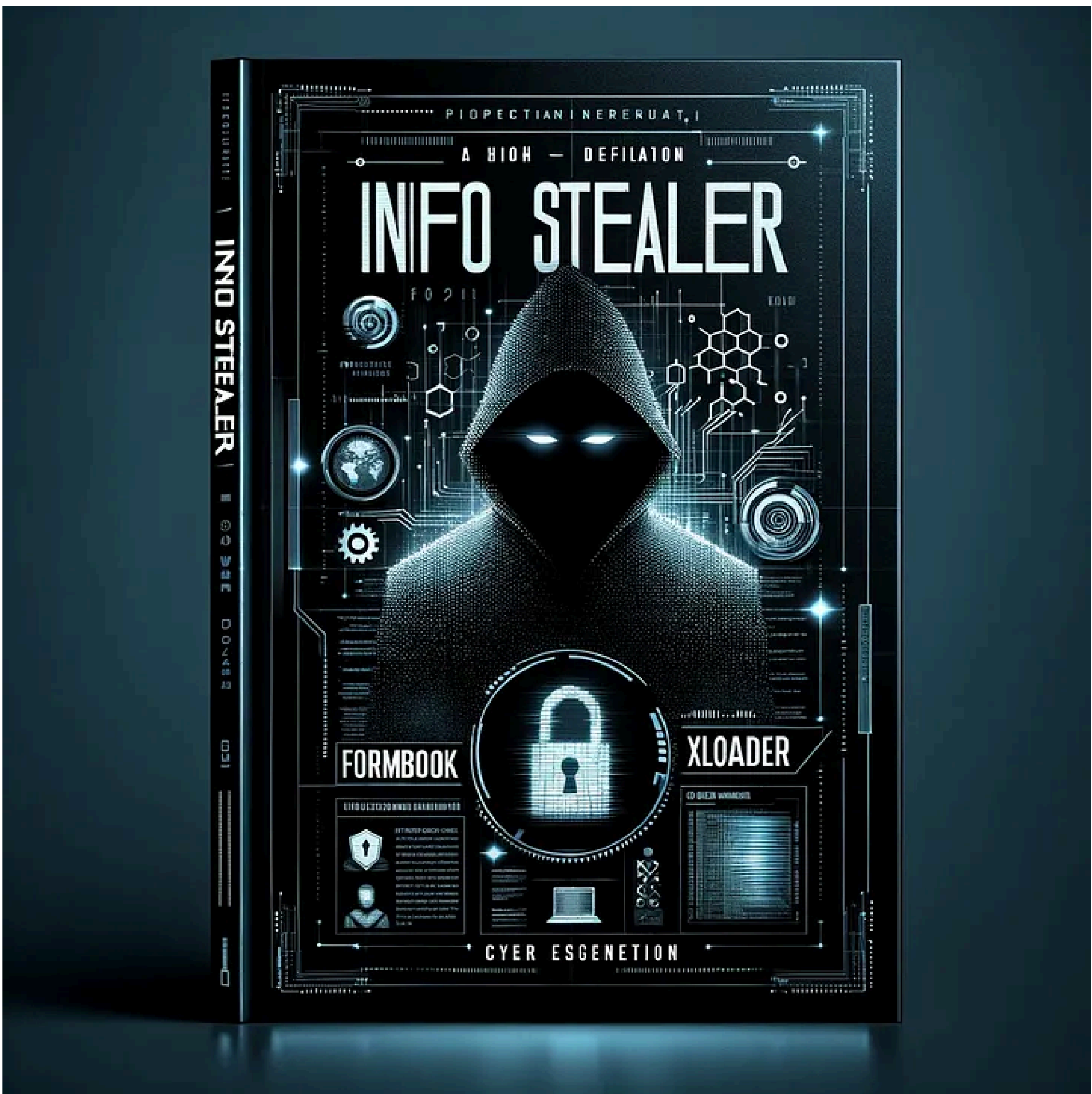
Layers of Deception: Analyzing the Complex Stages of XLoader 4.3 Malware Evolution

By Shayan Ahmed Khan

Published: 2024-04-25 · Archived: 2026-04-05 17:45:58 UTC

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.

Press enter or click to view image in full size



Key Findings:

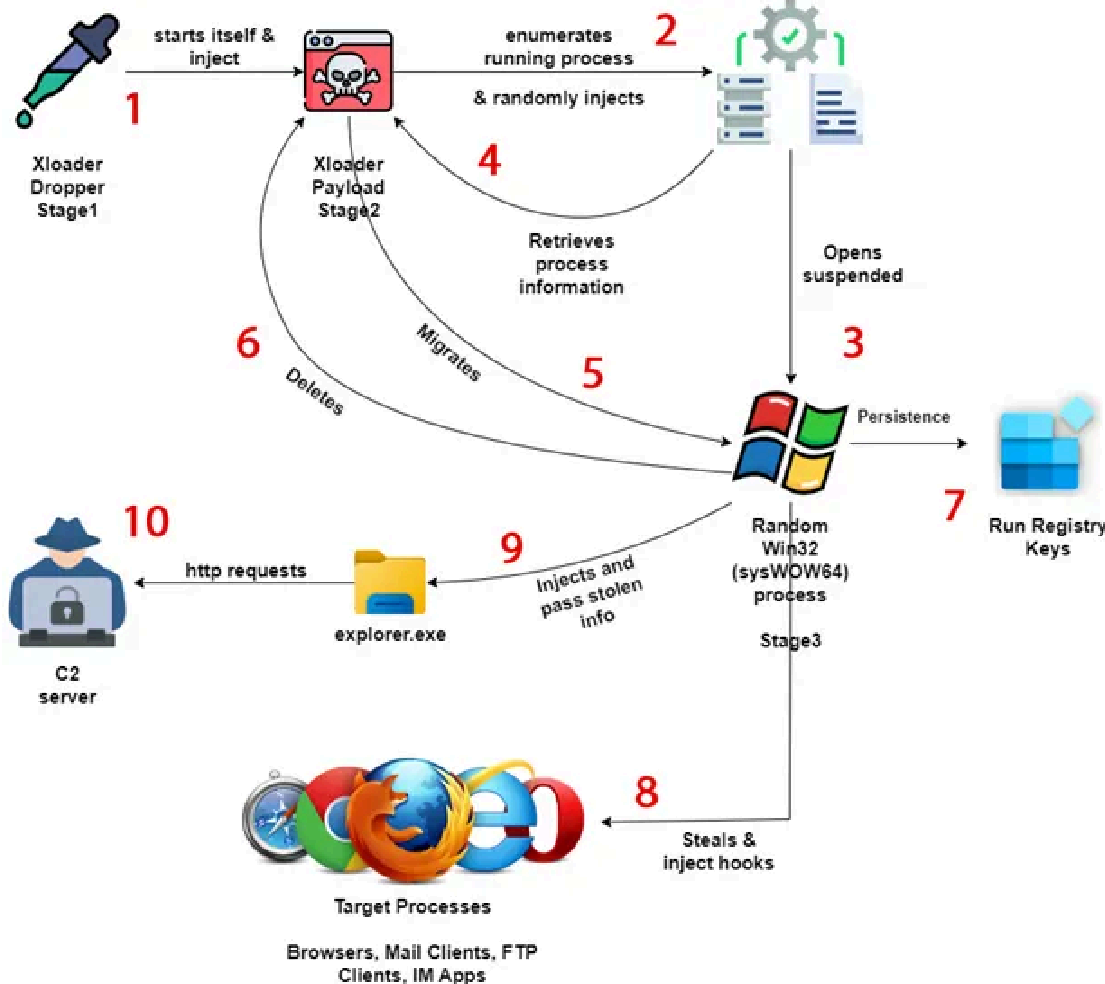
- **Initial Dropper:** Xloader uses a similar initial dropper as some of the other infostealers like Remcos RAT and Agent Tesla. The initial dropper is a dotnet executable file, which contains multiple embedded **DLLs** which are extracted and decrypted at run-time to launch the payload which is the actual malware. The payload is launched using **Process Hollowing** in either itself or another running process, depending upon the configuration of the initial dropper.
- **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.
- **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.
- **Custom Encryption Algorithms:** It uses a **Custom RC4** encryption/decryption algorithm with additional subtraction operations.

- **API/String/Libraries Hashing:** Xloader uses **CRC32/BZIP2** hashing algorithm for its strings, libraries and APIs to hide its internal working.
- **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.
- **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. The process is known as "**Lagos Island Method**" according to FireEye.
- **Persistence:** Xloader adds persistence using Run Registry Keys and copying itself in Program Files (x86).
- **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.
- **Process Injection:** Xloader relies heavily on process injection. It infects multiple processes in its execution and even migrate to a different process.
- **Decoy C2s:** It uses a combination of decoy C2 servers and made significant effort to hide its real C2.
- **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.

Check out my Github Repo for Malware Analysis Series!!!

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:

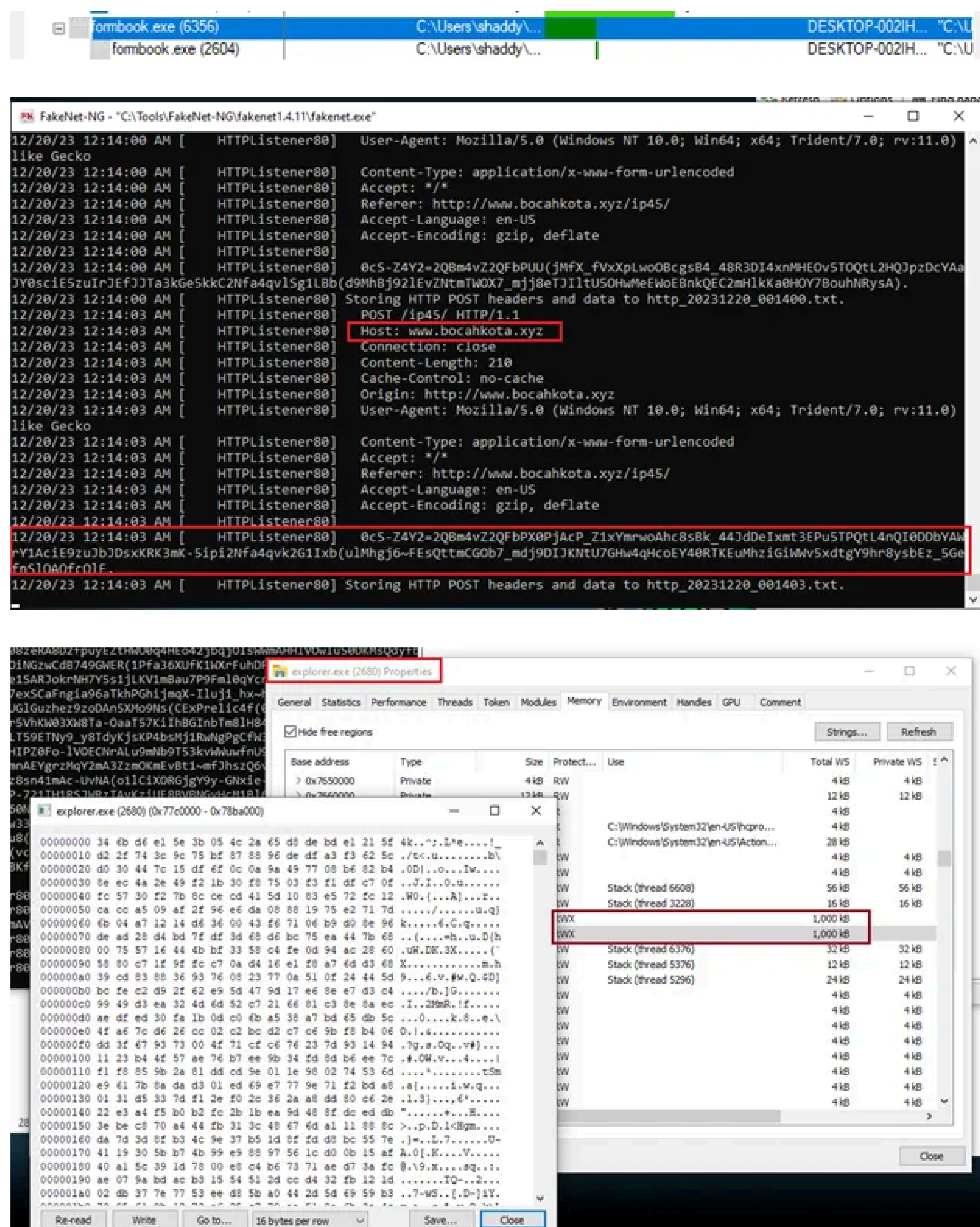
Starting with the initial detonation of xloader. I have detonated the malware in my isolated analysis environment in the presence of procmon, wireshark and other such analysis tools. **Nothing happened!!!** Which likely suggests that there are anti-analysis techniques in the malware. I tried detonating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

Press enter or click to view image in full size

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.

```
220  // Token: 0x06000022 RID: 34 RVA: 0x00003704 File Offset: 0x00001904
221  private void InitializeComponent()
222  {
223      ChartArea chartArea = new ChartArea();
224      Legend legend = new Legend();                    JUNK
225      Series series = new Series();
226      ComponentResourceManager componentResourceManager = new ComponentResourceManager(typeof(View));
227      this.main_chart = new Chart();
228      this.a_tb = new TextBox();
229      this.label1 = new Label();
230      this.label2 = new Label();
231      this.label3 = new Label();
232      this.label4 = new Label();                       JUNK
233      List<byte> list = new List<byte>();
234      byte[] array = (byte[])componentResourceManager.GetObject("Quartz");
235      Array.Reverse(array);
236      list.AddRange(array);
237      list.AddRange((byte[])componentResourceManager.GetObject("Versa"));
238      list.AddRange((byte[])componentResourceManager.GetObject("Zinc"));
239      this.label5 = new Label();
240      this.label6 = new Label();
241      this.label7 = new Label();
242      this.label8 = new Label();
243      this.label9 = new Label();
244      this.label10 = new Label();
245      this.label11 = new Label();
246      this.label12 = new Label();
247      this.label13 = new Label();
248      this.label14 = new Label();
249      this.label15 = new Label();
250      this.label16 = new Label();                      JUNK
251      this.label17 = new Label();
252      this.label18 = new Label();
253      this.accept_button = new Button();
254      this.next_button = new Button();
255      this.n_tb = new TextBox();
256      this.k_tb = new TextBox();
257      this.lambda_tb = new TextBox();
258      this.h_tb = new TextBox();
259      this.min_beta_tb = new TextBox();
260      this.max_beta_tb = new TextBox();
```

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

· Quartz which is also reversed

· Versa

· Zinc

Get Shayan Ahmed Khan’s stories in your inbox

Join Medium for free to get updates from this writer.

Remember me for faster sign in

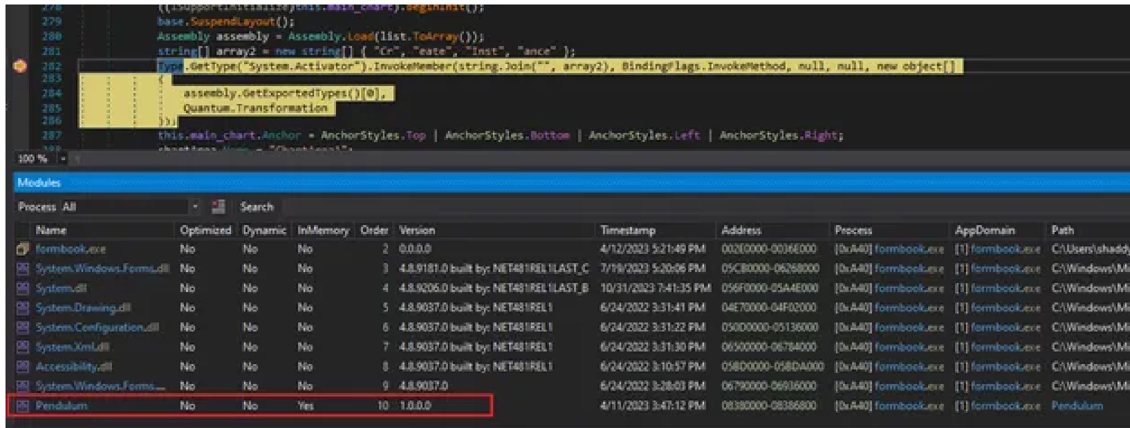
These 3 are the malicious resources that are combined and loaded at run-time for further execution. After going through a lot of junk code, I came across the line of code that resolves this assembly at run-time and create instance of resource followed by loading the first method using **System.Activator**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

```
276  this.min_e_tb = new TextBox();
277  this.max_e_tb = new TextBox();
278  ((ISupportInitialize)this.main_chart).BeginInit();
279  base.SuspendLayout();
280  Assembly assembly = Assembly.Load(list.ToArray());
281  string[] array2 = new string[] { "Cr", "eate", "Inst", "ance" };
282  Type.GetType("System.Activator").InvokeMember(string.Join("", array2), BindingFlags.InvokeMethod, null, null, new object[]
283  {
284      assembly.GetExportedTypes()[0],
285      Quantum.Transformation
286  });
```

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.

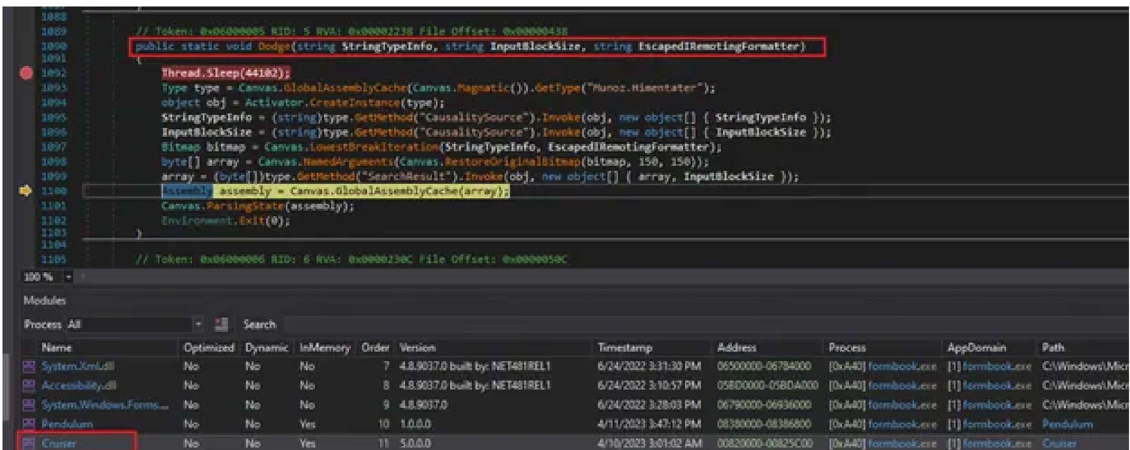
- The runtime binary that has been loaded can be seen in the modules window.

- 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.



