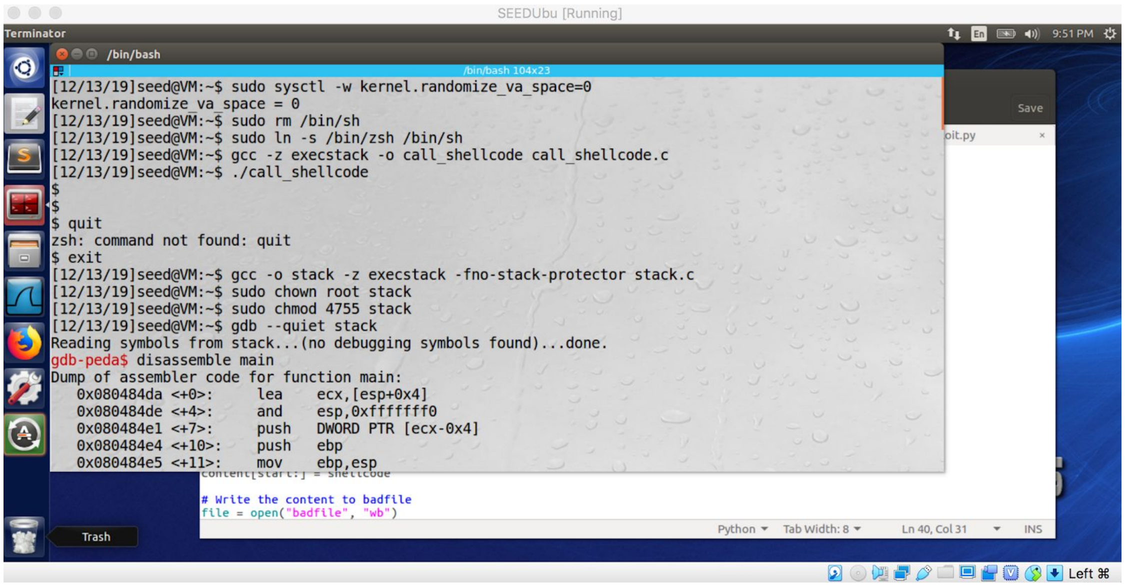
Gregory Gertsen
TCSS 481
Buffer Overflow Lab

ALL TASKS COMPLETED
<u>Lab Tasks</u>

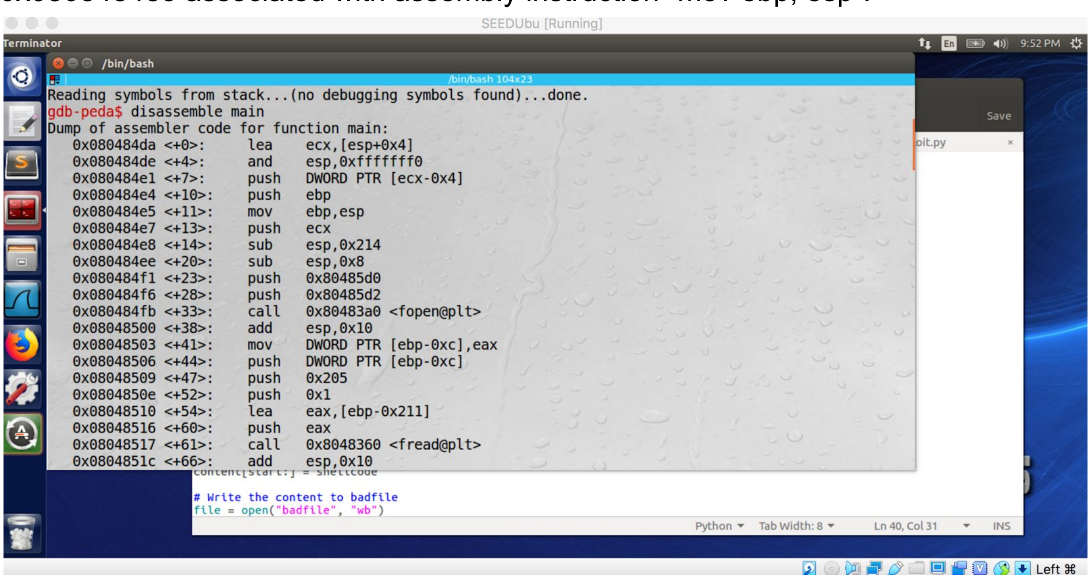
2.2 Task 1: Running Shellcode

Entering commands for disabling countermeasures, and compiling and running shell code. Also initiating a disassemble on the stack.c file's main method using gdb debugger.

