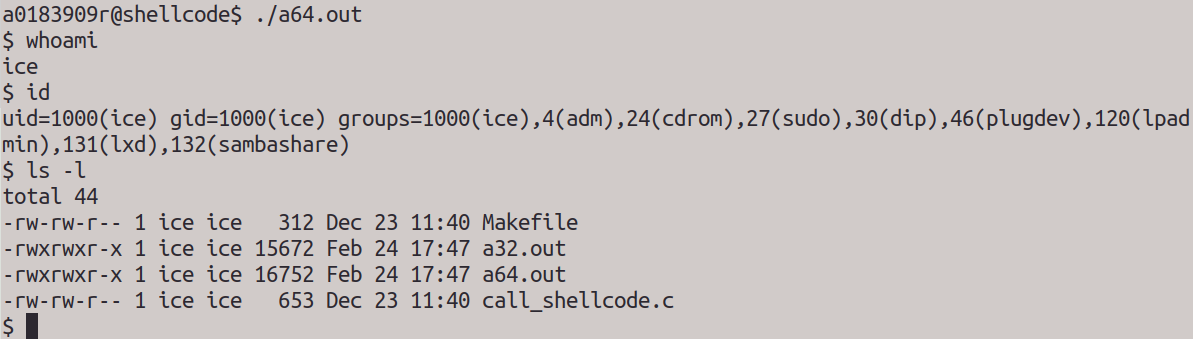
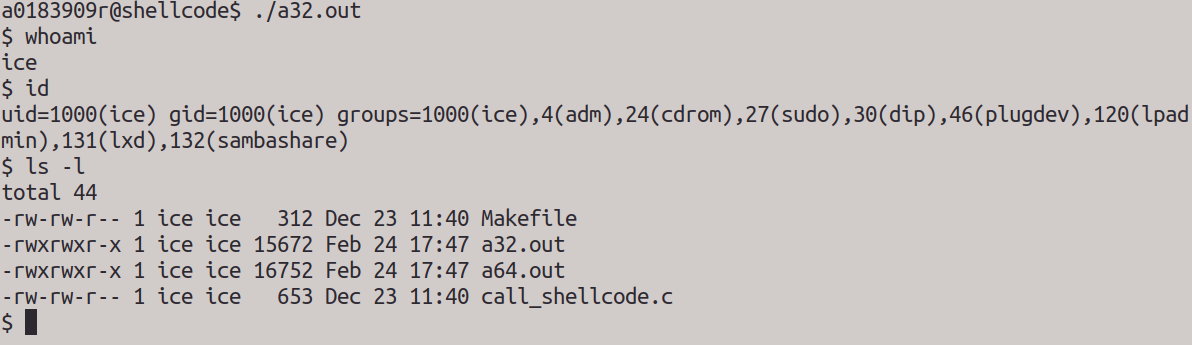
|  |  |
| --- | --- |
| **CS4238 Computer Security Practice Assignment 1** | **Clement Cheng A0183909R e0310704@u.nus.edu** |

# Task 1

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.



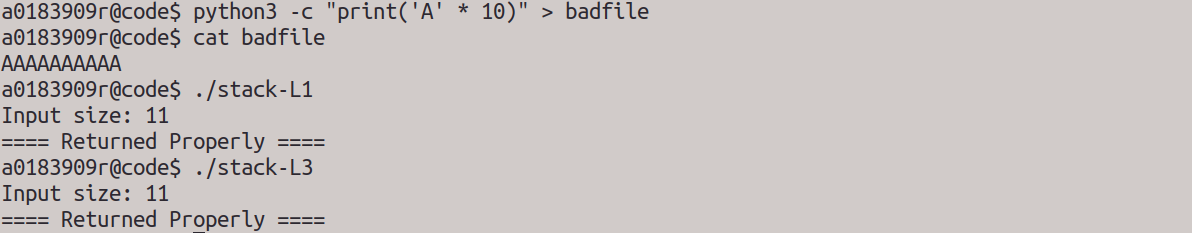
*Figure 1 – 32-bit shell from executing a32.out*



