

Title:

The Importance of Time:

How Functions in C and Symbolic Links can be Exploited for Attacks with Race Conditions

Abstract:

Lab 6 consisted of exploiting a race-condition vulnerability. We utilized privileged race-condition vulnerability, running attacks in parallel by iterating our attack over and over again as well as exploiting the `fopen()` function in C and `sleep()`. Additionally, we utilized the sticky symlink protection and its race conditions to carry out our attacks. Overall, this lab demonstrated the importance of understanding race conditions and how they can be exploited as well as get in the way of our attacks.

Aim:

The aim of this lab was to gain first-hand experience on race-condition vulnerabilities. Additional topics included sticky symlink protections and the principle of least privilege, which will all be expanded upon further in this report.

Introduction and Background:

Race conditions occur when multiple processes access and manipulate the same data concurrently, and the outcome of the execution depends on the particular order in which the access takes place. If a privileged program has a race-condition

vulnerability, attackers can run a parallel process to “race” against the privileged program, with an intention to change

Method:

The method and tools used in the lab revolved around exploiting linux commands and root privileges, as well as C code, specifically the `fopen()` function as well as implementing multiple linux commands.

We also utilized the `unlink()` and `symlink()` functions to create symbolic links.

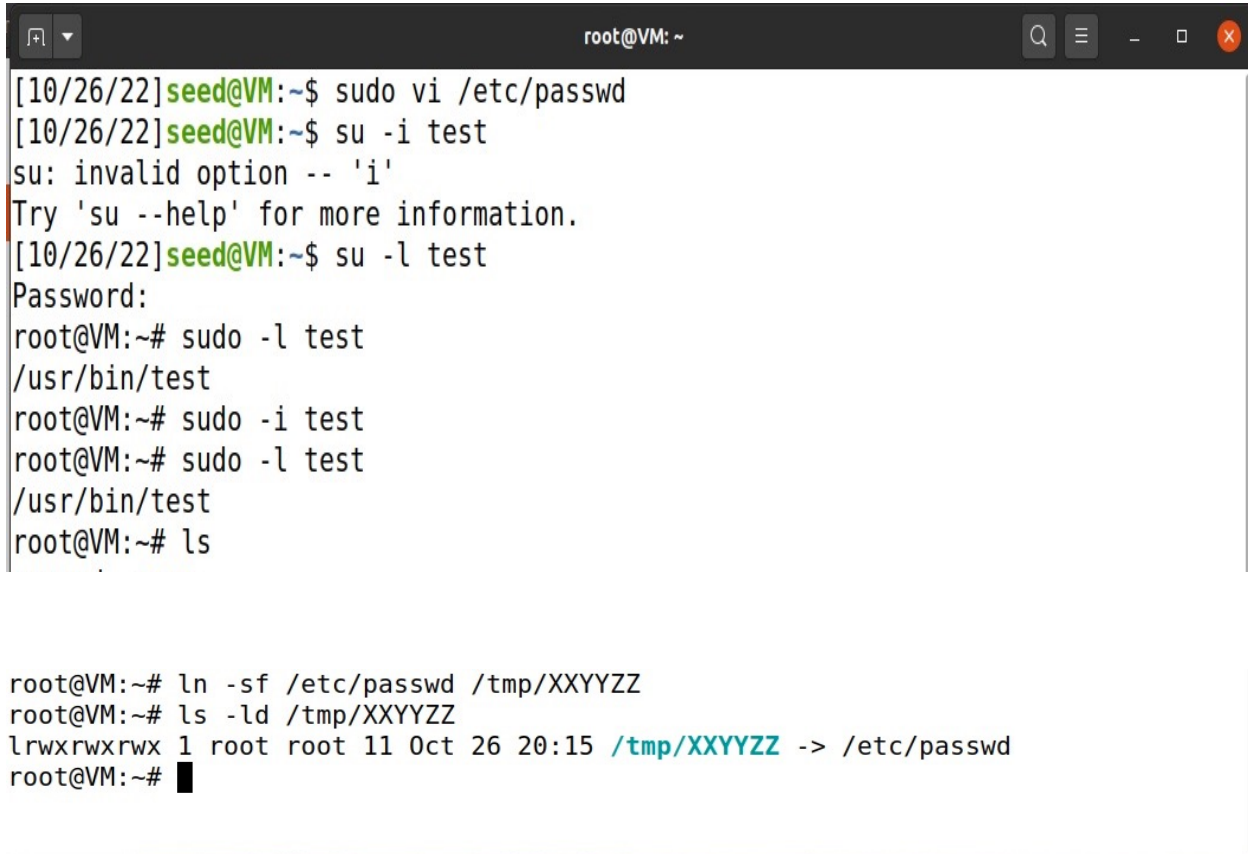
Lastly, utilizing all these commands, we were able to exploit concurrent program running time, changing privileges of the `tmp` directory, and exploit race conditions

Results and Discussion:

The initial setup in this lab included turning off countermeasures using the commands that alter symlinks and allowing root to write files in the `tmp` directory that are owned by others. After using the commands specified, we set up the UID program, `vulp`, owned by root.