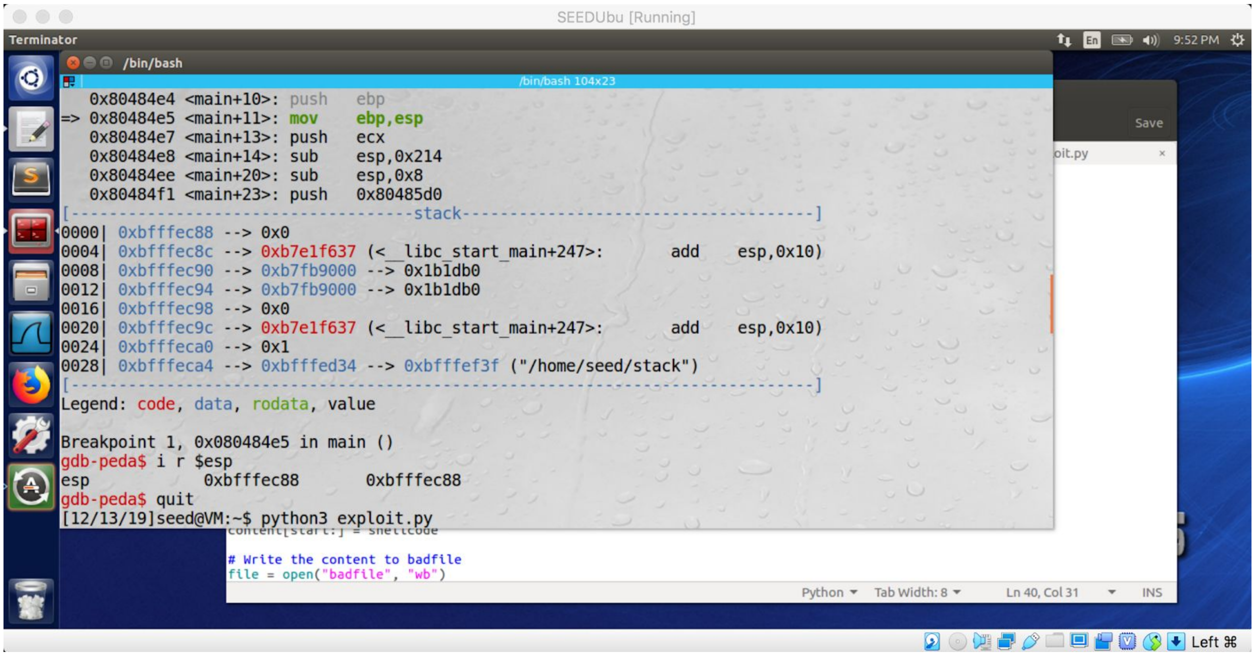


2.4 Task 2: Exploiting the Vulnerability

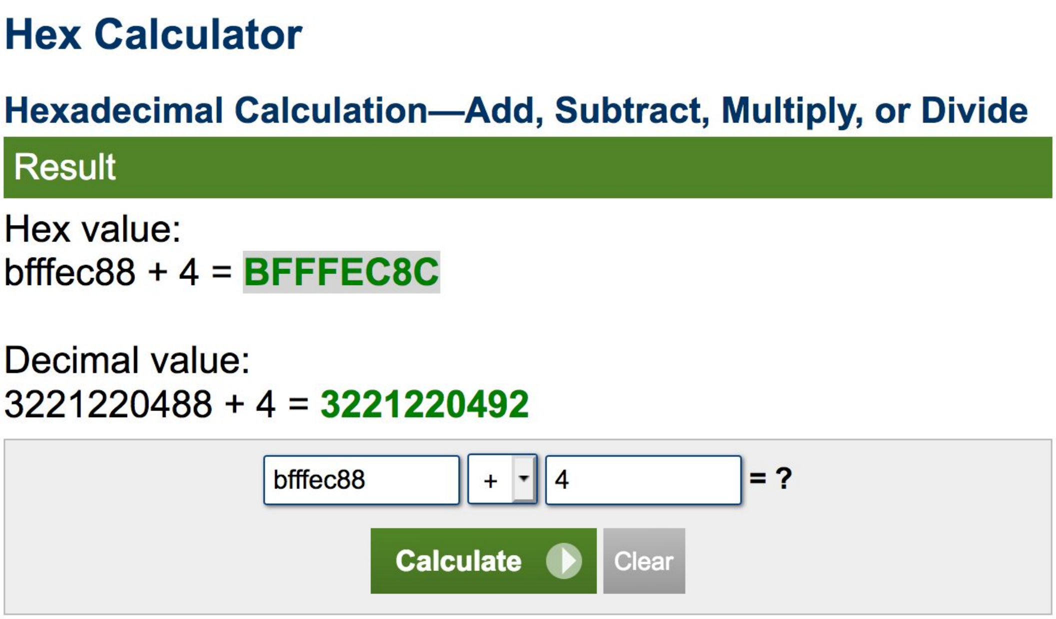
Identifying the base of the register with the disassembler. Put a breakpoint at address 0x0808484e5 associated with assembly instruction "mov ebp, esp".