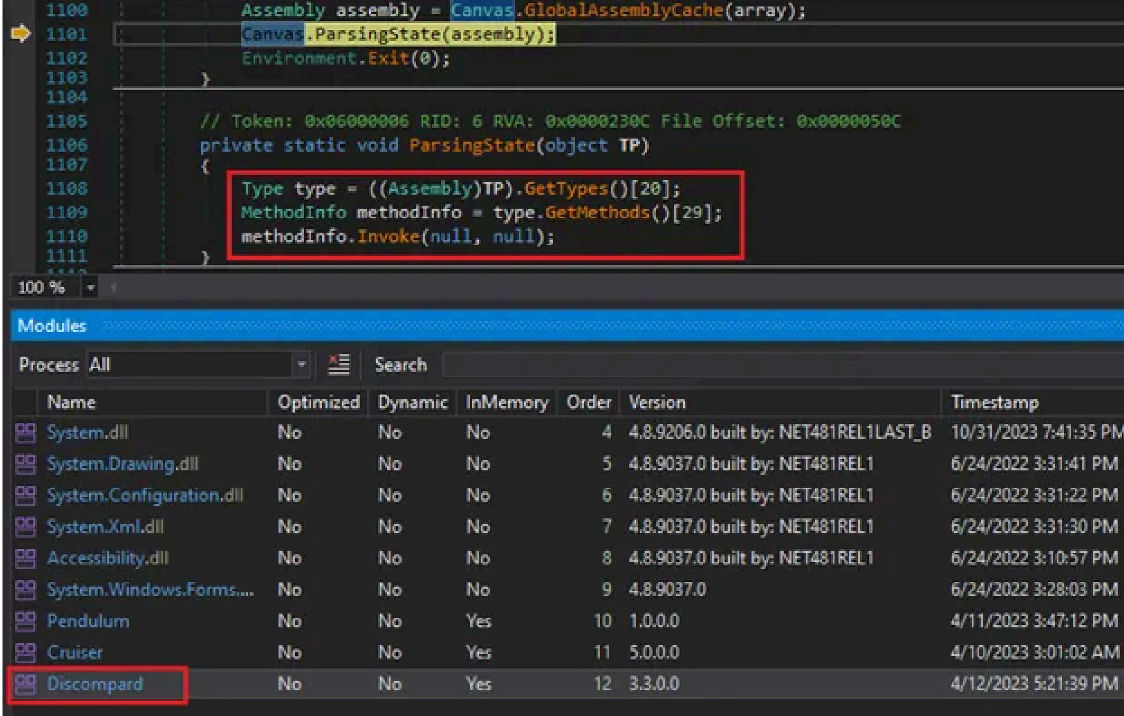
There are further binaries being resolved from the resource of first loaded DLL which is Pendulum. In the modules tab, we can trace which dlls are being added and keep following through.

- 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.

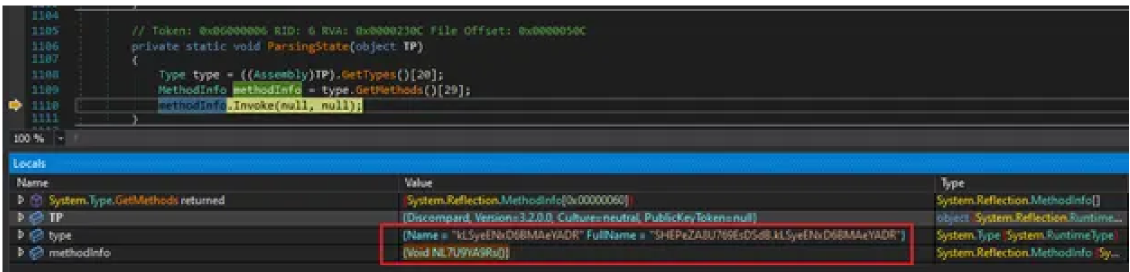


- The last resource that has been decrypted and loaded is called **Discompard.dll**.
- In the method of ParsingState, it could be seen that a method from this assembly is being called for further execution of malware.

  Press enter or click to view image in full size

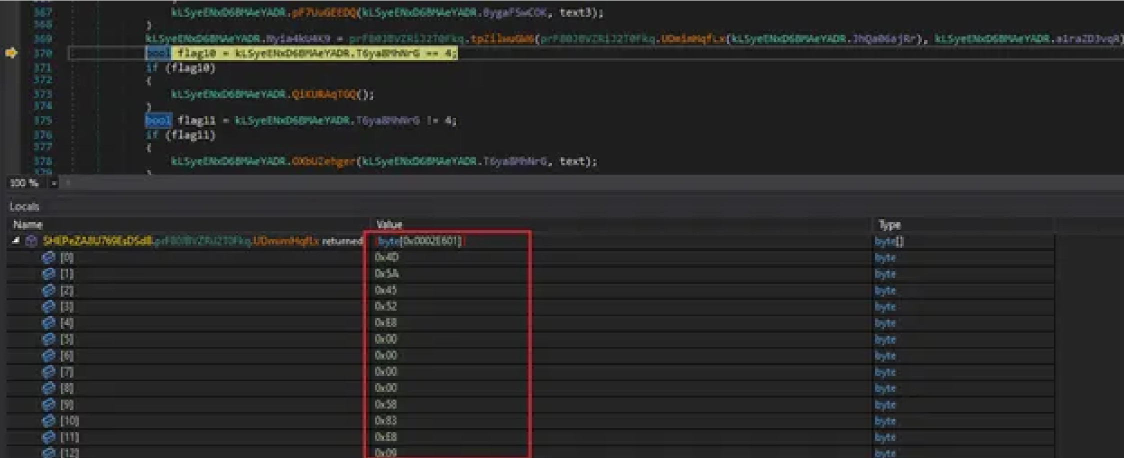


- We can also see the names of classes and methods that are being called from this assembly in the locals. Using this information, we can then setup another breakpoint in the **Discompard.dll** method and continue debugging the 3rd resource.
- Again, we can explore the third binary and setup a breakpoint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

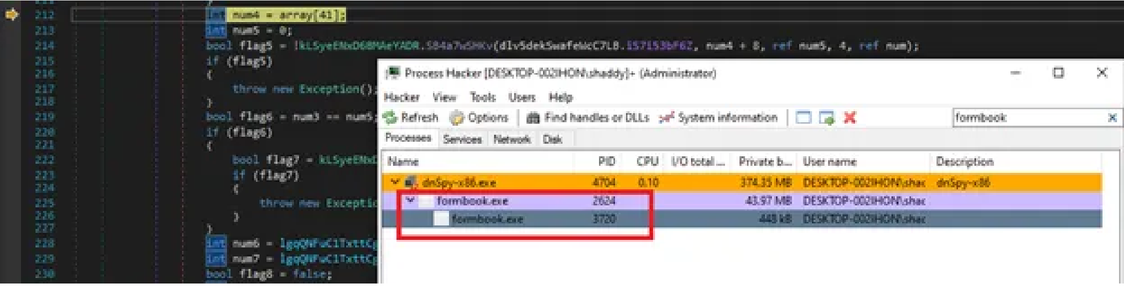


We have now entered the method called by the previous dll. This binary is highly obfuscated with random variable and class names. Normally, what I do is that I check if a deobfuscator like de4dot or some other tool is able to deobfuscate such a binary. If it is possible then I patch the resource and 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.

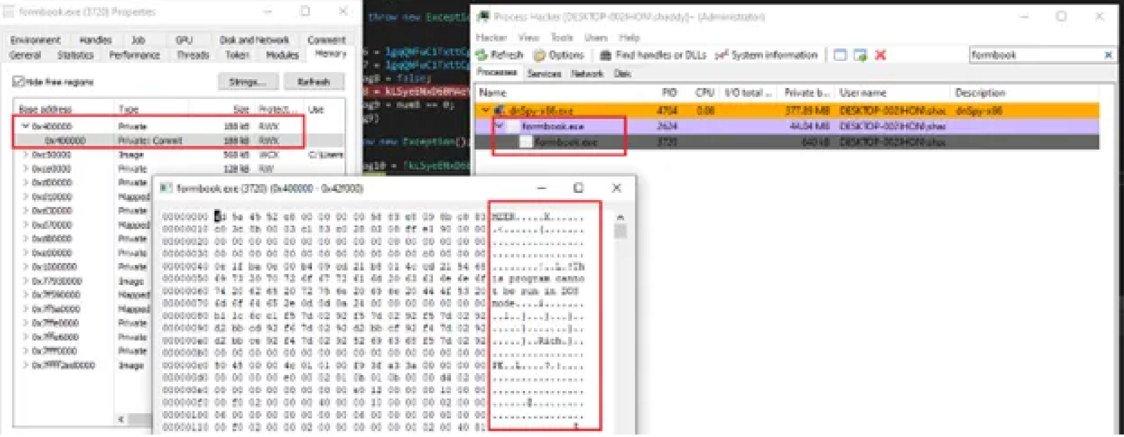
- I kept stepping over and checking the variables and function returns.
- It skipped most of the flags but then I stepped over a function and a return value shows that another binary has been returned. The MZ bytes (4D 5A) could be seen in the array.



- 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.

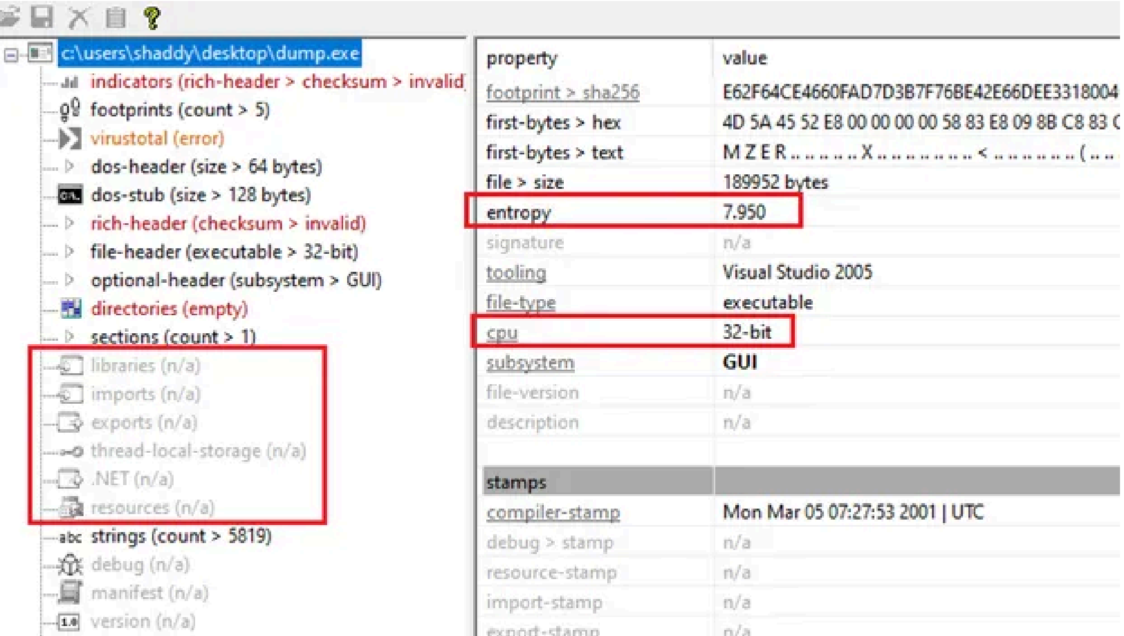


- This is the exact behavior of process hollowing.
- I dumped this shellcode to analyze the malware separately as a second stage payload.
- The stage2 malware is the real xloader payload.

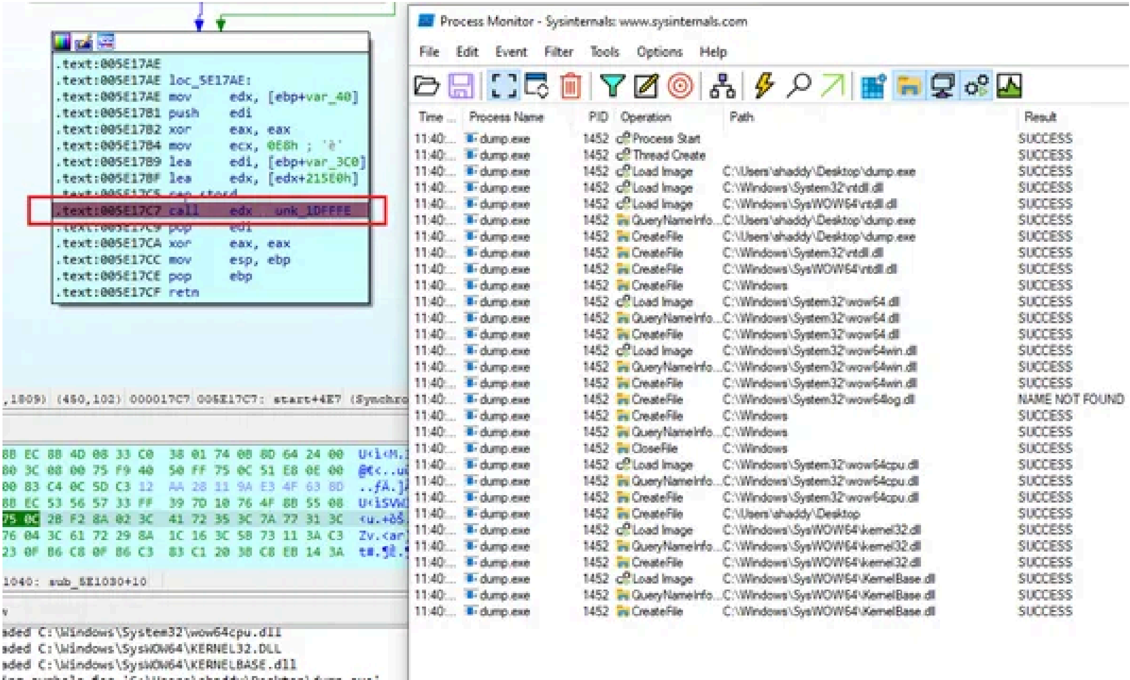
Stage2: Xloader 4.3

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



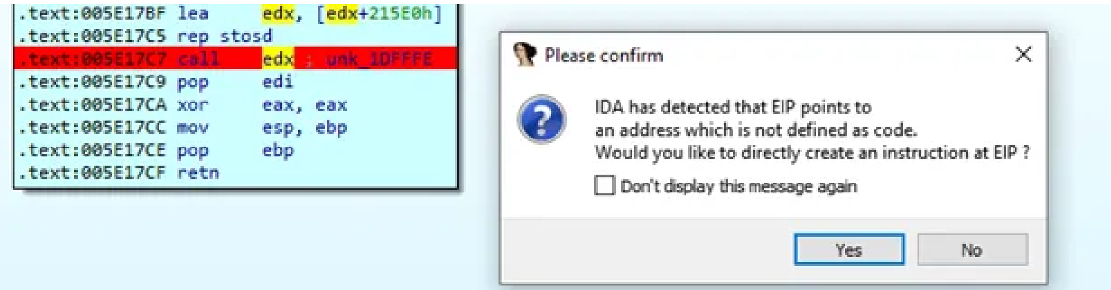
- The start of malware is fairly simple, it loads some necessary libraries before going to the malicious code
- It also performs some other kind of computations, probably decompressing some of its malicious code
- After the calculations, I came across a call to edx which leads to an unidentified code



- 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

```
.text:006025E4 dd 9F663DC1h, 3970D06Bh, 33036615h, 84F18000h, 7B992D01h, 0E8C3059Ah, 0
.text:006025E4 dd 8B55C358h, 0FD7B45ECh, 467B15C1h, 80B7D392h, 1D0312F4h, 825B1603h, 635ED9B8h
.text:006025E4 dd 9A0D850Bh, 0C2106547h, 0E8C3EAADh, 0, 8B55C358h, 9D2D19ECh, 7CB949C3h
.text:006025E4 dd 7C3569F8h
EDX .text:00602DE4 dd 0E1581FB0h, 7F000027h, 7F2D69ECh, 0F82FE3B6h, 300000D7h, 0EAB30805h
.text:00602DE4 dd 0DD25C101h, 0B946C102h, 0C6993523h, 33427E94h, 0AC03352Dh, 343DC001h
.text:00602DE4 dd 4DE518C3h, 47CA0D28h, 0C04135E5h, 0E50E105h, 350155CDh, 892A51D9h, 5E6625C1h
.text:00602DE4 dd 8122391Ch, 46F2D5F9h, 353D2393h, 8BBB7D55h, 6D61CE15h, 16158760h, 0AB9341D5h
.text:00602DE4 dd 1AC1EA81h, 3532F045h, 8C99FE2Ah, 51000DC0h, 0C19FFBFFh, 21A0FC1Dh, 19BEDFF5h
.text:00602DE4 dd 2163080Bh, 7C87353Dh, 953D2954h, 4A6A5A6Fh, 3EF32D0Bh, 2503AEE7h, 84D9948Dh
.text:00602DE4 dd 27F9DEA9h, 0E03DC093h, 3342B71Ch, 0DFC4FE81h, 0E8C3D86Ch, 0, 8B55C358h
.text:00602DE4 dd 0FAE84BECh, 81FFFFFFh, 9C74E8DBh, 25C19046h, 10D6CF96h, 0A0DC8040h
.text:00602DE4 dd 3B1A878Ah, 27051174h, 4A1CC476h, 13DA73A6h, 0A4E9732Dh, 1D1A524Bh, 58E0891Ch
.text:00602DE4 dd 446B2D69h, 0E77041C5h, 2D8A0000h, 9CF2C70Ah, 143C05C0h, 0B78618Dh, 4F50952Dh
.text:00602DE4 dd 61D8729h, 0E8EDA5D8h, 0FFFFFFA5h, 6175E815h, 3D094A33h, 79E11D66h, 0AA690569h
.text:00602DE4 dd 459F4154h, 1DC10000h, 2799628Ch, 0C635005Fh, 1C9905Fh, 7192EB25h, 2DC057E8h
.text:00602DE4 dd 0EAC9B3B0h, 0BC2D13DFh, 0E8285549h, 0FFFFFF69h, 1EE80D09h, 830F9379h
.text:00602DE4 dd 0FFFFFF5Dh, 17730587h, 350AD331h, 433B4722h, 80083588h, 2D1B6A5Eh, 5A45B1C8h
.text:00602DE4 dd 94C2F75Fh, 0B0679E15h, 2D87AEF2h, 340A8016h, 6E71528h, 3D1B45C7h, 3F6D906Dh
.text:00602DE4 dd 0FF23810Fh, 0FC81FFFFh, 0FE4E6671h, 621125C1h, 0C01EF6F2h, 8868F625h
.text:00602DE4 dd 3D194965h, 3DCA8925h, 0FEFF25C1h, 30224A23h, 14DCE605h, 80D1A25h, 0C1FC8B30h
.text:00602DE4 dd 46FC872Dh, 5A508FAAh, 0F9553735h, 0CD2D8741h, 0FB60436h, 0FFFEDC81h
.text:00602DE4 dd 7168FFh, 1DC11331h, 0F7C58F83h, 293511CDh, 29B79018h, 56D68D0Dh, 0FD1533B3h
.text:00602DE4 dd 4FA47F28h, 73E3050Ah, 15C14728h, 2CC18792h, 3D3DC111h, 3427939Bh, 0C1F83D23h

00022DE4 00602DE4: .text:00602DE4 (Synchronized with EIP)

Hex View-1
005E1000  55 8B EC 8B 4D 08 33 C0  38 01 74 0B 8D 64 24 00  U‹ì‹M.3À8.t..d$.
005E1010  40 80 3C 08 00 75 F9 40  50 FF 75 0C 51 E8 0E 00  @€<..uù@Pÿu.Qè..
005E1020  00 00 83 C4 0C 5D C3 12  AA 28 11 9A E3 4F 63 8D  ..ƒÄ.]Ã.ª(.šãOc.
005E1030  55 8B EC 53 56 57 33 FF  39 7D 10 76 4F 8B 55 08  U‹ìSVW3ÿ9}.vO‹U.
005E1040  8B 75 0C 2B F2 8A 02 3C  41 72 35 3C 7A 77 31 3C  ‹u.+òŠ.<Ar5<zw1<
005E1050  5A 76 04 3C 61 72 29 8A  1C 16 3C 5B 73 11 3A C3  Zv.<ar)Š..<[s.:Ã
005E1060  74 23 0F B6 C8 0F B6 C3  83 C1 20 3B C8 EB 14 3A  t#.¶È.¶ÃƒÁ ;Èë.:
```

- 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.

