Flare-On 3: Challenge 9 Solution - GUI.exe

Challenge Author: Jon Erickson

Overview

This challenge was written as an educational exercise with three goals in mind.

1) Improve .Net Reversing/ Debugging skills
2) Improve knowledge of ConfuserEx software protector
3) Introduction to secret sharing concepts

With this in mind I created a .NET executable that contained multiple resources in a Russian doll configuration, which contains multiple layers. The main executable and all resources have been protected with ConfuserEx using varies options with increasing levels of difficulty. This allows the challenger to experience ConfuserEx protections in an approachable setting where you can use the knowledge gained from previous layers of the challenge for the next. Each layer of the challenge contains one or more shares, which need to be combined to get the key.

Main Form

When you execute the GUI.exe executable you see that a Windows GUI starts up with a picture of a Russian Doll, text that gives a hint that we need to 'combine all 6 shares' and a Start button. Trying to click the Start button brings up a simple message box stating "Try Again…", as shown in Figure 1.

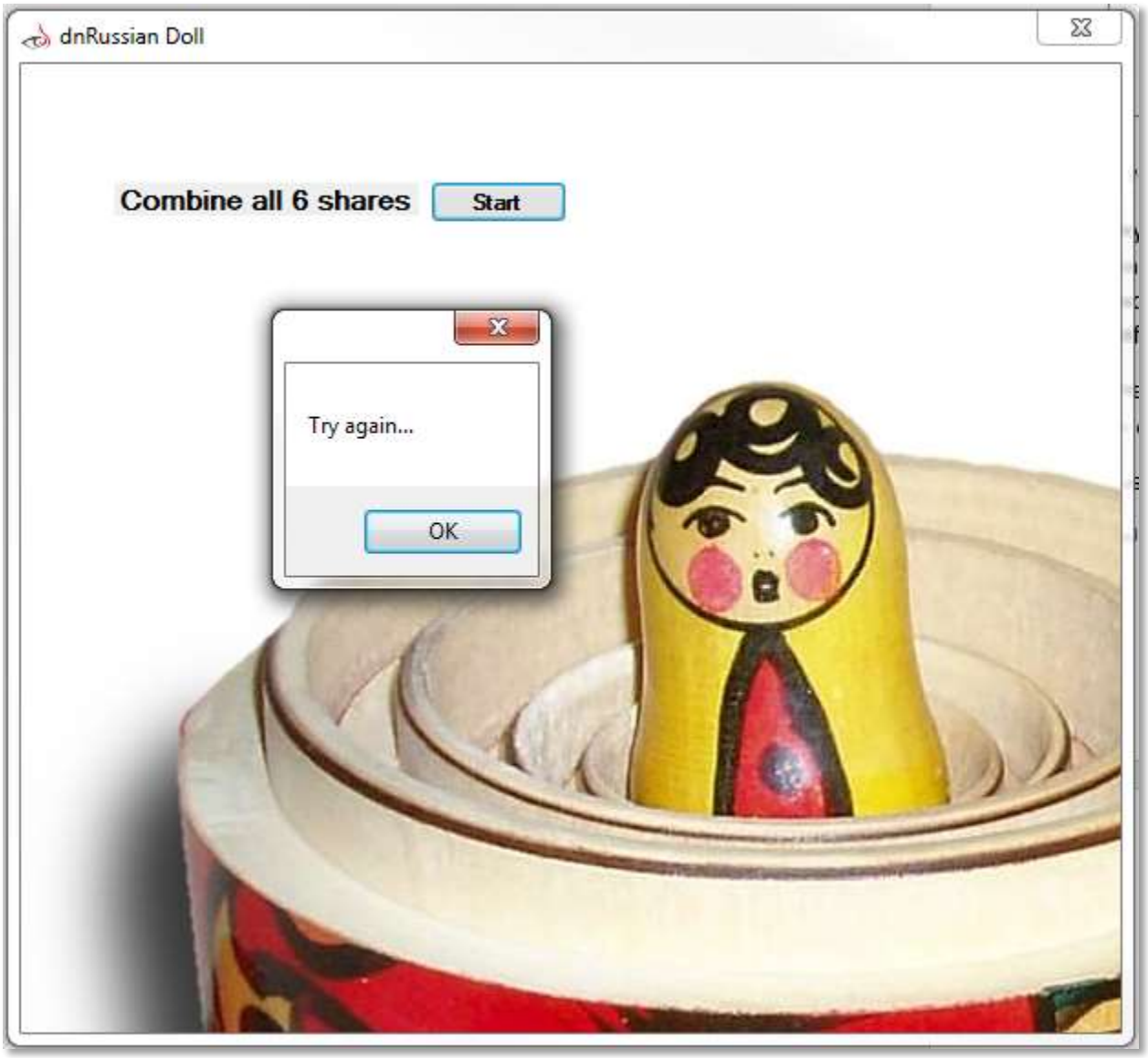

Figure 1- Main Form Fail

Simply running strings on the binary reveals some interesting things.

```
System.Reflection
KEY_For_layer1
DecompressBuffer
Share:1-d8effa9e8e19f7a2f17a3b55640b55295b1a327a5d8aebc832eae1a905c48b64
ConfusedByAttribute
GUI.Form1.resources
zkJAVxIqxkFRgAbwYbkIjRikpkgJA
layer1
layer2
layer3
share6
combine
Load
AppDomain
DeflateStream
System.IO.Compression
CompressionMode
ConfuserEx v1.0.0
```

Figure 2- Strings

Right away we see share number one in plain text. We now know the format to expect while we continue our analysis. We can also see that the thing binary has been protected with ConfuserEx v1.0.0.

