28 - 30 September, 2022 / Prague, Czech Republic

COMBATING CONTROL FLOW FLATTENING IN .NET MALWARE

Georgy Kucherin

Kaspersky, Russia

georgy.kucherin@gmail.com

ABSTRACT

It has become increasingly popular for developers of targeted malware to create sophisticated custom obfuscators that make use of control flow transformation techniques. One such technique is control flow flattening. It basically rearranges lines of code in a chaotic manner, thus making analysis tedious.

While control flow flattening has previously been researched in the context of C and C++ binaries, virtually no attention has been given to unflattening .NET programs. Furthermore, existing deobfuscation software that removes unflattening in C/C++ programs cannot be applied to .NET binaries. That is because .NET code is compiled not to x86 assembly, but to a virtual bytecode called Common Intermediate Language.

Advanced threat actors favour using control flow flattening to protect their .NET implants. For example, this obfuscation was employed in early versions of the Kazuar backdoor. It has also recently been spotted in DoubleZero, a wiper discovered in March 2022 in Ukraine.

In this research, we detail how to remove control flow flattening from .NET binaries, taking the DoubleZero wiper as an example. To perform unflattening, we modify the source code of *de4dot*, a popular .NET deobfuscation framework. We first explain how to restore DoubleZero's original control flow. Then we move from general to specific and demonstrate how to add a custom deobfuscator module to *de4dot*. Afterwards, we describe how *de4dot*'s block deobfuscation component comes in useful for implementing the unflattening algorithm. Along the way, we give multiple tips on how to use other *de4dot* features.

The paper is accompanied by extensively commented code of the unflattener. It manages to successfully deobfuscate all of DoubleZero's functions. It is possible to use this code as a basis for creating unflatteners of other .NET malware families.

INTRODUCTION

Control flow flattening is an obfuscation technique that is considered quite difficult to combat. For this reason, it is often used in malware to considerably slow down the process of its reverse engineering. For example, it is quite common to encounter flattened C or C++ samples protected with OLLVM, an open-source obfuscator. Examples of such malware include FinSpy and Emotet. In the case of C and C++, control flow flattening can be removed using plug-ins for the *Hex-Rays* decompiler [1]. However, there is not much information available about how to perform unflattening of C# (.NET) binaries. This is despite the fact that this obfuscation can be encountered in C# malware as well. For instance, it has recently been found in the DoubleZero wiper, which was discovered in Ukraine in 2022. In this paper we will discuss how to unflatten DoubleZero's code and make its functions with thousands of lines look pretty.

We will first examine a sample of DoubleZero and find out how control flow flattening is implemented in it. Then, we will describe the steps required to remove the flattening from the sample. Finally, and most importantly, we will explain in great detail how to implement the unflattening algorithm using *de4dot*, a framework for deobfuscating .NET binaries. We assume that the readers of this paper have no previous experience of working with this framework. While explaining how to code the unflattener, we will provide all the necessary information about *de4dot*'s internals.

TAKING A LOOK AT DOUBLEZERO

The sample of DoubleZero that we will be looking at (SHA256: `30b3cbe8817ed75d8221059e4be35d5624bd6b5dc921d4991a7adc4c3eb5de4a`) can be downloaded for free from vx-underground [2]. In order to better understand the contents of this paper, we strongly recommend that the reader downloads the sample and follows along with our steps.

First, we will open the sample in *dnSpy*, a .NET decompiler. Upon looking at DoubleZero's `Main` function (Figure 1), we observe that it has 1,392 lines. The code itself is heavily obfuscated and difficult to read.

In order to better understand what kind of obfuscation we are dealing with in this sample, we will take a look at another function that has many fewer lines of code, namely `_6d604a38de5c5e96._a39a27ee5cddb1d4`. In this function, we observe the code as shown in Figure 2.

On lines 4-10 in the snippet shown in Figure 2, we initialize an array of five integer arrays. Then, on line 12, we iterate over each integer array, using the variable `i` to access its elements. On line 13, we pass each integer array to the `X1BFovEdxME9D.HbkAc6Pu` function that returns an integer. Since this function accepts an array of integers with large numbers and returns a rather small integer (such as 13 or 121), we can deduce that `X1BFovEdxME9D.HbkAc6Pu` is an integer decryption function.

Depending on the value of the decrypted integer, we can either:

- Go to line 15 (if the integer equals 13)
- Go to line 18 (if the integer equals 121)
- Go to the next iteration of the loop (if the integer is neither 13 nor 121).

```
        while (num11 < 5)
        {
            int num2 = FXgigIXPNb5.AfIRQbGj((int[])array10[num11], 0, 0);
            if (num2 != 15)
            {
                if (num2 == 26 && ex2.Message != "TEujU4")
                {
                    throw ex2;
                }
            }
            else
            {
                IsolatedStorageFilePermission isolatedStorageFilePermission = (IsolatedStorageFilePermission)obj10;
            }
            num11++;
        }
        goto IL_956;
    }
    goto IL_76E;
}
if (num != 539)
{
    if (num == 574)
    {
        CodeTypeDeclarationCollection codeTypeDeclarationCollection = (CodeTypeDeclarationCollection)obj;
    }
}
IL_956:
i++;
```

