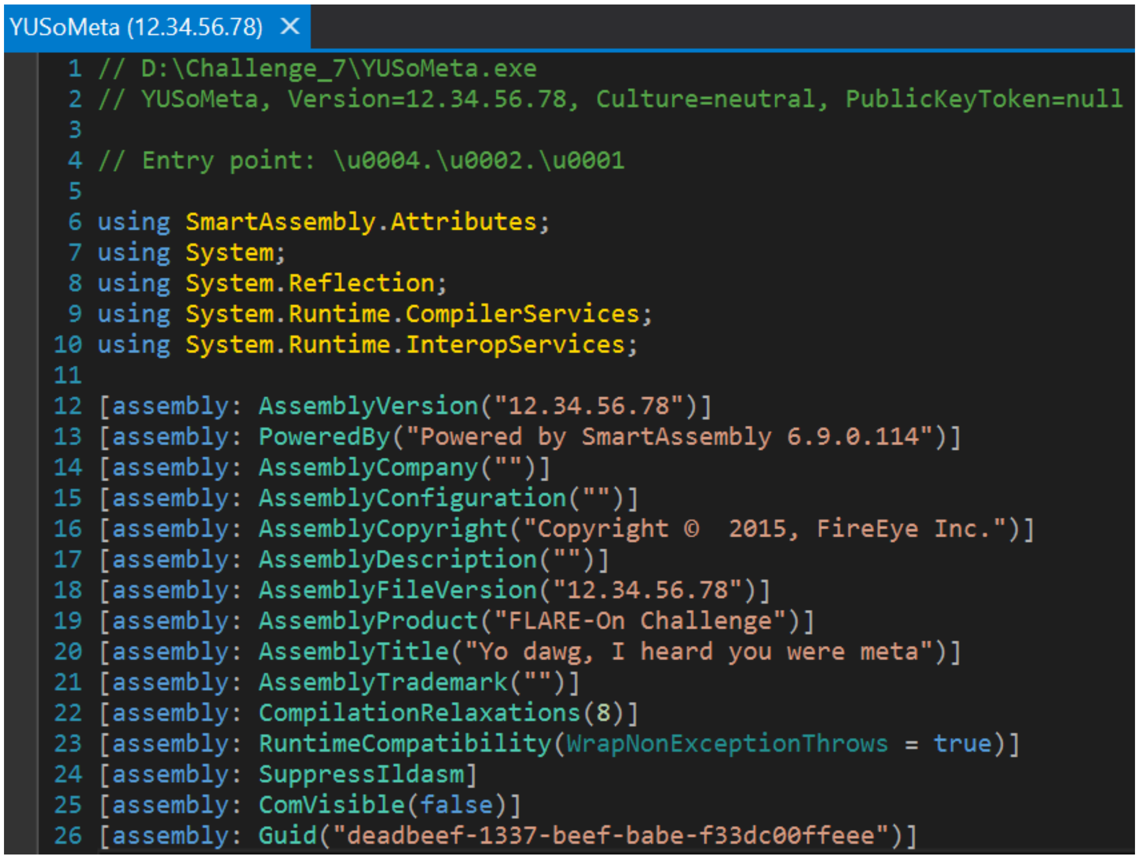
Challenge #7 Solution

by Matt Graeber

`YUSoMeta.exe` is an obfuscated .NET executable that claims to be 100% tamper proof. The goal of this challenge is to provide the correct password in the hopes of revealing the password to the next challenge. Normally, one would be able to open this executable in a .NET decompiler, navigate to the entry point method, and then look for the password comparison logic. Loading this executable into my new favorite decompiler – `dnspy`, however reveals that it was obfuscated with `SmartAssembly 6.9.0.114`.

