10
Kernel Hardening and Process Isolation

Although the Linux kernel is already fairly secure by design, there are still a few ways to lock it down even more. It's simple to do, once you know what to look for. Tweaking the kernel can help prevent certain network attacks and certain types of information leaks. (But fear not – you don't have to recompile a whole new kernel to take advantage of this.)

With process isolation, our aim is to prevent malicious users from performing either a vertical or a horizontal privilege escalation. By isolating processes from each other, we can help prevent someone from taking control of either a root user process or a process that belongs to some other user. Either of these types of privilege escalation could help an attacker either take control of a system or access sensitive information.

In this chapter, we'll take a quick tour of the `/proc` filesystem and show you how to configure certain parameters within it to help beef up security. Then, we'll turn to the subject of process isolation and talk about various methods to ensure that processes remain isolated from each other.

In this chapter, we'll cover the following topics:

- Understanding the `/proc` filesystem
- Setting kernel parameters with sysctl
- Configuring the `sysctl.conf` file
- An overview of process isolation
- Control groups
- Namespace isolation
- Kernel capabilities
- SECCOMP and system calls
- Using process isolation with Docker containers

- Sandboxing with Firejail
- Sandboxing with Snappy
- Sandboxing with Flatpak

So, if you're ready and raring, we'll start by looking at the `/proc` filesystem.

Understanding the /proc filesystem

If you `cd` into the `/proc` directory of any Linux distro and take a look around, you'll be excused for thinking that there's nothing special about it. You'll see files and directories, so it looks like it could just be another directory. In reality, though, it's very special. It's one of several different pseudo-filesystems on the Linux system. (The definition of the word pseudo is fake, so you can also think of it as a fake filesystem.)

If you were to pull the primary operating system drive out of a Linux machine and mount it as the secondary drive on another machine, you'll see a `/proc` directory on that drive, but you won't see anything in it. That's because the contents of the `/proc` directory is created from scratch every time you boot a Linux machine, and then it's cleared out every time you shut down the machine. Within `/proc`, you'll find two general classes of information:

- Information about user-mode processes
- Information about what's going on at the kernel-level of the operating system

We'll look at user-mode processes first.

Looking at user-mode processes

If you use the `ls` command within `/proc`, you'll see a whole bunch of directories that have numbers as their names. Here's a partial listing from my CentOS VM:

```
[donnie@localhost proc]$ ls -l
total 0
dr-xr-xr-x. 9 root root 0 Oct 19 14:23 1
dr-xr-xr-x. 9 root root 0 Oct 19 14:23 10
dr-xr-xr-x. 9 root root 0 Oct 19 14:23 11
dr-xr-xr-x. 9 root root 0 Oct 19 14:23 12
dr-xr-xr-x. 9 root root 0 Oct 19 14:23 13
dr-xr-xr-x. 9 root root 0 Oct 19 14:24 1373
dr-xr-xr-x. 9 root root 0 Oct 19 14:24 145
dr-xr-xr-x. 9 root root 0 Oct 19 14:23 15
dr-xr-xr-x. 9 root root 0 Oct 19 14:23 16
```

```
dr-xr-xr-x. 9 root root 0 Oct 19 14:23 17
. . .
. . .
```

Each of these numbered directories corresponds to the **Process ID** (**PID**) number of a user-mode process. On any Linux system, PID 1 is always the init system process, which is the first user-mode process that starts when you boot a machine.

On Debian/Ubuntu systems, the name of PID 1 is `init`. On RHEL/CentOS systems, it's called systemd. Both systems run the systemd init system, but the Debian/Ubuntu folk have chosen to retain the old `init` name for PID 1.

Within each numbered directory, you'll see various files and subdirectories that contain information about a particular running process. For example, in the `1` directory, you'll see what pertains to the init process. Here's the partial listing:

```
[donnie@localhost 1]$ ls -l
ls: cannot read symbolic link 'cwd': Permission denied
ls: cannot read symbolic link 'root': Permission denied
ls: cannot read symbolic link 'exe': Permission denied
total 0
dr-xr-xr-x. 2 root root 0 Oct 19 14:23 attr
-rw-r--r--. 1 root root 0 Oct 19 15:08 autogroup
-r--------. 1 root root 0 Oct 19 15:08 auxv
-r--r--r--. 1 root root 0 Oct 19 14:23 cgroup
--w-------. 1 root root 0 Oct 19 15:08 clear_refs
-r--r--r--. 1 root root 0 Oct 19 14:23 cmdline
-rw-r--r--. 1 root root 0 Oct 19 14:23 comm
. . .
. . .
```

As you can see, there are a few symbolic links that we can't access without root privileges. When we use `sudo`, we can see where the symbolic links point:

```
[donnie@localhost 1]$ sudo ls -l
total 0
dr-xr-xr-x. 2 root root 0 Oct 19 14:23 attr
-rw-r--r--. 1 root root 0 Oct 19 15:08 autogroup
-r--------. 1 root root 0 Oct 19 15:08 auxv
-r--r--r--. 1 root root 0 Oct 19 14:23 cgroup
--w-------. 1 root root 0 Oct 19 15:08 clear_refs
-r--r--r--. 1 root root 0 Oct 19 14:23 cmdline
-rw-r--r--. 1 root root 0 Oct 19 14:23 comm
-rw-r--r--. 1 root root 0 Oct 19 15:08 coredump_filter
-r--r--r--. 1 root root 0 Oct 19 15:08 cpuset
lrwxrwxrwx. 1 root root 0 Oct 19 15:08 cwd -> /
```

```
. . .
. . .
```

You can use the `cat` command to view the contents of some of these items, but not all of them. However, even when you can view the contents, you won't be able to make much sense of what's there, unless you're an operating system programmer. Rather than trying to view the information directly, you're better off using either `top` or `ps`, which pull their information from `/proc` and parse it so that humans can read it.

