

Bypass

Capture the Flag Challenge.

Link: Challenge can be found [here](#).

Overview: The task of this CTF is to Extract the Flag from windows exe, that is achieved by bypassing the authentications safeguards.

Method: Upon downloading the Challenge – the first thing I notice is in this challenge I have to crack a Windows executable rather than Linux ELF file. So it requires different investigation methods and different tools to deal with it.

The first order of business is to static analyze the file – understand with what system it was compile and does it runs on 32 bits or 64 bits.

So, I checked the required information with ‘file Bypass.exe’:

```
amit@amit-VirtualBox: ~/Downloads/capture the flags/Bypass
amit@amit-VirtualBox:~/Downloads/capture the flags/Bypass$ file Bypass.exe
Bypass.exe: PE32 executable (console) Intel 80386 Mono/.Net assembly, for MS Windows
amit@amit-VirtualBox:~/Downloads/capture the flags/Bypass$
```

The output tells that the executable is 32bit .NET program. Meaning I will have to use .NET disassemble/decompile tools in order to reverse engineer the exe Program.

I also tried to run ‘strings’ on the exe in order to check what strings it contains:

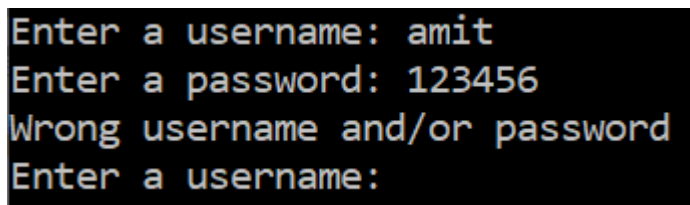
```
amit@amit-VirtualBox: ~/Downloads/capture the flags/Bypass
amit@amit-VirtualBox:~/Downloads/capture the flags/Bypass$ strings Bypass.exe
.text
.farc
0:relloc
t #0;
2901f
3:001V7h
Vg1q9
55Eu
21105
Gz0Qmk
1C(2na
15w:c
EEh:5
4 0x
2:454
34: n\
1077
D10G
10p1
PK:a
1EE:
1#ET*5
3*LB3
1ae]
vwr sg =
B53B
w4.0.30319
#strings
#GUID
#tob
ReadInt32
ReadInt64
#module:
System.IO
get_IV
GenerateIV
mscorlib
Read
N:UnderManaged
set_Mode
CryptoStreamMode
CipherMode
get_Unicode
HTBChallenge
IDisposable
ReadDouble
ReadSingle
Console
ReadLine
WriteLine
Dispose
Write
GuidAttribute
AttributeUsageAttribute
 ComVisibleAttribute
AssemblyTitleAttribute
AssemblyTrademarkAttribute
TargetFrameworkAttribute
AssemblyFileVersionAttribute
AssemblyConfigurationAttribute
AssemblyDescriptionAttribute
CompilationRelaxationsAttribute
AssemblyProductAttribute
AssemblyCopyrightAttribute
AssemblyCompanyAttribute
RuntimeCompatibilityAttribute
ReadSByte
HTBChallenge.exe
set_BlockSize
Encoding
System.Runtime.Versioning
ReadString
get_Length
GetManifestResourceStream
CryptoStream
```

The immediate thing I can, or rather can not see from the strings list – is the flag is not listed among the strings.

Rather what I can see is the first ‘words’ are unintelligible parts of what seems to be encrypted memory, that is implied that further below there are words such as ‘CipherMode’ or ‘CryptoStreamMode’ that implies some sort of memory decryption, for the exact purpose the flag can would not be directly obtained using static analysis methods.

More key words that were found implies that the program structure is to accept user input, and output some content in accordance of the input.

The next thing I tried is to run the program, initially it was tried in Virtual Machine due to safety reasons. Upon running I was requested to enter username and password.

