Analyzing the nasty .NET protection of the Ploutus.D malware.

Archived: 2026-04-05 14:30:35 UTC

Twitter: @s4tan

EDIT: The source code is now online: https://github.com/enkomio/Conferences/tree/master/HackInBo2018

Recently the ATM malware Ploutus.D reappeared in the news as being used to attack US ATM ([1]). In this post I'll show a possible analysis approach aimed at understanding its main protection. The protection is composed of different layers of protection, I'll focus on the one that, in my hopinion, is the most annoying, leaving the others out. If you want a clear picture of all the implied protections, I strongly recommend you to take a look at the *de4dot* Reactor deobfuscator code.

Introduction

Reversing .NET malware, in most cases, is not that difficult. This is mostly due to the awesome tool dnSpy ([2]), which allows debugging of the decompiled version of the Assembly. Most of the .NET malware use some kind of loader which decrypts a blob of data and then loads the result through a call to the *Assembly.Load* method ([3]).

From time to time some more advanced protection are involved, like the one analysed by Talos in [4]. What the article doesn't say is that in this specific case the malware uses a multi files assembly ([5]).

This implies that instead of using the *Assembly.Load* method, it uses the way less known *Assembly.LoadModule* method ([6]). This protection method is a bit more difficult to implement but I have to say that is way more effective as obfuscation. The malware also encrypt the method bodies and decrypt them only when necessary. This protection is easily overcome by calling the "*Reload All Method Bodies*" command in dnSpy at the right moment (as also showed in the Talos article).

Ploutus.D is also protected with an obfuscator which encrypts the method bodies and decrypts them only when necessary. The protector used is *.NET Reactor* ([7]) as also pointed out in a presentation by Karspersky ([8]). This particular protection is called *NecroBit Protection*, and from the product website we can read that:

> NecroBit is a powerful protection technology which stops decompilation. NecroBit replaces the CIL code within methods with encrypted code. This way it is not possible to decompile/reverse engineer your method source code.

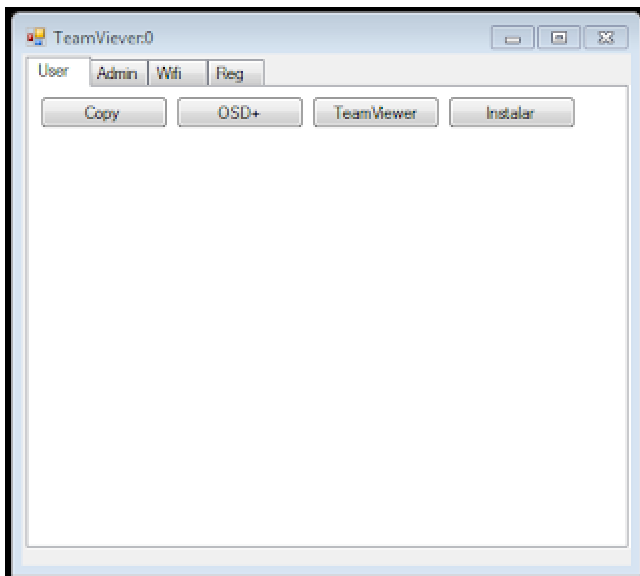
The difference with the previous case is that if we try to use the "Reload All Method Bodies" feature in dnSpy, it will fail (this is not technically correct since there is nothing to reload as we will see).

