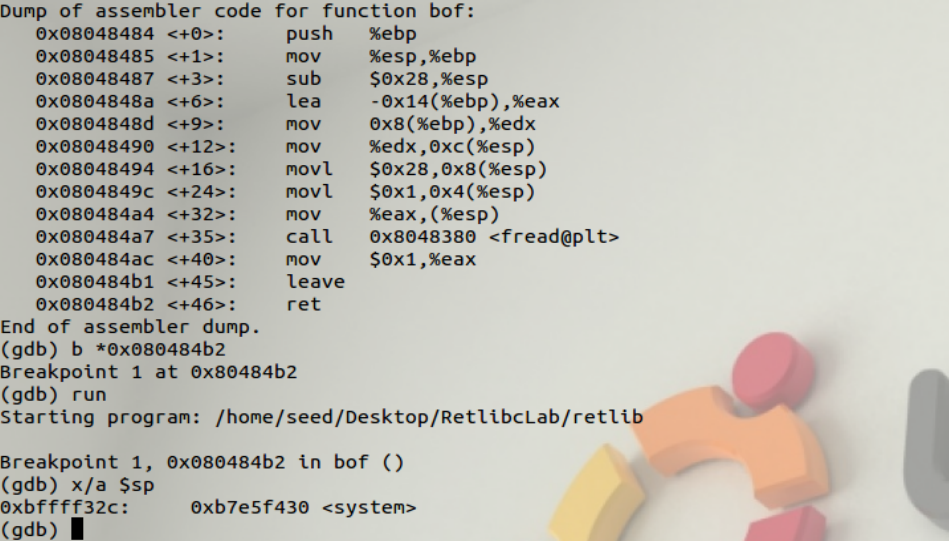
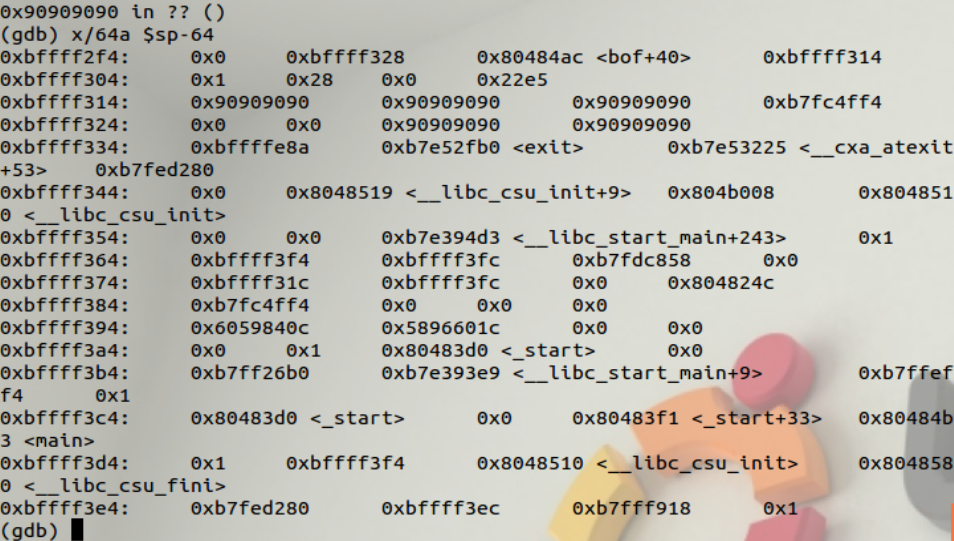
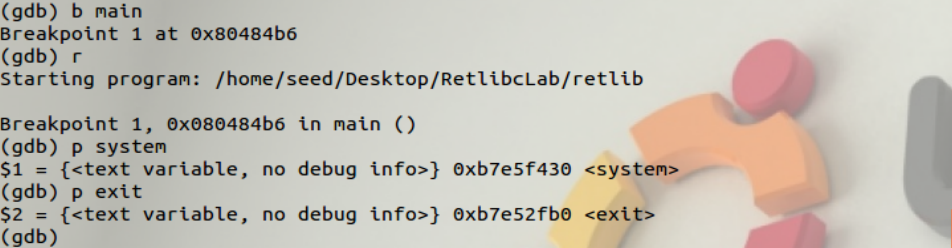
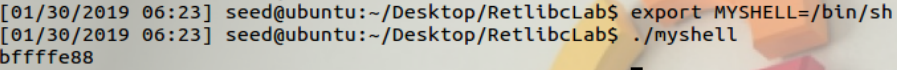
