Tanmay Fadnavis

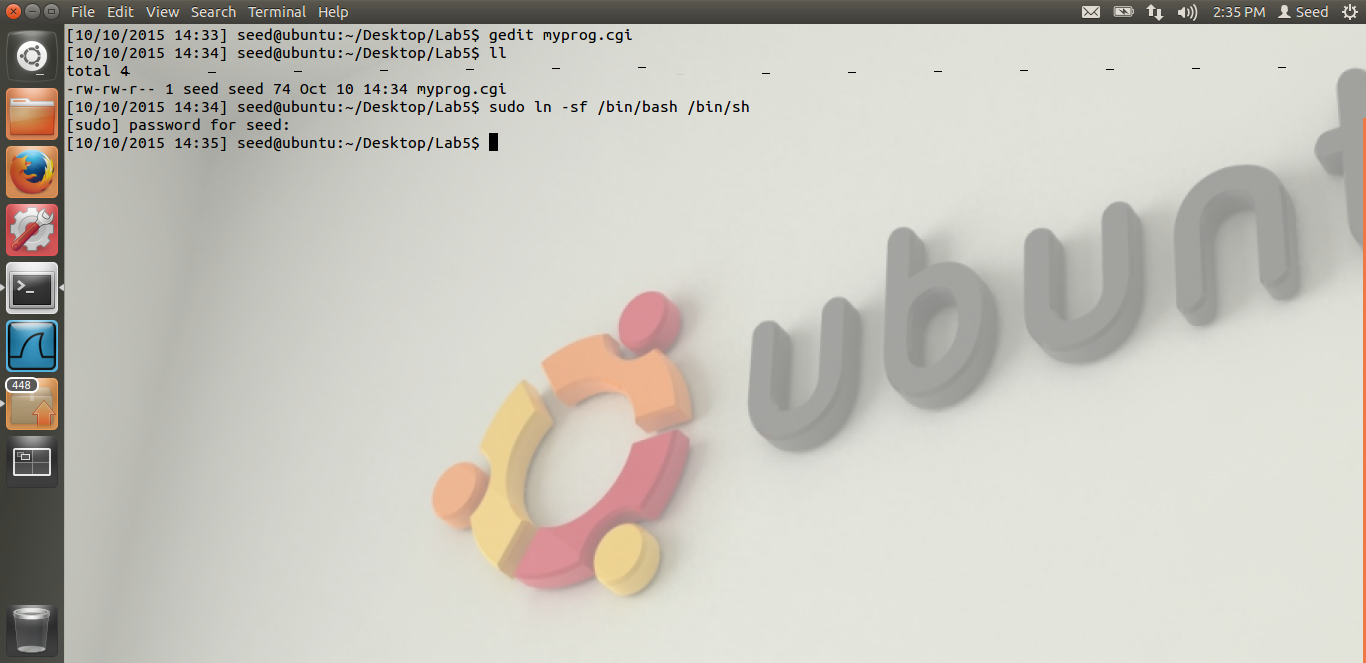
SUID: 971141760

Computer Security

Lab-5

# TASK 1: ATTACK CGI PROGRAMS

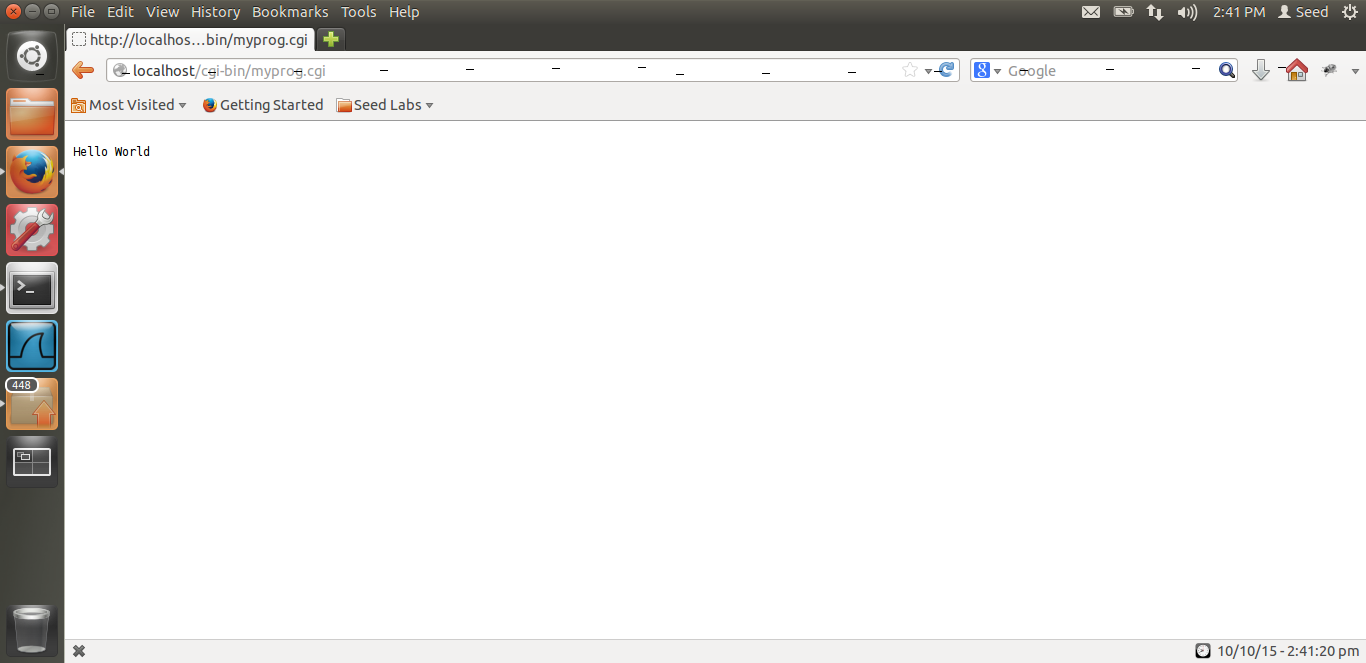
The screen shots for Task 1 are below.



