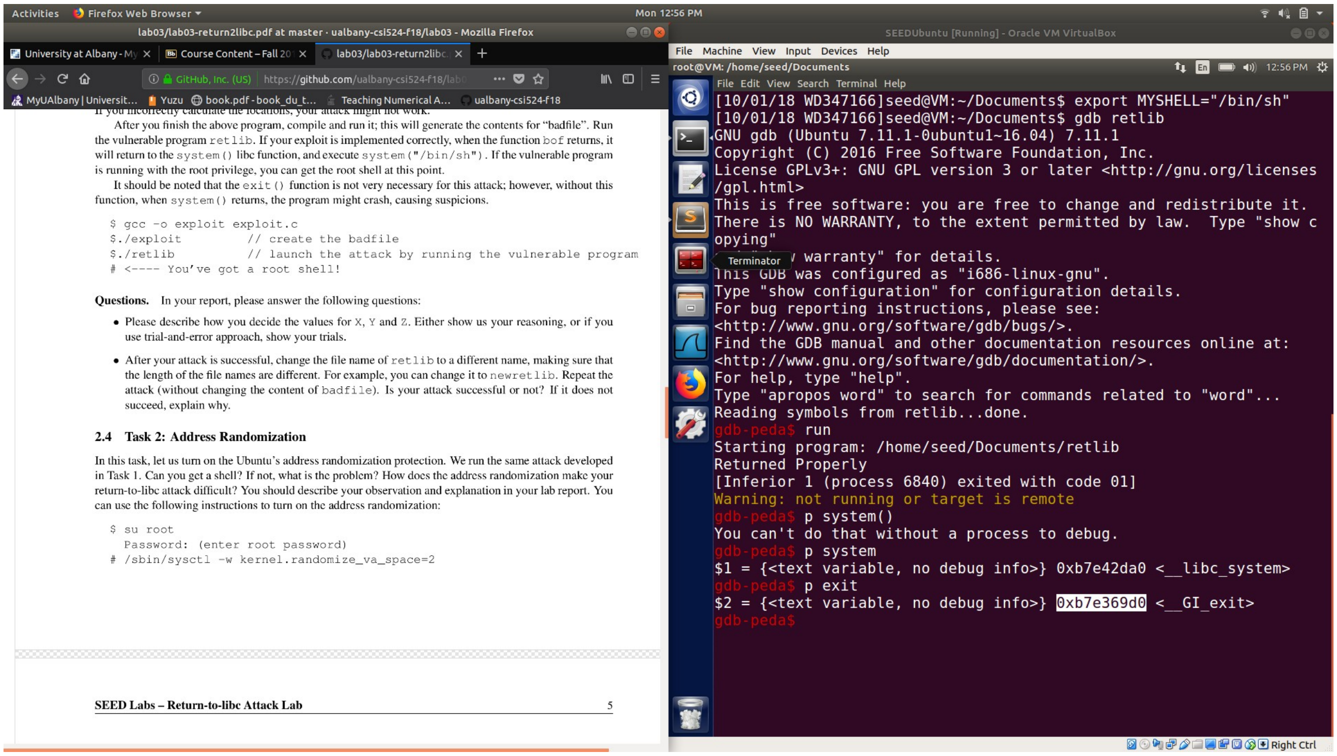
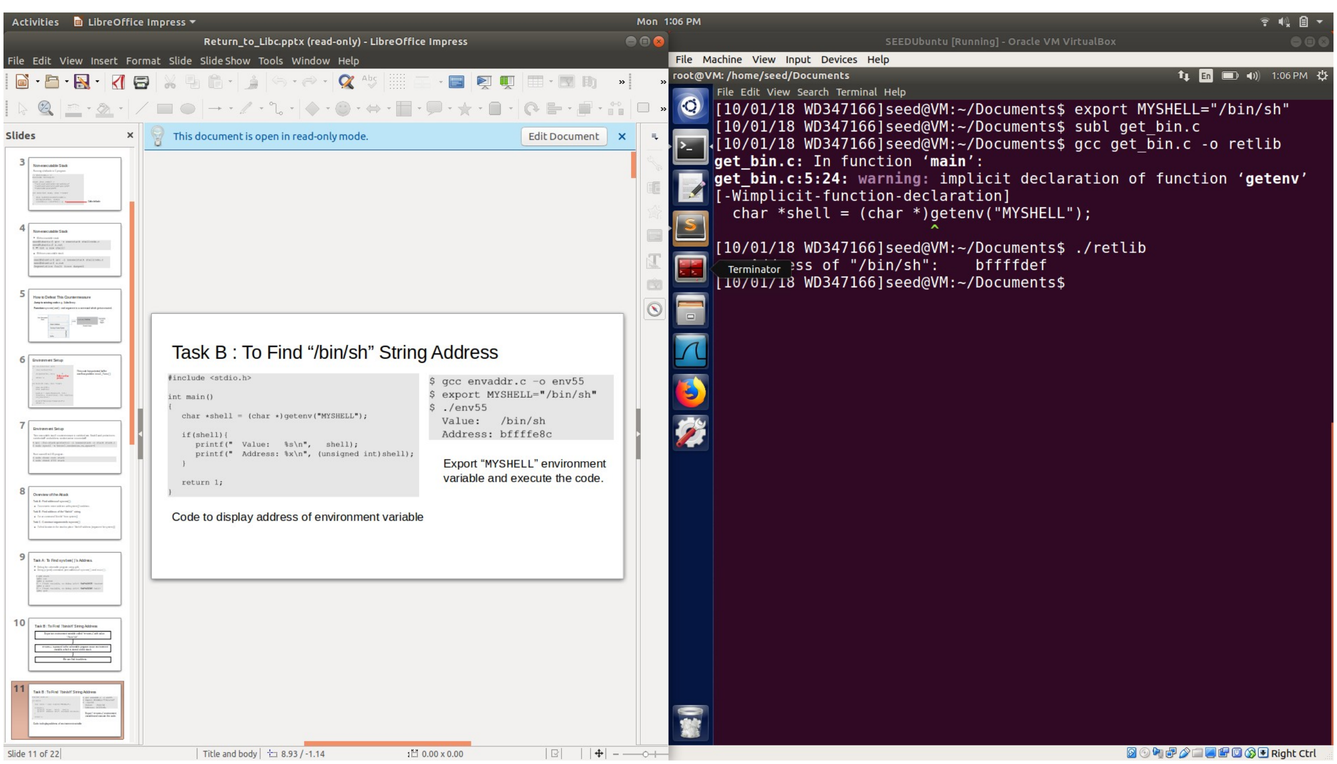
William Dahl
ICSI 424 Information Security
Lab03
October 10th, 2018

