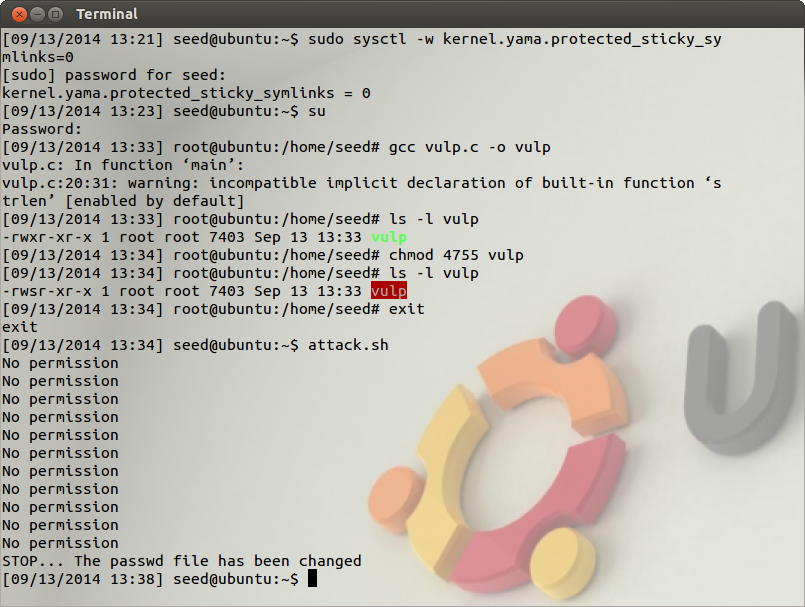
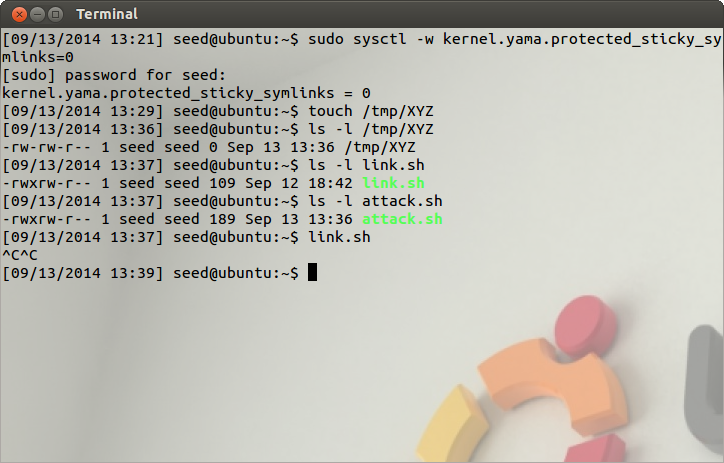
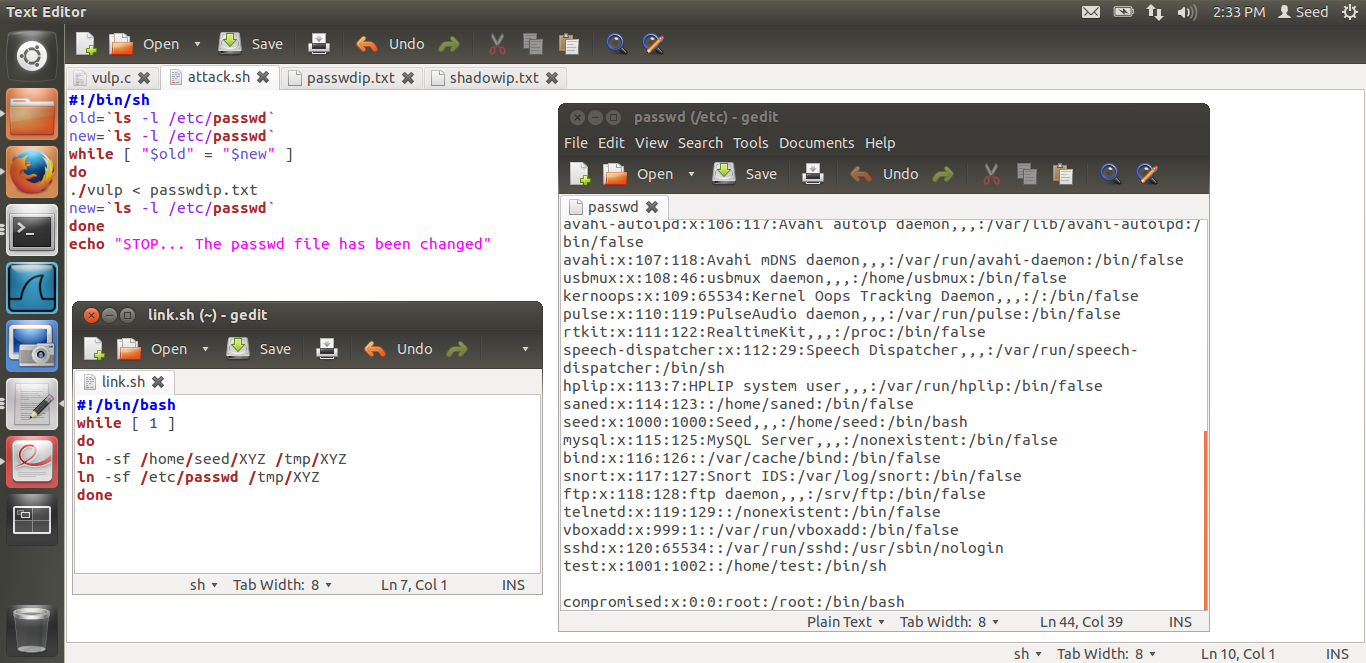
**Race-condition Vulnerability Lab Report**

