Lab 5

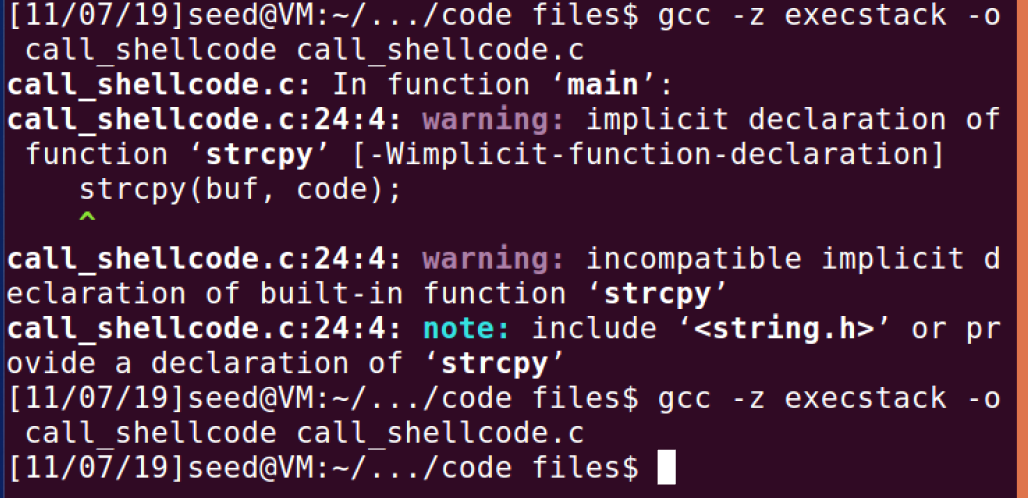
#### Cmpsc 443

### -

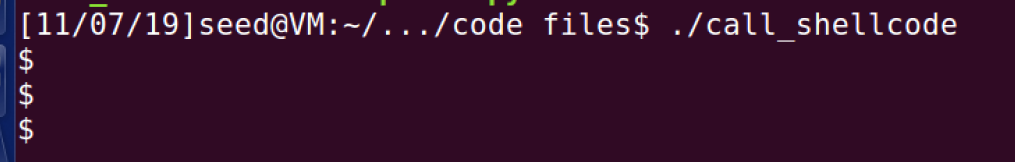
# Task 1: **Running Shellcode**

* I added the next line to the shellcode file, because it was missing.