Reversing Ploutus.D obfuscation

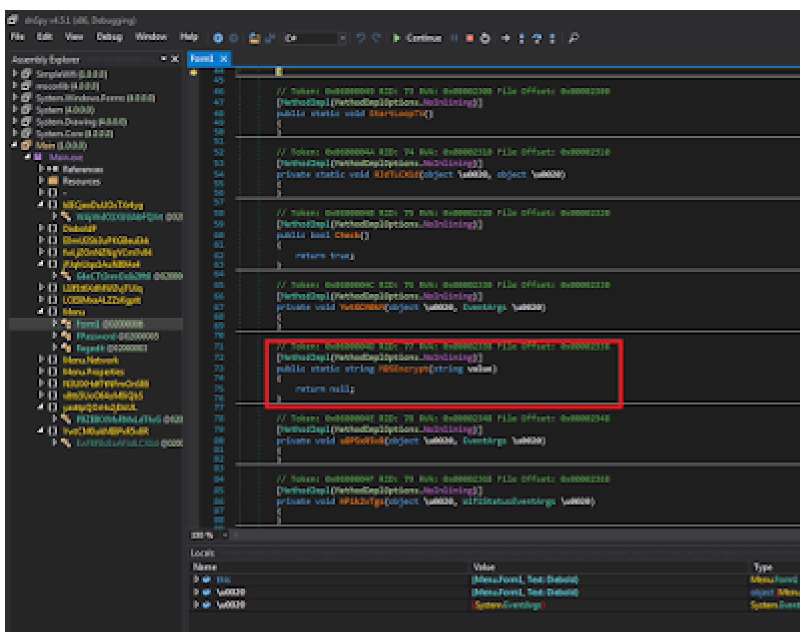
To write this blog post I have reversed the sample with MD5 ae3adcc482edc3e0579e152038c3844e. When I start to analyse a .NET malware, as first task I ran my tool **Shed** ([9]) in order to have a broad overview of what the

malware does and to try to extract dynamically loaded Assemblies. In this case I was able to extract some useful strings (like the configured backend usbtest[.]ddns[.]net) but not the Assembly with the method bodies decrypted (however this is not an error and as we will see it is the correct behaviour).

The next step is to debug the program with dnSpy. If you run it the following Form will be displayed:



I started to dig a bit on the classes that extend the Form class in order to identify which commands are supported. Unfortunately most of the methods of these classes are empty, as can be seen from the following screenshot:



It is interesting to note that all the static constructors are not empty. All of them are pretty simple (in some cases they have just one instruction), what it is interesting is that all of them call the same method: **P9ZBIKXMsRMxLdTfcG.Nf9E3QXmJD();**, which is marked as *internal unsafe static void Nf9E3QXmJD()*.

By analysing it, the thing start to get interesting since this method is pretty huge, especially since it implements a very annoying control flow obfuscation. It is interesting to notice that if we set a breakpoint on this method and re-

start the debugging session, it is amongst the first methods invoked by the program. Scrolling through the code we can find the following interesting statement:

```
if (P9ZBIKXMsRMxLdTfcG.Ax6OYTY7tiMf4Yu1B4(P9ZBIKXMsRMxLdTfcG.XnSi7dQe0TUTJbDcxg(P9ZBIKXMsRMxLdTfcG.C
```

This piece of code is particularly interesting, since it tries to identify the *clrjit.dll* module. Once found, it identifies the CLR version, which in my case is 4.0.30319.0. Then, it extracts the resource *m7fEJg2w6sBe9LM3D3.i4tjc9Xt0Vhu5G72Uh*.

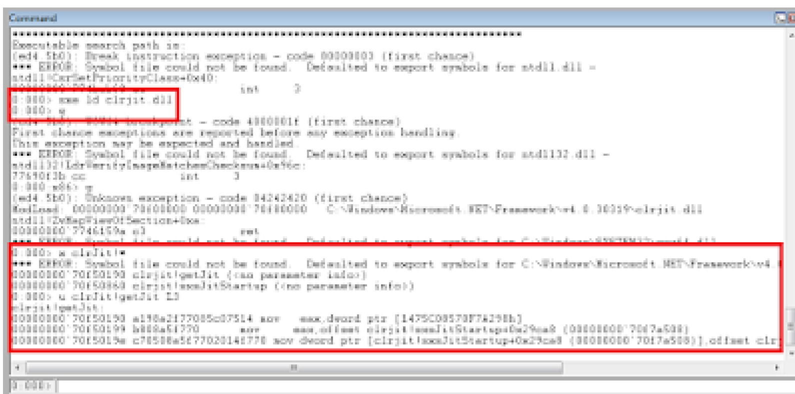
After a while the *getJit* string appears in the execution. This function is exported by *clrjit.dll* and it is a very important method since it allows to get a pointer to the *compileMethod* method. To know more about it you could refer to my Phrack article about .NET program instrumentation ([10]). We can also identify a call to the *VirtualProtect* method.