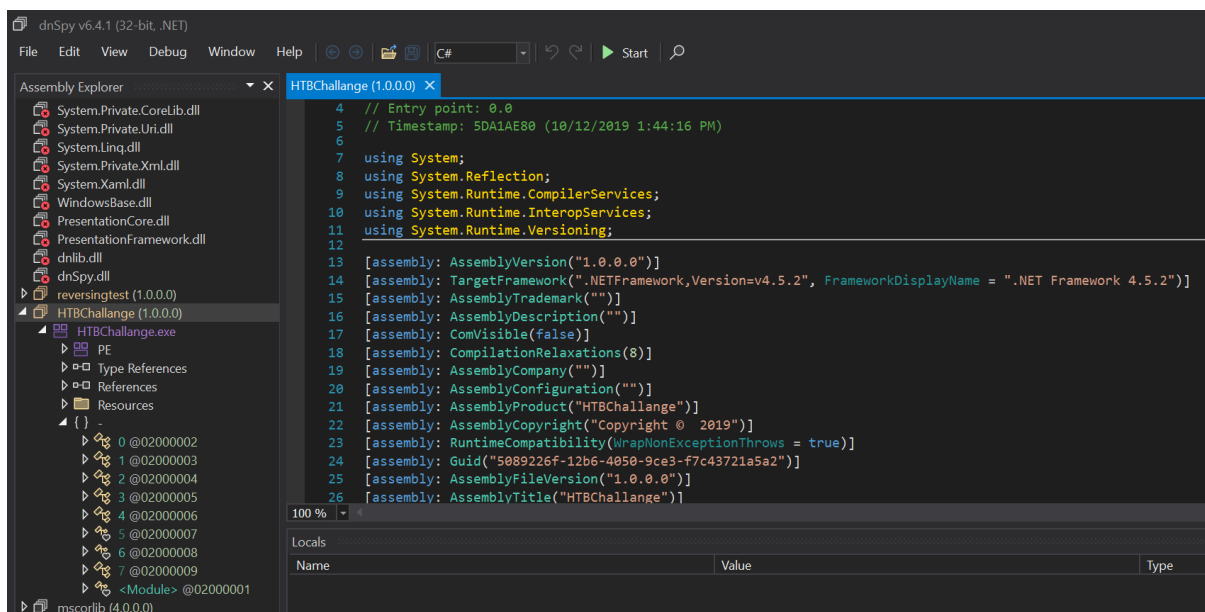


```
Enter a username: amit
Enter a password: 123456
Wrong username and/or password
Enter a username:
```

At this stage I of course don’t know the required credentials. So more in-depth analysis is required – Decompile the program.

The tool of choice in order to Decompile the program is ‘DnSpy-32’ - .NET debugger and assembly Editor.

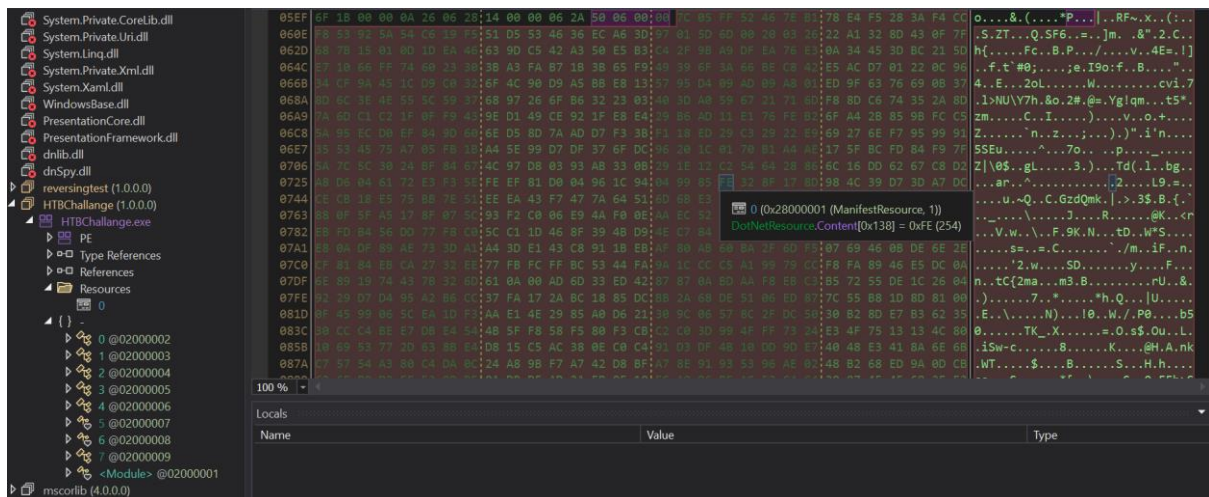
I inserted the Bypass.exe to the ‘Dnspy’ editor:



And started to looking around:

It seems upon first glance that the exe has 2 main components:

The first one is the resources library that has some file called '0' that leads us to some memory section within the program:



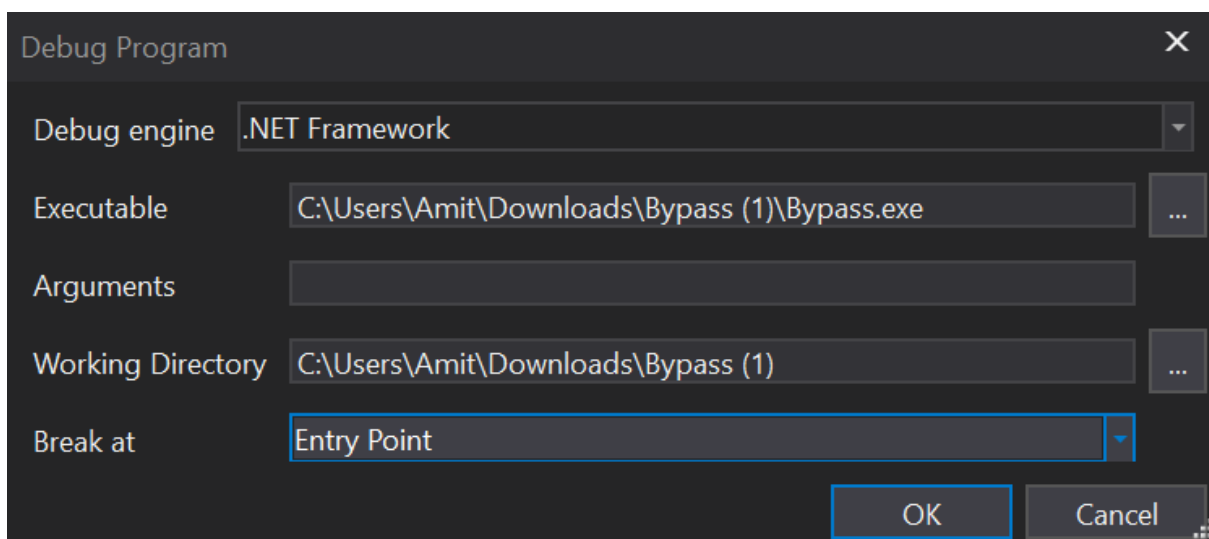
It seems there stored the encrypted memory containing the flag and other relevant data for the software run.

The second one is the code itself:

The code contains some different classes named with numbers for obfuscation, and no 'main' is visible.

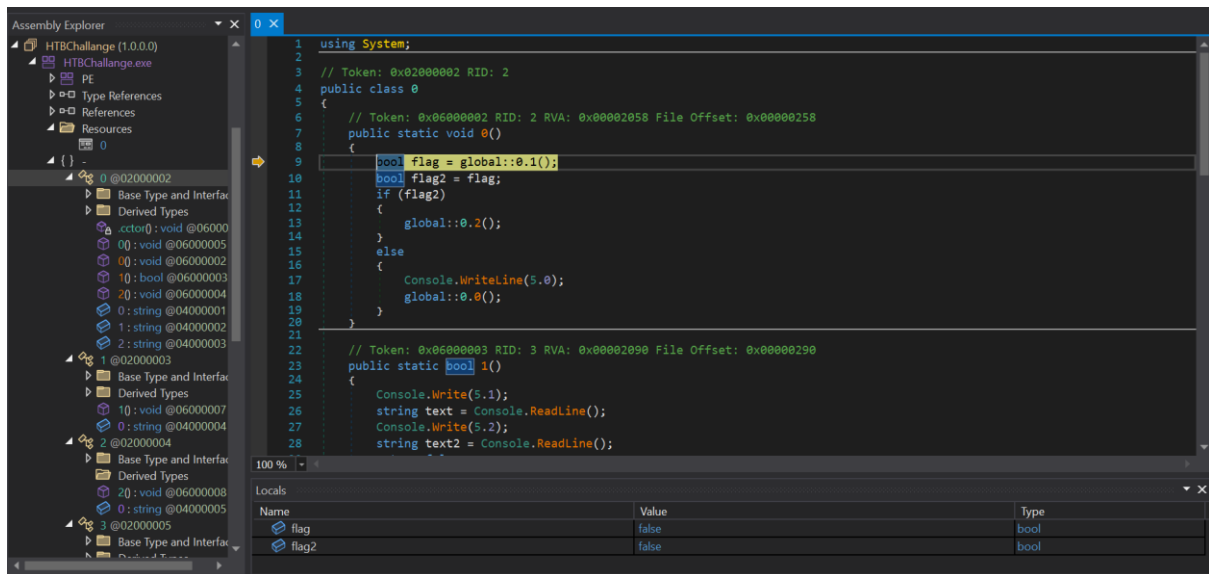
The First thing I needed is to discover the entry point in the program.