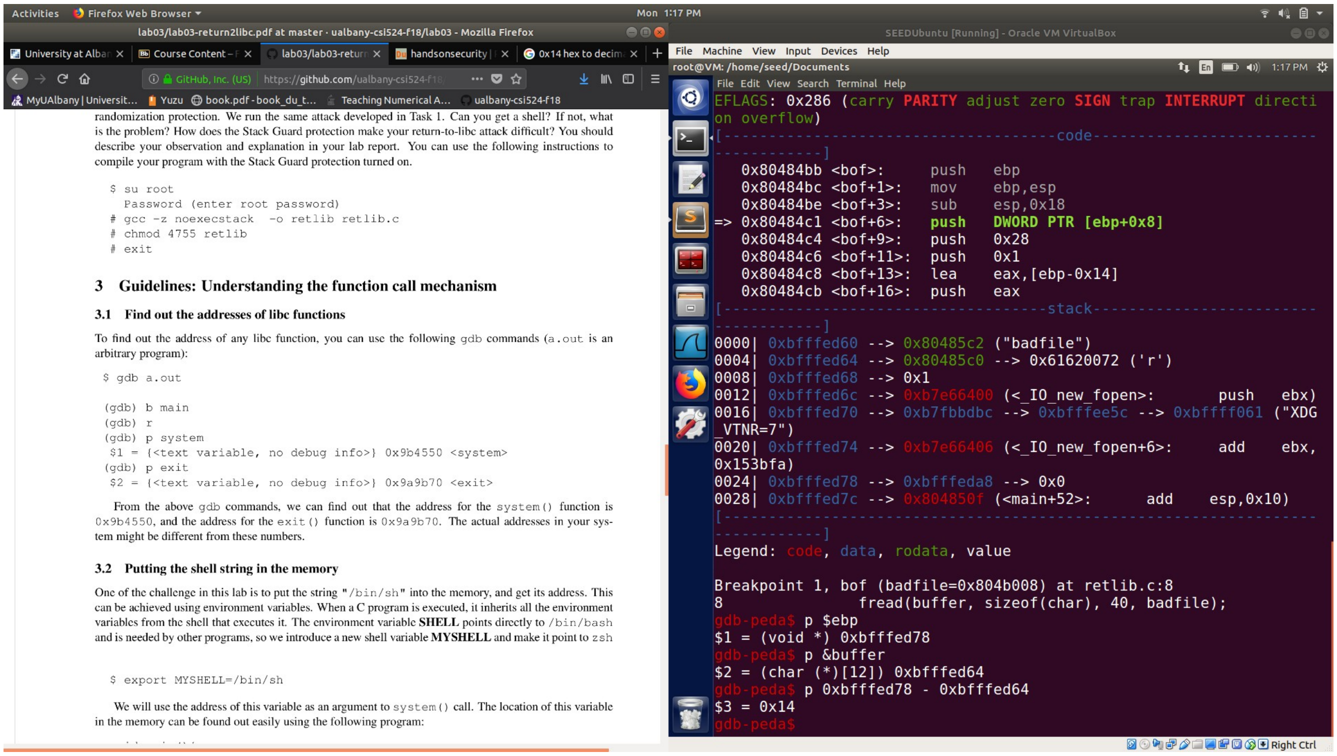
Task 1: In this screen shot I ran the retlib file through gdb and got the address of system() and exit()



