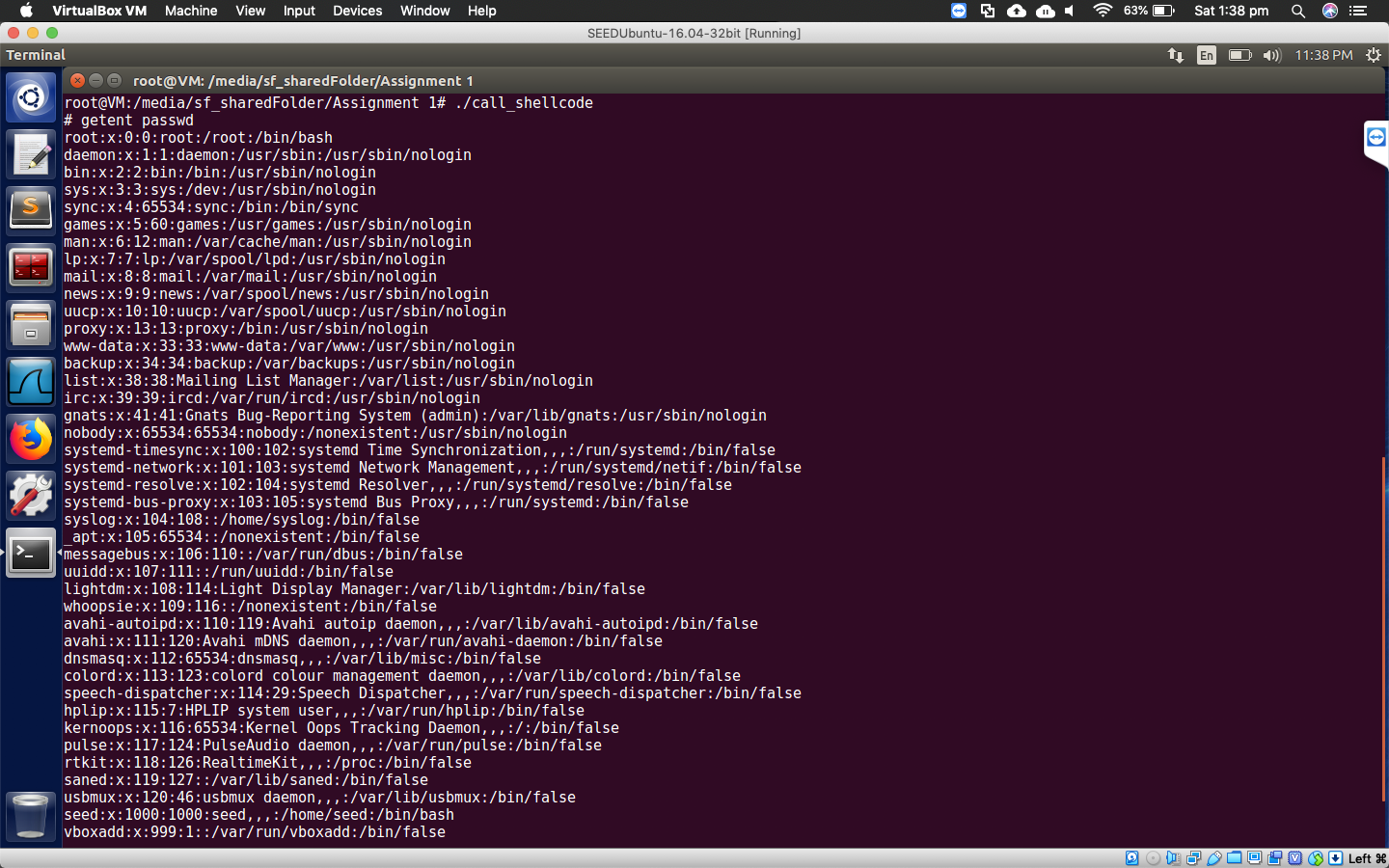
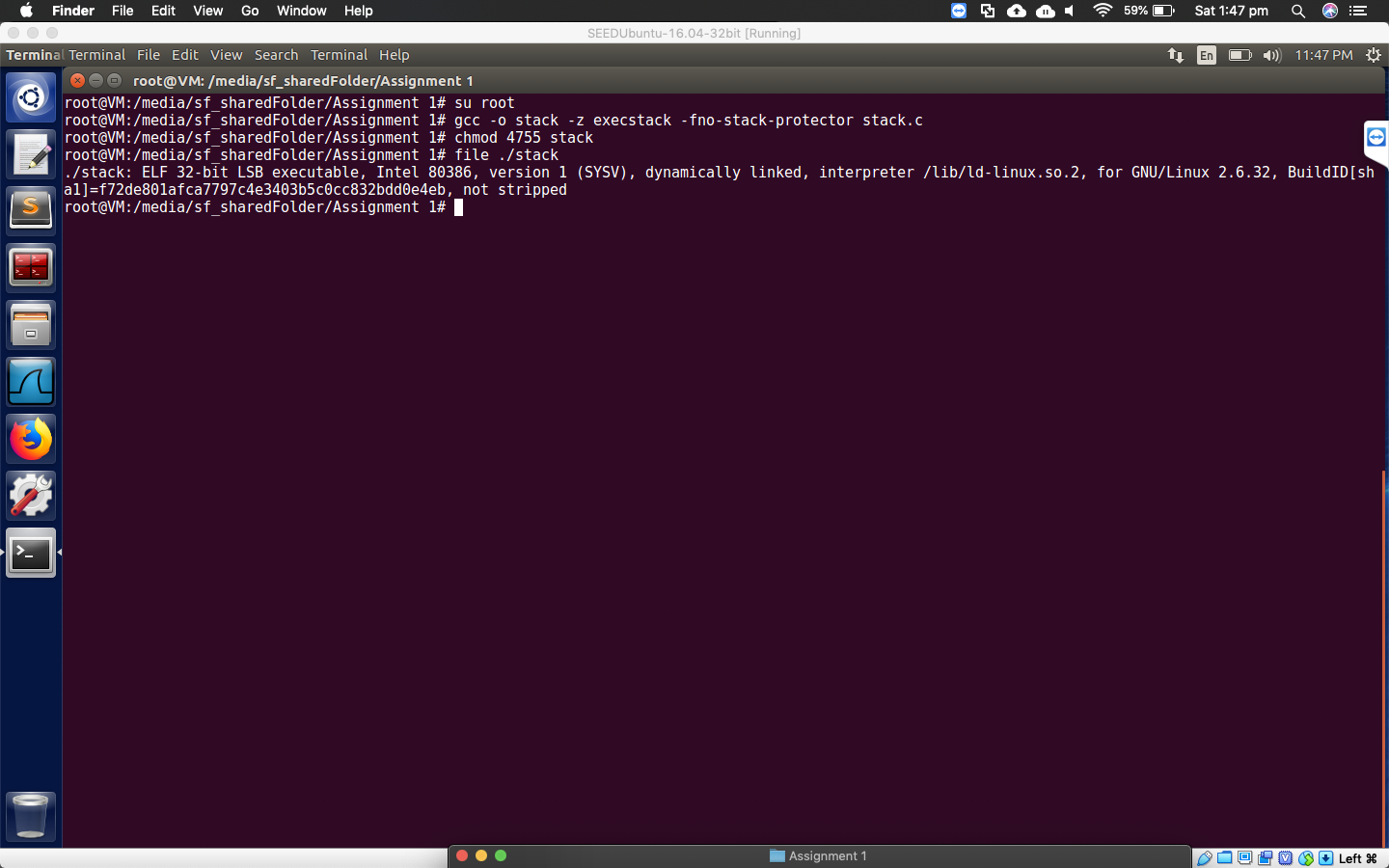
# FIT3173 – Assignment 1 – James Schubach - 29743338

