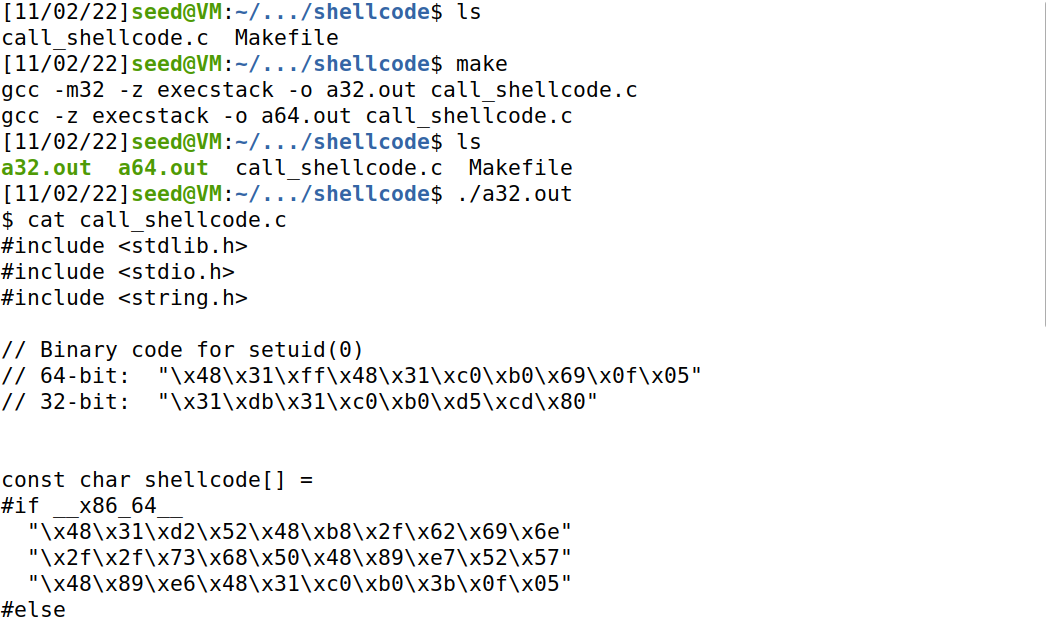
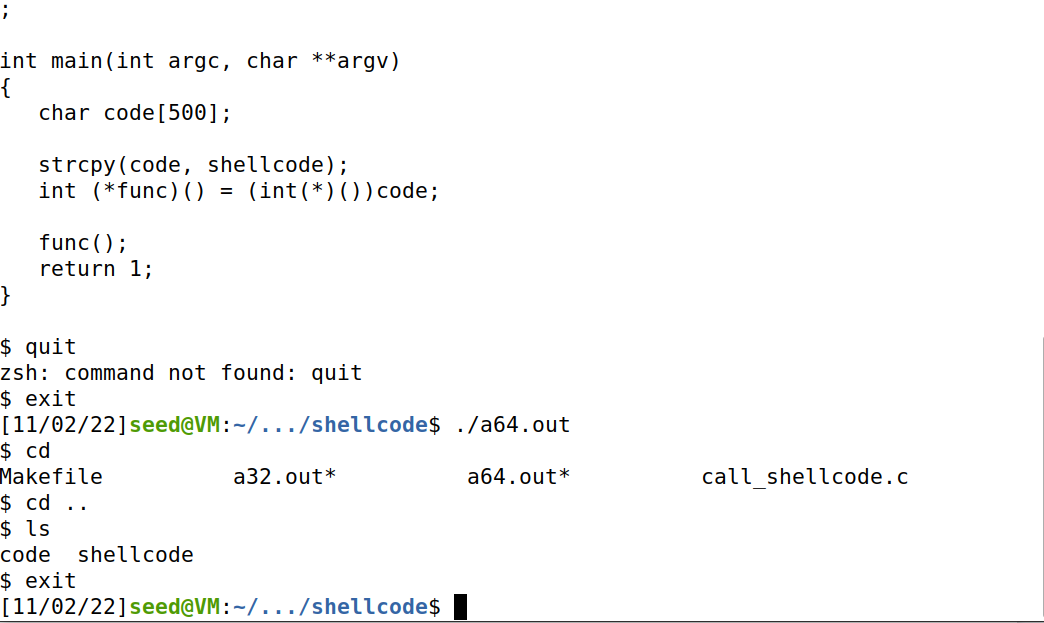
Jourdon Freeman

