LAB- 04

By :- Faraz Ahmed

Shellcode

Task 1

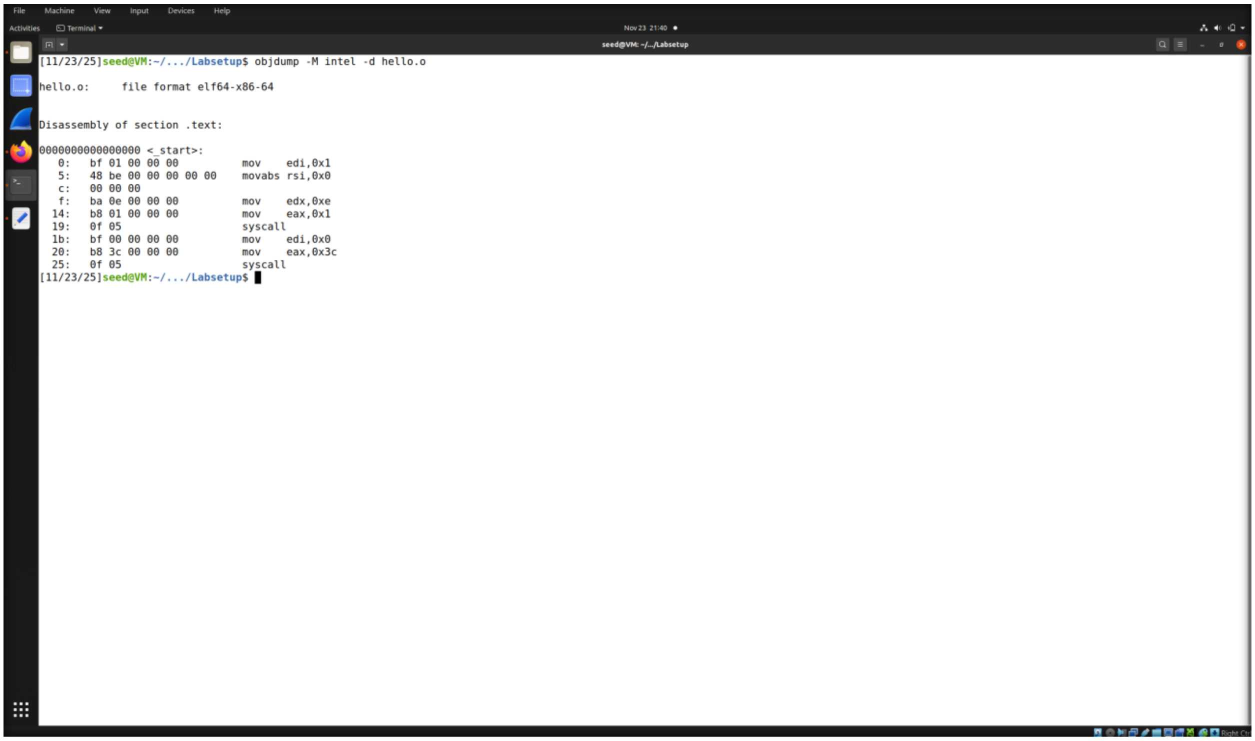
- First we use command “nasm -f elf64 hello.s -o hello.o” (highlighted in Screenshot 1) which will help us to convert the assembly code into an object file and then use “ld hello.o -o hello” (also highlighted in Screenshot 1) which will creates it into a final executable program.
- Then we use command “./hello” to run that executable file.