The protection applied to the main GUI application was: Constant Protection (Strings) and Resource Protection. The effects of these protections are obvious when the file in loaded into dnSpy.

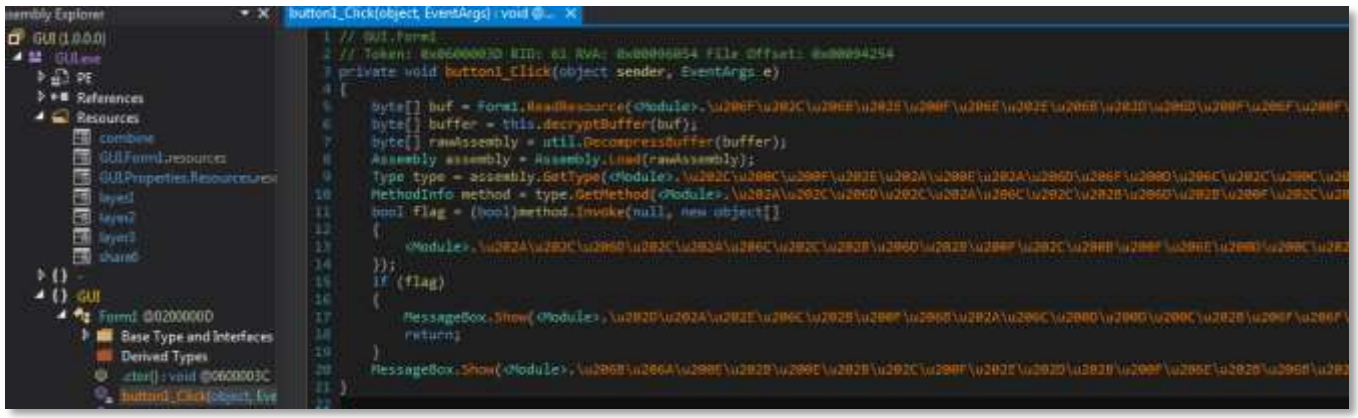

Figure 3- Main GUI dnSpy

Figure 3 shows GUI.exe loaded in dnSpy. While examining the tree view on the left side you will see all of the resources that are embedded into this binary: combine, layer1, layer2, layer3, share6. If you try

to save the resources using the dnSpy application, it fails due to the ConfuserEx protection. The constants protection can be seen on the right side pane of the application. No strings are viewable. These strings have been replaced with function calls. This is a feature of the constant protections that have been applied, but more on this later.

Now that we have the code loaded we can see that its purpose is to read a resource, decrypt it, decompress it, and dynamically load it using reflection. This is a common technique used by many .NET packers and is useful to know this paradigm.

After loaded, it invokes a method in the dynamically loaded assembly. The result of this method call is a Boolean. Based on this result it will display one of two MessageBox's to a user. We have already seen the "Try again…" message, so we will need to figure out what causes the return code of the invoked function call.

To recover the contents of the layer which is being loaded. I set a breakpoint on Assembly.Load and save the byte array to disk.

```
0:000> !bpmd mscorlib_ni System.Reflection.Assembly.Load

0:000> !CLRStack -p
OS Thread Id: 0x208c (0)
ESP       EIP
0038ead8 540f29e4 System.Reflection.Assembly.Load(Byte[])
    PARAMETERS:
        rawAssembly = 0x064c4c48

0:000> !DumpObj 0x064c4c48
Name: System.Byte[]
MethodTable: 53cb37b8
EEClass: 53a6eb8c
Size: 119308(0x1d20c) bytes
Array: Rank 1, Number of elements 119296, Type Byte
Element Type: System.Byte
Fields:
None

0:000> !DumpArray -length 1 0x064c4c48
Name: System.Byte[]
MethodTable: 53cb37b8
EEClass: 53a6eb8c
Size: 119308(0x1d20c) bytes
Array: Rank 1, Number of elements 119296, Type Byte
Element Methodtablse: 53cb3868
[0] 064c4c50

0:000> .writemem c:\Users\SomeUser\Documents\FLAREON\layer1.bin 064c4c50 L?0x1d20c
```

Figure 4- Recovery of layer assembly

The above method is using Windbg with the SOS extension. It uses the *bpmd* command to set a managed code break point on *Assembly.Load*, and it uses the *DumpObj* and *DumpArray* commands to get information object a managed code object, the byte array. It then uses the standard *.writemem* command to write the contents of memory to a file on disk.

