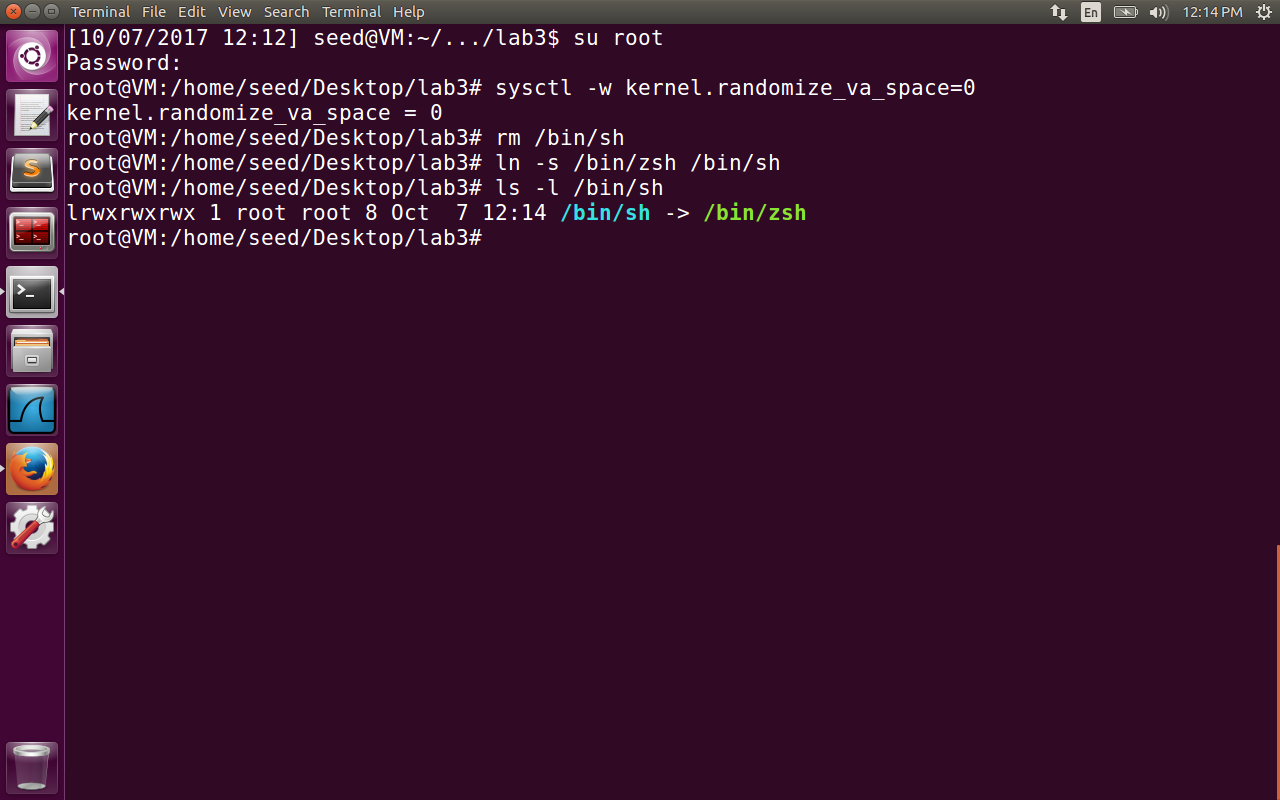
Return-to-libc Attack Lab

Name: JASHWANTH REDDY GANGULA

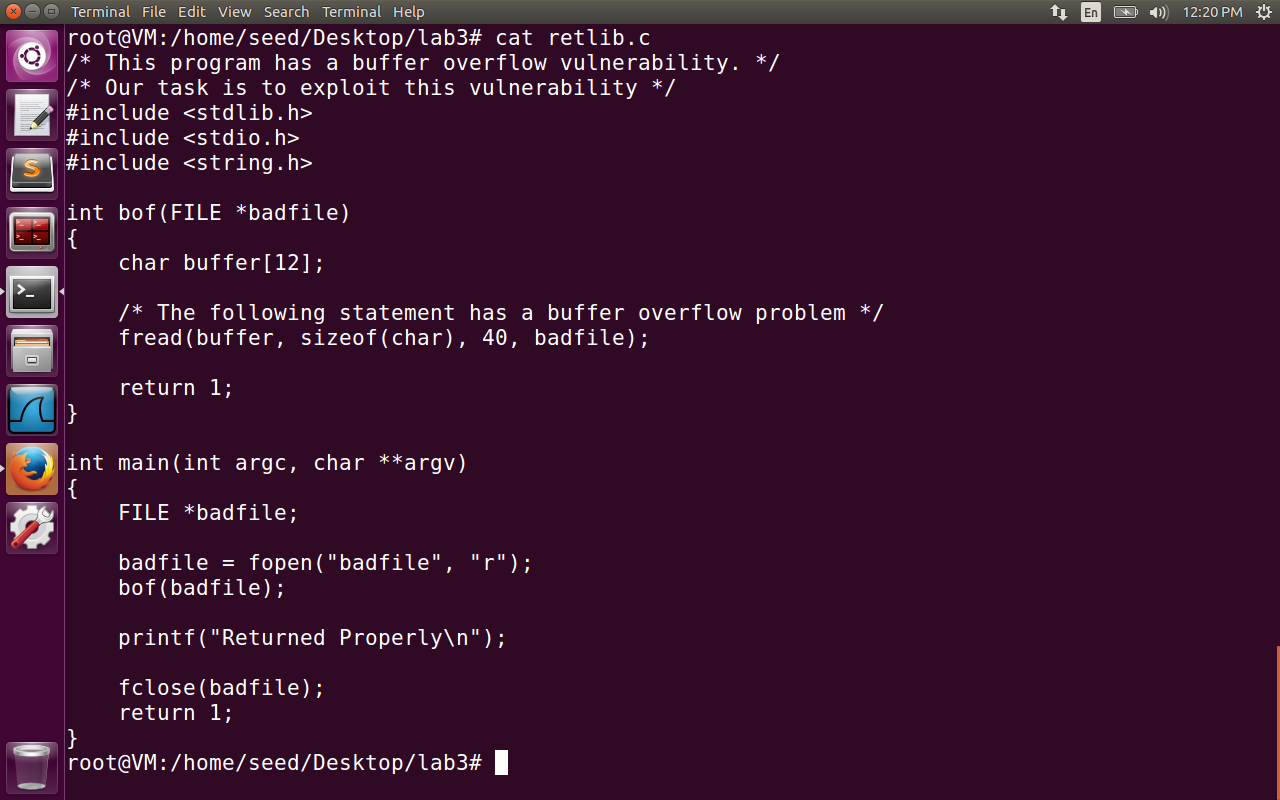
SUID: 646254141

2.1 Initial Setup

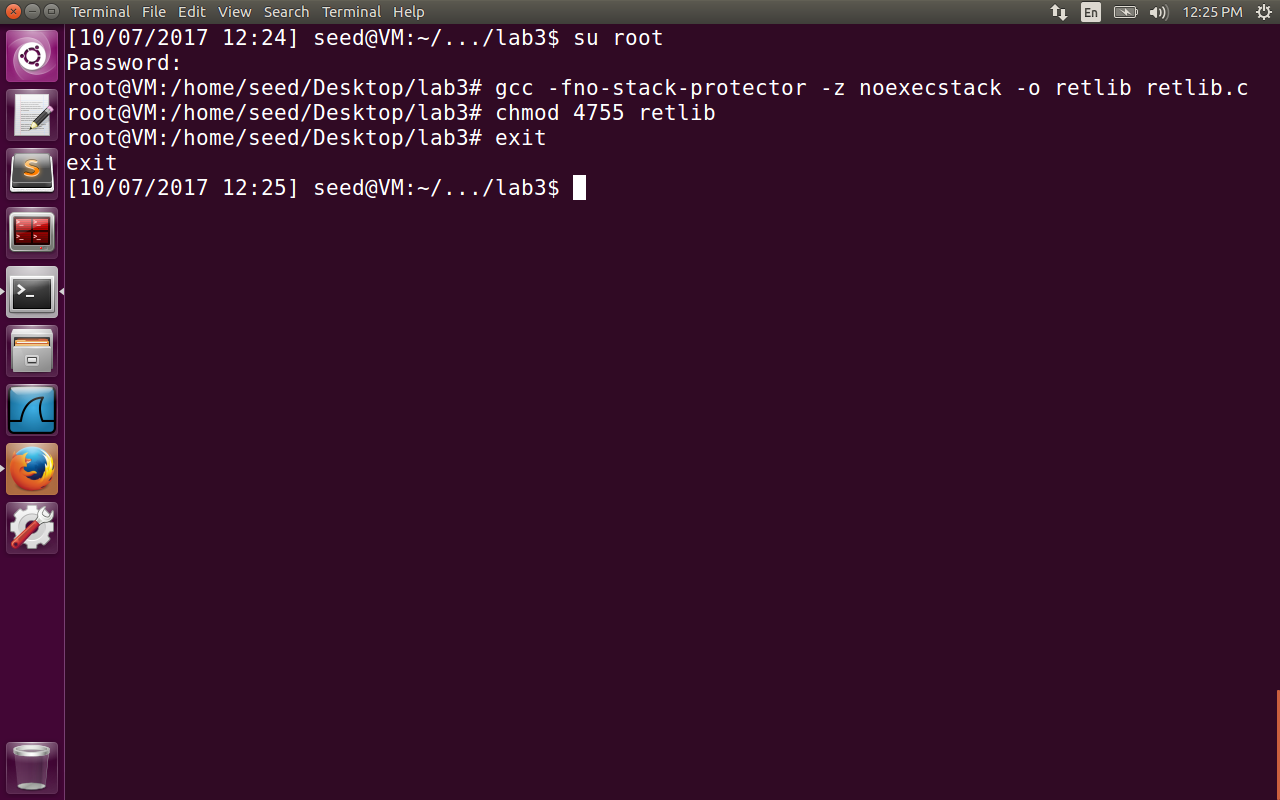


Ubuntu and several other Linux-based systems uses address space randomization to randomize the starting address of heap and stack. This makes guessing the exact addresses difficult; guessing addresses is one of the critical steps of buffer-overflow attacks. We disable the ASLR feature. Also in 16.04 VM dash shell is known to drop set-uid previliges. Hence we symbolically link /bin/sh to /bin/zsh.