Figure 1: A snippet of DoubleZero's Main function.

```
object obj = null;
int i = 0;
object[] array = new object[]
{
    new int[] {90,2089875171,90,1318285745,10,90,886806518,10,180},
    new int[] {90,-986721120,90,1218371032,20,90,1318285745,10,90,886806518,10,180},
    new int[] {90,-986721052,90,1218371032,20,90,1318285745,10,90,886806518,10,180},
    new int[] {90,2089875154,90,1318285745,10,90,886806518,10,180},
    new int[] {90,2089875047,90,1318285745,10,90,886806518,10,180}
};

while (i < 5) {
    int num = X1BFovEdxME9D.HBkAc6Pu((int[])array[i], 0, 0);
    if (num == 13) {
        goto IL_29A;
    }
    if (num == 121) {
        <code omitted>
    }
    IL_2A1:
    i++;
    continue;
    IL_29A:
    RequestCachingSection requestCachingSection = (RequestCachingSection)obj;
    goto IL_2A1;
}
```

Figure 2: Obfuscated code.

By looking at this snippet of code, we can conclude that this obfuscation technique rearranges lines of code in a random order. During execution, the program uses an array of integer arrays to determine the correct order in which lines should be executed. This exact type of obfuscation is called *control flow flattening*.

If we examine the code that is executed if the variable num is equals 121 (which has been omitted from Figure 2), we encounter another layer of flattening that looks more complex (Figure 3).

```
int j = 0;
object[] array2 = new object[] { <15 integer arrays> };
while (j < 15)
{
    int num2 = X1BFovEdxME9D.HBkAc6Pu((int[])array2[j], 0, 0);
    if (num2 <= 542) {
        if (num2 != 219) {
            if (num2 != 494) {
                if (num2 == 542) {
                    _9f6717951b3535fb甸檄ꝶ喽썩歷瓶額렆엥숀拐ꝺ抹鼻梣詰浏뢷蟹經헬렛쯩갿츮昤.RtlAdjustPrivilege(9UL, true, false, ref flag);
                }
            } else {
                FieldDirection fieldDirection = (FieldDirection)obj2;
            }
        }
    } else if (num2 != 544) {
        if (num2 != 559) {
            if (num2 == 581) {
                _9f6717951b3535fb甸檄ꝶ喽썩歷瓶額렆엥숀拐ꝺ抹鼻梣詰浏뢷蟹經헬렛쯩갿츮昤.RtlAdjustPrivilege(18UL, true, false, ref flag);
            }
        } else {
            _9f6717951b3535fb甸檄ꝶ喽썩歷瓶額렆엥숀拐ꝺ抹鼻梣詰浏뢷蟹經헬렛쯩갿츮昤.RtlAdjustPrivilege(17UL, true, false, ref flag);
        }
    } else {
        _9f6717951b3535fb甸檄ꝶ喽썩歷瓶額렆엥숀拐ꝺ抹鼻梣詰浏뢷蟹經헬렛쯩갿츮昤.RtlAdjustPrivilege(19UL, true, false, ref flag);
    }
    j++;
}
```

Figure 3: Another example of flattened code.

In this sample we are also dealing with *nested flattening*: a layer of flattened code may contain another layer of flattened code inside it.

MANUALLY DEOBFUSCATING CONTROL FLOW FLATTENING

As we have just noted, functions in the DoubleZero sample contain multiple flattening layers that make the process of manually removing flattening from the code quite tedious. However, before trying to remove control flow flattening in an automated way, we first need to understand how to get rid of flattening manually. As an example, we will remove flattening from the code shown in Figure 3.

First, we need to get the order in which flattened lines of code are executed. Recall from the previous section that this execution sequence is stored in the `array2` variable. This variable contains 15 integer arrays. To extract the execution sequence, we need to pass each one of the 15 arrays into the `X1BFovEdxME9D.HbkAc6Pu` function that performs integer decryption. By doing that, we will obtain 15 integers that encode the execution order.

The easiest way to decrypt the integers is via *dnSpy*'s C# interactive console, which can be opened via the 'View' menu. In order for the decryption to work, we need to paste the contents of the decryption function and the encrypted integers into the console. Then, we need to invoke the decryption function for every encrypted integer, as shown in Figure 4.

```
.                         },
.                         new int[]
.                         {
.                             90,
.                             -886806298,
.                             90,
.                             886806518,
.                             10,
.                             180
.                         }
.                     };
> List<int> executionOrder = new List<int>();
. for (int i = 0; i < 15; i++) { executionOrder.Add(HBkAc6Pu((int[])array2[i], 0, 0)); }
> executionOrder
List<int>(15) { 542, 98, 204, 579, 200, 559, 234, 492, 423, 581, 544, 179, 140, 495,
  220 }
```

Figure 4: Decrypting the execution order array using dnSpy's C# interactive console feature.

As a result, we get the following execution order list: `{542, 98, 204, 579, 200, 559, 234, 492, 423, 581, 544, 179, 140, 495, 220}`.