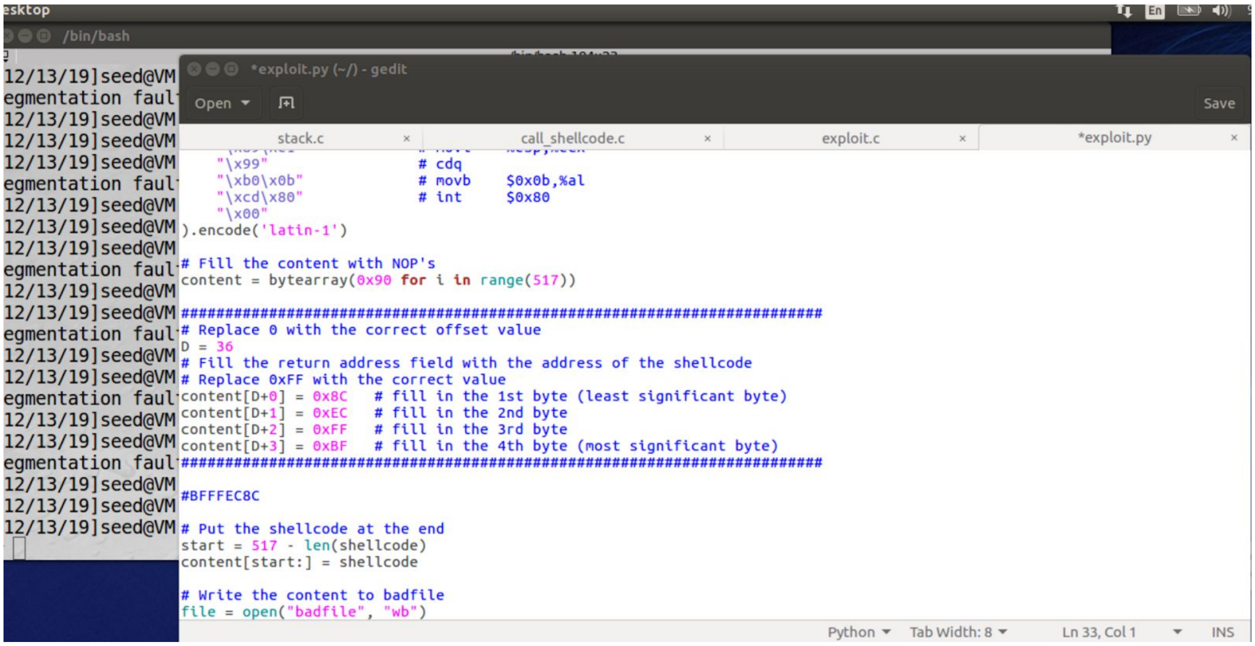
Use the command i r $esp to get the base pointer address. From the base pointer address we can calculate the return address. We can see that our base pointer address is 0x0bfffec88.