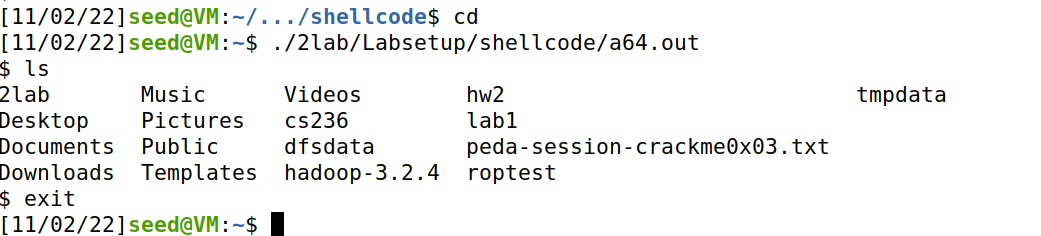
862006435

Lab 2: Stack Buffer Overflow

# Task 1: Getting Familiar with Shellcode

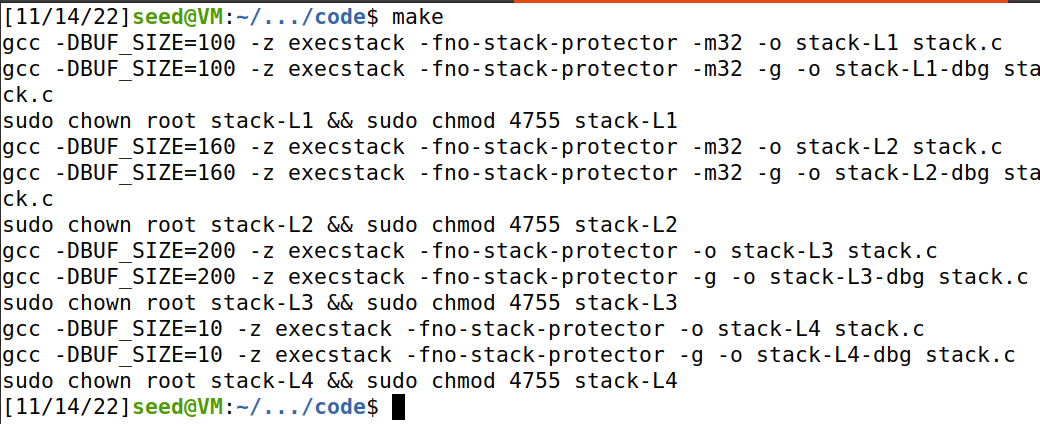






Unlike the normal terminal, the shell terminal generated from the overflow doesn’t recognize backspace. The shell starts off in whatever folder it is run in and grants the person running it to go anywhere they want.

