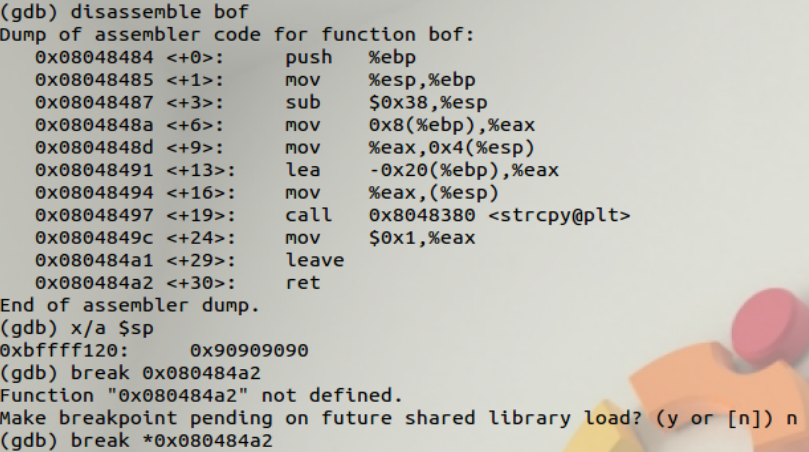
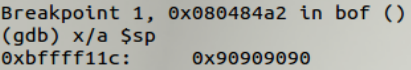
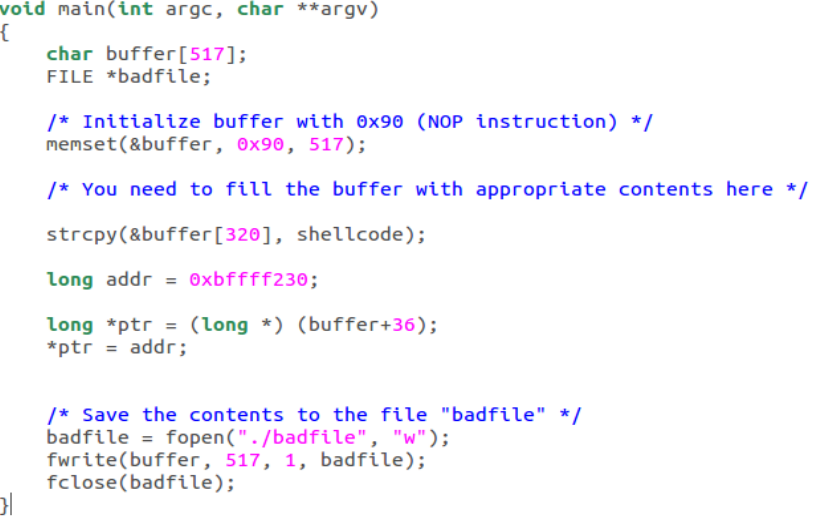
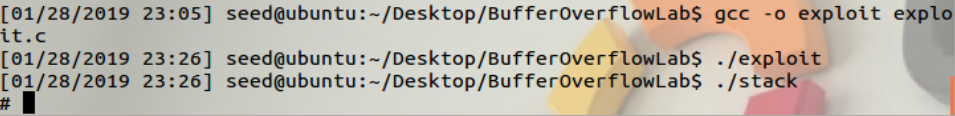
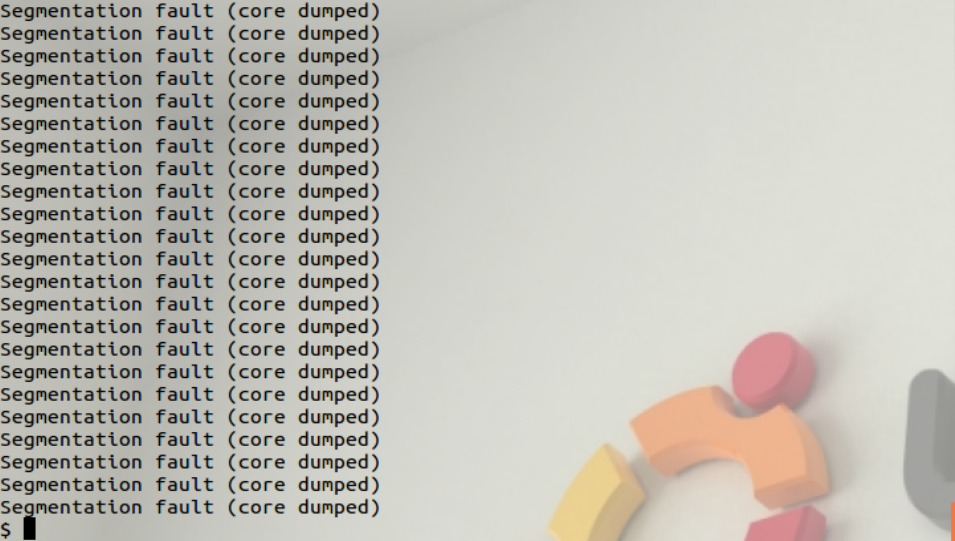
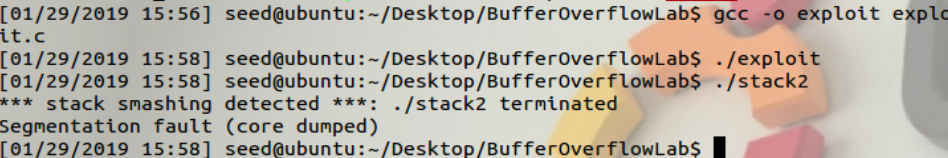
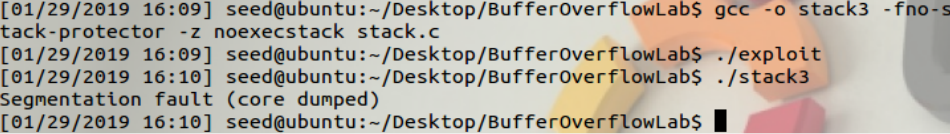
**Buffer Overflow Lab**

* Task #1: Exploiting the vulnerability.