**Task 1:**

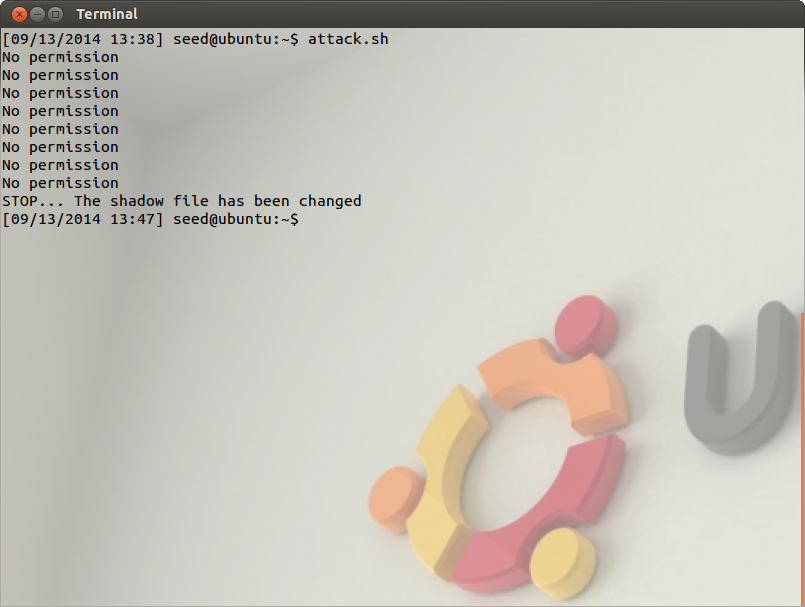
**  
Figure 1.1**

**  
Figure 1.2**

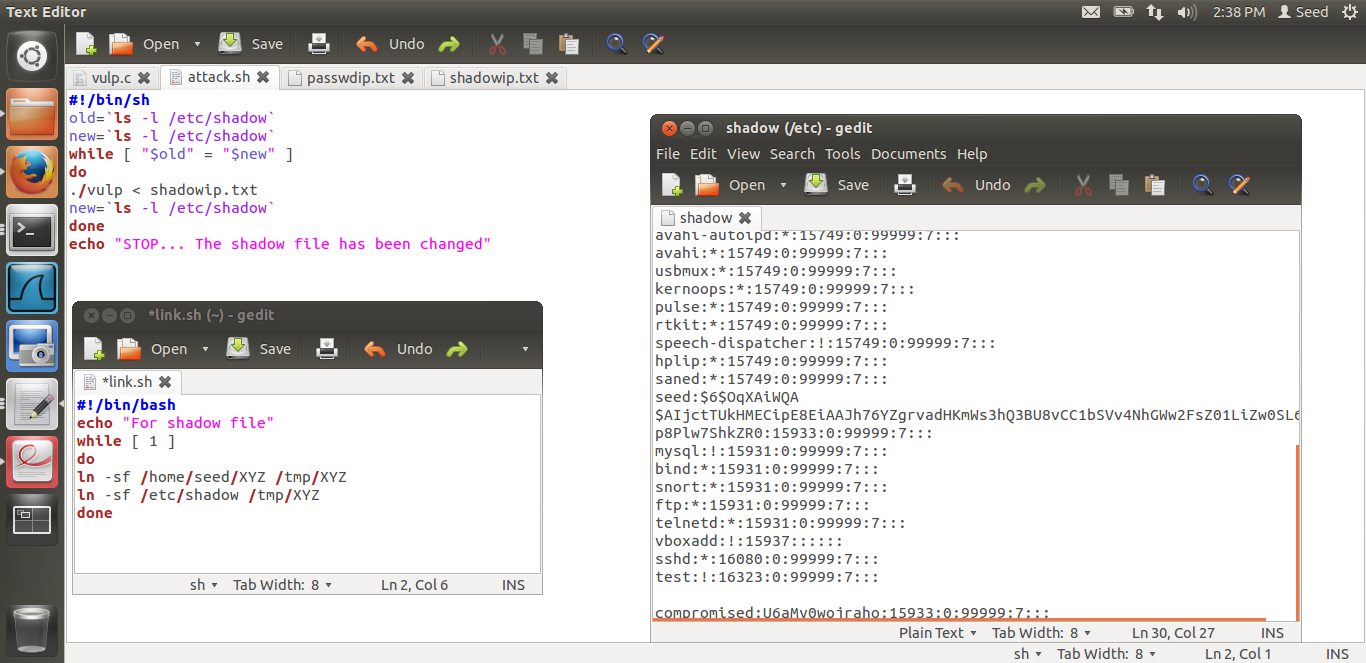
Link.sh file is running to link and unlink the root owned and the user owned files to exploit race condition

**  
Figure 1.3**

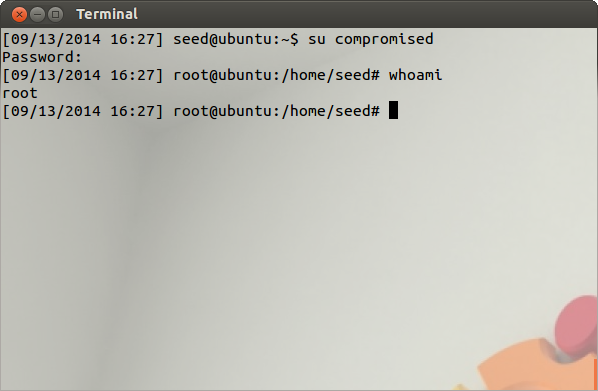
Addition to passwd file after success of attack

**  
Figure1. 4**

**  
Figure 1.5**

**  
Figure 1.6**

Addition to shadow file for creating new root user

****

New user called compromised is created.

Indicates that root user is issuing commands from shell

**Figure 1.7**

**Observations:**

1. Using the command

*$ sudo sysctl- w kernel.yama.protected\_sticky\_symlinks=0*

We turn off the built-in protection provided by the operating system. We turn this bit off in figure 1.1.

1. We create an executable, *vulp* and make it a Set-UID root program by using

*‘chmod 4755 vulp’*

1. **Overwrite any file that belongs to root(/etc/passwd)  
   Gain root privileges(/etc/shadow)**  
   We write a shell script "attack.sh" that checks repeatedly if the */etc/passwd*(figure1.1) and */etc/shadow*(figure 1.4) file has been modified or not. Also it continues to provide our input file using direction operator(<) for */etc/passwd* (figure 1.4) and */etc/shadow* (figure 1.6)file repeatedly so that whenever the condition of the window attack becomes true we are able to write to the two root restricted files in order to complete the attack and create a user with blank password.

