Dissecting ConfuserEx - Reference proxies

Written for http://www.rtn-team.cc/ by ubbelol

In this paper I'll cover how the Reference Proxy protection works in ConfuserEx. There's a few different "modes" that this protection can be ran in, but I'll start by covering the easiest "Mild" mode.

Table of contents:

Tools used:

- WinDbg with SOS extension
- CFF Explorer
- Reflector
- ILSpy
- SimpleAssemblyExplorer

Target files:

- Mild: http://www.multiupload.nl/2JYU3CQQY0
- Strong: http://www.multiupload.nl/GCBXL6IVV8
- x86: http://www.multiupload.nl/IAIF3NCAFJ

Introduction

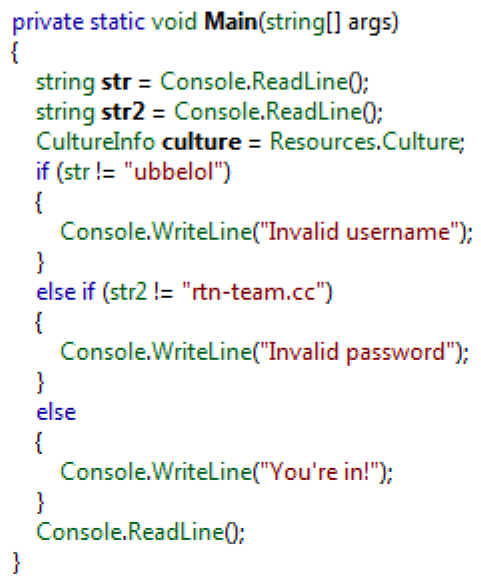
This protection is deployed in order to hide calls to different methods referenced in the assembly. It makes it much harder to follow the flow of a method by obscuring calls so that you can't directly understand them by looking at the code.

If you're interested in this kind of stuff, feel free to check out my blog where I'll try to put up more content similar to this: http://ubbecode.wordpress.com/

Mild mode

When this protection is ran in “mild” mode it simply analyzes your code and finds parts of it that can be turned into static methods. When it finds suitable code it rips that straight out of the original method and puts it into a new one. Here’s an example of what protected code can look like:

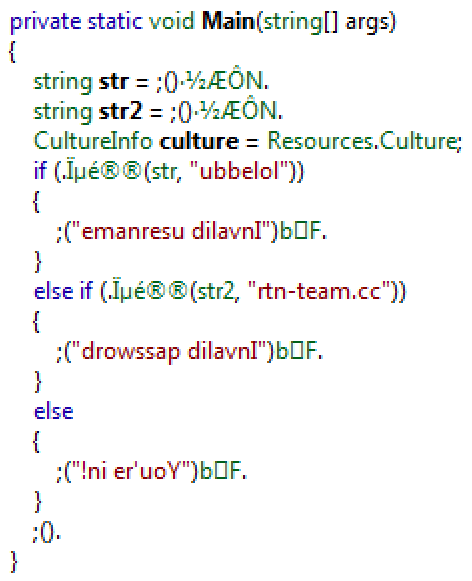
Before:



```
private static void Main(string[] args)
{
   string str = Console.ReadLine();
   string str2 = Console.ReadLine();
   CultureInfo culture = Resources.Culture;
   if (str != "ubbelol")
   {
      Console.WriteLine("Invalid username");
   }
   else if (str2 != "rtn-team.cc")
   {
      Console.WriteLine("Invalid password");
   }
   else
   {
      Console.WriteLine("You're in!");
   }
   Console.ReadLine();
}
```

$\longrightarrow$

After:



```
private static void Main(string[] args)
{
   string str = ;()·½ÆÔN.
   string str2 = ;()·½ÆÔN.
   CultureInfo culture = Resources.Culture;
   if (.Ĩµé®®(str, "ubbelol"))
   {
      ;("emanresu dilavnI")b□F.
   }
   else if (.Ĩµé®®(str2, "rtn-team.cc"))
   {
      ;("drowssap dilavnI")b□F.
   }
   else
   {
      ;("!ni er'uoY")b□F.
   }
   ;().
}
```

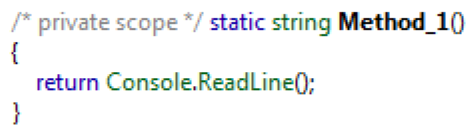
It’s quite easy to tell what’s going on when you have the original code right beside the obfuscated one, and even just following the calls in the obfuscated methods can give you a good understand on what’s actually happening. But it takes a significantly longer time to analyze methods, especially big ones. If we look at the top of the method we can see that

string str = Console.ReadLine();

is turned into

string str = Method_1();

Let’s follow Method_1() and see what it looks like:



```
/* private scope */ static string Method_1()
{
   return Console.ReadLine();
}
```

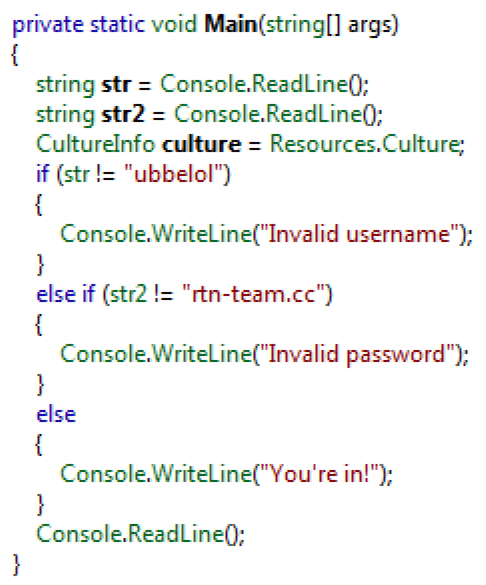
Reversing this obfuscation is trivial. Simply replace the call to Method_1 with the body of Method_1. Do this for every “proxy” and you’ll get the original body. De4dot is capable of automatically finding these static methods and inlining them, so there’s no need to cover it any further. Let’s move on to the “strong” mode.