<u>Screenshot 1: Running the sample code using “./hello”.</u>

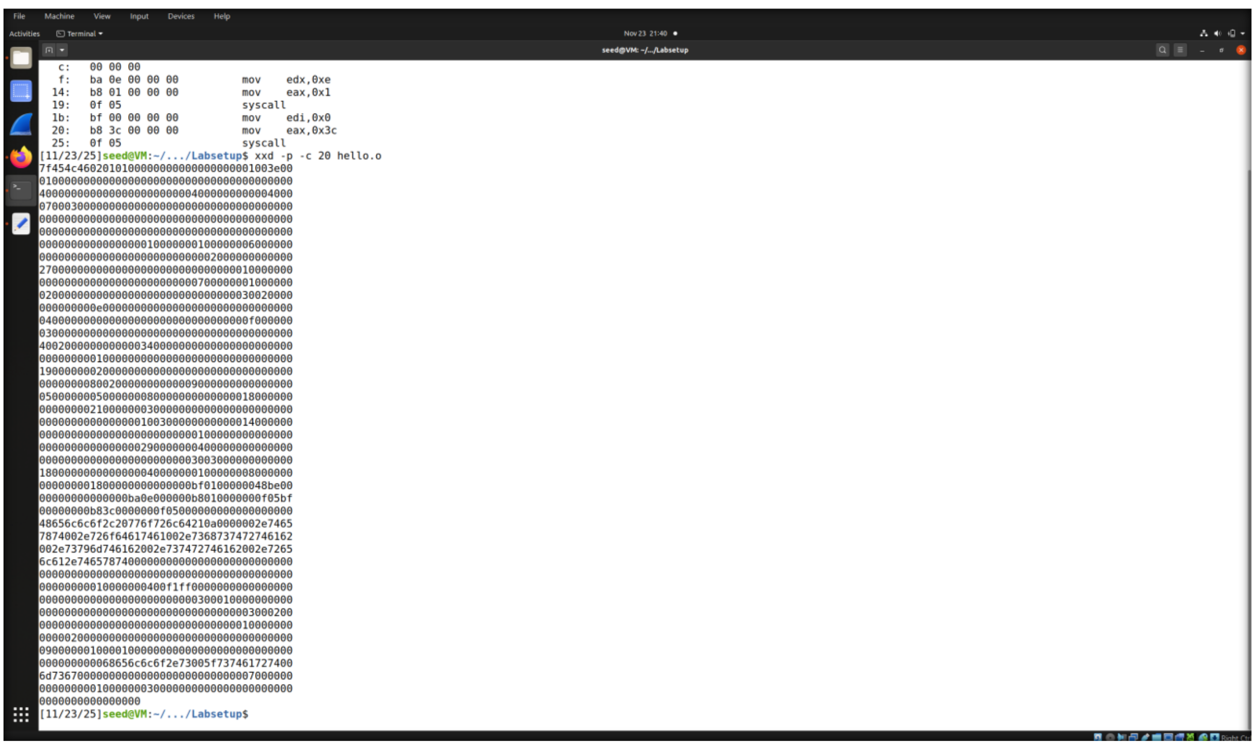
- We extract the machine code bytes, we used two commands:-

I. objdump -M intel -d hello.o which is as highlighted in Screenshot 2.

II. xxd -p -c 20 hello.o which is also highlighted in Screenshot 3.



Screenshot 2: Displaying machine code in binary using objdump.



Screenshot 3: Displaying machine code in binary using xxd.

Task 2.a

- Now we will perform the same commands which we used in task 1 to run “mysh64” as observed in Screenshot 4.

<u>**Screenshot 4: Running sample code successfully using “./mysh64”.**</u>

- So, overall GDB session can prove three things (refer all these commands in Screenshot 5 as highlighted) :-

I. “Call” pushed the address- using x/gx $rsp which 0x7fffffffedc8: 0x00000000004000a8 which means that the address is on stack.

II. “Pop rbx” moved it to rbx- using stepi, we execute pop rbx which then after executing- print $rbx, we get that same address which is now in rbx.

III. Now that address points us to “/bin/sh”- using x/s $rbx which gives us 0x4000a8: "/bin/sh" which gives us the string.

<u>Screenshot 5: Using gdb to show that the address of string /bin/sh is pushed.</u>

- If we look at the code of "mysh64.s", we can observe argv[] array by:-

I. Pop rbx- rbx helps to point "/bin/sh"
II. Mov [rbx+8], rbx- this is first line which sets argv[0]
III. Mov rax, 0x00
IV. Mov [rbx+16}, rax- this is second line which sets argv[1]

So, every address has its content saved like:-

I. Rbx+0 has "/bin/sh\0"
II. Rbx+8 has "AAAAAAAA"
III. Rbx+16 has "BBBBBBBB"

Now, mov [rbx+8], rbx which stores the address of "/bin/sh" into the memory at "rbx+8" with the help of argv[0].

And mov [rbx+16], rax (where rax= 0) which stores 0 (NULL) into the memory at "rbx+16" with the help of argv[1].

Therefore, argv[0] has address of "/bin/sh" (stored ar rbx+8) and argv[1] has 0 (stored at rbx+16).

- For the execve arguments,

I. Mov rdi, rbx which is 1^{st} argument as rdi has path to executable of rbx.
II. Lea rsi, [rbx+8] which is 2^{nd} argument as argv array of rbx+8.