One other thing I would like to point out at this time is how to view JIT generated code. To view the JIT generated code for button1_Click I will again use the SOS extension.

```
0:000> !Token2EE * 0x0600003D
Module: 003d2c5c (GUI.exe)
Token: 0x0600003d
MethodDesc: 003d6c84
Name: GUI.Form1.button1_Click(System.Object, System.EventArgs)
JITTED Code Address: 00653128

0:000> !U 00653128
Normal JIT generated code
GUI.Form1.button1_Click(System.Object, System.EventArgs)
Begin 00653128, size fd
>>> 00653128 55              push    ebp
00653129 8bec            mov     ebp,esp
0065312b 57              push    edi
0065312c 56              push    esi
0065312d 8bf1            mov     esi,ecx
0065312f ba00773d00      mov     edx,3D7700h (MD: <Module>.ï,gnirtS.metsyS[[ï
))23tnIU(]]bilrocsm
00653134 b90cfbce10      mov     ecx,10CEFB0Ch
00653139 e8faf7ffff      call    00652938 (<Module>.ï,nonaC__.metsyS[[ï
)60000060 :nekoTdm ,)23tnIU(]]bilrocsm
0065313e 8bc8            mov     ecx,eax
00653140 ff15746c3d00    call    dword ptr ds:[3D6C74h]
(GUI.Form1.ReadResource(System.String), mdToken: 0600003b)
00653146 8bd0            mov     edx,eax
00653148 8bce            mov     ecx,esi
0065314a e82592d8ff      call    003dc374 (GUI.Form1.decryptBuffer(Byte[]),
mdToken: 0600003e)
0065314f 8bc8            mov     ecx,eax
00653151 ff15f87d3d00    call    dword ptr ds:[3D7DF8h]
(GUI.util.DecompressBuffer(Byte[]), mdToken: 06000041)
00653157 8bc8            mov     ecx,eax
00653159 e886f8a953      call    mscorlib_ni+0x6b29e4 (540f29e4)
(System.Reflection.Assembly.Load(Byte[]), mdToken: 06001c1f)
0065315e 8bf0            mov     esi,eax
00653160 ba90763d00      mov     edx,3D7690h (MD: <Module>.ï,gnirtS.metsyS[[
))23tnIU(]]bilrocsm
00653165 b9b5351216      mov     ecx,161235B5h
0065316a e839f5ffff      call    006526a8 (<Module>.ï,nonaC__.metsyS[[
)70000060 :nekoTdm ,)23tnIU(]]bilrocsm
0065316f 8bd0            mov     edx,eax
00653171 8bce            mov     ecx,esi
00653173 8b01            mov     eax,dword ptr [ecx]
00653175 ff5060          call    dword ptr [eax+60h]
00653178 8bf0            mov     esi,eax
0065317a ba70773d00      mov     edx,3D7770h (MD: <Module>.ï,gnirtS.metsyS[[ï
))23tnIU(]]bilrocsm
0065317f b9b7419f20      mov     ecx,209F41B7h
00653184 e83ffaffff      call    00652bc8 (<Module>.ï,nonaC__.metsyS[[ï
)40000060 :nekoTdm ,)23tnIU(]]bilrocsm
```

```
00653189 8bd0            mov     edx,eax
0065318b 8bce            mov     ecx,esi
0065318d 3909            cmp     dword ptr [ecx],ecx
0065318f e8bcce6153      call    mscorlib_ni+0x230050 (53c70050)
(System.Type.GetMethod(System.String), mdToken: 06000d6a)
00653194 8bf8            mov     edi,eax
00653196 ba01000000      mov     edx,1
0065319b b9be3fa453      mov     ecx,offset mscorlib_ni+0x3fbe (53a43fbe)
006531a0 e89befd6ff      call    003c2140 (JitHelp: CORINFO_HELP_NEWARR_1_OBJ)
006531a5 8bf0            mov     esi,eax
006531a7 ba70773d00      mov     edx,3D7770h (MD: <Module>.‮,gnirtS.metsyS[[‮
))23tnIU(]]bilrocsm
006531ac b98c077411      mov     ecx,1174078Ch
006531b1 e812faffff      call    00652bc8 (<Module>.‮,nonaC__.metsyS[[‮
)40000060 :nekoTdm ,)23tnIU(]]bilrocsm
006531b6 50              push    eax
006531b7 8bce            mov     ecx,esi
006531b9 33d2            xor     edx,edx
006531bb e89c61f953      call    mscorwks!JIT_Stelem_Ref (545e935c)
006531c0 6a00            push    0
006531c2 6a00            push    0
006531c4 56              push    esi
006531c5 6a00            push    0
006531c7 8bcf            mov     ecx,edi
006531c9 33d2            xor     edx,edx
006531cb 8b01            mov     eax,dword ptr [ecx]
006531cd ff9098000000    call    dword ptr [eax+98h]
006531d3 8bf0            mov     esi,eax
006531d5 813e1448c853    cmp     dword ptr [esi],offset mscorlib_ni+0x244814
(53c84814)
006531db 740c            je      006531e9
006531dd 8bd6            mov     edx,esi
006531df b91448c853      mov     ecx,offset mscorlib_ni+0x244814 (53c84814) (MT:
System.Boolean)
006531e4 e8e01e0d54      call    mscorwks!JIT_Unbox (547250c9)
006531e9 0fb64604        movzx   eax,byte ptr [esi+4]
006531ed 85c0            test    eax,eax
006531ef 7418            je      00653209
006531f1 bae0773d00      mov     edx,3D77E0h (MD: <Module>.‮,gnirtS.metsyS[[
))23tnIU(]]bilrocsm
006531f6 b98fdececd      mov     ecx,0CDCEDE8Fh
006531fb e858fcffff      call    00652e58 (<Module>.‮,nonaC__.metsyS[[
)80000060 :nekoTdm ,)23tnIU(]]bilrocsm
00653200 8bc8            mov     ecx,eax
00653202 e851d24e52      call    System_Windows_Forms_ni+0x7f0458 (52b40458)
(System.Windows.Forms.MessageBox.Show(System.String), mdToken: 060048b3)
00653207 eb16            jmp     0065321f
00653209 ba18763d00      mov     edx,3D7618h (MD: <Module>.‮,gnirtS.metsyS[[
))23tnIU(]]bilrocsm
0065320e b96d89561b      mov     ecx,1B56896Dh
```

```
00653213 e800eeffff      call    00652018 (<Module>.I,nonaC__.metsyS[[
)50000060 :nekoTdm ,)23tnIU(]]bilrocsm
00653218 8bc8            mov     ecx,eax
0065321a e839d24e52      call    System_Windows_Forms_ni+0x7f0458 (52b40458)
(System.Windows.Forms.MessageBox.Show(System.String), mdToken: 060048b3)
0065321f 5e              pop     esi
00653220 5f              pop     edi
00653221 5d              pop     ebp
00653222 c20400          ret     4
```