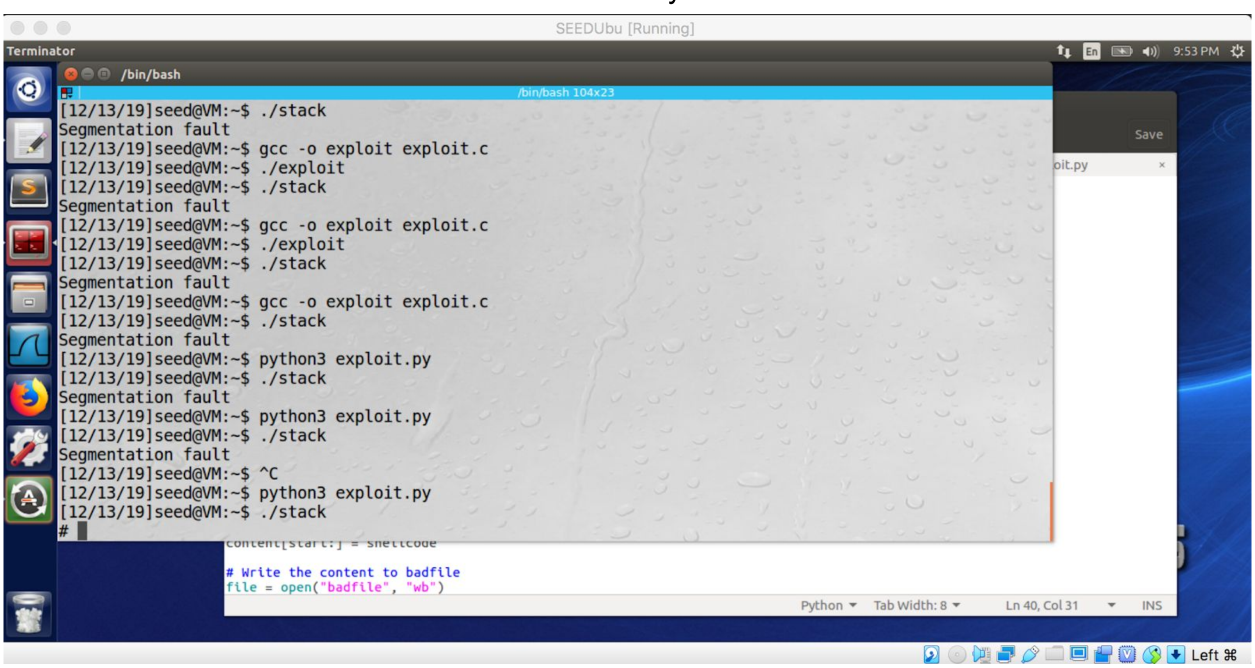
We then add 4 to the base pointer address to get the return address which equals 0xbfffec8c



Now that we know the base address we just need to narrow down the offset value which we can assume is near the 30s because there are 24 bytes for a buffer, 4 bytes for a previous frame pointer, and 4 bytes for an environment variable. The actual offset value is 36. The NOPs act as a sled pushing us towards our desired code we want executed.