I'm assuming that most of you are already familiar with `top` and `ps`. For those who aren't, here's the short explanation.

`ps` provides a static display of what's going on with your machine's processes. There are loads of option switches that can show you different amounts of information. My favorite `ps` command is `ps aux`, which provides a fairly complete set of information about each process.

`top` provides a dynamic, constantly changing display of the machine's processes. Some option switches are available, but just invoking `top` without any options is usually all you need.

Next, let's look at the kernel information.

Looking at kernel information

Within the top level of `/proc`, the files and directories that have actual names contain information about what's going on with the Linux kernel. Here's a partial view:

```
[donnie@localhost proc]$ ls -l
total 0
. . .
dr-xr-xr-x. 2 root root 0 Oct 19 14:24 acpi
dr-xr-xr-x. 5 root root 0 Oct 19 14:24 asound
-r--r--r--. 1 root root 0 Oct 19 14:26 buddyinfo
dr-xr-xr-x. 4 root root 0 Oct 19 14:24 bus
-r--r--r--. 1 root root 0 Oct 19 14:23 cgroups
-r--r--r--. 1 root root 0 Oct 19 14:23 cmdline
-r--r--r--. 1 root root 0 Oct 19 14:26 consoles
-r--r--r--. 1 root root 0 Oct 19 14:24 cpuinfo
. . .
```

As with the user-mode stuff, you can use `cat` to look at some of the different files. For example, here's a partial output of the `cpuinfo` file:

```
[donnie@localhost proc]$ cat cpuinfo
processor    : 0
vendor_id    : AuthenticAMD
cpu family    : 16
model        : 4
model name    : Quad-Core AMD Opteron(tm) Processor 2380
stepping    : 2
microcode    : 0x1000086
cpu MHz        : 2500.038
cache size    : 512 KB
physical id    : 0
siblings    : 1
core id        : 0
cpu cores    : 1
. . .
```

Here, you can see the type and speed rating of my CPU, its cache size, and the fact that this CentOS VM is only running on one of the host machine's eight CPU cores. (Doing this on the Fedora host operating system would show information about all eight of the host machine's cores.)

Yes, you did read that right. I really am using an antique, Opteron-equipped HP workstation from 2009. I got it from eBay for a very cheap price, and it runs beautifully with the LXDE spin of Fedora. And the OpenSUSE machine that you'll see mentioned in other parts of this book is the exact same model and came from the same vendor. (So, now you know just how cheap I really am.)

However, for our present purposes, we don't need to go into the nitty-gritty details of everything that's in `/proc`. What's more important to our present discussion is the different parameters that can be set from within `/proc`. For example, within the `/proc/sys/net/ipv4` directory, we can see lots of different items that can be tweaked to change IPv4 network performance. Here's a partial listing:

```
[donnie@localhost ipv4]$ pwd
/proc/sys/net/ipv4
[donnie@localhost ipv4]$ ls -l
total 0
-rw-r--r--. 1 root root 0 Oct 19 16:11 cipso_cache_bucket_size
-rw-r--r--. 1 root root 0 Oct 19 16:11 cipso_cache_enable
-rw-r--r--. 1 root root 0 Oct 19 16:11 cipso_rbm_optfmt
-rw-r--r--. 1 root root 0 Oct 19 16:11 cipso_rbm_strictvalid
dr-xr-xr-x. 1 root root 0 Oct 19 14:23 conf
```

```
-rw-r--r--. 1 root root 0 Oct 19 16:11 fib_multipath_hash_policy
-rw-r--r--. 1 root root 0 Oct 19 16:11 fib_multipath_use_neigh
-rw-r--r--. 1 root root 0 Oct 19 16:11 fwmark_reflect
-rw-r--r--. 1 root root 0 Oct 19 16:11 icmp_echo_ignore_all
. . .
. . .
```

We can use the `cat` command to view each of these parameters, like so:

```
[donnie@localhost ipv4]$ cat icmp_echo_ignore_all
0
[donnie@localhost ipv4]$
```

So, the `icmp_echo_ignore_all` parameter is set to `0`, which means that it's disabled. If I were to ping this machine from another machine, assuming that the firewall is configured to allow that, this machine would respond to the pings. We have several ways to change that if need be. Some of these are as follows:

- `echo` a new value into the parameter from the command line.
- Use the `sysctl` utility from the command line.
- Configure the `/etc/sysctl.conf` file.
- Add a new `.conf` file that contains the new configuration to the `/etc/sysctl.d` directory.
- Run a command from within a shell script.

Let's go ahead and look at these different methods in detail.