# Task 2: Understanding the Vulnerable Program



L1: 100

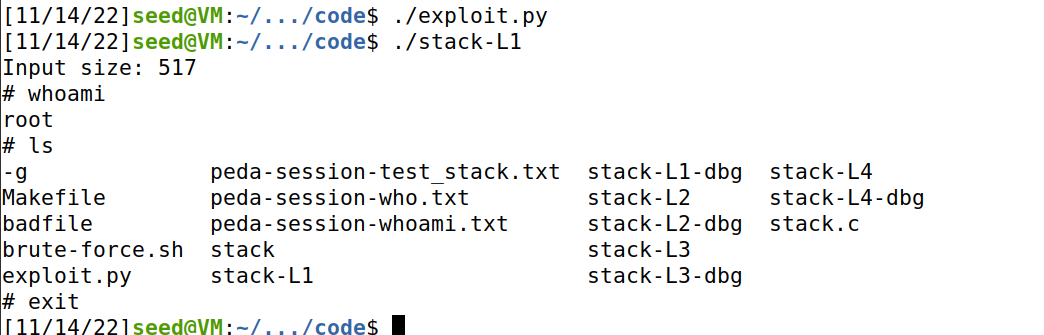
L2: 160

L3: 200

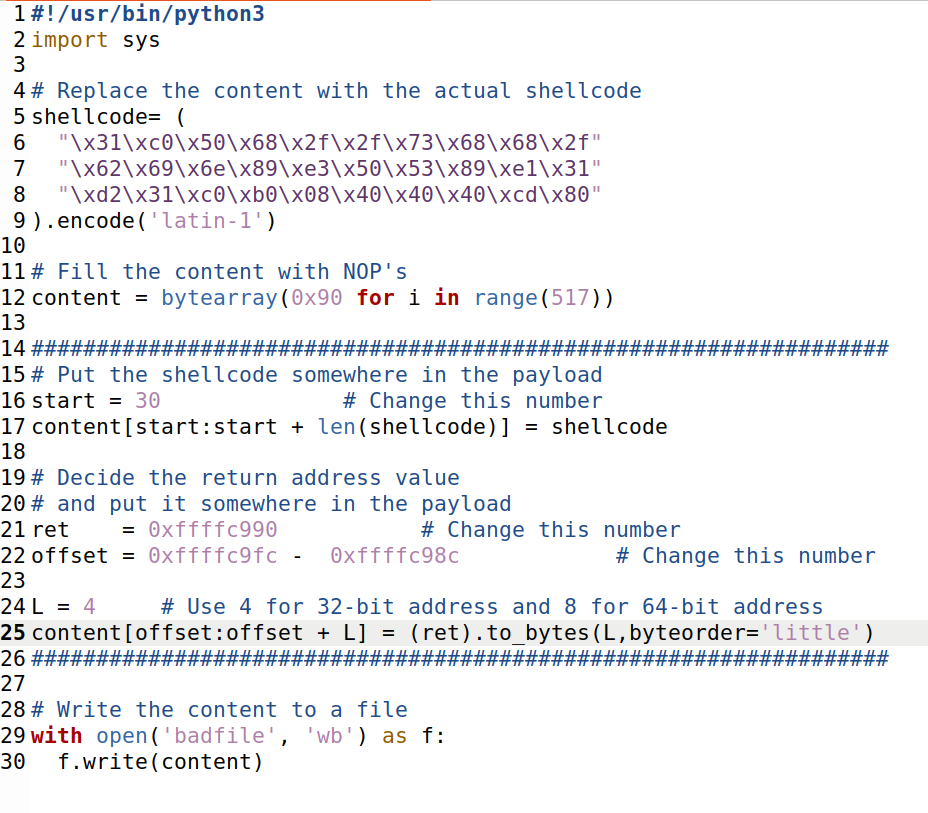
L4: 10

# Task 3: Launching Attack on 32-bit Program

### Terminal

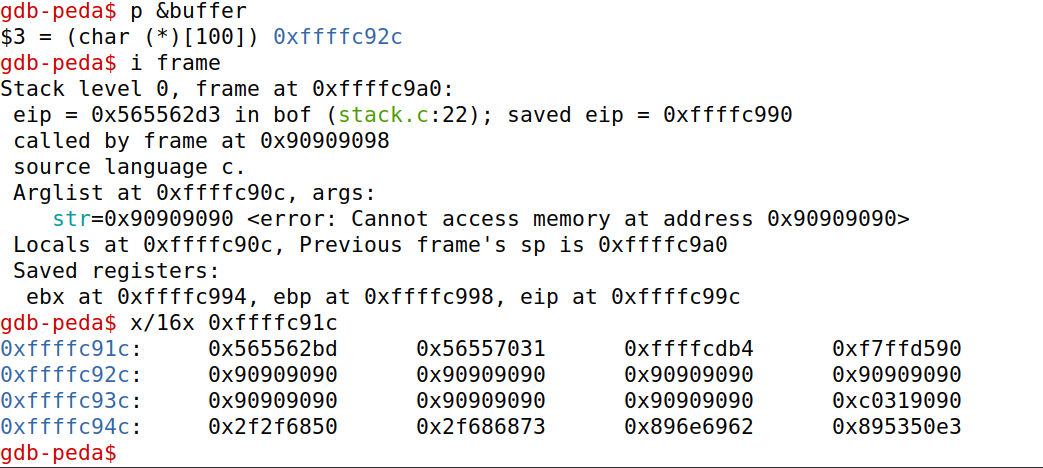


### exploit.py



The exploit file gave us 3 values to change: start, ret, and offset. The array we are inputting is called ‘content’. The content array begins with the NOP operation (x90) repeated 517 times. Because of the overhead GDB provides, the exact addresses are different between what GDB says and what the program actually uses. However, the distance between everything is the same. Using the print function, I was able to find the address stack-L1 is actually using for the buffer and adjusted the addresses in the exploit file accordingly.