*./vulp < passwdip.txt  
./vulp < shadowip.txt*

We write a shell script "link.sh" that is used to switch the file pointer between the normal user's file (/home/seed/XYZ) and the root owned files (/etc/passwd and /etc/shadow).

For */etc/shadow* file,

*ln –sf /home/seed/XYZ /tmp/XYZ  
ln –sf /etc/shadow /tmp/XYZ*

For */etc/passwd* file,

*ln –sf /home/seed/XYZ /tmp/XYZ  
ln –sf /etc/passwd /tmp/XYZ*

1. **Competition Question-Understanding of the question:** How to shorten the window where the file creation happens after access and no file is present, which leads to a loss of race-condition as the ln –sf command cannot force remove /tmp/XYZ error as the operation is not permitted on files owned by root. When this issue happens the race condition is lost and the link has to be removed by the root user to allow for attack to take place again. The reason that this happens is that a context switch occurs between the link and unlink commands in case of the C-program.

**Explanations:**

1. The sticky bit is used to enforce DAC (Discretionary Access Control). ‘*/tmp*’ is a world-writable sticky directory. According to the documentation, “symlinks in world-writable sticky directories (e.g. /tmp) cannot be followed if the follower and directory owner do not match the symlink owner.” This bit needs to be turned off for us to be able to succeed with the attack.
2. Performing *‘ls –l vulp’* tells us that vulp is now a Set-UID root executable.
3. **Overwrite any file that belongs to root(/etc/passwd)  
   Gain root privileges(/etc/shadow)**  
   "attack.sh" runs a loop to keep checking if the files have been modified or not, so that we are informed when the attack is successful. Also we repeatedly provide the input to *vulp* which requires a user input. This is done because we will not be able to provide the required input fast enough for the window.  
   "link.sh" repeatedly switches the symbolic link between the normal user’s file and the root owned file, in an attempt to have the *access()* check succeed and then switch the file to the root owned file just before the *open()* function is called, thus gaining privileged access to a root-owned file.  
   On gaining access to the */etc/passwd* file we add an entry that will allow us to gain access to root with just a blank password as input for an account that is created similarly but this time by accessing the */etc/shadow* file. When we access the */etc/shadow* file we add the following entry to the file to create a new user "compromised" with root privileges,

*compromised:U6aMv0Woiraho:15933:0:99999:7: : :*

Here, compromised is the users name and *U6aMv0Woiraho* is the encryption for a blank password.

