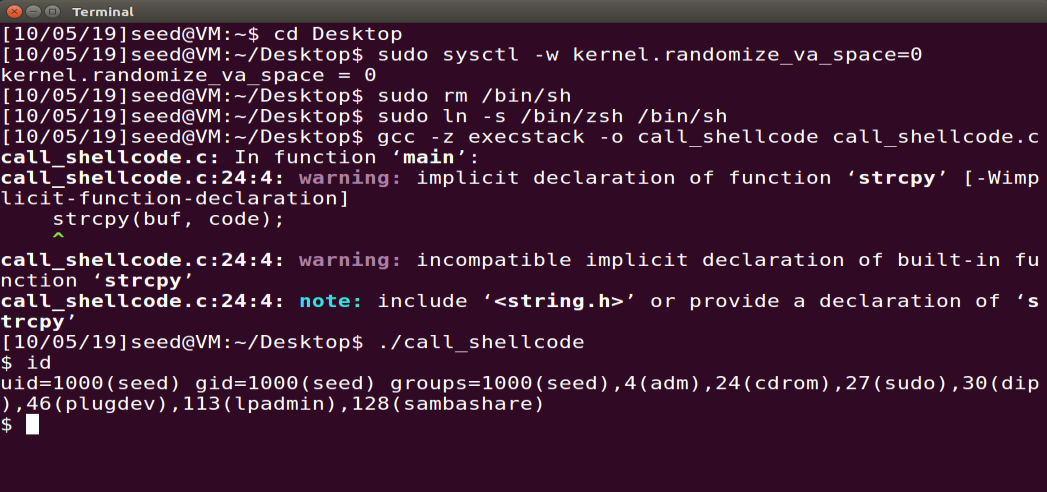
**Task 1 : Running Shellcode**

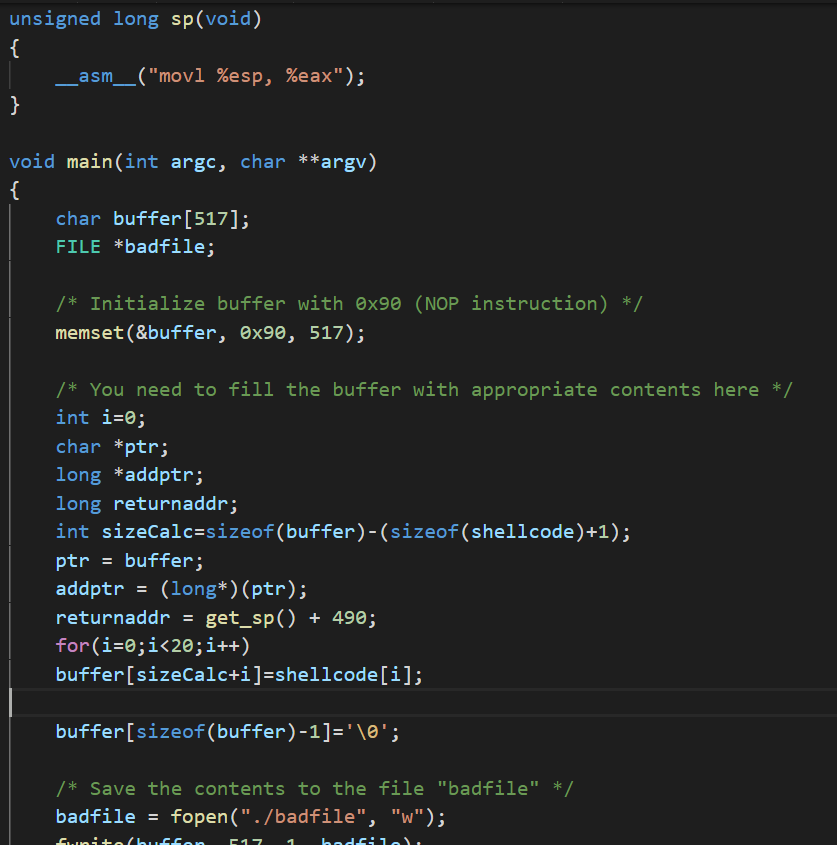
* Firstly I had to run “sudo sysctl -w kernel.randomize\_va\_space=0” to disable address randomization.
* Then I ran “sudo rm /bin/sh” and “sudo ln -s /bin/zsh /bin/sh” to change my bin/sh to bin/zsh.
* I compiled the call\_shellcode.c i.e “gcc -z execstack -o call\_shellcode call\_shellcode.c”
* I executed ./call\_shellcode and gained access to root as shown below.



