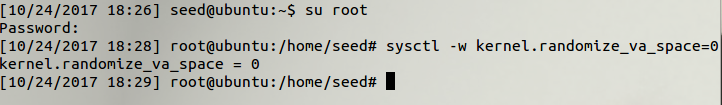
Daniel Oliveros – A02093272

CS 5460

**Assignment 5 – Buffer Overflow**

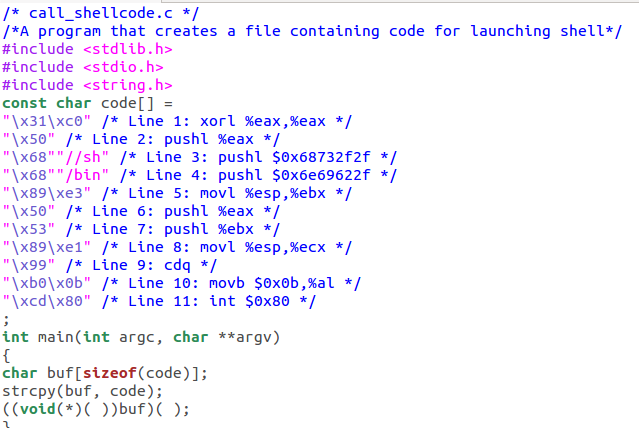
**2.1**

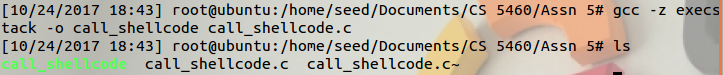
I started by disabling the features the lab told me to:



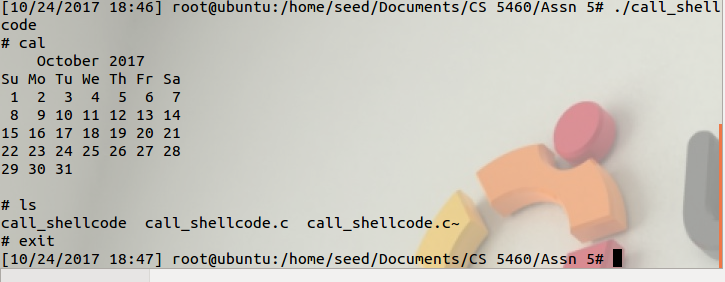
**2.2**

I copied the code onto my own call\_shellcode.c file, shown below, compiled and ran it. As expected, it gave me the most powerful thing I could want, A shell.



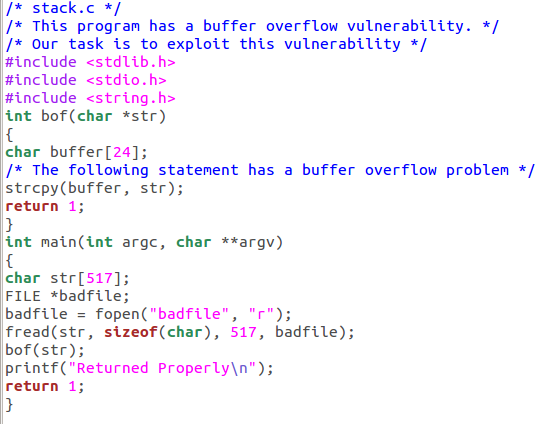


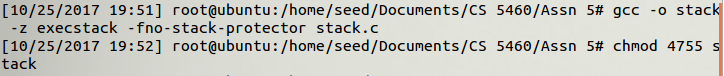
I ran some commands with the shell, and sure enough they worked.



**2.3**

I copied and compiled the vulnerable program, also making sure to set chmod to 4755. The purpose of the chmod change is to allow us to do all sorts of random fun things with our code. It sets the user ID to root with the 4 in the front, it also gives read, write, and execute permissions to the file owner and the group, as well as giving read and execute permissions to any other users.

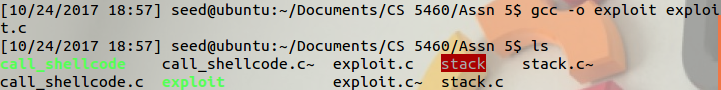




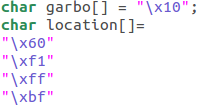
**2.4**

For this task, my method was to create a single char[] named garbo and paste it all over my buffer in different locations. I made it mode all throughout until I found the general area where my ebp and eip should be. I also filled up my buffer with nop slips and put the shell script at the end of the buffer.

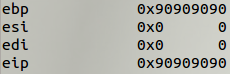
I compiled the exploit like this:



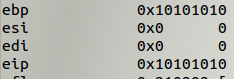
Using these variables to mess around and find my important addresses in memory:



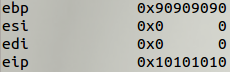
And this is how my results started to look like. I started by noticing both my ebp and eip were overwritten by my buffer, so I figured I’d just have to rewrite that buffer with something I could recognize when checking for the location of things.



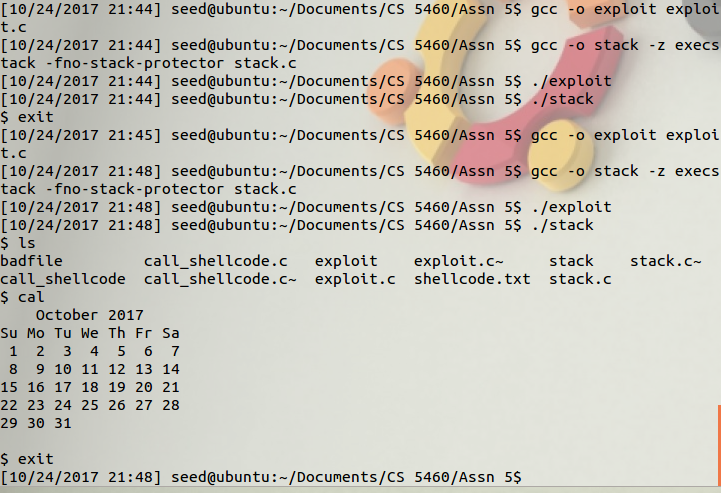
I eventually reached this point, where I’d managed to overwrite both the ebp and eip, guessing by the way they’re laid out in memory I figured I would have to move around the general area I was at to find exactly where I wanted to be.