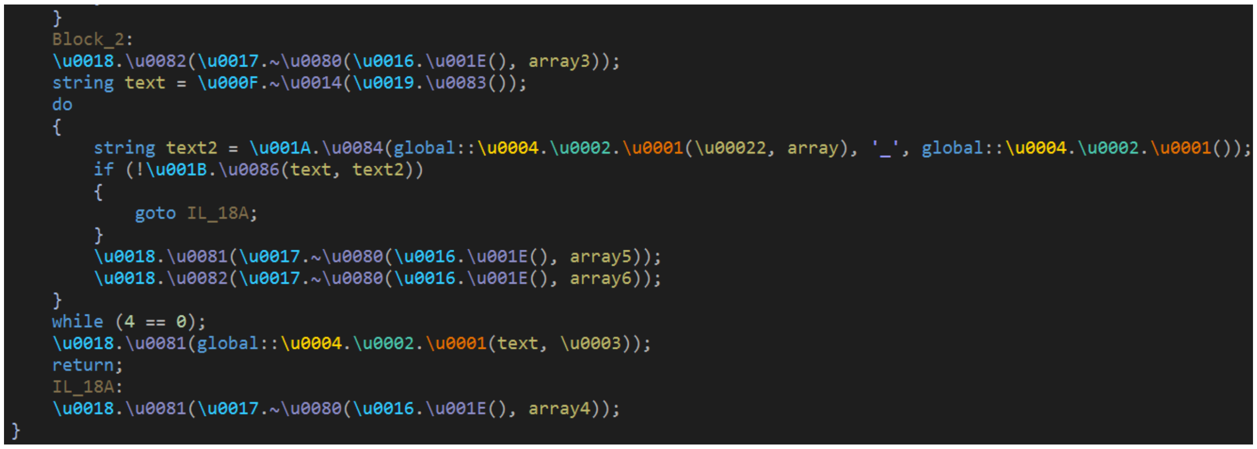


YUSoMeta (12.34.56.78)

```
// D:\Challenge_7\YUSoMeta.exe
// YUSoMeta, Version=12.34.56.78, Culture=neutral, PublicKeyToken=null

// Entry point: \u0004.\u0002.\u0001

using SmartAssembly.Attributes;
using System;
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;

[assembly: AssemblyVersion("12.34.56.78")]
[assembly: PoweredBy("Powered by SmartAssembly 6.9.0.114")]
[assembly: AssemblyCompany("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCopyright("Copyright ©  2015, FireEye Inc.")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyFileVersion("12.34.56.78")]
[assembly: AssemblyProduct("FLARE-On Challenge")]
[assembly: AssemblyTitle("Yo dawg, I heard you were meta")]
[assembly: AssemblyTrademark("")]
[assembly: CompilationRelaxations(8)]
[assembly: RuntimeCompatibility(WrapNonExceptionThrows = true)]
[assembly: SuppressIldasm]
[assembly: ComVisible(false)]
[assembly: Guid("deadbeef-1337-beef-babe-f33dc00ffeee")]
```

Figure 1: YUSoMeta..exe assembly attributes

Looking at a snippet of the entry point method, it is not immediately obvious what is going on due the obfuscation.



```
    }
    Block_2:
    \u0018.\u0082(\u0017.~\u0080(\u0016.\u001E(), array3));
    string text = \u000F.~\u0014(\u0019.\u0083());
    do
    {
        string text2 = \u001A.\u0084(global::\u0004.\u0002.\u0001(\u00022, array), '_', global::\u0004.\u0002.\u0001());
        if (!\u001B.\u0086(text, text2))
        {
            goto IL_18A;
        }
        \u0018.\u0081(\u0017.~\u0080(\u0016.\u001E(), array5));
        \u0018.\u0082(\u0017.~\u0080(\u0016.\u001E(), array6));
    }
    while (4 == 0);
    \u0018.\u0081(global::\u0004.\u0002.\u0001(text, \u0003));
    return;
    IL_18A:
    \u0018.\u0081(\u0017.~\u0080(\u0016.\u001E(), array4));
}
```

Figure 2: Obfuscated YUSoMeta.exe entry point method

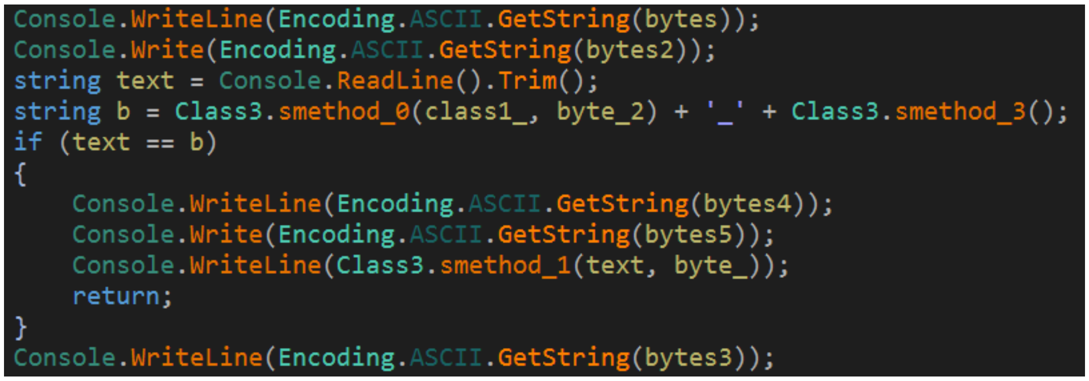
At this point, rather than attempting to understand all of the obfuscated logic, I will see if de4dot – an extremely powerful .NET deobfuscation utility that deobfuscates most .NET executables out of the box - will get the job done. I executed the following in order to attempt to deobfuscate the executable:

```
D:\Challenge_7>de4dot YUSoMeta.exe -o YUSoMeta_deobfuscated.exe

de4dot v3.1.41592.3405 Copyright (C) 2011-2014 de4dot@gmail.com
Latest version and source code: https://github.com/0xd4d/de4dot
21 deobfuscator modules loaded!

Detected SmartAssembly 6.9.0.114 (D:\Challenge_7\YUSoMeta.exe)
Cleaning D:\Challenge_7\YUSoMeta.exe
Renaming all obfuscated symbols
Saving D:\Challenge_7\YUSoMeta_deobfuscated.exe
```