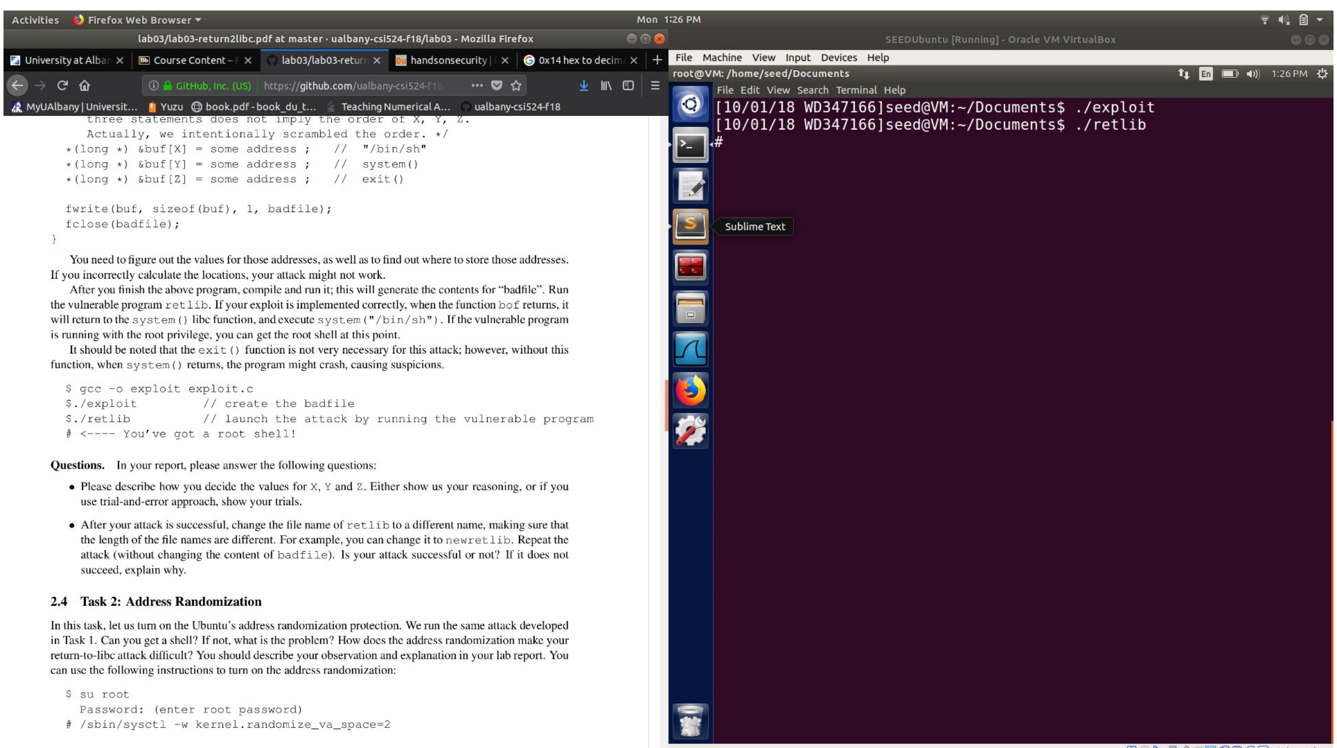
In this screen shot I exported the environment variable MYSHELL=”bin/sh”. Then I ran a program that is called the same thing as my vulnerable program that print out the address of the MYSHELL environment variable.