Setting kernel parameters with sysctl

The traditional method that you'll see in older Linux textbooks is to `echo` a value into a `/proc` parameter. This doesn't directly work with `sudo`, so you'll need to use the `bash -c` command to force the command to execute. Here, you can see me changing the value for the `icmp_echo_ignore_all` parameter:

```
[donnie@localhost ~]$ sudo bash -c "echo '1' >
/proc/sys/net/ipv4/icmp_echo_ignore_all"
[donnie@localhost ~]$ cat /proc/sys/net/ipv4/icmp_echo_ignore_all
1
[donnie@localhost ~]$
```

With the value set to `1`, this machine will now ignore all ping packets, regardless of how the firewall is configured. Any value you set like this is temporary and will go back to its default setting when you reboot the machine.

Next in the list after this one is the `icmp_echo_ignore_broadcasts` setting, which looks as follows:

```
[donnie@localhost ipv4]$ cat icmp_echo_ignore_broadcasts
1
[donnie@localhost ipv4]$
```

It's already enabled by default, so out of the box, Linux is already immune to **Denial-of-Service** (**DoS**) attacks that involve ICMP broadcast flooding.

Configuring `/proc` parameters with echo is old hat, and personally, I don't like to do it. It's better to use `sysctl`, which is the more modern way of doing business. It's easy to use, and you can read all about it in the `sysctl` man page.

To see a list of all the parameter settings, just do the following:

```
[donnie@localhost ~]$ sudo sysctl -a
abi.vsyscall32 = 1
crypto.fips_enabled = 1
debug.exception-trace = 1
debug.kprobes-optimization = 1
dev.hpet.max-user-freq = 64
dev.mac_hid.mouse_button2_keycode = 97
dev.mac_hid.mouse_button3_keycode = 100
dev.mac_hid.mouse_button_emulation = 0
dev.raid.speed_limit_max = 200000
dev.raid.speed_limit_min = 1000
dev.scsi.logging_level = 0
fs.aio-max-nr = 65536
. . .
. . .
```

To set a parameter, use the `-w` option to write the new value. The trick to this is that the forward slashes in the directory path are replaced by dots, and you ignore the `/proc/sys` part of the path. So, to change the `icmp_echo_ignore_all` value back to `0`, we'll do this:

```
[donnie@localhost ~]$ sudo sysctl -w net.ipv4.icmp_echo_ignore_all=0
net.ipv4.icmp_echo_ignore_all = 0
[donnie@localhost ~]$
```

In this case, the change is permanent because I'm just changing the parameter back to its default setting. Normally, though, any changes we make like this only last until we reboot the machine. Sometimes, that's okay, but sometimes, we might need to make the changes permanent.

Configuring the sysctl.conf file

There are some significant differences between the default configurations of Ubuntu and CentOS. Both use the `/etc/sysctl.conf` file, but on CentOS, that file doesn't have anything except for some explanatory comments. Ubuntu and CentOS both have files with default settings in the `/usr/lib/sysctl.d/` directory, but there are more for CentOS than there are for Ubuntu. On Ubuntu, you'll find other files with default values in the `/etc/sysctl.d` directory. On CentOS, that directory only contains a symbolic link that points back to the `/etc/sysctl.conf` file. Also, you'll find that some things are hardcoded into the Linux kernel and aren't mentioned in any of the configuration files. In true Linux fashion, every distro has a different way of configuring all this, just to ensure that users remain thoroughly confused. But that's okay. We'll try to make sense of it, anyway.

Configuring sysctl.conf – Ubuntu

In the `/etc/sysctl.conf` file on an Ubuntu machine, you'll see lots of comments and a few examples of things that you can tweak. The comments provide good explanations of what the various settings do. So, we'll start with it.

Much of this file contains settings that can help improve networking security. Toward the top of the file, we can see this:

```
# Uncomment the next two lines to enable Spoof protection (reverse-path
filter)
# Turn on Source Address Verification in all interfaces to
# prevent some spoofing attacks
#net.ipv4.conf.default.rp_filter=1
#net.ipv4.conf.all.rp_filter=1
```

A spoofing attack involves a bad actor who sends you network packets with spoofed IP addresses. Spoofing can be used for a few different things, such as DoS attacks, anonymous port scanning, or tricking access controls. These settings, when enabled, cause the operating system to verify if it can reach the source address that's in the packet header. If it can't, the packet is rejected. You may be wondering why this is disabled since it seems like such a good thing. However, this isn't the case: it is enabled in another file. If you look in the `/etc/sysctl.d/10-network-security.conf` file, you'll see it enabled there. So, there's no need to uncomment these two lines.

Next, we can see this:

```
# Uncomment the next line to enable TCP/IP SYN cookies
# See http://lwn.net/Articles/277146/
# Note: This may impact IPv6 TCP sessions too
#net.ipv4.tcp_syncookies=1
```

One form of DoS attack involves sending massive amounts of SYN packets to a target machine, without completing the rest of the three-way handshake. This can cause the victim machine to have lots of half-open network connections, which would eventually exhaust the machine's ability to accept any more legitimate connections. Turning on SYN cookies can help prevent this type of attack. In fact, SYN cookies are already turned on in the `/etc/sysctl.d/10-network-security.conf` file.

Here's the next thing we see:

```
# Uncomment the next line to enable packet forwarding for IPv4
#net.ipv4.ip_forward=1
```

Uncommenting this line would allow network packets to flow from one network interface to another in machines that have multiple network interfaces. Unless you're setting up a router or a Virtual Private Network server, leave this setting as is.

