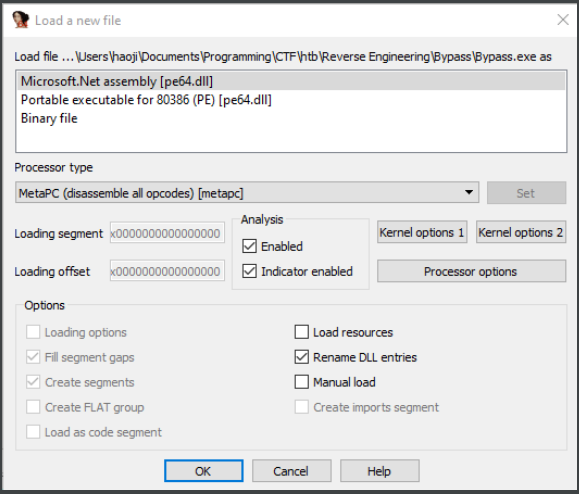
- dnSpy
- Windows Operating System

ANALYSIS

When I first open up the program in IDA Pro, I noticed that it is a .NET program that requires a different decompiler (see Fig 4a). The decompiler I used is dnSpy which includes a debugger in the decompiler, making it very useful.



Load a new file

Load file ...\Users\haoji\Documents\Programming\CTF\htb\Reverse Engineering\Bypass\Bypass.exe as

```
Microsoft.Net assembly [pe64.dll]
Portable executable for 80386 (PE) [pe64.dll]
Binary file
```

Processor type: MetaPC (disassemble all opcodes) [metapc]

Loading segment: 0x0000000000000000

Loading offset: 0x0000000000000000

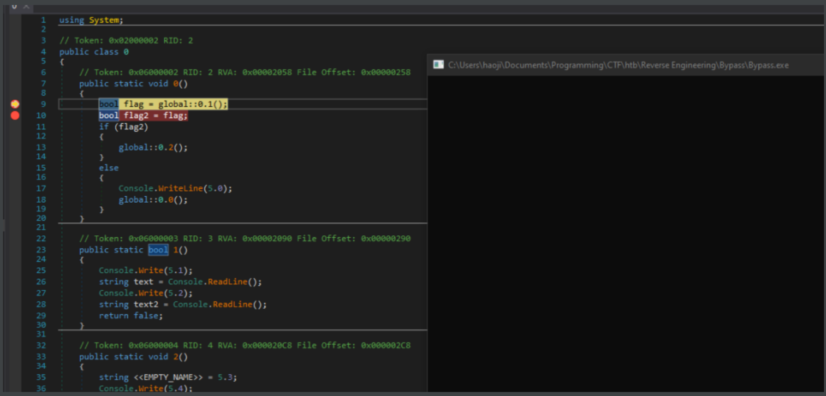
Analysis: [✓ Enabled] [✓ Indicator enabled]

Options:
- [ ] Loading options
- [✓] Fill segment gaps
- [✓] Create segments
- [ ] Create FLAT group
- [ ] Load as code segment
- [ ] Load resources
- [✓] Rename DLL entries
- [ ] Manual load
- [ ] Create imports segment

Fig 4a. .NET file detected by IDA

While trying to run the file in *dnSpy*, I noticed that it is a 32-bit which requires us to run the x86 version of *dnSpy.exe*. Therefore, make sure you open the correct version before continuing the write-up.

Setting the breakpoint at the start of the program as well as the line of code after it, we can see that the program actually stops there before prompting us to input the username. This indicates that *Class 0* is indeed the entry class.



```
1   using System;
2
3   // Token: 0x02000002 RID: 2
4   public class 0
5   {
6       // Token: 0x06000002 RID: 2 RVA: 0x00002058 File Offset: 0x00000258
7       public static void 0()
8       {
9           bool flag = global::0.1();
10          bool flag2 = flag;
11          if (flag2)
12          {
13              global::0.2();
14          }
15          else
16          {
17              Console.WriteLine(5.0);
18              global::0.0();
19          }
20      }
21
22      // Token: 0x06000003 RID: 3 RVA: 0x00002090 File Offset: 0x00000290
23      public static bool 1()
24      {
25          Console.Write(5.1);
26          string text = Console.ReadLine();
27          Console.Write(5.2);
28          string text2 = Console.ReadLine();
29          return false;
30      }
31
32      // Token: 0x06000004 RID: 4 RVA: 0x000020C8 File Offset: 0x000002C8
33      public static void 2()
34      {
35          string <<EMPTY_NAME>> = 5.3;
36          Console.Write(5.4);
```

C:\Users\haoji\Documents\Programming\CTF\htb\Reverse Engineering\Bypass\Bypass.exe

Fig 4b. Breakpoint at line 9 triggered before prompting for username

If we look at the code in line 9, we will notice that it calls for *1()* which is from the same class. Looking at the code in *1()*, it is quite obvious that the 2 *Console.write()* and *Console.read()* will prompt for username and password and taking in our input.