Figure 5 - JIT Generated Code

It is not important to read all the generated code above. What is important to get out of the above output is that it is possible view the disassembly for JIT code. I've highlighted all of the important function calls and you can see how this matches up with our previous knowledge of the button1_Click function from dnSpy. This method for view JIT disassembly is a use tool that we will use later in our analysis. Also the note is the commands used in the above example. *Token2EE* is a SOS extension command that is used to convert a token into a method description, and *U* will display an annotated disassembly of the method body.

LAYER1

We can now load the Layer1 dll, which we saved during our prior analysis. It has been protected with 3 additional protections: Invalid metadata, Anti ildasm, and Anti tamper. The results of this makes dnSpy sad ☹, as shown in Figure 6.

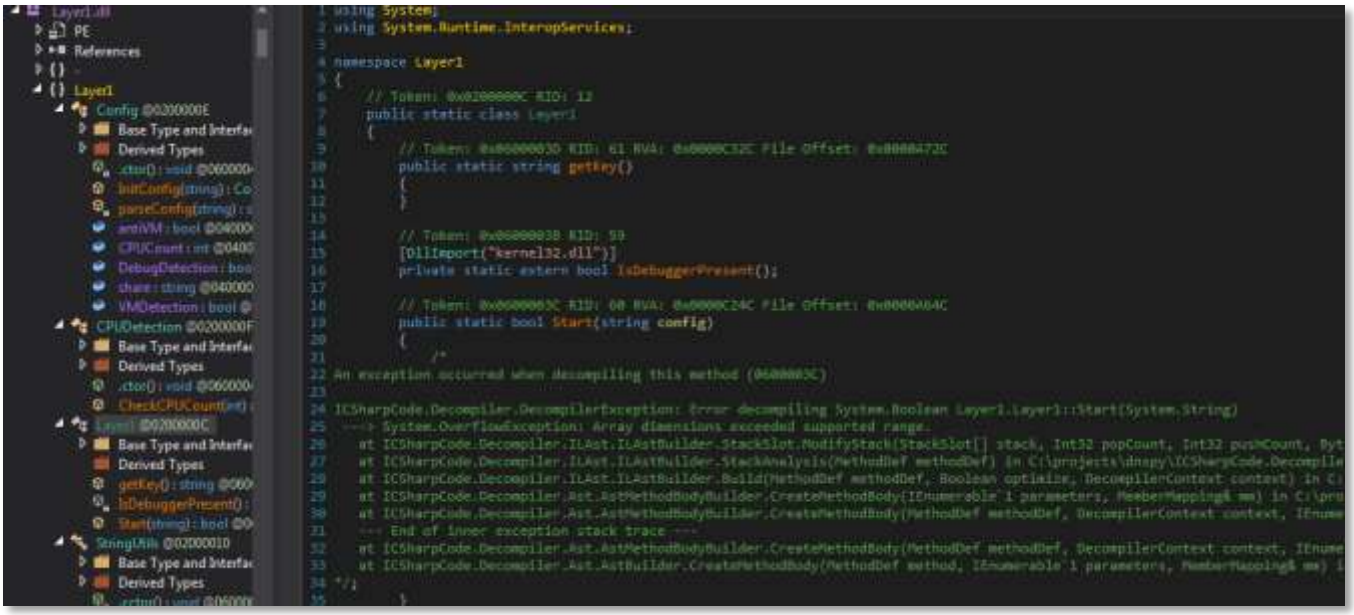

Figure 6 - Layer1 Protected

The Anti tamper protection encrypts all the IL opcodes so that static analysis fails. These opcodes are decrypted at runtime during the modules class constructor.

```
static <Module>()
```

Figure 7 - Module class constructor

One way to approach this is by debugging the JIT generated code. When I showed how we could view the disassembly of the button1_Click method in the MAIN FORM section there was a call to *GetMethod*:

```
0065318f e8bcce6153      call    mscorlib_ni+0x230050 (53c70050)
(System.Type.GetMethod(System.String), mdToken: 06000d6a)
```