Hence, Mov rdi, rbx sets 1^{st} argument to the command path and Lea rsi, [rbx+8] sets 2^{nd} argument to pointer of the argv[] array.

Task 2.b

- Here are some observed null bytes from the output of objdump as observed in Screenshot 6. (all zero bytes are highlighted in Screenshot 6)

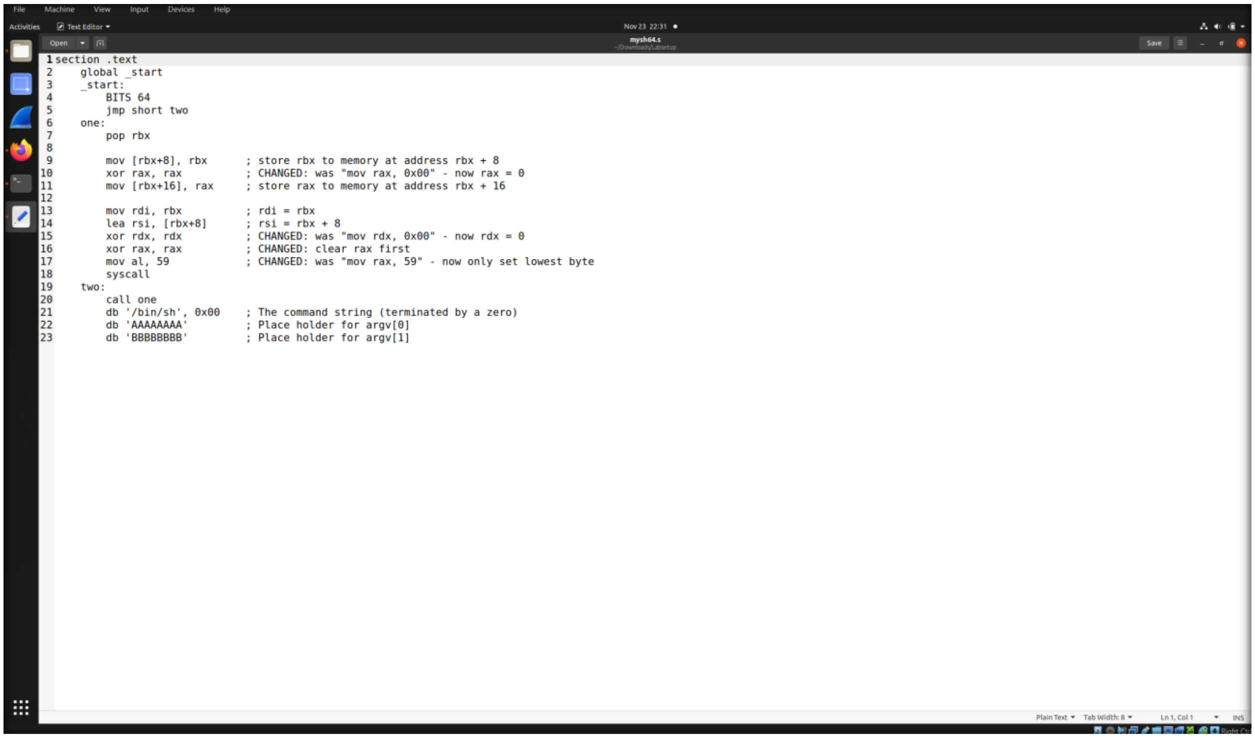
I. Line 7: b8 00 00 00 00 mov eax, 0x0
II. Line 17: ba 00 00 00 00 mov edx, 0x0
III. Line 1c: b8 3b 00 00 00 mov eax, 0x3b