2.2 The Vulnerable Program



The retlib.c program has buffer overflow vulnerability. We need to fill the contents of the badfile. We need to jump to the system() function of libc library which is already loaded to memory. From the system() we will jump to exit() function for the program not to crash and get the root shell access.



The retlib.c is compiled with stack guard disabled. The objective of this lab is to show that even after giving the noexectsack option, the attack can be done. After compilation made the retlib a set-uid program.

2.3 Exploiting the Vulnerability

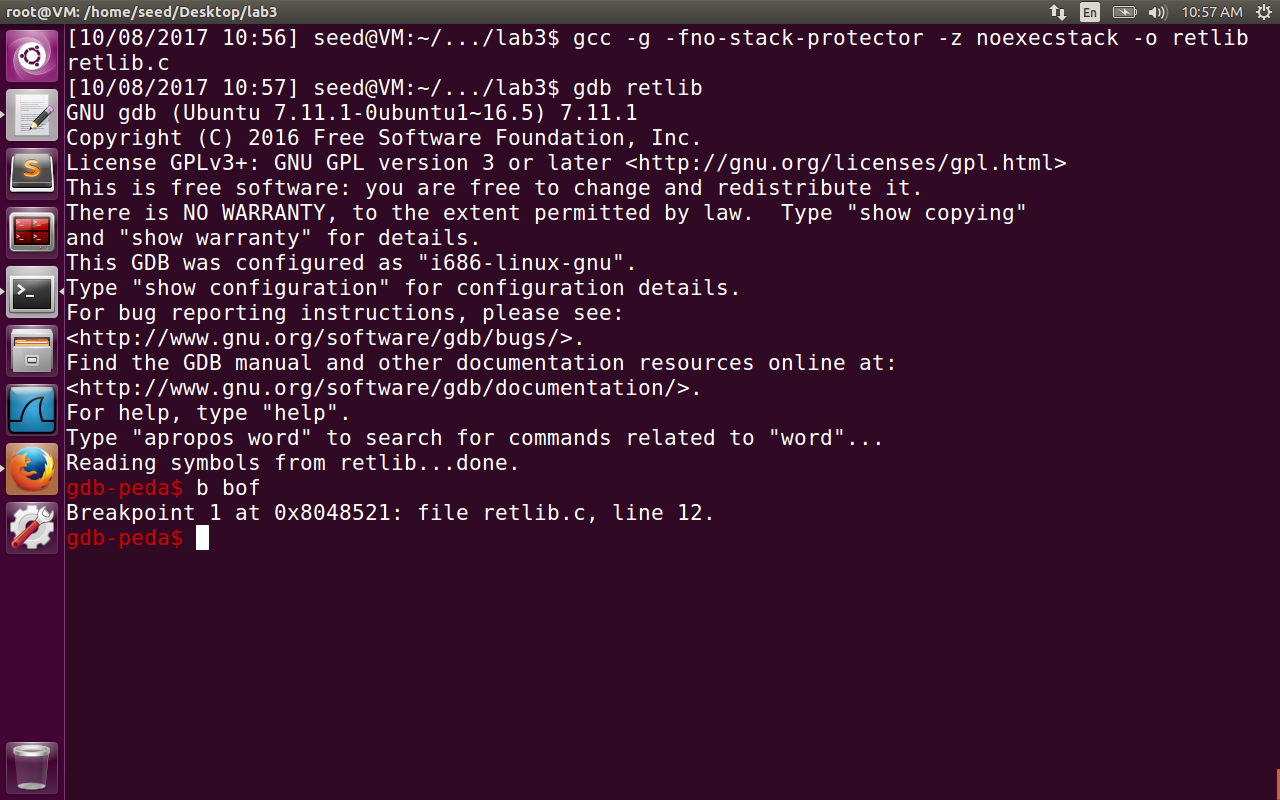
First the retlib.c is compiled using gdb option as a normal set-uid program to find the address of