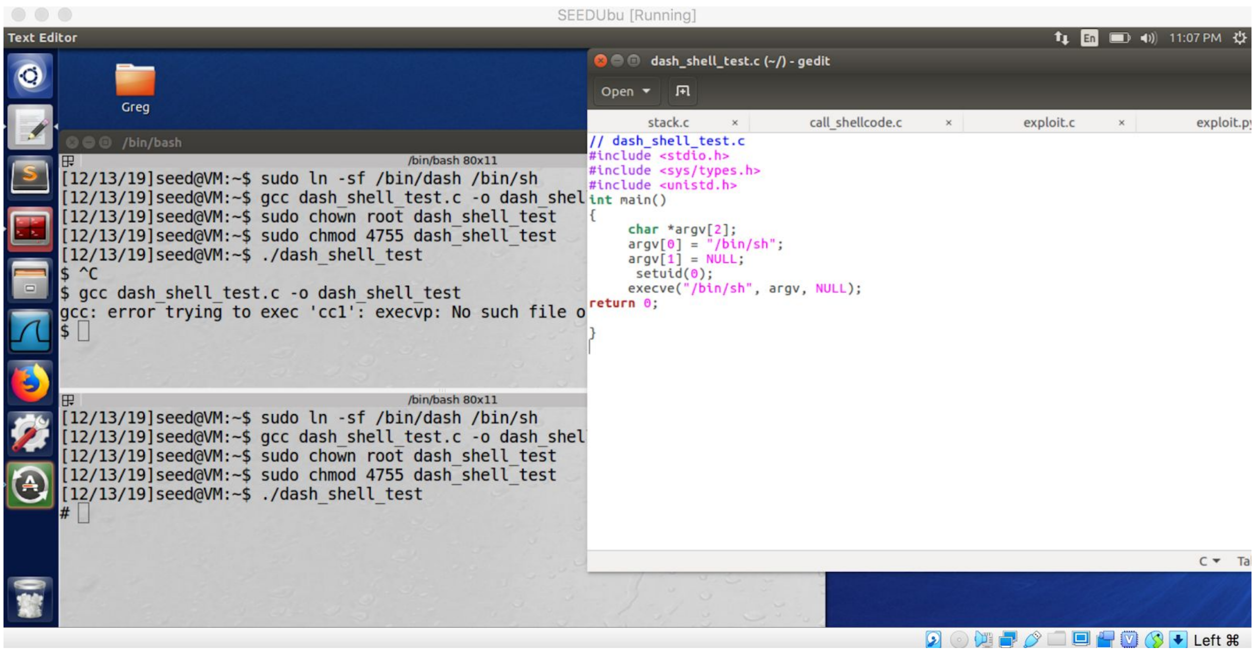


Running the program with different offset values gives us a segmentation fault until it doesn't, when we get it right it gives us root access. The offset value is 36 and the return address is 0xbfffec8c. You can see that we have root access symbolized with #.

