Dissecting ConfuserEx - Constants protection

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

In this paper I'll explain how the constants protection works in ConfuserEx. This is one of the most common protections in obfuscators, this is what hides strings and other constants in your assembly.

Introduction

This protection works by taking every string and constant in the assembly, encrypt and compress them and put them into a resource. Generally in obfuscators the enrypted data is stored in an embedded resource. This was the case in Confuser 1.9, but in ConfuserEx it's done a bit differently with a more creative way.

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/

Let’s start by looking at what encrypted constants look like in an assembly:

Original

```
private static void Main(string[] args)
{
   Console.WriteLine("test");
   Console.ReadLine();
}
```

Obfuscated

```
private static void Main(string[] args)
{
   Console.WriteLine(;((807150671)<gnirts>
   Console.ReadLine();
}
```

Note that it’s actually *method<**string**>(807150671)*, the reason it looks weird is because the name of the method has a unicode character than overrides the left-to-right writing style with a right-to-left one.[1] As you see this completely obscures the string, and makes it hard to follow. Let’s follow the call and see what it does (from now on I’ll use DnSpy[2] to decompile):

```
// <Module>
internal static T smethod_5<T>(uint uint_0)
{
    uint_0 = (uint_0 * 1723248181u ^ 3552646435u);
    uint num = uint_0 >> 30;
    T result = default(T);
    uint_0 &= 1073741823u;
    uint_0 <<= 2;
    if ((ulong)num == 1uL)
    {
        int count = (int)<Module>.byte_0[(int)((UIntPtr)
        result = (T)((object)Encoding.UTF8.GetString(<Mo
    }
    else
    {
        if ((ulong)num == 2uL)
        {
            T[] array = new T[1];
            Buffer.BlockCopy(<Module>.byte_0, (int)uint_
            result = array[0];
        }
        else
        {
            if ((ulong)num == 3uL)
            {
                int num2 = (int)<Module>.byte_0[(int)((U
                int length = (int)<Module>.byte_0[(int)(
                Array array2 = Array.CreateInstance(type
                Buffer.BlockCopy(<Module>.byte_0, (int)u
                result = (T)((object)array2);
            }
        }
    }
    return result;
}
```

[1] http://en.wikipedia.org/wiki/Right-to-left_mark
[2] http://ubbecode.wordpress.com/2014/05/03/dnspy-more-powerful-net-decompiler/

I’ll call this method the “*retrieval method*”. This looks pretty confusing (no pun intended), but it makes sense if you look at the ConfuserEx source of the method above:

```
private static T Get<T>(uint id) {
        id = (uint)Mutation.Placeholder((int)id);
        uint t = id >> 30;

        T ret = default(T);
        id &= 0x3fffffff;
        id <<= 2;

        if (t == Mutation.KeyI0) {
                int l = b[id++] | (b[id++] << 8) | (b[id++] << 16) | (b[id++] << 24);
                ret = (T)(object)Encoding.UTF8.GetString(b, (int)id, l);
        }
                // NOTE: Assume little-endian
        else if (t == Mutation.KeyI1) {
                var v = new T[1];
                Buffer.BlockCopy(b, (int)id, v, 0, Mutation.Value<int>());
                ret = v[0];
        }
        else if (t == Mutation.KeyI2) {
                int s = b[id++] | (b[id++] << 8) | (b[id++] << 16) | (b[id++] << 24);
                int l = b[id++] | (b[id++] << 8) | (b[id++] << 16) | (b[id++] << 24);
                Array v = Array.CreateInstance(typeof (T).GetElementType(), l);
                Buffer.BlockCopy(b, (int)id, v, 0, s - 4);
                ret = (T)(object)v;
        }
        return ret;
}
```

We see that it does some arithmetic to find out what sort of object it should return. There are 3 forms of an object it can return; a string, a single object or an array of objects. The first if block reads a string from the data and returns it. The second if block reads 1 object of type T from the data and returns it. The third if block creates an array of object with type T and fills it with “l” amount of objects read from the data and returns the array.