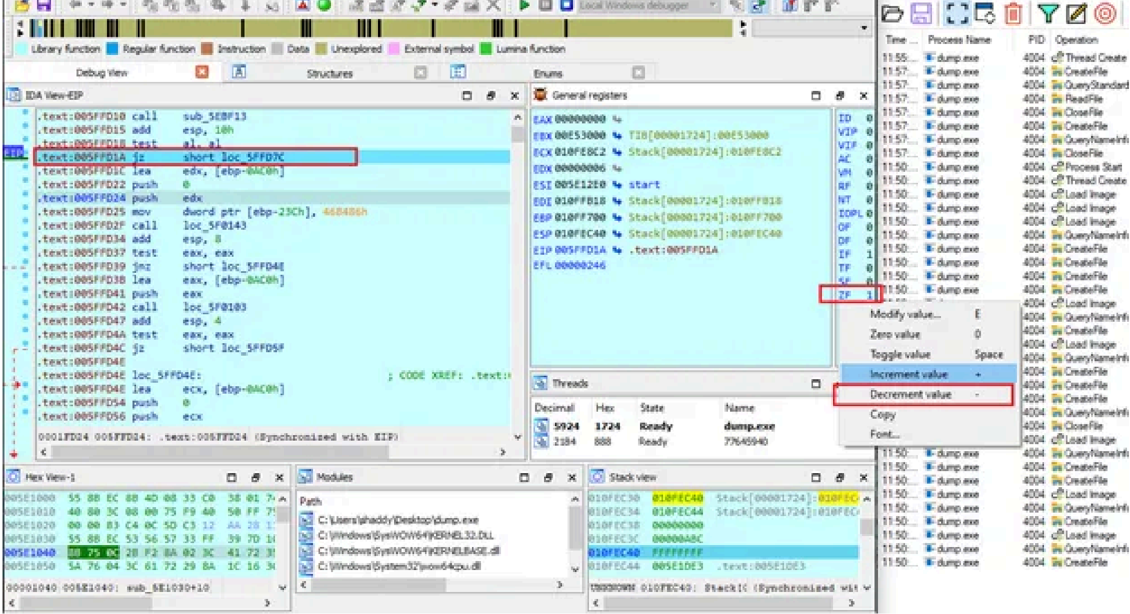


After going through the newly resolved chunk of code, my program exited without doing anything else. I understood that there are anti-analysis techniques involved in this malware. So, my battle started with defeating anti-analysis techniques provided in the section below.

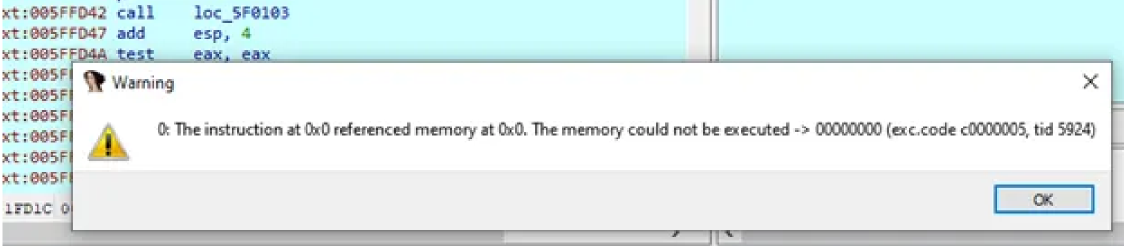
Defeating Anti-Analysis:

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



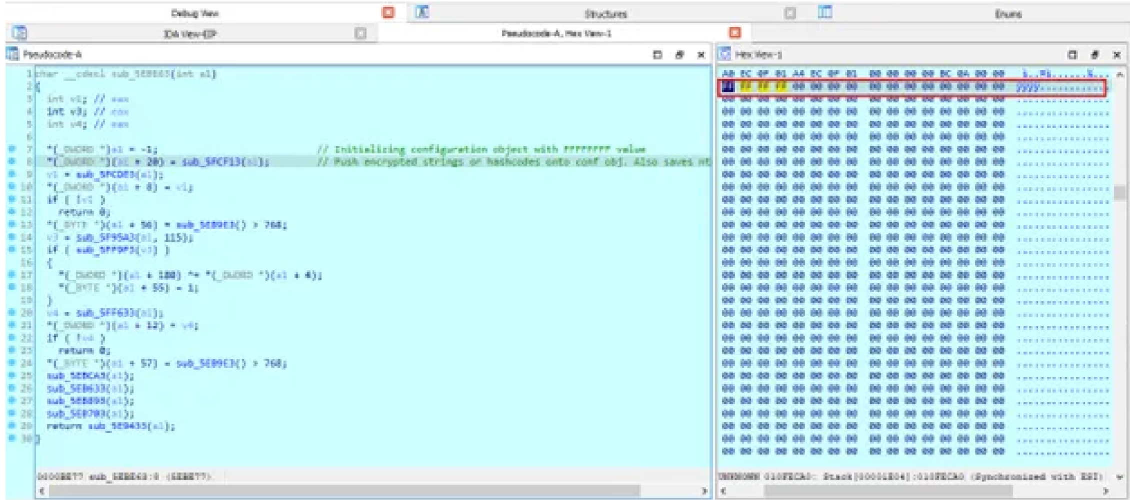
- It continues the program, however it throughs exception right after stepping over a few functions.
- This patch will not work
- The malware is dependent upon the values that this flag is setting somewhere



TAKE # 2: FAILED

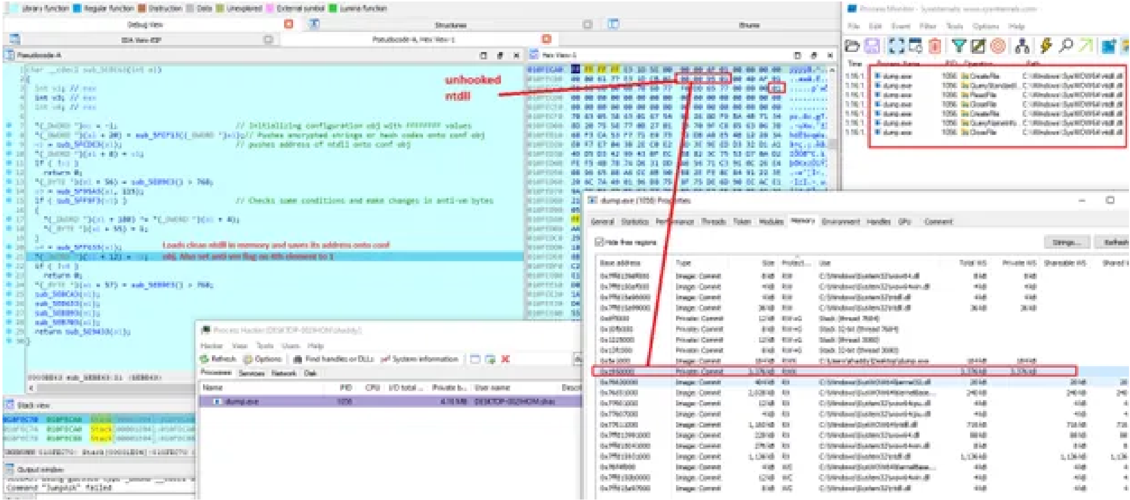
The configuration object:

- 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

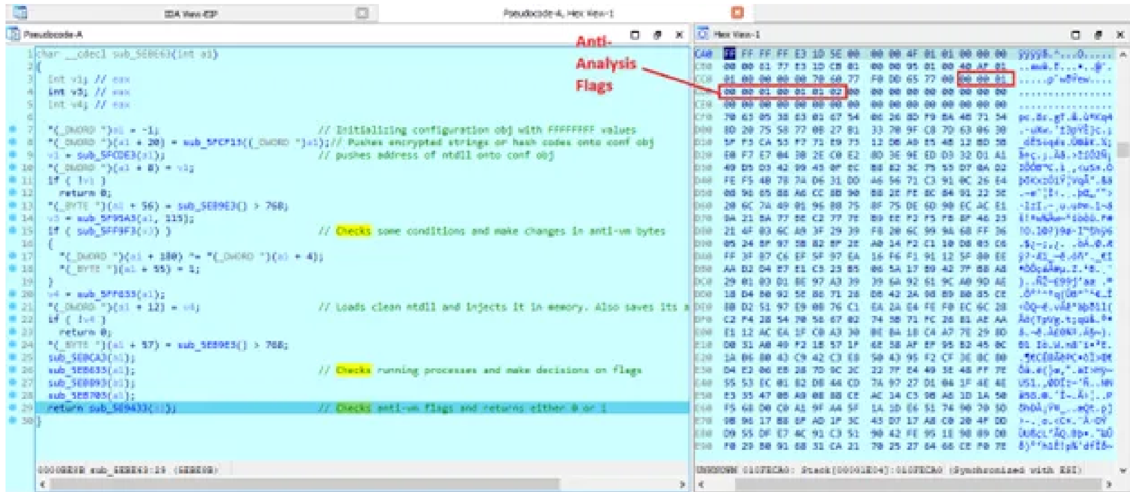
- I stepped over each function and monitored changes in memory side by side.
- Every function is contributing to the conf obj.
- The function in the screenshot below is loading a clean ntdll in the memory and saves it address on the conf obj
- Also, it is setting value in anti-vm flags that starts from 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.



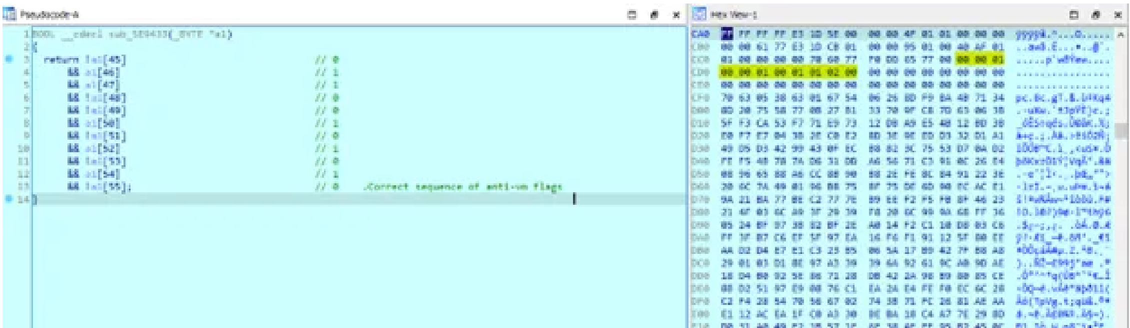
- Continuing with the execution.
- It checks other anti-vm checks
- Like taking snapshot of running processes and filtering out if any of those processes are listed by the malware
- In the screenshot, we can see that it detected **procmon** in running processes

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:



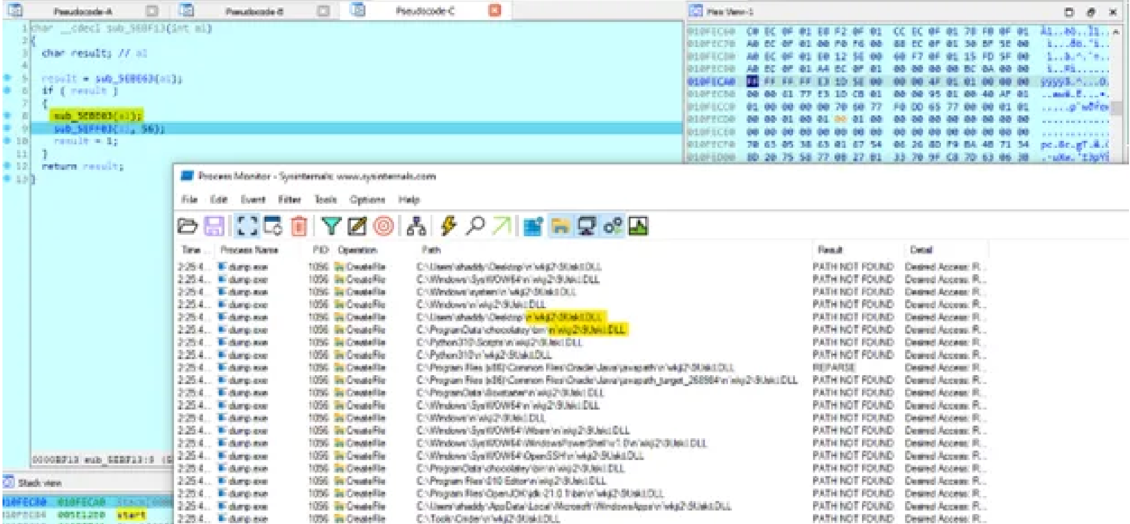
- The last function is matching the anti-vm flags with the sequence it requires to progress.
- As can be seen in the screenshot, my sequence doesn't match to what it should be,
- It means the malware has either **detected the debugger** or tools like **procmon** or some other parameter
- Therefore, the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



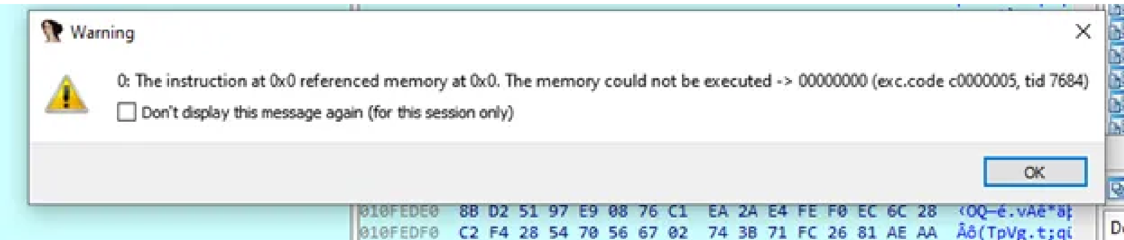
- So, in take # 2 of defeating anti-reverse engineering or anti-vm techniques, I simply patched the sequence of these flags in the memory to the required sequence.
- Patching memory, and moving onto the execution should work, because these flags are being used somewhere ahead in the program. So, simply 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.

```
CC0  01 00 00 00 00 70 60 77  F0 DD 65 77 00 00 01 01
CD0  00 00 01 00 01 00 01 00  00 00 00 00 00 00 00 00
```

- 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.

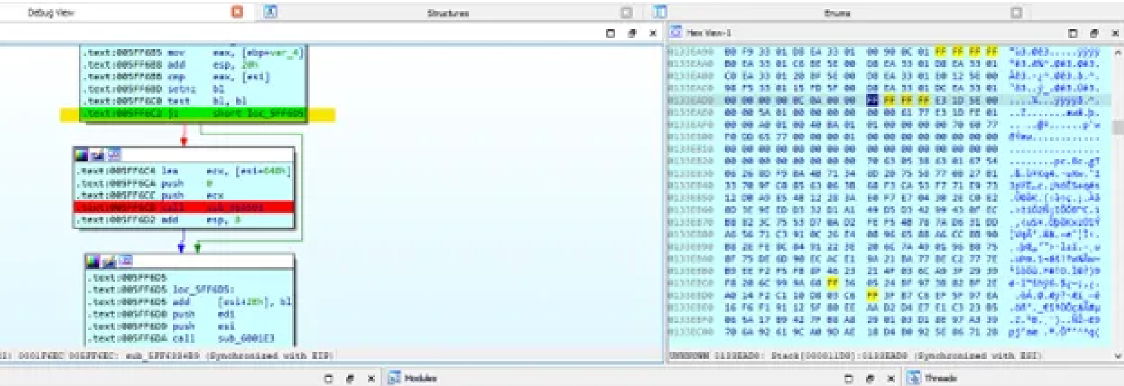


- I let the malware continue and again it crashed, because it was not able to decrypt its configuration and hence looking for encrypted dll names.
- So that means, I might be missing some important function and because it is detecting the debugger, it would be skipping some important function.

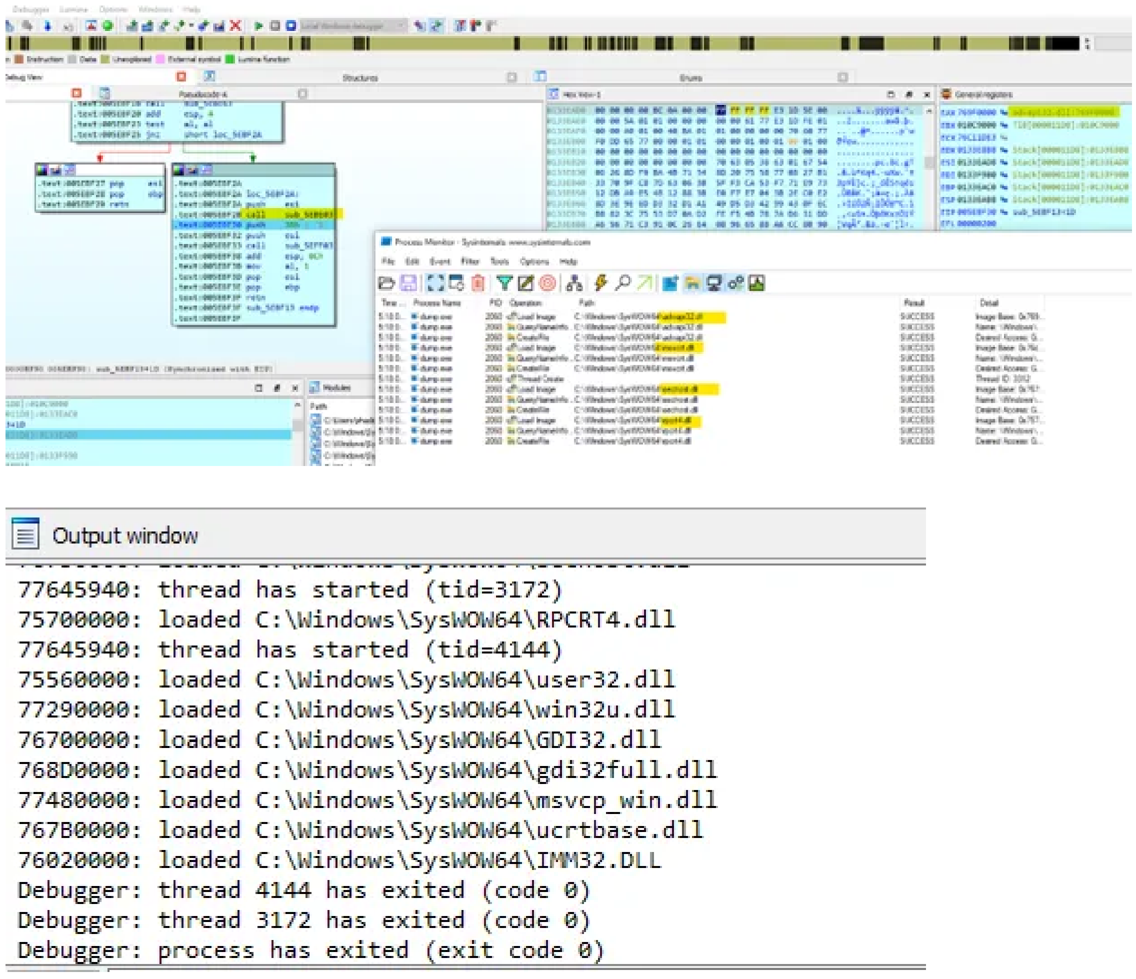


TAKE # 3: PASSED

- 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.