In the above screen shot, we see that I am linking my /bin/sh shell to /bin/bash shell. Reason is that the shell shock vulnerability is in the bash shell.



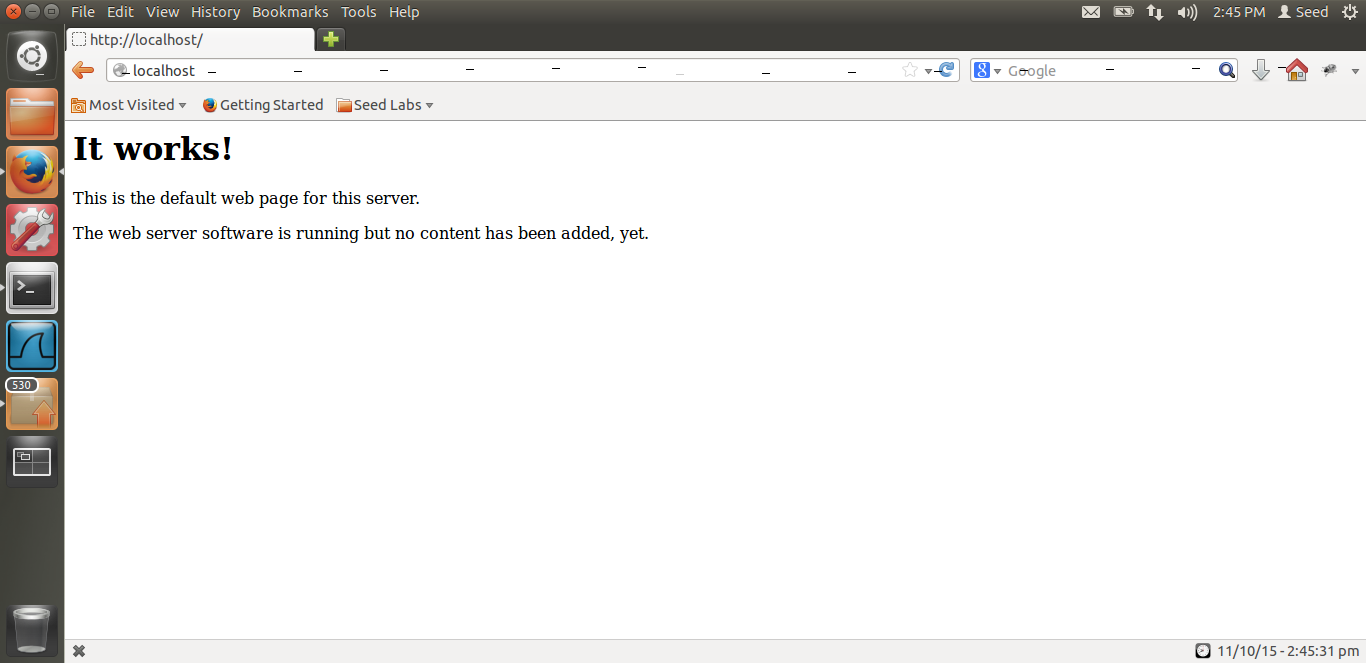
In the above screen shot, we see that, I have compiled myprog.cgi, given by Dr. Du. I have made it owned by the root and copied it to the apache server’s /usr/lib/cgi-bin folder as that is the default folder of the Apache server to execute the cgi files.



This is just the screen shot when I am hitting the url link of myprog.cgi



Screen shot that I have installed the curl application and when I am running the curl command, I am getting “Hello World”.



This is the webpage that we get, when I type localhost. This is the page before the shell shock attack. In the /var/www directory, there is an index.html file. This is that file. Its owned by www-data. After the attack, I am going to remove that file. Even though it is owned by the www-data and when I am hitting my user program cgi, due to the shell shock vulnerability, I can remove it.