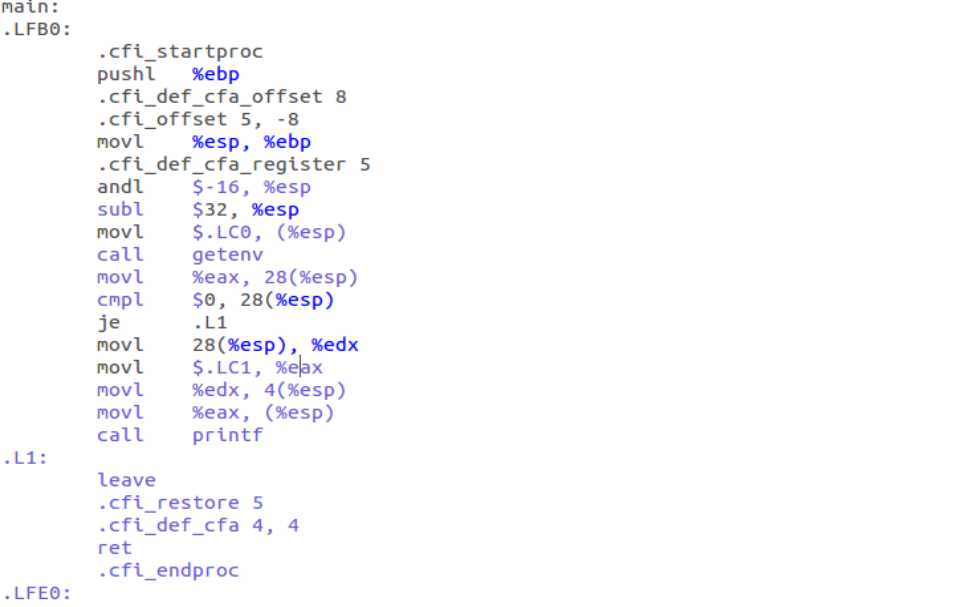
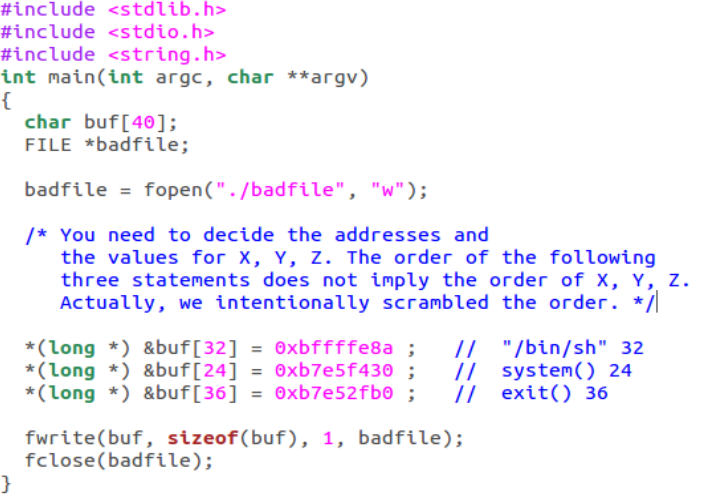
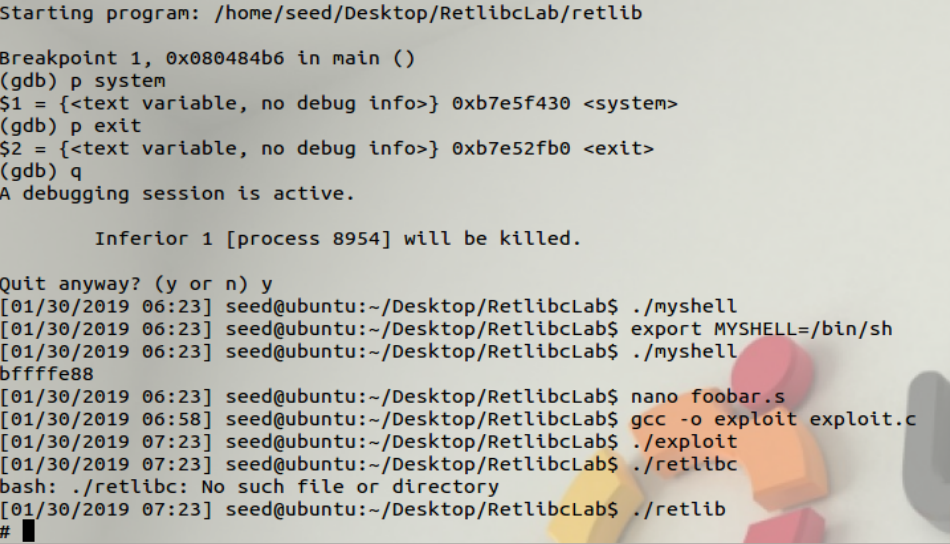
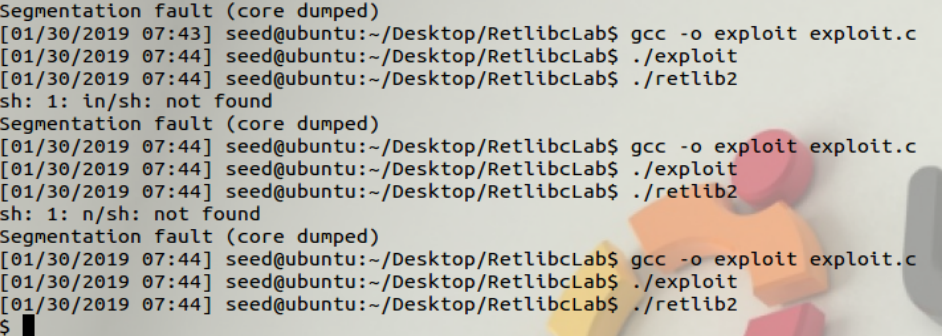