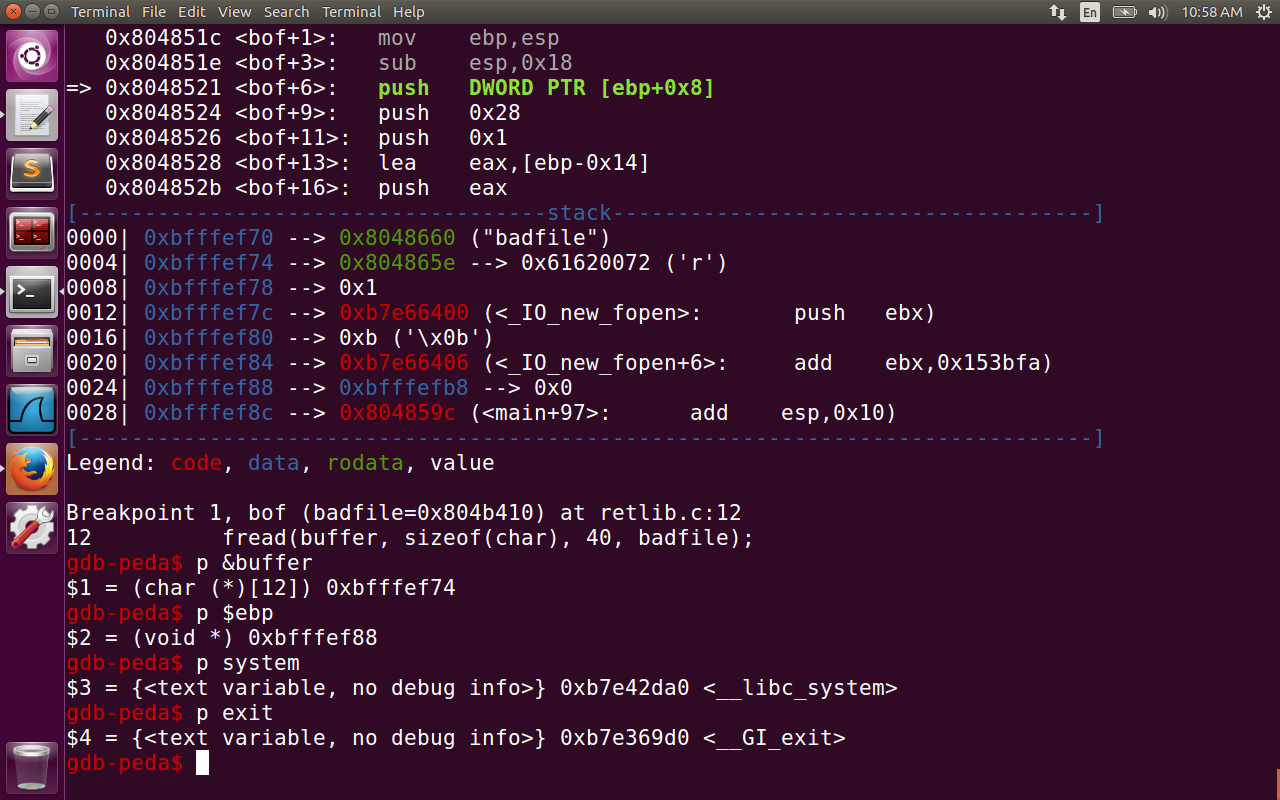
1)address of system

2)address of exit function

3)address of ebp

4)address of buffer array starting





Found the address of ebp = 0xbffffef88