Strong mode

Normal encoding

This mode is a bit more interesting. This uses delegates and a few tricks in order to hide the call. Let’s take a look at what the same method looks like with strong proxies:

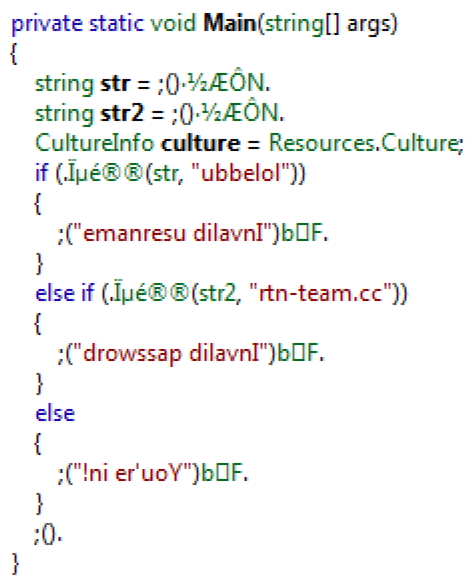
Before:



```
private static void Main(string[] args)
{
    string str = Console.ReadLine();
    string str2 = Console.ReadLine();
    CultureInfo culture = Resources.Culture;
    if (str != "ubbelol")
    {
        Console.WriteLine("Invalid username");
    }
    else if (str2 != "rtn-team.cc")
    {
        Console.WriteLine("Invalid password");
    }
    else
    {
        Console.WriteLine("You're in!");
    }
    Console.ReadLine();
}
```

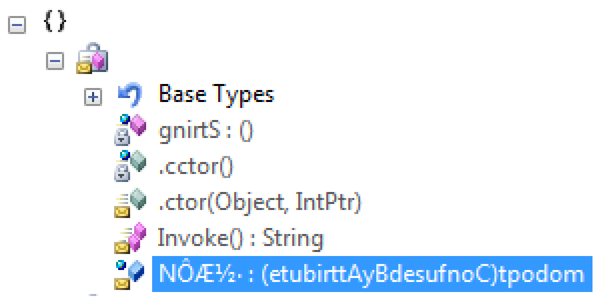
→

After:



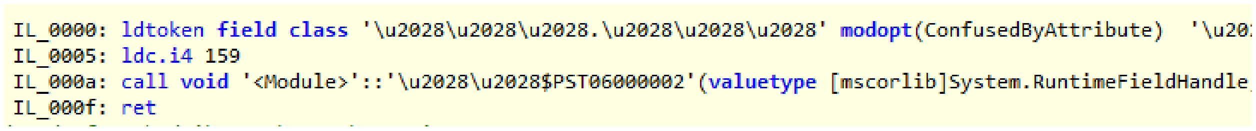
```
private static void Main(string[] args)
{
    string str = ;()·½ÆÔN.
    string str2 = ;()·½ÆÔN.
    CultureInfo culture = Resources.Culture;
    if (.Ĭµé®®(str, "ubbelol"))
    {
        ;("emanresu dilavnI")b□F.
    }
    else if (.Ĭµé®®(str2, "rtn-team.cc"))
    {
        ;("drowssap dilavnI")b□F.
    }
    else
    {
        ;("!ni er'uoY")b□F.
    }
    ;().
}
```

But if we try to follow the first call, we’ll just get to a type that looks like this:



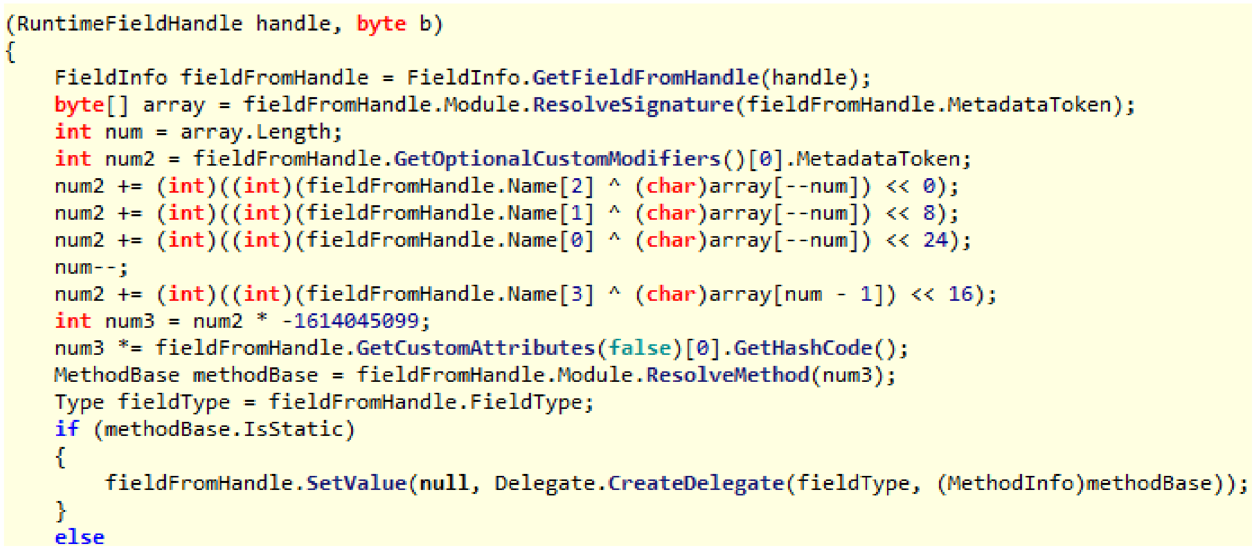
```
{}
    Base Types
    gnirtS : ()
    .cctor()
    .ctor(Object, IntPtr)
    Invoke() : String
    NÔÆ½· : (etubirttAyBdesufnoC)tpodom
```