If we match the execution order list with the code in Figure 3, we will notice that some numbers from the list are missing in the code (for example, the number 98 at index 1 of the list cannot be seen in any of the if conditions). As these numbers do

not correspond to any lines of flattened code, we will have to remove them from our list. After doing that, we get the following list: `{542, 559, 581, 544}`.

Finally, we should place the flattened code lines in the correct order by copying and pasting them. By doing that, we obtain the following unflattened code (Chinese characters were omitted from the code for clarity):

```
_9f6717951b3535fb.RtlAdjustPrivilege(9UL, true, false, ref flag);
_9f6717951b3535fb.RtlAdjustPrivilege(17UL, true, false, ref flag);
_9f6717951b3535fb.RtlAdjustPrivilege(18UL, true, false, ref flag);
_9f6717951b3535fb.RtlAdjustPrivilege(19UL, true, false, ref flag);
```

DISCUSSING HOW TO AUTOMATE CONTROL FLOW UNFLATTENING

In the previous section, we performed the following operations in order to get rid of control flow flattening:

1. Extracted the execution order of flattened code from an array.
2. Reordered lines of flattened useful code in accordance with the execution order.

Now we need to think of a way to automate these two actions. Extracting the execution order does not look difficult to do. As we remember, the execution order is stored as an array (with type `object []`) of integer arrays (`int []`). This combination of two types can be used as a pattern to detect arrays containing the execution sequence. As we will see in the next section, such a pattern matching algorithm is efficient enough and can be implemented in a few lines of code.

Line reordering is trickier to automate. Before being able to perform the reordering itself, we should first think of how to match a number from the execution order to the lines of code that correspond to this number. For example, we should be able to determine programmatically that in the case of Figure 3, number 542 corresponds to the line `_9f6717951b3535fb.RtlAdjustPrivilege(9UL, true, false, ref flag);`.

A solution that perhaps seems obvious is to implement the number-to-code matching algorithm at the source code level. We can decompile the binary into C# and then extract lines of code by matching the following pattern:

```
if (num == EXECUTION_ORDER_NUMBER) {
  <code to extract>
}
```

Unfortunately, this solution has a problem. To understand what is wrong with it, we will take another look at the code in Figure 2. If `num == 13`, then we jump to line 24 via the `goto` statement on line 15. If we apply the pattern that we described above, we will not be able to detect code at lines 23-25 and include it into the deobfuscated code. While it is possible to devise an algorithm that would process these `goto` statements correctly, implementing such an algorithm could be tricky.

```
12  while (i < 5) {
13      int num = X1BFovEdxME9D.HBkAc6Pu((int[])array[i], 0, 0);
14      if (num == 13) {
15          goto IL_29A;
16      }
17      if (num == 121) {
18          <code omitted>
19      }
20      IL_2A1:
21      i++;
22      continue;
23      IL_29A:
24      RequestCachingSection requestCachingSection = (RequestCachingSection)obj;
25      goto IL_2A1;
26  }
```

Figure 5: Flattened code from Figure 2 with a problematic if statement.

Maybe a less intuitive but a more efficient algorithm can be devised if we visualize flattened code as a graph. This graph, which is commonly referred to as a *control flow graph*, consists of code blocks that are called *basic blocks*. This graph also contains arrows that are drawn whenever there is a branch (an if statement or a loop) in the code. To better understand what a control flow graph looks like, study the code in Figure 6 and its corresponding control flow graph in Figure 7.

```
1  object[] array2 = new object[] {...}
2  while (j < 3)
3  {
4      int num2 = X1BFovEdxME9D.HBkAc6Pu((int[])array2[j], 0, 0);
5      if (num2 == 11) {
6          while (Program.F1()) {
7              Thread.Sleep(1000);
8          }
9      } else if (num2 == 73) {
10         if (Program.F2()) {
11             Program.F3(0);
12         }
13     } else if (num2 == 67) {
14         ProgramClass.F4();
15     }
16     j++;
17 }
18 return;
```

Figure 6: Flattened code with conditions and loops.