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

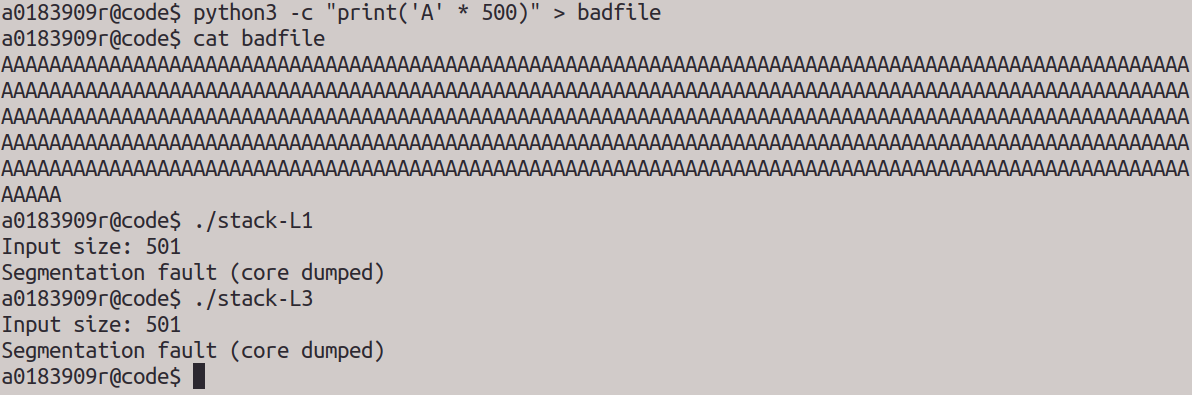
# Task 2

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.



*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.

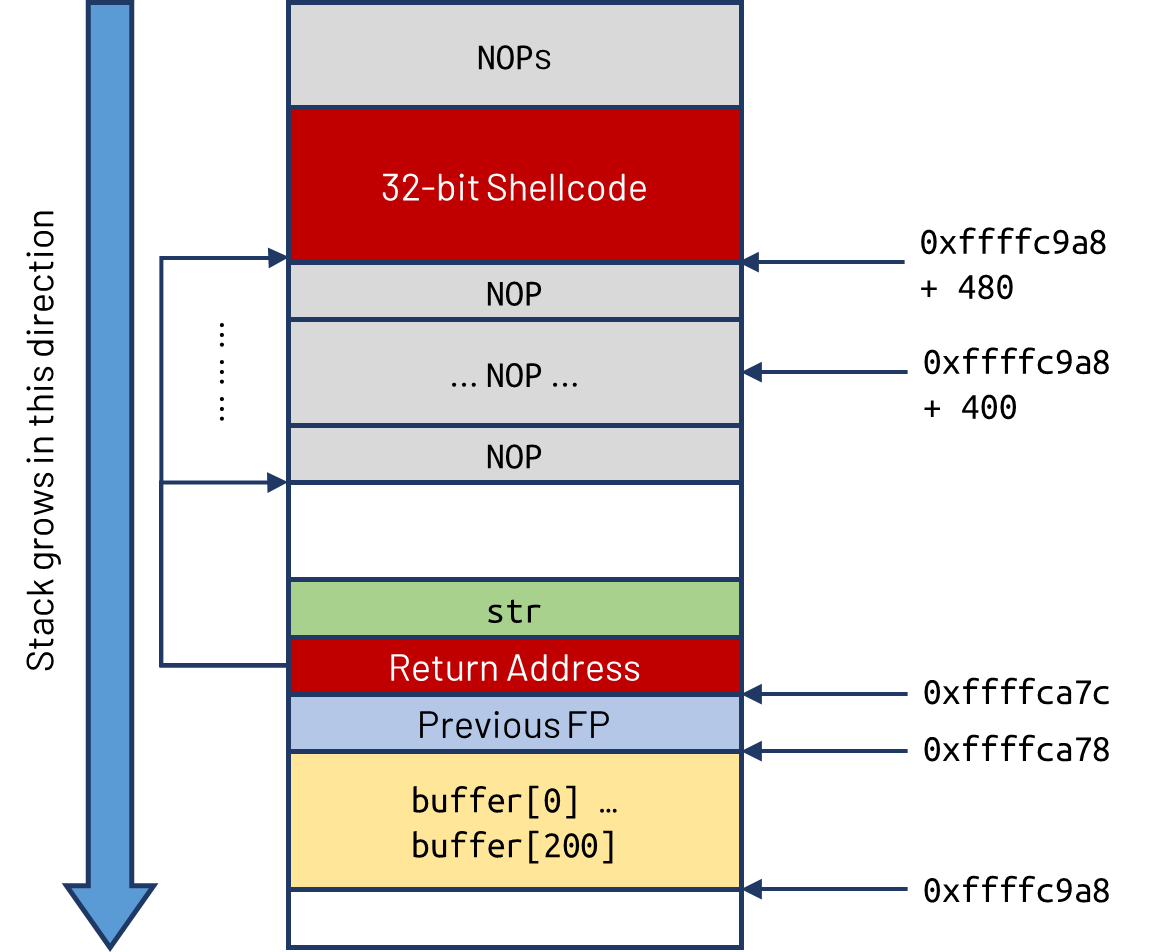


*Figure 4 – Running 32-bit and 64-bit programs with a badfile consisting of a long string*

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 NOPs 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:



*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).