# 3.2 - Task 1 - Question 1

So, in this screenshot we see by calling the shellcode file we get access to the bash line, from here I am able to use any commands I would like. In this example I use “getent passwd”. This works by exploiting strcpy, this allows us to inject our shellcode and run a bash line.

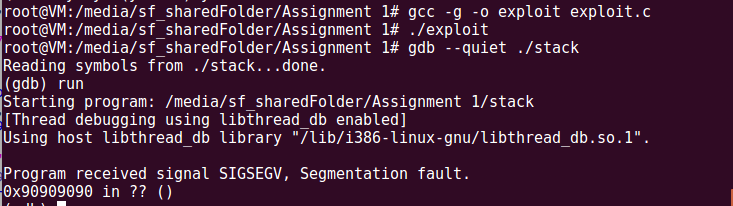


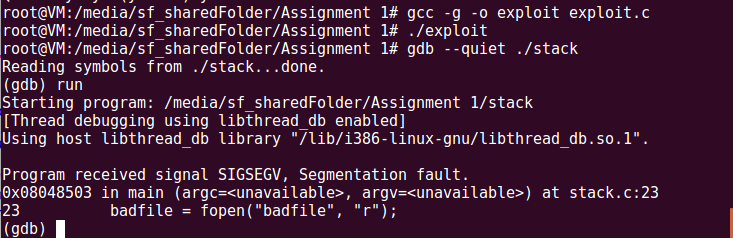
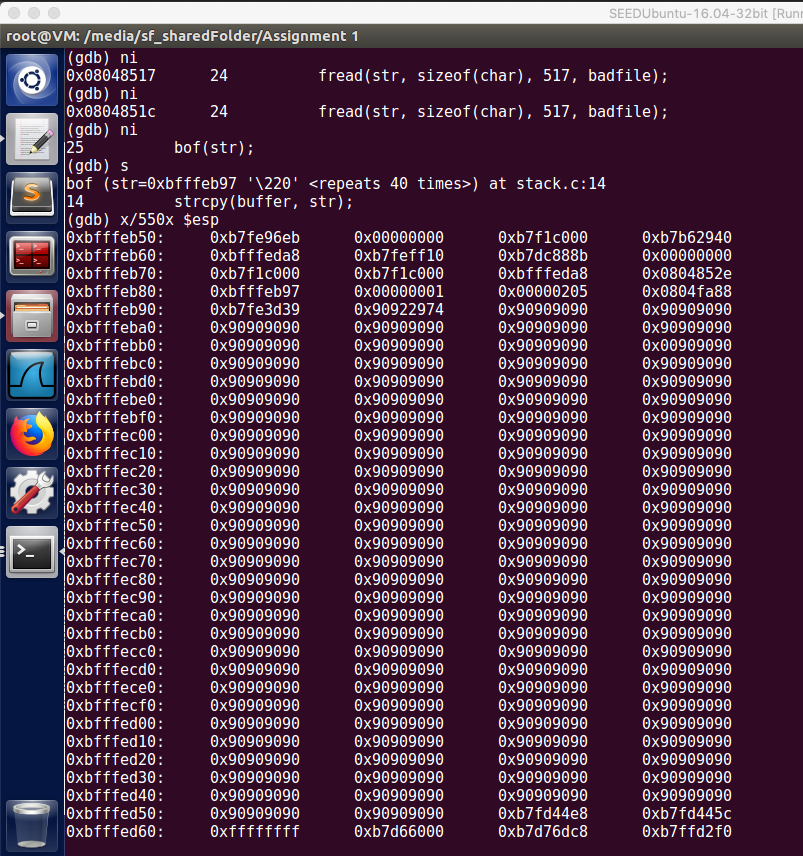
# 3.3