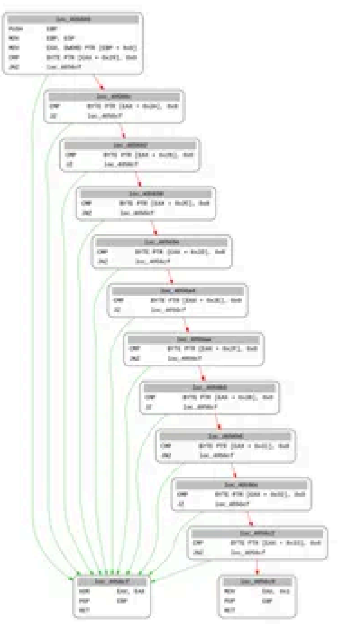


- Now this time, I stepped over the function that loads libraries and instead of encrypted names, the full names of libraries have been seen and successfully loaded as can be seen in procmon.
- I let the program continue without any other interaction and the debugger exited with status code 0, which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



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/

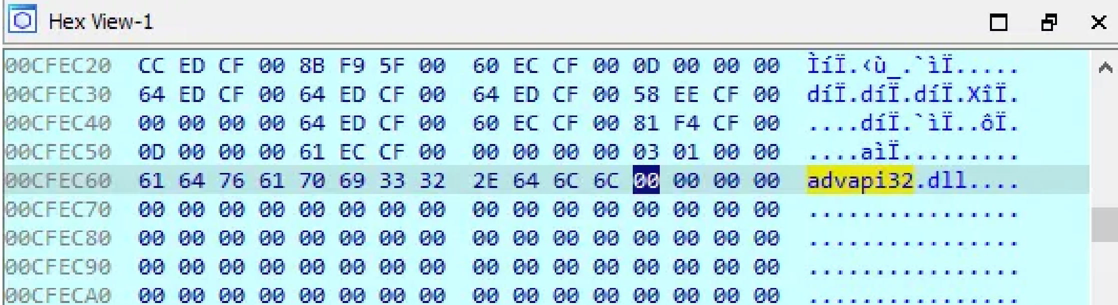
Decryption/Deobfuscation Routine:

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.

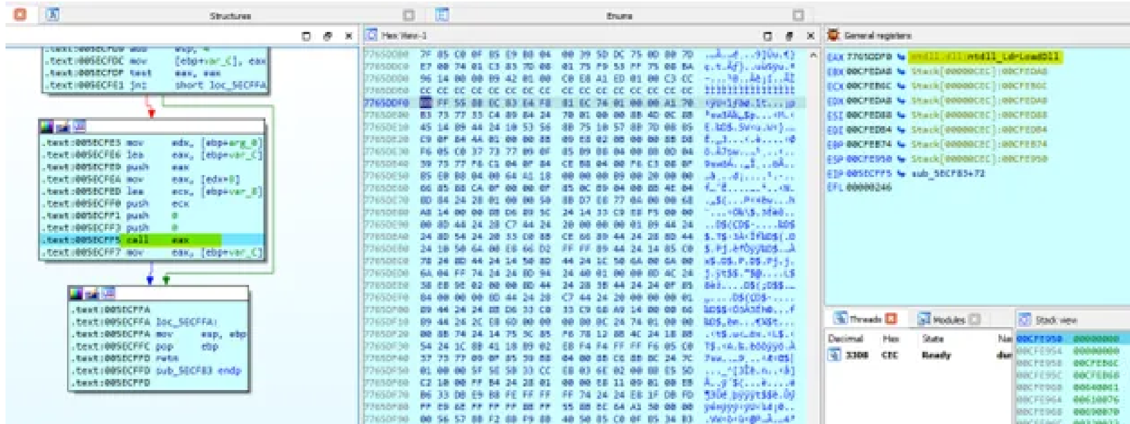
Decrypting Library Names:

- 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  í¤Wù$d~îvÅ´É¾.:÷
00CFEBB0  A5 13 49 6D 95 31 38 D5  86 5F E3 04 C8 E5 DF 58  ¥.Im•18Õ†_ã.ÈåßX
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  ................
```

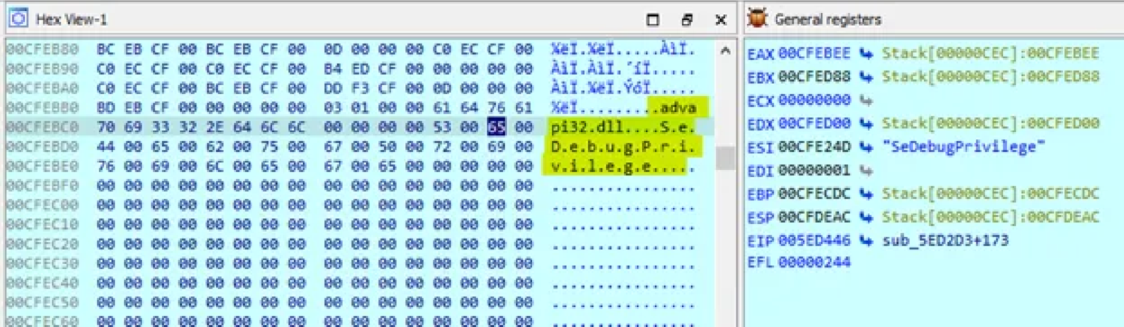


- These libraries are then loaded by the native function **LdrLoadDll**



Decrypting API Names:

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

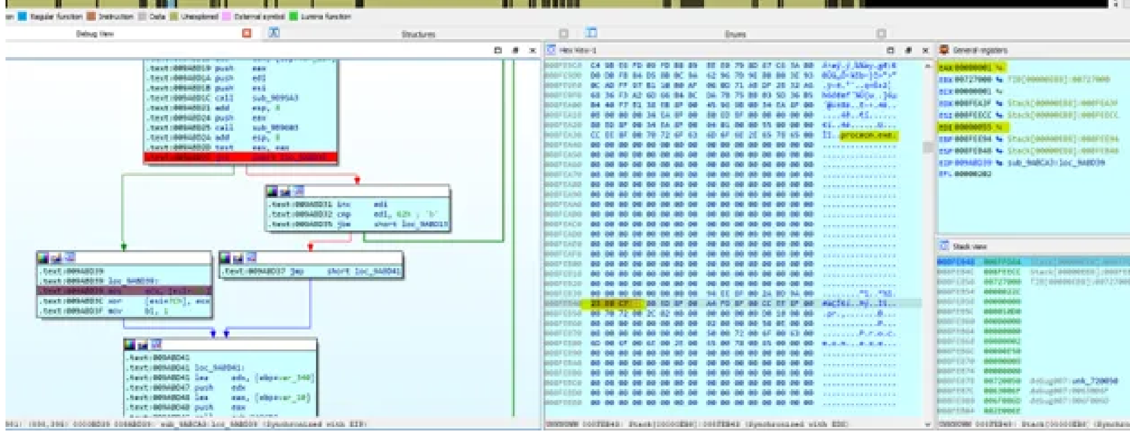


Press enter or click to view image in full size

```
General registers
EAX 76A0A230 ↳ advapi32.dll:advapi32_LookupPrivilegeValueW
EBX 008F0000 ↳ TIB[00000F7C]:008F0000
ECX 76C11DE3 ↳
EDX 00B21DE3 ↳ debug030:00B21DE3
ESI 0070F55C ↳ Stack[00000F7C]:0070F55C
EDI 0070F97C ↳ Stack[00000F7C]:0070F97C
EBP 0070EA04 ↳ Stack[00000F7C]:0070EA04
ESP 0070E9EC ↳ Stack[00000F7C]:0070E9EC
EIP 005FEAD2 ↳ sub_5FEAB3+1F
EFL 00000204
```

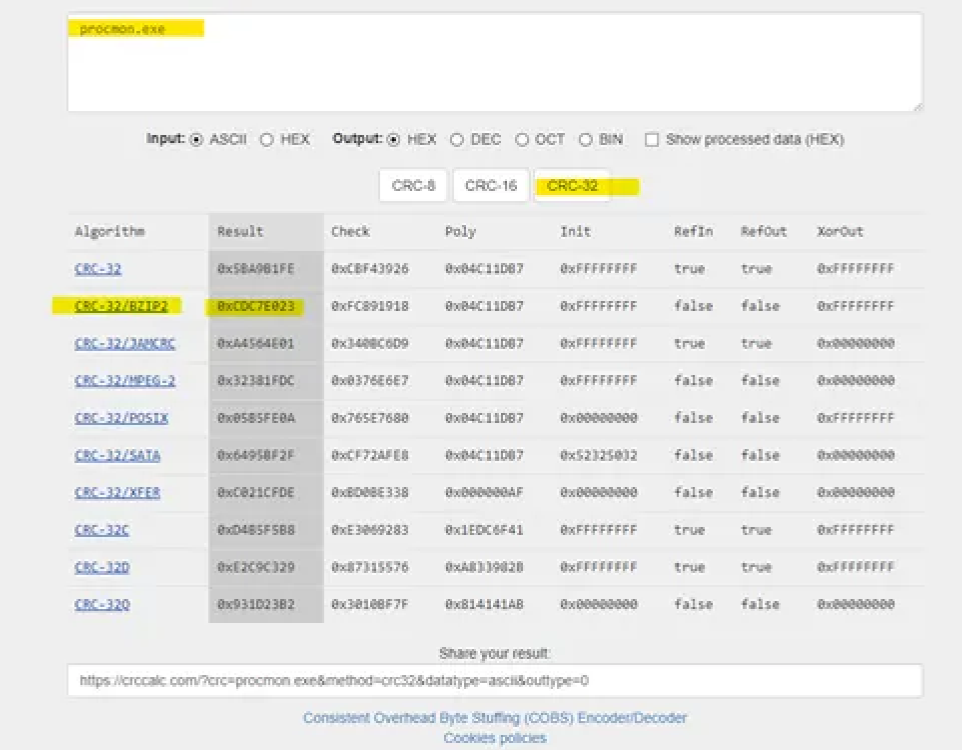
Computing String Hashes:

- 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



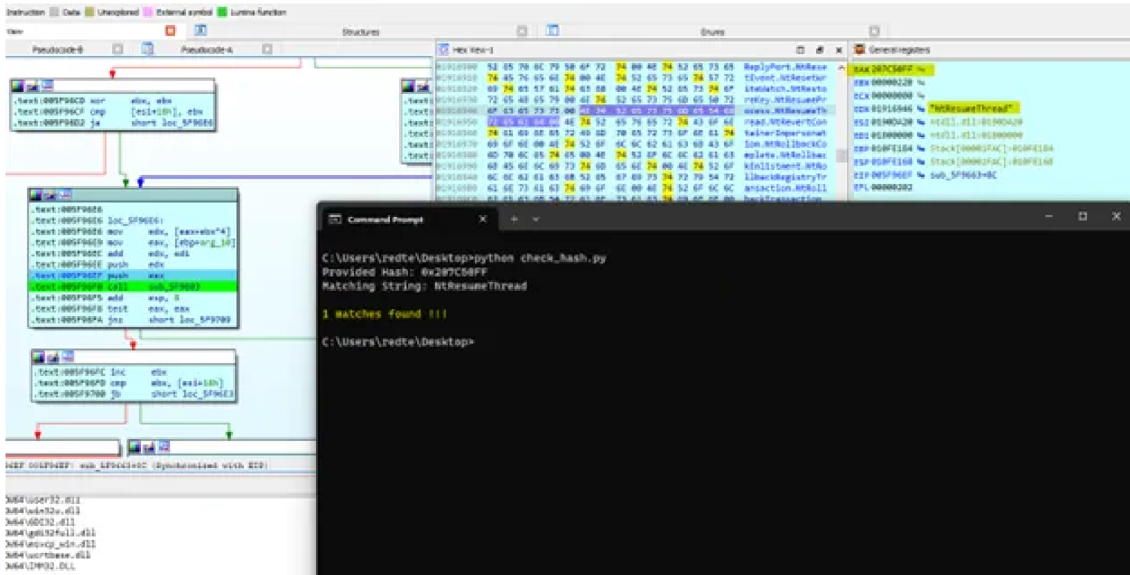
All the hashes that it checks are listed below:

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

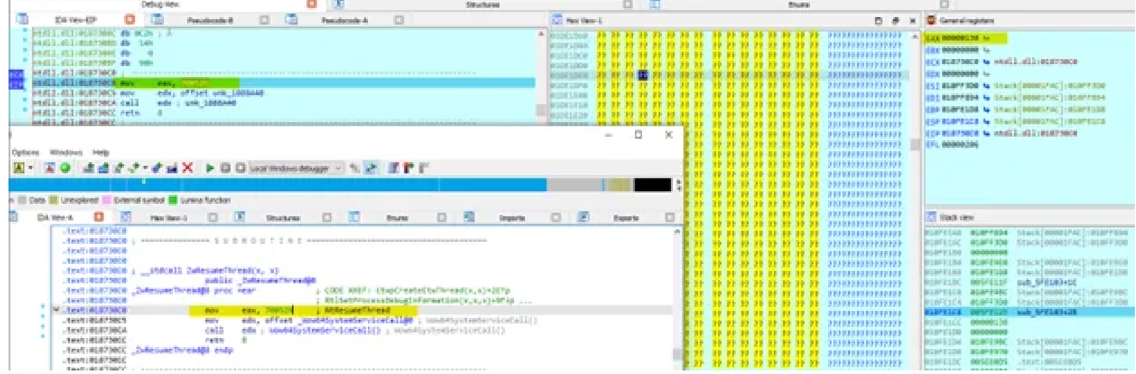
Computing API Hashes:

- Similar to strings hashes
- The APIs that are being loaded from injected **ntdll** (Lagos Island Method) 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.



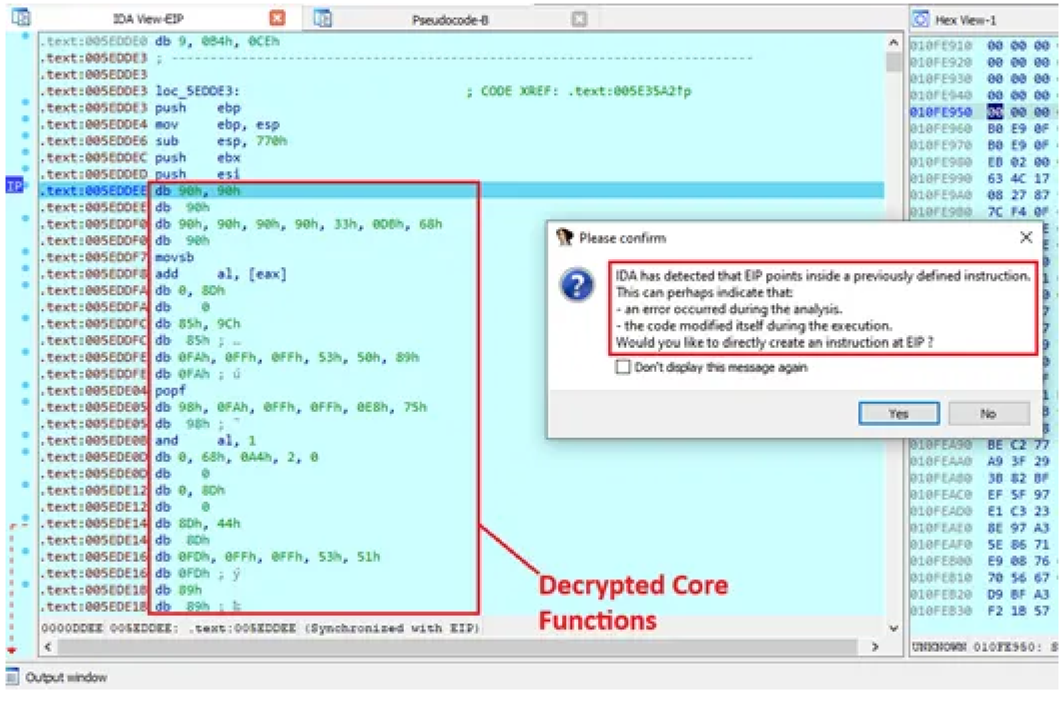
- I know the hashing function, so instead of stepping through this native assembly of hundreds of functions in a loop, I have just setup the breakpoint on that function by writing IDA python script and just continuing again and again to see the decrypted APIs
- The List of APIs that I found are listed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

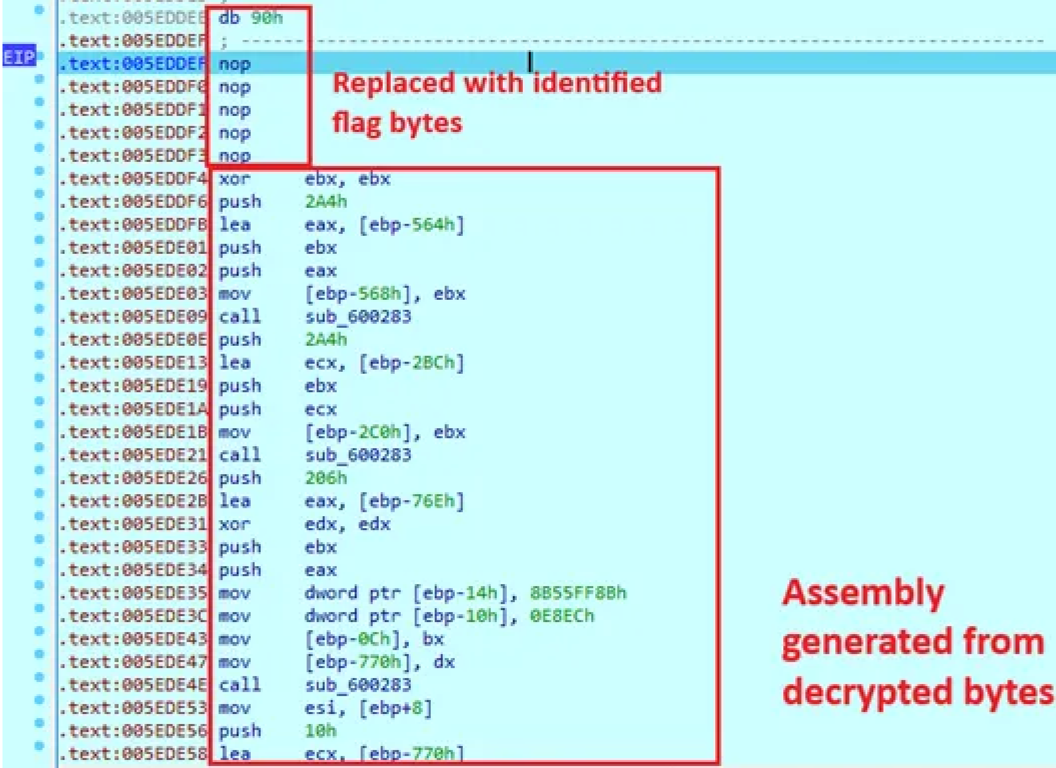
1	NtOpenDirectoryObject	
2	NtCreateMutant	
3	RtlSetEnvironmentVariable	
4	NtCreateSection	
5	NtMapViewOfSection	
6	NtOpenProcess	
7	RtlAllocHeap	
8	NtQueryInformationToken	
9	NtProtectVirtualMemory	
10	NtCreateFile	
11	NtDelayExecution	
12	NtReadVirtualMemory	
13	NtOpenThread	
14	NtReadFile	
15	NtUnmapViewOfSection	
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.

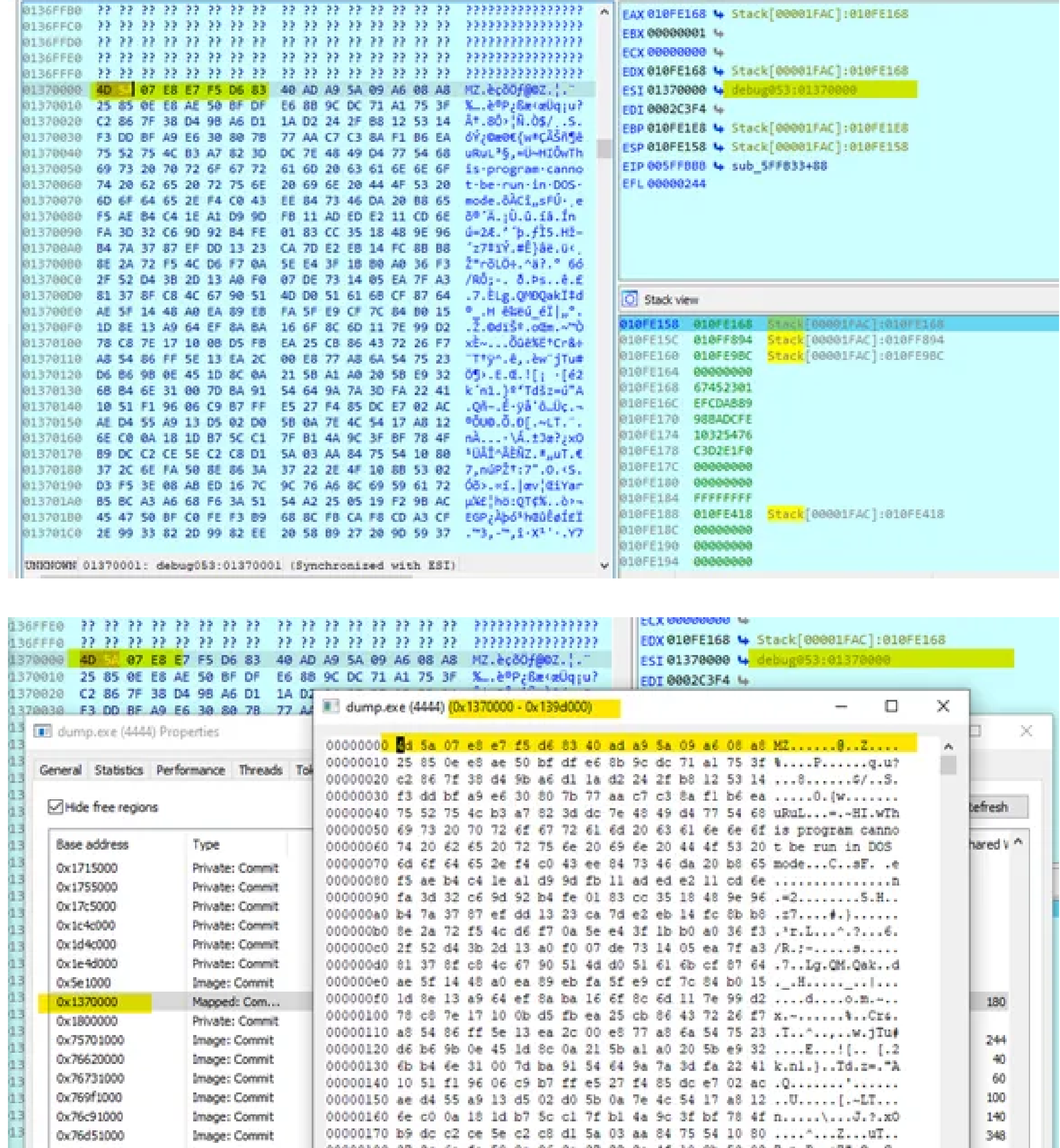




Understanding the detailed technical methodology of decrypting these encryption and obfuscation techniques. This following blog by **zscaler** is an excellent resource.

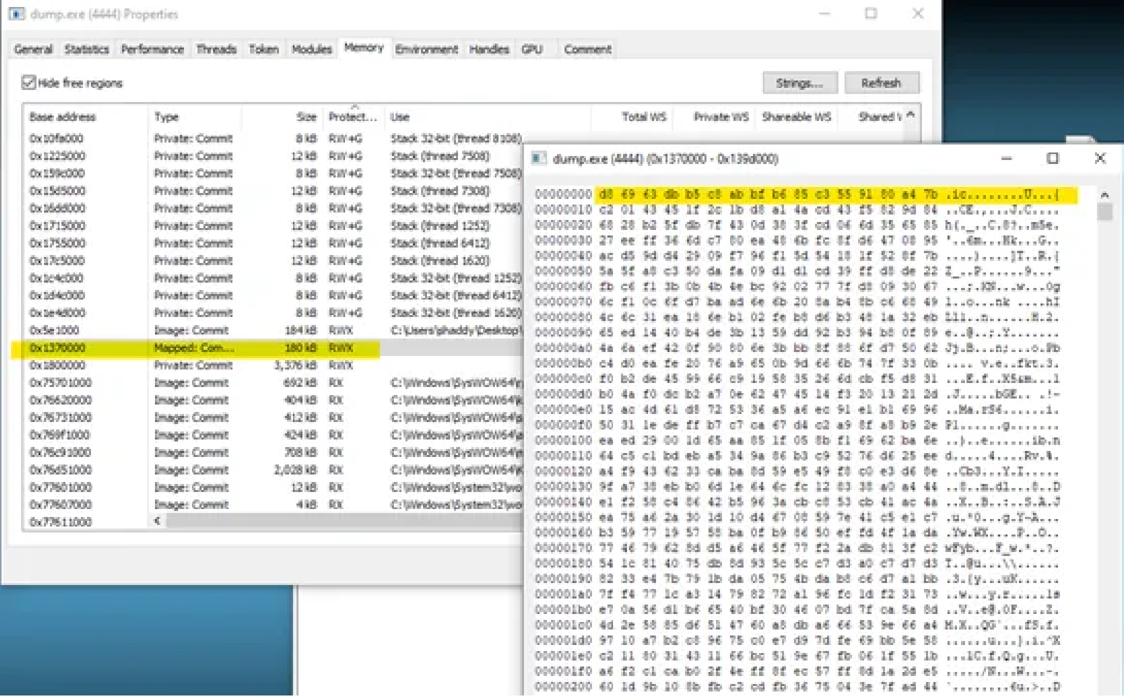
Partially Decrypted Shellcode:

- 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