**Task 2: Exploiting the Vulnerability**

*Observation on successful access of root shell:*

* I had to turn off address randomization, make the stack executable and also disable the stack guard protection.
* Compile the exploit program and construct the badfile.
* After making the changes to exploit.c. I compiled using “gcc -o exploit exploit.c” and ran “./exploit” which creates the the badfile and then ran “./stack”.
* Execute the stack program, the output is shell prompt indicating that we have exploited the buffer overflow mechanism and /bin/sh shell code has been executed.
* My exploit.c is modified as follows :

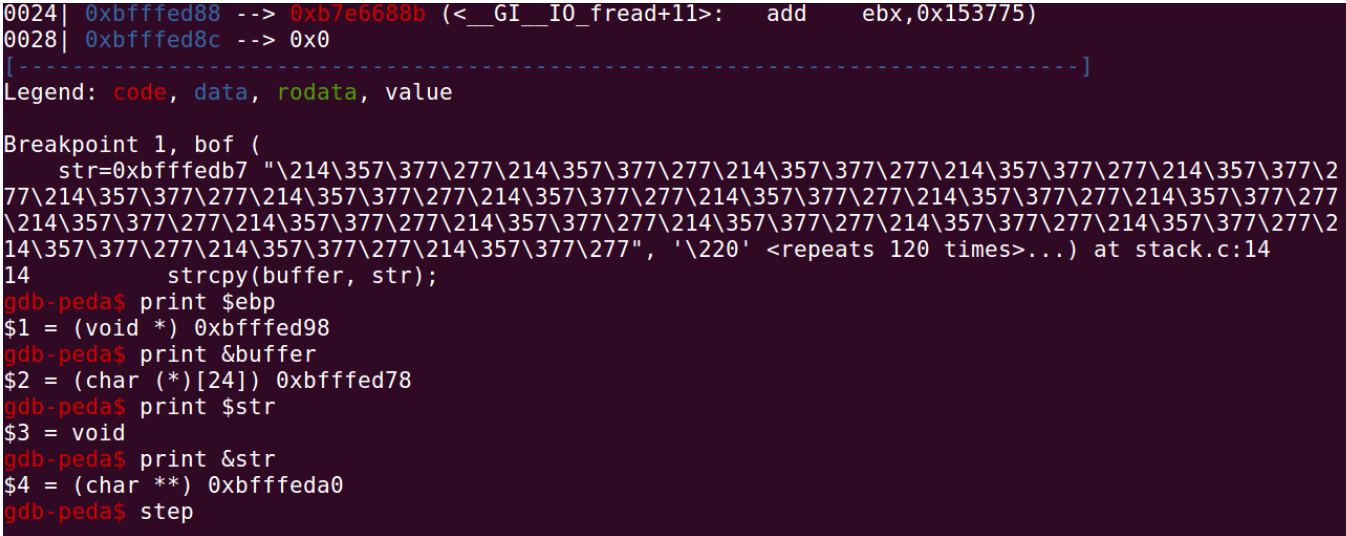


*Explanation for my code :*

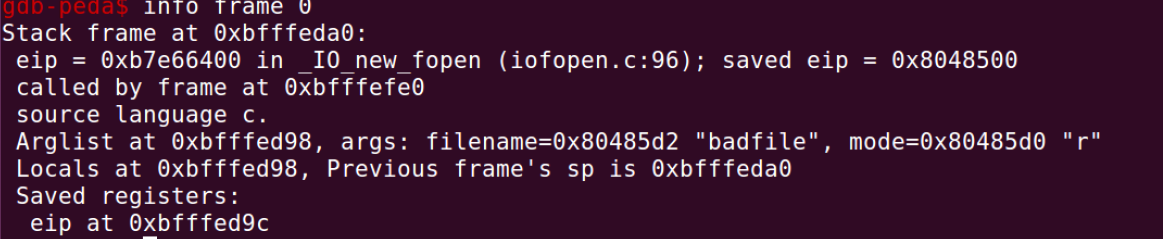
* My exploit code writes the buffer and overflows it with NOP which allows for the execution of the next line of command.
* My exploit code allows for the overflow of the stack and also points the code to execute malicious code.

