

In this section, will discuss about:

- Kernel Exploit
 - Service Exploit
 - Weak File Permissions
 - Sudo
 - Cron Jobs
 - SUID / SGID Executables
 - Password & Keys
 - NFS
-

Kernel Exploit:

- Kernels are the core of any operating system
- Think of it as a layer between application software and the actual computer hardware
- The kernel has complete control over the operating system
- Exploiting a kernel vulnerability can result in execution as the root user

Finding Kernel Exploits


Finding and using kernel exploits is usually a simple process:

1. Enumerate kernel version (`uname -a`)
2. Find matching exploits (Google, ExploitDB, GitHub)
3. Compile and run.

 **Beware, Kernel exploits can often be unstable and may be one-shot or cause a system crash**

GET TOOL - Linux Exploit Suggester 2

<https://github.com/jondonas/linux-exploit-suggester-2>

 **If Kernel is running on Linux debian 2.6.32-5 THINK OF dirty cow CVE-2016-5195**

```
./linux-exploit-suggester-2.pl -k 2.6.32
```

Service Exploit:

- Service are programs that run in the background, accepting input or performing regular tasks
- If vulnerable services are running as root, exploiting them can lead to command execution as root
- Service exploits can be found using Searchsploit, Google and Github, just like with kernel exploits

How to Check:

```
$ ps aux | grep "^root"
```

With any results, try to identify the version number of the program being executed by enumerating program versions:

```
$ <program> --version
```

```
$ <program> -v
```

On debian-like distributions, dpkg can show installed programs and their version:

```
$ dpkg -l | grep <program>
```

On systems that use rpm, the following achieves the same:

```
$ rpm -qa | grep <program>
```

🔗 If mysqld is running on Ver 5.1.73-1

<https://www.exploit-db.com/exploits/1518>

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:

```
$ ssh -R <local-port>:127.0.0.1:<service-port> <username>@<local-machine>
```

The exploit code can now be run on your local machine at whichever port you choose.

Weak File Permissions:

- Certain system files can be taken advantage of to perform privilege escalation if the permissions on them are too weak.
- 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 that way.

Low Hanging Fruit: /etc/shadow

- The /etc/shadow file contains user password hashes, and by default is not readable by any user except for root. (Misconfiguration, if so)
- If we are able to read the contents of the /etc/shadow file, we might not be able to crack the root user's 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.

Low Hanging Fruit: /etc/passwd

- The /etc/passwd historically contain user password hashes
- For backwards compatibility, if the second field of a user row in /etc/passwd contains a password hash, it takes precedent over the hash in /etc/shadow
- If we can write to /etc/passwd, we can easily enter a known password hash for the root user, and then use the su command to switch to the root user
- 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 no 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.

Therefore, it is always worth exploring the file system looking for readable backup files. Some common places include user home directories, the `/` (root) directory, `/tmp` and `/var/backups`

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 disallowed) to run:

```
$ sudo -l
```

Known password:

By far the most obvious privilege escalation with `sudo` is to use `sudo` as it was intended!

If your low privileged user account can use `sudo` unrestricted (eg; you can run any programs) and you know the user's password, privilege escalation is easy, by using the "switch user" (`su`) command to spawn a root shell:

```
$ sudo su
```

Other methods:

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

```
$ sudo -s
```

```
$ sudo -i
```

```
$ sudo /bin/bash
```

```
$ sudo passwd
```

Even if there are no "obvious" methods for escalating privileges, we may be able to use a shell escape sequence.

Shell Escape Sequences:

Even if we are restricted to running certain programs via `sudo`, it is something possible to "escape" the program and spawn a shell.

Since the initial program runs with root privileges, so does the spawned shell.

A list of programs with their shell escape sequences can be found here:

<https://gtfobins.github.io/>

Abusing Intended Functionality:

- If a program doesn't have an escape sequence, it may still be possible to use it to escalate privileges.
- If we can read files owned by root, we may be able to extract useful information (eg; passwords, hashes, keys)

- If we can write to files owned by root, we may be able to insert or modify information.

Environment Variables:

- Programs run through `sudo` can inherit the environment variables from the user's environment.
- In the `/etc/sudoers` config file, if the `env_reset` option 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 user's 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.

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.
-

Cron Jobs:

- Cron jobs are programs or scripts which users can schedule to run at specific times or intervals.
- Cron jobs 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 configuration for cron jobs.
- User crontabs are usually located in `/var/spooled/cron` or `/var/spool/cron/crontabs`
- The system-wide crontab is located at `/etc/crontab`

File Permissions:

- Misconfiguration of file permissions associated with cron jobs can lead to easy privilege escalation.
- 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.