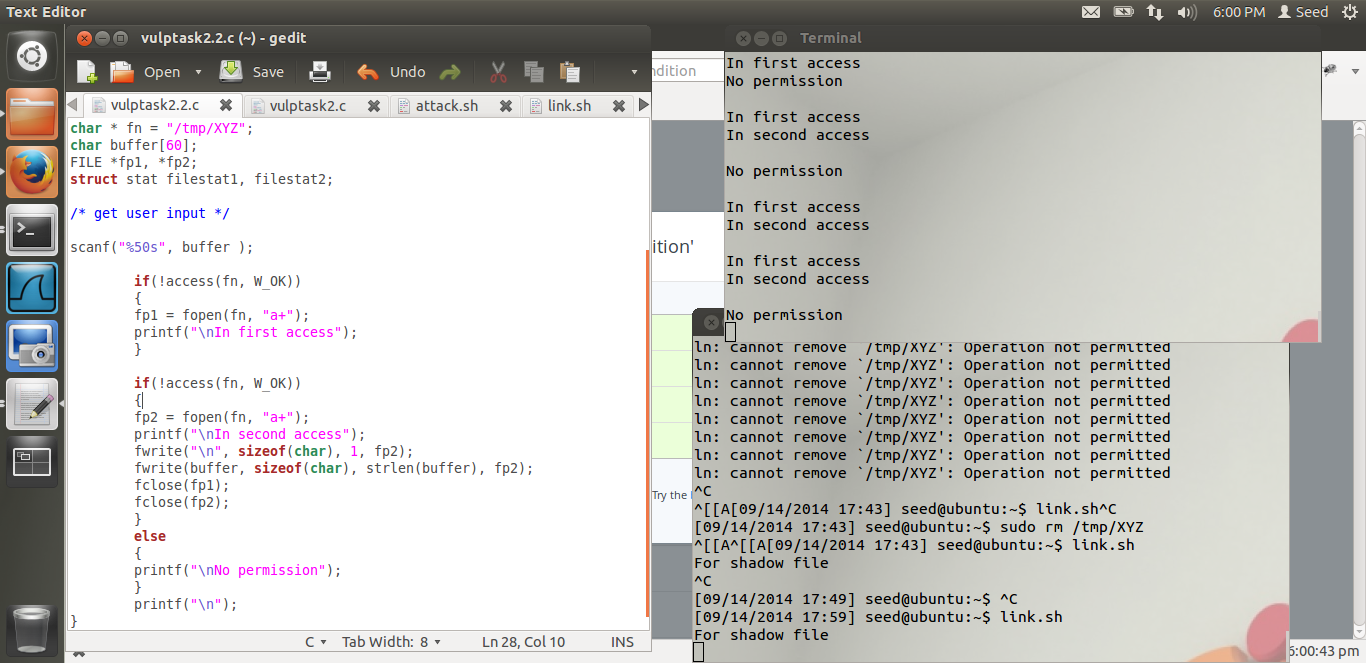
Addition to */etc/passwd* file,

*compromised:x:0:0:root:/root:/bin/bash*

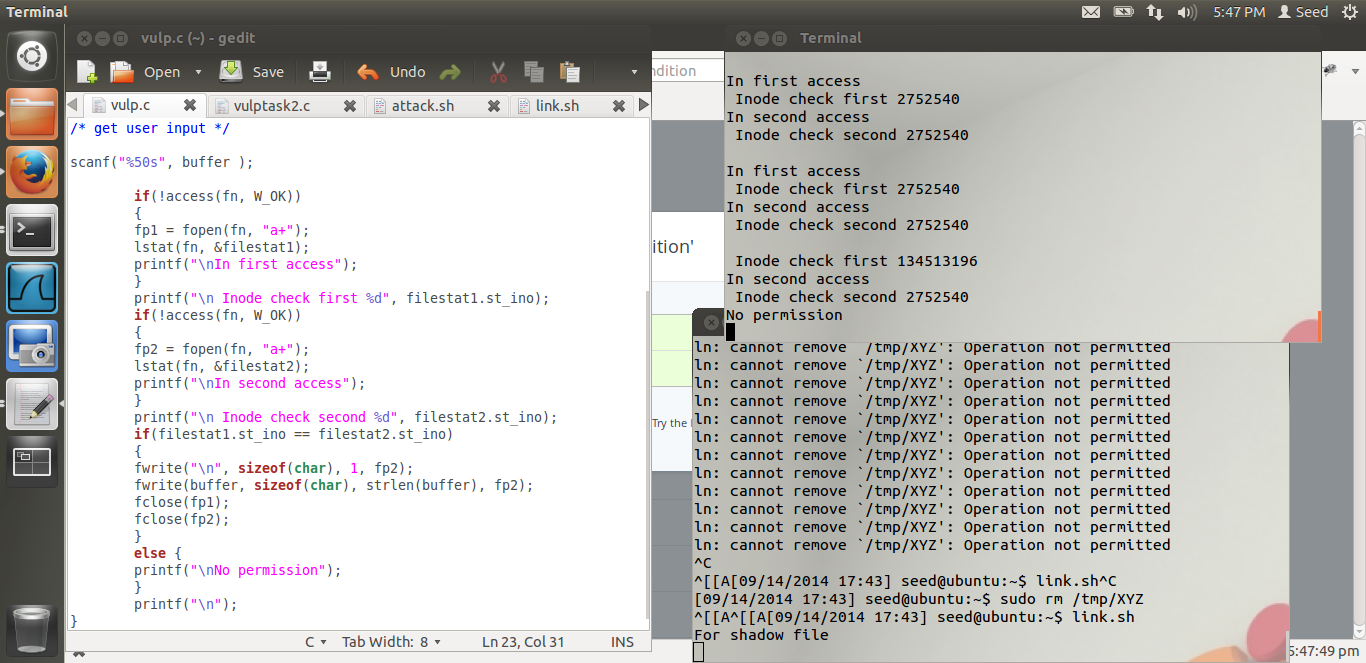
Here, compromised is the name of the new user with a blank password and root privileges, having /bin/bash as the default shell. And 0 as the UID (0 is the UID for root).