Figure 8 - GetMethod call

If you set a breakpoint on the address immediately following the call to *GetMethod*, we will have the code in a state in which it has been loaded and decrypted. At this point we can use the *Token2EE* command to find the JIT address of the "*Layer1.Start*" Method.

```
0:000> !Token2EE * 0x0600003C
Module: 02375f00 (Layer1, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null)
Token: 0x0600003c
MethodDesc: 02376354
Name: Layer1.Layer1.Start(System.String)
Not JITTED yet. Use !bpmd -md 02376354 to break on run.

0:000> !bpmd -md 02376354
MethodDesc = 02376354
Adding pending breakpoints...
0:000> g
JITTED Layer1!Layer1.Layer1.Start(System.String)
Setting breakpoint: bp 04AFF848 [Layer1.Layer1.Start(System.String)]
Breakpoint 9 hit
0:000> !CLRStack -p
OS Thread Id: 0x208c (0)
ESP       EIP
0038e584 04aff848 Layer1.Layer1.Start(System.String)
    PARAMETERS:
        config = 0x02830664

0:000> !DumpObj 0x02830664
Name: System.String
MethodTable: 53cb0d48
EEClass: 53a6d66c
Size: 226(0xe2) bytes
 (C:\windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll)
String: no/-|-\no/-|-\no/-|-\2/-|-\shareShare:2-
f81ae6f5710cb1340f90cd80d9c33107a1469615bf299e6057dea7f4337f67a3
Fields:
      MT    Field   Offset                 Type VT     Attr    Value Name
53cb2f94  4000096        4         System.Int32  1 instance      105 m_arrayLength
53cb2f94  4000097        8         System.Int32  1 instance      104 m_stringLength
53cb1a28  4000098        c          System.Char  1 instance       6e m_firstChar
53cb0d48  4000099       10        System.String  0   shared   static Empty
    >> Domain:Value  0042c920:02811198 <<
53cb1978  400009a       14        System.Char[]  0   shared   static
WhitespaceChars
    >> Domain:Value  0042c920:02811748 <<
```

Figure 9 - Start Method Config

Figure 9 showed the method of setting a breakpoint on the start method. Once this breakpoint was hit I viewed the CLR stack by running the "*CLRStack –p*" command. This showed the address of the config object, which was passed in. I then dumped the objects contents using the *DumpObj* command. This config object is a string, which contains Share number 2.

Figure 10 shows the JITd code for the Layer1 Start method. I've greatly simplified this output just showing the call instructions. From this alone we can get a good idea of the code flow. It calls the

methods: *InitConfig*, *get_ProcessorCount*, *IsDebuggerPresent*, *Layer1.getKey*, *AES_Decrypt*, *DecompressBuffer*, *Assembly.Load*, *GetMethod*, and *Invoke*. Although it is not displayed below, this start method returns the Boolean it receives from the invocation of Layer1 back to the main GUI form.

```
0:000> !U eip
Normal JIT generated code
Layer1.Layer1.Start(System.String)
Begin 04aff848, size 165
04aff864 ff15c8713702    call    dword ptr ds:[23771C8h]
(Layer1.Config.InitConfig(System.String), mdToken: 06000043)
04aff87d e89ec7164f      call    mscorlib_ni+0x22c020 (53c6c020)
(System.Environment.get_ProcessorCount(), mdToken: 06000a70)
04aff8a3 e8a0d18dfb      call    003dca48 (Layer1.Layer1.IsDebuggerPresent(),
mdToken: 0600003b)
04aff8b6 ff1568633702    call    dword ptr ds:[2376368h] (Layer1.Layer1.getKey(),
mdToken: 0600003d)
04aff8c9 ff15bc723702    call    dword ptr ds:[23772BCh] (<Module>.†
)40000060 :nekoTdm ,)23tnIU(]]bilrocsm ,nonaC__.metsyS[[†
04aff8d1 ff1524733702    call    dword ptr ds:[2377324h]
(Layer1.util.ReadResource(System.String), mdToken: 0600003f)
04aff8d9 e822c30f4f      call    mscorlib_ni+0x1bbc00 (53bfbc00)
(System.Text.Encoding.get_UTF8(), mdToken: 060028cd)
04aff8e5 ff908c000000    call    dword ptr [eax+8Ch]
04aff8ef ff1518733702    call    dword ptr ds:[2377318h]
(Layer1.util.AES_Decrypt(Byte[], Byte[]), mdToken: 0600003e)
04aff8f7 ff1530733702    call    dword ptr ds:[2377330h]
(Layer1.util.DecompressBuffer(Byte[]), mdToken: 06000040)
04aff8ff e8e0305f4f      call    mscorlib_ni+0x6b29e4 (540f29e4)
(System.Reflection.Assembly.Load(Byte[]), mdToken: 06001c1f)
04aff910 ff15ac733702    call    dword ptr ds:[23773ACh] (<Module>.†
)60000060 :nekoTdm ,)23tnIU(]]bilrocsm ,nonaC__.metsyS[[
04aff91c ff5060          call    dword ptr [eax+60h]
04aff92b ff15bc723702    call    dword ptr ds:[23772BCh] (<Module>.†
)40000060 :nekoTdm ,)23tnIU(]]bilrocsm ,nonaC__.metsyS[[†
04aff937 e81407174f      call    mscorlib_ni+0x230050 (53c70050)
(System.Type.GetMethod(System.String), mdToken: 06000d6a)
04aff949 e8f2278cfb      call    003c2140 (JitHelp: CORINFO_HELP_NEWARR_1_OBJ)
04aff95a ff15bc723702    call    dword ptr ds:[23772BCh] (<Module>.†
)40000060 :nekoTdm ,)23tnIU(]]bilrocsm ,nonaC__.metsyS[[†
04aff965 e8f299ae4f      call    mscorwks!JIT_Stelem_Ref (545e935c)
04aff978 ff9098000000    call    dword ptr [eax+98h]
04aff98f e83557c24f      call    mscorwks!JIT_Unbox (547250c9)
04aff99e e89122ae4f      call    mscorwks!JIT_EndCatch (545e1c34)
```

