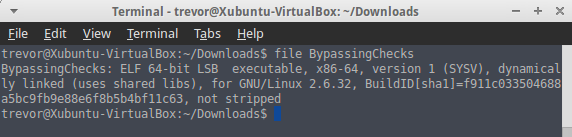
**Intro To Reverse - BypasingChecks Solution**

**Introduction:** BypassingChecks is a medium level reversing challenge designed to improve students reverse engineering skills by learning to bypass simple checks found in binaries. Learning to bypass simple checks is useful because you can then control the execution flow of a program without actually meeting those requirements. This writeup assumes you have basic knowledge with the debugger and disassembler of your choice.

**Task:** Using the debugger and disassembler of your choice, bypass the simple check in the binary to retrieve the flag.

**Solving:** This writeup will be solved using GDB Debugger and Hopper disassembler, but you’re free to use whatever tools you’re comfortable with. Let’s begin by performing a basic analysis against the binary using the ‘file’ and ‘strings’ command.



There are two pieces of information that are important at the moment. First, this file is a 64-bit elf binary. This means that file is executable, and it was compiled on a linux system. Second, this binary is ‘not stripped’ meaning this binary was compiled with debugging info enabled “gcc source.cpp -g”. Compiling binaries with debugging info enabled allows you to read high level code inside of the debugger / disassembler. The alternate is a ‘stripped’ binary that wasn’t compiled with debugging information. This means we would only be able to read assembly / machine code in the debugger / disassembler.

Next let’s take a look at the output from the strings command.

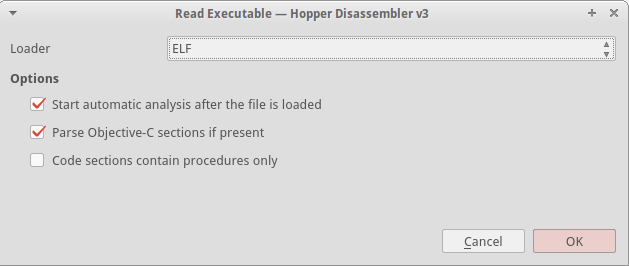


The above screenshot is only a partial output. The majority of the output produced by strings is rather uninteresting but I’ve gone ahead and highlighted what could be an important piece of information later. Notice that there’s two strings somewhere in this binary named ‘check\_key’ and ‘flag\_data’. It’s possible those two strings could be of use later when you begin reversing the binary.

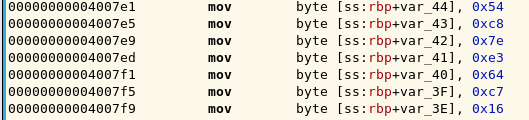
Finally, finish the basic analysis by executing the binary and seeing what this program does. First you’ll need to chmod the binary so you can execute it on your system ‘chmod 755 binaryName’. Upon execution the program asks for us to input a key. I tried ‘12345’ the program then output “Wrong” and quit. Trying several other inputs resulted in the same behavior. It becomes pretty clear you need to figure out the magic key, or bypass the check that compares your key with the correct key.

Screenshot - 072516 - 23:42:23.png

Go ahead and open up your disassembler of choice and let’s get to reversing this binary. The default options for disassembling the binary should work without any problems.

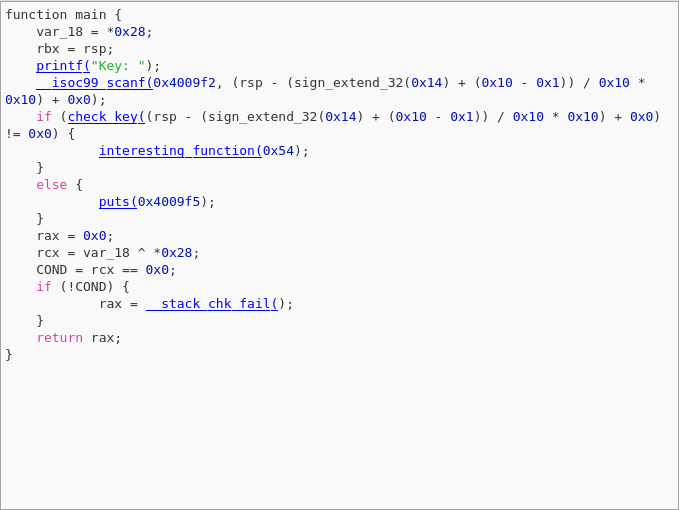


Let’s start by taking a look at main() to try and figure out what this executable is doing. I’ll also highlight a very useful feature of Hopper, although I believe IDA-Pro and Radare2 have similar functionalities. Below is a snippet from main()



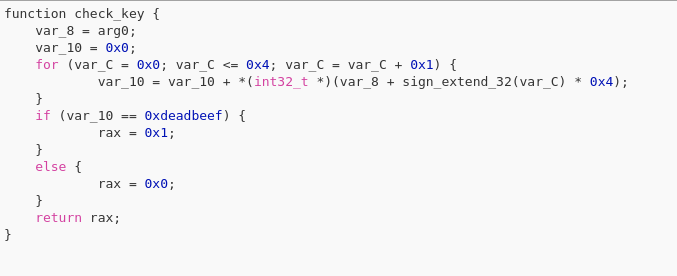
It seems as though one of the first things main() performs when the program is executed is to move a large chunk of hexadecimal characters onto the stack. This could quite possibly be the program generating the flag for us. Use a Hex to ASCII converter to find out for yourself!