```
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
```

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.........
```

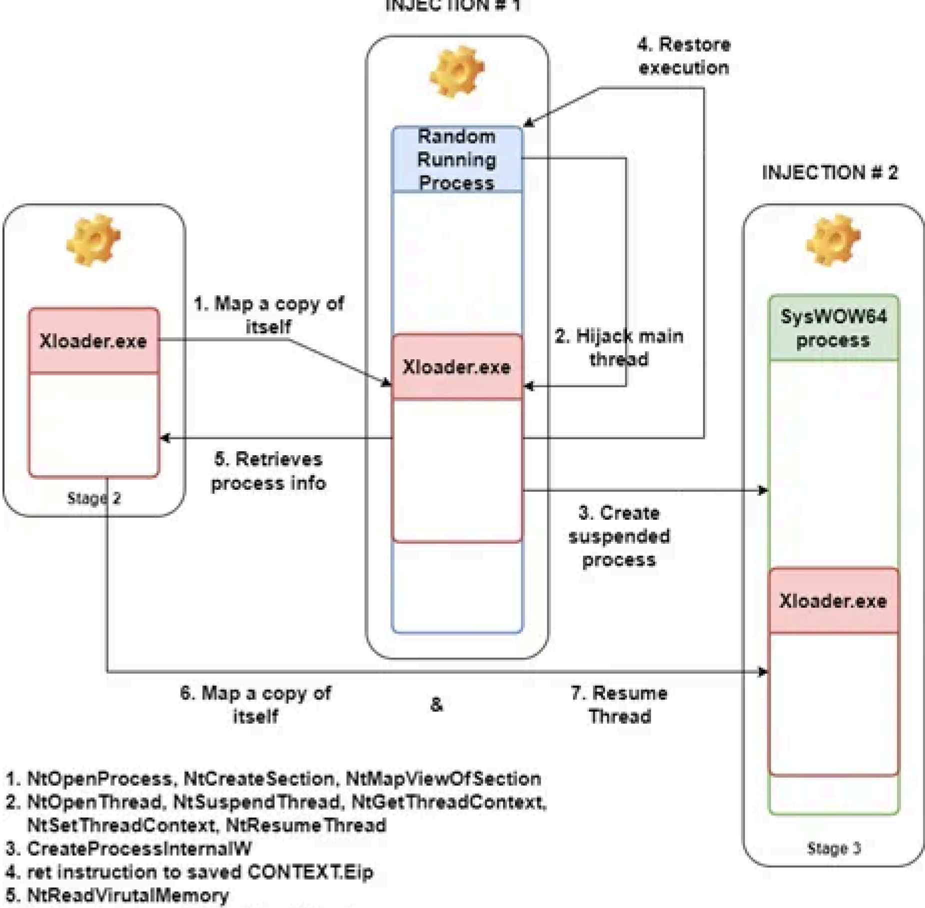
```
73 00 65 00 72 00 76 00  69 00 63 00 65 00 73 00  s.e.r.v.i.c.e.s.
2E 00 65 00 78 00 65 00  00 00 00 00 00 00 00 00  ..e.x.e.........
```

Process Injection:

Xloader Injection Overview:

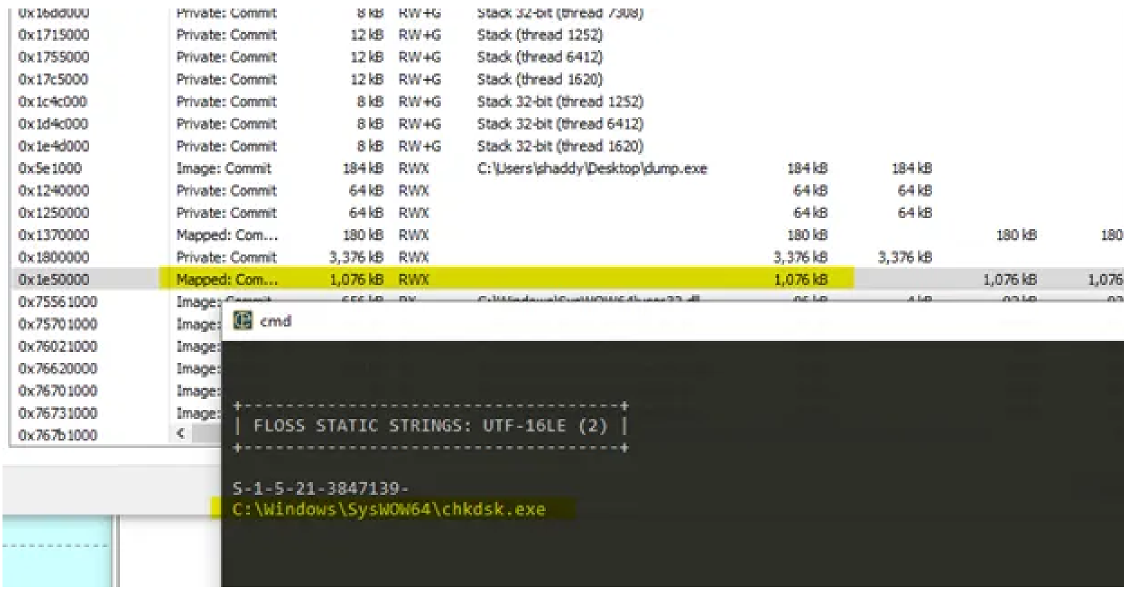
Xloader stage2 performs two process injections:

- **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

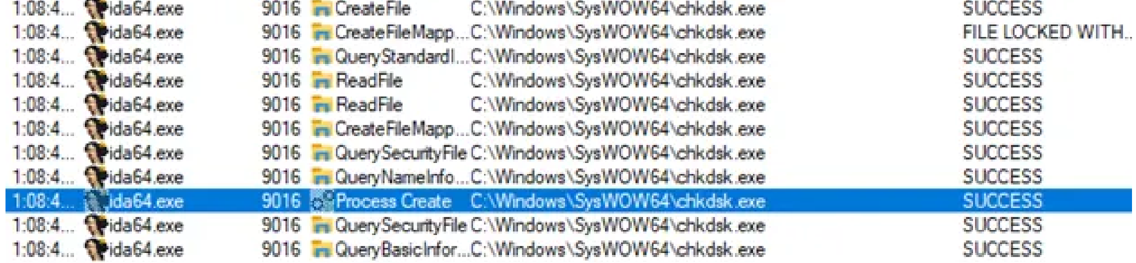


Injection # 1

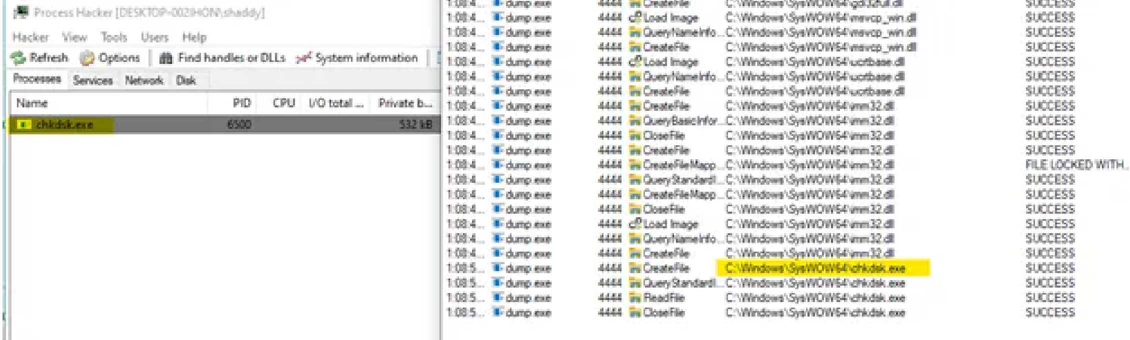
- Another memory has been reserved in the malware with RWX memory region.
- I have dumped this new region and extracted the strings
- It has a single static string which contains the name of the target process



- 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.

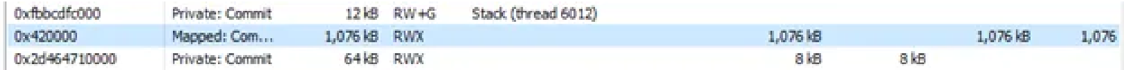


- This is also one of the anti-analysis techniques used by xloader. It doesn't directly open the process itself but injects shellcode in some random process which in turn opens the SysWOW64 randomized binary in a suspended state and then retrieves its process information and continue with the execution.



- 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.

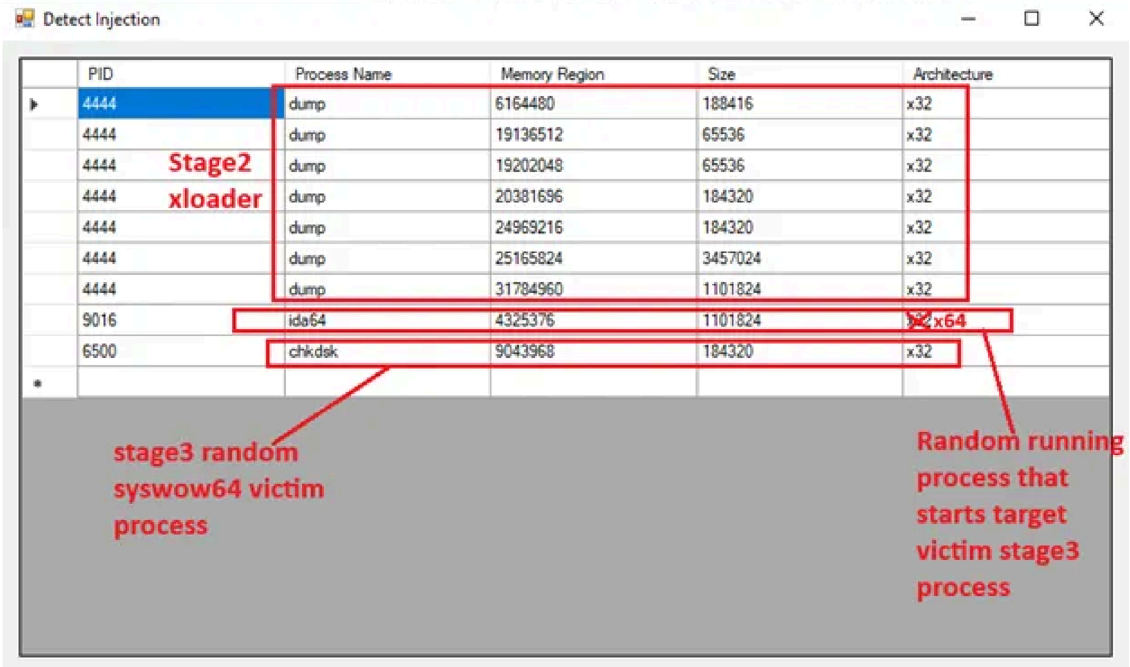
- This is just a **dead code** after opening the target process in suspended state.



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.