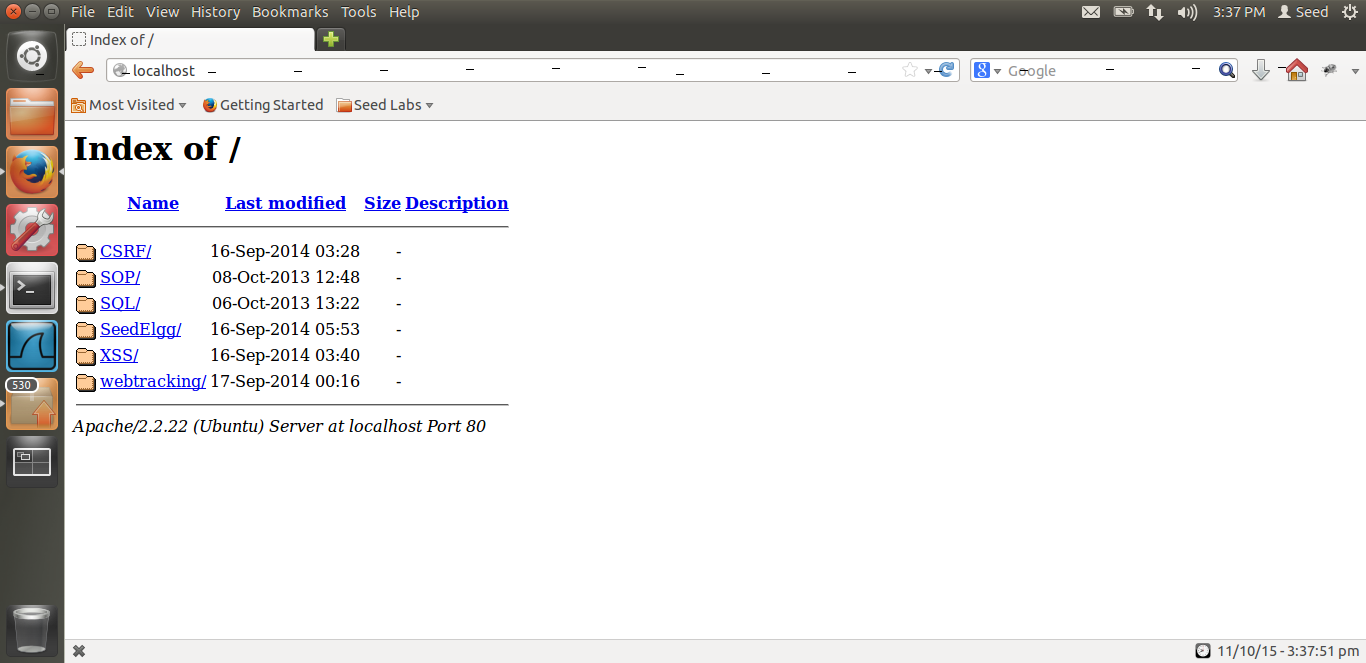


In the above screen shot, we can see the attack. As you can see in the top-most line, the curl command which I execute. The curl command is as below.

Curl –A “() { :; }; /bin/rm /var/www/index.html” \http://localhost/cgi-bin/myprog.cgi

Thus, in the above command, due to the shell shock vulnerability, I am able to delete the index.html file owned by www-data. The reason is, when Apache server executes CGI program, bash shell is being invoked before the CGI program’s execution. Thus, when the bash shell is invoked, in the bash shell, the functions are stored like environment variables. The function name is stored as key, and the definition is stored as value. But as the parsing is not done correctly, a malformed function definition also gets stored. Thus, in my curl command, I was able to remove the index file, using the /bin/rm command and specifying the path, /var/www/index.html. Thus, when the bash shell is invoked before the CGI program, the environment variables are evaluated and my malicious code is run successfully. Thus I am able to successfully do the attack.

After the attack when I do localhost, we can see that the index.html file has been removed.



Thus, in the above screen shot, we can see that, when I access localhost again, I can see the directory contents instead of the index.html, as I have removed only the index.html. I could have removed the entire /www directory as well.

Thus I am able to do the attack successfully.

**Vulnerability in variables.c**

In the bash shell code, in the variables.c program, in the function initialize\_shell\_variables,

We can see the below code.

**if (privmode == 0 && read\_but\_dont\_execute == 0 && STREQN ("() {", string, 4))**

{

string\_length = strlen (string);

temp\_string = (char \*)xmalloc (3 + string\_length + char\_index);

strcpy (temp\_string, name);

temp\_string[char\_index] = ' ';

strcpy (temp\_string + char\_index + 1, string);