All of this assembly gets a tad tedious to deal with though doesn’t it? Let’s have Hopper do some heavy lifting for us and generate some high level pseudo code for the main() function.



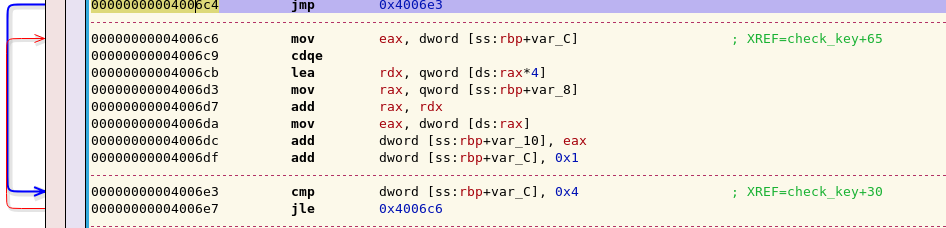
That’s much better! Using the pseudo code generated by Hopper it’s much easier to get a better understanding of a program's functionality. In this case the main() function executes a printf() statement that outputs “Key:” to the screen, then takes our user input via a call to scanf(). Once user input is read a conditional if() calls check\_key() to validate our input against the correct key. If check\_key() returns anything other than 0 interesting\_function() will be executed. Otherwise a simple puts() function is executed. I’m going to go out on a limb and say that puts() function is responsible for printing “Wrong!” to the screen.

Since we now know there’s a function responsible for validating our input, let’s take a look at the function and see if we can’t figure out how our input is validated. Below is pseudo code generated by Hopper for the check\_key() function.

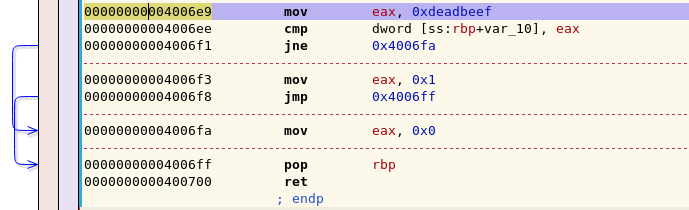


Inside of the check\_key() function two variables are created, var\_8 and var\_10. Then the program enters a for() loop which appears to take the data from var\_8 and place it into var\_10. Next the program enters a conditional if() statement which compares the data in var\_10 to 0xdeadbeef. If this comparison returns true the value of the $rax register is set to 1, otherwise the $rax register is set to 0. Then the function returns the value of the $rax register. For instructional purposes I’ve segmented the assembly instructions below.

For Loop:



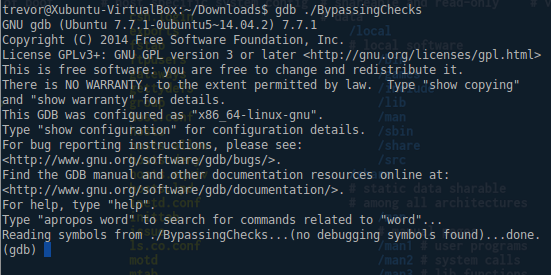
If Statement:



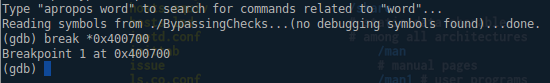
It should be clear now that your input needs to evaluate true against 0xdeadbeef to get the proper return value from the check\_key() function. However, 0xdeadbeef is a [magic number in hexspeak](https://en.wikipedia.org/wiki/Hexspeak) for an uninitialized segment of memory. This means there isn’t a valid key to compare your input against, so no matter what you enter the program will never give us the flag.

But there’s still hope! Hypothetically if you did input the correct key the value of $rax register would be set to a value of 1 to indicate the correct key was input by the user. Why don’t you use GDB to set a breakpoint at the address 0x400700 right before the function returns and set the value of $rax to 1 manually? Try it out.

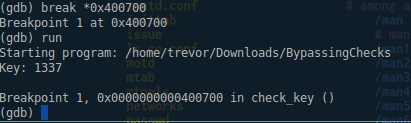
Open the binary in GDB by typing ‘gdb ./binarytoexecute’ in the command line.



Next you want to set a breakpoint at address 0x400700 to pause the execution of the program right before the check\_key() function returns. This is done so you can set the value of $rax to 1 before the function returns. Setting a breakpoint is as easy as running ‘break \*0xADDRESS’ inside of GDB



Then you want to run the program inside of gdb so the breakpoint we just set is hit. Type ‘run’ inside of gdb to execute a program. The program will execute as it normally would outside of gdb until we hit our breakpoint.



Finally before you continue the program's execution you want to set the value stored in the $rax register to 1. Faking the program into thinking you entered a valid key. You can manually set the value stored inside a register with the command ‘set $register = value’. Additionally, you can inspect the current value stored in a register with the command ‘i r register’ where ‘i’ stands for inspect, and ‘r’ for register.

VirtualBox_Xubuntu_31_07_2016_13_49_09.png

After you’ve set the value contained in the $rax register to 1 you’re free to continue the programs execution by typing ‘c’ or ‘continue’.

VirtualBox_Xubuntu_31_07_2016_14_02_10.png

Flag: flag\_is\_you\_know\_cracking!!!

Congratulations, you now know how to bypass simple checks found inside of binaries to control the execution flow inside of a program.