The start value is the index in the content array the shellcode will begin at. In order to determine this, we need to figure out its length and the total space in the local address space in the bof function. We can do this using GDB.

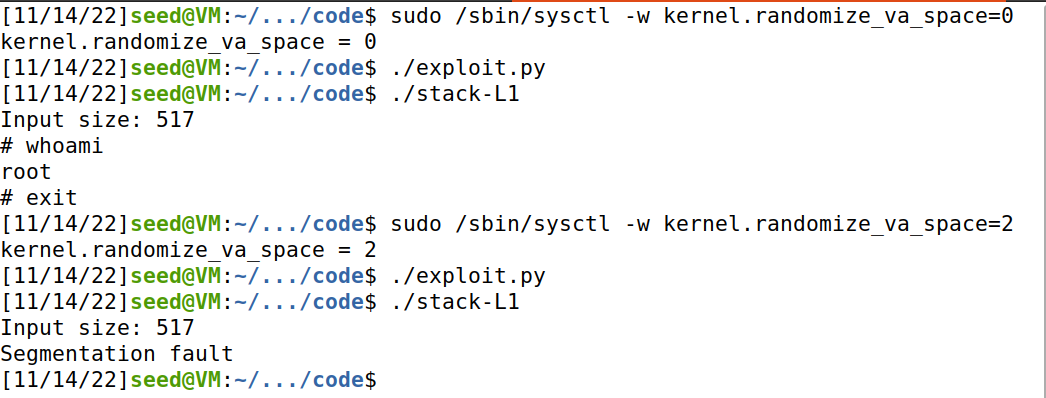


Here we can see that the start of the buffer/content array begins at 0xffffc92c, EBP is at 0xffffc998, and EIP is at 0xffffc99c. The gap between the start of the buffer and EIP is the space we have available which is 116 bytes. Given the size of the shellcode is 30 bytes, the value of start can be anything less than 86.

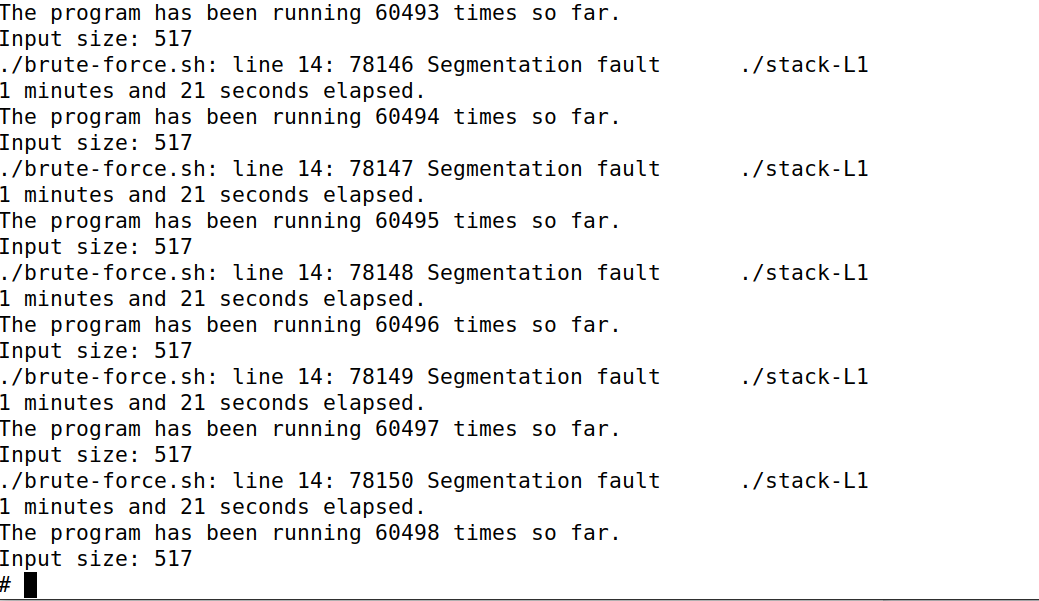
RET is the value that goes into EIP and the address the stack pointer goes back to in the buffer. The value of RET can be anything in the NOP sled before the shellcode starts. In this case, I chose 0xffffc990 (in the GDB version I tested with, I used 0xffffc940).

The offset is the difference between the start of the buffer and EBP plus L. L was given and the other two were found earlier. All that has to be done is to plug in.