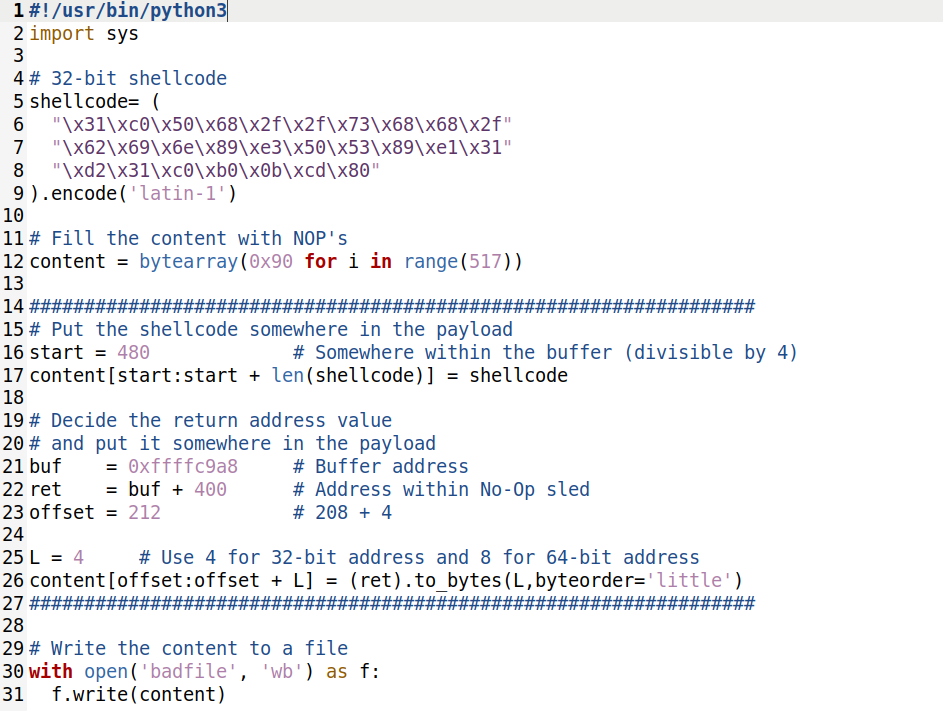
*Figure 7 – Addresses of EBP and buffer local variable*

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:



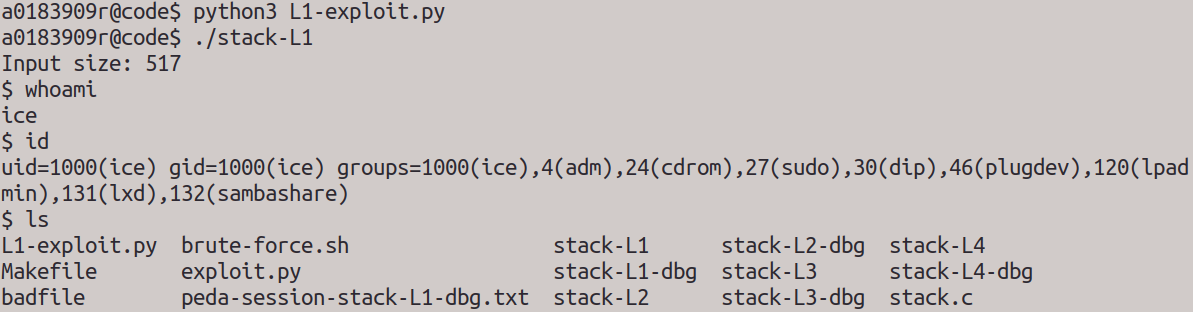
*Figure 8 – exploit.py script for stack-L1*

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 NOPs, except for the shellcode and the new return address, which are located in specific positions in the payload.