So I tried to run the program on debugging mode:



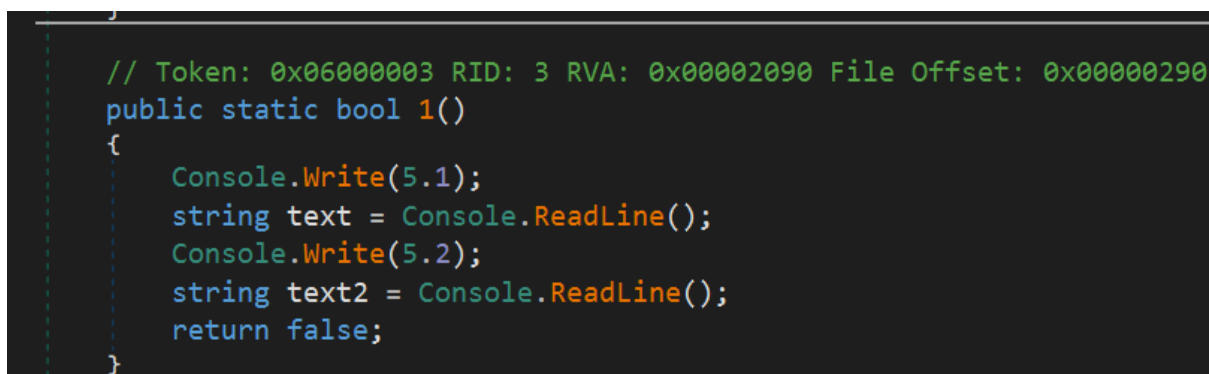
Such that it will break on entry point, and reveals it.

On debugging: I immediately observed that the entry point of the program is the class '0'.



And upon initialization – 2 variables are assigned: flag and flag2, both are Boolean variables initially set to 'false'.

Before proceeding with the debugging – I observed that the flag value will be assigned according to the returned value of function '0.1' – so I took a look at that:

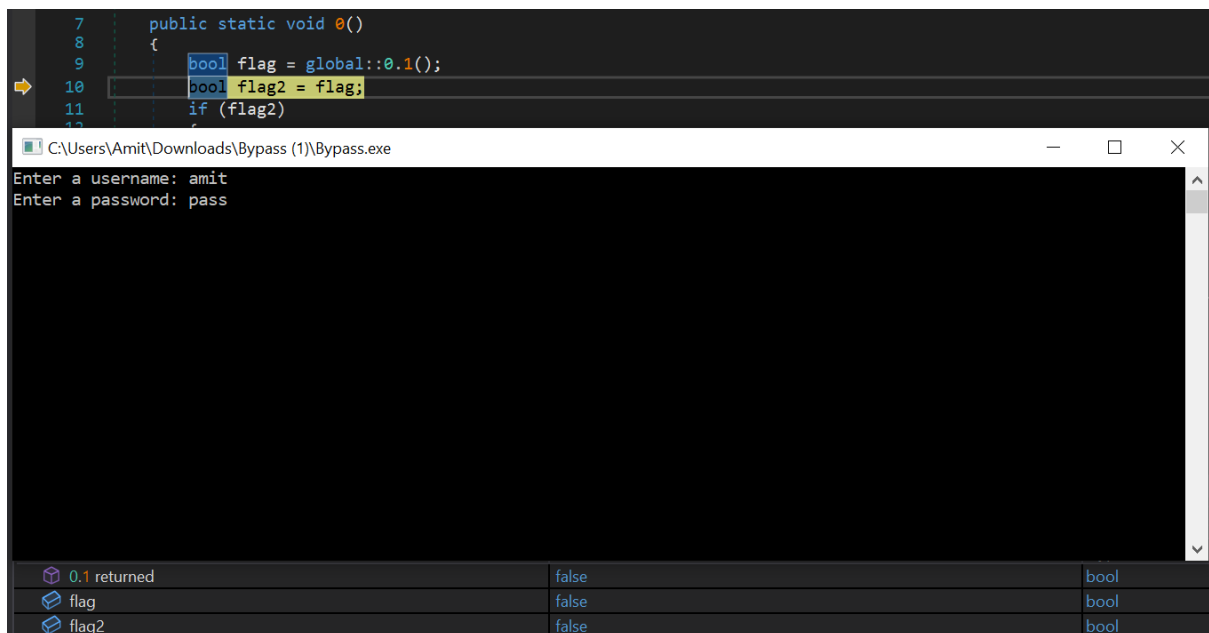


The function takes 2 inputs, and returns false.