This doesn’t give us much information at all. We still have no clue what the original call was. But we can see that the type has a static constructor (.cctor) which will be executed before anything else in that type. Let’s take a look at the code of .cctor:



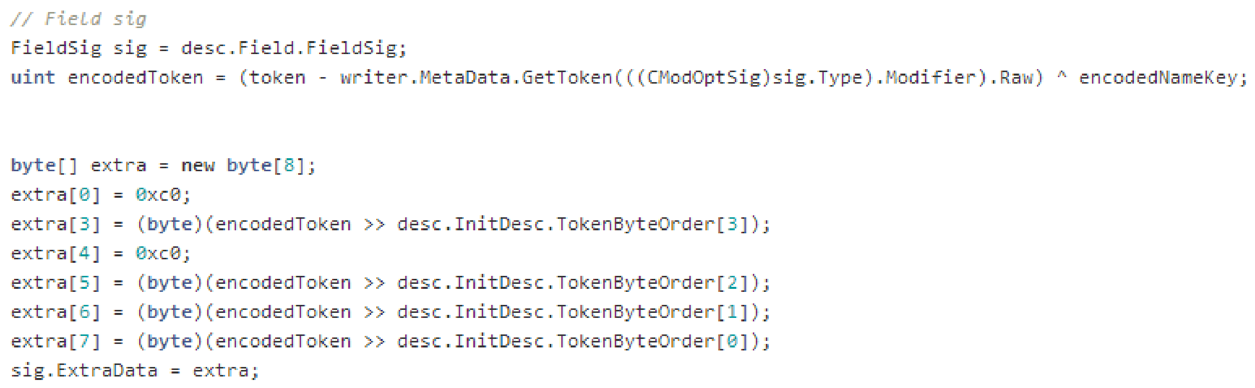
```
IL_0000: ldtoken field class '\u2028\u2028\u2028.\u2028\u2028\u2028' modopt(ConfusedByAttribute)  '\u20
IL_0005: ldc.i4 159
IL_000a: call void '<Module>'::'\u2028\u2028$PST06000002'(valuetype [mscorlib]System.RuntimeFieldHandle
IL_000f: ret
```

It loads the field we just looked at, loads a int32 onto the stack and calls a method in <Module>. Let’s follow the call:



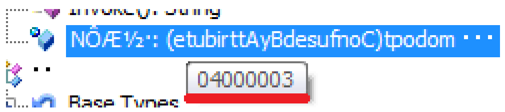
```
(RuntimeFieldHandle handle, byte b)
{
    FieldInfo fieldFromHandle = FieldInfo.GetFieldFromHandle(handle);
    byte[] array = fieldFromHandle.Module.ResolveSignature(fieldFromHandle.MetadataToken);
    int num = array.Length;
    int num2 = fieldFromHandle.GetOptionalCustomModifiers()[0].MetadataToken;
    num2 += (int)((int)(fieldFromHandle.Name[2] ^ (char)array[--num]) << 0);
    num2 += (int)((int)(fieldFromHandle.Name[1] ^ (char)array[--num]) << 8);
    num2 += (int)((int)(fieldFromHandle.Name[0] ^ (char)array[--num]) << 24);
    num--;
    num2 += (int)((int)(fieldFromHandle.Name[3] ^ (char)array[num - 1]) << 16);
    int num3 = num2 * -1614045099;
    num3 *= fieldFromHandle.GetCustomAttributes(false)[0].GetHashCode();
    MethodBase methodBase = fieldFromHandle.Module.ResolveMethod(num3);
    Type fieldType = fieldFromHandle.FieldType;
    if (methodBase.IsStatic)
    {
        fieldFromHandle.SetValue(null, Delegate.CreateDelegate(fieldType, (MethodInfo)methodBase));
    }
    else
```

At first glance this looks really confusing. If we look at the first line, we see that it’s retrieveing a field handle from the field passed from the caller. It then reads the #Blob signature of the field, and puts it in a byte array. Usually the signature of the field is just a basic description of the field in order for the runtime to read it properly. But ConfuserEx adds some of it’s own data there. It encodes a metadata token into the signature, right after the original signature.

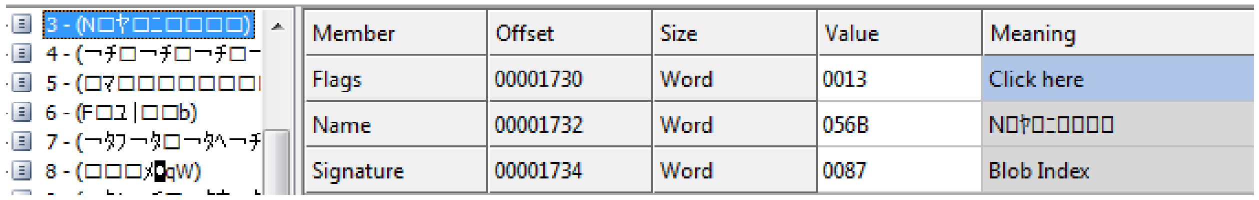


```
// Field sig
FieldSig sig = desc.Field.FieldSig;
uint encodedToken = (token - writer.MetaData.GetToken(((CModOptSig)sig.Type).Modifier).Raw) ^ encodedNameKey;


byte[] extra = new byte[8];
extra[0] = 0xc0;
extra[3] = (byte)(encodedToken >> desc.InitDesc.TokenByteOrder[3]);
extra[4] = 0xc0;
extra[5] = (byte)(encodedToken >> desc.InitDesc.TokenByteOrder[2]);
extra[6] = (byte)(encodedToken >> desc.InitDesc.TokenByteOrder[1]);
extra[7] = (byte)(encodedToken >> desc.InitDesc.TokenByteOrder[0]);
sig.ExtraData = extra;
```