Figure 10 - Layer1.Start

We need to make sure our system has 2 or more CPUs and doesn't return true if a call is made to *IsDebuggerPresent*. There is also a call to *Layer1.getKey()*. It appears that the result of this method will

be used as a key for the *AES_Decrypt* method.

When analyzing the JIT code of the Layer1.getKey method. We can see some interesting function calls and strings references. The following calls are made: *System.IO.Directory.GetDirectories*, *System.Security.Cryptography.MD5.Create*, *System.Security.Cryptography.HashAlgorithm.ComputeHash*, and *System.Convert.ToBase64String*.

This is followed by a line that has a reference to a base64 string.

```
04afd48a 8b359c314e06    mov     esi,dword ptr ds:[64E319Ch]
("UtYSc3XYLz4wCCfrR5ssZQ==")
```

Figure 11 - Base64 string

The base64 string above when converted to binary is: **52d6127375d82f3e300827eb479b2c65**

If you google that hash you will find results for the word "sharing". Based on the function call context that is close to this hash, it appears to be enumerating directories looking for the name "sharing".

After creating the directory "sharing" in the current working directory for the process. I received an additional breakpoint event for *Assembly.Load*.

```
0:000> !CLRStack -p
OS Thread Id: 0x208c (0)
ESP       EIP
0038e554 540f29e4 System.Reflection.Assembly.Load(Byte[])
    PARAMETERS:
        rawAssembly = 0x09d41128

0038e558 06c7242c Layer1.Layer1.Start(System.String)
    PARAMETERS:
        config = <no data>
```

Figure 12 - Saving Layer2

As Figure 12 shows above, a call to *Assembly.Load* is being made from Layer1. This means that we have successfully passed this layer. We can use the same method described above to save this layer to a file on disk.

LAYER2

After loading the layer2 assembly into dnSpy you will notice most of the same method names from the previous level. The only change from a ConfuserEx perspective is the addition of the control flow

protection.

Performing an analysis of the Layer2.getKey method we again see some interesting function calls and strings references. The following calls are made: *Microsoft.Win32.RegistryKey.GetSubKeyNames()*, *System.Security.Cryptography.MD5.Create*, *System.Security.Cryptography.HashAlgorithm.ComputeHash*, and *System.Convert.ToBase64String*.

The strings reference again is a base64 string. "Xr4ilOzQ4PCOq3aQ0qbuaQ=="

The base64 string above when converted to binary is: **5ebe2294ecd0e0f08eab7690d2a6ee69**

If you Google that hash you will find results for the word "secret". Based on the function call context that is close to this hash, it appears to be enumerating directories looking for the name "secret".

At this point I'll use procmon to see which registry subkeys the challenge is enumerating as shown in Figure 13.

RegQueryKey	HKCU	SUCCESS	Query: Cached, SubKeys: 12, Values: 0
RegEnumKey	HKCU	SUCCESS	Index: 0, Name: AppEvents
RegEnumKey	HKCU	SUCCESS	Index: 1, Name: Console
RegEnumKey	HKCU	SUCCESS	Index: 2, Name: Control Panel
RegEnumKey	HKCU	SUCCESS	Index: 3, Name: Environment
RegEnumKey	HKCU	SUCCESS	Index: 4, Name: EUDC
RegEnumKey	HKCU	SUCCESS	Index: 5, Name: Identities
RegEnumKey	HKCU	SUCCESS	Index: 6, Name: Keyboard Layout
RegEnumKey	HKCU	SUCCESS	Index: 7, Name: Network
RegEnumKey	HKCU	SUCCESS	Index: 8, Name: Printers
RegEnumKey	HKCU	SUCCESS	Index: 9, Name: Software
RegEnumKey	HKCU	SUCCESS	Index: 10, Name: System
RegEnumKey	HKCU	SUCCESS	Index: 11, Name: Volatile Environment

Figure 13 - Registry Subkeys

After creating the subkey "secret" in HKCU, I received an additional breakpoint event for Assembly.Load.

```
0:000> !CLRStack -p
OS Thread Id: 0x208c (0)
ESP       EIP
0038dfc0 540f29e4 System.Reflection.Assembly.Load(Byte[])
    PARAMETERS:
        rawAssembly = 0x03ad5628

0038dfc4 06df17bb Layer2.Layer2.Start(System.String)
    PARAMETERS:
        config = <no data>
```

Figure 14 - Raw assembly for Layer3

At this point we can again use the same technique to save this layer to disk using the Windbg command *.writemem.*

NOTE: Layer2 contains an additional anti-emulation check by calling the method *IsVideoCardFromEmulator()*. This method returns a Boolean and can be trivially be bypassed by changing the return value of EAX from 1 to 0 while using a debugger.