As on running I was required to enter username and password – I inferred that the required inputs of the function corresponds to the username and password.

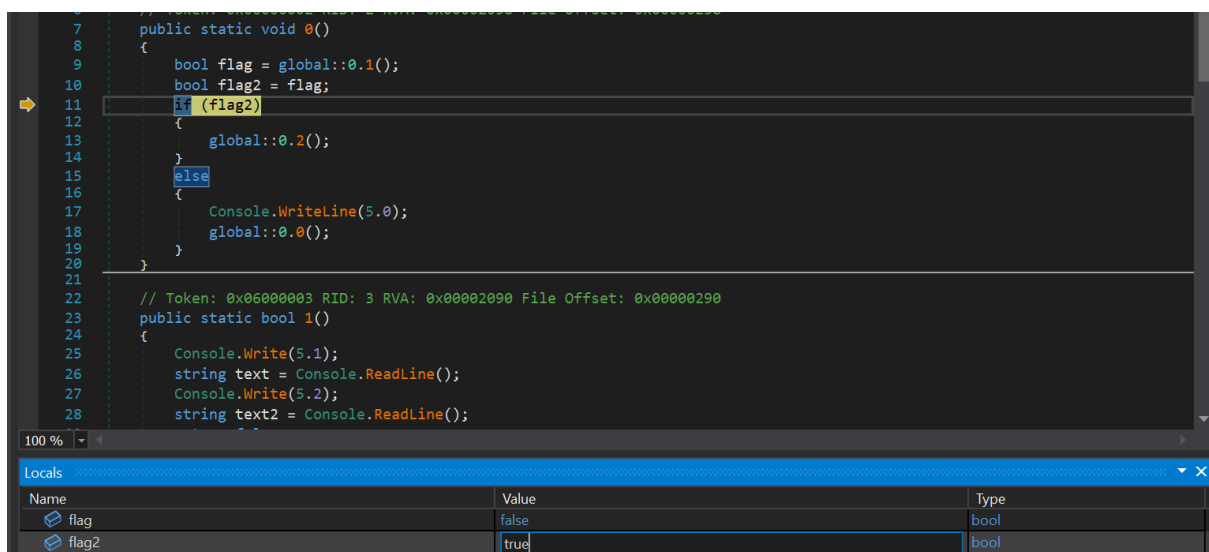
The conclusion from this is clear – there are not any correct username and password in order to obtain the flag.

So in order to proceed I might require a different method.

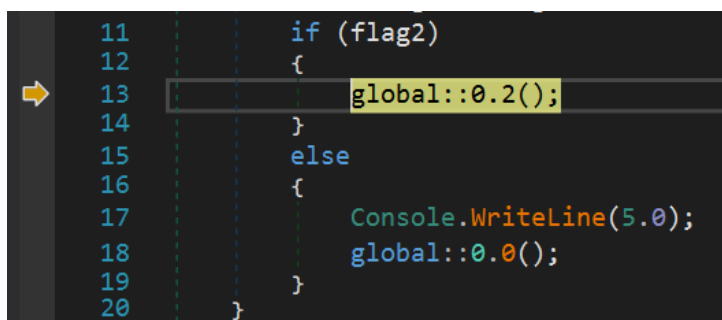


Upon entering the credentials I indeed observed that function '0.1()' returns false. And flag and flag2 variables accordingly are being assigned to false.