If we look up the signature passed to the method with SimpleAssemblyExplorer to get the token:

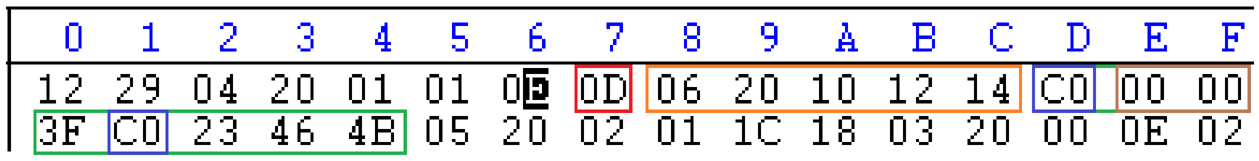


We see it's in row 3 of the Field table.



Member	Offset	Size	Value	Meaning
Flags	00001730	Word	0013	Click here
Name	00001732	Word	056B	N□ҿ□□□□□□
Signature	00001734	Word	0087	Blob Index

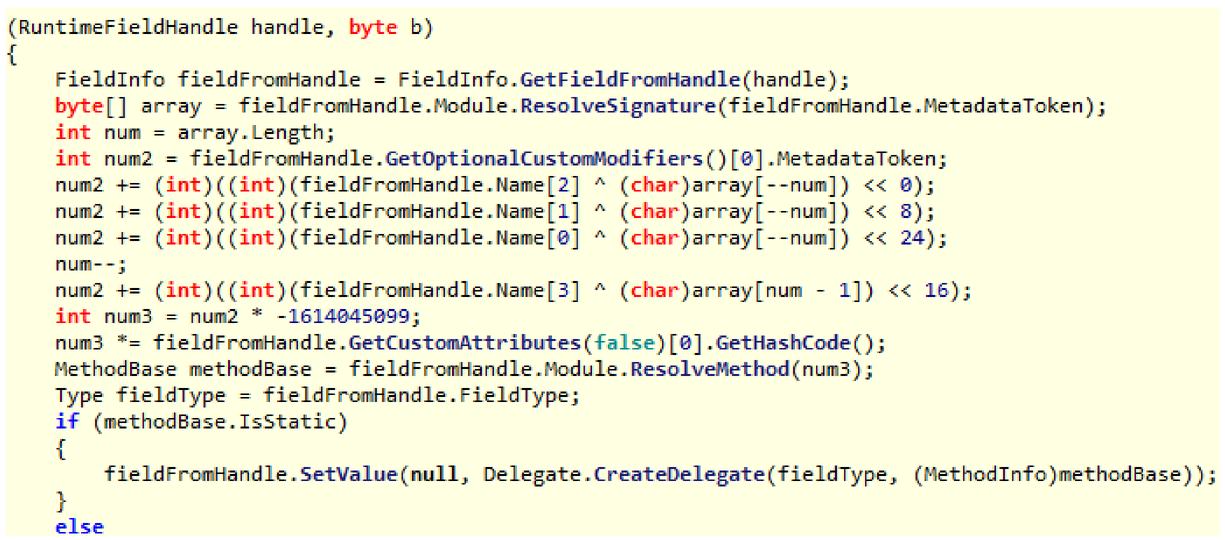
Let's look at the signature in the #Blob stream at offset **0087**:



```
 0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
12 29 04 20 01 01 0D 0D 06 20 10 12 14 C0 00 00
3F C0 23 46 4B 05 20 02 01 1C 18 03 20 00 0E 02
```

The red byte is the size of the signature, in this case 0x0D (13)
The orange bytes represent the original signature.
The blue bytes are static 0xC0 bytes.
The brown bytes are static 0x00 bytes.
The green bytes represent the token encoded by Confuser.

Let's look at the resolver method again:

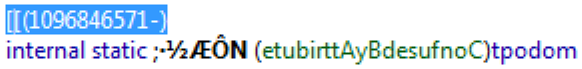


```
(RuntimeFieldHandle handle, byte b)
{
    FieldInfo fieldFromHandle = FieldInfo.GetFieldFromHandle(handle);
    byte[] array = fieldFromHandle.Module.ResolveSignature(fieldFromHandle.MetadataToken);
    int num = array.Length;
    int num2 = fieldFromHandle.GetOptionalCustomModifiers()[0].MetadataToken;
    num2 += (int)((int)(fieldFromHandle.Name[2] ^ (char)array[--num]) << 0);
    num2 += (int)((int)(fieldFromHandle.Name[1] ^ (char)array[--num]) << 8);
    num2 += (int)((int)(fieldFromHandle.Name[0] ^ (char)array[--num]) << 24);
    num--;
    num2 += (int)((int)(fieldFromHandle.Name[3] ^ (char)array[num - 1]) << 16);
    int num3 = num2 * -1614045099;
    num3 *= fieldFromHandle.GetCustomAttributes(false)[0].GetHashCode();
    MethodBase methodBase = fieldFromHandle.Module.ResolveMethod(num3);
    Type fieldType = fieldFromHandle.FieldType;
    if (methodBase.IsStatic)
    {
        fieldFromHandle.SetValue(null, Delegate.CreateDelegate(fieldType, (MethodInfo)methodBase));
    }
    else
```

