Daniel Oliveros

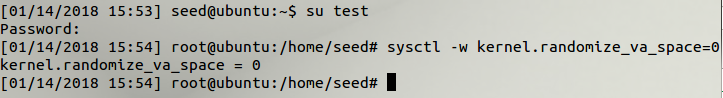
Garrett Bogart

Spring 2018 – Independent Study

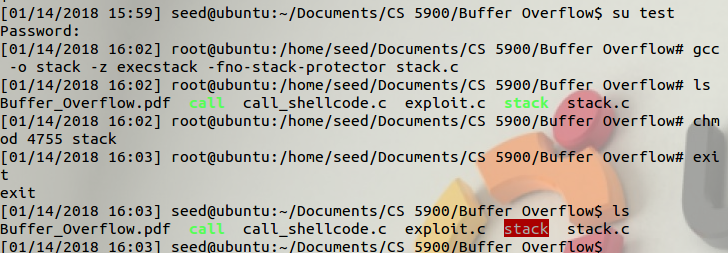
Buffer Overflow Vulnerability Lab

**Task 1**

Ubuntu comes with multiple features to prevent buffer overflow attacks. The first step of this lab is to disable the randomized memory of the stack. To do so you must be long in as a root user.



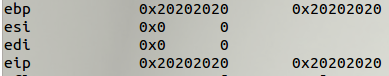
Then when you compile the stack program, the vulnerable program, you have disable the stack protector and the non-executable stack. Compiled using gcc -o stack -z execstack -fno-stack-protector stack.c.



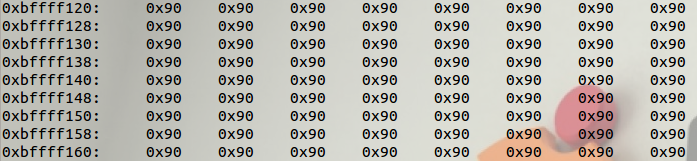
After the safety features have been disabled the stack executable should be highlighted in red.

The buffer overflow made it so we would overwrite the eip and ebp. We want to use the eip for our own sake in order to execute the shell code and open a terminal. In order to find where our eip was being stored we needed to go around our buffer and change values throughout it. This would allow us to see exactly where the eip was being stored.

Eventually, we were able to see this

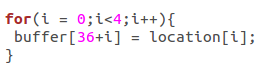


These were the garbage values we put in our buffer. By knowing they were now overwriting the eip we could pinpoint the locations in the stack we were looking to target. Once we found this location we looked for an address in the stack we could jump to in order to jump into our NOP sled.

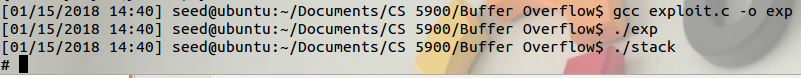


Any address in this block will take us into our NOP sled, which will in turn execute the shell code to open a terminal. Setting the eip to one of these addresses should help us exploit this vulnerability. When adding these values to the stack, we must do it by reversing the order of each byte. These lines of code take care of that:



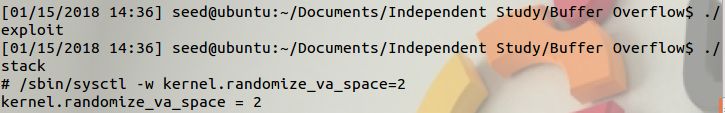


The result is that, once the program has been compiled and run, we get a root shell:



**Task 2**

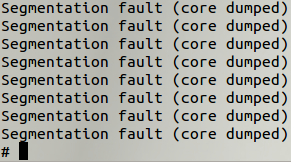
This part is asking us to enable the randomized stack location. Using our newly acquired root access, or logging in as root we can enable this safety feature.

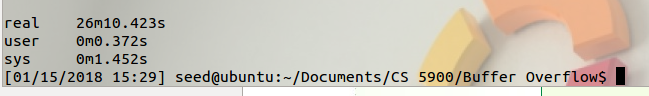
****

Since this vastly decreases our chances of success it is recommended to run the stack executable in a loop from the terminal.

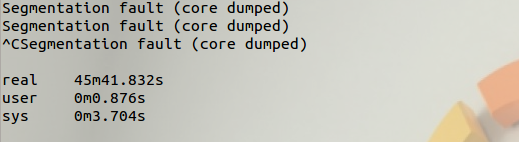


If you are lucky





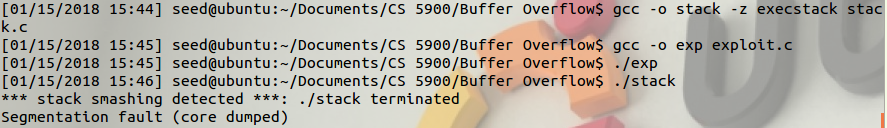
If you are unlucky



At which point you might want to give up

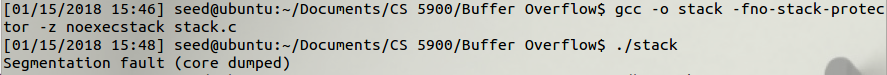
**Task 3**

Stack guards generally take the form of a canary value. These values is an unknown byte that the computer keeps track of. If a canary value is overwritten by something then the computer can detect that stack smashing has occurred. If the canary value is known it is possible to overwrite it and continue on with an attack.

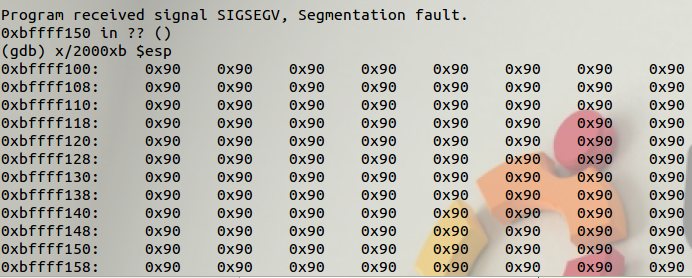


**Task 4:**

When running the program after compiling it with the noexecstack flag we get this error.



This error happens because despite the eip redirecting the program back to our NOP sled, the computer will not run any shell code it finds in the buffer. This makes it impossible for a program following our set up to exploit the buffer overflow vulnerability we were looking to take advantage of.



**Issues:**

1. For unknown reasons, when picking an address for the eip that was around the start of our NOP sled we would get segmentation faults when running the code outside of the debugger. This was very inconsistent and didn’t seem to make much sense. To get around it, simply pick a location further down the NOP sled.
2. Task 2 can take far too long. Probably good to inform people beforehand, as some may not be expecting this.

**Ideas for Improvement:**

1. Rearrange how tasks are organized, task 2 would be best at the end so students can complete more of the assignment first, and then put it aside while the last task runs. This could also lower the chance they’ll accidentally forget to turn off address randomization.