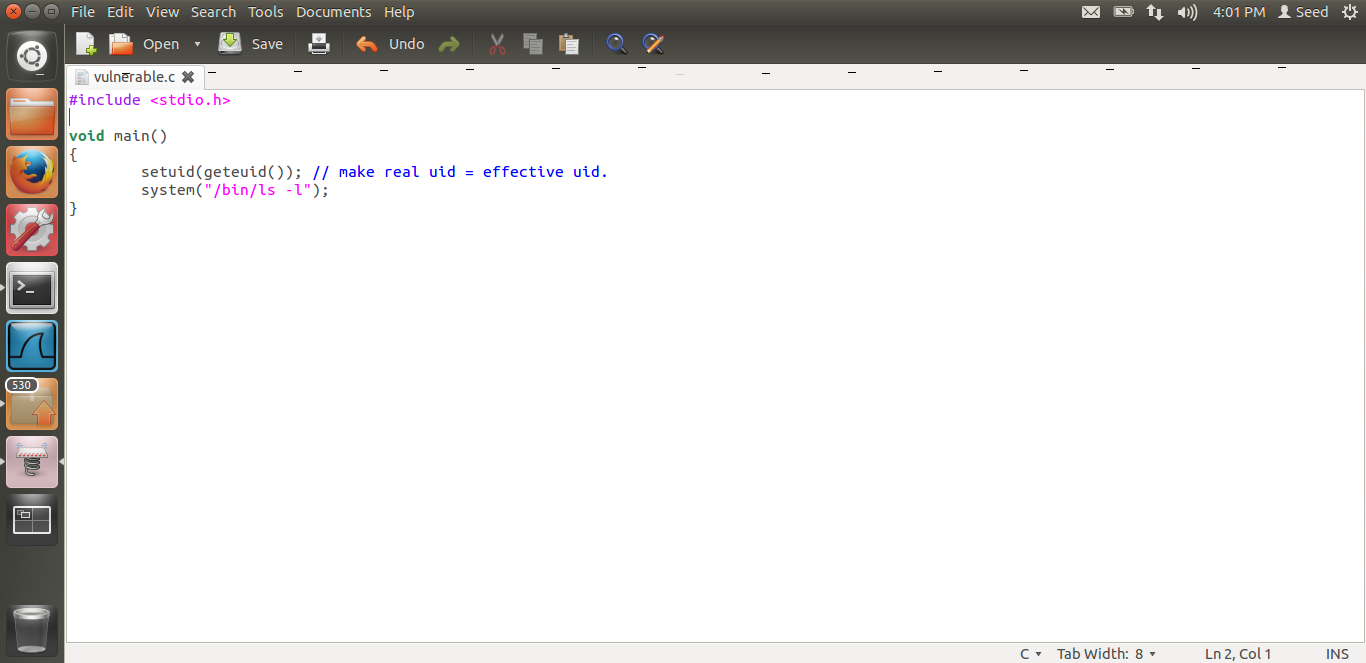
**parse\_and\_execute (temp\_string, name, SEVAL\_NONINT|SEVAL\_NOHIST);**

Here, we can see that in the if condition, the check is done for the pattern () { i.e. the function definition. If its true, the code converts it to the environment variable. The issue is, there is no check for whether the function is defined properly.

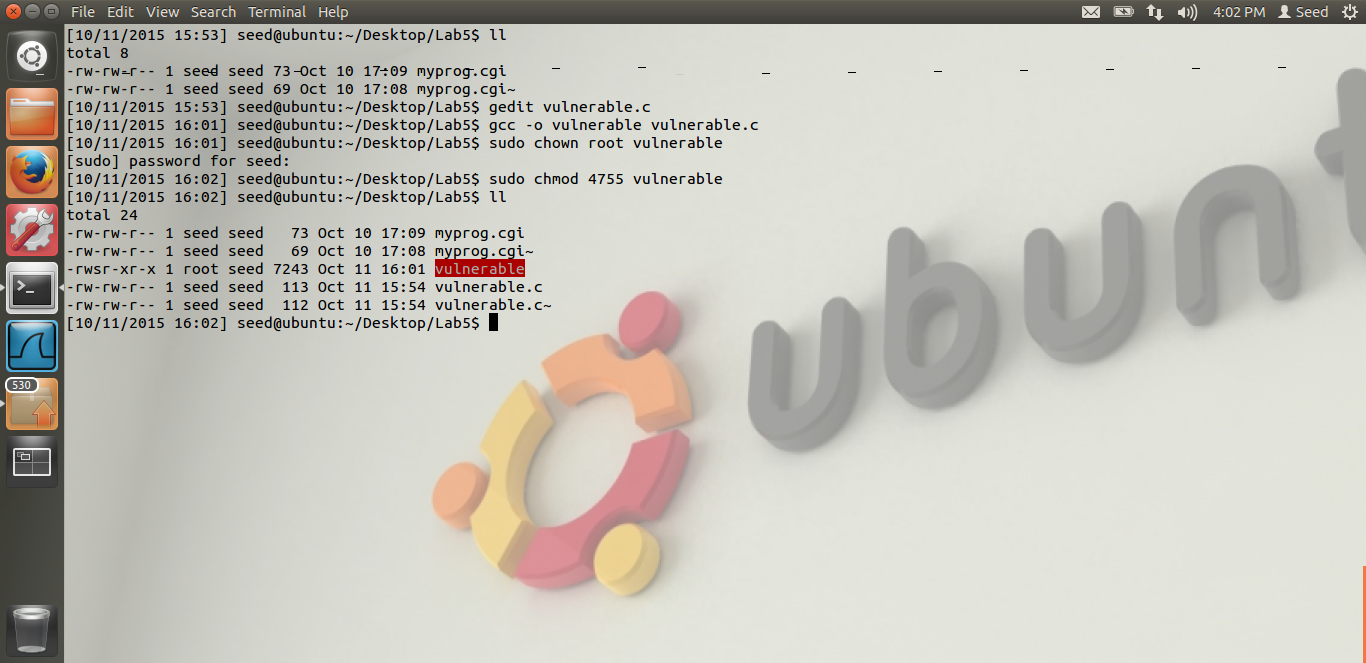
Also, the parse\_and\_execute function, it just parses and executes whatever is given to it. Thus, it parses the strlen(string), i.e. whatever is written in the function definition and even after the definition, as there is no check for the legitimate function definition. Thus any attacker can put malicious code after the function definition and that code will be executed. This is the shell shock vulnerability.