*How did I do it:*

* I used gdb debugger to find return address.
* Inserted a breakpoint at the start of the function where buffer overflow may occur.
* Printed the address of the start of the buffer.
* Printed the value of the ebp register.
* Calculated where the return address is, so that I can change the return address and exploit the vulnerability.



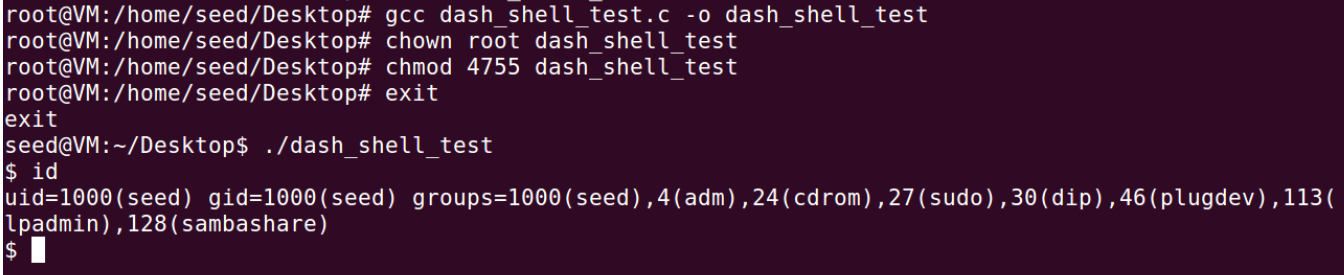
* Later, to confirm where the return address is, I looked at eip which points to previous frame pointer and the value at eip register. Both have the same value.
* This value must be overridden so that buffer overflow can be exploited and the program can be executed.



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

*How did I do it:*

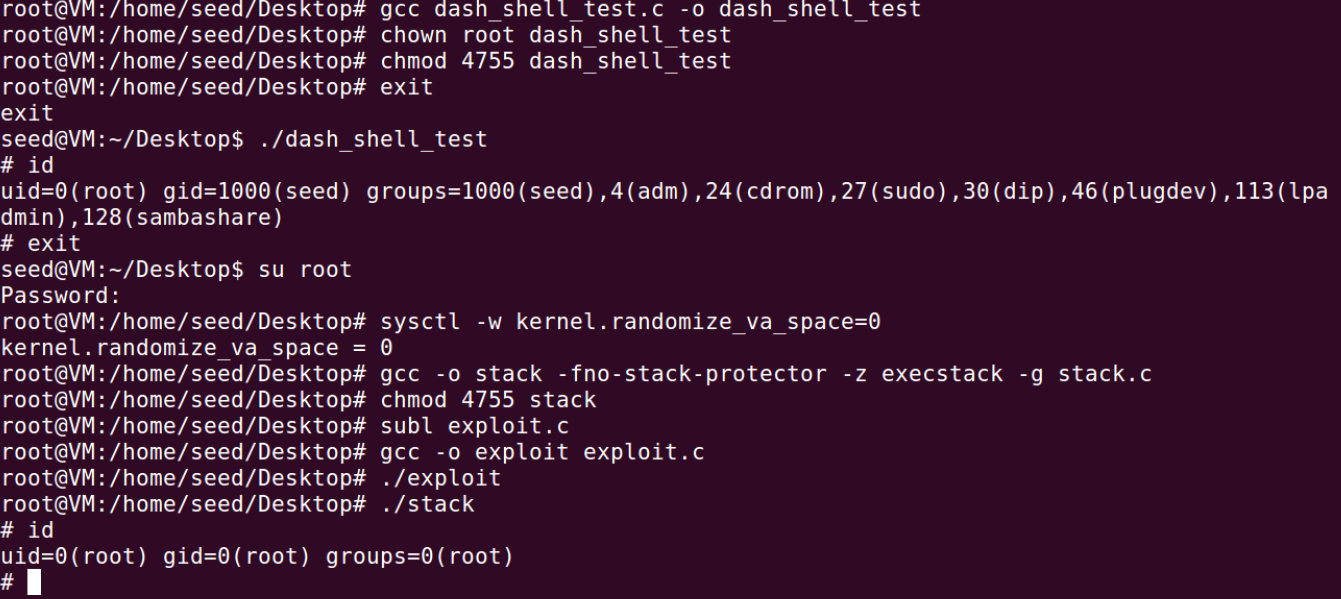
* First change the /bin/sh , so it points back to /bin/dash using “sudo rm /bin/sh” and “sudo ln -s /bin/dash /bin/sh”.
* I compiled the program dash\_shell\_test.c without uncommenting the setuid(0) statement.
* I observed that uid is not zero yet as that of a root user see the image below