**Launching the attack**

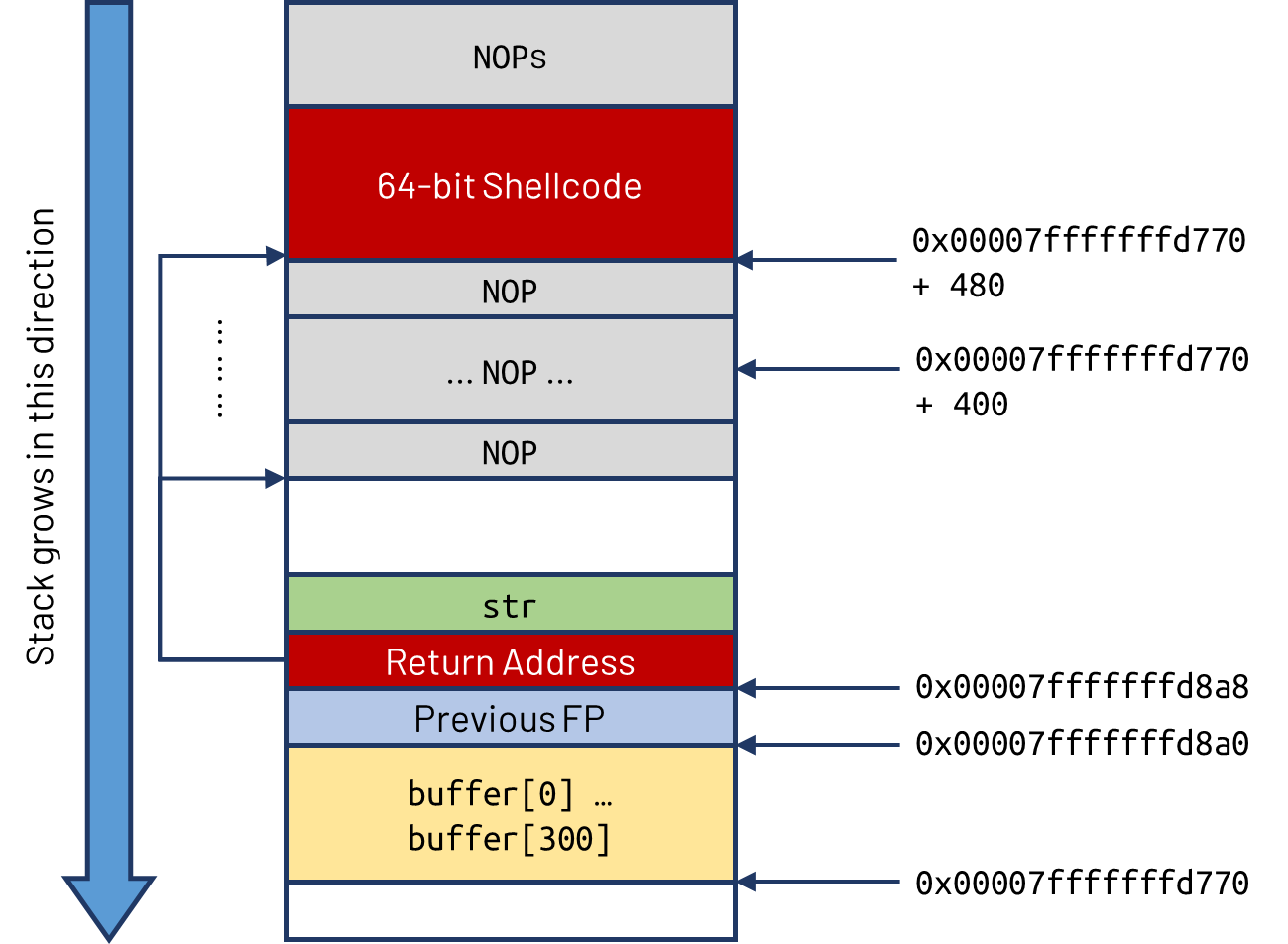
I ran the exploit.py script to obtain the corresponding badfile. Then, I executed stack-L1 and was able to obtain a 32-bit shell successfully.