In this scree shot I ran the retlib through gdb again this time with a breakpoint at the vulnerable function bof and then I printed the address of $ebp and the beginning of the buffer and then got the address of $ebp within the buffer by subtracting the two.

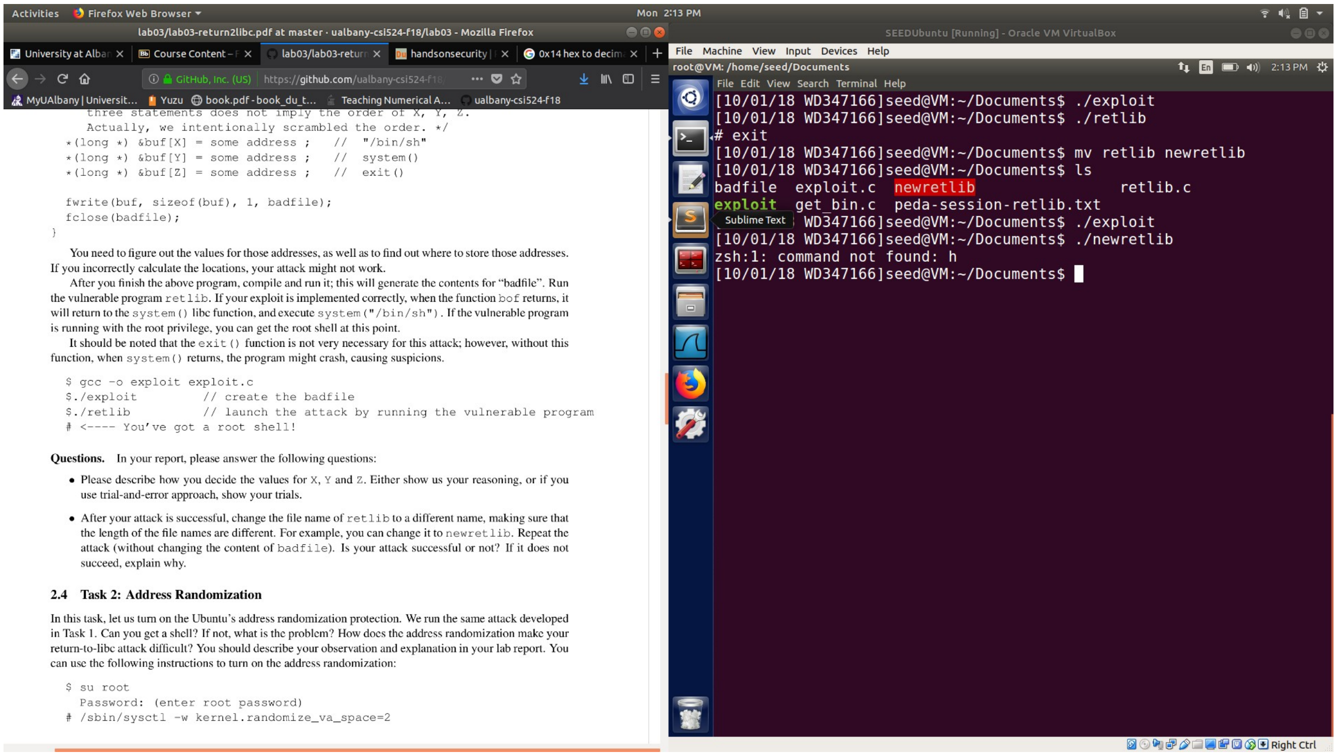


In this screen shot I run the exploit program and then the vulnerable program and end up with the root shell.