LAYER3

Layer3 in addition to all the other protections already mentioned, has the additional protections of "Anti Debug" and 'Anti Dump". These additional protections make the showing the disassembly as we did in layer1 and layer2 more complicated.

Following the approach from the previous layers, we set a breakpoint on the Layer3.Start method using *!bpmd*. Unfortunately, when we run our debugger, it never actually breaks. This is due to the additional protections. Instead of relying on *!bpmd*, we can set a breakpoint before the IL code is compiled to native code, wait for it to be compiled, and set a breakpoint on the newly compiled native code. The *mscorjit!CILJit::compileMethod* method is responsible for converting IL code to native code. The method is defined as:

```
    virtual CorJitResult __stdcall compileMethod (
            ICorJitInfo                 *comp,              /* IN */
            struct CORINFO_METHOD_INFO  *info,              /* IN */
            unsigned /* code:CorJitFlag */   flags,         /* IN */
            BYTE                        **nativeEntry,      /* OUT */
            ULONG                       *nativeSizeOfCode   /* OUT */
            ) = 0;
```

Figure 15 - mscorjit!CILJit::compileMethod definition

After the IL code is compiled to native code, it is copied to the *nativeEntry* variable. Our new plan is to set a breakpoint on the *compileMethod* function and then following this into the compiled native code. When the debugger stops at the *compileMethod* breakpoint, we use the *!CLRStack* command to show the method that is about to be compiled.

```
0:000> !CLRStack
OS Thread Id: 0x1408 (0)
ESP       EIP
0012d858 79065dbb [PrestubMethodFrame: 0012d858] <Module>..cctor()
```

Figure 16 - Before .cctor is compiled

We see the first time the *compileMethod* breakpoint is hit, the *.cctor* method for layer3 is about to be compiled. At the breakpoint, we take note of the *nativeEntry* location on the stack. This is where the native code will be copied to after it is compiled. We step through this function until the return and look at the *nativeEntry* and see x86 code. We set a breakpoint at this address and run the debugger. The debugger now breaks at the beginning of the *.cctor* method. We remove the *compileMethod* breakpoint and step through this function as we don't want to waste our time in the .cctor method. We again set a breakpoint on the *compileMethod* function and run the debugger. The next time this breakpoint is hit, we run *!CLRStack* to see which method is about to compile. In this case, it's mdToken 0600003f. This is the metadata token for the "Layer3.Start" method. Although with the extra obfuscation layer provided by ConfuserEx, we are unable to see the method name in the *!CLRStack* output. We again wait until the IL code is compiled and set a breakpoint on the native code as described earlier and run the debugger. We are now at the beginning of the Layer3.Start method. We run *!U eip* to see the disassembly for this method. Looking only at the method calls, we see the following:

```
0256b212 ff1584d8b201    call    dword ptr ds:[1B2D884h] (UNKNOWN, mdToken:
06000040)
0256b24c ff1548e9b201    call    dword ptr ds:[1B2E948h] (UNKNOWN, mdToken:
06000008)
0256b254 ff15c0e8b201    call    dword ptr ds:[1B2E8C0h] (UNKNOWN, mdToken:
06000043)
0256b26b e8349c4578      call    System_ni+0x584ea4 (7a9c4ea4)
(System.Diagnostics.Process.Start(System.Diagnostics.ProcessStartInfo), mdToken:
06003aa6)
0256b287 e8d4f2ffff      call    0256a560 (UNKNOWN, mdToken: 06000009)
0256b28f e8ec7c9077      call    mscorwks!JIT_WriteBarrierEAX (79e72f80)
0256b2ae ff1558e8b201    call    dword ptr ds:[1B2E858h] (UNKNOWN, mdToken:
0600000b)
0256b2b6 ff15c0e8b201    call    dword ptr ds:[1B2E8C0h] (UNKNOWN, mdToken:
06000043)
0256b2d9 e882f2ffff      call    0256a560 (UNKNOWN, mdToken: 06000009)
0256b2e3 e830b22b77      call    mscorlib_ni+0x766518 (79826518)
(System.IO.File.WriteAllBytes(System.String, Byte[]), mdToken: 06003549)
0256b2ed e82a6ddcfd      call    0033201c (JitHelp: CORINFO_HELP_NEWSFAST)
0256b2fe e8d5faffff      call    0256add8 (UNKNOWN, mdToken: 0600000c)
0256b307 e814714978      call    System_ni+0x5c2420 (7aa02420)
(System.Diagnostics.ProcessStartInfo..ctor(System.String), mdToken: 06003b03)
0256b31c e8df08d176      call    mscorlib_ni+0x1bbc00 (7927bc00)
(System.Text.Encoding.get_UTF8(), mdToken: 060028c1)
0256b328 ff908c000000    call    dword ptr [eax+8Ch]
0256b332 ff15b4e8b201    call    dword ptr ds:[1B2E8B4h] (UNKNOWN, mdToken:
06000042)
0256b355 e806f2ffff      call    0256a560 (UNKNOWN, mdToken: 06000009)
0256b35f e8b4b12b77      call    mscorlib_ni+0x766518 (79826518)
(System.IO.File.WriteAllBytes(System.String, Byte[]), mdToken: 06003549)
```

Figure 17 - Layer3.Start calls

Looking at the disassembly, most methods are referenced only by their metadata token. The *getKey* metadata token is 06000040. We see that is the first call.