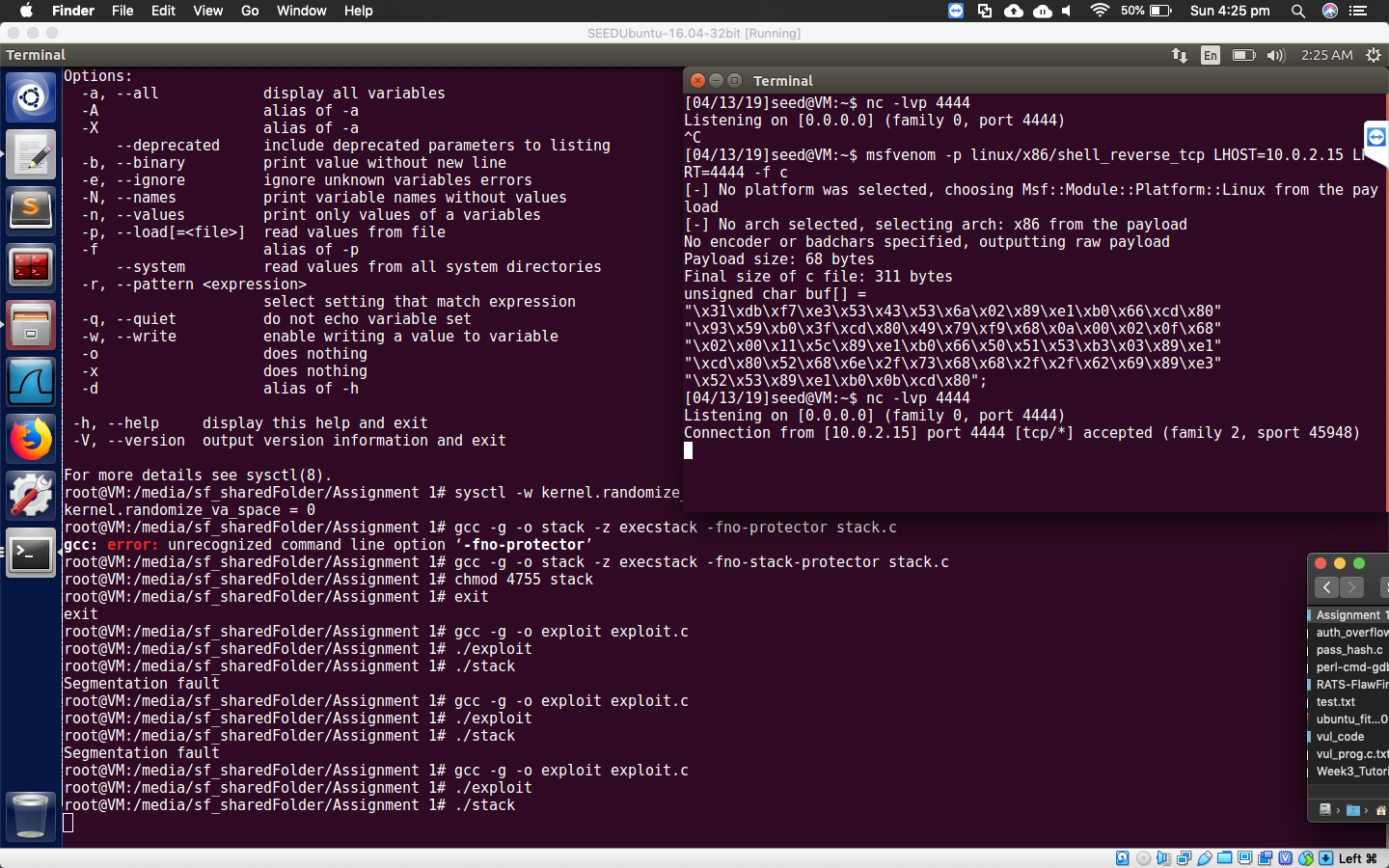


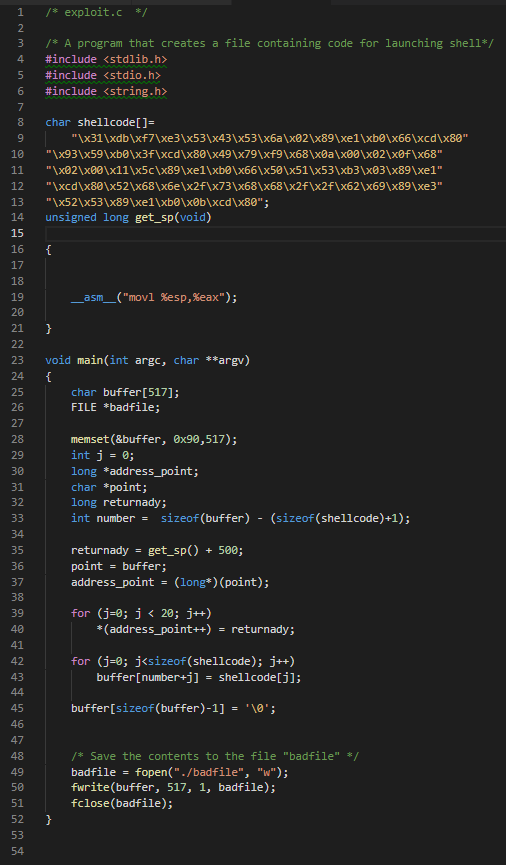
# 3.4 – Task 2 – Question 2

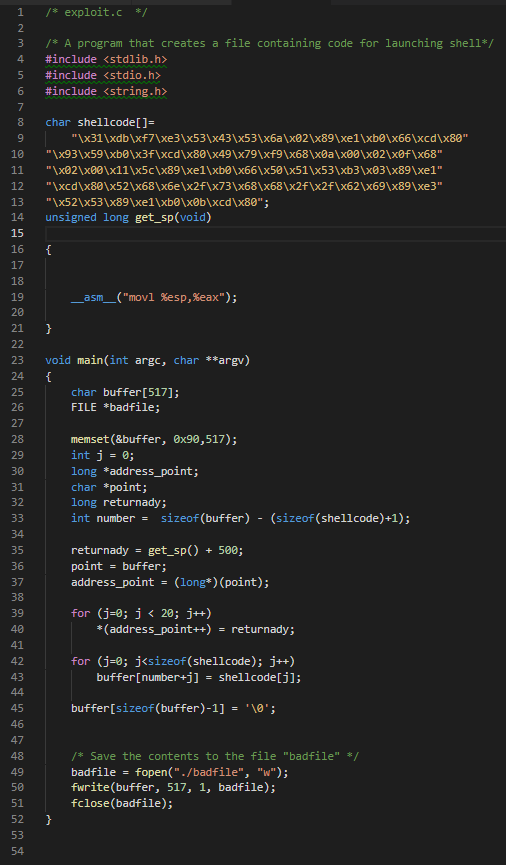
My code works by creating a buffer and filling it with NOP instructions which will help create our NOP slope. From here I calculate the address and return addresses, after this we fill the buffer with our shellcode one by one, and finally make the last bit of the buffer a /0. Our code then creates the badfile and writes the contents of the buffer to it. From here the stack code once again uses strcpy which we are taking advantage off as this doesn’t check for buffer overflows. Now we have our shellcode injected into the stack, from here the NOP sled will come into effect and slide into our shellcode. You can see that once it hits the shell code we get the active connection in the other terminal. We find the address very easily as we know that the return address is located at “movl %esp, %eax” + 500. This makes our job easier as we don’t need to analyse the stack each time and can programmatically find it.