And eventually I found it. The location where my eip was being overwritten at the exact point I needed to insert a location into. I overwrote my eip with a location just after it, in the nop slip space, and ran the command. I determined that with the way my code was laid out and such my eip would always be located around 12 bytes higher than the size of my buffer, which was rather helpful for my calculations.



I then proceeded to compile and run my files one last time, and the results were exactly what I was hoping for. A terminal opened up, and from there I was able to run some random shell commands to show I’d made it in.

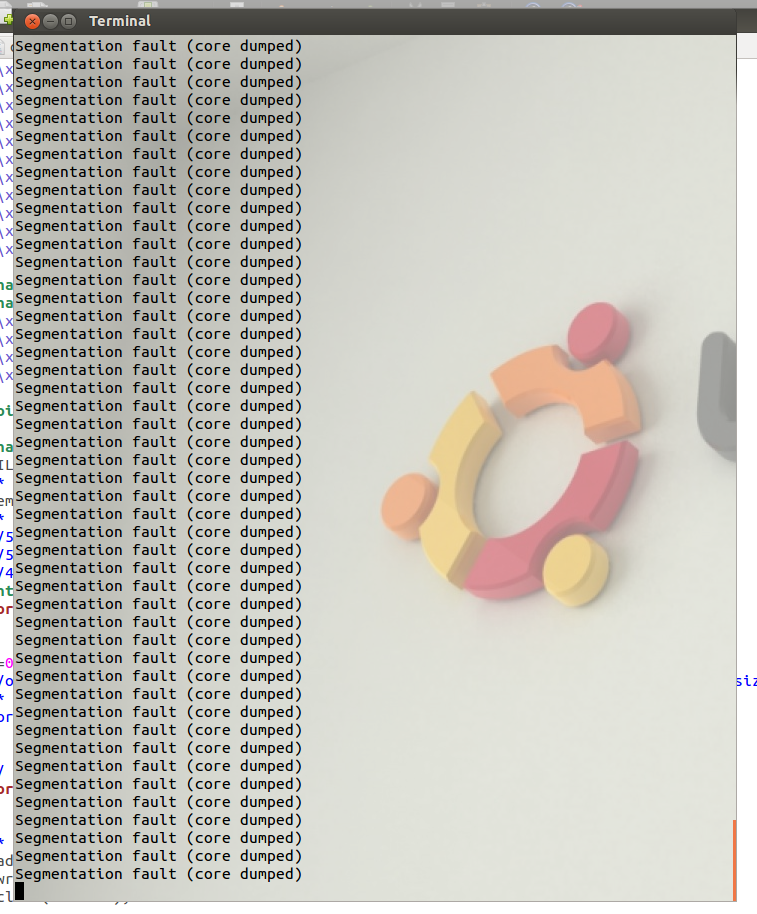


I understand how this works, and I think it was extremely fun to run this myself. Having a nicely sized buffer also allowed for simple inspection to be feasible and made plenty of my work much easier.

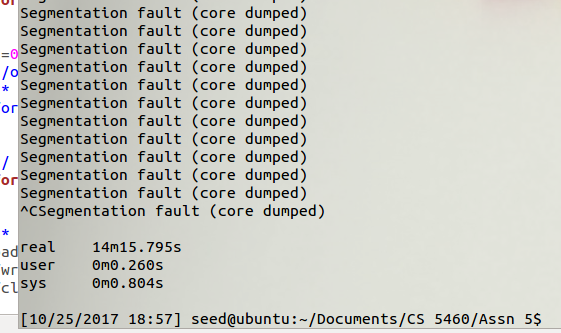
**2.5**

I set kernel.randomize\_va\_space back to 2, and proceeded to run my code in a loop. It never ended up finding the location it wanted, which sucked, but I know it is plausible for it to do it. A friend of mine got his to do it after 3 minutes through sheer luck. Mine ran for 14 minutes until I decided it was time to move on. So, I was unlucky, but I think it showcases how much more difficult it is to get this type of exploit to work when you’re aiming at a static location and the stack is being moved around to random places every time the code is ran.



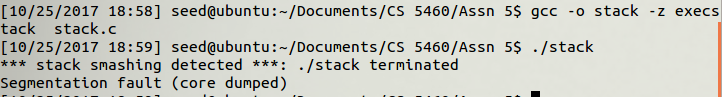


I eventually gave up waiting for the command to run, so I decided to let it be and accepted my bad luck.



**2.6**

During non-altered runs, the program instantly recognized that I was trying to smash the stack and killed the process. I read up a bit on this, and I think it may be able to figure out that stack smashing is being performed by choosing a random byte that shouldn’t be overwritten, and then alerting the program when it is changed, making it stop right then and there, as the unintended behavior may be malicious.



**2.7**

By compiling my stack file with the non-executable stack command enabled I could stop the attack we were performing from happening, this is due to the shell script becoming unusable once it’s reached. In the end, I was unable to circumvent this protection since it legitimately kills any way of performing this attack similar to what I was trying to do.