With the knowledge about the signature, we know what’s happening up until this line:

*num3 *= fieldFromHandle.GetCustomAttributes(false)[0].GetHashCode();*

This simply multiplies num3 with the hashcode of the field’s first attribute’s type:



```
[][(1096846571-)
internal static ;·½ÆÔN (etubirttAyBdesufnoC)tpodom
```

After all this is done, num3 contains the token of the original method. So we now know how the resolving is done. Let’s see if our theory is correct by manually resolving a proxy with WinDbg. Start by loading the file then breakpoint clrjit by typing:

0: 000>sxe ld:clrjit

when it hits the breakpoint, load the SOS extension by typing:

0: 000>.loadby SOS clr

We see in the code that it uses System.Reflection.Module.ResolveMethod(int32 token) in order to resolve the method. So we put a breakpoint on this method by typing:

0: 000>!name2ee mscorlib.dll System.Reflection.Module.ResolveMethod

Put a breakpoint on the first overload of the method with:
0: 000>!bpmd -md <methoddesc>

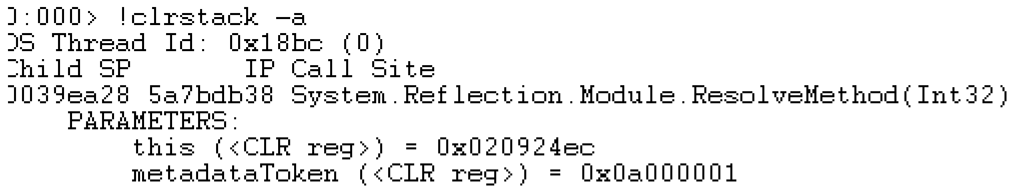
Once the breakpoint is set let the application run by typing:

0: 000>g

The breakpoint should be hit immediately. Now ResolveMethod is the last method on the call stack, and should have an Int32 parameter containing our token. Type:

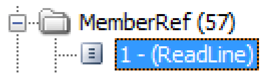
0: 000>!clrstack -a

This should be at the top:

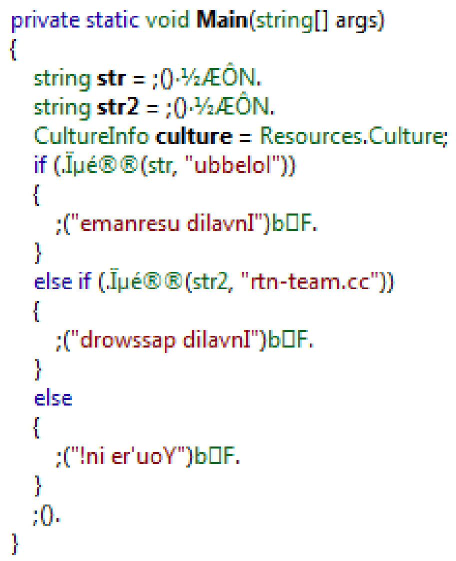


```
J:000> !clrstack -a
JS Thread Id: 0x18bc (0)
Child SP       IP Call Site
J039ea28 5a7bdb38 System.Reflection.Module.ResolveMethod(Int32)
    PARAMETERS:
        this (<CLR reg>) = 0x020924ec
        metadataToken (<CLR reg>) = 0x0a000001
```

Success! We have a valid metadata token pointing to row 1 in the MemberRef table. Let’s take a look in CFF explorer to see what this is:

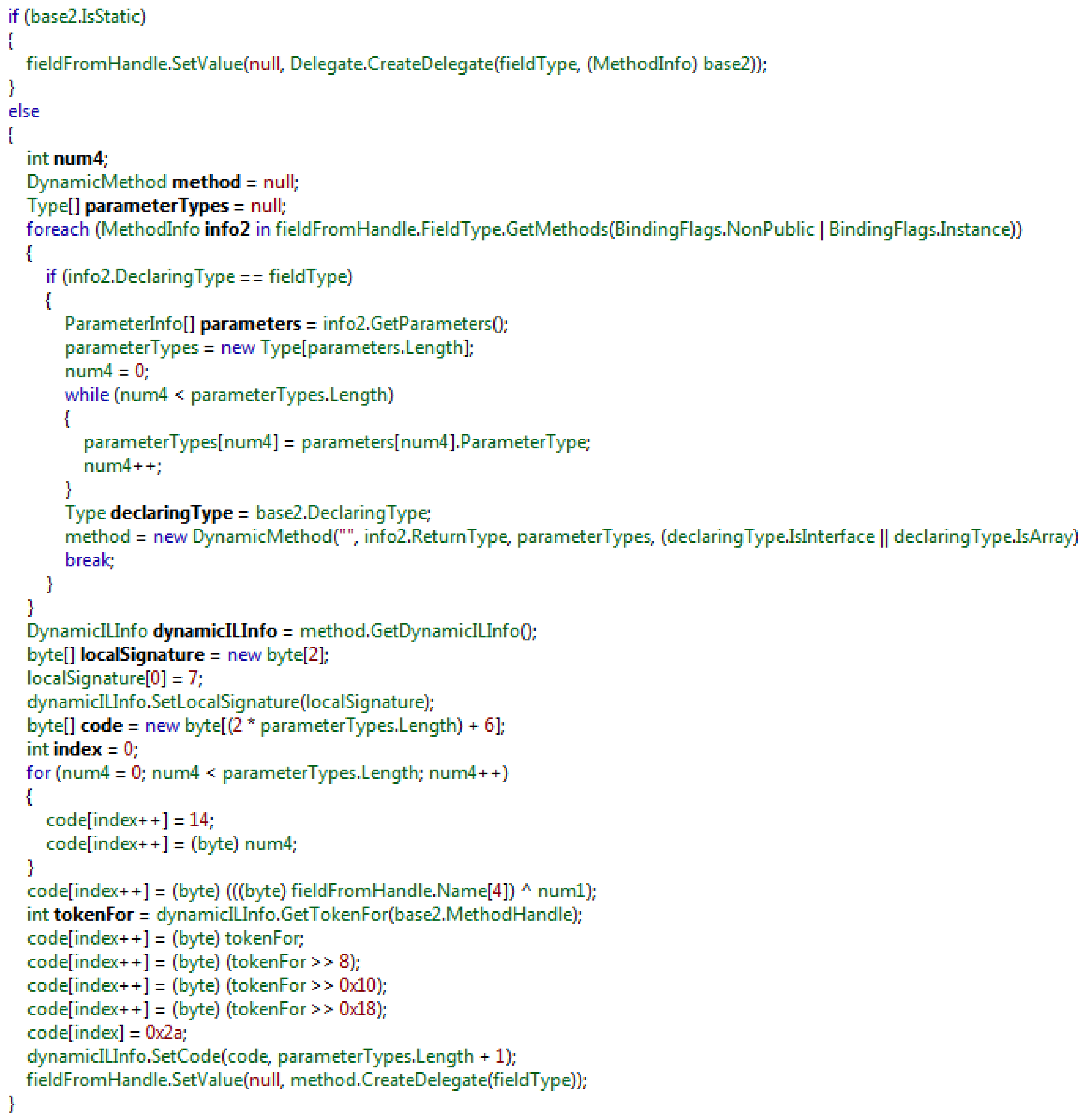


We can now tell that the two first lines in our Main method are actually calling Console.ReadLine();



```
private static void Main(string[] args)
{
   string str = ;()·½ÆÔN.
   string str2 = ;()·½ÆÔN.
   CultureInfo culture = Resources.Culture;
   if (.Ĩµé®®(str, "ubbelol"))
   {
      ;("emanresu dilavnI")b□F.
   }
   else if (.Ĩµé®®(str2, "rtn-team.cc"))
   {
      ;("drowssap dilavnI")b□F.
   }
   else
   {
      ;("!ni er'uoY")b□F.
   }
   ;()·
}
```

We can see how Confuser calls the original method in the second part of the resolver method:



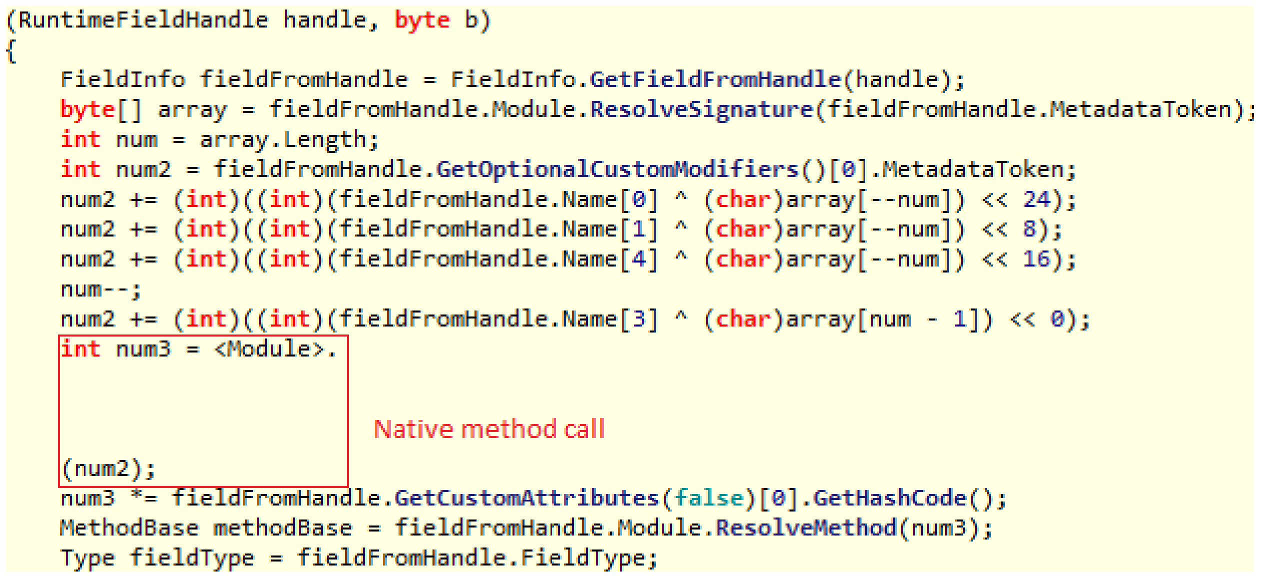
```
if (base2.IsStatic)
{
   fieldFromHandle.SetValue(null, Delegate.CreateDelegate(fieldType, (MethodInfo) base2));
}
else
{
   int num4;
   DynamicMethod method = null;
   Type[] parameterTypes = null;
   foreach (MethodInfo info2 in fieldFromHandle.FieldType.GetMethods(BindingFlags.NonPublic | BindingFlags.Instance))
   {
      if (info2.DeclaringType == fieldType)
      {
         ParameterInfo[] parameters = info2.GetParameters();
         parameterTypes = new Type[parameters.Length];
         num4 = 0;
         while (num4 < parameterTypes.Length)
         {
            parameterTypes[num4] = parameters[num4].ParameterType;
            num4++;
         }
         Type declaringType = base2.DeclaringType;
         method = new DynamicMethod("", info2.ReturnType, parameterTypes, (declaringType.IsInterface || declaringType.IsArray)
         break;
      }
   }
   DynamicILInfo dynamicILInfo = method.GetDynamicILInfo();
   byte[] localSignature = new byte[2];
   localSignature[0] = 7;
   dynamicILInfo.SetLocalSignature(localSignature);
   byte[] code = new byte[(2 * parameterTypes.Length) + 6];
   int index = 0;
   for (num4 = 0; num4 < parameterTypes.Length; num4++)
   {
      code[index++] = 14;
      code[index++] = (byte) num4;
   }
   code[index++] = (byte) (((byte) fieldFromHandle.Name[4]) ^ num1);
   int tokenFor = dynamicILInfo.GetTokenFor(base2.MethodHandle);
   code[index++] = (byte) tokenFor;
   code[index++] = (byte) (tokenFor >> 8);
   code[index++] = (byte) (tokenFor >> 0x10);
   code[index++] = (byte) (tokenFor >> 0x18);
   code[index] = 0x2a;
   dynamicILInfo.SetCode(code, parameterTypes.Length + 1);
   fieldFromHandle.SetValue(null, method.CreateDelegate(fieldType));
}
```

