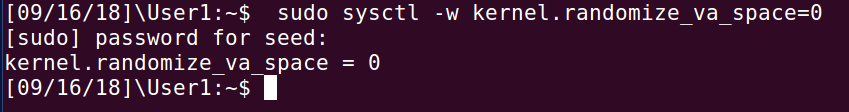
**Buffer Overflow Vulnerability Lab**

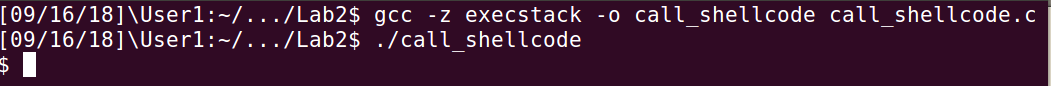
**Karan Amrutesh**

**Turning Off Countermeasures**

* Address Space Randomization



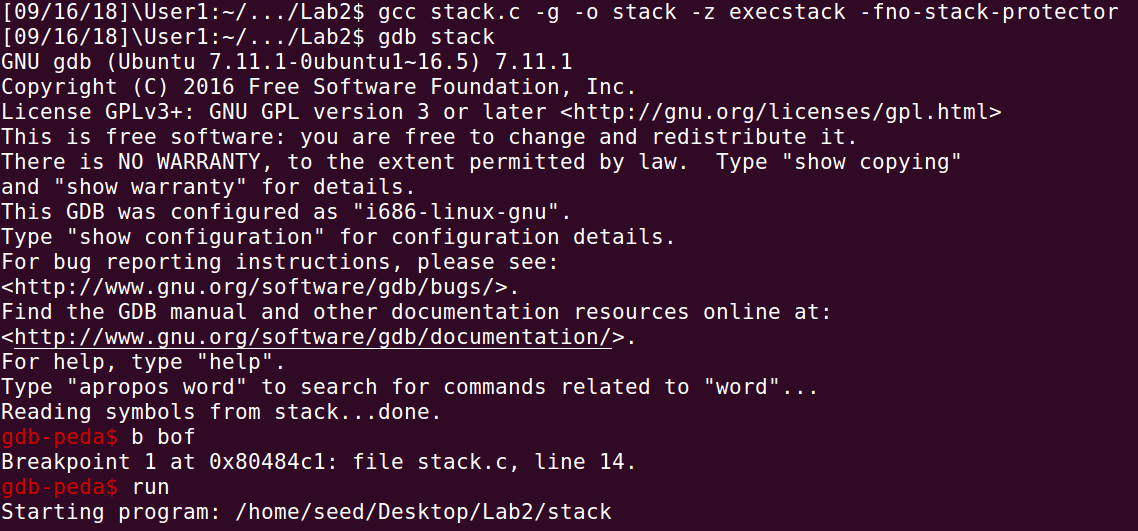
**Task 1: Running Shellcode**



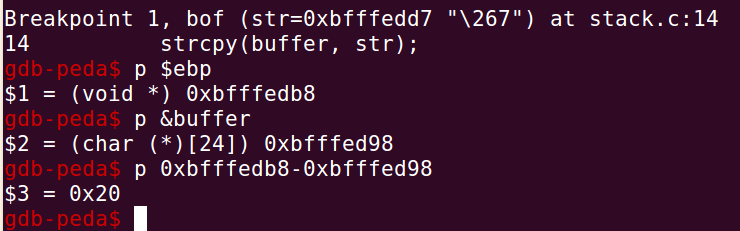
* We can see that a new shell is launched when we execute the program. Using this, we can execute any vulnerable programs.