<u>**Screenshot 6: Using objdump to check for null bytes in mysh64.**</u>

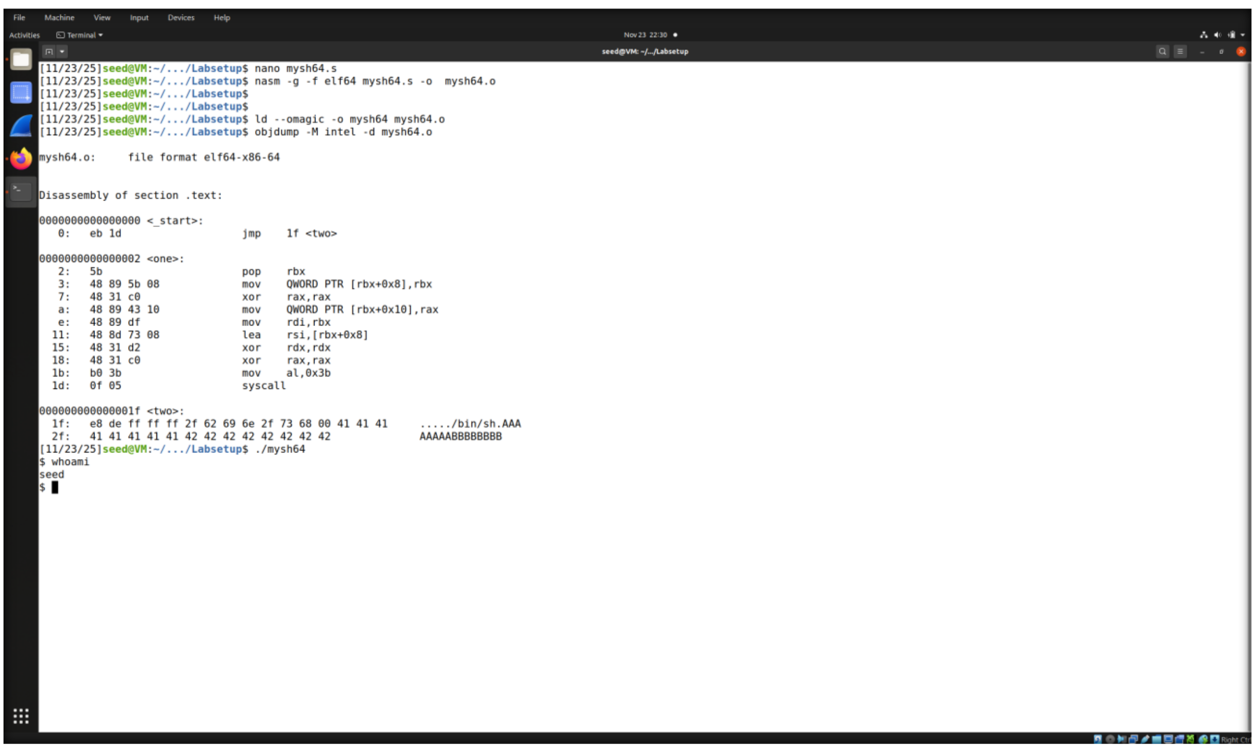
- To avoid every zero bytes, the fixes are:-

I. Fix for line 7- mov eax, 0x0 and replace it with xor rax, rax (with 48 31 c0)
II. Fix for line 17- mov edx, 0x0 and replace it with xor rdx, rdx (with 48 31 d2)
III. Fix for line 1c- mov eax, 0x3b and replace it with mov al, 59 (with b0 3b)

- So, we can refer to the code without any null bytes in Screenshot 7 and after using "./mysh64" which will run the script without 00 and give us the a shell as we can observed in Screenshot 8.