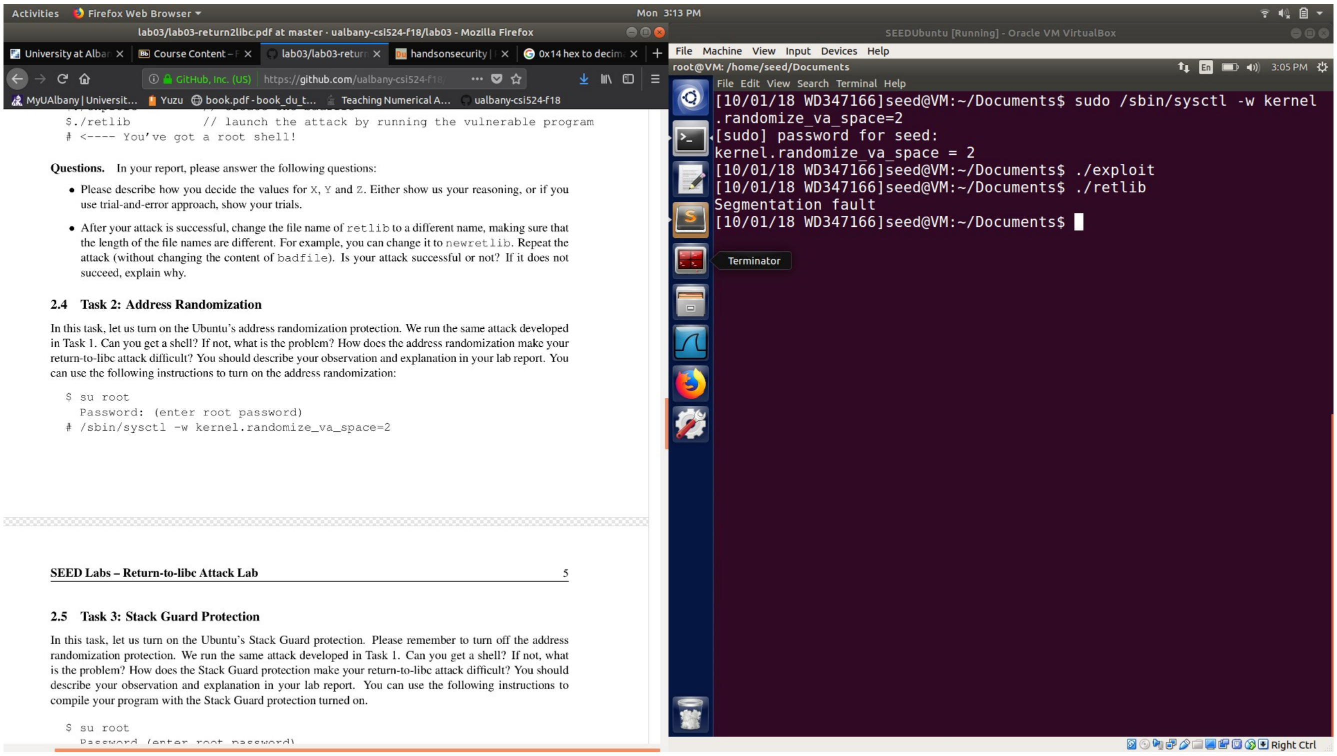
Questions:

- How I found the values of X, Y, Z is by getting the address of $ebp within the buffer which was 0x14, which is 20 in decimal. I then added 4 to the address of $ebp (24) to get the return address in the buffer and then over write it. I added 8 to the address of $ebp (28) to get address were exit() needed to be written to and then $ebp + 12 (32) to pass “bin/sh” as an argument to the system call.
- In the below screen shot I show the output after running the vulnerable program with a longer executable name.