Figure 3: Command Line output

There are a lot of options that you can provide `de4dot` which become necessary when deobfuscation fails out of the box. However, these options seen above are the default options I will use. The `-o` flag specifies the output file name of the new, deobfuscated executable.

Now, after loading `YUSoMeta_deobfuscated.exe` into `dnspy`, the logic of the entry point method becomes much clearer. So out of the box, de4dot correctly identified the fact that it was obfuscated with Smart Assembly and it deobfuscated everything accordingly.



```
Console.WriteLine(Encoding.ASCII.GetString(bytes));
Console.Write(Encoding.ASCII.GetString(bytes2));
string text = Console.ReadLine().Trim();
string b = Class3.smethod_0(class1_, byte_2) + '_' + Class3.smethod_3();
if (text == b)
{
    Console.WriteLine(Encoding.ASCII.GetString(bytes4));
    Console.Write(Encoding.ASCII.GetString(bytes5));
    Console.WriteLine(Class3.smethod_1(text, byte_));
    return;
}
Console.WriteLine(Encoding.ASCII.GetString(bytes3));
```

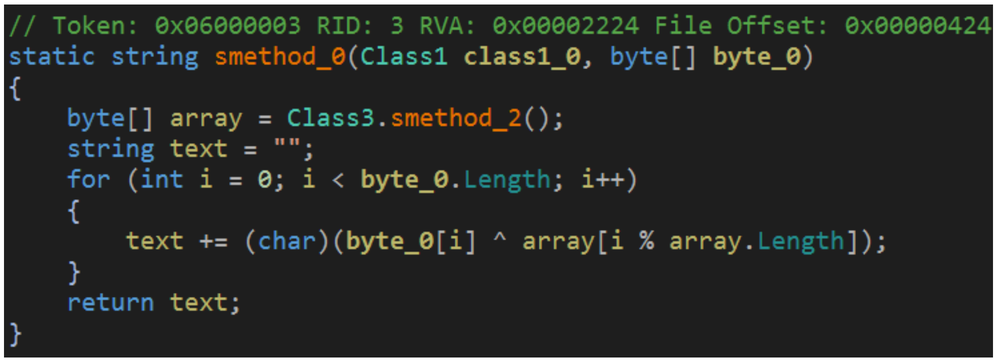
Figure 4: Deobfuscated entry point method snippet

It is now evident that user input (via `Console.ReadLine`) is compared against a string derived from the execution of two methods: `smethod_0` and `smethod_3`. Going back to the entry point method of the obfuscated entry point, the similarities in logic become a little clearer – specifically, you can see that two methods are executed and an underscore is inserted in between the strings returned. Take note of these methods because we may want to execute them later on.

Now at this point, I would typically be faced with the following options:

1. Investigate how `smethod_0` and `smethod_3` are implemented and attempt to recreate the logic in a C# project or PowerShell script or
2. Let the executable do the work for us by invoking those methods dynamically using the .NET reflection libraries.

Anyone should be able to read deobfuscated C# code and eventually figure out how the password is derived. I'm lazy however and I'd rather compel the executable to tell me what the password is. Let's investigate `smethod_0`.



