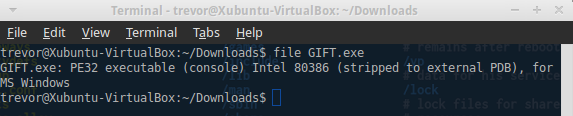
**Intro To Reverse Engineering - ILostThePassword Solution**

**Introduction:** ILostThePassword is an easy level challenge designed to teach students to perform basic reverse engineering analysis against a target binary. In this specific case we’ll be reversing the binary to find the password needed to retrieve the flag.

**Task:** Find the required password to retrieve the flag by disassembling the binary. There are many disassemblers out there, such as: [Hopper](https://www.hopperapp.com/), [GDB](https://www.gnu.org/software/gdb/), [IDA-Pro](https://www.hex-rays.com/products/ida/), [Immunity](https://www.immunityinc.com/products/debugger/), and [Radare2](http://radare.org/r/) just to name a few. There are also online debuggers, such as: [ODA](https://www.onlinedisassembler.com/odaweb/) and [Retargetable](https://retdec.com/decompilation/). Personally I would recommend using Hopper, it features an easy to use interface, and has all the features we will need for this challenge, and others under the demo version.

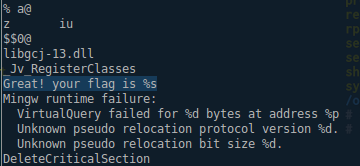
**Solving:** Today I’ll show you how to solve this challenge using [Hopper](https://www.hopperapp.com/). However, you’re free to use the disassembler of your choice. But first, let’s begin with a basic analysis.

Executing the ‘file’ command gives us the following information:



The binary we’ll be reversing today is a PE32 executable based on the [Intel 80386](https://en.wikipedia.org/wiki/Intel_80386) architecture for MS Windows, or in layman's terms an EXE which will run on Windows. Which means we’ll need access to a Windows machine if we want to run the binary.

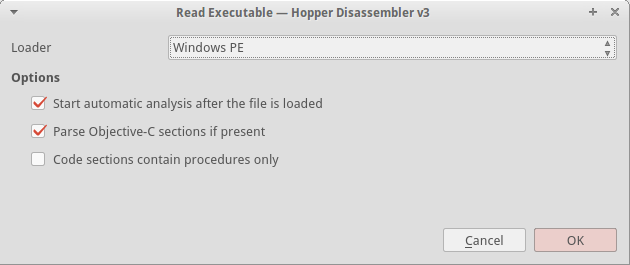
Executing the ‘strings’ command gives us the following information:



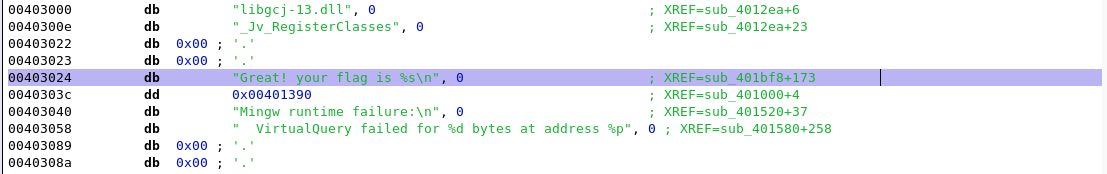
Above is a snippet containing the important information produced by the ‘strings’ command. The most important piece of info is highlighted: “Great! Your flag is %s”. Which means that string is somewhere in the program.

**Using Hopper:**

Let’s begin by opening Hopper then loading the executable, the default settings should be just fine.