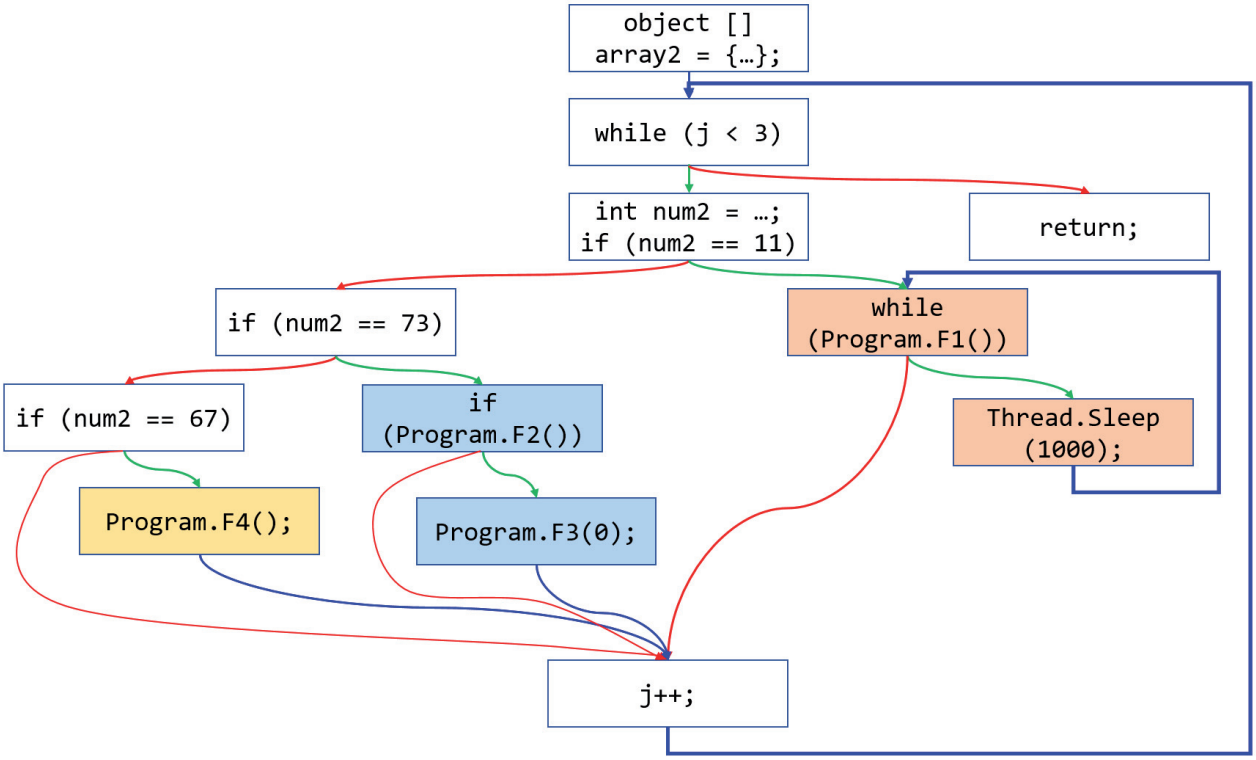

Figure 7: Control flow graph of the code in Figure 6. Flattened code is highlighted.

With control flow graphs, solving the problem of matching execution order numbers to code is easier. If we construct a control flow graph of a flattened function, we can use the following algorithm to perform matching:

1. Locate a basic block that contains a comparison of a num variable used for flattening (e.g. `if (num2 == 73)`).
2. Locate the block that is executed if the condition in step 1 is true. In the case of `num == 73`, this block contains the code `if (Program.F2())`.
3. Recursively traverse all blocks that can be reached from the block in step 2. When we reach an unvisited block, we assign it an execution order number from the comparison in step 1. If `num == 73`, we would visit the blocks that are highlighted in light blue in Figure 7.
4. If we reach the block that increments the index variable used for flattening (e.g. `j++` in Figure 7), we stop the traversal from step 3 in order not to fall into infinite recursion.

Once we match every number in the execution order to a set of blocks, we need to place all the blocks with flattened code in the correct order. As can be seen in Figure 7, some coloured blocks are connected to the block with `j++`. To perform

reordering, we will iterate over all blocks that are directly connected to the `j++` block. We will disconnect every such block from the `j++` block and instead connect it to the next coloured block in the execution order. For example, if the execution order array is `{67, 11, 73}`, we will connect the block with `Program.F4()` to the block with `while (Program.F1())`. As a result, we will get the following unflattened control flow graph:

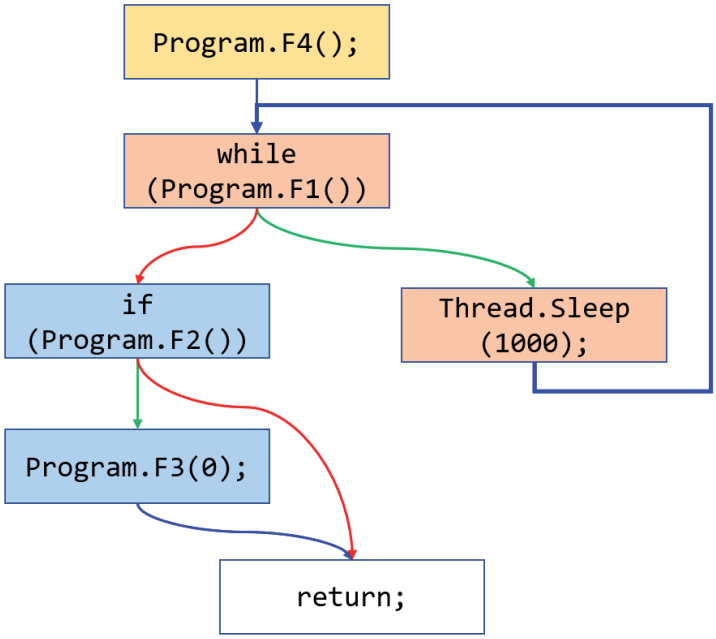

Figure 8: Unflattened control flow graph.

After reconnecting the blocks, all we have to do is convert the control flow graph back to C# code. The following is the unflattened code that corresponds to the control flow graph shown in Figure 8:

```
Program.F4();
while (Program.F1()) {
       Thread.Sleep(1000);
}
if (Program.F2()) {
       Program.F3(0);
}
return;
```

Now that we have discussed all the algorithms required to perform unflattening, we will proceed to the most interesting part: implementing the code that unflattens DoubleZero's code.