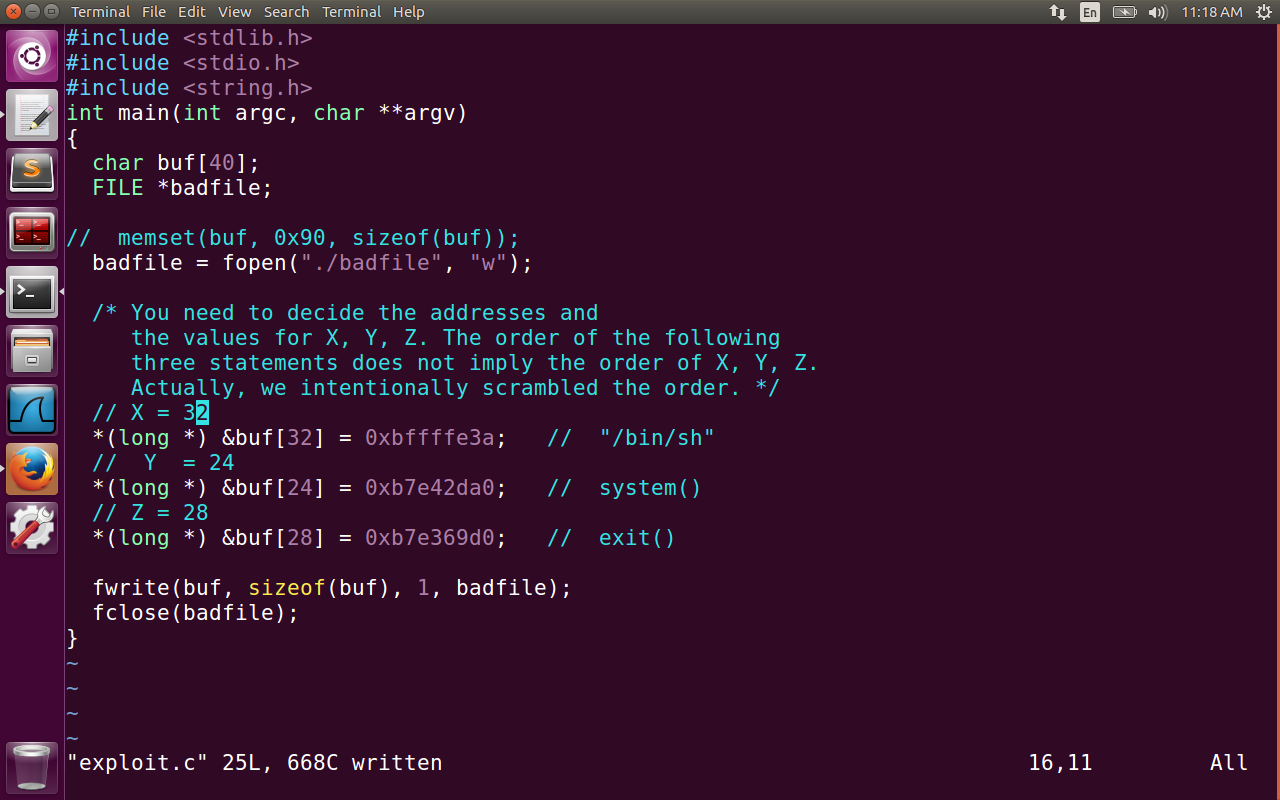
Address of buffer = oxbfffef74

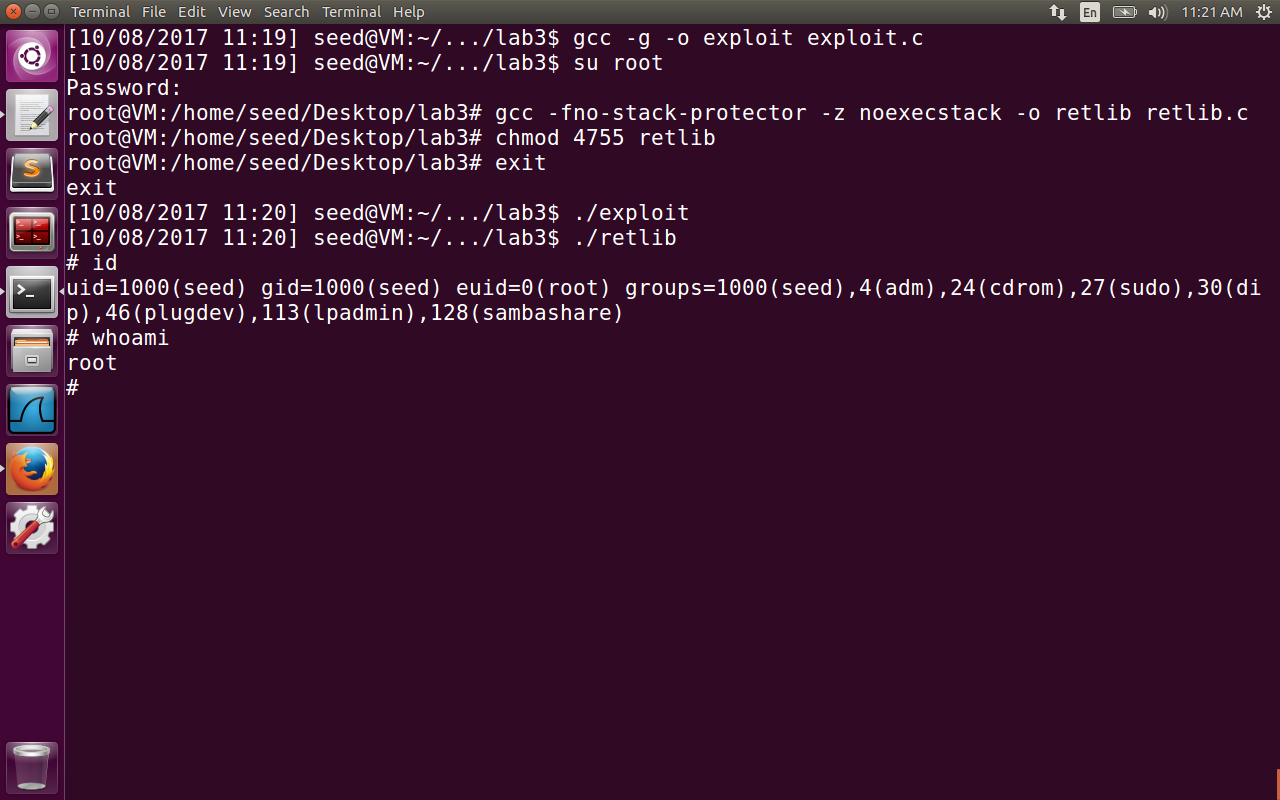
Difference = 0x14 in hexadecimal = 20 in decimal

So the return address is at offset of 24 from the ebp.