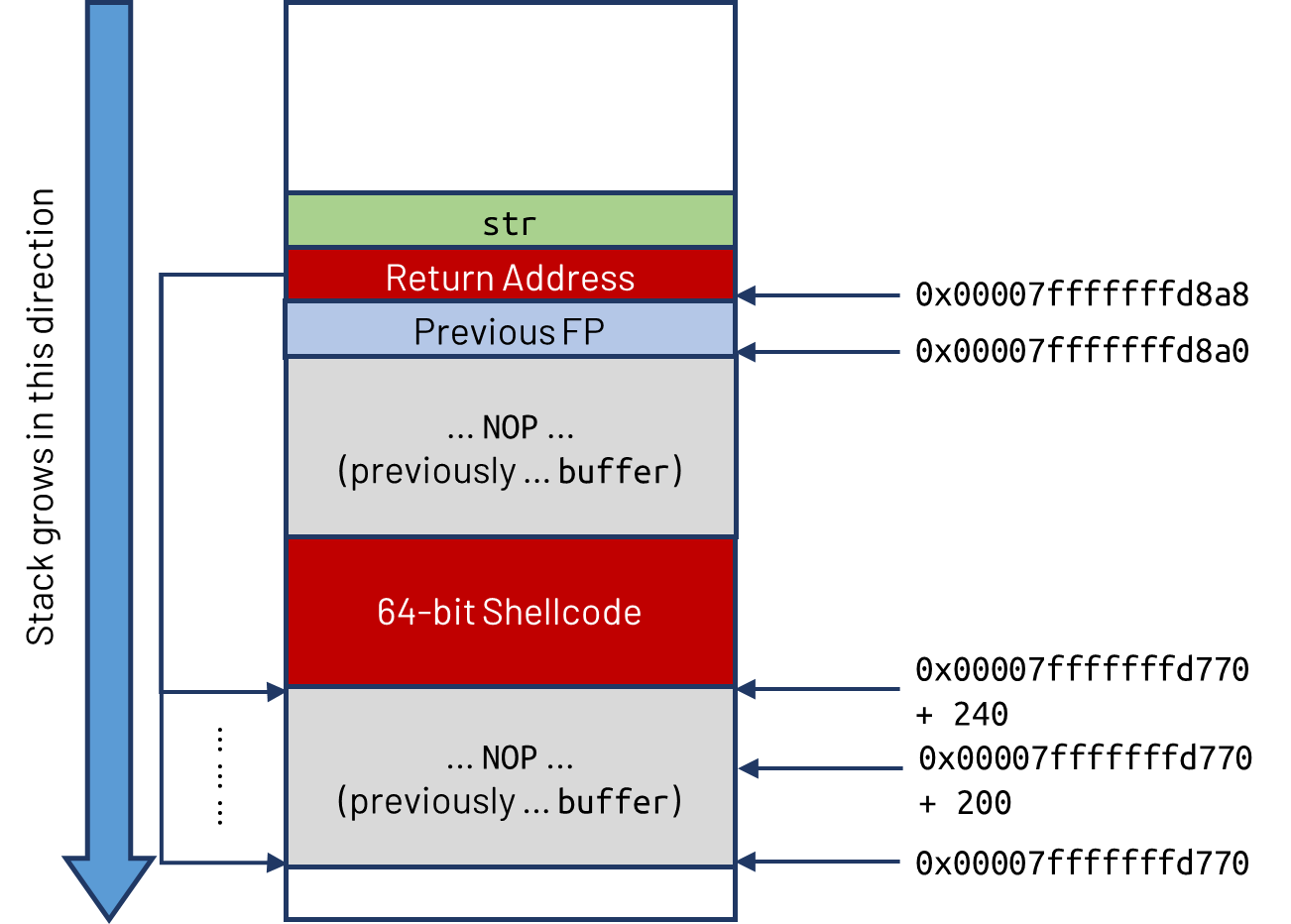
*Figure 9 – 32-bit shell obtained from stack-L1*

# 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**

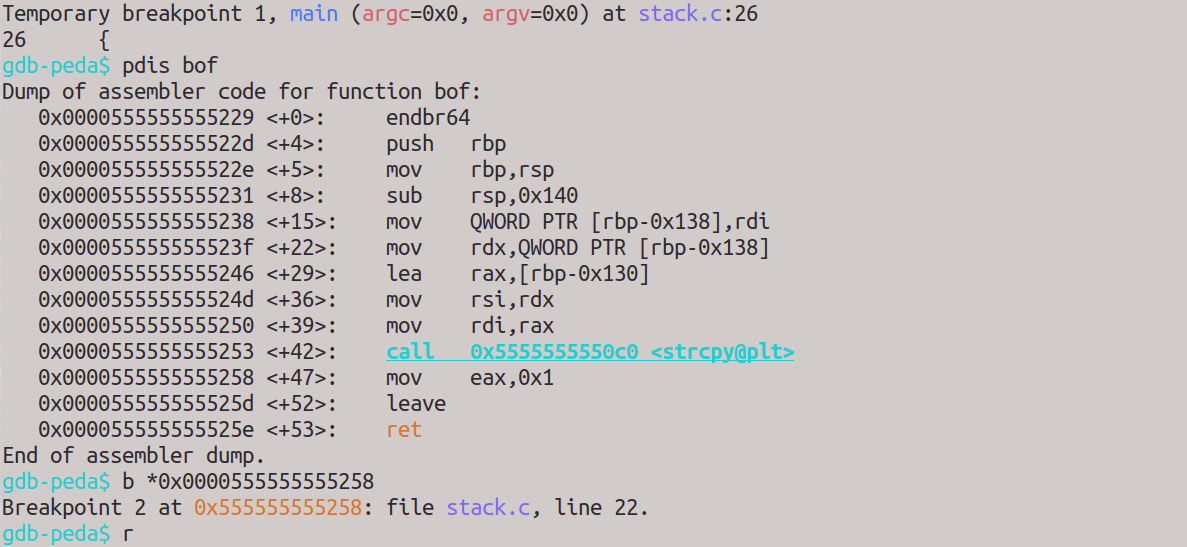
Similarly, we need the address of the start of the buffer local variable and the offset from the start of the buffer local variable to the saved return address.