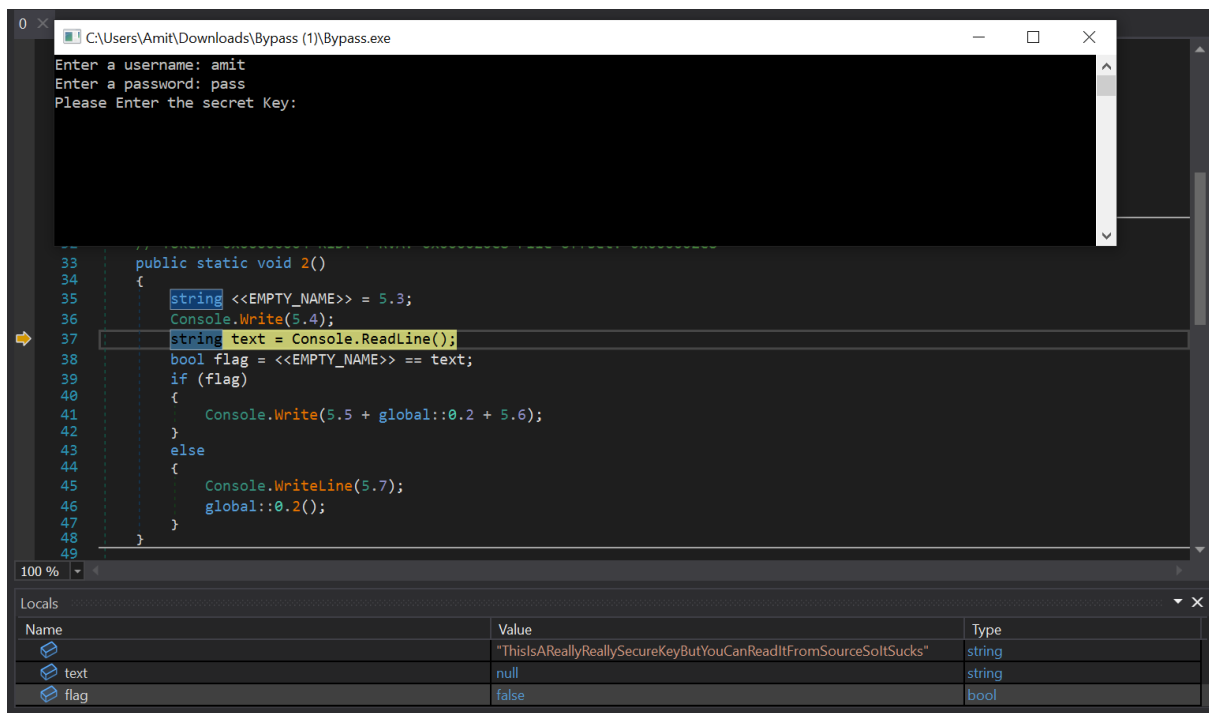
So in the next line – when the program checking flag2 value – I manually modified it to true:



It worked, and I got inside the 'true block':



I stepped in to '0.2()' function and advanced few steps:



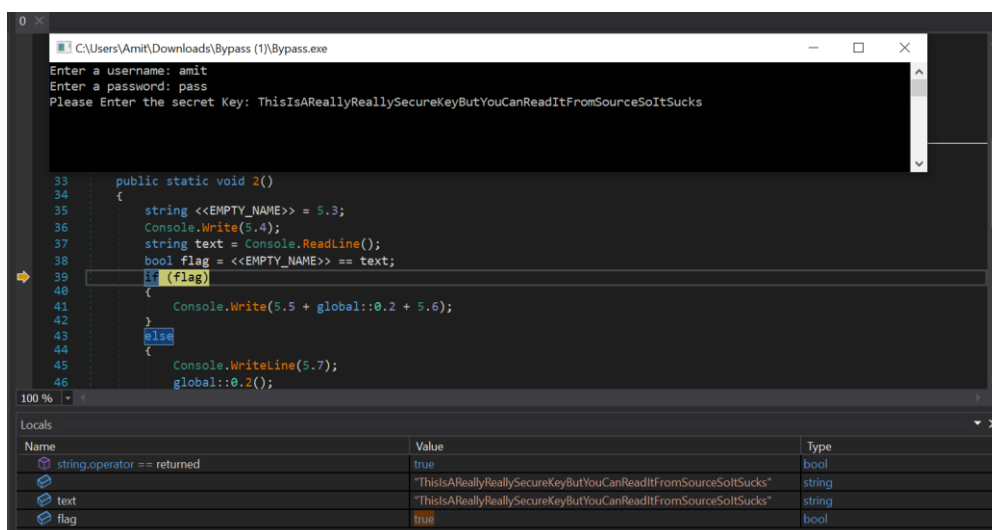
It seems the next stage is to enter secret key.

It also appears the secret key is being retrieved from variable '5.3' and its value appears right below on 'locals'. As the variable name is '<<EMPTY_NAME>>' – the variable itself on locals is nameless.