# TASK 2-A: ATTACK SET-UID PROGRAMS

The screen shots for task2 are below.



In the above screen shot, we can see the code provided by Dr. Du.



In the above screen shot, we can see I have compiled the vulnerable code, made its owner as root, and made it a SET-UID program.

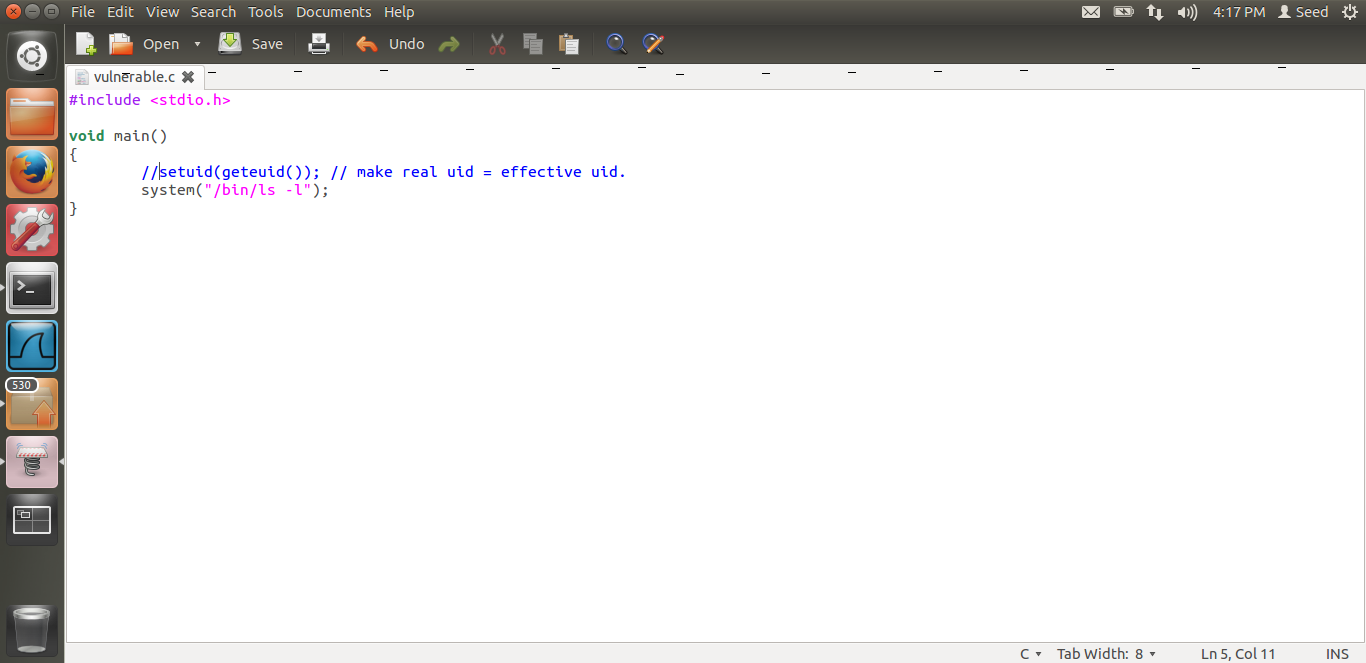


In this screen shot, I am doing the attack. i.e. I am exporting an environment variable call shellshock. If we see the definition, after the function is ended, I am putting my malicious code, i.e. I am executing /bin/su root, i.e. I am just doing su to root. As the code which is going to executed is a set-UID program owned by root, when I run it, I will get the root shell, as the bash shell will be invoked and su root will be executed, I will login as root.