**Task 2: Exploiting the Vulnerability**

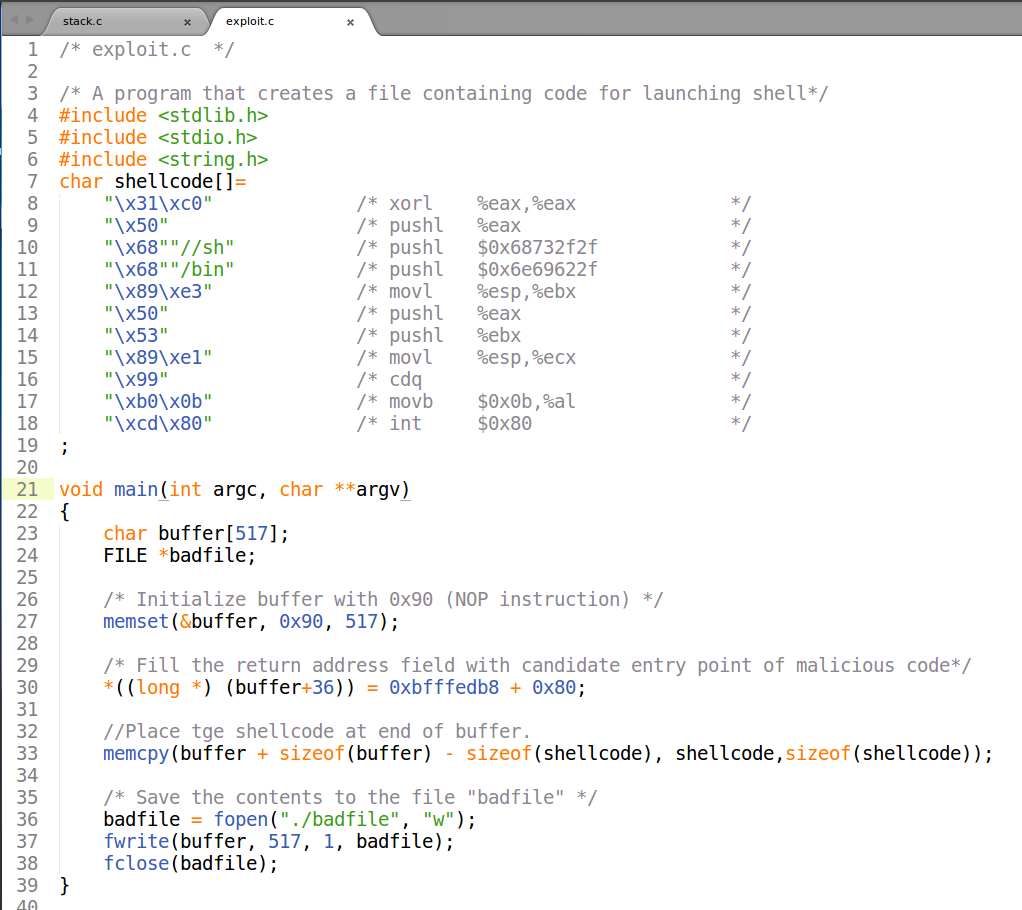
Compiling the vulnerable program and running the debugger:



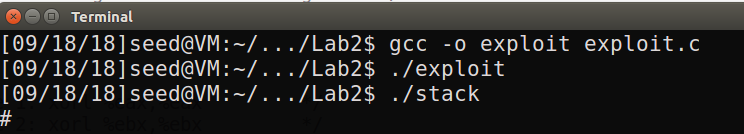
Getting the address:



* 0x20 = 32. Since the return address will be 4 bytes above where ebp points to, we get the distance, D to be 32+4= *36 (0x24)*.
* So we fill the return address to point to the entry point of the malicious code in the *exploit.c* program.
* Also, we place the shellcode towards the end of the buffer.