<u>Screenshot 7: Updated code for mysh64 with no null bytes.</u>

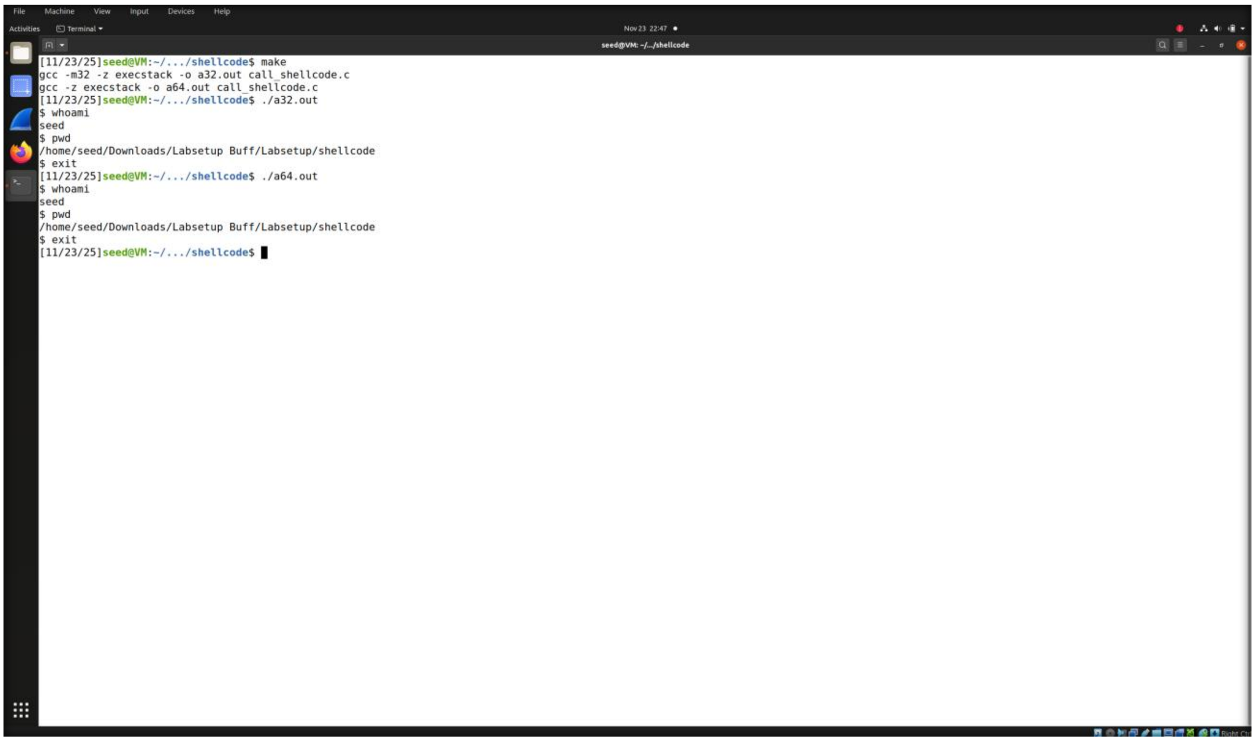


<u>Screenshot 8: Implementing updated code and checking for root shell.</u>

BufferOverflow

Task 1

- So, first we use make (as highlighted in Screenshot 9) command which will create two files- a32.out (32-bit) and a64.out (64-bit). Then we use ./a32out and ./a64.out to run both's shell prompt where if we type "whoami"- we can see seed as a reply as observed in Screenshot 9.



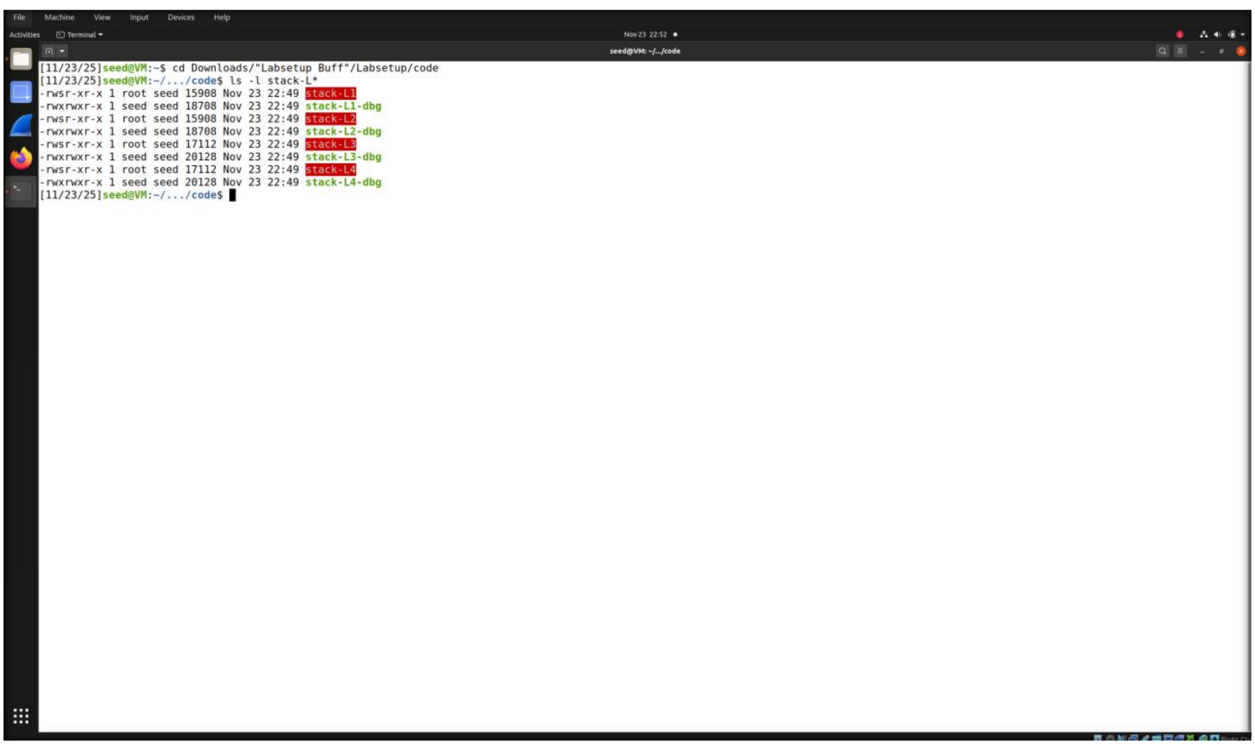
Screenshot 9: Running both call_shellcode.c for both the 32-bit and 64-bit version.