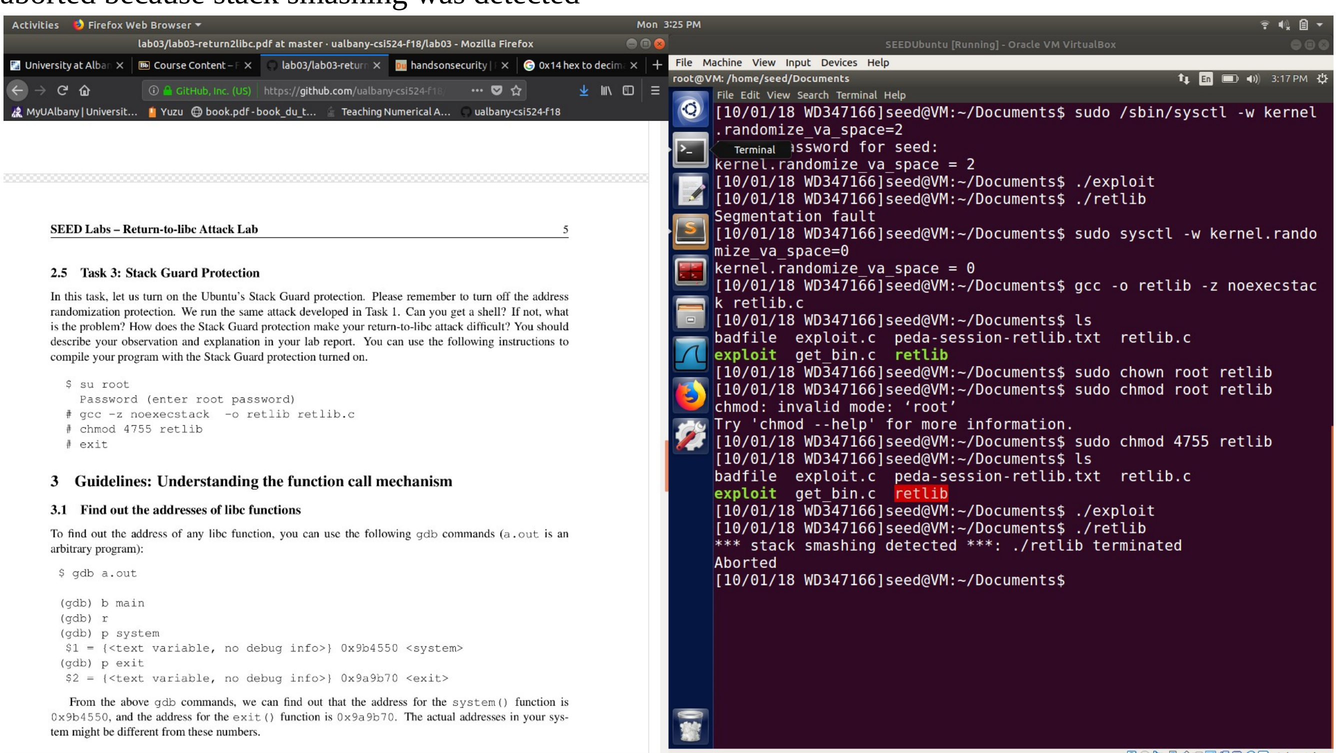
- The reason that the attack did not work is because when the name of the executable was changed then so was the address of the environment variable MYSHELL. The address the MYSHELL started at was moved down 3 bytes because the name of the executable was 3 bytes longer. So the h that we get is the h from “bin/sh”.

Task 2: In this scree shot I turn the randomized space to 2 and then run the attack again to which I get a segmentation fault



This is because when the randomized space is set to 2, the starting address for both the stack and the heap are random so the address that I put into the malicious code for the location of system and exit are wrong causing the program to go to an area of memory that does not exists within the stack.

Task 3: In this screen shot I recompiled the retlib.c code with the stack protector enabled and then changed it to a root owned SET_UID program and then ran it, which resulted in the program being aborted because stack smashing was detected



This is because the stack protector protects data from being purposefully overwritten in the stack through a buffer over flow in order to stop attacks and security breaches.