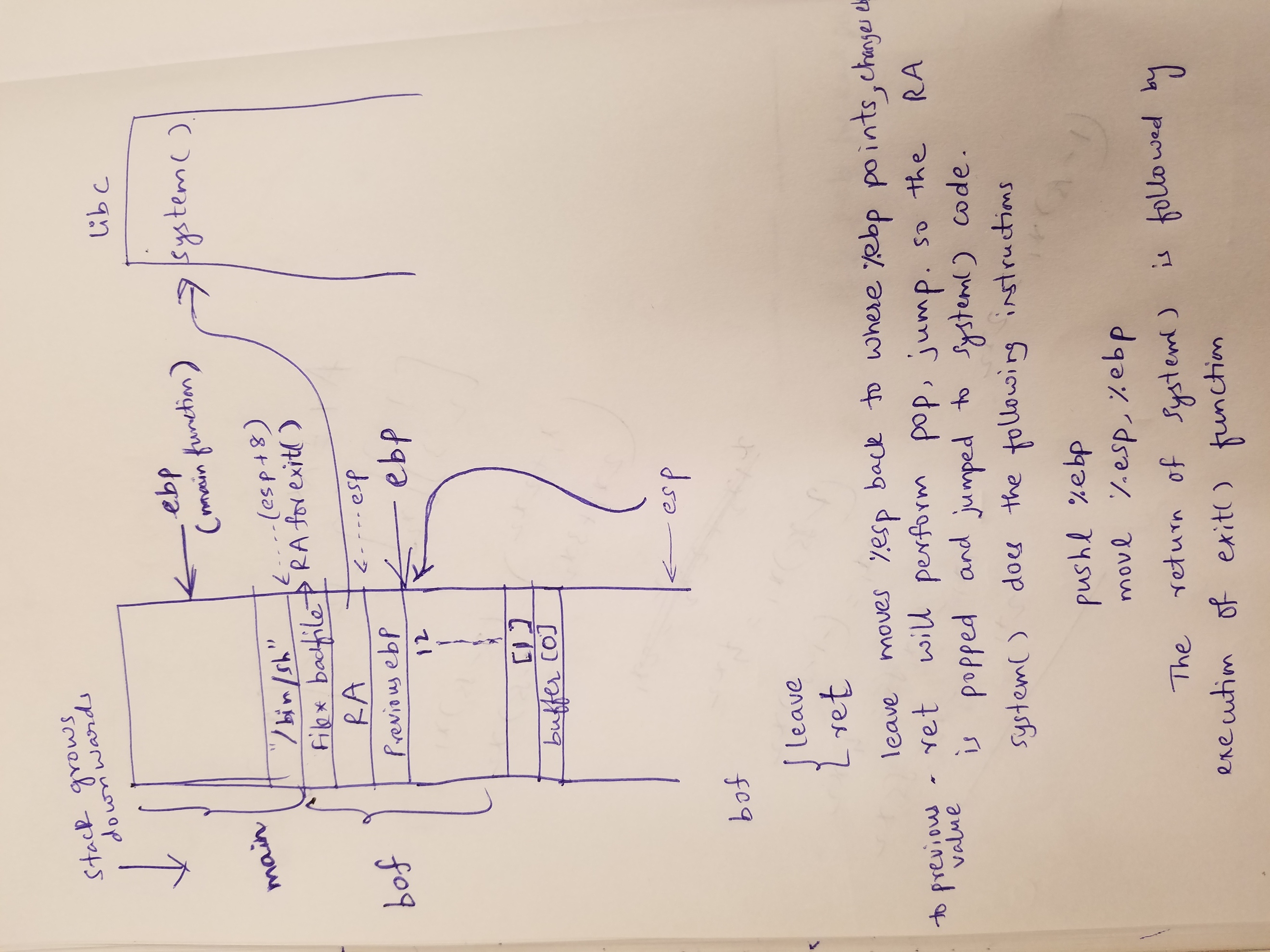
Above the return address, we try to fill the return address after the execution of system() function i.e., we insert exit() function so that the control goes to exit() for the program to exit gracefully. The next argument on the top of exit() is command arguments to the system() i.e., “/bin/sh” . So we overflow the buffer at addresses 24, 28 and 32 with system() address, exit() address and “bin/sh” to construct a new stack frame. The challenge is to find the address of /bin/sh environment variable in the following program. So we normally print the variable using printf to find the address and then comment it out after getting the value. All the values are filled in the exploit.c as follows:

Address of /bin/sh = 0xbffffe3a is found out by printing it in the MYSHELL variable in the retlib executable and comment it out before recompiling it again. The reason for doing this the MYSHELL address from the envaddr program can be different from the address MYSHELL is stored in retlib executable. We can also print the variable using envaddr executable and keep guessing the MYSHELL address in retlib by making trials of changing the address close to what we obtain from envaddr.