  + We are given exploit code which we need to finish for the exploit to work. Specifically, we are writing a badfile whose contents contain a buffer of 517 bytes and the exploitable program has set aside a buffer of 24 bytes and then reads in the badfile and strcpys the 517 bytes into the 24 byte buffer allowing us to overflow the buffer to run our malicious code. The program runs main which runs bof which contains our buffer and has a return address associated with it that we can overwrite.
  + After placing the shellcode randomly in the buffer a few times and failing, I figured that finding the return address and pointing it to my shellcode would work. One way we can do this is by first checking where the base pointer is located at the time of last instruction of bof.
  + After disassembling bof using gdb, we can see that the last instruction is at 0x080484a2. Thus, we can add a break at this address.
  + We know that the stack pointer is pointing to the return address since the function is now returning at this point. This address 0xbffff11c is thus our return address that we want to overwrite. Currently it is filled with NOP’s as a result of my random placement of the buffer, resulting in error.
  + But now that I know what the return address is, I should figure out which address I want to overwrite it with. This simply needs to be an address before my shellcode so that the when the function returns, it returns right before my shellcode and runs it thereby allowing me to open a shell.
  + My buffer consists of a bunch of NOP’s, shellcode placed 320 bytes in, then more NOP’s. Thus, by mapping the stack again but allowing for many more addresses, we can find the place where our shellcode is written. On the line of 0xbffff230 the NOP’s end and other characters are placed. This is start of the shellcode.
  + We also know this is the start of the shellcode because I placed it exactly 320 bytes into the buffer. Quickly calculating we can count 19 lines between them of 16 bytes each (16\*19) plus 8 bytes before and after => (16\*19) + (8\*2) = 320.
  + Jumping to the address 0xbffff230 should be more than sufficient to get the shellcode running.
  + Now that we know the return address and the address to overwrite it with, all we need to do is overwrite it using our buffer. The return address 0xbffff11c is in our buffer, so we can find it on the stack.
  + Simply counting from the start of our buffer, each segment being 4 bytes with 9 segments until our return address (0xbffff11c), we get 4\*9 = 36 bytes. That is, the return address is located 36 bytes from our buffer. All we need to do now is put it in the exploit.
  + Running through the code: We first strcpy the shellcode 320 bytes into our buffer then place the address 0xbffff230 exactly 36 bytes after the buffer (overwriting the value for 0xbffff11c).
  + Compile the exploit and run it to create badfile, then run the program.
  + We opened a shell!
* Task #2: Address Randomization.
  + Allowing for address randomization, our program from task 1 should still work given enough time but it can be made faster with some techniques. After waiting a long time and not hitting anything, I decided to move the shellcode towards the end of my buffer making it less likely for my address to send it somewhere past my shellcode and thereby miss it completely. In addition to this, the address randomization (ASLR) on the VM only changes these bits 0xbfXXXXXc, where the X’s are randomized. So I found the median of 1,…9,a,…,f to be 8, and so I replaced the last four X’s with 8’s and the first with an ‘a’. Within 15 minutes I was able to open a shell.
* Task #3: Stack Guard.
  + After recompiling the vulnerable program with stack guard enabled we can repeat task 1 but we receive a prompt that states “stack smashing detected” and the program is immediately terminated. Seemingly with stack guard enabled, it is a lot more difficult to perform a buffer overflow exploit as any overflow will be detected by the program.
* Task #4: Non-executable Stack
  + After recompiling the vulnerable program with a nonexecutable stack and repeating task 1, we receive an error that says “Segmentation fault (core dumped)”.
  + Our code seems to be working as intended, but a nonexecutable stack seems to make it so that shellcode can’t be run on the stack. Therefore, another method of attack will have to be used to open a shell.