1. **Competition Question- Answer with explanation.**A system call is made between the link and unlink calls in the C program, this leads to a forced context switch. However when we run the script and use the ln command as a whole, even though it is not an atomic command, the chances of a context switch are reduced drastically thus making sure that the probability of losing the race condition is lesser.

**Task 2:**

  
Figure 2.1

Failure probability increases

****

**Figure 2.2**

Inode check doesnot allow race condition attack to succeed

**Observations:**

1. On adding more than one access and open commands the time for the attack to succeed increases drastically, the attack succeed only once and took around 15 minutes to succeed. Whereas the attack with just one access and open check took less than a minute to succeed.

With increases in the number of access and open checks the amount of time required for the attack to succeed increased drastically and resulted in dead situation more often than not.

***/\*vulp.c\*/***

***#include <stdio.h>***

***#include <unistd.h>***

***#include <sys/types.h>***

***#include <sys/stat.h>***

***#include <stdlib.h>***

***#include <fcntl.h>***

***int main()***

***{***

***char \* fn = "/tmp/XYZ";***

***char buffer[60];***

***FILE \*fp1, \*fp2;***

***struct stat filestat1, filestat2;***

***/\* get user input \*/***

***scanf("%50s", buffer );***

***if(!access(fn, W\_OK))***

***{***

***fp1 = fopen(fn, "a+");***

***lstat(fn, &filestat1);***

***printf("\nIn first access");***

***}***

***if(!access(fn, W\_OK))***

***{***

***fp2 = fopen(fn, "a+");***

***lstat(fn, &filestat2);***

***printf("\nIn second access");***

***fwrite("\n", sizeof(char), 1, fp2);***

***fwrite(buffer, sizeof(char), strlen(buffer), fp2);***

***fclose(fp1);***

***fclose(fp2);***

***}***

***else {***

***printf("\nNo permission");***

***}***

***printf("\n");***

***}***

1. On adding the inode check using the lstat function, the attack did not succeed and very often the dead condition for this attack would become true, that is, when the access check is performed and the right privileges are obtained and before the unlink(which requires the removal of the linked file) and symlink for the file to a root file can be done the open function is called, the open function opens a file that exists and if the file is not found it creates a file with the given name and opens it in the mode that has been called by the function.
2. We add this code to add protection to the code from race vulnerability,