We've been looking at nothing but IPv4 stuff so far. Here's one for IPv6:

```
# Uncomment the next line to enable packet forwarding for IPv6
#  Enabling this option disables Stateless Address Autoconfiguration
#  based on Router Advertisements for this host
#net.ipv6.conf.all.forwarding=1
```

In general, you'll also want to leave this one commented out, as it is now. Disabling Stateless Address Autoconfiguration on machines in an IPv6 environment would mean that you'd need to either set up a DHCPv6 server or set static IPv6 addresses on all hosts.

The next section controls ICMP redirects:

```
# Do not accept ICMP redirects (prevent MITM attacks)
#net.ipv4.conf.all.accept_redirects = 0
#net.ipv6.conf.all.accept_redirects = 0
# _or_
# Accept ICMP redirects only for gateways listed in our default
# gateway list (enabled by default)
# net.ipv4.conf.all.secure_redirects = 1
#
```

Allowing ICMP redirects can potentially allow a **Man-in-the-Middle** *(***MITM***)* attack to be successful. Uncommenting the two lines in the top section of this snippet would completely disable ICMP redirects. The bottom line in the bottom section allows redirects, but only if they come from a trusted gateway. This one is a bit deceiving, because even though this line is commented out, and even though there's nothing about this in any of the other configuration files, secure redirects are actually enabled by default. We can see this by filtering our `sysctl -a` output through `grep`:

```
donnie@ubuntu1804-1:/etc/sysctl.d$ sudo sysctl -a | grep 'secure_redirects'
net.ipv4.conf.all.secure_redirects = 1
net.ipv4.conf.default.secure_redirects = 1
net.ipv4.conf.docker0.secure_redirects = 1
net.ipv4.conf.enp0s3.secure_redirects = 1
net.ipv4.conf.lo.secure_redirects = 1
donnie@ubuntu1804-1:/etc/sysctl.d$
```

Here, we can see that secure redirects are enabled on all network interfaces. But if you're sure that your machine will never get used as a router, it's still best to completely disable ICMP redirects. (We'll do that in just a bit.)

The final networking item in this file involves martian packets:

```
# Log Martian Packets
#net.ipv4.conf.all.log_martians = 1
#
```

Now, if you're as old as I am, you might remember a really silly television show from the '60s called My Favorite Martian – but this setting has nothing to do with that. Martian packets have a source address that normally can't be accepted by a particular network interface. For example, if your internet-facing server receives packets with a private IP address or a loopback device address, that's a martian packet. Why are they called martian packets? Well, it's because of someone's statement that these packets are not of this Earth. Regardless, martian packets can exhaust network resources, so it's good to know about them. You can enable logging for them either by uncommenting the line in the preceding snippet or by placing an override file in the `/etc/sysctl.d` directory. (We'll also do that in just a bit.)

The following snippet is a kernel parameter for the Magic system request key:

```
# Magic system request key
# 0=disable, 1=enable all
# Debian kernels have this set to 0 (disable the key)
# See https://www.kernel.org/doc/Documentation/sysrq.txt
# for what other values do
#kernel.sysrq=1
```

When this parameter is enabled, you can perform certain functions, such as shutting down or rebooting the system, sending signals to processes, dumping process debug information, and several other things, by pressing a sequence of Magic Keys. You would do this by pressing the *Alt* + *SysReq* + command-key sequence. (The *SysReq* key is the *PrtScr* key on some keyboards, while the command-key is the key that invokes some specific command.) A value of `0` for this would completely disable it, and a value of `1` would enable all Magic Key functions. A value greater than 1 would enable only specific functions. In this file, this option appears to be disabled. However, it's actually enabled in the `/etc/sysctl.d/10-magic-sysrq.conf` file. If you're dealing with a server that's locked away in a server room and that can't be remotely accessed from a serial console, this might not be a big deal. However, for a machine that's out in the open or that can be accessed from a serial console, you might want to disable this. (We'll do that in a bit as well.)

The final setting in `/etc/sysctl.conf` prevents the creation of hard links and symbolic links under certain circumstances:

```
# Protected links
#
# Protects against creating or following links under certain conditions
# Debian kernels have both set to 1 (restricted)
# See https://www.kernel.org/doc/Documentation/sysctl/fs.txt
#fs.protected_hardlinks=0
#fs.protected_symlinks=0
```

Under certain circumstances, bad guys could possibly create links to sensitive files so that they can easily access them. Link protection is turned on in the `/etc/sysctl.d/10-link-restrictions.conf` file, and you want to leave it that way. So, don't ever uncomment these two parameters.

That pretty much covers what we have in Ubuntu. Now, let's look at CentOS.

Configuring sysctl.conf – CentOS

On CentOS, the `/etc/sysctl.conf` file is empty, except for a few comments. These comments tell you to look elsewhere for the default configuration files and to make changes by creating new configuration files in the `/etc/sysctl.d` directory.

The default security settings for CentOS are pretty much the same as they are for Ubuntu, except they're configured in different places. For example, on CentOS, the spoof protection (`rp_filter`) parameters and the link protection parameters are in the `/usr/lib/sysctl.d/50-default.conf` file.