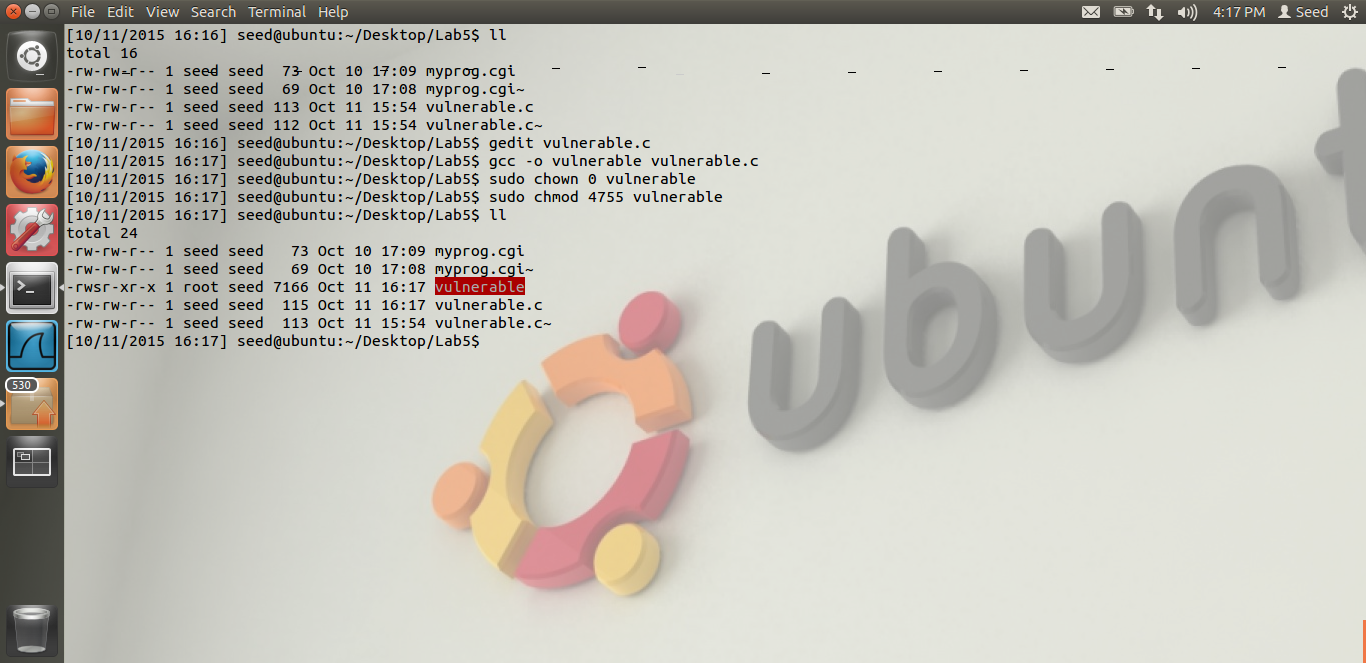
Thus, from the screen shot we can see, when I execute the vulnerable program, I get the root shell. Thus, this I confirm it by doing the id command. We can see that the uid is root. Thus, I was able to do the attack successfully and exploit the shellshock vulnerability and gain the root privilege.

# TASK 2-B: REMOVING setuid(geteuid)

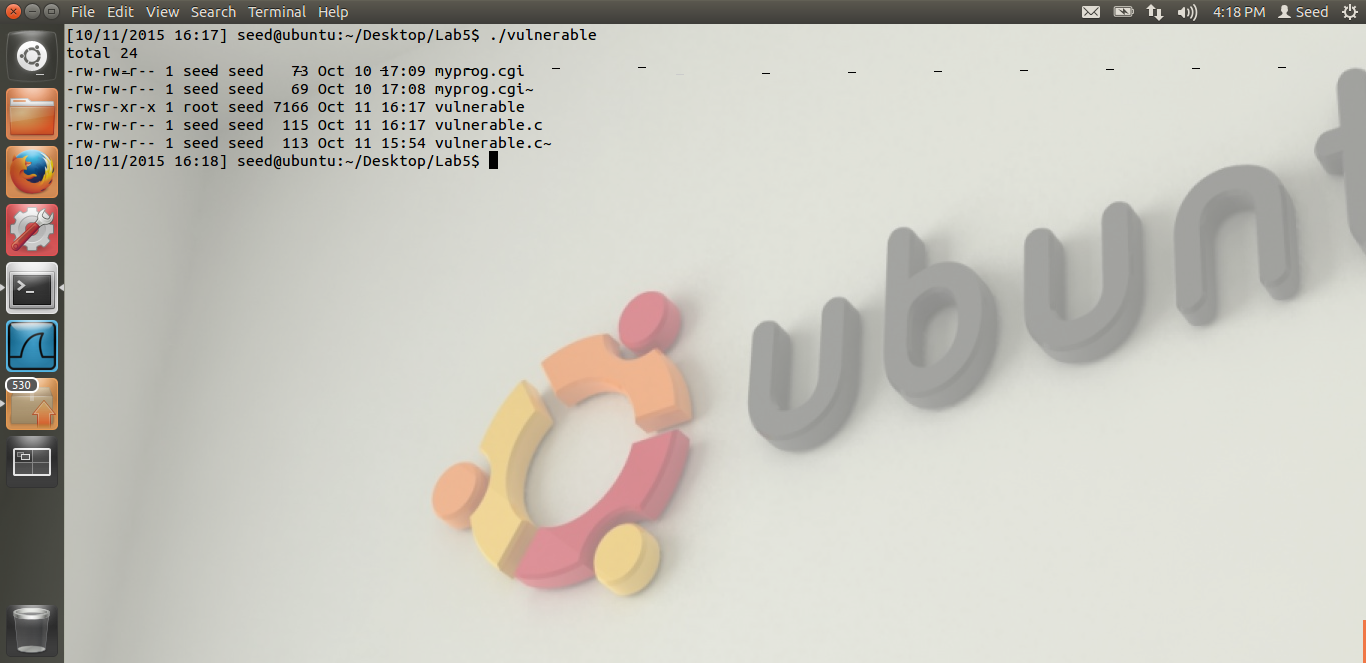
The screen shots for task-2 are below.