***if (filestat1.st\_ino == filestat2.st\_ino)***

***{***

***fwrite("\n", sizeof(char), 1, fp2);***

***fwrite(buffer, sizeof(char), strlen(buffer), fp2);***

***fclose(fp1);***

***fclose(fp2);***

***}***

***else***

***{***

***printf("\nNo permission");***

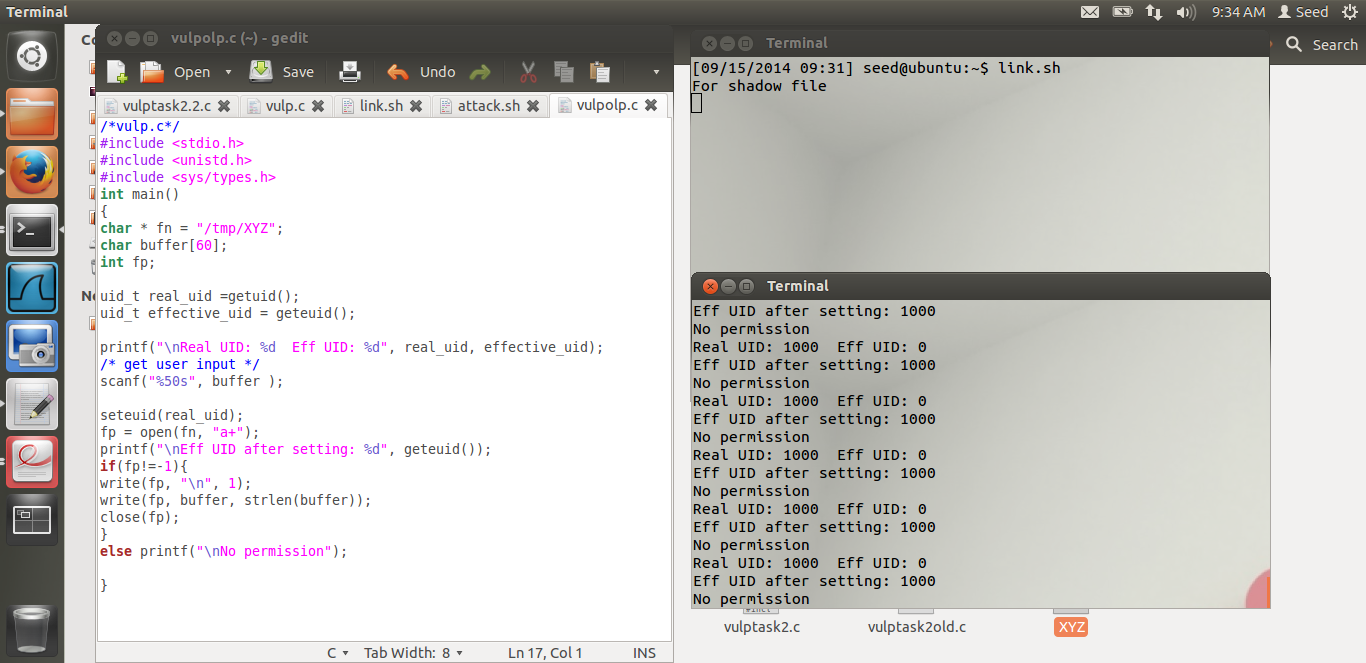
***}***

This code only writes to the given file if the inode value for both the file descriptors is same, if the value is different the program will just print a message saying “No Permission”

**Explanations:**

1. The reason for better security against race vulnerability by increasing the number of checks is that, the window between an access check and an open function call decreases when the number of checks increases thus giving lesser time for the link attack to succeed.
2. The lstat() function returns information about the file or the link that the file pointer points to and provides the information to the buffer that has been passed to it as a parameter. The inode check is an absolute way of removing race condition checks as it allows to set a condition to perform a write only when both the files have the same information, and unless both files are same they cannot have the same information.
3. The code checks if both the files have same inode value and if they do, it allows fir a write, otherwise it doesnot allow the write to happen.