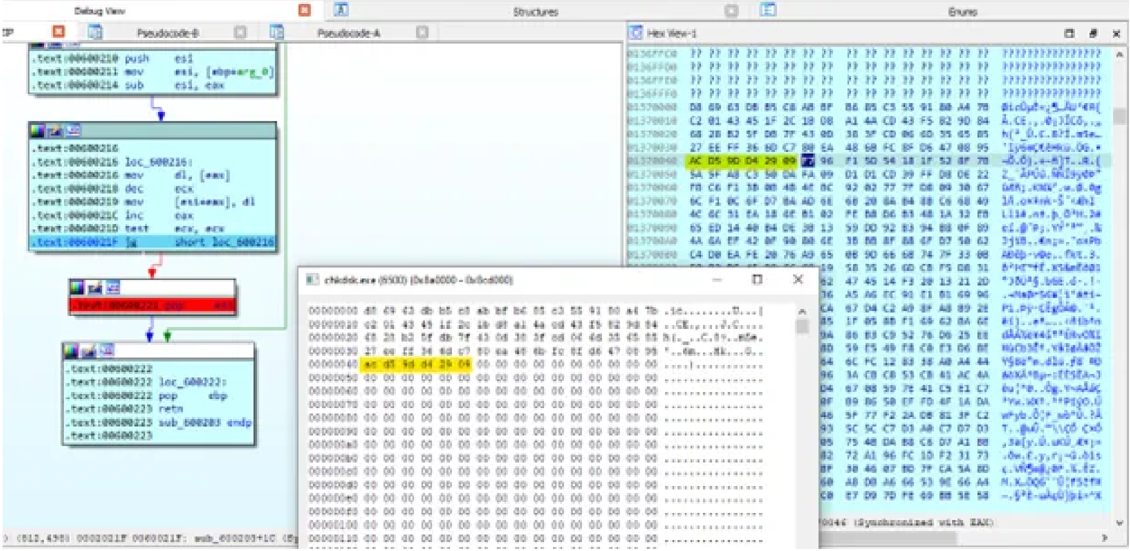


Detect Injection

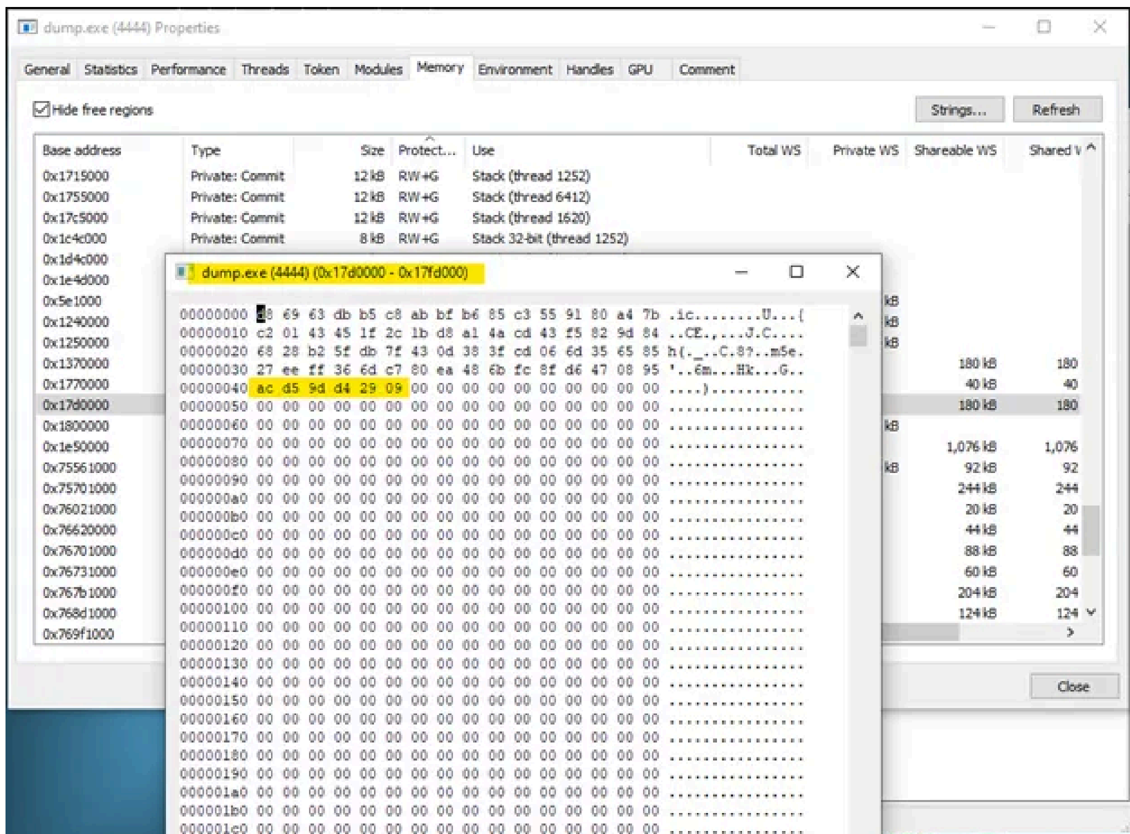
PID	Process Name	Memory Region	Size	Architecture
4444	dump	6164480	188416	x32
4444	dump	19136512	65536	x32
4444	dump	19202048	65536	x32
4444	dump	20381696	184320	x32
4444	dump	24969216	184320	x32
4444	dump	25165824	3457024	x32
4444	dump	31784960	1101824	x32
9016	ida64	4325376	1101824	x64
6500	chkdsk	9043968	184320	x32

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

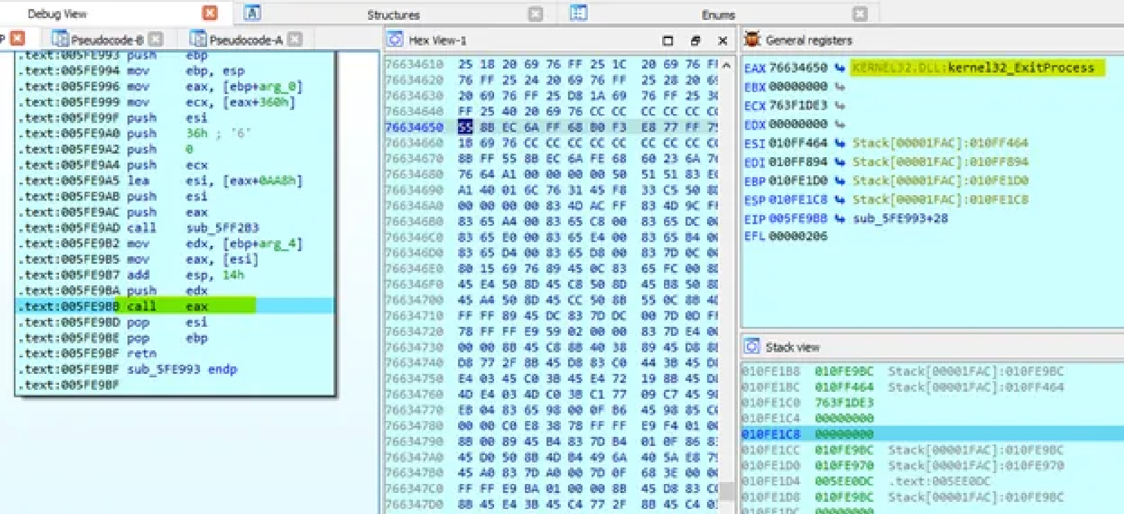
- The smaller buffer contains the original chkdsk.exe bytes.
- I also found the function that writes shellcode in the **180KB** empty buffer.
- This is also a shared memory region between the formbook payload and victim process of chkdsk.exe
- Because the buffer is simultaneously being written in both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

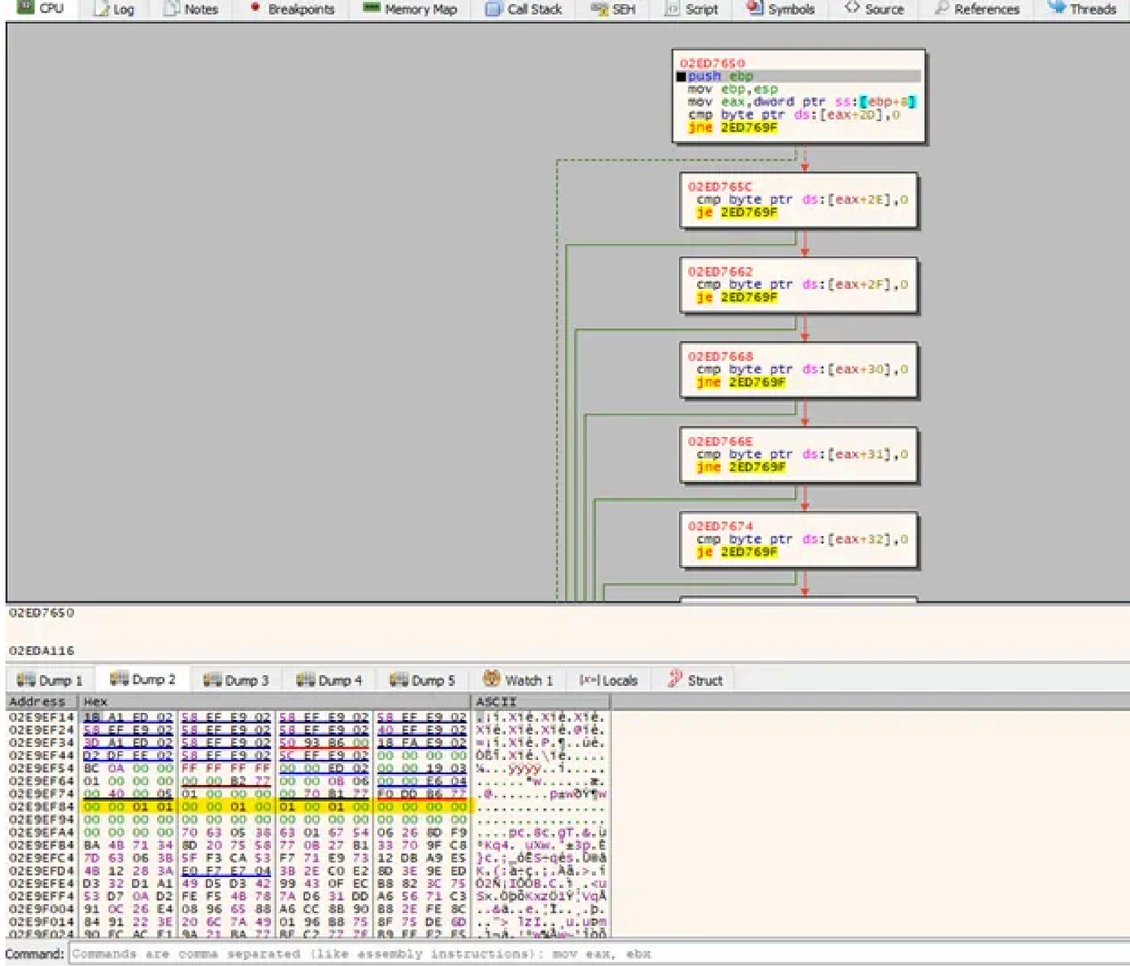


Stage 3: Partially Decrypted Xloader 4.3

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:

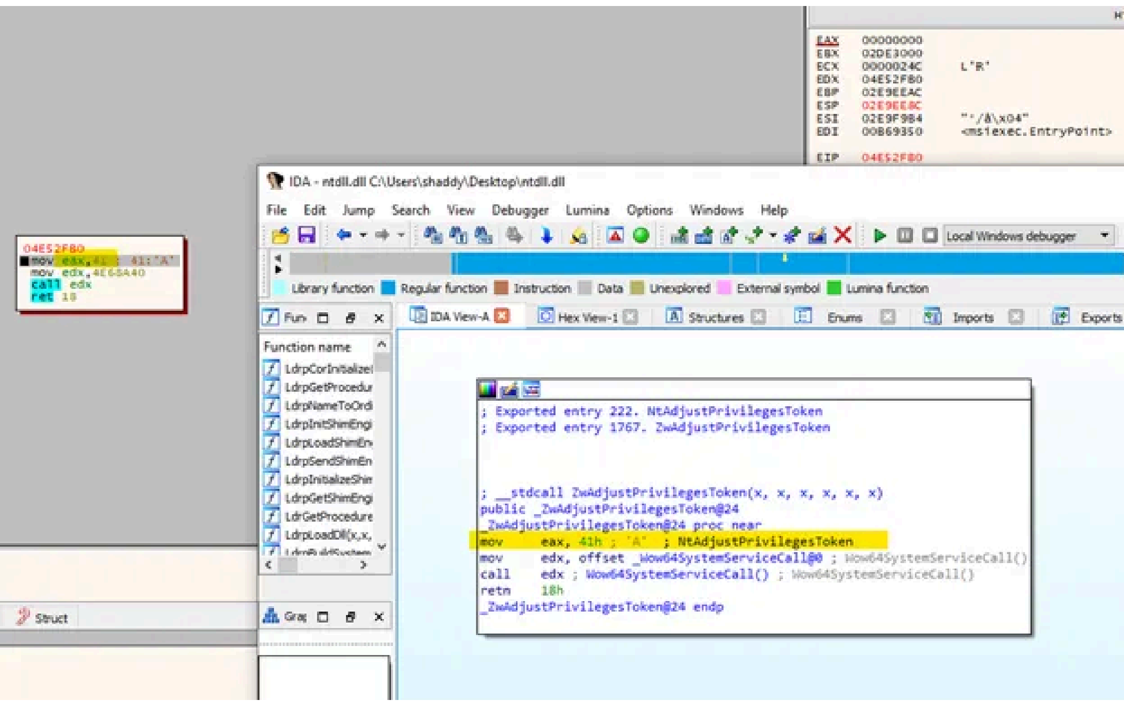
Defeating Anti-Analysis:

- 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



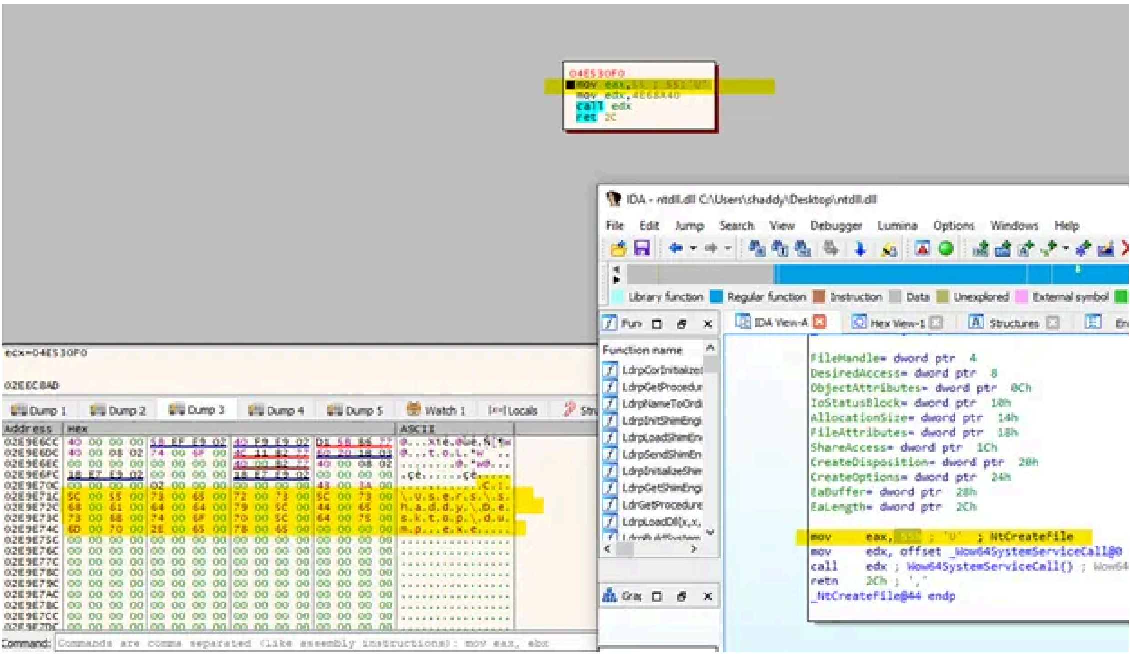
Decryption/Deobfuscation:

- 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:
- 1. LookupPrivilegeValue
- 2. SeDebugPrivilege
- 3. NtAdjustPrivilegeToken



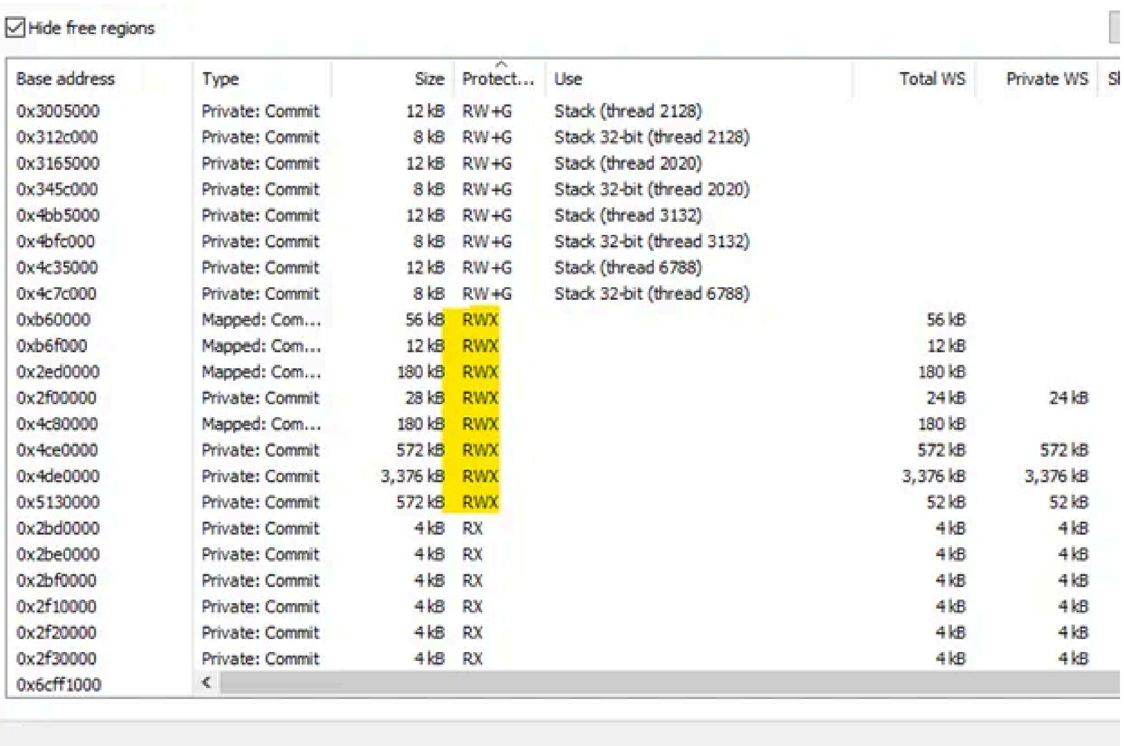
Indicator Removal:

- It will delete the stage2 malware with following sequence of APIs
- 1. NtCreateFile
- 2. NtQueryInformationFile
- 3. NtReadFile
- 4. NtClose
- 5. ZwDeleteFile



Process Injection:

- The next series of APIs being used are:
- 1. NtCreateSection
- 2. NtMapViewOfSection
- 3. NtAllocateVirtualMemory
- 4. NtOpenProcessToken
- 5. NtQueryInformationToken
- 6. ConvertSidToStringW
- 7. 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