Now there are 2 ways to proceed:

- Enter the value.
- Modify the flag value to true upon condition checking.

I chose option a:



It works, flag value is set to true.

I entered the true block and...

```
C:\Users\Amit\Downloads\Bypass (1)\Bypass.exe
Enter a username: amit
Enter a password: pass
Please Enter the secret Key: ThisIsAreallyReallySecureKeyButYouCanReadItFromSourceSoItSucks
Nice here is the Flag:HTB{SuP3rC00lFL4g}
```

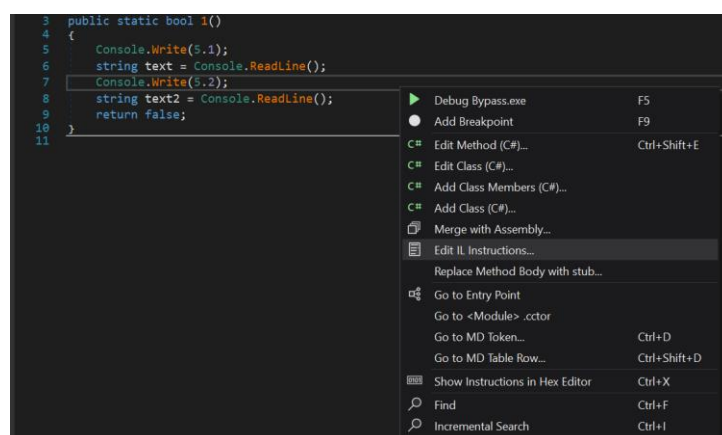
Got indeed the flag!

An alternative way to bypass stage 1 – the ‘0.1’ function

```
// Token: 0x06000003 RID: 3 RVA: 0x00002090 File Offset: 0x00002090
public static bool 1()
{
    Console.Write(5.1);
    string text = Console.ReadLine();
    Console.Write(5.2);
    string text2 = Console.ReadLine();
    return false;
}
```

Is to modify the assembly command of the function such that the false value will be true.

It can be done with right click on method and selecting ‘Edit IL instructions’



Here we can see the assembly commands of the function:

Index	Offset	OpCode	Operand
0	0000	nop	
1	0001	ldsflld	string '5'::'1'
2	0006	call	void [mscorlib]System.Console::Write(string)
3	000B	nop	
4	000C	call	string [mscorlib]System.Console::ReadLine()
5	0011	stloc.0	
6	0012	ldsflld	string '5'::'2'
7	0017	call	void [mscorlib]System.Console::Write(string)
8	001C	nop	
9	001D	call	string [mscorlib]System.Console::ReadLine()
10	0022	stloc.1	
11	0023	ldc.i4.0	
12	0024	stloc.2	
13	0025	br.s	14 (0027) ldloc.2
14	0027	ldloc.2	
15	0028	ret	

What we have to do in order to modify the 'return false' to 'return true' – is to change line 11 command from 'ldc.i4.0' to 'ldc.i4.1'.

After it is done and saved:

```

3  public static bool 1()
4  {
5      Console.Write(5.1);
6      string text = Console.ReadLine();
7      Console.Write(5.2);
8      string text2 = Console.ReadLine();
9      return true;
10 }
11

```

Now the credentials authentication function will return true for any username and password.

After saving it and running it normally without debugging:

```

Enter a username: amit
Enter a password: pass
Please Enter the secret Key:

```

It worked! The custom credentials worked.

So, both methods – opcode modification and runtime variable modification during debugging can be used to bypass the initial authentication.

Conclusions:

The challenge was the first challenge I tried with exe reverse engineering.

In which I had to prior experience.

Most of the time was in researching the right tool and method in order to reverse engineer .net executable.

First, I tried x32gdb, which disassemble the program to its assembly opcodes and work with the same way I did with Linux ELF executables.

However, it proved to be too complex and time costly.

After some online researching I came to the conclusion that decompiling the executable with dnSpy32 is much easier.

It took me a bit of time to learn to work with the dnSpy32. To see what it can do, how to modify assembly values, how to read the decompiled code.

Eventually the method to solve the challenge is to bypass code sections by modifying variables values during runtime. This method was also used in Linux challenges when I modified registered and memory values during runtime in order to reach the desired code section.

However, in here the debugging was more 'high level' as the tool did the hard work of decompile the assembly code for me, rather than deal with the assembly, registers and opcodes myself. But that's a start.

I hope in future .NET challenges to deal with the assembly itself.