As we can see from the screen shots, I have comment the line “setuid(geteuid())”.



Thus, as we can see, I have compiled the code and made it a set-UID program. Now let us execute and see what happens.



In the above screen shot, we can see that I have executed the vulnerable program. But instead of getting the root shell like in the previous example, the ls –l command is being executed. Thus I was not able to do the attack successfully.

**Reason:** The reason is, as we have commented the setuid part, the real userid and the effective user id are different for this program. The shell shock attack works only if the real user id and the effective user id are the same. If these 2 user ids are not the same, the function defined in the environment variable is not executed at all. Thus, even though we have defined a malicious env variable, due to the difference in the real and effective user ids, the attack was not successful. In the below code, the privmode becomes non-zero if the effective effective user-id and the real user-id are not same. Hence the if condition fails and the environment variables are not evaluated at all. Hence the attack fails.

**Vulnerability in variales.c**

In the bash shell code, in the variables.c program, in the function initialize\_shell\_variables,

We can see the below code.

**if (privmode == 0 && read\_but\_dont\_execute == 0 && STREQN ("() {", string, 4))**

{

string\_length = strlen (string);

temp\_string = (char \*)xmalloc (3 + string\_length + char\_index);

strcpy (temp\_string, name);

temp\_string[char\_index] = ' ';

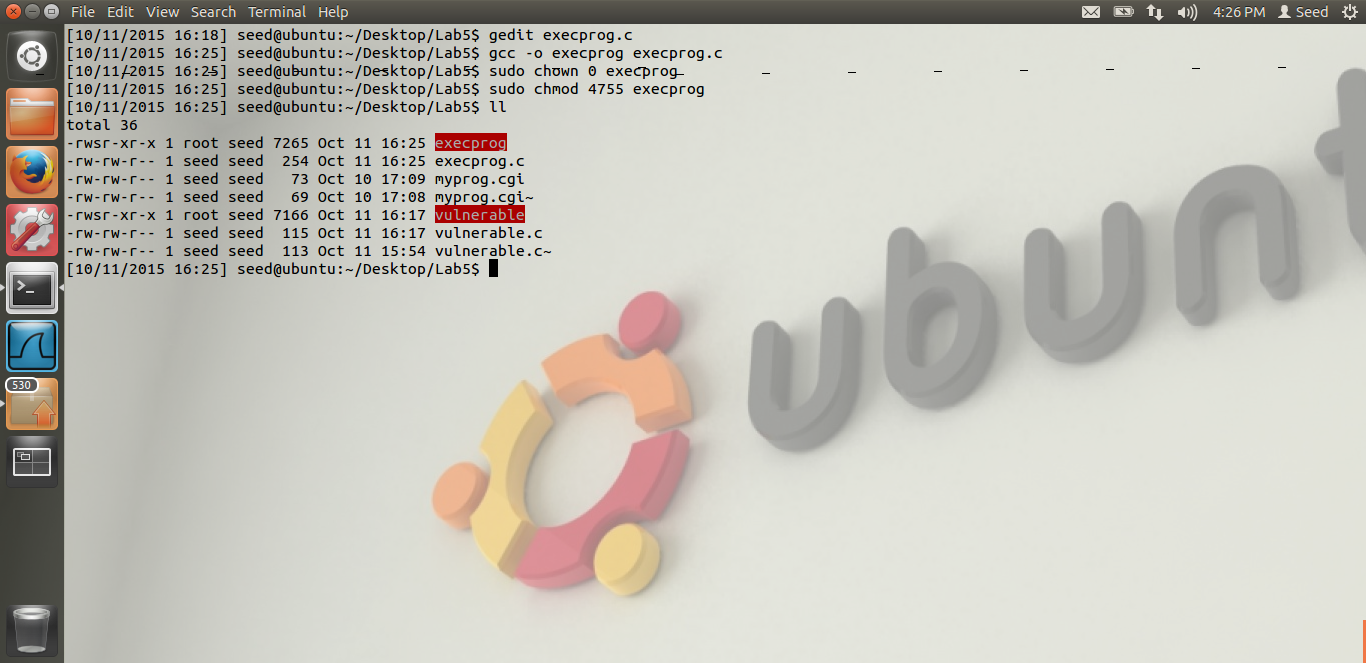
strcpy (temp\_string + char\_index + 1, string);