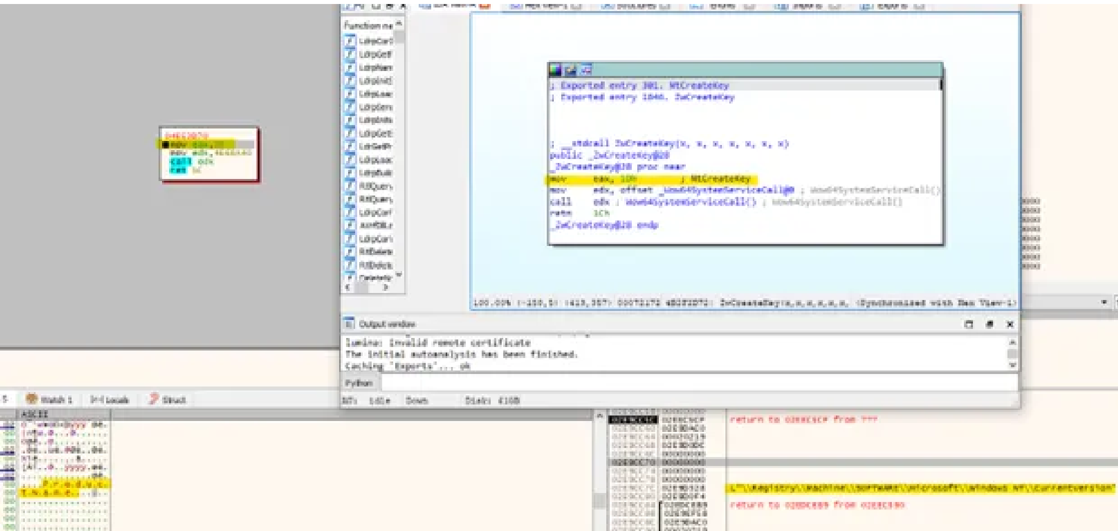


Hide free regions

Base address	Type	Size	Protect...	Use	Total WS	Private WS
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						

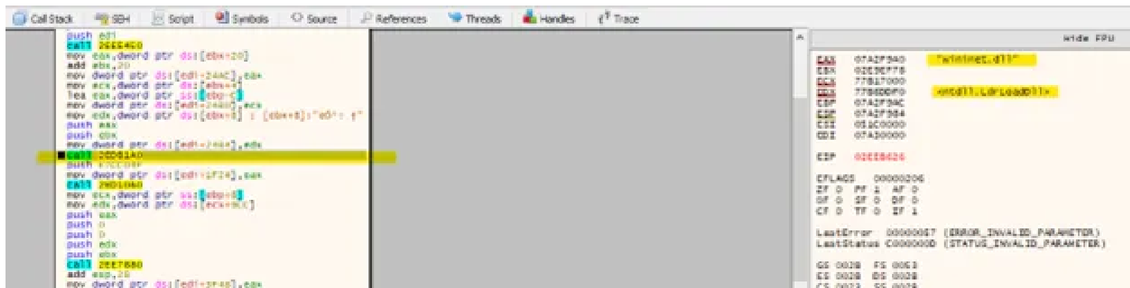
System Information Discovery:

- It retrieves the system information from the Registry like the **“Product Name”, “CurrentBuild”** of OS etc
- 1. NtCreateKey
- 2. NtQueryValueKey



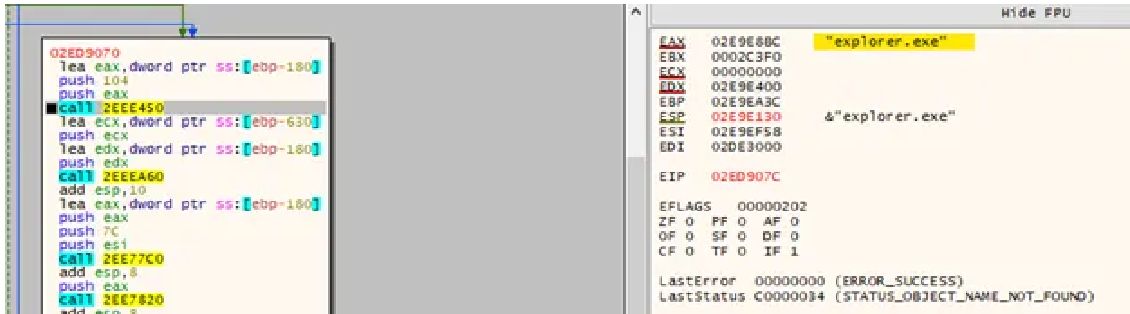
Dynamic Library/API resolution:

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



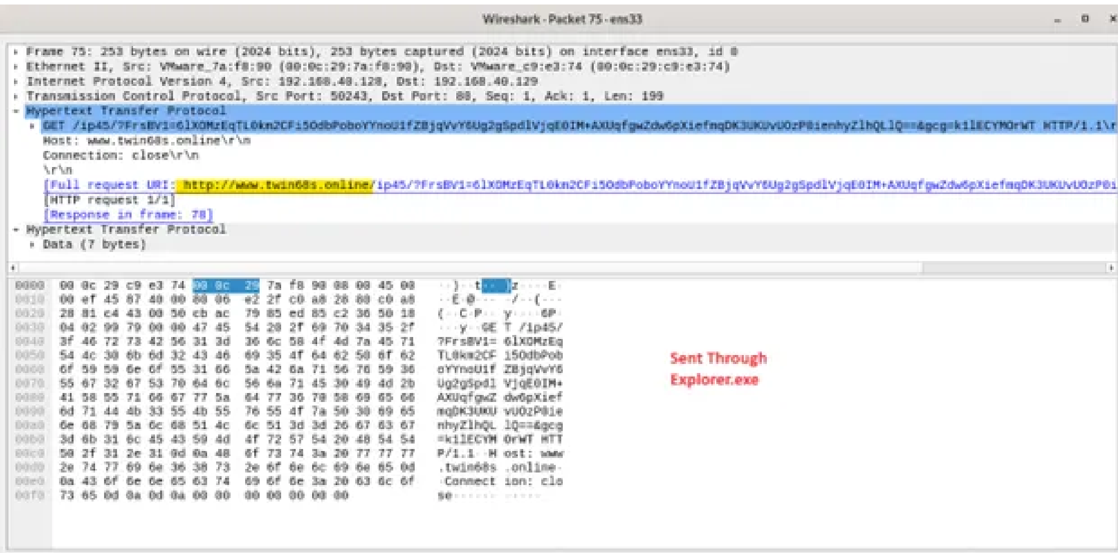
Process Enumeration & Injection:

- 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"**
- 2. NtCreateMutant
- 3. NtCreateSection
- 4. NtMapViewOfSection
- 5. NtDelayExecution
- 6. NtAllocateVirtualMemory



Press enter or click to view image in full size

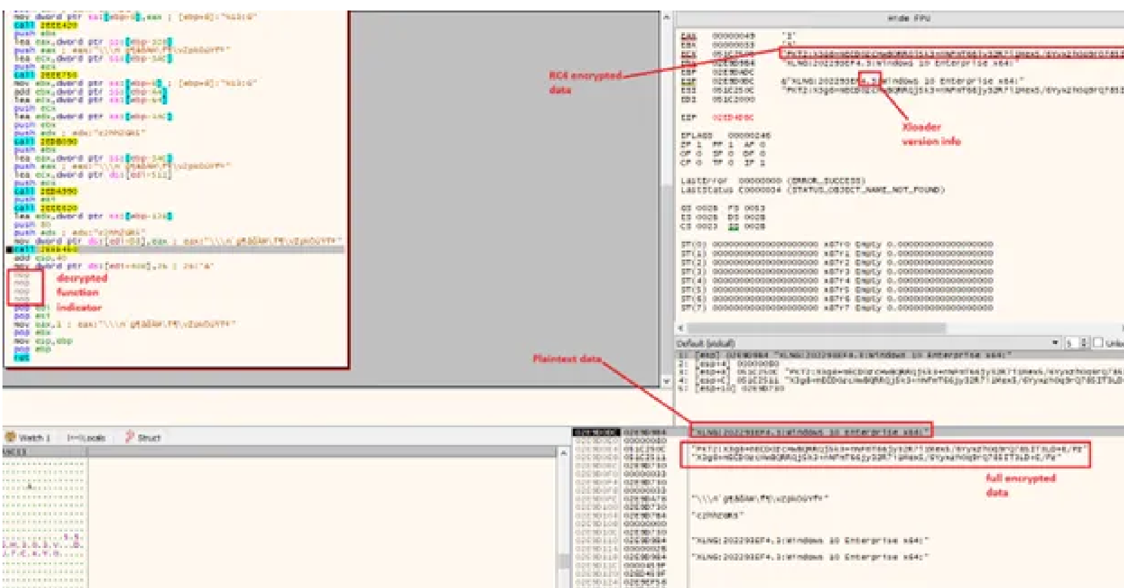
	PID	Process Name	Memory Region	Size
▶	3528	explorer	180617216	1060864
	748	x32dbg	100466688	65536



Bot registration:

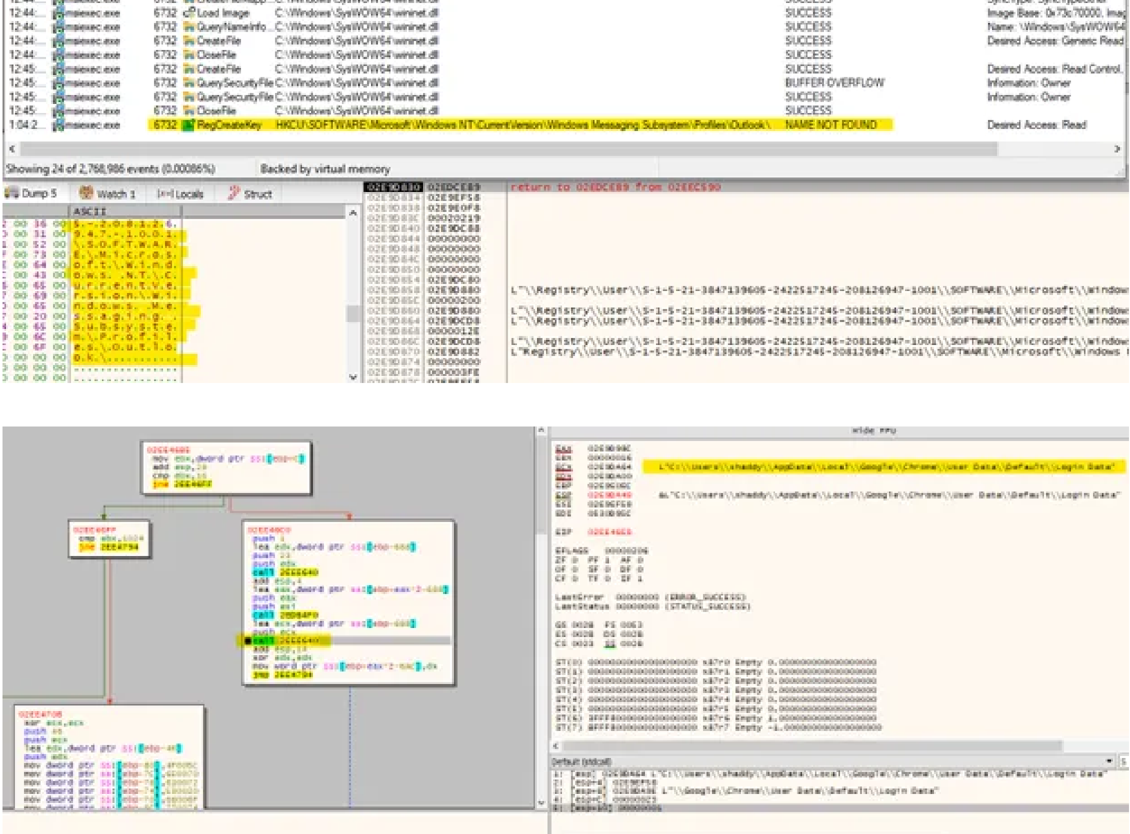
- The data it collects and sends in the first request is provided below:
- The Magic word: XLNG
- Bot ID: 202293EF
- Xloader Version: 4.3
- OS: Windows 10 Enterprise x64
- Username: base64_encoded

Press enter or click to view image in full size



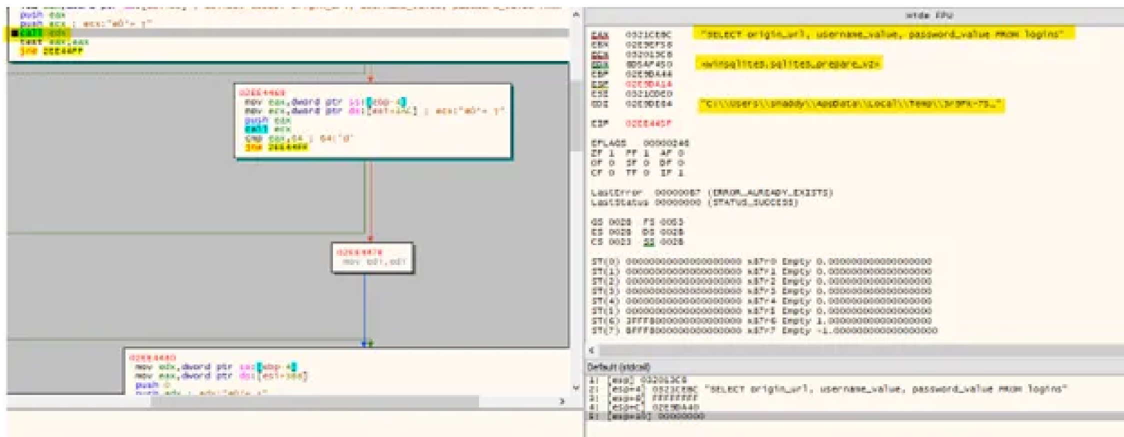
Stealer:

- 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

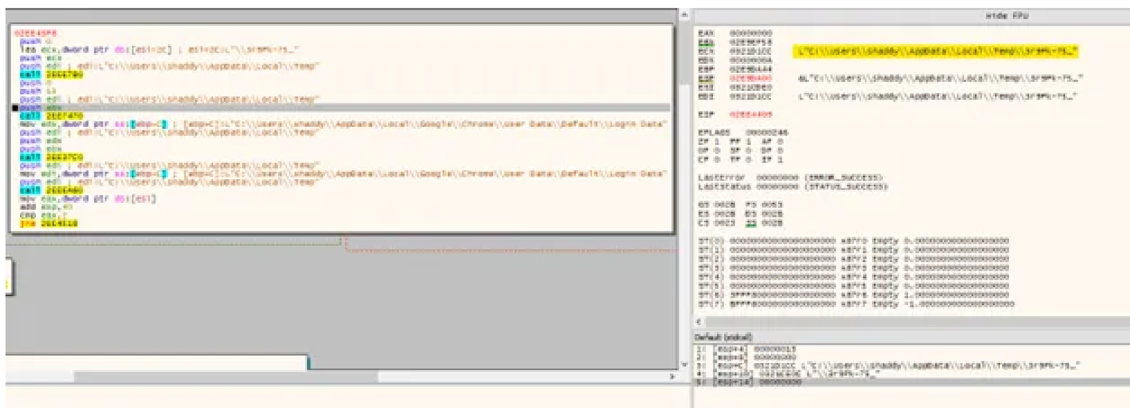


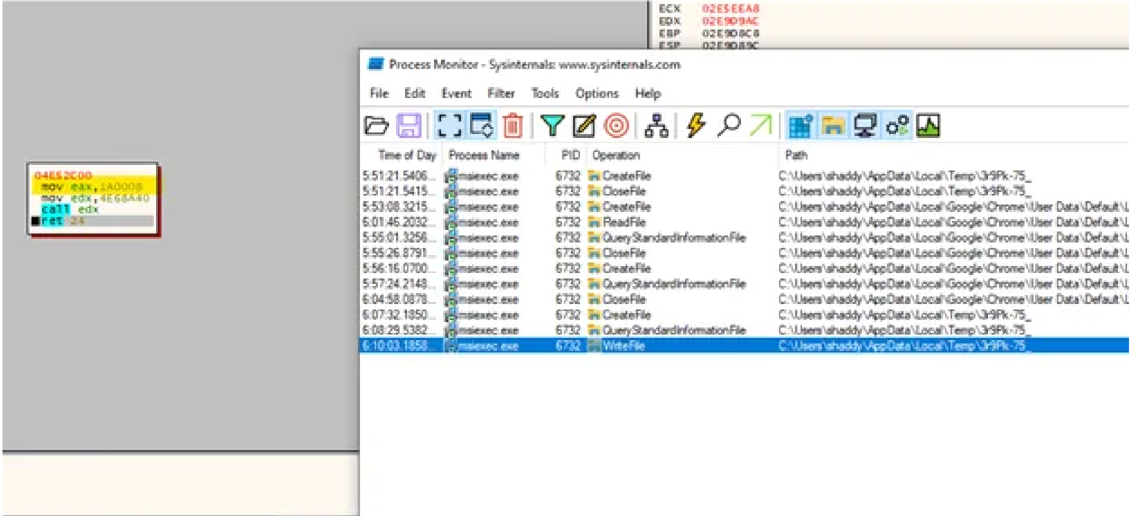

```
EAX   6D5A1060   <winsqlite3.sqlite3_step>
EBX   6D5A0000   "MZ"
ECX   6D63C578   winsqlite3.6D63C578
EDX   000000BD   "½"
EBP   02E9DA18
ESP   02E9D988
ESI   0321CDE0
EDI   02E9EF78

EIP   02EEADDE

EFLAGS   00000283
ZF 0  PF 0  AF 0
OF 0  SF 1  DF 0
CF 1  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) 0000000000000000000 x87r0 Empty 0.000000000000000000
ST(1) 0000000000000000000 x87r1 Empty 0.000000000000000000
ST(2) 0000000000000000000 x87r2 Empty 0.000000000000000000
ST(3) 0000000000000000000 x87r3 Empty 0.000000000000000000
ST(4) 0000000000000000000 x87r4 Empty 0.000000000000000000
ST(5) 0000000000000000000 x87r5 Empty 0.000000000000000000
ST(6) 3FFF8000000000000000 x87r6 Empty 1.000000000000000000
ST(7) BFFF8000000000000000 x87r7 Empty -1.000000000000000000

Default (stdcall)
1: [esp] 4AB491F7
2: [esp+4] 02E9EF78
3: [esp+8] 6D63C148 "YC\t"
4: [esp+C] 00000000
5: [esp+10] 00000000
```

```
EAX   7761A8B0   <crypt32.CryptUnprotectData>
EBX   02E9EF58
ECX   776B93E0   crypt32.776B93E0
EDX   000000FA   'ú'
EBP   02E9DA18
ESP   02E9D9C8
ESI   0321CDE0
EDI   02E9EF78

EIP   02EEAE8B

EFLAGS   00000212
ZF 0  PF 0  AF 1
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) 0000000000000000000 x87r0 Empty 0.000000000000000000
ST(1) 0000000000000000000 x87r1 Empty 0.000000000000000000
ST(2) 0000000000000000000 x87r2 Empty 0.000000000000000000
ST(3) 0000000000000000000 x87r3 Empty 0.000000000000000000
ST(4) 0000000000000000000 x87r4 Empty 0.000000000000000000
ST(5) 0000000000000000000 x87r5 Empty 0.000000000000000000
ST(6) 3FFF8000000000000000 x87r6 Empty 1.000000000000000000
ST(7) BFFF8000000000000000 x87r7 Empty -1.000000000000000000
```