With these information we can start to make some assumption, like that the malware hook the *compileMethod* method in order to force the compilation of the real MSIL bytecode. Let's verify our assumption, in order to do so we need to change tool, in particular we will use *WinDbg* with the *SOS extension* (if you want to know more about debugging .NET applications with WinDbg take a look at my presentaion [11]).

In order to inspect the program at the right moment, we will set an exception when the *clrjit.dll* library is loaded. This is easily done with the command:

```
sxe ld clrjit.dll
```

once that this exception is raised let's inspect the *clrjit* module as showed in the following image:



The *getJit* method is an exported by *clrjit* dll and returns the address of the *VTable* of an *ICorJitCompiler* object, where the first item is a pointer to the *compileMethod* method, as can be seen from the source code ([12]). But, since we don't trust the source code, let's debug the *getJit* method till the *ret* instruction and inspect the return value stored in *eax*:

as can be seen from the image above, the address of the *compileMethod* is at *0x70f049b0*. Now let's the program run until the main windows is displayed and then break the process in the debugger. Let's display again the content of the *VTable* (which was *0x70f71420*).

As can be seen from the image above the value of the first entry of the *VTable* changed to from *0x70f049b0* to *002a0000*. So our assumption about the hooking of the *compileMethod* was right :)

Now we want to identify which method hooked the *compileMethod* method. To do this we will load the *SOS* extension (with the command *.loadby SOS clrjit*), set a breakpoint at the *compileMethod* method and when the brakpoint hits, type *!CLRStack* command to see which method was set as replacement. In order to trigger the *compileMethod* breakpoint I clicked on a random button in the interface.

from the image above we can spot that the interested method is *qtlEIBBYuV*. Find below the decompiled code of the metohd (I have renamed the argument names and added some comments):

```
[MethodImpl(MethodImplOptions.NoInlining)]
internal static uint qtlEIBBYuV(IntPtr \u0020, IntPtr a1, IntPtr \u0020, [MarshalAs(UnmanagedType.U4)] uint \u0020, IntPtr \u0020, ref uint \u0020)
{
// thisi s a pointer to the COREINFO_METHOD_INFO structure
IntPtr ptr = \u0020;
if (P9ZBIKXMsRMxLdTfcG.KOnEge1tX2)
```

```
{
ptr = a1;
}
long num;
if (IntPtr.Size == 4)
{
num = (long)Marshal.ReadInt32(ptr, IntPtr.Size * 2); (1)
}
else
{
num = Marshal.ReadInt64(ptr, IntPtr.Size * 2);
}
// load the ILCode address
object obj = P9ZBIKXMsRMxLdTfcG.k6dbsY0qhy[num];
if (obj == null)
{
// proceed with the standard compilation of the method
return P9ZBIKXMsRMxLdTfcG.u8lbx6nt2g(\u0020, a1, \u0020, \u0020, \u0020, ref \u0020);
}
// allocate a pinned memory buffer in order to copy the real IL code
P9ZBIKXMsRMxLdTfcG.QjP40LkcxwMvQVmG1a qjP40LkcxwMvQVmG1a = (P9ZBIKXMsRMxLdTfcG.QjP40LkcxwMvQVmG1a)obj; (2)
IntPtr intPtr = Marshal.AllocCoTaskMem(qjP40LkcxwMvQVmG1a.wsQbTfcGfS.Length);
Marshal.Copy(qjP40LkcxwMvQVmG1a.wsQbTfcGfS, 0, intPtr, qjP40LkcxwMvQVmG1a.wsQbTfcGfS.Length);
if (qjP40LkcxwMvQVmG1a.KJXb28UyS5)
{
```

```
// call VirtualProtect if necessary
a1 = intPtr;
a1 = (uint)qjP40LkcxwMvQVmG1a.wsQbTfcGfS.Length;
P9ZBIKXMsRMxLdTfcG.fptEBhe4Kh(\u0020, qjP40LkcxwMvQVmG1a.wsQbTfcGfS.Length, 64, ref P9ZBIKXMsRMxLdTfcG.Tufbd24KkS);
return 0u;
}
// write back the address of the COREINFO_METHOD_INFO.ILCode
Marshal.WriteIntPtr(ptr, IntPtr.Size * 2, intPtr); (3)
// write back the real length, field COREINFO_METHOD_INFO.ILCodeSize
Marshal.WriteInt32(ptr, IntPtr.Size * 3, qjP40LkcxwMvQVmG1a.wsQbTfcGfS.Length); (4)
uint result = 0u;
if (\u0020 != 216669565u || P9ZBIKXMsRMxLdTfcG.iJnbQwwaCg)
{
// call the real compileMethod with the real ILcode
result = P9ZBIKXMsRMxLdTfcG.u8lbx6nt2g(\u0020, a1, \u0020, \u0020, \u0020, ref \u0020); (5)
}
else
{
P9ZBIKXMsRMxLdTfcG.iJnbQwwaCg = true;
}
return result;
}
```