This is pretty self explanitory code. It basically creates a dynamic method, inserts a ldarg instruction for each parameter, and then a call to the original method. Once it created the dynamic method it retrieves a delegate to execute it and assigns the proxy field with it. So the process goes:

Main() calls the delegate → .cctor of delegate calls the resolver method which assigns the delegate with a dynamically created method pointing to original call → delegate is executed.

x86 encoding

The x86 encoding is really similar to the normal encoding, except for the fact that instead of multiplying the encoded token with a static key, it retrieves it through a native method:

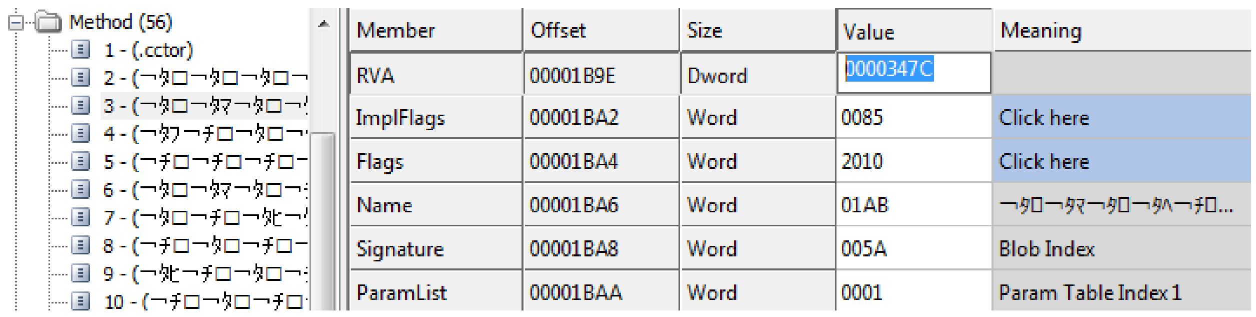


```
(RuntimeFieldHandle handle, byte b)
{
    FieldInfo fieldFromHandle = FieldInfo.GetFieldFromHandle(handle);
    byte[] array = fieldFromHandle.Module.ResolveSignature(fieldFromHandle.MetadataToken);
    int num = array.Length;
    int num2 = fieldFromHandle.GetOptionalCustomModifiers()[0].MetadataToken;
    num2 += (int)((int)(fieldFromHandle.Name[0] ^ (char)array[--num]) << 24);
    num2 += (int)((int)(fieldFromHandle.Name[1] ^ (char)array[--num]) << 8);
    num2 += (int)((int)(fieldFromHandle.Name[4] ^ (char)array[--num]) << 16);
    num--;
    num2 += (int)((int)(fieldFromHandle.Name[3] ^ (char)array[num - 1]) << 0);
    int num3 = <Module>.



    (num2);
    num3 *= fieldFromHandle.GetCustomAttributes(false)[0].GetHashCode();
    MethodBase methodBase = fieldFromHandle.Module.ResolveMethod(num3);
    Type fieldType = fieldFromHandle.FieldType;
```

This makes it a bit harder to tell what's happening in the method. But we can still use the same technique with WinDbg as I showed earlier in order to get the real token and resolve the original call. But we can take a look at the native method to see what it does. If we follow the call in ILSpy we see:

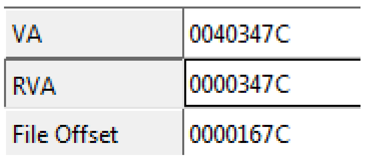
```
IL_00a2: ldloc.3
IL_00a3: call int32 '<Module>'::'\u2028\u2028\u2028\u2028$PST06000003'(int32)
IL_00a8: stloc.s 4
```

We can tell from PST0**6**00000**3** that our target method is in row **3** in the **Method** table. Open this method in CFF explorer and copy the RVA:



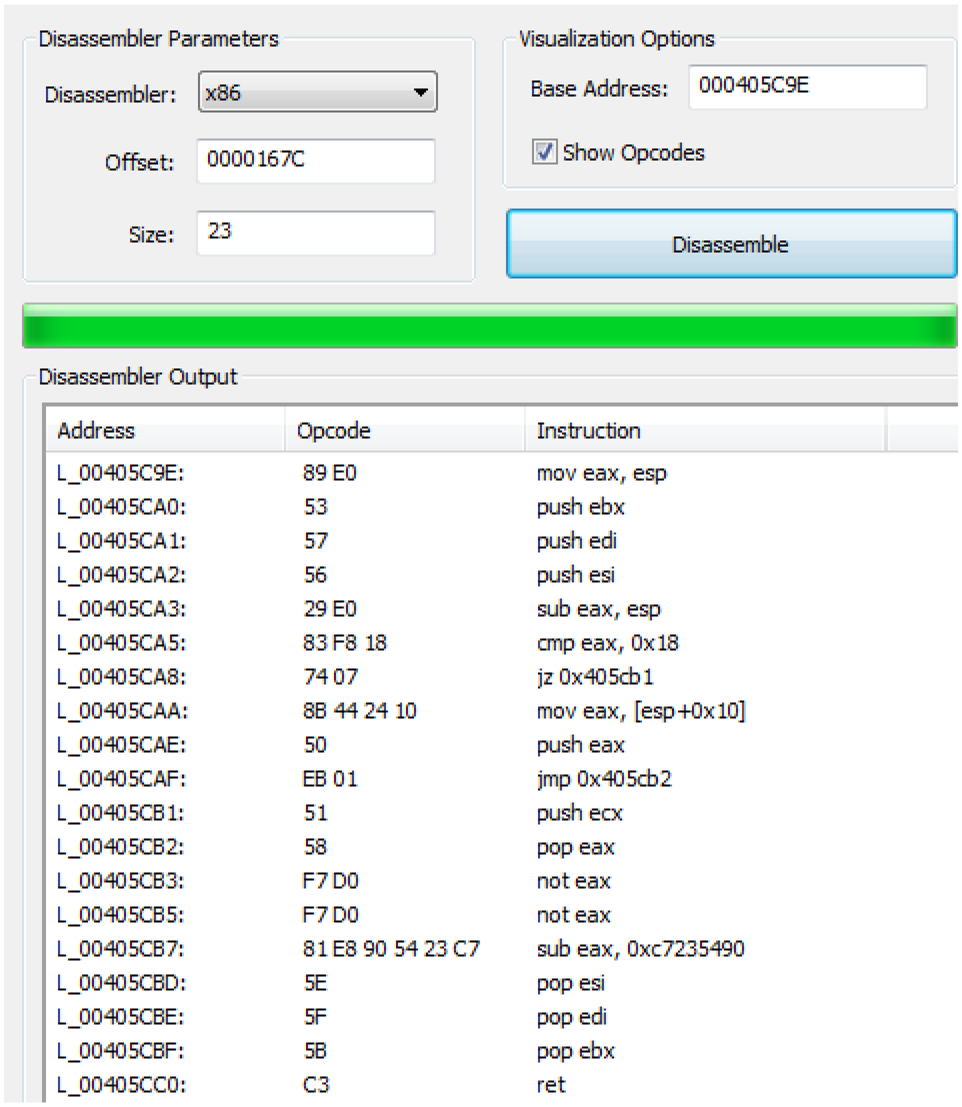
Member	Offset	Size	Value	Meaning
RVA	00001B9E	Dword	0000347C	
ImplFlags	00001BA2	Word	0085	Click here
Flags	00001BA4	Word	2010	Click here
Name	00001BA6	Word	01AB	¬ﾀ□¬ﾀﾏ¬ﾀ□¬ﾀﾍ¬ﾁ□...
Signature	00001BA8	Word	005A	Blob Index
ParamList	00001BAA	Word	0001	Param Table Index 1

In this case **0000347C**. Go to Address Converter to get the file offset:



VA	0040347C
RVA	0000347C
File Offset	0000167C

Copy the File Offset, in this case **0000167C** and go to Quick Disassembler, input the offset and disassemble:



Address	Opcode	Instruction
L_00405C9E:	89 E0	mov eax, esp
L_00405CA0:	53	push ebx
L_00405CA1:	57	push edi
L_00405CA2:	56	push esi
L_00405CA3:	29 E0	sub eax, esp
L_00405CA5:	83 F8 18	cmp eax, 0x18
L_00405CA8:	74 07	jz 0x405cb1
L_00405CAA:	8B 44 24 10	mov eax, [esp+0x10]
L_00405CAE:	50	push eax
L_00405CAF:	EB 01	jmp 0x405cb2
L_00405CB1:	51	push ecx
L_00405CB2:	58	pop eax
L_00405CB3:	F7 D0	not eax
L_00405CB5:	F7 D0	not eax
L_00405CB7:	81 E8 90 54 23 C7	sub eax, 0xc7235490
L_00405CBD:	5E	pop esi
L_00405CBE:	5F	pop edi
L_00405CBF:	5B	pop ebx
L_00405CC0:	C3	ret

We don't really have to think about this method though unless we attempt to statically decrypt the token. In that case we have to emulate this native method, which I talked a bit about at the end in my last paper:
https://drive.google.com/file/d/0B4NTaiQMcteTMk9fRFdQbFNKMEk/edit?usp=sharing

Thanks for reading. I hope you learned a thing or two about how ConfuserEx does reference proxies, and how to resolve them yourself, both manually and automatically. If you want you can look at my WinDbg session dump if you couldn’t follow the debugging part:
http://pastebin.com/cRCtVJFJ