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:



*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).



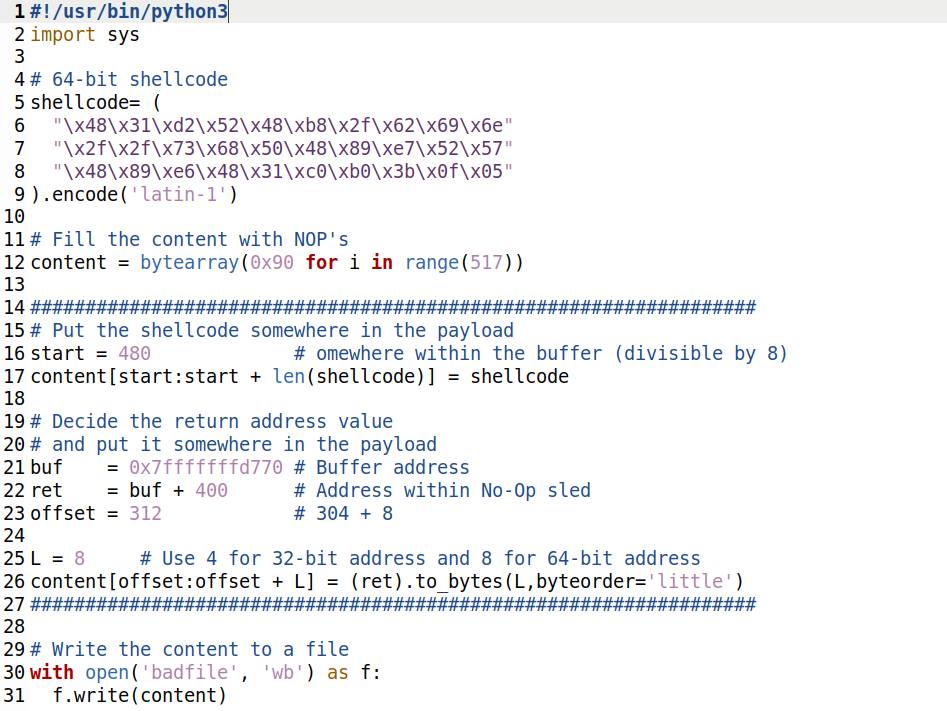
*Figure 12 – Addresses of RBP and buffer local variable*

Offset from start of buffer to saved return address

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:



*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)**

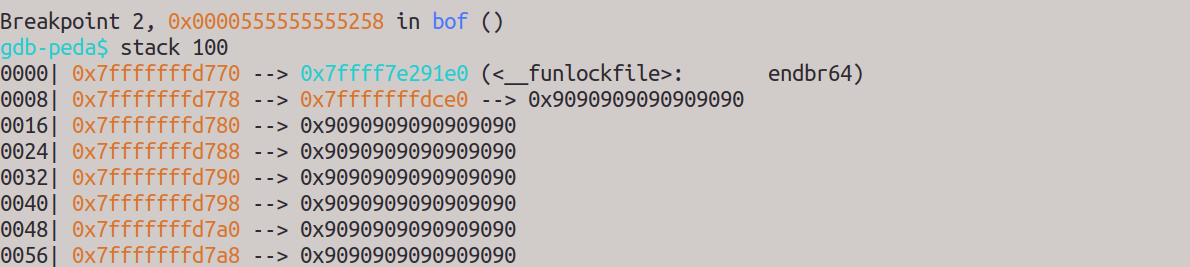
I ran the exploit.py script to obtain the corresponding badfile. Then, I executed stack-L3 and… obtained an illegal instruction error.