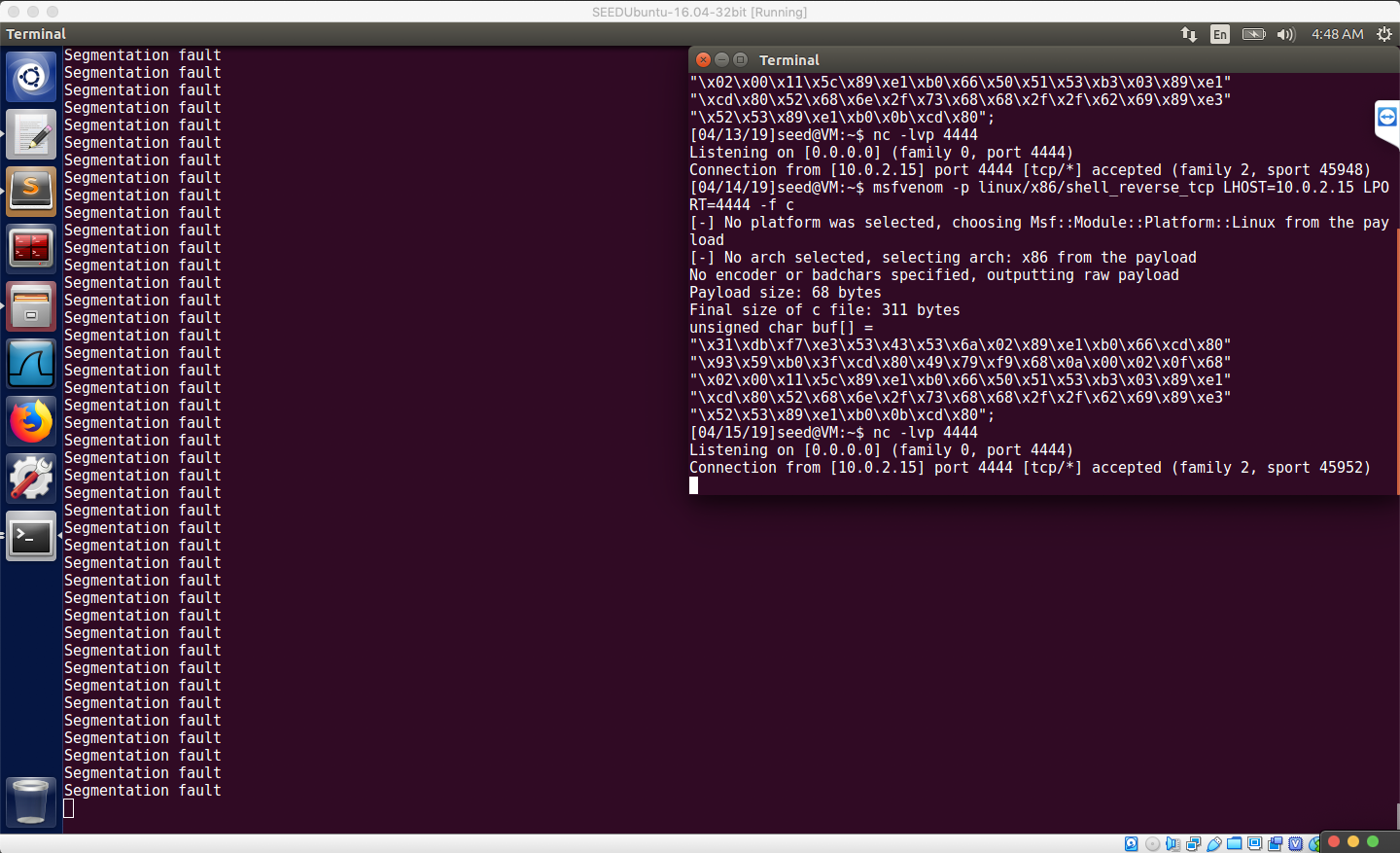
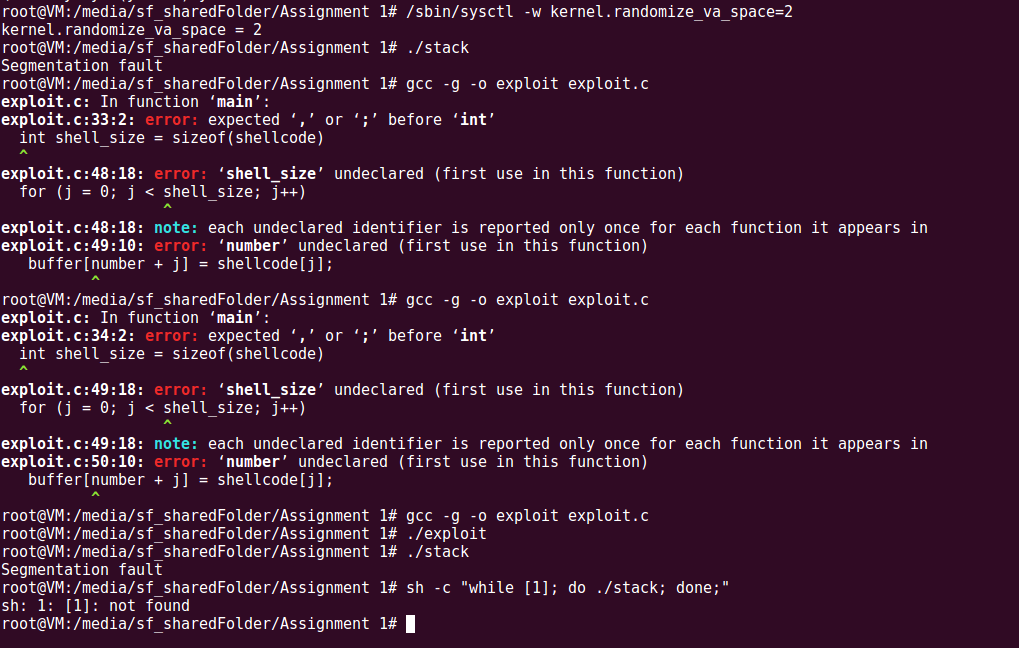
# https://i.imgur.com/9DBIKby.png