#include <string.h>

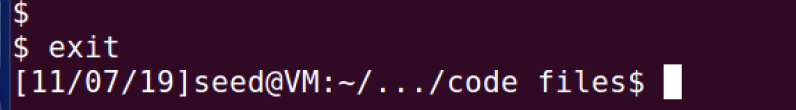


We can see that the program launched a new shell

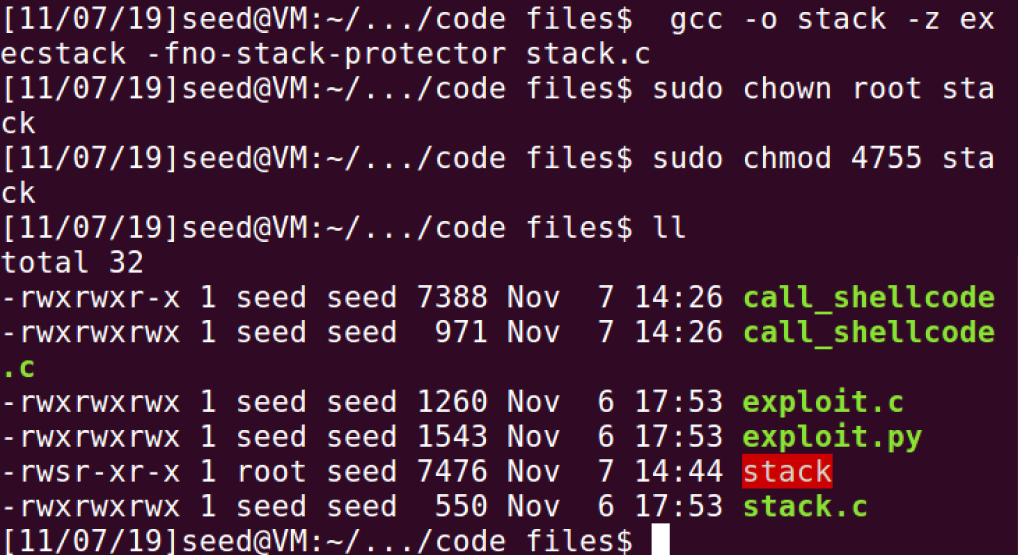


## The Vulnerable Program

We will exit the new shell



Compile the program and set the ownership and enable the permissions



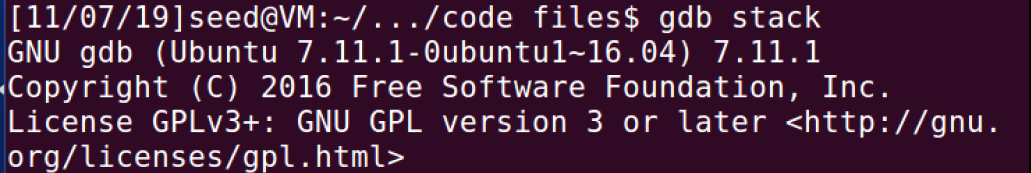
# Task 2: Exploiting the Vulnerability

We will write our shellcode a random place, which is in the boundaries of the stack’s frame.

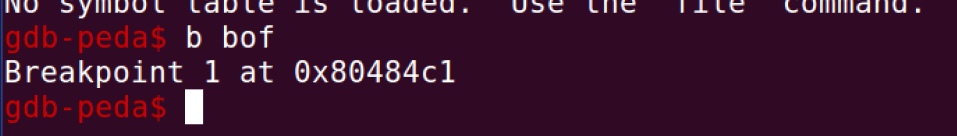
**exploit.c:**



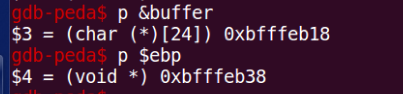
We will use gdb command to use the linux debugger while running ./stack



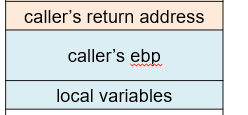
Putting breakpoint at bof()



Inside of bof() we will check the addresses of buffer – the start of our local variables frame, and of ebp register – the base of our stack.



0xbfffeb18 – 0xbfffeb38 = 0x00000020

Which is 32 in decimal.

The return address is saved in ebp-4 🡪

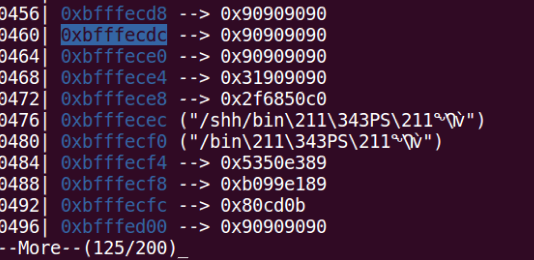
**Therefore, the needed address is buffer+36.**

**exploit.c:**



We will use random address which is between the start of our function and the end of the buffer

(because the instructions in those addresses are Nops, which means that the pc will proceed to the next address again and again until it reaches the start of the code, we “injected” in the frame)



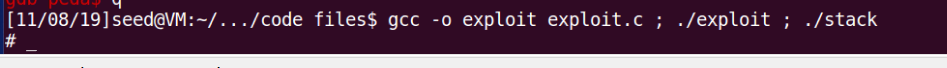
We will put the marked address in buffer + 36



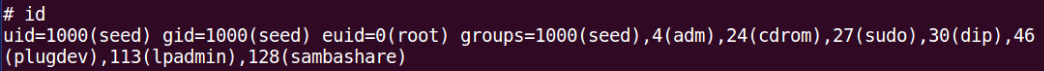
Our final exploit.c code:



And as we can see, running stack will grant us root permission

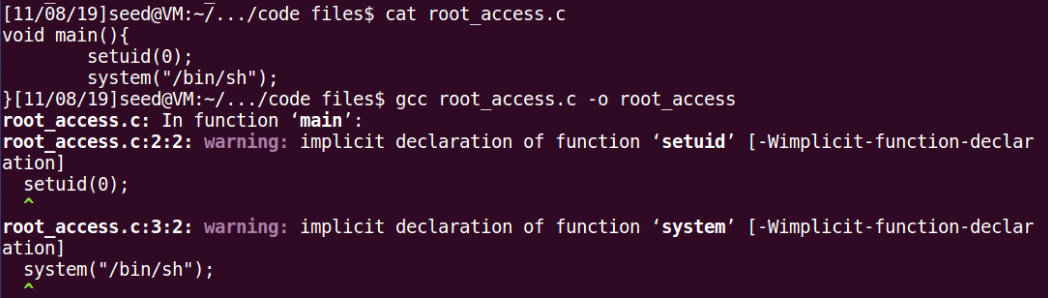


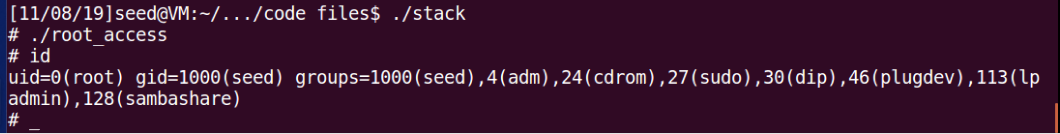
Checking our user’s id



uid=1000(seed) and euid=0(root)

we will fix this



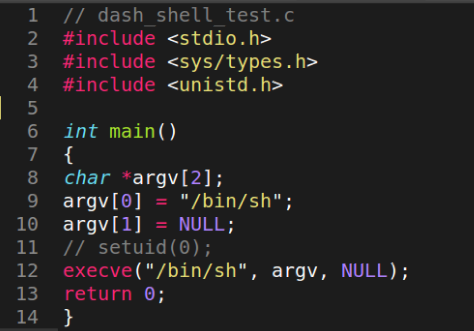


# Task 3: Defeating dash’s Countermeasure

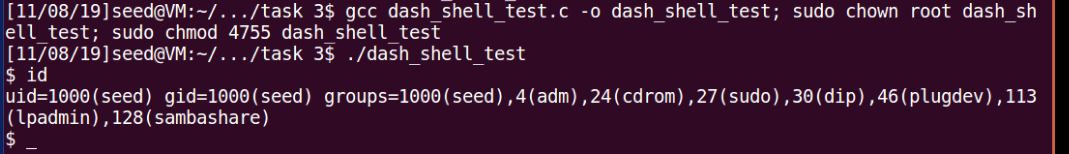
Configuring back /bin/sh



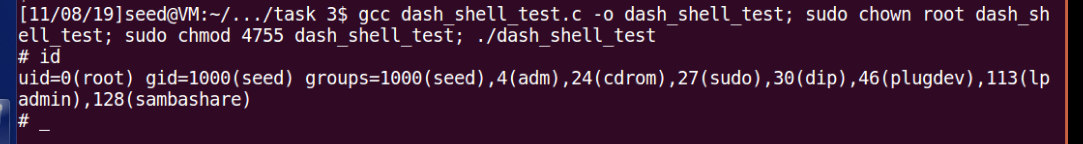
We will create the dash\_shell\_test.c file



Compile and run it with setuid() commented

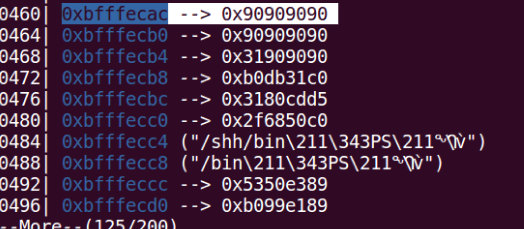


Now we will compile and run dash\_shell\_test.c with setuid() not commented



As we can see the uid in the second run shows us as root. In addition, we can see that the $ sign shows that we are not root (PS2) and that # sign shows that we are root (PS1)

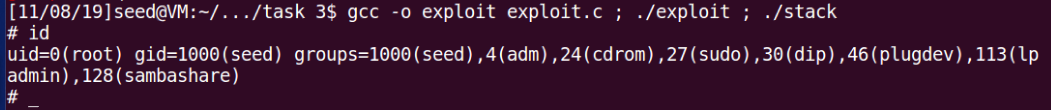
For running the program from task 2 again, we will add the needed code and check what is the address that we will want to put in buffer+36