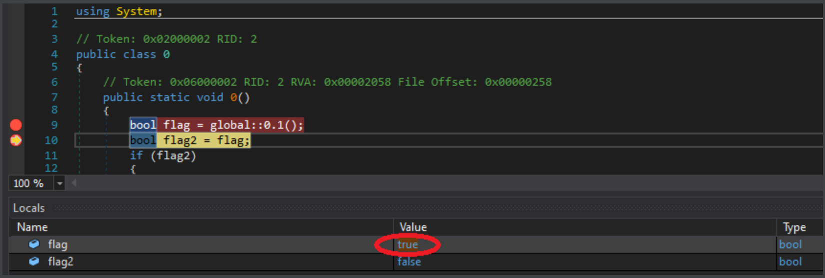
```
public static bool 1()
{
        Console.Write(5.1);
        string text = Console.ReadLine();
        Console.Write(5.2);
        string text2 = Console.ReadLine();
        return false;
}
```

If we look at *1()* carefully, we will notice that the result will always return FALSE. Therefore, there is no correct username and password.

We can just input a random username and password then depend on the breakpoint at line 10 to change the value returned.

```
Enter a username: a
Enter a password: a
```

Once the breakpoint hits line 10, the control will be given back to dnSpy. We can then change the value of *flag1* since the breakpoint hits before the line of code execute (see Fig 4c). Thus, changing *flag1* to True at line 10's breakpoint will cause *flag2* to be assigned to TRUE from *flag1* once we continue the execution of the debugger.



```
1   using System;
2
3   // Token: 0x02000002 RID: 2
4   public class 0
5   {
6       // Token: 0x06000002 RID: 2 RVA: 0x00002058 File Offset: 0x00000258
7       public static void 0()
8       {
9           bool flag = global::0.1();
10          bool flag2 = flag;
11          if (flag2)
12          {
```

100 %

Locals

Name	Value	Type
flag	true	bool
flag2	false	bool

Fig 4c. Modify *flag1*'s value from FALSE to TRUE

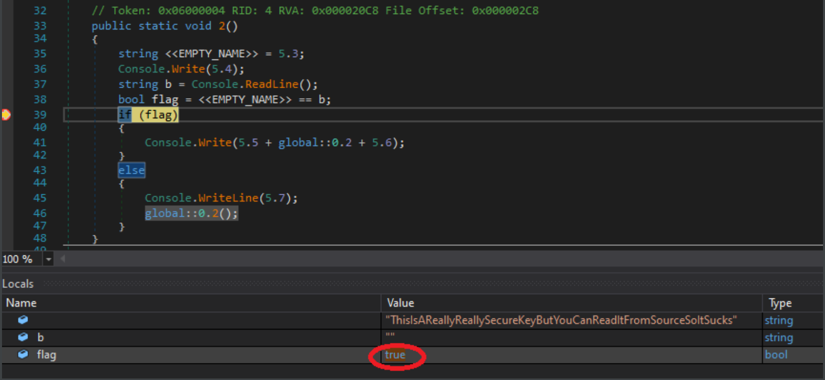
Once we continue the execution, we will notice that we are prompted to input the secret key.

```
Please Enter the secret Key:
```

If we look at *2()* in *class 0* which was called in line 13, we will notice that another prompt occurs where the result will be checked against *<<EMPTY_NAME>>* whose value comes from *class 5*'s static variable *3*.

```
public static void 2()
{
        string <<EMPTY_NAME>> = 5.3;
        Console.Write(5.4);
        string b = Console.ReadLine();
        bool flag = <<EMPTY_NAME>> == b;
        if (flag)
        {
                Console.Write(5.5 + global::0.2 + 5.6);
        }
        else
        {
                Console.WriteLine(5.7);
                global::0.2();
        }
}
```

Looking at *class 5*'s variable *3*, it gets more and more complex where it retrieves the value from a binary reader. Luckily we have dnSpy where we can use our debugger to get the value from. Alternatively, we can just set the breakpoint at line 13 which is right at the IF statement which compares the value of the variable *flag*. Once the breakpoint hits, we can modify its value to true.



```
32      // Token: 0x06000004 RID: 4 RVA: 0x000020C8 File Offset: 0x000002C8
33      public static void 2()
34      {
35          string <<EMPTY_NAME>> = 5.3;
36          Console.Write(5.4);
37          string b = Console.ReadLine();
38          bool flag = <<EMPTY_NAME>> == b;
39          if (flag)
40          {
41              Console.Write(5.5 + global::0.2 + 5.6);
42          }
43          else
44          {
45              Console.WriteLine(5.7);
46              global::0.2();
47          }
48      }
```

Name	Value	Type
	"ThisIsAReallyReallySecureKeyButYouCanReadItFromSourceSoItSucks"	string
b	""	string
flag	true	bool

Fig 4d. Modify *flag* variable to TRUE

In line 41, the flag will be printed. As the program will close immediately after the flag is printed since the Bypass program is spawned by dnSpy's debugger, we will need to do Step-over execution instead of pressing the "Continue" button. To Step-over execution, press F10 on your keyboard. Keep pressing and observe the console, the flag will eventually be printed on the console.

FLAG OBTAINED

```
Enter a username: a
Enter a password: a
Please Enter the secret Key: a
Nice here is the Flag:HTB{SuP3rC00lFL4g}
```

Flag: **HTB{SuP3rCoolFL4g}**