Task 2

- Now we do the same for code folder by using command "make" again (highlighted below) which compiles stack.c into multiple versions: stack-L1, stack-L2, stack-L3, stack-L4 as observed in Screenshot 10.

Screenshot 10: Compiling stack.c into multiple versions.

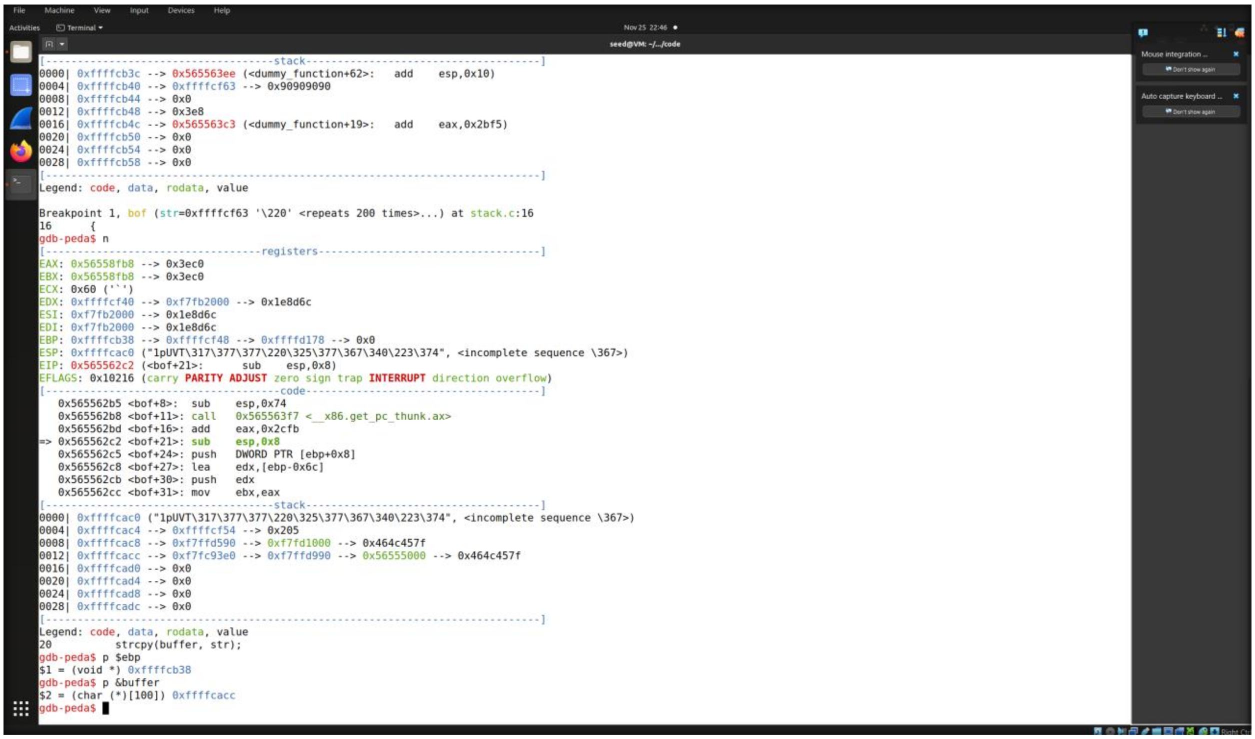
- We can also verify all of the files using "ls -l stack-*" as highlighted in Screenshot 11.



Screenshot 11: Confirming multiple version of stack.c.