Wildcards:

- When a wildcard character `*` is provided to a command as part of an argument, the shell will first perform *filename expansion* (also known as globbing) on the wildcard
- This process replaces the wildcard with a space-separated list of the file and directory names in 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 executed, it is possible to pass command line options (eg; `-h`, `--help`) to commands by creating files with these names.

The following commands should show how this works:

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


Filenames are not simply restricted to simple options like `-h` or `--help`

In fact we can create filenames that match complex options: `--option=key=value`

GTFOBins (<https://gtfobins.github.io>) can help determine whether a command has command line options which will be useful for our purposes.

SUID/SGID Executables:

- 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 `find` command to locate files with the SUID or SGID bits set:

```
$ find / -type f -a \( -perm -u+s -o -perm -g+s \) -exec ls -l {} \; 2> /dev/null
```

Shell Escape Executables:

- Just as we were able to use shell escape sequences with programs running via `sudo`, we can do the same with SUID / SGID files.
- A list of programs with their shell escape sequences can be found here: <https://gtfobins.github.io/>
- Refer to the previous section on shell escape sequences for how to use them

LD_PRELOAD & LD_LIBRARY_PATH:

You may be thinking: why we can't just use the same `LD_PRELOAD` and `LD_LIBRARY_PATH` environment variable tricks we used with `sudo` privilege escalation?

By default, this is disabled in Linux, due to the obvious security risk it presents!

Both these environment variables get ignored when SUID files are executed.

Known Exploits:

- Certain programs install SUID files to aid their operation
- Just as services which run as root can have vulnerabilities we can exploit for a root shell, so too can these SUID files
- Exploits can be found using Searchsploit, Google, and GitHub, in the same way we find exploits for Kernels and Services

PATH Environment Variable:

- The PATH environment variable contains a list of directories where the shell should try to find programs
- If a program tries to execute another program, but 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

Finding Vulnerable Programs:

- 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 of use

Running `strings` against a file:

```
$ strings /path/to/file
```

Running `strace` against a command:

```
$ strace -v -f -e execve <command> 2>&1 | grep exec
```

Running `ltrace` against a command:

```
$ ltrace <command>
```

Abusing Shell Features #1:

- In some shells (notably 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

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 use 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 a SUID file runs another program via Bash (eg; by using `system()`) these environment variables can be inherited
- 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

Passwords & Keys:

- While it might seem like a long shot, weak password storage and password re-use can be easy ways to escalate privileges
- While the root user's account password is hashed and stored securely 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 others 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 switching to the root user with a discovered password

Configuration Files:

- Many services and programs use configuration (config) files to store settings
- If a service needs to authenticate to something, it might store the credentials in a 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:

- NFS (Network File System) 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 server's export list:
`$ showmount -e <target>`
- Similar 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 an 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
- While this behavior is default, it can be disabled !

no_root_squash:

- no_root_squash 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
-

Golden Trove - Strategy

1. Check your user (`id` , `whoami`)
2. Run Linux Smart Enumeration with increasing levels.
3. Run LinEnum & 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 course, and other Linux PrivEsc cheatsheets online
(eg; <https://blog.g0tmi1k.com/2011/08/basic-linux-privilege-escalation/>)

Strategy:

- Spend some time and read over the results of your enumeration
- If Linux Smart Enumeration level 0 or level 1 finds something interesting, make a note of it
- Avoid rabbit holes by creating a checklist of things you need for the privilege escalation method to work
- Have a quick look around for files in your user's home directory and other common locations
(eg; `/var/backup` , and `/var/logs`)
- If your user has a history file, read it, it may have important information like commands or even passwords
- Try things that don't have many steps first, eg; `sudo` , cron jobs, SUID files
- Have a good look at root processes, enumerate their versions and search for exploits
(eg; `ps aux`)
- Check for internal ports that you might be able to forward to your attacking machine
- If you still don't have root, re-read your full enumeration dumps and highlight anything that seems odd
- This might be a process or file name you aren't familiar with, an "unusual" filesystem configured (on Linux, anything that isn't ext, swap, or tmpfs), or even a username
- At this stage you can also start to think about Kernel Exploits (last resort)

Don't Panic

Privilege Escalation is tricky

Practice makes perfect

Remember: in a exam setting, it might take a while to find the method, but the exam is always intended to be completed within a timeframe. Keep searching !