- 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
- 1. NtCreateFile

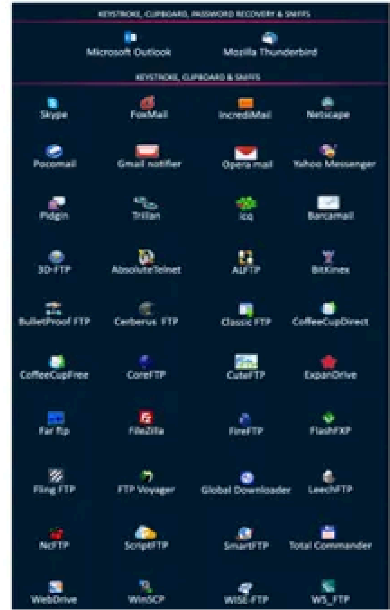
- 2. NtQueryInformationFile
- 3. NtReadFile
- 4. NtWriteFile





Web-Browsers

Mail clients, FTP clients, IM apps

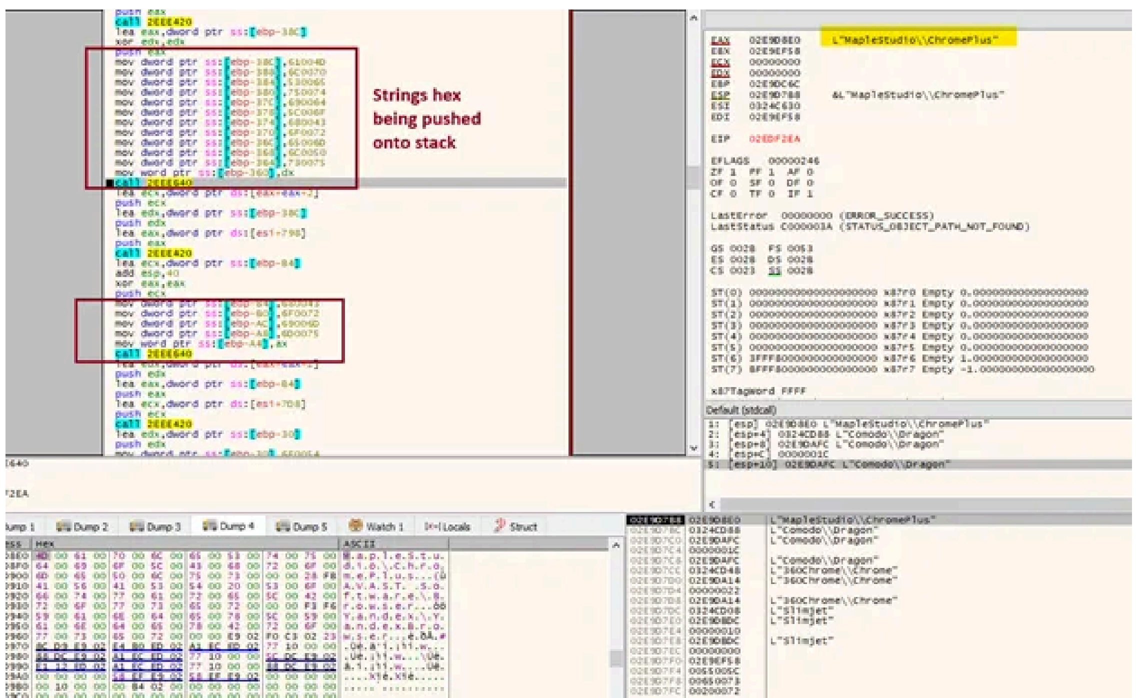


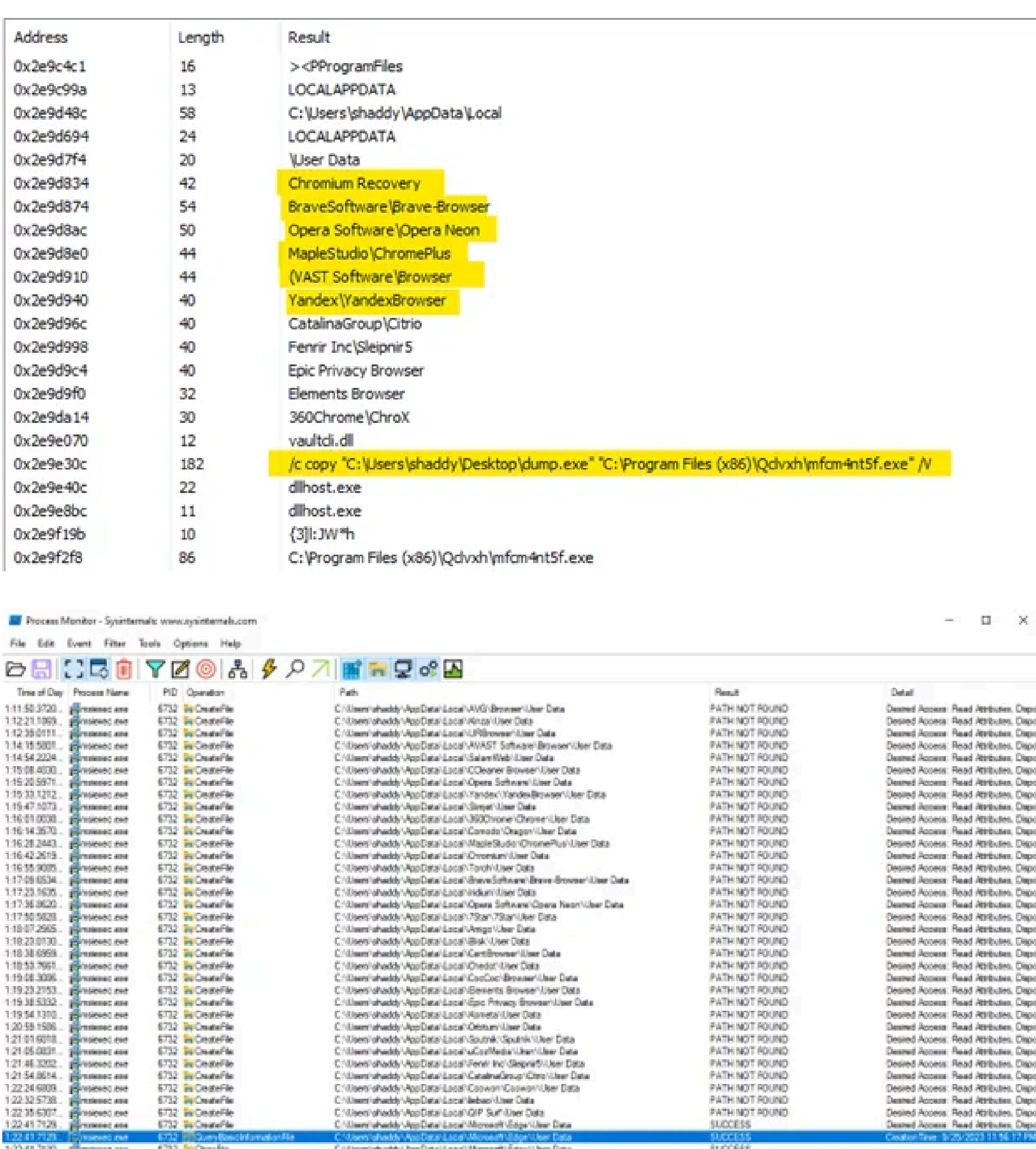
Targeted Processes & Applications

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.

Press enter or click to view image in full size

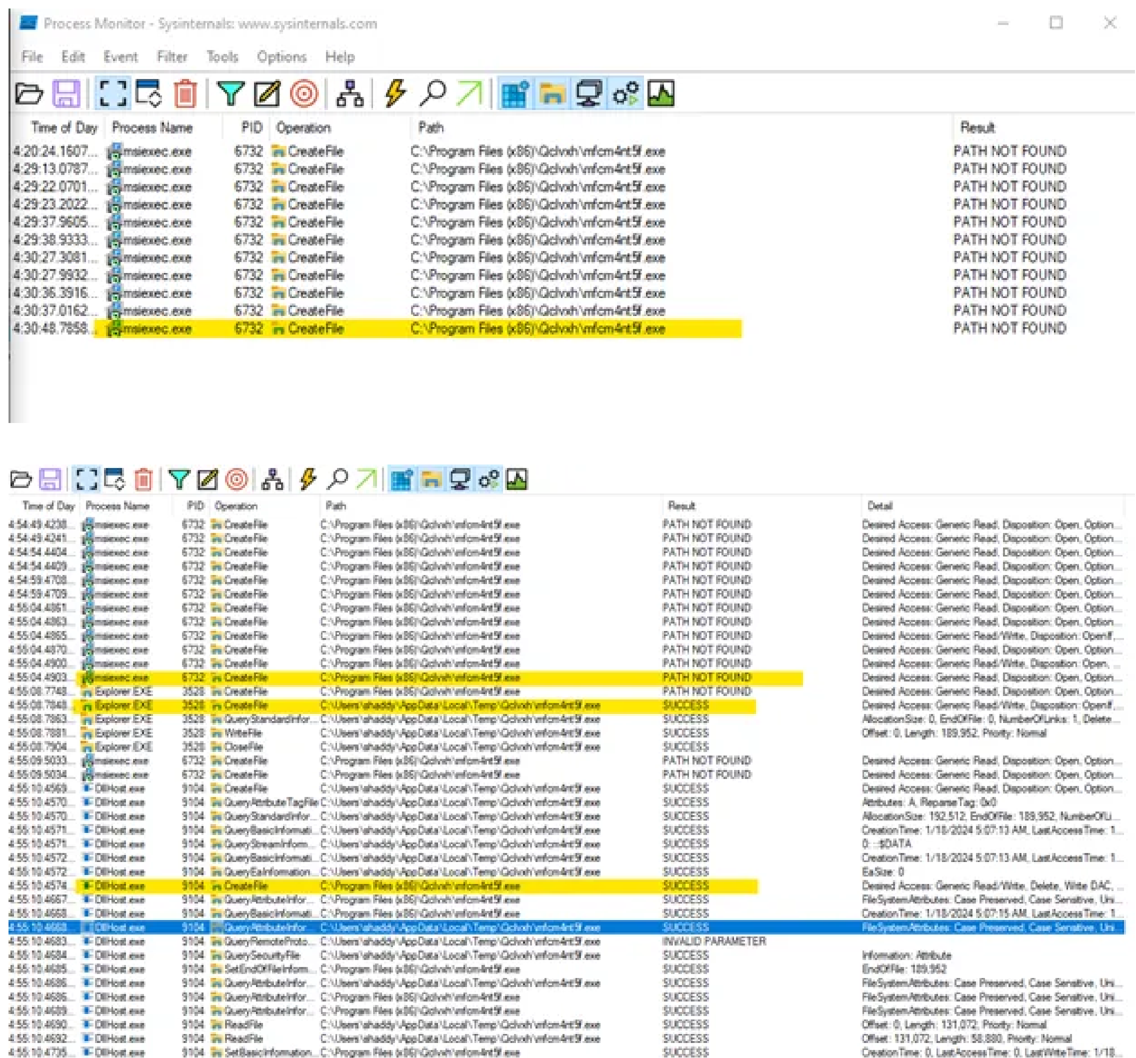




Address	Length	Result
0x2e9c4c1	16	><PProgramFiles
0x2e9c99a	13	LOCALAPPDATA
0x2e9d48c	58	C:\Users\shaddy\AppData\Local
0x2e9d694	24	LOCALAPPDATA
0x2e9d7f4	20	\User Data
0x2e9d834	42	Chromium Recovery
0x2e9d874	54	BraveSoftware\Brave-Browser
0x2e9d8ac	50	Opera Software\Opera Neon
0x2e9d8e0	44	MapleStudio\ChromePlus
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

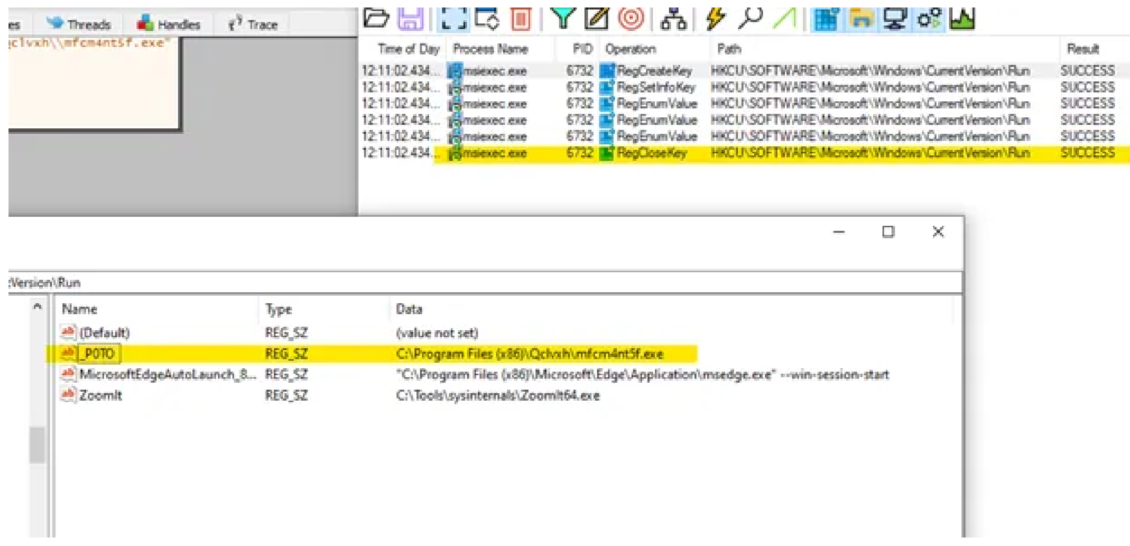
Privilege Escalation:

- 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



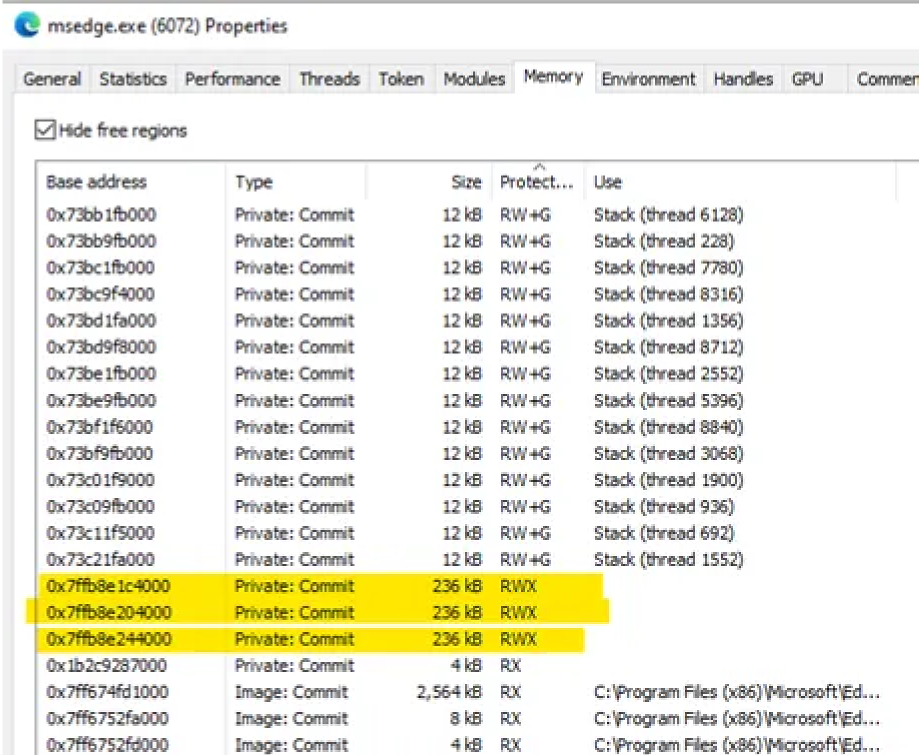
Persistence:

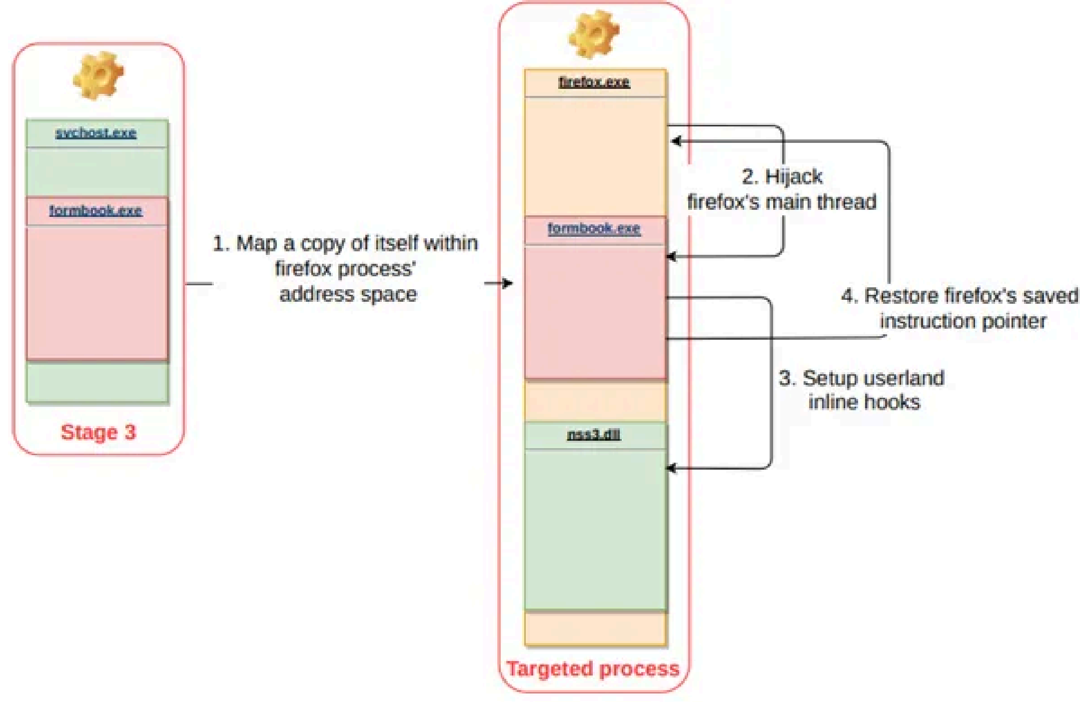
- After the malware is copied in Program Files
- It achieves persistence by adding Run Registry Keys
- It uses the API **NtCreateKey**



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
- 1. Victim Process
- 2. Stage3 Malware
- 3. Explorer
- The xloader is stuck in a loop here
- On every loop, it does the following:
- 1. Enumerates all running processes
- 2. Set inline hooks in targeted processes if found (by injecting code)
- 3. Steal clipboard data
- 4. Tries to create a file in program files
- 5. Adds registry in RunKeys
- 6. 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.





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

Setting hooks

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

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

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/
- https://cloud.google.com/blog/topics/threat-intelligence/formbook-malware-distribution-campaigns
- https://www.stormshield.com/news/in-depth-formbook-malware-analysis-obfuscation-and-process-injection/

Source: https://medium.com/@shaddy43/layers-of-deception-analyzing-the-complex-stages-of-xloader-4-3-malware-evolution-2dcb550b98d9