**parse\_and\_execute (temp\_string, name, SEVAL\_NONINT|SEVAL\_NOHIST);**

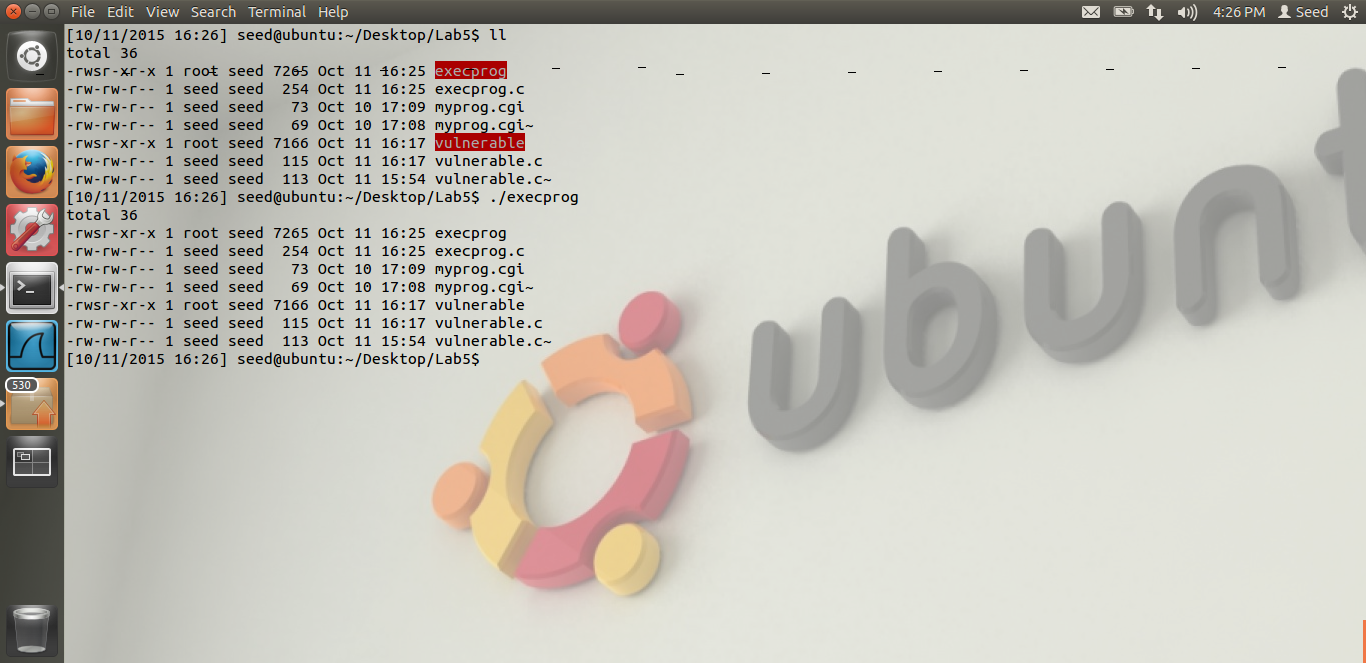
Thus, in the above code, we can see that, only if the privmode ==0, i.e. only when the effective user id and the real user id are the same, only then the if condition is satisfied and the environment variables are executed. In task 2-b, this condition fails, hence our malicious environment variable is not executed.

# TASK 2-C: USING EXECVE

The screen shot for task 2-c are belowe.



Thus, I have compiled the execve program provided by Dr. Du. And made it owned by root and set-UID as seen from the screen shot. Let us try executing this.



From the above screen shot, we can see that, even though I have executed the program, the ls –l command is run and our attack is not successful. Even though we have made the real and the effective use id as the same, we were not able to do the attack successfully.

**Reason:** The reason is very simple. Execve() unlike the system() function call does not open a shell. Thus execve() is a very secure call and it directly invokes the system call. Thus, as the shell is not invoked, there is no question of the shellshock vulnerability when we are using the execve(). Hence, we are not able to do the attack successfully.

# TASK 3

2. What is the fundamental problem of the Shellshock vulnerability? What can we learn from this vulnerability?

The fundamental problem of the shellshock vulnerability is that, it doesn’t check whether the function definition is complete or nor. i.e. in the variables.c, we can see that **STREQN ("() {",**

Only the opening of the function syntax is checked. Everything after that is parsed and executed. Hence there is a scope for the attacker to put the malicious code after opening the bracket and this is what exactly happens in the shell shock attack. Thus, not checking the correct function definition is the fundamental problem of this vulnerability.

We learn that, putting a check at the right places is very important. The only check which was performed in the bash was the opening brackets and the parenthesis and it was assumed that everything after that is a proper function definition. But this gives the attacker a window to exploit this. Especially as the code is being executed without even checking. Hence we should not assume anything about the input. If we see that there is a function, we need to make sure that the function is defined properly. Thus, these checks from the security perspective are very important.