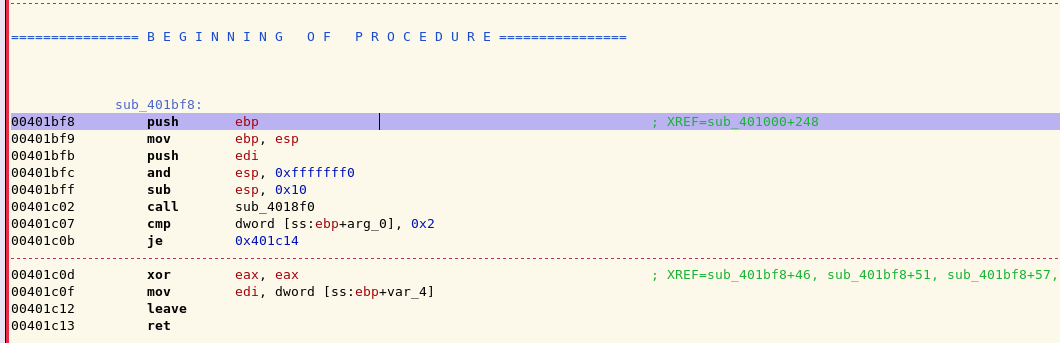


Once you click ‘OK’ hopper should disassemble the executable and drop us in at the EntryPoint which is listed under the ‘labels’ tab. I said previously that the string “Great! Your flag is %s” would be a good starting point. Let’s see if we can’t find it inside of the executable. Switch over to the ‘strings’ tab in Hopper and you should see it. Click on the string and Hopper will jump to its reference in memory.

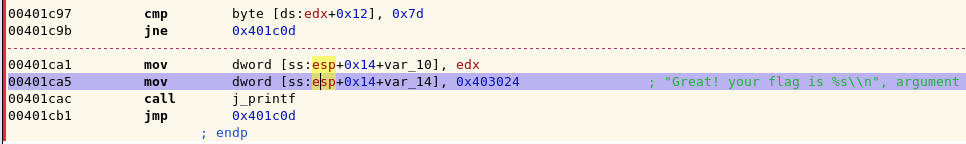


At address 0x00403024 you should see the string, off to the right you should see the information “XREF=sub\_401bf8+173”. There are two pieces of information that line tells us. First, this string is referenced at function stub 401bf8. Functions in the disassembler are referred to as ‘sub\_xxxxx’ which stands for subroutine. This is because when we compile a program the compiler doesn’t save user named functions. Second the ‘+173’ portion tells us the exact offset in the subroutine where the string is used.

Let’s navigate to sub\_401bf8 by switching to the ‘labels’ tab in Hopper and clicking on the subroutine. Hopper should then jump to the beginning of the subroutine. If you followed those steps correctly your screen should look something like the following.



Now let’s try and find the string “Great! Your flag is %s” within this subroutine. Start by scrolling down through the subroutine, don’t worry about the particular instructions you’re scrolling past at the moment, I’ll explain those later. Around address 0x00401ca5 we’ll find what we’re looking for.



Good, we’ve found a function / subroutine that will print out the string “Great your flag is %s”, chances are this subroutine is responsible for printing the flag we need to pass the challenge, let’s get into the meat of this challenge. From the previous highlighted address begin to scroll up and look closely at the instructions being executed, you should notice a pattern.



Hopefully you’ve noticed a large amount of ‘cmp’ instructions followed immediately by a ‘jne’ instruction. Let me break down what’s happening here. Basically, when the program executes the current subroutine each one of these ‘cmp’ (compare) instructions will be executed. If any one of the comparison returns false then the next ‘jne’ (jump not equal) instruction will be executed and the program will stop executing the current subroutine and return to executing instructions elsewhere in the program.

To give a more concrete example let's examine the ‘cmp byte [ds:edx], 0x45’ instruction located at address 0x00401c28 from the previous screenshot. When this instruction is executed by the program a comparison will be made between the hardcoded value ‘0x45’ and whatever happens to be stored in the edx register. More likely than not the edx register contains user input. If the comparison returns false then the jne (jump not equal) instruction located directly after the compare will be executed and the program will leave the current subroutine.

Here’s the most important part though. We can see the exact values our input is being compared against. ‘0x45’ is hexadecimal for the ASCII character ‘E’, ‘0x4b’ corresponds to the ASCII character ‘K’. Let’s start at the very first compare and convert the hexadecimal values to ASCII and see what the final result is.

I’ve compiled a table below omitting the ‘0x’ portion of hexadecimal to save space:

[45, 4b, 4f, 7b, 74, 68, 69, 73, 5f, 69, 73, 5f, 61, 5f, 67, 69, 66, 74, 7d]

Now let’s use an online hexadecimal to ASCII conversion [website](http://www.rapidtables.com/convert/number/hex-to-ascii.htm) to convert the values. The output is below:

EKO{this\_is\_a\_gift}

Congratulations, you’ve successfully reversed your first binary.