But it doesn’t look like this actually decrypts anything, just retrieveing. But we can see that in every case it returns a value read from a static **byte**[] field in the <Module> type. If we analyze the field and find where it’s assigned from, we find this method:

```
internal static void smethod_1()
{
    uint num = 32u;
    uint[] expr_0A = new uint[32];
    RuntimeHelpers.InitializeArray(expr_0A, fieldof(<Module
    uint[] array = expr_0A;
    uint[] array2 = new uint[16];
    uint num2 = 654932407u;
    for (int i = 0; i < 16; i++)
    {
        num2 ^= num2 >> 12;
        num2 ^= num2 << 25;
        num2 ^= num2 >> 27;
        array2[i] = num2;
    }
    int num3 = 0;
    int num4 = 0;
    uint[] array3 = new uint[16];
    byte[] array4 = new byte[num * 4u];
    while ((long)num3 < (long)((ulong)num))
    {
        for (int j = 0; j < 16; j++)
        {
            array3[j] = array[num3 + j];
        }
        array3[0] = (array3[0] ^ array2[0]);
        array3[1] = (array3[1] ^ array2[1]);
        array3[2] = (array3[2] ^ array2[2]);
        array3[3] = (array3[3] ^ array2[3]);
        array3[4] = (array3[4] ^ array2[4]);
        array3[5] = (array3[5] ^ array2[5]);
        array3[6] = (array3[6] ^ array2[6]);
        array3[7] = (array3[7] ^ array2[7]);
        array3[8] = (array3[8] ^ array2[8]);
        array3[9] = (array3[9] ^ array2[9]);
        array3[10] = (array3[10] ^ array2[10]);
        array3[11] = (array3[11] ^ array2[11]);
        array3[12] = (array3[12] ^ array2[12]);
        array3[13] = (array3[13] ^ array2[13]);
        array3[14] = (array3[14] ^ array2[14]);
        array3[15] = (array3[15] ^ array2[15]);
        for (int k = 0; k < 16; k++)
        {
            uint num5 = array3[k];
            array4[num4++] = (byte)num5;
            array4[num4++] = (byte)(num5 >> 8);
            array4[num4++] = (byte)(num5 >> 16);
            array4[num4++] = (byte)(num5 >> 24);
            array2[k] ^= num5;
        }
        num3 += 16;
    }
    <Module>.byte_0 = <Module>.smethod_0(array4);
}
```

Now this looks like some decryption code. In this method we can also see what I talked about earlier about it not retrieving data from a resource, instead it creates an **uint**[] with all the data. I'm not gonna go into detail on everything about the decryption. You could study the source[3] yourself to find out. The important thing to note is that at the last line:

```
<Module>.byte_0 = <Module>.smethod_0(array4);
```

This assigns all the decrypted data to the static byte array byte_0.

This seems simple enough, but as always with Confuser there's not just 1 method that everything goes through. There's several retrieval methods which looks the same except for different keys and values for the arithmetics.

This is in short how the constants protection works at runtime.

---

[3] https://github.com/yck1509/ConfuserEx/blob/master/Confuser.Runtime/Constant.cs#L8

How do we reverse this?

There are 2 main ways of reversing the protection; dynamically or statically. Decrypting the strings dynamically is arguably the easiest way and it can be done by simply invoking the retrieval method with the same parameter as in the actual assembly. But you need to have the data array initialized and decrypted before. To reverse it statically you need to follow these steps:

1. Read all the values in the method that initializes the data array, and decrypt the constants.
2. Find each call to the retrieval methods, gather the parameters for them.
3. Read all the values in each retrieval method.
4. Go through each call and decrypt them using the parameter and the values in the corresponding retrieval method.

Simple enough.

There's not much more to explain about this protection, but I'll show you how you can dump the decrypted constants from memory using WinDbg:

Start by setting a breakpoint on clr-/mscorjit modload and then run the application by typing:

0: 000> sxe ld:clrjit / sxe ld:mscorjit
0: 000> g

When the modload is hit, load the SOS extension by typing:

0: 000> .loadby SOS clr / .loadby SOS mscorwks

When SOS is loaded we can dump the appdomain to see all active modules:

0: 000> !dumpdomain

You should see your target module in the list:

Assembly: 004b17c8 [C:\...\...\test-cleaned.exe]
ClassLoader: 004b1890
SecurityDescriptor: 004b0d20
* Module Name*
00282eac *C:\Users\mattias\Documents\ConfuserEx\Confused\test-cleaned.exe*

Dump the method table of this module:

0:000> !dumpmodule -mt ***00282eac***

You'll see:

Types defined in this module

MT TypeDef Name

00283560 *0x02000001 <Module>* ←
0028351c 0x0200000b <Module>+Struct4

Dump the method descriptors in the first method table:

0:000> !dumpmt -md ***00283560***

You'll see:

MethodDesc Table
Entry MethodDe JIT Name
0028c011 00283414 NONE <Module>..cctor()
0028c01d 00283434 NONE <Module>.smethod_2(UInt32)
0028c021 00283448 NONE <Module>.smethod_3(UInt32)
0028c025 0028345c NONE <Module>.smethod_4(UInt32)
0028c029 00283470 NONE <Module>.smethod_5(UInt32)
0028c02d 00283484 NONE <Module>.smethod_6(UInt32)
0028c015 0028341c NONE <Module>.smethod_0(Byte[])
0028c019 ***00283428*** *NONE <Module>.smethod_1()*

The last entry is the Initialize method in ConfuserEx (the method that decrypts all constants).
Put a breakpoint on it and run:

0: 000> !bpmd -md 00283428
0: 000> g

When we hit the breakpoint we look at the call stack by typing:

0: 000> !clrstack

You should see:

Child SP IP Call Site
001ae75c ***003b0070*** *<Module>.smethod_1()*
001ae760 003b0059 <Module>..cctor()
001ae8d8 58153de2 [GCFrame: 001ae8d8]
001af4cc 58153de2 [DebuggerClassInitMarkFrame: 001af4cc]

Since we know that when this method is finished executing, all the constants are decrypted in memory, we need to put a breakpoint right when this method finishes. First we need to get the MethodDesc:

0: 000> !ip2md ***003b0070***

You'll see:

MethodDesc: ***00193428***
Method Name: <Module>.smethod_1()
Class: 00191870
MethodTable: 00193560
mdToken: 06000003
Module: 00192eac
IsJitted: yes
CodeAddr: 00350070
Transparency: Critical

Dump the jitted code of this method:

0: 000> !u ***00193428***

Scroll down until you see the "*ret*" instruction:

00350070 55 push ebp
00350071 8bec mov ebp,esp
00350073 57 push edi
00350074 56 push esi
00350075 53 push ebx
…
00350499 5b pop ebx
0035049a 5e pop esi
0035049b 5f pop edi
0035049c 5d pop ebp

***0035049d** c3   ret*

Put a breakpoint on it, and run the application:

0: 000> bu 0035049d
0: 000> g

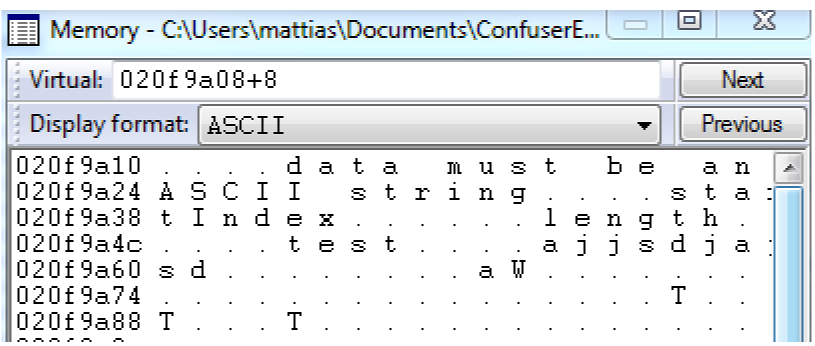
We should hit the breakpoint immediately. This means everything is decrypted in memory, and the **byte**[] object data should be set. Let's dump all objects:

0: 000> !dso

Our target object will be in the EAX register:

OS Thread Id: 0x1604 (0)
ESP/REG Object   Name
*eax   **020f9a08** System.Byte[]*
002EF89C 030f2160 System.Object[]   (System.Object[])
002EFA50 020f1238 System.SharedStatics

Open up the Memory View in WinDbg (**View→Memory** or **Alt+5**), and go to address of object + 8 and change display format to ASCII:



Success! We can see all the decrypted strings.

So there you go. I hope you learned a thing or two, and get some ideas on how the protection works as well as how to reverse/decrypt it. If you had issues following the WinDbg guide feel free to take a look at my session dump: http://pastebin.com/6Tx2KRAb

Thanks for reading!