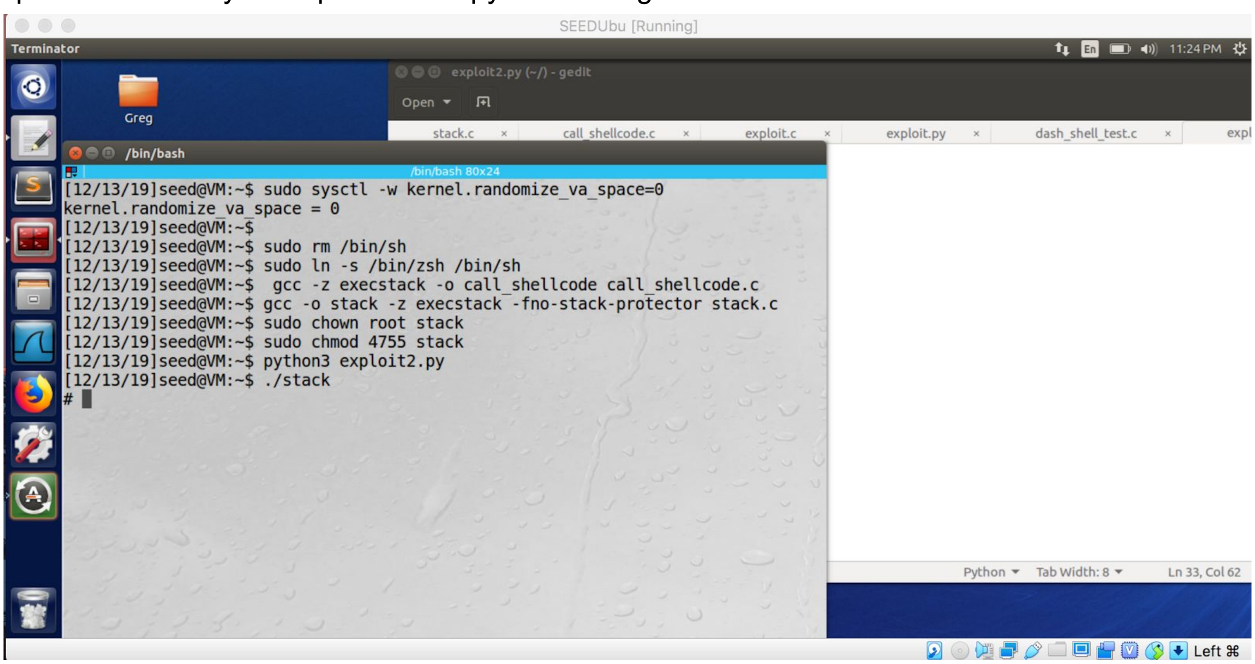


2.5 Task 3: Defeating dash's Countermeasure

Running the required commands, compiling and running first with commented out "setuid(0);" gives us a normal user prompt. The second time with "setuid(0);" uncommented we get a root prompt.



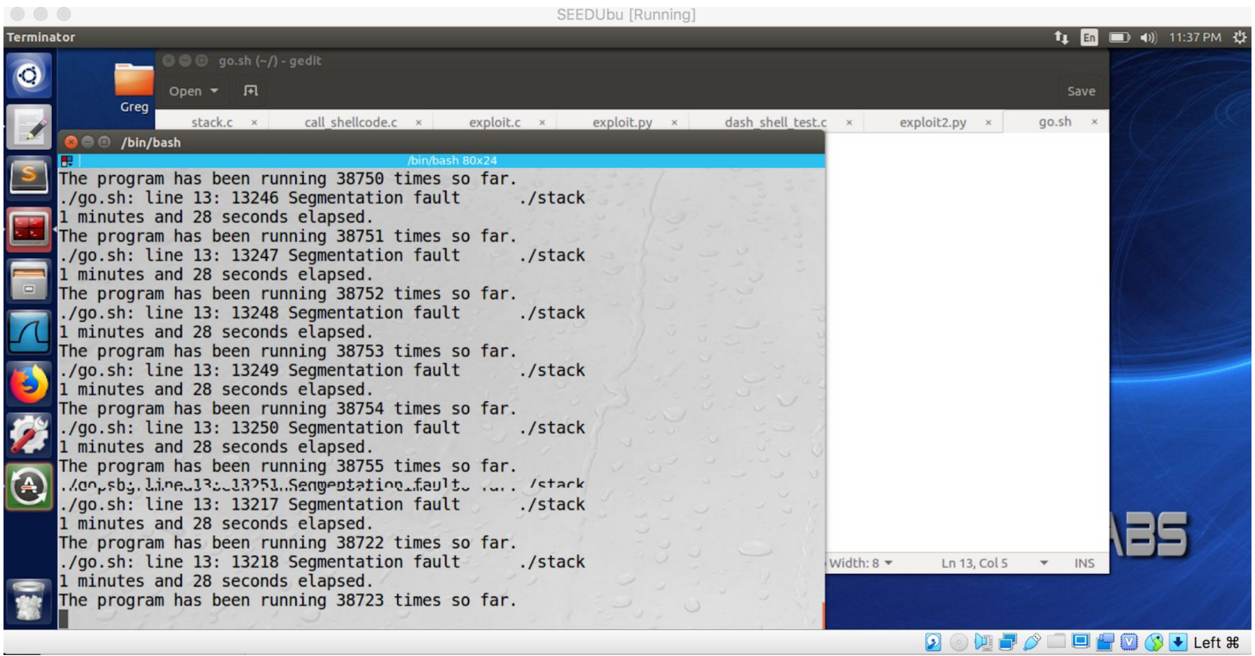
For the second part of this task we are asked to run the same attack again from task 2 using the updated assembly code put into the python file. I got the same results. Root access.