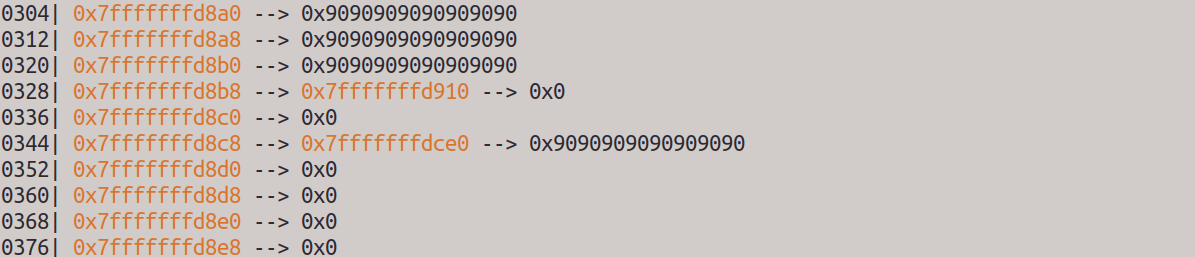


*Figure 14 – Failed attempt in obtaining shell from stack-L3*

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:





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

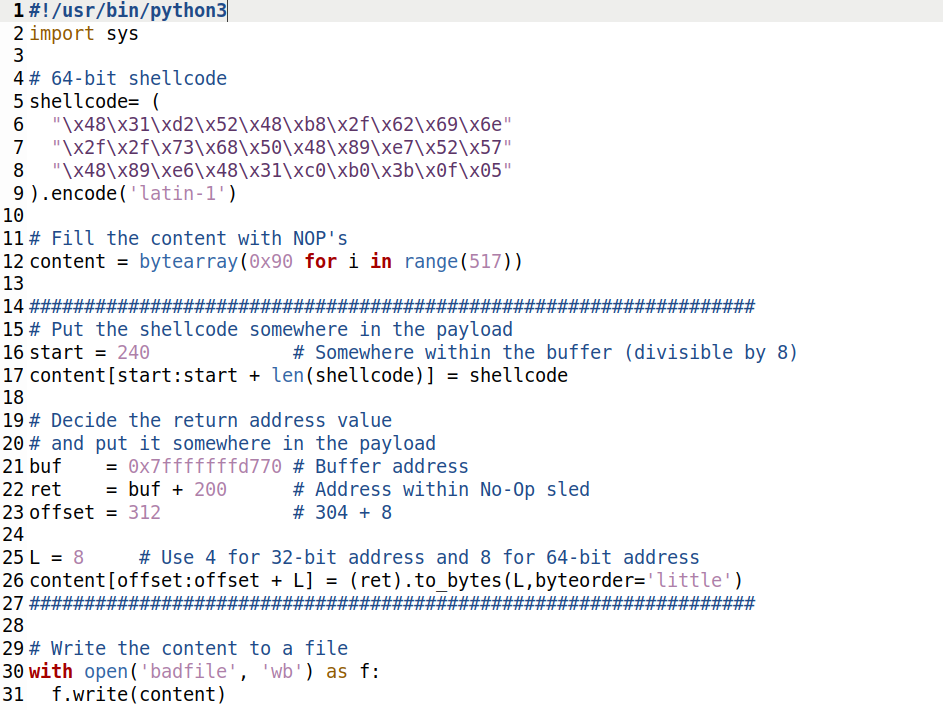
**Overcoming the zero bytes in 64-bit addresses**

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:

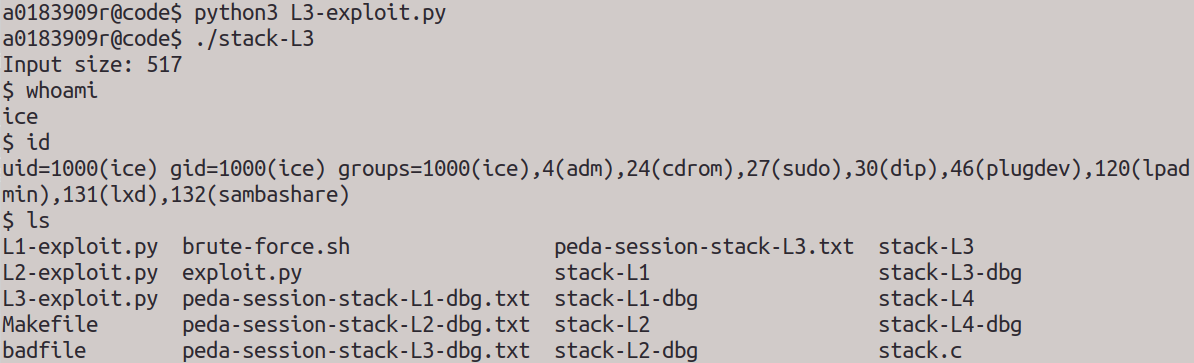


*Figure 16 – exploit.py script for stack-L3 (2)*

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

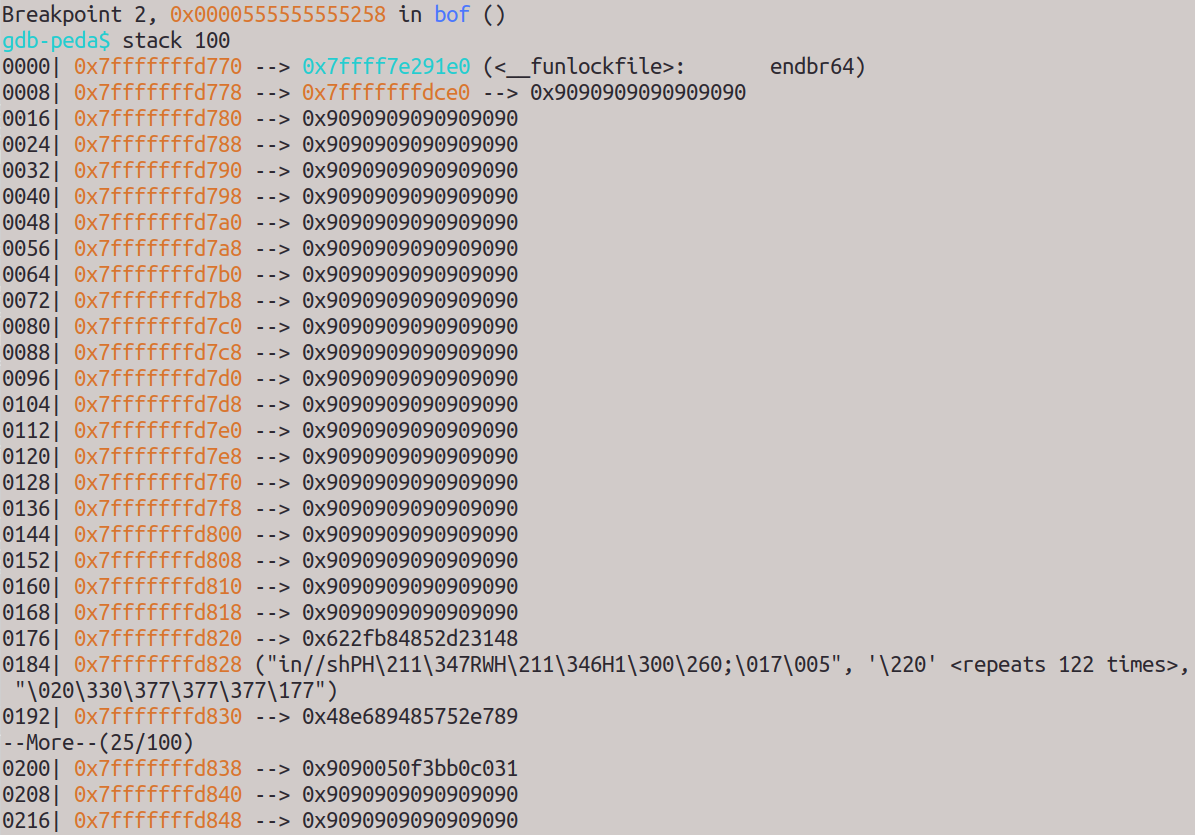
**Launching the attack (2)**

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:



*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.



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

# Task 7

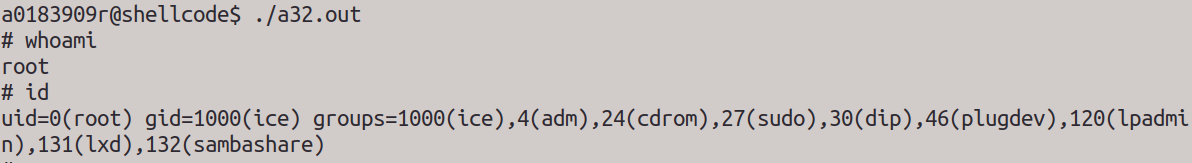
Using the binary code for setuid(0) provided in the call\_shellcode.c source code, I prepended the extension to the beginning of the previous shellcodes as shown in the figure below.



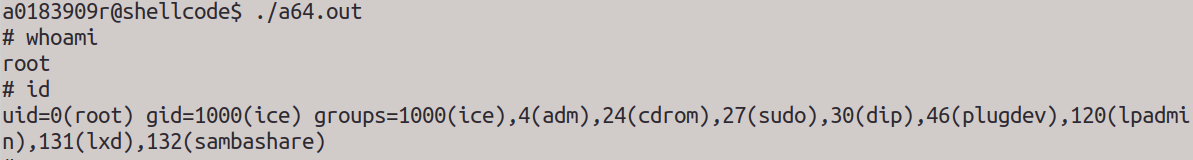
*Figure 19 – Extensions to 32-bit and 64-bit shellcodes*

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.



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



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