Next, we chose to target the password file `/etc/passwd`, which is not writable by normal users. We then attempted to get the one way hash value of the password from the shadow file.

(SCREENSHOT 1: logging in)



```
root@VM: ~
[10/26/22] seed@VM:~$ sudo vi /etc/passwd
[10/26/22] seed@VM:~$ su -i test
su: invalid option -- 'i'
Try 'su --help' for more information.
[10/26/22] seed@VM:~$ su -l test
Password:
root@VM:~# sudo -l test
/usr/bin/test
root@VM:~# sudo -i test
root@VM:~# sudo -l test
/usr/bin/test
root@VM:~# ls

root@VM:~# ln -sf /etc/passwd /tmp/XXYYZZ
root@VM:~# ls -ld /tmp/XXYYZZ
lrwxrwxrwx 1 root root 11 Oct 26 20:15 /tmp/XXYYZZ -> /etc/passwd
root@VM:~#
```

After this step, we started to launch the race condition attack, by simulating a slow machine and using the `fopen()` function in our C code. The slow machine will yield control to the operating system for 10 seconds, allowing us to add a root account to the system. We were able to do this by creating another symbolic link with the previous commands and adding another root account to the tmp file.

(SCREENSHOT 2 and 3: How and carrying out the attack)

```
seed@VM: /tmp
[10/26/22] seed@VM: /tmp$ ls
config-err-ETypEv
_MEI53iX0g
_MEId2ICFL
_MEIJNPLKR
ssh-SXVdcRIA7UzB
systemd-private-252d1a8aa49a454092d725c96dd1725a-colord.service-NMXlKi
systemd-private-252d1a8aa49a454092d725c96dd1725a-fwupd.service-G04hrh
systemd-private-252d1a8aa49a454092d725c96dd1725a-ModemManager.service-GYahlj
systemd-private-252d1a8aa49a454092d725c96dd1725a-switcheroo-control.service-gkiA
gf
systemd-private-252d1a8aa49a454092d725c96dd1725a-systemd-logind.service-wo2qTf
systemd-private-252d1a8aa49a454092d725c96dd1725a-systemd-resolved.service-Nx0MBg
systemd-private-252d1a8aa49a454092d725c96dd1725a-systemd-timesyncd.service-MbMYW
f
systemd-private-252d1a8aa49a454092d725c96dd1725a-upower.service-usvc5f
tracker-extract-files.1000
tracker-extract-files.125
VMwareDnD
XYZ
[10/26/22] seed@VM: /tmp$ ls -ld /tmp/XYZ
lrwxrwxrwx 1 seed seed 9 Oct 26 20:11 /tmp/XYZ -> /dev/null
[10/26/22] seed@VM: /tmp$ vi XYZ
[10/26/22] seed@VM: /tmp$
```

```
[10/26/22] seed@VM: ~/.../Labsetup-4$ ./target_process.sh
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
No permission
```

Lastly, without using the sleep() function in our program, we must also carry out this attack of adding another root user. We use the “ln -s” commands to

make/change symbolic links. Running this multiple times we attempt to get perfect timing to run in parallel as done previously with the sleep() function. Although, our attack was unsuccessful.

Now attempting an improved attack method, we attempt to exploit the race condition of using unlink() and symlink() in our attack program. We exploited this by making unlink and symlink() atomic with a system call.

Q1: Why does the attack in 2C make the attack more reliable?

- The attack in 2C makes the attack more reliable because we make unlink() and symlink() atomic, therefore the race condition we were not exploiting will not prevent us from carrying out our attack and using the victim race condition. Essentially, we can use unlink() and symlink() in this attack without it getting in our own way.

Q2: How do the Ubuntu countermeasures work against these kinds of attacks?

- Ubuntu has built-in protection against race condition attacks. It restricts who can follow a symlink. Additionally, Ubuntu introduces a mechanism that prevents the root from writing to the files in /tmp that are owned by others.

References:

- [1] S. Hameed, *How to create symbolic links in ubuntu*, 01-Jan-1969.
[Online]. Available: https://linuxhint.com/create_symbolic_link_ubuntu/.
[Accessed: 28-Oct-2022].