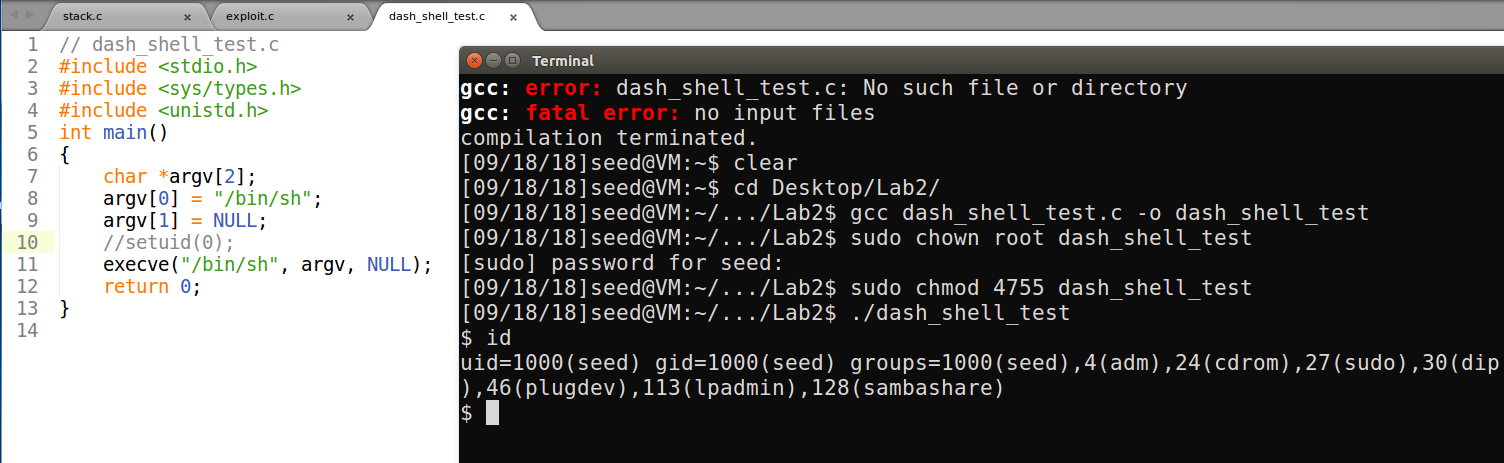


* Then we compile and run the *exploit.c* program that creates the *badfile* with the shellcode.
* Finally running the vulnerable program will give us the root shell (as *stack.c* is a set-UID program) due to copying excess data from the *badfile* which causes buffer overflow.
* This is because the return address section of the function *main* will be overwritten with the return address (that points to the malicious code) in the *badfile*.

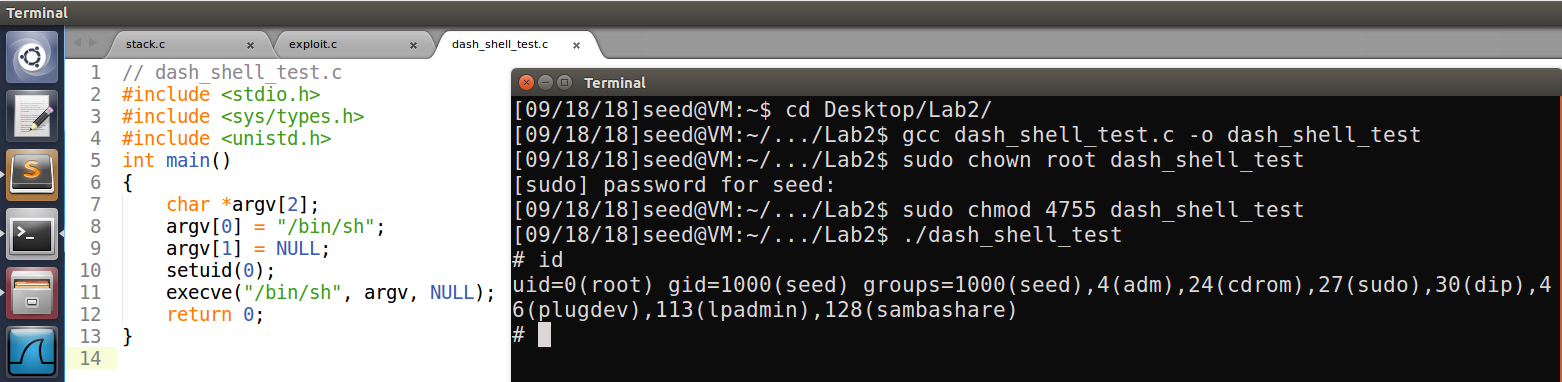


**Task 3: Defeating dash’s Countermeasure**

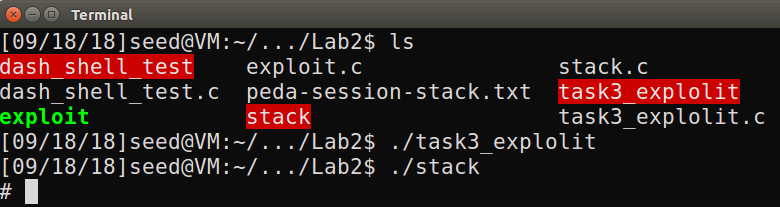
* Since the setUID program is trying to invoke the shell with root privileges, the dash shell drops privileges when it detects that the effective UID does not equal to the real UID. Hence, it opens the normal shell with UID equal to that of seed.



* To overcome this countermeasure, we change the real user ID of the victim process to zero before invoking the dash program. Here, the setUID program successfully invokes the root shell:

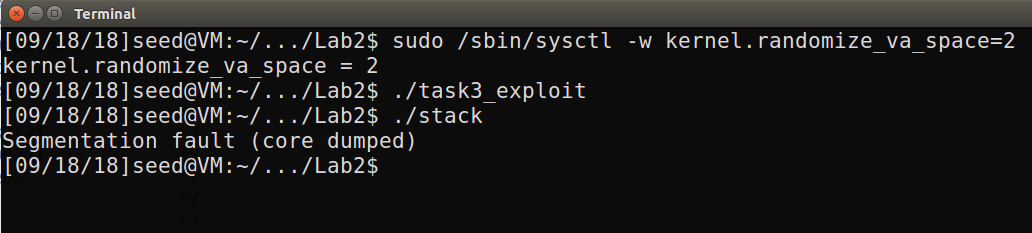


* Modifying the *exploit.c* program in task 2 to include the *setuid(0)* system call in the shellcode and executing the same:
* We can see that the root shell is invoked.