IMPLEMENTING CONTROL FLOW UNFLATTENING WITH DE4DOT

In order to implement control flow unflattening, we will use *de4dot*, an open source C# framework used for deobfuscation of .NET binaries. As we will see in this section, *de4dot* has all the capabilities that we need to implement unflattening algorithms. It is able to construct control flow graphs of functions, as well as perform reconnection of basic blocks, code patching and many other useful things. While explaining how to implement unflattening, we will provide a thorough description of *de4dot*'s components that come in useful for deobfuscation of .NET binaries. We also advise the reader to download the deobfuscator code from [3] and look at the code to better understand the concepts described in this section.

Adding our own deobfuscator to de4dot

To start working with *de4dot*, we first need to install a recent version of *Visual Studio* with the .NET desktop development package (the free Community edition suffices). Then, we need to download *de4dot*'s source code from its official repository, `https://github.com/de4dot/de4dot`. Finally, we need to open either the `de4dot.netcore.sln` or `de4dot.netframework.sln` file in *Visual Studio*.

In order to extend *de4dot*, we will add a new deobfuscator to the framework. To do that, we need to locate the Solution Explorer window, open the de4dot.code project and navigate to the `deobfuscators` folder. As we can see, this folder contains other folders with many deobfuscator implementations. To add our own deobfuscator, we need to create a subfolder called `DoubleZero` in the `deobfuscators` folder.

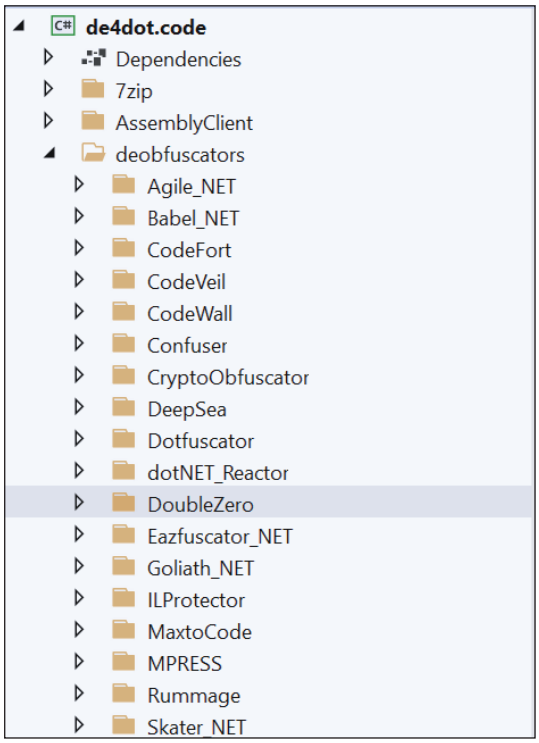

Figure 9: The Solution Explorer window with a new folder for DoubleZero's deobfuscator.

Once we create the folder, we need to add a code file that contains a description of our deobfuscator. To do that, we will use a template file that can be downloaded from [4]. This file should be placed in the created `DoubleZero` folder under the name `Deobfuscator.cs`. In order for the deobfuscator to work, we need to make the following changes to this file:

- Specify a unique namespace name for the deobfuscator (e.g. `de4dot.code.deobfuscators.DoubleZero`)
- Assign long (e.g. `DoubleZero Obfuscator`) and short (`dblz`) names to the deobfuscator. These names should be assigned to the `THE_NAME` and `THE_TYPE` variables, respectively.

The template file also defines the following functions that should be defined, but may not contain any useful code:

- `ScanForObfuscator`: a function that should check for the presence of the obfuscator that we are combating.
- `DetectInternal`: a function that should return a positive number if the binary is obfuscated, and zero if it isn't.
- `GetStringDecryptorMethods`: a function that should return methods that perform string decryption. We will not be dealing with string decryption while performing unflattening, so we make this function return an empty list.

Once we are done with filling the `Deobfuscator.cs` file, we should head to the `Program.cs` file located in the `de4dot.cui` project. In this file, we are interested in a function called `CreateDeobfuscatorInfos`. This function has a list called local that contains many calls to `DeobfuscatorInfo()`. In order for *de4dot* to recognize our newly created deobfuscator, we need to place the line `new de4dot.code.deobfuscators.DoubleZero.DeobfuscatorInfo()` into the list initializer.

```
    new de4dot.code.deobfuscators.dotNET_Reactor.v4.DeobfuscatorInfo(),
    new de4dot.code.deobfuscators.Eazfuscator_NET.DeobfuscatorInfo(),
    new de4dot.code.deobfuscators.Goliath_NET.DeobfuscatorInfo(),
    new de4dot.code.deobfuscators.ILProtector.DeobfuscatorInfo(),
    new de4dot.code.deobfuscators.MaxtoCode.DeobfuscatorInfo(),
    new de4dot.code.deobfuscators.MPRESS.DeobfuscatorInfo(),
    new de4dot.code.deobfuscators.Rummage.DeobfuscatorInfo(),
    new de4dot.code.deobfuscators.Skater_NET.DeobfuscatorInfo(),
    new de4dot.code.deobfuscators.SmartAssembly.DeobfuscatorInfo(),
    new de4dot.code.deobfuscators.Spices_Net.DeobfuscatorInfo(),
    new de4dot.code.deobfuscators.Xenocode.DeobfuscatorInfo(),
    new de4dot.code.deobfuscators.DoubleZero.DeobfuscatorInfo(),
};
```

Figure 10: How the code of the CreateDeobfuscatorInfos function should look after adding our deobfuscator to the list.