**Task 3:**



Changes in effective user id and real user id as a protection mechanism

**Figure 3.1**

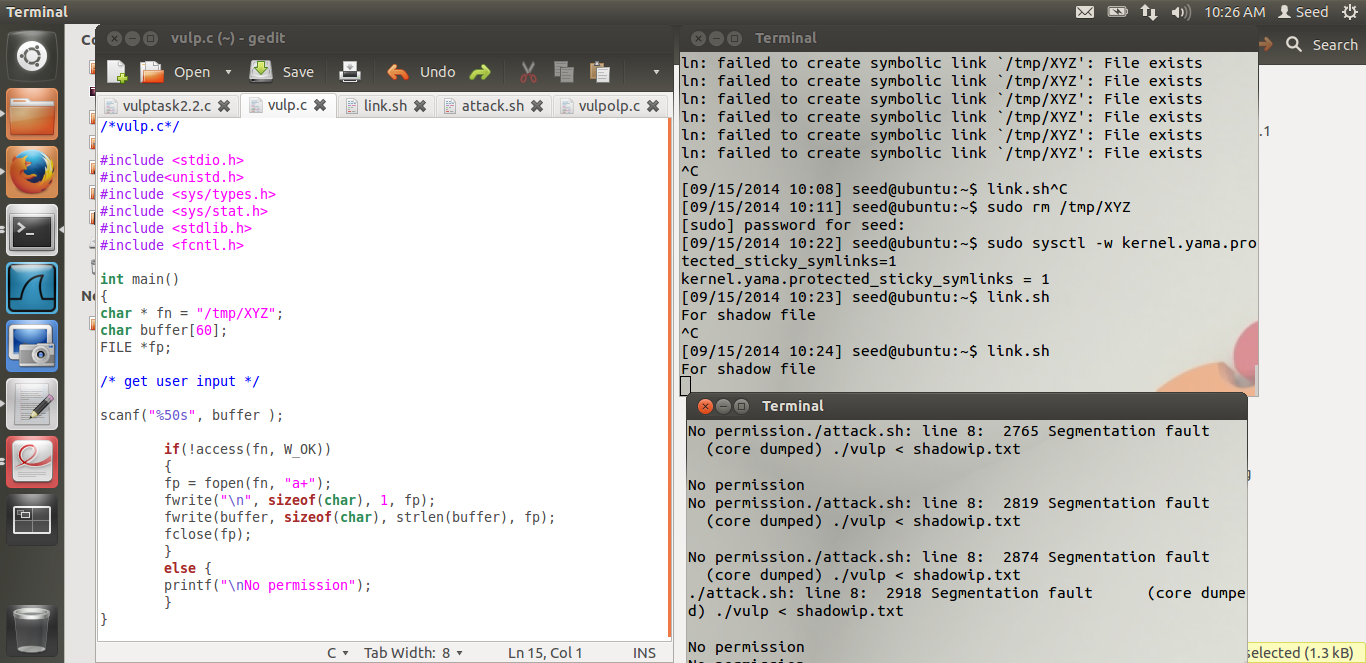
**Observations:**

1. Even after running the program for over 1 hour, there was no success, instead the race condition is lost many times when the attack is being tried, as /tmp/XYZ is created with root as the owner, thus disallowing for the *link.sh* script to work and switch the links in between the */etc/shadow* file and the normal users file.

**Explanations:**

1. Setting the effective user id to the user id of the user running the program takes into account the *Principle of Least Privileges* and sets the user id to the required privilege and does not give complete set-UID root privileges, the root privilege can be returned to the program by calling the *seteuid()* function and passing the effective user id that was stored earlier.

**Task 4:**

****

**Figure 4.1**

**Observations:**

1. On running the following command,

*$ sudo sysctl- w kernel.yama.protected\_sticky\_symlinks=1*

the sticky bit is set to one, symlinks in world-writable sticky directories cannot be resolved if the follower/user of the symlink and the directory owner do not match.

1. Segmentation fault occurs when we try to access the file that does not have the same user and the same owner.

**Explanations:**

1. *Yama* provides control over the symlink-safe procedure of the program. TOCTOU(Time of Check, Time of Use) attacks will not succeed if this bit is turned on, this is a patch that was released to take care of race condition attacks.
2. Segmentation fault occurs because the user that is trying to access the link is not the owner of the link, which is the primary objective of the *sticky\_symlinks* bit.
3. **Questions:**

* **Why does this protection scheme work?**

It works because, the bug generally happens in share, world-writable directories that have the “sticky” bit set by setting the sticky bit on a directory, we can prevent the users from deleting other users’ files. This in effect does not allow non-file owners to follow symlinks that points to the files of other owners

* **Is this a good protection?**

This protection can be argued to be a good protection, as it only changes the behaviour of sticky directories with respect to symlinks. But the protection may lead to weird and random behaviour at runtime because of a sticky bit that users are not aware of being enabled (as understood by *Linus Trovalds*).

* **Limitations of this protection scheme?**

This protection scheme will break applications that rely on multiple users being able to access world writable directories that have symlinks and are not the owner of the symlinks as well as the original file. Especially, in corporate world this type of implementation is more common to provide a layer of abstraction.