We will pick 0xbfffecac and write it at exploit.c



Running a modified copy of the program from task 2



Which is the same as the one we got in task 2 after we ran the second program that included:



# Task 4: Defeating Address Randomization

Turn on the Ubuntu’s address randomization



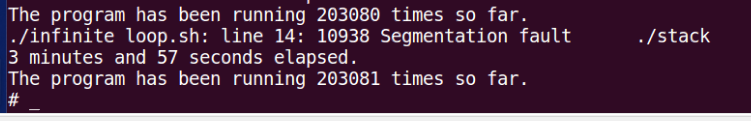
Running the code from task 2 after enabling the address randomization



As we can see, after enabling the ASLR our program does not work.  
why is that?  
When the ASLR is enabled, the frames of the stack are not organized as we assumed in task 2.

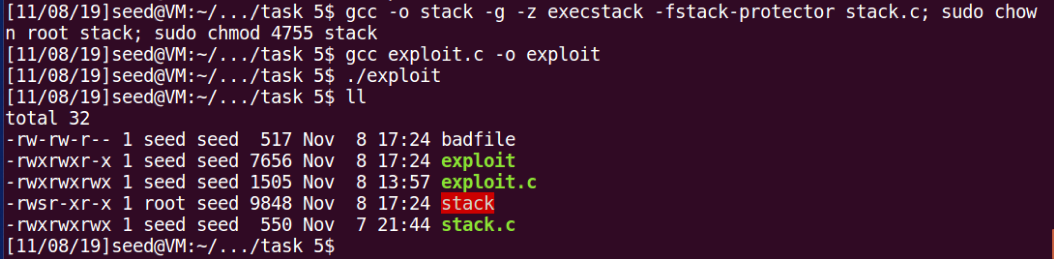
Therefore, the callee’s return address is not necessarily located at buffer+36 as we found in task 2.

Running the infinite loop script:

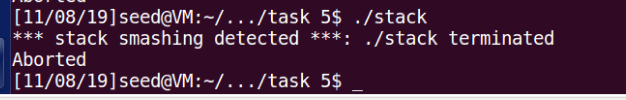


# Task 5: Turn on the StackGuard Protection

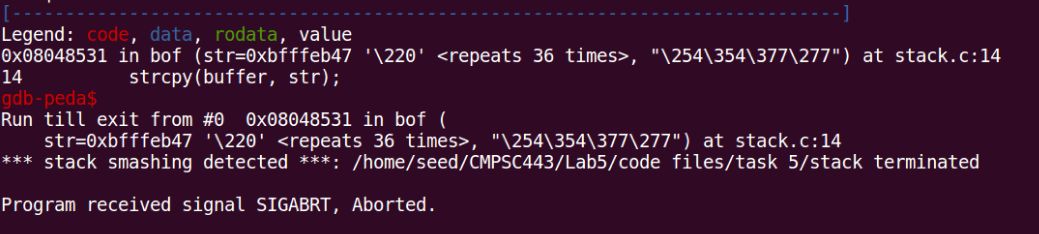
Compile exploit.c and stack.c again



When trying to run our program we get an error message:



When inside gdb we can see the line that gave us the abort signal



At the line pointed by our purple arrow, we can see that the function strcpy repeated its action 35 times without getting an abort signal.

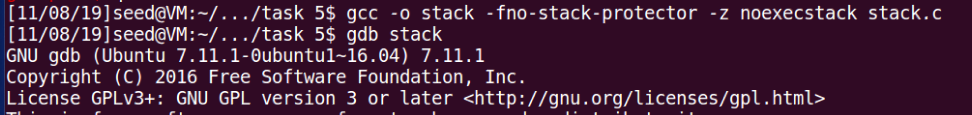
In the 36th time we have received an abort signal.

And as we know from previous tasks, the start of our caller’s frame at buffer + 36.

Which is the exact address that writing to it sent us an abort signal.

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

Compile and run stack with `noexecstack` flag



As we can see, we get SIGSEGV signal when we try to execute an instruction within our stack frame.  
0xbfffecac is the address we put in the caller’s return address (at buffer+36).  
(the gray shade of the upper nops shows that they were not executed)