By piping a `sysctl -a` command into `grep`, you'll also see that `syncookies` are enabled:

```
[donnie@centos7-tm1 ~]$ sudo sysctl -a | grep 'syncookie'
net.ipv4.tcp_syncookies = 1
[donnie@centos7-tm1 ~]$
```

The same is true for `secure_redirects`:

```
[donnie@centos7-tm1 ~]$ sudo sysctl -a | grep 'secure_redirects'
net.ipv4.conf.all.secure_redirects = 1
net.ipv4.conf.default.secure_redirects = 1
net.ipv4.conf.enp0s3.secure_redirects = 1
net.ipv4.conf.lo.secure_redirects = 1
net.ipv4.conf.virbr0.secure_redirects = 1
net.ipv4.conf.virbr0-nic.secure_redirects = 1
[donnie@centos7-tm1 ~]$
```

Curiously, you won't find any settings for either `secure_redirects` or `syncookies` in any CentOS configuration file. To try to solve this mystery, I did a grep search throughout the whole filesystem for these text strings. Here's part of what I found by searching for `syncookies`:

```
[donnie@centos7-tm1 /]$ sudo grep -ir 'syncookies' *
. . .
. . .
boot/System.map-3.10.0-123.el7.x86_64:ffffffff819ecf8c D
sysctl_tcp_syncookies
boot/System.map-3.10.0-123.el7.x86_64:ffffffff81a5b71c t init_syncookies
. . .
. . .
```

The only places where `grep` finds either the `syncookies` or the `secure_redirects` text strings are in the `System.map` files in the `/boot` directory. So, my best guess is that these values are hardcoded into the kernel so that there's no need to configure them in a `sysctl` configuration file.

Setting additional kernel-hardening parameters

What we've seen so far isn't too bad. Most of the parameters that we've looked at are already set to their most secure values. But is there room for improvement? Indeed there is. You wouldn't know it by looking at any of the configuration files, though. On both Ubuntu and CentOS, quite a few items have default values that aren't set in any of the normal configuration files. The best way to see this is to use a system scanner, such as Lynis.

Lynis is a security scanner that shows lots of information about a system. (We'll cover it in more detail in `Chapter 13`, *Vulnerability Scanning and Intrusion Detection*.) For now, we'll just cover what it can tell us about hardening the Linux kernel.

After you run a scan, you'll see a `[+] Kernel Hardening` section in the screen output. It's fairly lengthy, so here's just part of it:

```
[+] Kernel Hardening
------------------------------------
 - Comparing sysctl key pairs with scan profile
 - fs.protected_hardlinks (exp: 1) [ OK ]
 - fs.protected_symlinks (exp: 1) [ OK ]
 - fs.suid_dumpable (exp: 0) [ OK ]
 - kernel.core_uses_pid (exp: 1) [ OK ]
 - kernel.ctrl-alt-del (exp: 0) [ OK ]
 - kernel.dmesg_restrict (exp: 1) [ DIFFERENT ]
 - kernel.kptr_restrict (exp: 2) [ DIFFERENT ]
 - kernel.randomize_va_space (exp: 2) [ OK ]
 - kernel.sysrq (exp: 0) [ DIFFERENT ]
. . .
. . .
```

Everything that's marked as `OK` is as it should be for the best security. What's marked as `DIFFERENT` should be changed to the suggested `exp:` value that's within the pair of parentheses. (**exp** stands for **expected**.) Let's do that now in a hands-on lab.

Hands-on lab – scanning kernel parameters with Lynis

Lynis is in the normal repositories for Ubuntu and in the EPEL repository for CentOS. It's always a few versions behind what you can get directly from the author's website, but for now, that's okay. When we get to `Chapter 13`, *Vulnerability Scanning and Intrusion Detection*, I'll show you how to get the newest version. Let's get started:

1. Install Lynis from the repository by doing the following for Ubuntu:

```
sudo apt update
sudo apt install lynis
```

   Do the following for CentOS 07:

```
sudo yum install lynis
```

   Do the following for CentOS 08:

```
sudo dnf install lynis
```

2. Scan the system by using the following command:

```
sudo lynis audit system
```

3. When the scan completes, scroll back up to the `[+] Kernel Hardening` section of the output. Copy and paste the sysctl key pairs into a text file. Save it as `secure_values.conf` in your own home directory. The contents of the file should look something like this:

```
 - fs.protected_hardlinks (exp: 1) [ OK ]
 - fs.protected_symlinks (exp: 1) [ OK ]
 - fs.suid_dumpable (exp: 0) [ OK ]
 - kernel.core_uses_pid (exp: 1) [ OK ]
 - kernel.ctrl-alt-del (exp: 0) [ OK ]
 - kernel.dmesg_restrict (exp: 1) [ DIFFERENT ]
 - kernel.kptr_restrict (exp: 2) [ DIFFERENT ]
 - kernel.randomize_va_space (exp: 2) [ OK ]
 - kernel.sysrq (exp: 0) [ DIFFERENT ]
 - kernel.yama.ptrace_scope (exp: 1 2 3) [ DIFFERENT ]
 - net.ipv4.conf.all.accept_redirects (exp: 0) [ DIFFERENT ]
 - net.ipv4.conf.all.accept_source_route (exp: 0) [ OK ]
. . .
. . .
```

4. Use `grep` to send all of the `DIFFERENT` lines to a new file. Name it `60-secure_values.conf`:

```
grep 'DIFFERENT' secure_values.conf > 60-secure_values.conf
```

5. Edit the `60-secure_values.conf` file to convert it into the sysctl configuration format. Set each parameter to the `exp` value that's currently within the pairs of parentheses. The finished product should look something like this:

```
kernel.dmesg_restrict = 1
kernel.kptr_restrict = 2
kernel.sysrq = 0
kernel.yama.ptrace_scope = 1 2 3
net.ipv4.conf.all.accept_redirects = 0
net.ipv4.conf.all.log_martians = 1
net.ipv4.conf.all.send_redirects = 0
net.ipv4.conf.default.accept_redirects = 0
net.ipv4.conf.default.log_martians = 1
net.ipv6.conf.all.accept_redirects = 0
net.ipv6.conf.default.accept_redirects = 0
```

6. Copy the file to the `/etc/sysctl.d` directory:

```
sudo cp 60-secure_values.conf /etc/sysctl.d/
```

7. Reboot the machine to read in the values from the new file:

```
sudo shutdown -r now
```

8. Repeat *step 2*. Most items should now show up with their most secure values. However, you might see a few `DIFFERENT` lines come up. That's okay; just move the lines for those parameters into the main `/etc/sysctl.conf` file and reboot the machine again.