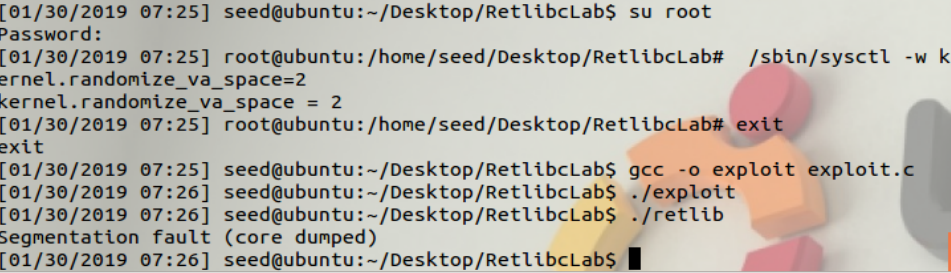
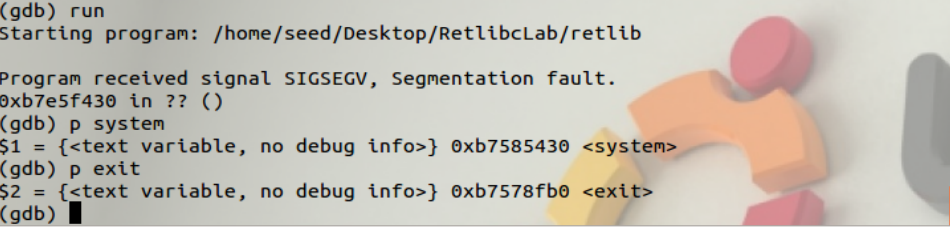
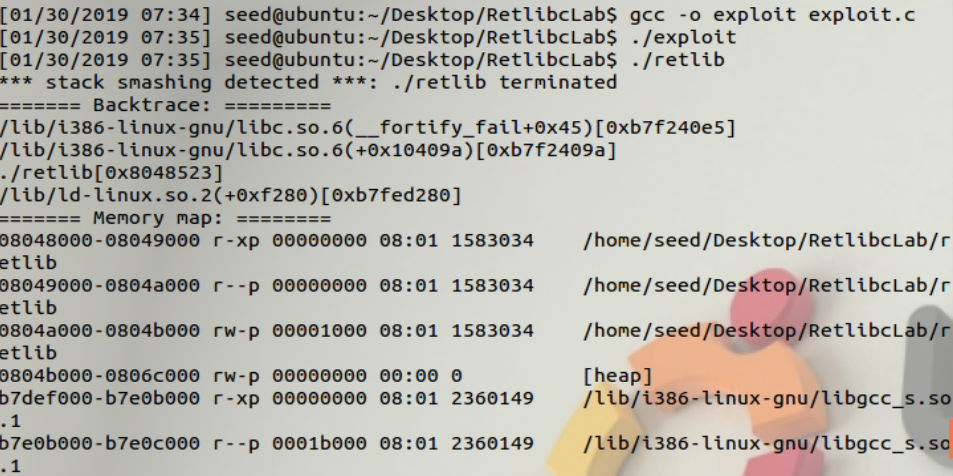
**Return to libc attack**

* Task #1: Exploiting the vulnerability.