If everything is done correctly, the code should now compile. If we launch the compiled *de4dot* executable without any arguments, it should display information about our newly created deobfuscator.

```
Type dblz (DoubleZero Obfuscator)
  --dblz-name REGEX
                  Valid name regex pattern ((^<.*)|(^[a-zA-Z_<{$][a-zA-Z_0-9<>{}$.`-]*$))

String decrypter types
  none            Don't decrypt strings
  default         Use default string decrypter type (usually static)
  static          Use static string decrypter if available
  delegate        Use a delegate to call the real string decrypter
  emulate         Call real string decrypter and emulate certain instructions

Multiple regexes can be used if separated by '&'.
Use '!' if you want to invert the regex. Example: !^[a-z\d]{1,2}$&!^[A-Z]_\d+$&^[\w.]+$
```

Figure 11: Information about the deobfuscator that should be displayed by de4dot if everything so far has been done correctly.

De4dot's block deobfuscators

As our code unflattening algorithm relies on working with control flow graphs and basic blocks, the first thing we need is a function that can construct control flow graphs of functions. Luckily for us, we will not need to code everything from scratch. *De4dot* implements the `IBlocksDeobfuscator` interface, which allows control flow graphs of functions to be traversed and edited. The interface has the following functions:

```
public interface IBlocksDeobfuscator {
   bool ExecuteIfNotModified { get; }
   void DeobfuscateBegin(Blocks blocks);
   bool Deobfuscate(List<Block> allBlocks);
}
```

To use this interface, we should create a class that implements the `IBlocksDeobfuscator` interface. In particular, we are interested in the `Deobfuscate` method of the interface. This method is called for every function in the executable that we are deobfuscating. Its argument, `allBlocks`, contains a list of all basic blocks that the function has. In this function, we may edit the control flow graph as much as we want. In case we decide to modify our control flow graph, the function `Deobfuscate` should return true. Otherwise, it should return false.

Aside from our own block deobfuscator, *de4dot* implements its own block deobfuscators that perform optimizations such as unreachable code removal. While working on a function, *de4dot* calls all available block deobfuscators in a loop. If any available deobfuscator modifies the control flow graph, *de4dot* calls every deobfuscator again. The loop continues until all deobfuscators stop making changes to the control flow graph. The code snippet below should give additional understanding of how block deobfuscation works in *de4dot*.

```
do {
    iterations++;
    modified = false;
    RemoveDeadBlocks();
    MergeBlocks();

    blocks.MethodBlocks.GetAllBlocks(allBlocks);

    if (iterations == 0)
        modified |= FixDotfuscatorLoop();

    modified |= Deobfuscate(userBlocksDeobfuscators, allBlocks);
    modified |= Deobfuscate(ourBlocksDeobfuscators, allBlocks);
    modified |= DeobfuscateIfNotModified(modified, userBlocksDeobfuscators, allBlocks);
    modified |= DeobfuscateIfNotModified(modified, ourBlocksDeobfuscators, allBlocks);
} while (modified);
```

Figure 12: A snippet of de4dot's source code that calls available block deobfuscators in a loop.

In order to add a block deobfuscator, we need to head to our `Deobfuscator.cs` class and override the `BlockDeobfuscators` member inside the `Deobfuscator` class, as shown below:

```
public override IEnumerable<IBlocksDeobfuscator> BlocksDeobfuscators {
        get {
                var list = new List<IBlocksDeobfuscator>();
                list.Add(new Unflattener(this));
                return list;
        }
}
```

In the code above, `Unflattener` is a class that implements the `IBlocksDeobfuscator` interface.

After defining the `DeobfuscateBegin` method (which does nothing in our case) and the `ExecuteNotModified` variable (we set its `get` accessor to return false), we start working on the most important `Deobfuscate` method. This method will iterate over all basic blocks in the function code and try to locate flattened code and unflatten it.

Extracting the execution order array

As we remember from Figure 7, a layer of flattened code starts with an assignment to the execution order variable with type `object []`. In our code, we will iterate over all basic blocks with a range-based for loop and detect assignments to a variable with type `object []`. An important point to mention is that *de4dot*'s basic blocks contain not lines of code but Intermediate Language (IL) instructions. Intermediate Language can be thought of as a special assembly language for .NET binaries. Having to deal with IL instructions rather than lines of C# code is not much of a problem because we can match code lines to IL instructions with *dnSpy*. We can browse IL code by selecting IL in *dnSpy*'s top panel.