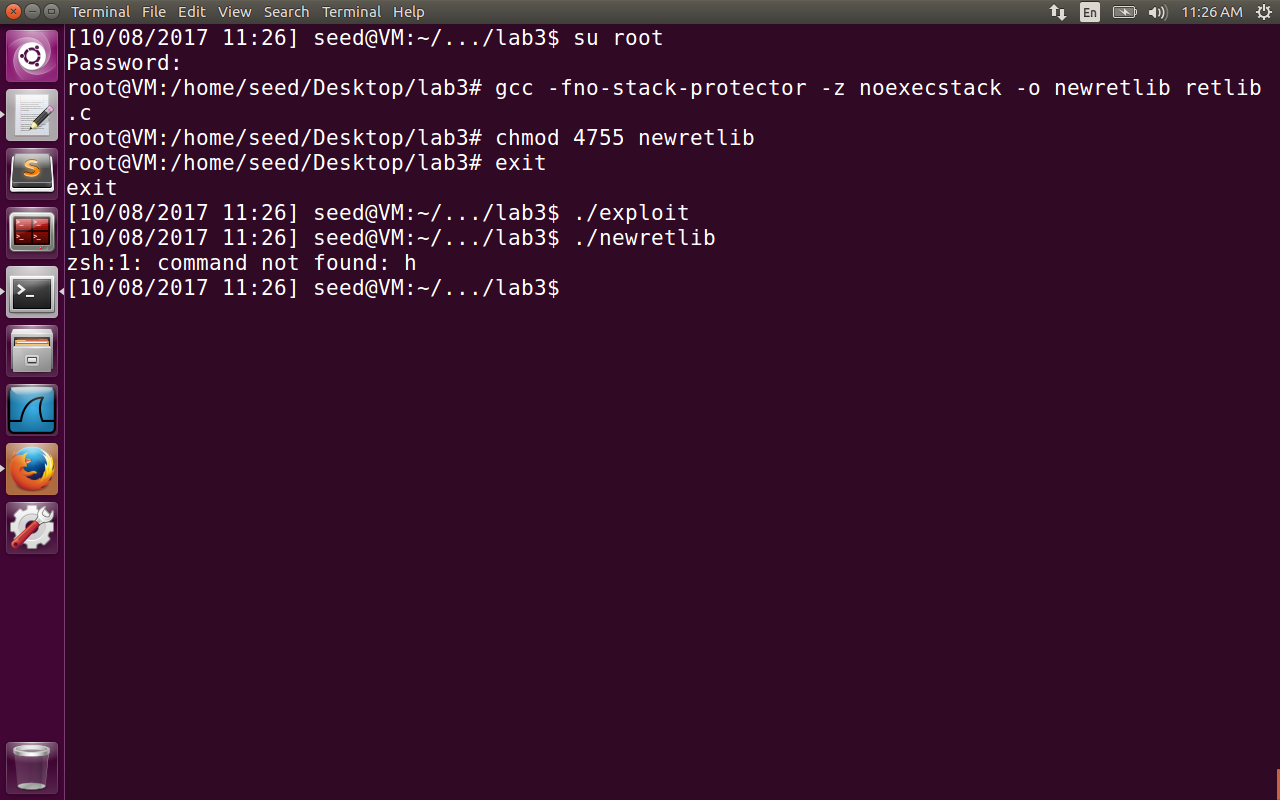


The above image contains the content of exploit.c and we got the root shell after compiling the exploit.c file. We then go to root to compile retlib.c with disabling stack protector and noexecstack options. We make the retlib a set-uid program. Then we exit from root, run both the exploit and retlib executables to get the root shell access. Please check that eid of the shell is zero.



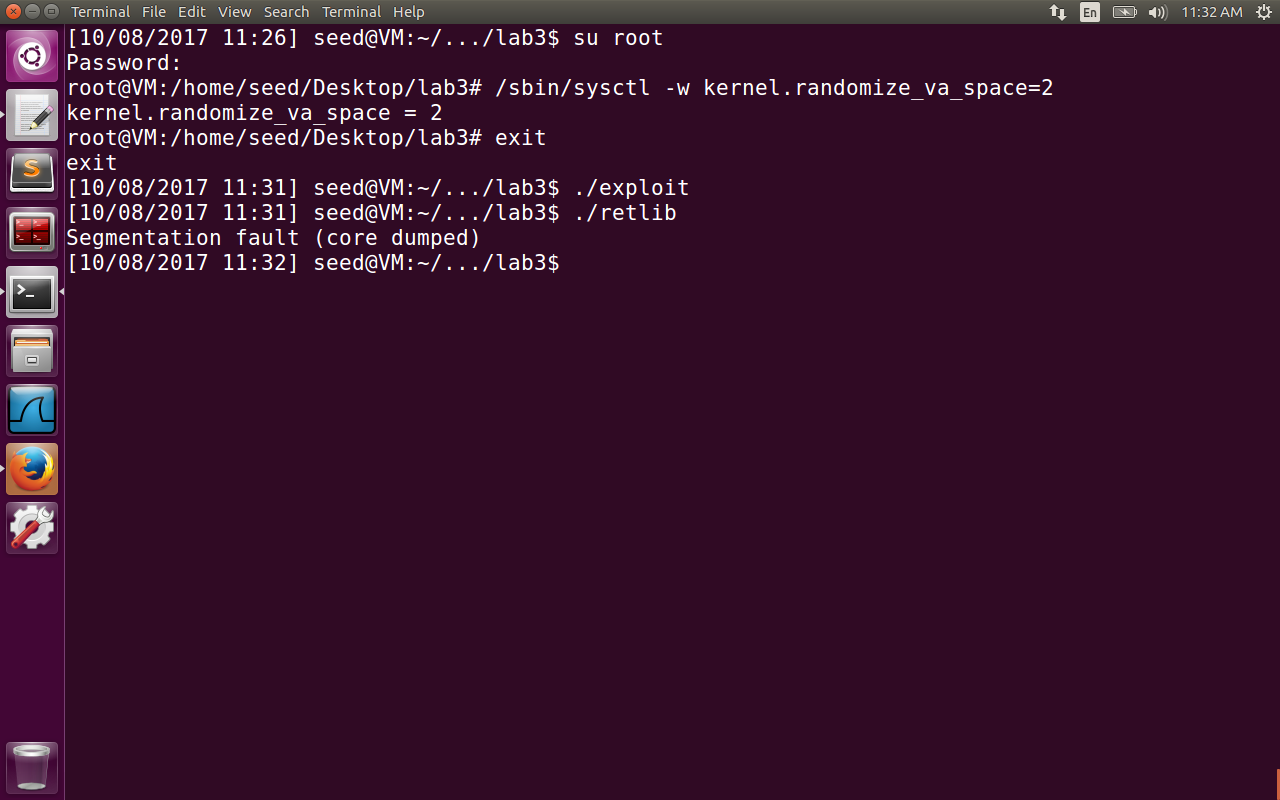
The diagram drawn manually tries to explain how the attack is done with the help of stack frames. When the bof() function is called with ‘call bof()’ (assembly instruction), first the eip (Next instruction after call bof())is pushed into stack (RA after stack frame of bof is done with execution), previous ebp is pushed to stack, the new ebp is updated with stack frame of bof(), followed by local variables of bof() function. The bof() at the end of execution calls leave and ret instructions. The leave instructions pushed esp back to ebp pointer (which is clearing the stack of bof()). The ebp is then changed to point to previous ebp. The ret instruction pops the return address which is where we fill the address of system() function of libc library. Now the system() again executes the instructions pushl %ebp, movl %esp, %ebp. The %ebp is now moved to some random place in system() function, previous ebp is stored in the stack frame of system() function. The leave and ret instructions after system() moves the esp back to RA on bof() which is where we store the address of exit() function. The arguments for system() is stored on the top of RA(where exit() address is stored). So ebp+8 is where store the argument “/bin/sh” to be used by the system() function. So the RA of system, RA for exit(),arguments are at offset at 4,8 and 12 from ebp of bof() stack frame gdb debug session.