We repeat the process of breaking on the *compileMethod* function until at the beginning of the *getKey* method. We may have to keep running until the breakpoint at the end of the *compileMethod* shows the metadata token 06000040.

Looking through this code, we see MD5 hashing functions and a string: "KTUxM5ElLBtBBAdJXNCW/g==".

Near this code we see a WMI query, but we don't know what is being searched. We set a breakpoint at the following line to identify the search string:

```
027d4f7b e898a6d564      call    System_Management_ni+0x1f618 (6752f618)
(System.Management.SelectQuery..ctor(System.String), mdToken: 060001f2)
```

Figure 18 - SelectQuery dissassembly line

Once we break before this call, we dump the parameter to this method:

```
0:000> !DumpObj edx
Name: System.String
MethodTable: 79330b70
EEClass: 790ed66c
Size: 52(0x34) bytes
 (C:\Windows\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll)
String: Win32_UserAccount
```

Figure 19 - Parameter for the WMI query

We see the WMI query is for user accounts. Now, we go back to the string we found earlier:

"KTUxM5ElLBtBBAdJXNCW/g==". When base64 decoded, and converted to hex we see the following MD5:

2935313391252c1b410407495cd096fe

When Googling this has we can see that it is looking for a username called "shamir" must exist. We create an account with the username shamir and run the debugger. At this point the challenge should drop two files to the working directory. share6-decoded.png and ssss-combine.exe.

SHARE6

This is just a plain png file that contains the contents of share number 6.

Figure 20 - Share6 png

COMBINE

ssss-combine.exe is a windows executable which can be used to combine all of the shares found throughout the challenge to produce the key. The original prototype of this challenge did not include this binary. However during initial testing it was found that there were multiple implementations of the sharmir secure sharing algorithm and they are not compatible. Therefore we made the decision to include a binary that would be able to combine the shares, as the main purpose of the challenge was to

learn and practice .Net analysis and debugging.

At this point we have only showed how to get shares 1,2, and 6. The other shares were hidden in unused strings, which in turn were protected by the ConfuserEx protector. The easiest way to recover these strings is to search memory of the target process for “Share:”

```
0:012> s -a 0 l?80000000 "Share:"
0049a82f  53 68 61 72 65 3a 31 2d-64 38 65 66 66 61 39 65  Share:1-d8effa9e
02b4227c  53 68 61 72 65 3a 32 2d-66 38 31 61 65 36 66 35  Share:2-f81ae6f5
02b9b3c8  53 68 61 72 65 3a 33 2d-35 32 33 63 62 35 63 32  Share:3-523cb5c2
02ba0114  53 68 61 72 65 3a 34 2d-30 34 62 35 38 66 62 64  Share:4-04b58fbd
02bed05c  53 68 61 72 65 3a 35 2d-35 38 38 38 37 33 33 37  Share:5-58887337
```

Figure 21 - Share strings in memory

```
C:\Users\SomeUser\Documents\FLAREON>ssss-combine.exe -t 6
Shamir Secret Sharing Scheme - $Id$
Copyright 2005 B. Poettering, Win32 port by Alex.Popov@leggettwood.com

Enter 6 shares separated by newlines:
Share [1/6]: 1-d8effa9e8e19f7a2f17a3b55640b55295b1a327a5d8aebc832eae1a905c48b64
Share [2/6]: 2-f81ae6f5710cb1340f90cd80d9c33107a1469615bf299e6057dea7f4337f67a3
Share [3/6]: 3-523cb5c21996113beae6550ea06f5a71983efcac186e36b23c030c86363ad294
Share [4/6]: 4-04b58fbd216f71a31c9ff79b22f258831e3e12512c2ae7d8287c8fe64aed54cd
Share [5/6]: 5-5888733744329f95467930d20d701781f26b4c3605fe74eefa6ca152b450a5d3
Share [6/6]: 6-a003fcf2955ced997c8741a6473d7e3f3540a8235b5bac16d3913a3892215f0a
Resulting secret: Shamir_1s_C0nfused@flare-on.com
```

Figure 22 - Solution

Conclusion

The goals for this challenge were to help improve .NET malware analysis skills by exposing challengers to the ConfuserEx software protector. Since this software is open source, many .Net software protectors have similar concepts and implementations. I was happy to see the high percentage of people solved this challenge. I guess this means it was too easy?

References

https://yck1509.github.io/ConfuserEx/

https://en.wikipedia.org/wiki/Secret_sharing

https://msdn.microsoft.com/en-us/library/bb190764(v=vs.110).aspx