What is interesting from the code above is that:

- it reads the address of the *COREINFO_METHOD_INFO* structure at (1)
- writes back the real MSIL bytecode at (2)
- updates the fields *ILCode* and *ILCodeSize* at (3) and (4)
- finally call the original *compileMethod* at (5)

In this way, it is sure that the correct MSIL code is compiled and executed (for more info on this structure please refer to [10,12]).

Finally, we have a pretty good understanding of how the real code is protected, now we can try to implement a simple program which dumps the real MSIL bytecode and rebuilds the assembly. The de4dot tool, instead, uses a different approach, which is based on emulating the decryption code of the method body and then rebuild the assembly.

Let's the code speak

A possible approach to dump the real MSIL bytecode is:

- Hook the *compileMethod* before the malware
- Force all static constructors to be invoked and force compilation of all methods via *RuntimeHelpers.PrepareMethod*. This will ensure that we are able to grab all the ILCode of the various methods.
- When the hook is invoked store the values of the fields ILCode and ILCodeSize. We have to record also which method is currently compiled, this is done with the code *getMethodInfoFromModule* from [10].
- Rebuild the assembly by using Mono.Cecil or dnlib (my choice)

However, for this specific case, I'll use a slightly different approach, which is not as generic as the previous one but it is simpler and more interesting imho :)

As we have seen from the code above, the *P9ZBIKXMsRMxLdTfcG.k6dbsY0qhy* is a dictionary of objects which contains the real MSIL bytecode as value and as key the address of the MSIL buffer. What we can do is to read the value of this object via reflection and rebuild the original binary. All this without implying the hooking of any methods :)

I have implemented a simple program that extracts those values via reflection, calculates the address of each method and rebuild the assembly. If you want to take a look it, here is the code.

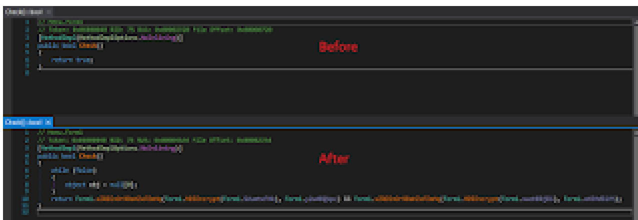
```
open System
open System.Linq
open System.Reflection
open System.Runtime.CompilerServices
open System.Collections
open System.Collections.Generic
open System.Diagnostics
open Microsoft.Diagnostics.Runtime
open dnlib.DotNet
```

```
open dnlib.DotNet.Emit
open dnlib.IO

let runAllStaticConstructor(assembly: Assembly) =
assembly.GetTypes()
|> Seq.iter(fun assemblyType ->
RuntimeHelpers.RunClassConstructor(assemblyType.TypeHandle)
)

let getMsilStorage(assembly: Assembly) =
let assemblyType = assembly.GetType("yasttpQOrHx2jEkiUL.P9ZBIKXMsRMxLdTfcG")
let storageField = assemblyType.GetField("k6dbsY0qhy", BindingFlags.NonPublic ||| BindingFlags.Static)
let hashTable = storageField.GetValue(null) :?> Hashtable
hashTable.Cast<DictionaryEntry>()
|> Seq.map(fun kv ->
// extract bytes array
let bytesField =
kv.Value.GetType().GetFields(BindingFlags.Instance ||| BindingFlags.NonPublic)
|> Seq.find(fun field -> field.FieldType.IsArray)
let msilBytes = bytesField.GetValue(kv.Value) :?> Byte array
(kv.Key :?> Int64, msilBytes))
|> Map.ofSeq

let getAssemblyBaseAddress(assembly: Assembly) =
let pid = Process.GetCurrentProcess().Id
let dataTarget = DataTarget.AttachToProcess(pid, uint32 5000, AttachFlag.Passive)
let runtime = dataTarget.ClrVersions.[0].CreateRuntime()
let assemblyModule = runtime.Modules |> Seq.find(fun m -> m.Name.Equals(assembly.Location))
```

```
let baseAddress = int32 assemblyModule.ImageBase
dataTarget.Dispose()
baseAddress

[<EntryPoint>]
let main argv =
let fullPath = @"PloutusD_d93342bd12ef44d92bf58ed2f0f88443385a0192804a5d0976352484c0d37685.exe"
let assembly = Assembly.LoadFile(fullPath)

// run all static constructor to fill the protection dictionary containing the real MSIL bytecode
runAllStaticConstructor(assembly)

// retrieve the dictionary via reflection. This is higly dependant to a specific sample, the field name may change from sample to sample
let msilStorage = getMsilStorage(assembly)

// get the malware base addres in order to calculate the method body address
let baseAddress = getAssemblyBaseAddress(assembly)

// use a modified version of dnlib in order to set the real MSIL bytecode
let dnModule = ModuleDefMD.Load(fullPath)
dnModule.Types
|> Seq.map(fun t -> t.Methods)
|> Seq.concat
|> Seq.filter(fun dnMethod -> dnMethod.HasBody)
|> Seq.iter(fun dnMethod ->
dnMethod.Body.KeepOldMaxStack <- true

// skip the body header
let offset = int32 dnMethod.Body.HeaderSize
```