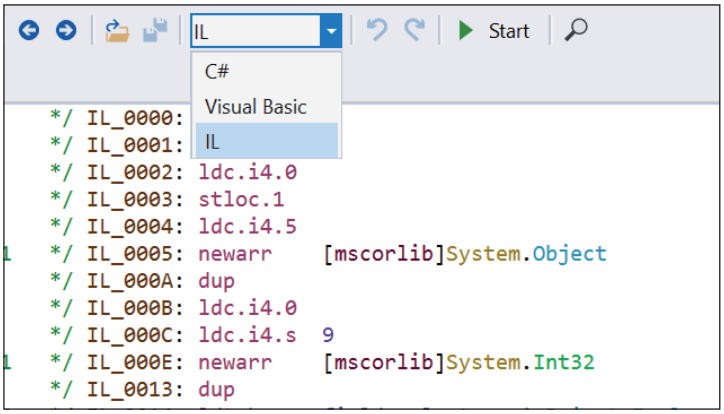

Figure 13: Switching to IL code in dnSpy.

If we compare the decompiled C# code and IL instructions, we will notice that the line `object [] array = new object [] {…}` is transformed into two IL instructions:

```
ldc.i4.s <number of members in the object [] array>
newarr [mscorlib]System.Object.
```

In order to detect assignments to the execution order array in *de4dot*, we will be looking for all `newarr` instructions with the `System.Object` operand. We can iterate over instructions in a block with the following code:

```
foreach (var instr in block.Instructions) {
   ...
}
```

De4dot allows us to examine the instruction opcodes and operands through the `instr.OpCode` and `instr.Operand` fields, respectively. Knowing that, we can write the following condition to detect the assignments we need:

```
instr.OpCode == OpCodes.Newarr && instr.Operand.ToString() == "System.Object"
```

Once we detect an assignment with the code above, we will attempt to extract the execution order from the `object []` array. By looking at the IL code, we will notice that `int []` arrays are added to the `object []` array with the following IL instructions:

```
ldtoken field valuetype <long line omitted>
call System.Runtime.CompilerServices.RuntimeHelpers::InitializeArray
stelem.ref
```

If we look at the documentation for the `ldtoken` instruction [5], we will find out that its operand contains a token. A token can be viewed as an analogy for an address of a global variable. In our case, this token references an `int []` array. We can get the array value with *de4dot* by accessing the `InitialValue` field of the instruction operand. The array that we get through `InitialField` has the type `byte []`, however, we need to get an array with type `int []`. To convert a `byte []` array to an `int []` array, we need to group every four bytes into a 32-bit little-endian integer. We should also recall that this array of integers represents an encrypted execution order number. To decrypt it, we can use the same strategy as we did while performing manual unflattening. We can copy the integer decryption function from *de4dot*, paste it in our code and then invoke it for the extracted integer array. By iterating over all instructions in the basic block and finding the pattern with the ldtoken instruction in it, we can successfully get all the other execution order numbers.

Matching execution order numbers to flattened code

Once we extract the execution order array from IL instructions, we can proceed to the second step: locating flattened code and matching it to numbers from the execution order. In order to do that, we need to understand how to traverse control flow graphs in *de4dot*. Apart from the `Instructions` list that we used earlier, the Block class has the following useful fields:

- `Block FallThrough`: the block that can be reached directly from the current block through an unconditional jump or a false branch of a condition.
- `List <Block> Targets`: a list of blocks that can be reached from the current block through a conditional jump.
- `List<Block> Sources`: a list of blocks that fall through or branch to the current block.

The illustration shown in Figure 14 helps to understand the relationship between these class fields.

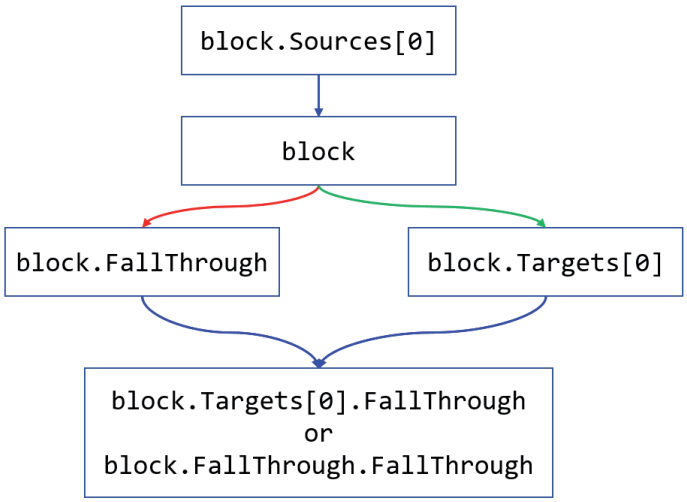

Figure 14: Relationship between blocks and their fallthrough blocks, sources and targets.