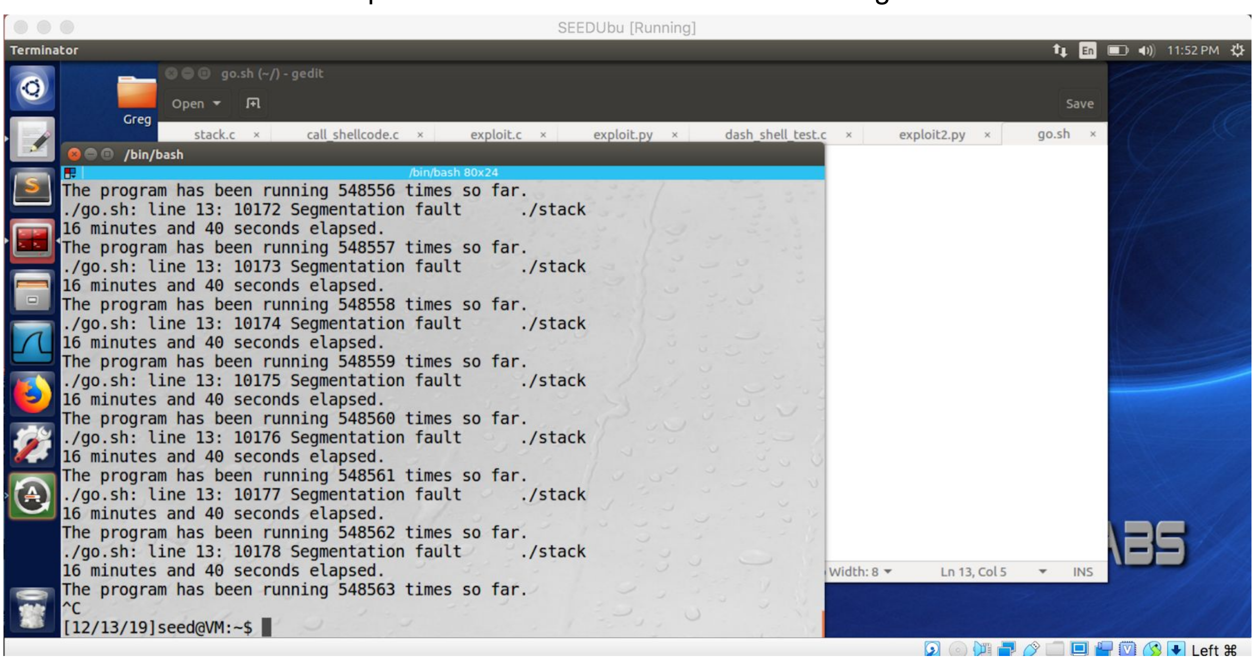
2.6 Task 4: Defeating Address Randomization

Address randomization is turned on which means the stack base address can have 2^19 = 524, 288 possibilities. So we use an infinite looping program to run our same attack from task 2.

Image of infinite buffer overflow attack running to test all available address possibilities.