# Task 8: Defeating Address Randomization

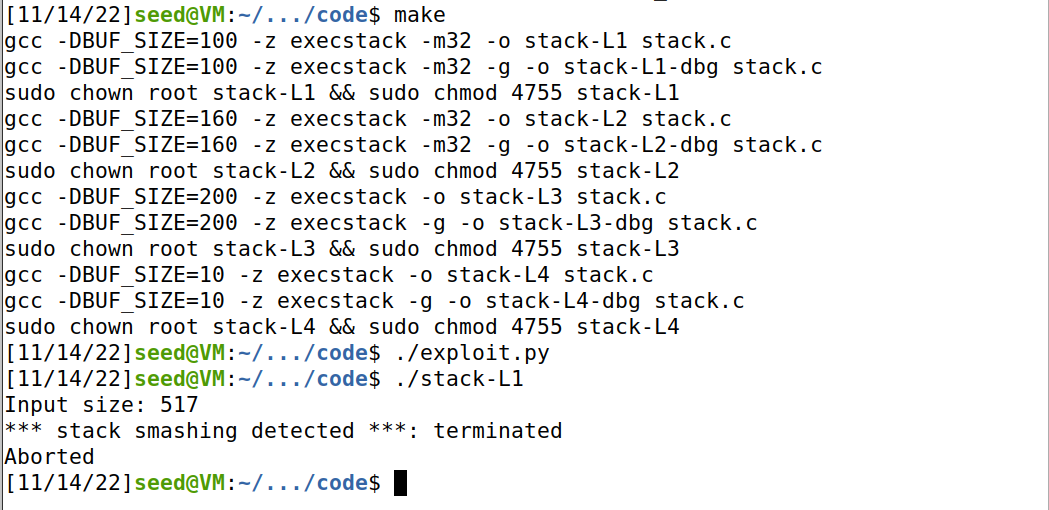


Once randomization is turned back on, the program segfaults. This is because the addresses have now changed so the values I chose for exploit.py are now incorrect.

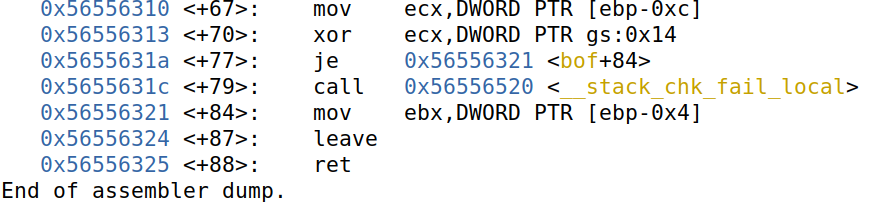
When running the brute-force shellcode, it took 60498 runs before stack-L1 used the addresses I placed in the exploit file.

# Task 9: Experimenting With Other Countermeasures

### StackGuard On

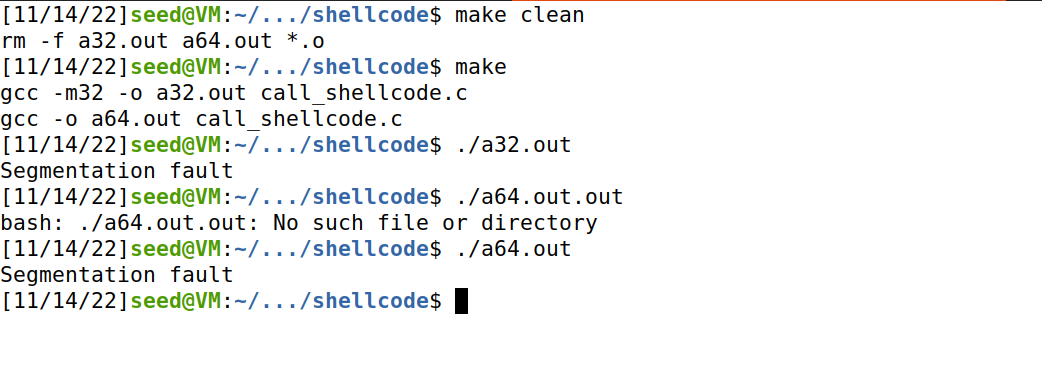


When StackGuard is on, the shellcode is detected and the program aborts.

****

In the disassembly in GDB, there is a new line stating that the local stack check had failed. After hitting next, the program seems to abort after it finishes setting up the stack.

### Non-executable Stack



Without the -z execstack flag, both programs simply segfault since nothing in the stack can be executed.