# https://i.imgur.com/HJF5B5s.png

# https://i.imgur.com/DlzHaFH.pnghttps://i.imgur.com/HJF5B5s.png

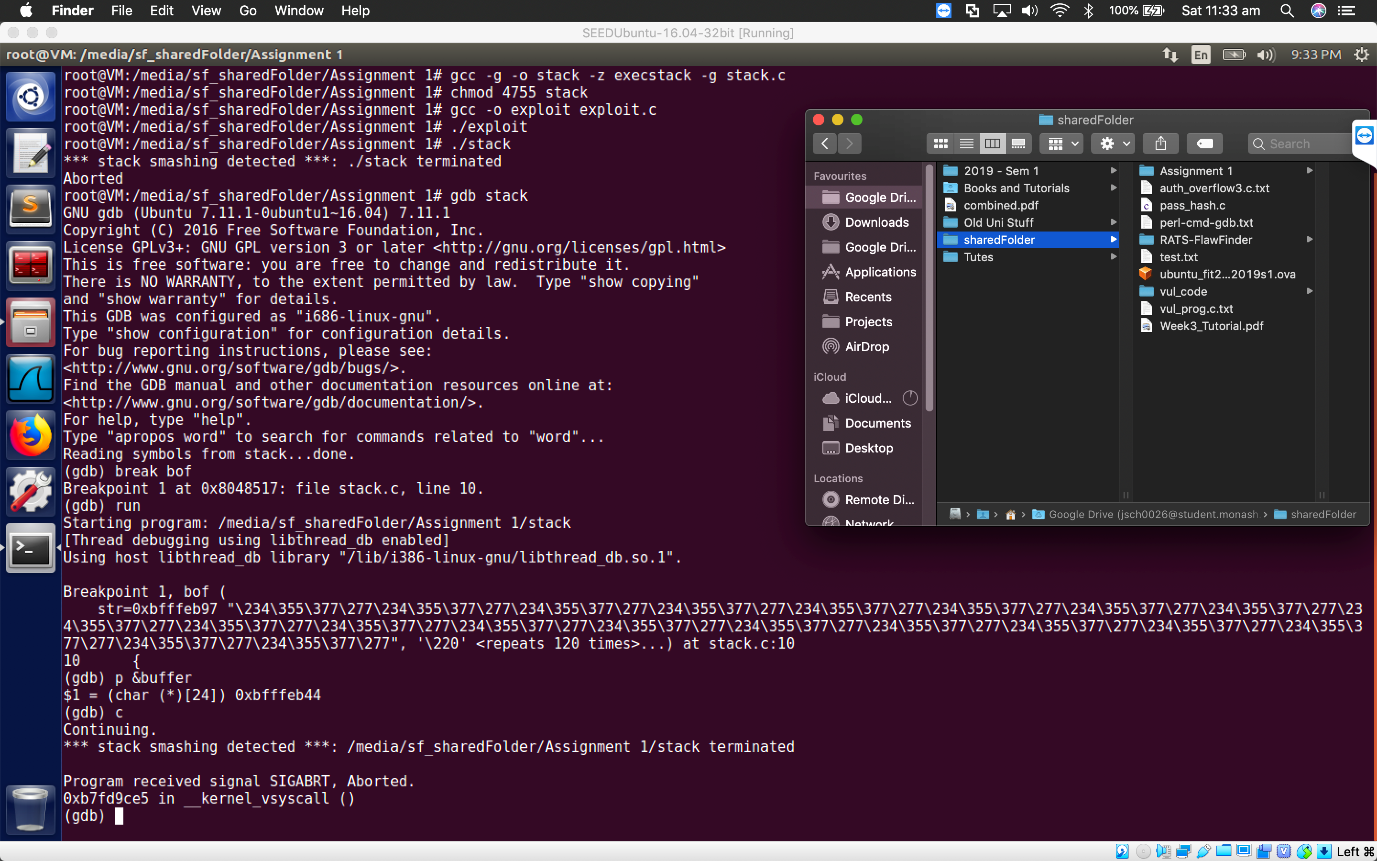
# 3.5 - Task 3 – Question 3

We can find the shell, but not straight away. This is because the stack address is being randomize each time it runs. So even by knowing the return address it’s not going to work. This is due the locations of where executables are loaded into memory. However, if you decide to loop the attack it will eventually work, however this is entirely due to probability. As eventually the code will get the right return address and be able to implement our shellcode.