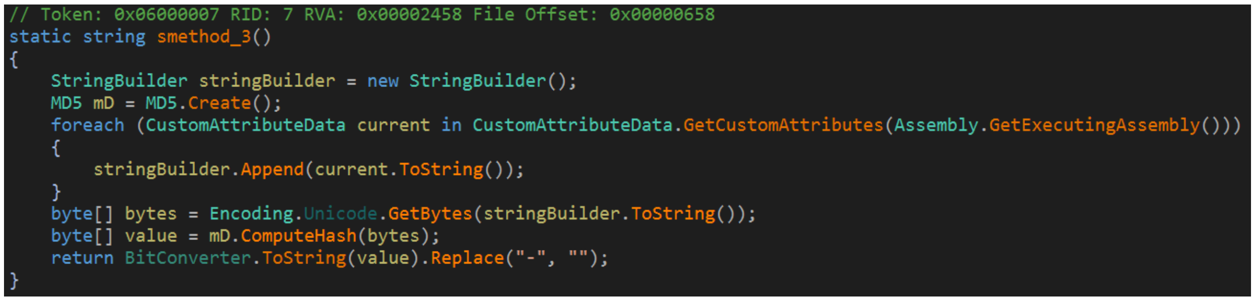
```
// Token: 0x06000003 RID: 3 RVA: 0x00002224 File Offset: 0x00000424
static string smethod_0(Class1 class1_0, byte[] byte_0)
{
    byte[] array = Class3.smethod_2();
    string text = "";
    for (int i = 0; i < byte_0.Length; i++)
    {
        text += (char)(byte_0[i] ^ array[i % array.Length]);
    }
    return text;
}
```

Figure 5: Password portion #1 decoder

`smethod_0` takes the result of `smethod_2` and uses it as the basis for a rolling XOR key for the byte array passed in. The byte array passed in to `smethod_0` is the following:

```
31,100,116,97,0,84,69,21,115,97,109,29,79,68,21,104,115,104,21,84,78
```

Next, let's briefly look at the implementation of `smethod_3`.



```
// Token: 0x06000007 RID: 7 RVA: 0x00002458 File Offset: 0x00000658
static string smethod_3()
{
    StringBuilder stringBuilder = new StringBuilder();
    MD5 mD = MD5.Create();
    foreach (CustomAttributeData current in CustomAttributeData.GetCustomAttributes(Assembly.GetExecutingAssembly()))
    {
        stringBuilder.Append(current.ToString());
    }
    byte[] bytes = Encoding.Unicode.GetBytes(stringBuilder.ToString());
    byte[] value = mD.ComputeHash(bytes);
    return BitConverter.ToString(value).Replace("-", "");
}
```

Figure 6: Password portion #2 derivation method

This is a static method that doesn't take any arguments and returns an MD5 hash string.

So now we are armed with the following information:

1) The 1st portion of the password is derived by XOR decoding a byte array with a byte array returned from the smethod_2 method
2) The 2nd portion of the password is an MD5 hash of some unknown value.
3) An underscore is inserted in between each portion of the password

Now let's obtain the metadata token numbers for each of the methods that we'd like to execute from the obfuscated binary. For reference, a metadata token is a numeric identifier for any .NET member (including methods). Using the .NET reflection library, methods can be referenced by their metadata token. This is useful because obfuscation utilities will often rename methods to unprintable Unicode strings making it so that they cannot be easily referenced by name. In `dnspy`, the metadata token is visible at the top left of any method. They always begin with 0x06.

Solution #1

Once the metadata tokens are obtained for the methods we want to execute, a simple PowerShell script can be written to perform that password validation for us.

```
# Path to the obfuscated executable
$ObfuscatedFilePath = 'D:\Challenge_7\YUSoMeta.exe'

# Read in the obfuscated executable in memory. This will allow us to invoke its
# methods dynamically.
$Assembly = [Reflection.Assembly]::Load([IO.File]::ReadAllBytes($ObfuscatedFilePath))

# Get a reference to main assembly module.
# This exposes the ResolveMethod method.
$Module = $Assembly.ManifestModule

# Encoded password copied from the deobfuscated executable
$EncodedPassword = [Byte[]]
@(31,100,116,97,0,84,69,21,115,97,109,29,79,68,21,104,115,104,21,84,78)
```

```
# Obtained from the obfuscated executable
$ReturnXORKeyMetadataToken = 0x06000006
$ReturnXORKey = $Module.ResolveMethod($ReturnXORKeyMetadataToken)

# Invoke the method that obtains the XOR key
$XorKey = $ReturnXORKey.Invoke($null, @())

# Decode the forst part of the password
$DecodedPassword = for ($i = 0; $i -lt $EncodedPassword.Length; $i++) {
    $EncodedPassword[$i] -bxor $XorKey[$i % ($XorKey.Length)]
}

# Convert the decoded byte array to an ASCII string
$PWDPart1 = [Text.Encoding]::ASCII.GetString($DecodedPassword)
# metaprogrammingisherd

# Obtained from the obfuscated executable
$GetPWDPart2MetadataToken = 0x06000008
$GetPWDPart2 = $Module.ResolveMethod($GetPWDPart2MetadataToken)

# Invoke the method that obtains the 2nd portion of the password
$PWDPart2 = $GetPWDPart2.Invoke($null, @())
# DD9BE1704C690FB422F1509A46ABC988

# Concatenate both portions of the password
$Password = $PWDPart1 + '_' + $PWDPart2
# metaprogrammingisherd_DD9BE1704C690FB422F1509A46ABC988
```
Figure 7: PowerShell script to perform password validation