Task 3

- By observing the highlighted part in Screenshot 12, buffer address = 0xffffcacc and ebp address = 0xffffcb38

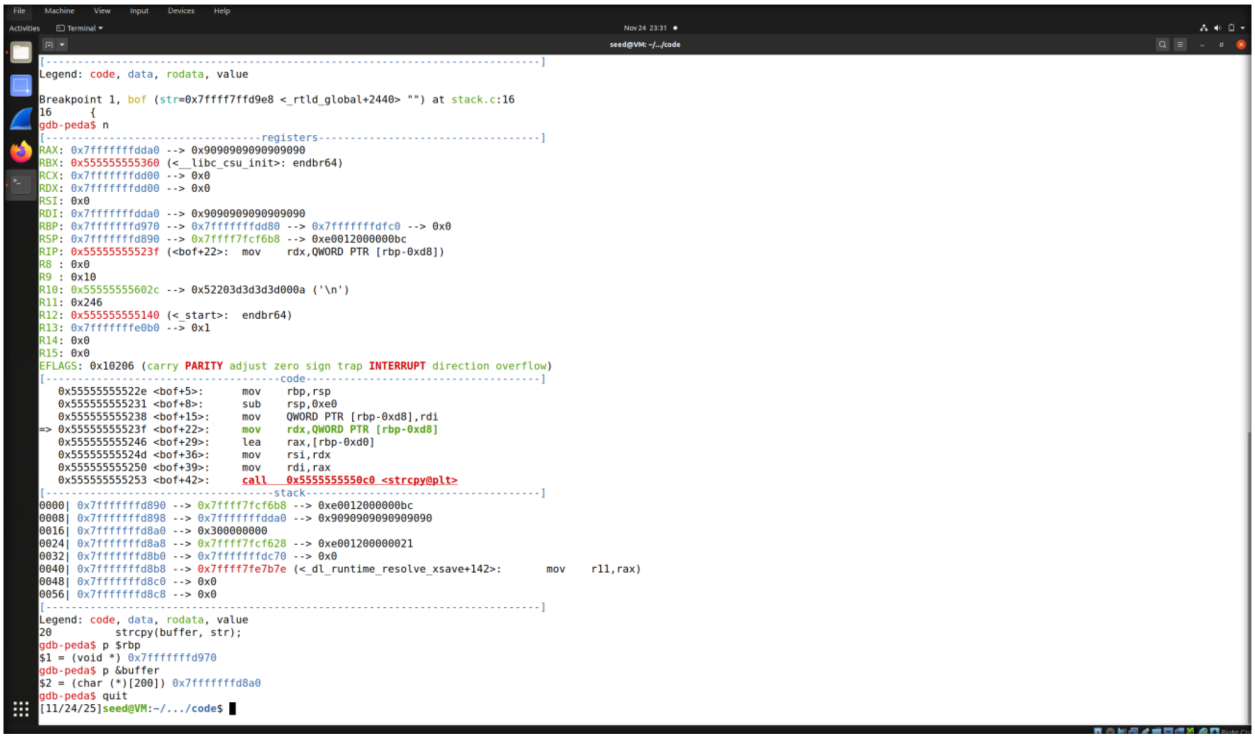


<u>Screenshot 12: Using gdb to get buffer and ebp values.</u>

1. **Offset**- offset = (ebp_address - buffer_address) + 4
   offset = (0xffffcb38 - 0xffffcacc) + 4
   offset = 0x6c + 4
   offset = 108 + 4
   offset = 112
2. **Ret**- ret = buffer_address + 300
   ret = 0xffffcacc + 300
   ret = 0xffffcacc + 0x12c
   ret = 0xffffcbf8
3. **Start**- start = 400

Task 5

- By observing the highlighted part in Screenshot 13, buffer address = 0x7fffffffd8a0 and rbp address = 0x7fffffffd970



Screenshot 13: Using gdb to get buffer and rbp values.

1. **Offset**- offset = rbp - buffer + 8
   offset = 0x7fffffffd970 - 0x7fffffffd8a0 + 8
   offset = 0xd0 + 8
   offset = 0xd8 = 216 (in decimal)
2. **Ret**- ret = buffer_address + 200
   ret = 0x7fffffffd8a0 + 200
   ret = 0x7fffffffd8a0 + 0xc8
   ret = 0x7fffffffd968
3. **Start**- start = 300