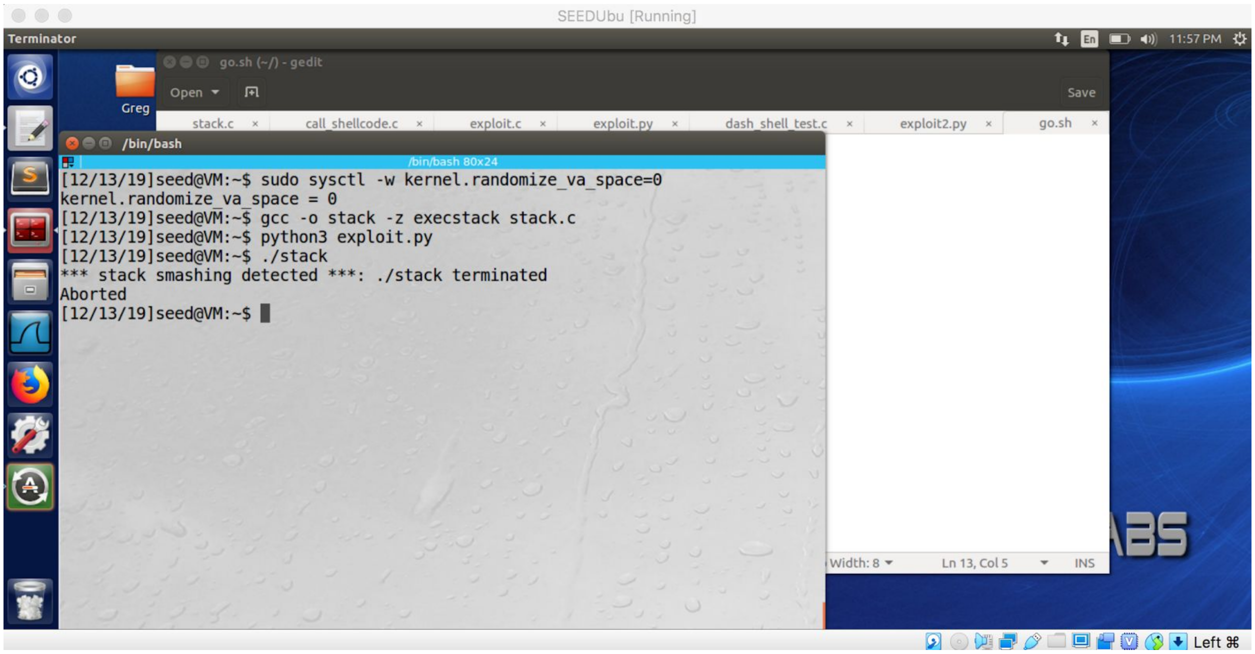


I ran the program for about 17 minutes straight and was not able to get root, however, I could see that with more time and patience the attack could still work using this brute force method.



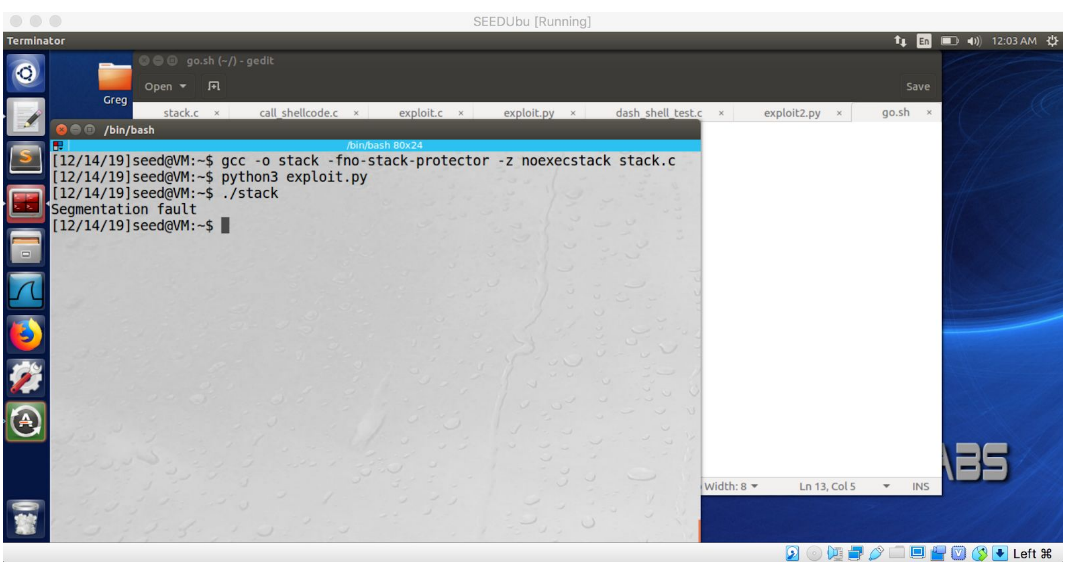
2.7 Task 5: Turn on the StackGuard Protection

For this task we run the same attack but we do not disable the stackGuard. The result is that the attack does not work and we get the following output from the terminal. Stack guard acts as a protection mechanism that detects from buffer overflow vulnerability.



2.8 Task 6: Turn on the Non-executable Stack Protection

In this task, we recompile our vulnerable program using the noexecstack option and we run the same attack. What happened was that the attack was not successful and I got a simple segmentation fault error. The non-executable stack acts as a protection mechanism and it provides protection that avoids code from being executed from the stack.



END