Solution #2

If the correct password is compared against the password input by the user, then won't the password just be present in memory? Probably. Could it really be as simple as attaching a debugger to the process and just pulling out the string? Yeah, probably. As the saying goes, there's no such thing as cheating in reverse engineering.

I'll attempt to locate the string using my favorite debugger – `WinDbg/cdb`. I'll make this relatively easy: I'll start the process in the debugger, enter a password, and let execution run until completion and hope that the password will just be sitting around in memory. Also, since we're dealing with a .NET executable, we have a very powerful `WinDbg` extension at our disposal – `sos.dll`. One of the features of `sos.dll` is that it allows you to dump .NET objects of any type. Since the password is a `System.String` object, we can use the debugger extension do dump all .NET strings in memory and hopefully find the password.

Lastly, before we get started, pretend you didn't already know what the password was. How could we derive it? Well, we could easily find out the length of the string. Recall that the first part of the password is an XOR encoded byte array of length 21, there is an underscore, and the second part of the password is an MD5 string. An MD5 string is 32 characters in length. So the total number of characters of the password is 54.

The following `cdb` commands will run `YUSoMeta.exe` to completion, load sos.dll, and dump all strings of length 54:

```
D:\Challenge_7>cdb -o YUSoMeta.exe

Microsoft (R) Windows Debugger Version 10.0.10158.9 AMD64
Copyright (c) Microsoft Corporation. All rights reserved.

CommandLine: YUSoMeta.exe

************* Symbol Path validation summary **************
Response                         Time (ms)     Location
Deferred
srv*C:\Symbols*http://msdl.microsoft.com/download/symbols
Symbol search path is:
srv*C:\Symbols*http://msdl.microsoft.com/download/symbols
Executable search path is:
ModLoad: 00000000`00ea0000 00000000`00ea8000   image00000000`00ea0000
ModLoad: 00007ff8`9b670000 00007ff8`9b831000   ntdll.dll
ModLoad: 00007ff8`919c0000 00007ff8`91a28000
C:\Windows\SYSTEM32\MSCOREE.DLL
ModLoad: 00007ff8`997c0000 00007ff8`9986d000
C:\Windows\system32\KERNEL32.dll
ModLoad: 00007ff8`98ab0000 00007ff8`98c8d000
C:\Windows\system32\KERNELBASE.dll
(10fc.d70): Break instruction exception - code 80000003 (first
chance)
ntdll!LdrpDoDebuggerBreak+0x30:
00007ff8`9b72e250 cc              int     3
0:000> g
Warning! This program is 100% tamper-proof!
Please enter the correct password: incorrectpassword
Y U tamper with me?
ntdll!NtTerminateProcess+0xa:
00007ff8`9b7037ba c3              ret
0:000> .loadby sos clr
0:000> .foreach ( stringobj { !dumpheap -short -type System.String })
{.if (dwo(${stringobj}+8) == 0n54) { .printf "%mu\n", ${stringobj}+C
} }

[System.Reflection.AssemblyConfigurationAttribute("")]
metaprogrammingisherd_DD9BE1704C690FB422F1509A46ABC988
```
Figure 8: Console Debugging Session

The `.foreach` command iterates over the address of each `System.String` object and compares the length field to 54. If there's a match, then it prints the string as a Unicode string.

The offsets of these fields become obvious if you were to inspect the bytes of a `System.String` object pointer.

What you just saw was a highly targeted method of pulling out the password. You could have just as easily pulled it out by dumping process memory and doing a strings search for all strings of length 54 that contain an underscore. There is a multitude of ways to approach the problem.

Lastly, you may have noticed that the generated password portions differed if you tried to generate them based on the deobfuscated version. The executable was designed to derive the password based on attributes only present in the obfuscated executable – hence, it being “tamper-proof”. This is why it’s beneficial to use the reflection libraries to invoke specific methods within the obfuscated executable.