This trick didn't completely work with CentOS 8. No matter what I did, the `net.ipv4.conf.all.forwarding` item remained enabled. However, the other items came out okay.

That's the end of the lab—congratulations!

We've already talked about some of the items that we changed in this procedure. Here's a breakdown of the rest of them:

- `kernel.dmesg_restrict = 1`: By default, any non-privileged user can run the `dmesg` command, which allows the user to view different types of kernel information. Some of this information could be sensitive, so we want to set this parameter to `1` so that only someone with root privileges can use `dmesg`.
- `kernel.kptr_restrict = 2`: This setting prevents `/proc` from exposing kernel addresses in memory. Setting this to `0` completely disables it, while setting it to `1` prevents non-privileged users from seeing the address information. Setting it to `2`, as we have here, prevents anyone from seeing address information, regardless of the person's privilege level. Note, though, that setting this to either `1` or `2` could prevent certain performance monitor programs, such as `perf`, from running. If you absolutely have to do performance monitoring, you might have to set this to `0`. (That's not as bad as it might sound, because having the `kernel.dmesg_restrict = 1` setting in place can help mitigate this issue.)
- `kernel.yama.ptrace_scope = 1 2 3`: This places restrictions on the `ptrace` utility, which is a debugging program that the bad guys can also use. `1` restricts ptrace to only debugging parent processes. `2` means that only someone with root privileges can use ptrace, while `3` prevents anyone from tracing processes with ptrace.

In this section, you learned how to configure various kernel parameters to help lock down your system. Next, we'll lock things down even more by restricting who can view process information.

Preventing users from seeing each others' processes

By default, users can use a utility such as `ps` or `top` to see everyone else's processes, as well as their own. To demonstrate this, let's look at the following partial output from a `ps aux` command:

```
[donnie@localhost ~]$ ps aux
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root          1  0.0  0.7 179124 13752 ?        Ss   12:05   0:03
/usr/lib/systemd/systemd --switched-root --system --deserialize 17
root          2  0.0  0.0      0     0 ?        S    12:05   0:00 [kthreadd]
root          3  0.0  0.0      0     0 ?        I<   12:05   0:00 [rcu_gp]
. . .
. . .
colord 2218 0.0 0.5 323948 10344 ? Ssl 12:06 0:00 /usr/libexec/colord
gdm 2237 0.0 0.2 206588 5612 tty1 Sl 12:06 0:00 /usr/libexec/ibus-engine-
simple
root 2286 0.0 0.6 482928 11932 ? Sl 12:06 0:00 gdm-session-worker [pam/gdm-
password]
donnie 2293 0.0 0.5 93280 9696 ? Ss 12:06 0:00 /usr/lib/systemd/systemd --
user
donnie 2301 0.0 0.2 251696 4976 ? S 12:06 0:00 (sd-pam)
donnie 2307 0.0 0.6 1248768 12656 ? S<sl 12:06 0:00 /usr/bin/pulseaudio --
daemonize=no
. . .
```

Even with just my normal user privileges, I can view processes that belong to the root user and various system users, as well as my own. (And if any of my cats were logged in, I'd also be able to view their processes.) This information can be quite useful to an administrator, but it can also help the bad guys. This information can help Joe or Jane Hacker plan an attack on your system, and it may also even reveal some sensitive information. The best way to deal with this is to mount the `/proc` filesystem with the `hidepid` option. You can do this by adding the following line to the end of the `/etc/fstab` file, like so:

```
proc    /proc    proc    hidepid=2   0    0
```

Then, remount `/proc`, like so:

```
sudo mount -o remount proc
```

Now, any user who doesn't have sudo privileges can only view his or her own processes. (Pretty slick, eh?)

The three values for the hidepid option are as follows:

`0`: This is the default, which allows all users to see each others' processes.
`1`: This allows all users to see other users' process directories within `/proc`. However, users will only be able to `cd` into their own process directories. Also, they'll only be able to see their own process information with `ps` or `top`.
`2`: This hides all other users' process information, including the process directories within `/proc`.

Now that you've seen the inner workings of the `/proc` filesystem and how to configure it for best security, let's look at *process isolation*.

Understanding process isolation

A primary objective of any network intruder is to gain the privileges that are required to perform his or her dirty deeds. This normally involves logging in as a normal user and then performing some sort of privilege escalation. A vertical escalation involves obtaining root privileges, while a horizontal escalation involves gaining the privileges of some other normal user. If the other normal user has any sensitive documents in folders that he or she can access, then a horizontal escalation might be all that the intruder requires. Discretionary Access Control and Mandatory Access Control can help out, but we also want to isolate processes from each other and ensure that processes run with only the lowest possible privileges.