* I compiled the program by uncommenting the setuid(0) statement to find that the uid is that of a root user now.
* I performed the Task 2 again and find out that I got access to the root shell in the process.

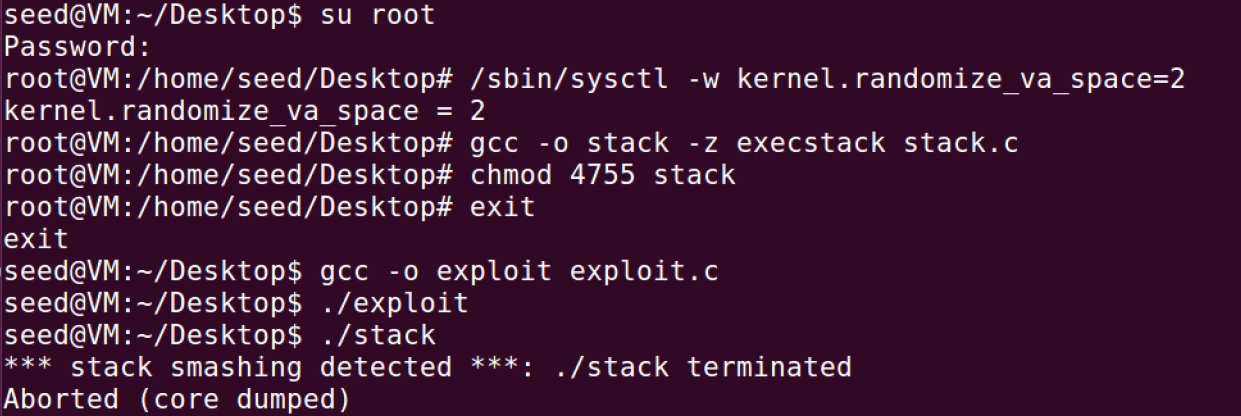
*Observation:*

* I observed that the uid is now zero.
* So can conclude that performing the attack on the vulnerable program when /bin/sh is linked to /bin/dash with the effect of setuid statement gives us access to root shell with both real and effective uid as that of a root user.



**Task 4: Defeating Address Randomization**

*How did I do it :*

* I set the address randomization to 2 using “sudo /sbin/sysctl -w kernel.randomize\_va\_space=2”
* I compiled and executed the exploit program which creates the bad file.
* I compiled the stack program with no stack guard protection and making the stack an executable stack.
* I made the stack program a set UID program owned by root. Run the stack till the buffer overflow is successful and the # prompt is returned.   
    
  

*Explanation:*

* I used the shell script provided to run the stack in an infinite loop. It approximately ran for around 8-9 minutes.
* Upon repeated execution using the while loop in the script, the attack becomes successful and I was able to gain root access.

**References :**

1. The URL below helped me a lot in understanding how the buffer overflow attack can be done: <https://www.cs.cornell.edu/courses/cs513/2007fa/paper.alpeh1.stacksmashing.html>