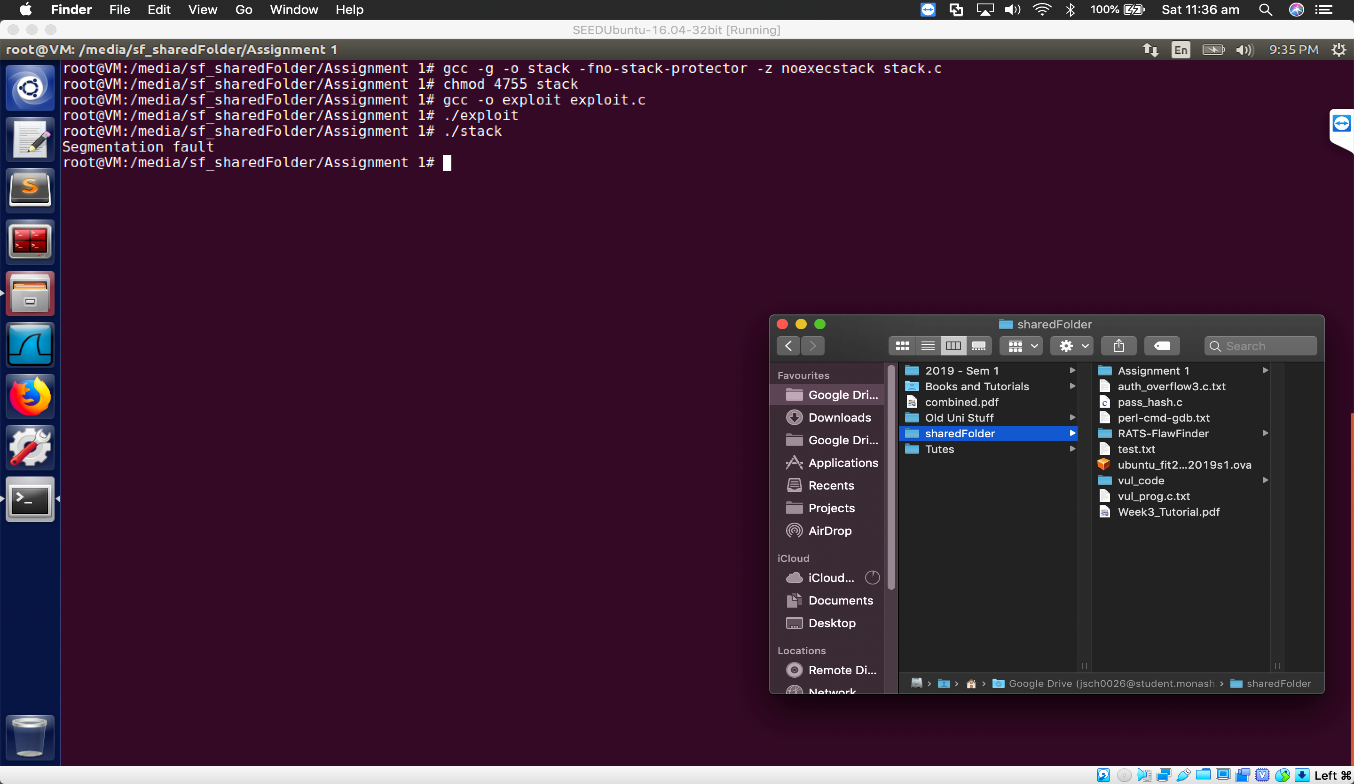
# 3.6 – Task 4 – Question 4

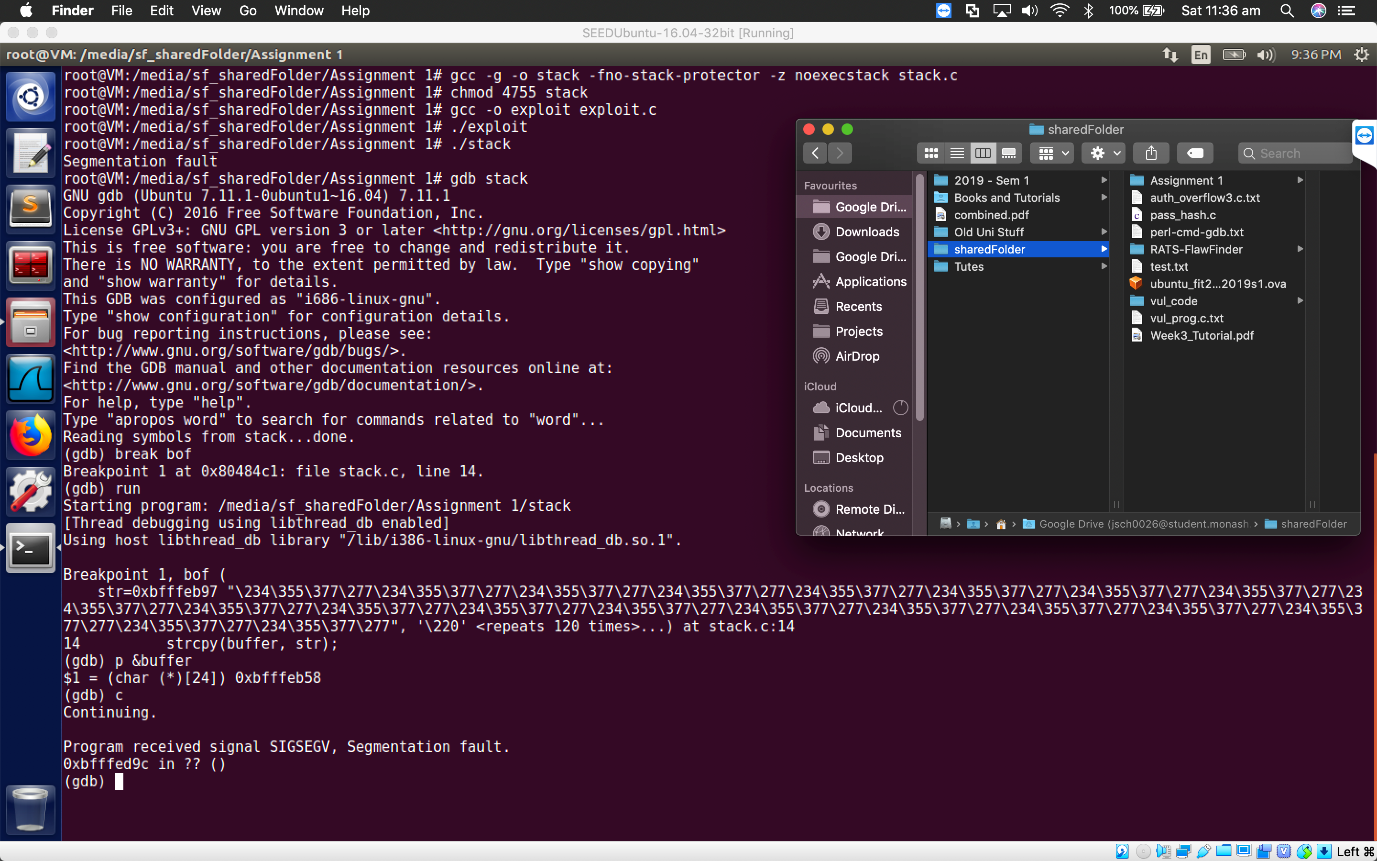
Stack guard is a tool on the OS that can detect buffer overflow vulnerabilities, it does this by analysing the stack and comparing it and the start and end. This value that is inserted between stack variables is called a canary, thus the canary will be overridden if buffer overflow occurs. At the end of the function, the canary is checked and if it’s different then we know that a buffer overflow has occurred. It then aborts and quits the program.