When planning a defense against these types of attacks, consider that the attacks could come from either outsiders or insiders. So, yes, you need to guard against attacks from your organization's own employees.

In this section, we'll look at the various Linux kernel features that facilitate process isolation. Then, we'll look at some cool ways to use these features.

Understanding Control Groups (cgroups)

Control Groups, more commonly called cgroups, were introduced back in 2010 in Red Hat Enterprise Linux 6. Originally, they were just an add-on feature, and a user had to jump through some hoops to manually create them. Nowadays, with the advent of the systemd init system, cgroups are an integral part of the operating system, and each process runs in its own cgroup by default.

With cgroups, processes run in their own kernel space and memory space. Should the need arise, an administrator can easily configure a cgroup to limit the resources that the process can use. This is not only good for security, but also for tuning system performance.

So, what is a cgroup? Well, it's really just a collection of processes that are grouped together for a particular purpose. Here's what you can do with cgroups:

- **Set resource limits**: For each cgroup, you can set resource limits for CPU usage, I/O usage, and memory usage.
- **Perform different accounting functions**: You can measure resource usage for each cgroup, which makes it easy to bill specific customers for the resources that they use.
- **Prioritize resources**: You can set limits on a user who's hogging resources like crazy.
- **Freezing, checkpointing, and restarting**: These functions are handy for troubleshooting. They allow you to stop a process, take a snapshot of the system state, and restore a system state from a backup.

There's not enough space to look at all of these functions, but that's okay. Right now, our primary interest is setting resource limits. With only some minor exceptions, things work the same on CentOS 7, CentOS 8, and Ubuntu 18.04. For now, we'll work with CentOS 8.

By default, each cgroup on the system has no defined resource limits. The first step in defining them is to enable accounting for CPU usage, memory usage, and I/O usage. We could do that by hand-editing the systemd service file for each service that we want to limit, but it's easier to just run a `systemctl` command, like so:

```
sudo systemctl set-property httpd.service MemoryAccounting=1
CPUAccounting=1 BlockIOAccounting=1
```

We've just turned on the accounting functions for the Apache web server on a CentOS 8 machine. (The command would be the same on an Ubuntu machine, except that we would have `apache2.service` instead of `httpd.service`.) Now, when we look in the `/etc/systemd/system.control` directory, we'll see that we've created an `httpd.service.d` directory. Within that directory are the files that turn on our accounting functions:

```
[donnie@localhost httpd.service.d]$ pwd
/etc/systemd/system.control/httpd.service.d
[donnie@localhost httpd.service.d]$ ls -l
total 12
-rw-r--r--. 1 root root 153 Oct 30 15:07 50-BlockIOAccounting.conf
-rw-r--r--. 1 root root 149 Oct 30 15:07 50-CPUAccounting.conf
-rw-r--r--. 1 root root 152 Oct 30 15:07 50-MemoryAccounting.conf
[donnie@localhost httpd.service.d]$
```

Inside each file, we can see two lines that modify the original `httpd.service` file in order to turn on accounting. For example, here's the one for `CPUAccounting`:

```
[donnie@localhost httpd.service.d]$ cat 50-CPUAccounting.conf
# This is a drop-in unit file extension, created via "systemctl set-
property"
# or an equivalent operation. Do not edit.
[Service]
CPUAccounting=yes
[donnie@localhost httpd.service.d]$
```

Now that we've enabled accounting for the Apache service, we can place some resource limits on it. (By default, there are no limits.) Let's say that we want to limit Apache to only 40% of CPU usage and 500 MB of memory usage. We'll set both limits with the following command:

```
[donnie@localhost ~]$ sudo systemctl set-property httpd.service
CPUQuota=40% MemoryLimit=500M
[donnie@localhost ~]$
```

This command created two more files in the `/etc/systemd/system.control/httpd.service.d/` directory:

```
[donnie@localhost httpd.service.d]$ ls -l
total 20
-rw-r--r--. 1 root root 153 Oct 30 15:07 50-BlockIOAccounting.conf
-rw-r--r--. 1 root root 149 Oct 30 15:07 50-CPUAccounting.conf
-rw-r--r--. 1 root root 144 Oct 30 15:18 50-CPUQuota.conf
-rw-r--r--. 1 root root 152 Oct 30 15:07 50-MemoryAccounting.conf
-rw-r--r--. 1 root root 153 Oct 30 15:18 50-MemoryLimit.conf
[donnie@localhost httpd.service.d]$
```

Let's `cat` one of them, just to see the format of the files:

```
[donnie@localhost httpd.service.d]$ cat 50-CPUQuota.conf
# This is a drop-in unit file extension, created via "systemctl set-
property"
# or an equivalent operation. Do not edit.
[Service]
CPUQuota=40%
[donnie@localhost httpd.service.d]$
```

We can allocate resources to other services in the same manner. For example, if this were a **Linux-Apache-MySQL/MariaDB-PHP** (**LAMP**) server, we could allocate a portion of the remaining CPU and memory resources to the PHP service, and the rest to the MySQL/MariaDB service.

LAMP is the bedrock for many popular Content Management Systems, such as WordPress and Joomla.