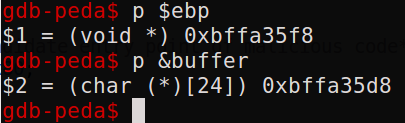


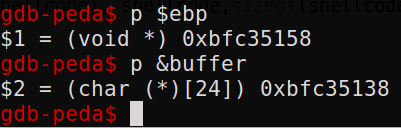
**Task 4: Defeating Address Randomization**

* Turning on address randomization will cause our program to crash:

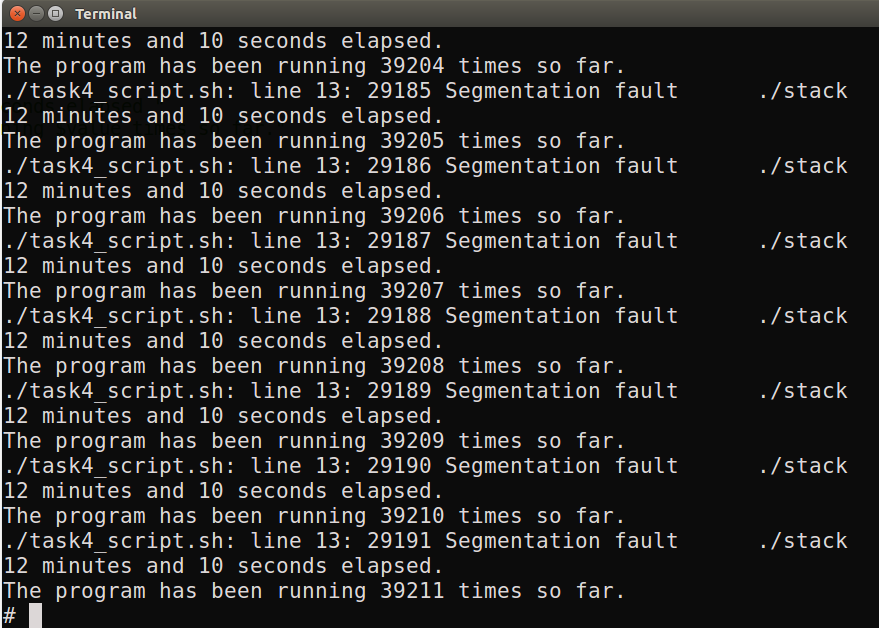


* This is because the return addresses we filled up in the *badfile* using *exploit.c* may not point to the correct address where the buffer is located.
* We can see that each time we run the stack program, the addresses allocated for the frame pointer and the buffer vary due to randomization.



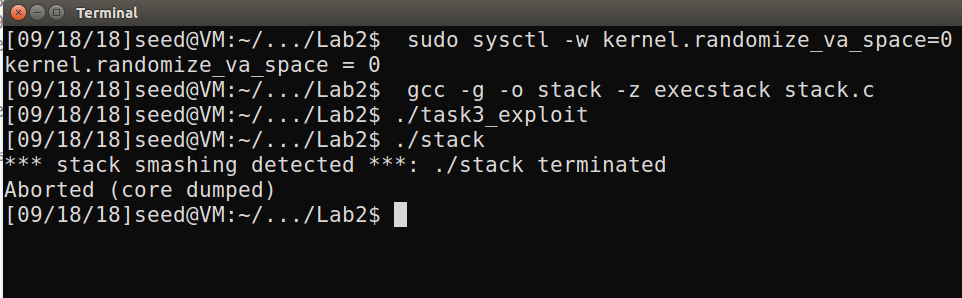


* Running the given shell code that uses the brute-force approach to attack the vulnerable program repeatedly:
* We can see that it ran for about 12 minutes until the address in the exploit program matched with the address allocated for the stack frame of the vulnerable program.



**Task 5: Turn on the StackGuard Protection**

* By not turning off the StackGuard option, we can see that the attack is not successful. This is because the gcc compilers have a stack guard that detects the stack based buffer overflows.



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

* By compiling with the *noexecstack* option, we make the stack non-executable. This means that the malicious code injected onto the stack using the *badfile*, does not get executed and hence we do not get the shell.
* This makes it difficult to put the code in the stack and execute it using the stack based buffer overflow.