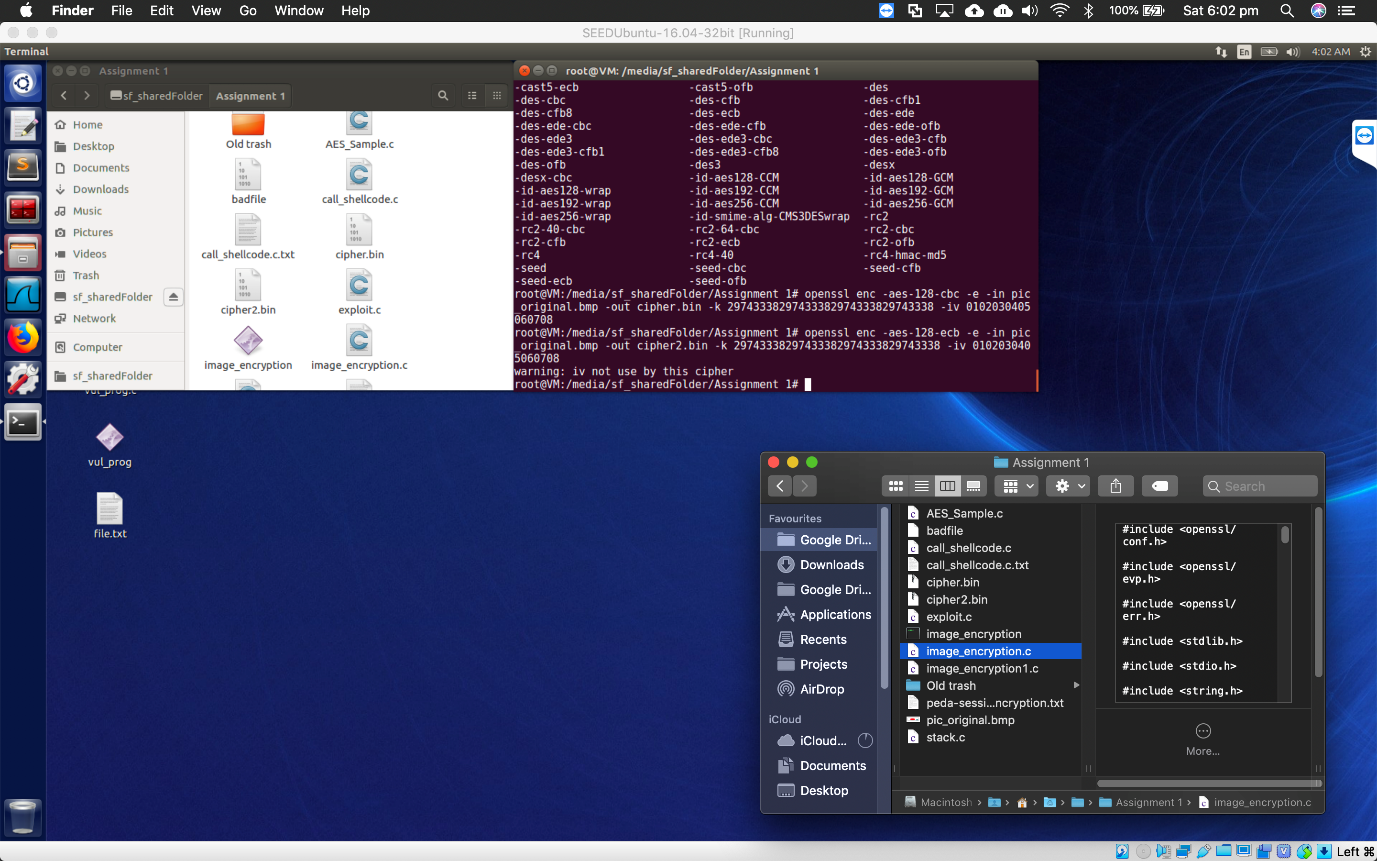
# 3.7 – Task 5 – Question 5

We cannot get a shell with a non-executable stack, this is because as the name suggest we cannot execute any binary or shellcode on the stack, this means that any of our code that is pushed to the stack won’t run. We do see on the stack our code is there, thus our buffer overflow attack worked, but our shellcode didn’t execute.

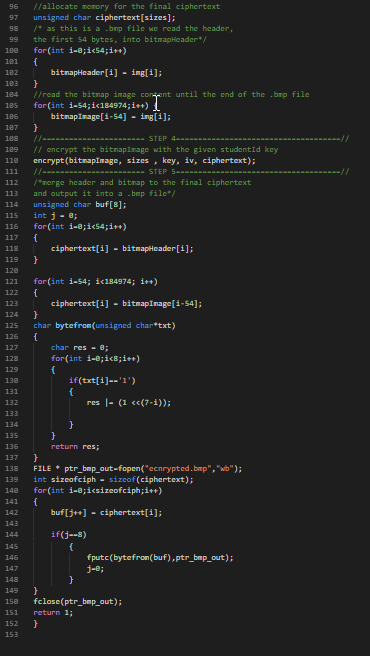




# Task 4 - 1 - Option 1



# https://i.imgur.com/twBTR7z.pngTask 4 -1 - Option 2



# Task 4 – Part 2

So, from the CBC encryption we cannot discern anything from the image, however from the ECB we can get a general idea of how the image looked, this is because of the way ECB works, ECB does not take any values from the previous encryption block and because the key isn’t changed each encryption we end up having the duplicates. E.g. say if we encrypt(2) and the ciphertext is a. Next time we come across a 2, it will also encrypt to an a. Thus, we get an image that looks discoloured, but the general shape is there. Though CBC works by feeding in information from the last encryption and thus making each new encryption different. This is reflected in the image as it’s a random mess.

