Dissecting ConfuserEx - x86 switch predicates

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

In this paper i'll try to explain how the native predicate switches in ConfuserEx works, and how you can debug them to uncover the actual flow of the method.

Tool used:

- Reflector
- ILSpy
- OllyDbg
- CFF Explorer

Target file:

- http://www.multiupload.nl/34A98ZNG8Y

Introduction

One of the new features in ConfuserEx is the improved control flow obfuscation. It's gotten much better, and is able to create switches that de4dot can't deobfuscate. This is partly because the switch instruction depends on a value that's retrieved through a native method, which makes it impossible for de4dot to emulate or evaluate where the first jump goes. Here's an example of what a "native switch" looks like in Reflector:

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/

```
Label_0006:
    switch (((374881865-)
    {
        case 0:
        case 3:
        case 5:
        case 6:
        case 7:
        case 8:
        case 15:
            goto Label_0006;

        case 1:
            str = Console.ReadLine();
            goto Label_0006;

        case 2:
            break;

        case 4:
            Console.WriteLine("Invalid username");
            goto Label_0006;

        case 9:
            Console.WriteLine("Invalid password");
            goto Label_0006;

        case 10:
            Console.ReadLine();
            goto Label_0006;

        case 11:
        {
            bool flag = !(str != "ubbelol");
```

But if you look at it, you might see that it seems to be an endless loop since every case jumps back to **Label_0006** which will just run the same code again. That's not the case. If we take a look at the IL code behind this we'll see why.

Let's look at the IL code of this block:

```
case 4:
    Console.WriteLine("Invalid username");
    goto Label_0006;
```

→

```
L_00e6: ldstr "Invalid username"
L_00eb: call void [mscorlib]System.Console::WriteLine(string)
L_00f0: ldc.i4 -1677661619
L_00f5: br L_0006
```

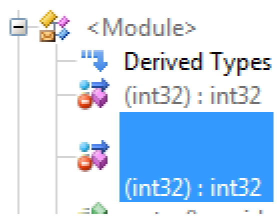
Here we see that it's actually pushing a new value onto the stack before jumping back to the switch, meaning it's not an endless loop. I'm not sure why Reflector doesn't interpret this correctly but it doesn't really matter. From now on i'll use ILSpy to view the code in this paper because it displays a more accurate code.

The first thing we have to do in order to make sense of this switch is to find out where the first jump goes. In order to find out we need to analyse the method that returns the value that the switch depends on. Lets look at the code:

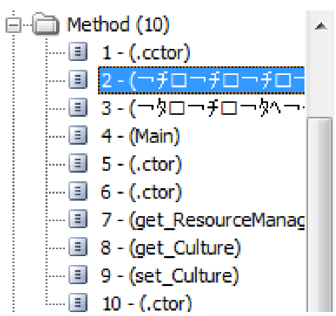
```
IL_0001: ldc.i4 -568188473
// loop start (head: IL_0006)
    IL_0006: call int32 '<Module>'::'\u2028\u2028$PST06000002'(int32)
    IL_000b: switch (IL_0092, IL_00a9, IL_0067, IL_00fa, IL_00e6, IL_0
```

We follow the call at IL_0006 and find:

```
 .method static privatescope pinvokeimpl
     int32 '\u2028\u2028$PST06000002' (
         int32 ''
     ) native unmanaged preservesig
```



Although the name is not in clear, readable characters ILSpy appends the metadata token of the method at the end of the name. This means we can tell from PST06000002 that our target method is in the **Method** table at row **2**.



Member	Offset	Size	Value	Meaning
RVA	000007CC	Dword	0000229C	
ImplFlags	000007D0	Word	0085	Click here
Flags	000007D2	Word	2010	Click here
Name	000007D4	Word	0062	¬ﾁ□¬ﾁ□¬ﾁ□¬ﾀ□¬ﾀ□...
Signature	000007D6	Word	002B	Blob Index
ParamList	000007D8	Word	0001	Param Table Index 1

Since it's a native method there's no JITting involved, which means we can open up the file in OllyDbg and look at what the method does:

Load the file into OllyDbg → press **View** in the top menu → **Executable modules** → find our module (dumped.exe) and double click it. Scroll to the top in the CPU window, and you'll find something similar to:

```
012C2014  .  1C0D0000    DD 00000D1C    MetaData.Size = 3356.
012C2018  .  03000200    DD 00020003    Flags = COMIMAGE_FLAGS_ILONLY|COMIMAGE_FLAGS_3
012C201C  .  04000006    DD 06000004    EntryPoint = 6000004
012C2020  .  40230000    DD 00002340    Resources.VirtualAddress = 2340
012C2024  .  B8000000    DD 000000B8    Resources.Size = 184.
012C2028  .  00000000    DD 00000000    StrongName.VirtualAddress = 0
012C202C  .  00000000    DD 00000000    StrongName.Size = 0
012C2030  .  00000000    DD 00000000    CodeManager.VirtualAddress = 0
012C2034  .  00000000    DD 00000000    CodeManager.Size = 0
012C2038  .  00000000    DD 00000000    VTableFixups.VirtualAddress = 0
012C203C  .  00000000    DD 00000000    VTableFixups.Size = 0
012C2040  .  00000000    DD 00000000    ExportAddr.VirtualAddress = 0
012C2044  .  00000000    DD 00000000    ExportAddr.Size = 0
012C2048  .  00000000    DD 00000000    NativeHeader.VirtualAddress = 0
012C204C  .  00000000    DD 00000000    NativeHeader.Size = 0
012C2050  .  06          DB 06          Flags_CodeSize = (CodeSize=1, Type=TinyFormat)
012C2051  $  2A          ret
012C2052  .  1E          DB 1E          Flags_CodeSize = (CodeSize=7, Type=TinyFormat)
```

this is the .NET directory of the executable, which means all method bodies should follow right after. Now to find our target method:

Copy the first four numbers in the most-left column (address), in this case *012C* → press **Ctrl+G** → input *012C* and then the RVA of the target method, in this case *229C:*

This will take us straight to the code:

```
012C229C  .  89E0           MOV EAX,ESP
012C229E  .  53             PUSH EBX
012C229F  .  57             PUSH EDI
012C22A0  .  56             PUSH ESI
012C22A1  .  29E0           SUB EAX,ESP
012C22A3  .  83F8 18        CMP EAX,18
012C22A6  .- 74 07          JE SHORT 012C22AF
012C22A8  .  8B4424 10      MOV EAX,DWORD PTR SS:[ESP+10]
012C22AC  .  50             PUSH EAX
012C22AD  .- EB 01          JMP SHORT 012C22B0
012C22AF  >  51             PUSH ECX
012C22B0  >  B8 D34C8552    MOV EAX,52854CD3
012C22B5  .  81E8 8C3E04E!  SUB EAX,E5043E8C
012C22BB  .  69C0 EFC2531'  IMUL EAX,EAX,1753C2EF
012C22C1  .  B9 A0134EDB    MOV ECX,DB4E13A0
012C22C6  .  81E9 7B554BC:  SUB ECX,C84B557B
012C22CC  .  BA 17CC8F57    MOV EDX,578FCC17
012C22D1  .  69D2 077E1C4I  IMUL EDX,EDX,4D1C7E07
012C22D7  .  31D1           XOR ECX,EDX
012C22D9  .  BA 29D19945    MOV EDX,4599D129
012C22DE  .  69D2 8F9E2C7:  IMUL EDX,EDX,722C9E8F
012C22E4  .  BB 419B643F    MOV EBX,3F649B41
012C22E9  .  F7D3           NOT EBX
012C22EB  .  31DA           XOR EDX,EBX
012C22ED  .  29D1           SUB ECX,EDX
012C22EF  .  5A             POP EDX
012C22F0  .  29D1           SUB ECX,EDX
012C22F2  .  29C8           SUB EAX,ECX
012C22F4  .  69C0 13C55C3I  IMUL EAX,EAX,3C5CC513
012C22FA  .  F7D8           NEG EAX
012C22FC  .  5E             POP ESI
012C22FD  .  5F             POP EDI
012C22FE  .  5B             POP EBX
012C22FF  .  C3             RETN
```

So now to find out where the first jump in the switch goes, put a breakpoint on the RETN instruction and run the debugged application (F9). Once we reached the breakpoint, look in the Registers (FPU) window in OllyDbg and look at what EAX contains:

```
Registers (FPU)
EAX 00000001
ECX D830EF64
EDX DE2221C7
EBX 0025EE04
ESP 0025ECC0
EBP 0025ED14
ESI 0054DF40
EDI 0025ED48
EIP 012C22FF dumped.012C22FF
```

EAX = 00000001 meaning first jump goes to **Case 1**:

```
case 1:
{
    string a = Console.ReadLine();
    arg_06_0 = 935101571;
    continue;
}
```

The application is now waiting on input. Enter anything into the console and then look at EAX again:

```
Registers (FPU)
EAX 0000000D
ECX 7E968CA8
EDX 37BC8483
EBX 0025EE04
ESP 0025ECC0
EBP 0025ED14
ESI 0054DF40
EDI 0025ED48
EIP 012C22FF dumped.012C22FF
```

EAX = 0000000D (13) meaning second jump goes to **Case 13**:

```
case 13:
{
    string a2 = Console.ReadLine();
    arg_06_0 = -747102535;
    continue;
}
```

If we keep doing this until the end of the method, we can see the actual flow of the code. Obviously this is not a viable method to deobfuscate the method, since it would take ages to do it manually and there are different cases depending on what is passed to the predicate, e.g:

```
case 0:
{
    bool flag;
    arg_06_0 = (flag ? 1776203624 : 1114015633);
    continue;
}
```

I hope that it at least gave you an idea on how the switches are implemented, and how they work with the code. There is no easy way around to deobfuscate the switches. In order to do it in automatically with an application you'll either have to create a small x86 emulator to determine the return value of each predicate, and from there repartition the IL blocks so they form a clean method, or dynamically invoke the method in order to get the return value.

Here's an example of how you could emulate the methods using BeaEngine and dnlib that I wrote (messy, but should work as a basic example):
https://github.com/UbbeLoL/ConfuserDeobfuscator/tree/x86emu/ConfuserDeobfuscator/ConfuserDeobfuscator/Engine/Routines/Ex/x86