	let ilAddress = int32 dnMethod.RVA + baseAddress + offset
	if msilStorage.ContainsKey(int64 ilAddress) then
	let realMsilBytes = msilStorage.[int64 ilAddress]
	dnMethod.Body.MaxStack <- uint16 50
	// set the body via raw buffer. This property is not present in the real dnlib code :P
	dnMethod.Body.RawBody <- new List<Byte>(realMsilBytes)
	Console.WriteLine("Rebuit: " + dnMethod.FullName)
	)
	// finally write back the new assembly
	dnModule.Write(fullPath + "_rebuilt")
	0

After dumped the real MSIL, we can see that now the methods are not empty anymore:



Conclusion

The purpose of this post was to show how to analyse, in an effective way, a strongly obfuscate malware with the help of different tools and the knowledge of the internal working of the .NET framework.

As an alternative, if you want to obtain a de-obfuscated sample I encourage you to use the de4dot tool (and to read the code since this project is a gold mine of information related to the .NET internals).

At the time of this writing the sample is not correctly deobfuscated by de4dot due to an error in the string decryption step. To obtain a deobfuscated sample with the real method body, just comment out the string decryption step in *ObfuscatedFile.cs*.

Too often developers underestimate the power of reflection and as a result it is not uncommon to bypass protection (included license verification code) only by using reflection and nothing more :)

References

[1] First ‘Jackpotting’ Attacks Hit U.S. ATMs - https://goo.gl/6WY14V

[2] dnSpy - https://github.com/0xd4d/dnSpy

[3] Assembly.Load Method (Byte[]) - https://goo.gl/owZtC1

[4] Recam Redux - DeConfusing ConfuserEx - https://goo.gl/oKgj1k

[5] How to: Build a Multifile Assembly - https://goo.gl/mVdHuU

[6] Assembly.LoadModule Method (String, Byte[]) - https://goo.gl/D6N797

[7] .NET REACTOR - http://www.eziriz.com/dotnet_reactor.htm

[8] Threat hunting .NET malware with YARA.pdf - https://goo.gl/RxEw1G

[9] Shed, .NET runtime inspector - https://github.com/enkomio/shed

[10] http://www.phrack.org/papers/dotnet_instrumentation.html

[11] .NET for hackers - https://www.slideshare.net/s4tan/net-for-hackers

[12] getJit() - https://github.com/dotnet/coreclr/blob/master/src/inc/corjit.h#L241

Source: http://antonioparata.blogspot.co.uk/2018/02/analyzing-nasty-net-protection-of.html