  + There are several parts to this return to libc attack. First, we can find the return address of the bof function in the program by using gdb to breakpoint at the final instruction of the bof function. As we know, the stack pointer will be pointing at our return address at this time (just as in buffer overflow lab).
  + With the knowledge of the return address (0xbffff32c), we can investigate our stack to find where it lies in regard to the buffer. We can fill the stack with NOP’s to make it easier to see where the buffer gets inserted and then calculate the distance.
  + We can see that it is 24 bytes away from the start of the buffer so now we have our location set for the return address. Next we want this return address to be the address of the system so we can then access /bin/sh. We can find the system address (and the exit address) using gdb on any program and using the functions below.
  + Our server address is 0xb7e5f430 and our exit address is 0xb7e52fb0.
  + Now, we can go ahead and place our /bin/sh code near system and we can do this using an environmental variable and then use the program provided in the lab to get its address.
  + The address to open a shell is 0xbffffe88.
  + Now that we have most of our values and addresses, we can begin constructing how the stack frame will look once our functions start calling.
  + Taking a look at assembly code for a sample program given in the lab, we can begin to see the instructions that happen once a function is called.
  + Every function begins with a push instruction and then a move instruction. The push instruction pushes ebp (old base pointer) onto the stack and the mov instruction makes the stack pointer (esp) to point to the old base pointer. Then there are arguments added to memory and the stack pointer moves to the end of those, but once it is finished it is then moved back, following a leave.
  + Every function has a leave instruction and a ret instruction. For leave, the ebp is moved into the esp which means the esp now points to above the old base pointer. Next, the ebp is popped and the old base pointer is gone. Ret then pops the return address and jumps us there. Further instructions add back more memory and the stack is the same it was beforehand.
  + We once we jump to system, we want to get to the address of bin/sh/. We can calculate this to be ebp+8, because first we have the argument, then the return address, then the old base pointer. We know from above that the esp points to the return address, so we need to go up 4 to bypass it, and then up 4 more to bypass the argument and reach the address of bin/sh/.
  + Therefore, we only need to go up 8, giving us 24+8=32.
  + Similarly, from that point on, we need only go up 4 more to reach the exit giving us 32+4=36.
  + Putting it in code, we have the following (no longer need NOP’s).
  + One thing to note is that the /bin/sh address wasn’t completely accurate when we found it using the other program, but it is very close. Adding 2 to the hexadecimal address gave 0xbffffe8a which caused the exploit to work.
  + We can also perform this exploit after we rename retlib, I chose to rename it to retlib2. It doesn’t work with the same /bin/sh/ address, but it is still very close to it. I changed the address back to 0xbffffe88 which is the address I found from the program given in the lab and I was able to spawn a shell.
* Task #2: Address randomization.
  + Turning on ASLR and then redo-ing task 1 gives us a segmentation fault error but not as easily fixable as it was in buffer overflow lab.
  + This is because even if we manage to hit the first overflow return address, the address of the system AND the exit also change, so we would have to be a lot more “lucky” than before.
* Task #3: Stack Guard Protection
  + We can now disable stack guard protection and redo task 1 and receive another error. This time our stack smashing is detected and we cannot do an attack using the overflow technique that we’ve been using. Another method must be used.