As we discussed in the previous section, flattened blocks of code branch out of blocks with if conditions that have the template `if (num == XX)`. If we look around DoubleZero's code, we will notice that the C# template we are looking for matches one of the three following IL instruction patterns:

Pattern 1	**Pattern 2**	**Pattern 3** `(if num == 0)`

`ldloc.s <variable>`
`ldc.i4.s <number from the execution order>`
`beq <flattened code>`
	`ldloc.s <variable>`
	`ldc.i4.s <number from the execution order>`
	`bne <not flattened code>`
		`ldloc.s <variable>`
		`brfalse <flattened code>`

In order to locate such if conditions, we create a function that traverses our control flow graph recursively. For every block that we visit, we first check whether it contains one of the three patterns above. Just like in the case with the execution order array, we handle pattern detection with the help of the Instructions field. If we get a match, we:

1. Extract the number from the execution order from the operand of the `ldc.i4.s` instruction (in the case of Pattern 1 or 2). In the case of Pattern 3, this number is 0.
2. Go either to the true (in case the of Pattern 1 or 3) or false (in case the of Pattern 2) branch of the matched block.
3. Add the block that we reached and its execution order number to a dictionary that we will call the *dictionary with if condition targets*. We will need this dictionary later when performing block reconnection.

4. Do another recursive traversal of the control flow graph, starting from the if condition target block. In this new traversal, we do not look for any patterns. Rather, we simply add every block and its execution order number from step 1 into another dictionary. From now on we will call this dictionary *the dictionary with all flattened blocks*.

In both cases, we have to stop the recursion every time we reach a block that increments the variable used for indexing the execution order array. To locate this block with *de4dot*, in accordance with Figure 7, we have to:

1. Go to the fallthrough block of the block that initializes the execution order array. By doing that, we reach the block that checks the condition of the while loop.
2. Iterate over the `Sources` list of the block that we reached in step 1. As we can see from Figure 7, this block has exactly two source blocks: the block with the assignment and the block with the increment.
3. Filter out the block with the assignment. By doing that, we will be left only with the block containing the increment.

After completing the recursion, we will have gathered two dictionaries:

- The dictionary with all flattened blocks
- The dictionary with if condition target blocks.

We will now use these two dictionaries to perform block reconnection.

Reconnecting flattened blocks

Surprisingly enough, it is very simple to implement block reconnection with *de4dot*. If, for example, we were to connect two blocks in a disassembler like *IDA* or *Ghidra*, we would need write a script that could inject jump instructions into blocks. With *de4dot*, we can implement reconnection with just one of the two lines of code:

- `sourceBlock.SetNewFallthrough(targetBlock)` (if we want the blocks to be connected through an unconditional jump)
- `sourceBlock.SetNewTarget(i, targetBlock)` (if we want the blocks to be connected through a conditional jump). In this code, `i` should be a valid index in the Targets list.

We will start the process of block reordering by connecting the block that initializes the execution order array to the first block in the execution order. In order to get the first block in the execution order, we have to:

1. Extract the number at index 0 from the execution order array.
2. Find the block in the if condition targets dictionary that corresponds to the number we have just extracted.

Then, we will iterate over the Sources list of the block responsible for incrementing the index variable that is used with the execution order array (reminder: in Figure 7, this block does the `j++` increment). For every block in this array, we:

1. Locate the execution order number of the current block through the dictionary with all flattened blocks.
2. Locate the index `i` that corresponds to this number in the execution order array.
3. Get the number at index `i + 1` of the execution order array.
4. Locate the block in the if condition targets dictionary that has the execution number from step 3.
5. Connect the current block with the block that we found in step 4.

Note that if the number in step 3 doesn't exist (this happens when `i` is the maximum available index in the execution order array), instead of performing steps 4 and 5 we need to connect the current block with the block executed after the flattening loop. In Figure 7, this block executes the return statement.

Once we are done with reconnecting blocks, we are finished with the unflattening process. We can now call our deobfuscator via the `de4dot.exe <path to the flattened DoubleZero sample> -p dblz` command line. *De4dot* should store the deobfuscated file at the `<path to the flattened DoubleZero sample>-cleaned` file. If we open it in *dnSpy*, we should see no flattened code in it.

CONCLUSION

In this paper, we looked at how we can perform control flow unflattening of the DoubleZero wiper. We described the unflattening algorithm and then implemented it using the *de4dot* framework. We also discussed how to create deobfuscators for *de4dot*, construct and traverse control flow graphs, as well as reorder basic blocks in them. Thanks to the fact that *de4dot* supports all these features out of the box, the implementation of the unflattening algorithm contains about 300 lines of code, which can be considered quite compact.

REFERENCES

[1] Rolles, R. Hex-Rays Microcode API vs. Obfuscating Compiler. Hey-Rays. September 2018. https://hex-rays.com/blog/hex-rays-microcode-api-vs-obfuscating-compiler/.

[2] DoubleZero sample. https://samples.vx-underground.org/APTs/2022/2022.03.22/Samples/30b3cbe8817ed75d8221059e4be35d5624bd6b5dc921d4991a7adc4c3eb5de4a.7z.

[3] De4dot code with the DoubleZero deobfuscator. https://github.com/gkucherin/de4dot.

[4] Deobfuscator template file. https://github.com/gkucherin/de4dot/blob/master/DeobfuscatorTemplate.cs.

[5] Documentation for the ldtoken instruction. https://docs.microsoft.com/dotnet/api/system.reflection.emit.opcodes.ldtoken.