Now the executable name of retlib is changed to newretlib and the same steps for the attack are executed. The attack is a failure, please find the screenshot below.



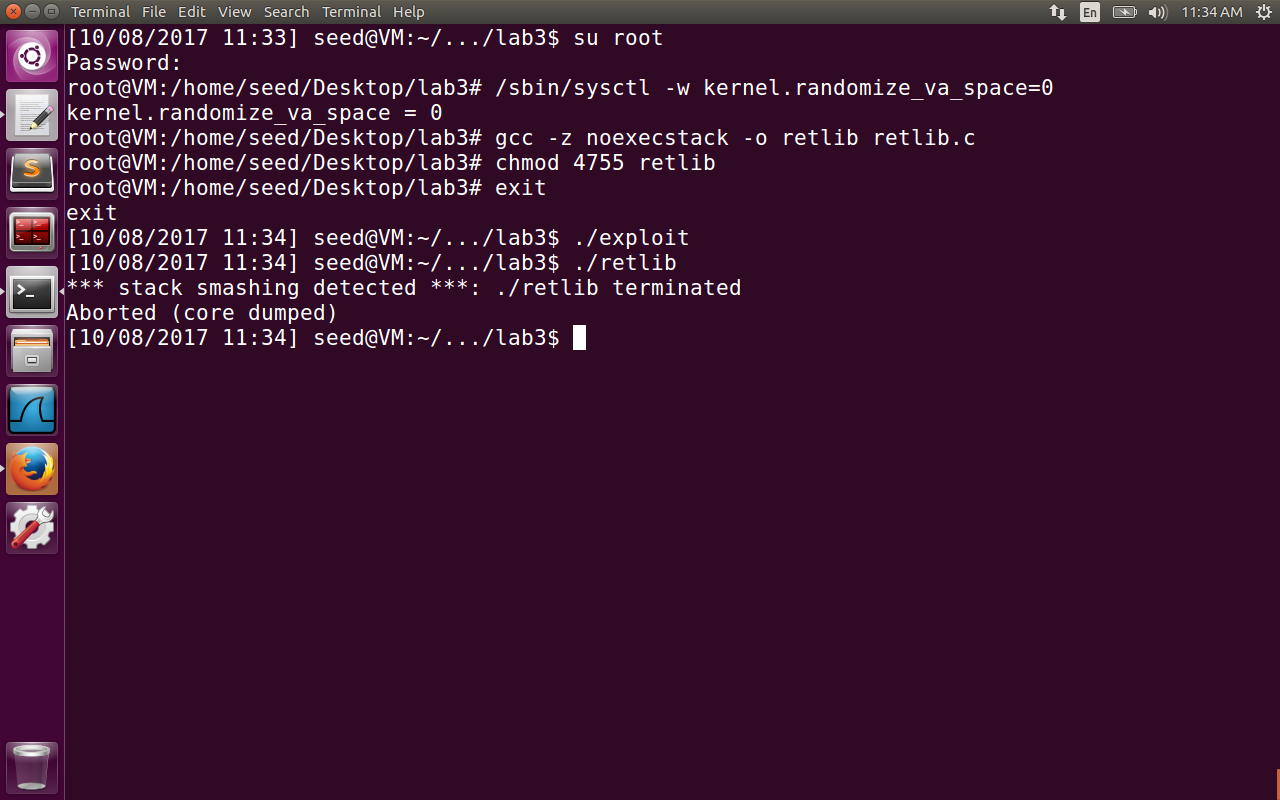
The problem is that now if we observer newretlib has 3 characters more than retlib. The address of “/bin/sh” is changed, because in the environment variable array, we also store the name of executable which shifts the address of MYSHELL variable by 3bytes. Hence the address given in the exploit file will try to execute zsh instead of /bin/zsh. Hence we get the command not found as expected.

**2.4 Task 2: Address Randomization**

****

We enable the address space randomization and perform the attack again. The segmentation fault(core dumped) error occurs. The return to libc attack needs to know the address of system() function in the address space. If memory is randomized guessing the address of system() function becomes difficult. We can run the script to continuously trying to attack, But the probability of success is small.

**2.5 Task 3: Stack Guard Protection**

****

Before enabling the stack guard, we again disable the address space randomization, now compile the program without the option –fno-stack-protector . By default for gcc version >=4.0

The stack protection is enabled by default. In this case, the random variable is inserted after arguments to the bof() and checked for value before returning from the bof() function(when ret assembly instruction is executed inside bof function). If the value of variable is observed to be changed, the stack guard smashes the stack by not allowing the stack to be overwritten with some content. Hence the return-to-libc attack won’t be successful in this case.