We can also place resource limits on user accounts. For example, let's limit Katelyn to 20% of CPU usage and 500 MB of memory usage. First, we need to get Katelyn's User ID number. We'll do that with the `id` command:

```
[donnie@localhost ~]$ id katelyn
uid=1001(katelyn) gid=1001(katelyn) groups=1001(katelyn)
[donnie@localhost ~]$
```

So, her UID is `1001`. Let's enable accounting for her and set her limits:

```
[donnie@localhost ~]$ sudo systemctl set-property user-1001.slice
MemoryAccounting=1 CPUAccounting=1 BlockIOAccounting=1

[donnie@localhost ~]$ sudo systemctl set-property user-1001.slice
CPUQuota=20% MemoryLimit=500M

[donnie@localhost ~]
```

If we look in the `/etc/systemd/system.control/user-1001.slice.d` directory, we'll see the same set of files that were created for the httpd service.

I've already mentioned the difference between doing this on CentOS and Ubuntu; that is, certain services have different names on each distro. In this case, the service is `httpd.service` on CentOS and `apache2.service` on Ubuntu. Other than that, things work the same for both Ubuntu 18.04 and CentOS 8.

On CentOS 7, there's no `system.control` directory within the `/etc/systemd` directory. Instead, the `httpd.service.d` directory is created within the `/etc/systemd/system` directory. When I tried to set limits for Katelyn for the first again, again with UID `1001`, CentOS 7 wouldn't allow me to do it until Katelyn logged in to activate her `slice`. Her files were created in the `/run/systemd/system/user-1001.slice.d` directory, which only contains ephemeral runtime files. So, unlike with CentOS 8, the files aren't persistent across reboots. This means that if you need to set user resource limits on CentOS 7, you need to be aware that they'll disappear once you reboot the machine.

There's a lot more to cgroups than what I have space to present here. But that's okay. In this section, we've looked at two ways cgroups can enhance security:

- They provide process isolation.
- Using them to limit resource usage can help prevent DoS attacks.

Next up, we'll take a brief look at the concepts of *namespaces* and *namespace isolation*.

Understanding namespace isolation

Namespaces are a kernel security feature that was introduced in Linux kernel version 2.4.19, all the way back in 2002. A namespace allows a process to have its own set of computer resources that other processes can't see. They're especially handy for times when you might have multiple customers sharing resources on the same server. The processes for each user will have their own namespaces. Currently, there are seven types of namespaces:

- **Mount (mnt)**: This is the original namespace, which was introduced in Linux kernel 2.4.19. At the time, this was the only namespace. This allows each process to have its own root filesystem that no other processes can see, unless you choose to share it. This is a good way of preventing information leakage.
- **UTS**: The UTS namespace allows each process to have its own unique hostname and domain name.
- **PID**: Every running process can have its own set of PID numbers. PID namespaces can be nested so that a parent namespace can see the PIDs of child namespaces. (Note that child namespaces can't see into the parent namespaces.)
- **Network (net)**: This allows you to create a whole virtual network for each process. Each virtual network can have its own subnets, virtual network interfaces, routing tables, and firewalls.
- **Interprocess Communication (ipc)**: This also prevents data leakage by preventing two processes from sharing the same memory space. Each running process can access its own memory space, but other processes will be blocked.
- **Control group (cgroup)**: This namespace hides the identity of the cgroup that a process is a member of.
- **User**: The User namespace allows a user to have different levels of privilege on different processes. For example, a user could have root-level privileges on one process, but only normal-user privileges on another process.

To see these namespaces, just go into any numbered directory within the `/proc` filesystem and view the contents of the `ns` directory. Here's an example from one of my machines:

```
[donnie@localhost ns]$ pwd
/proc/7669/ns
[donnie@localhost ns]$ ls -l
total 0
lrwxrwxrwx. 1 donnie donnie 0 Oct 30 16:16 cgroup -> 'cgroup:[4026531835]'
lrwxrwxrwx. 1 donnie donnie 0 Oct 30 16:16 ipc -> 'ipc:[4026531839]'
lrwxrwxrwx. 1 donnie donnie 0 Oct 30 16:16 mnt -> 'mnt:[4026531840]'
lrwxrwxrwx. 1 donnie donnie 0 Oct 30 16:16 net -> 'net:[4026531992]'
lrwxrwxrwx. 1 donnie donnie 0 Oct 30 16:16 pid -> 'pid:[4026531836]'
lrwxrwxrwx. 1 donnie donnie 0 Oct 30 16:16 pid_for_children ->
'pid:[4026531836]'
lrwxrwxrwx. 1 donnie donnie 0 Oct 30 16:16 user -> 'user:[4026531837]'
lrwxrwxrwx. 1 donnie donnie 0 Oct 30 16:16 uts -> 'uts:[4026531838]'
[donnie@localhost ns]$
```

The sharp-eyed among you will see that there's an extra item in this directory that we haven't covered. The `pid_for_children` item tracks PIDs in child namespaces.

Although it's certainly possible for you to create your own namespaces, you likely never will, unless you're a software developer. Most likely, you'll just use products that have namespace technologies already built into them. Some modern web browsers use namespaces to create a sandbox for each open tab. You can use a product such as Firejail to run a normal program within its own security sandbox. (We'll look at this a bit later.) Then, there's Docker, which uses namespaces to help isolate Docker containers from each other and from the host operating system.

We've just had a high-level overview of what namespaces are all about. Next, let's look at kernel capabilities.

Understanding kernel capabilities

When you perform a `ps aux` command — or a `sudo ps aux` command if you've mounted `/proc` with the `hidepid=1` or `hidepid=2` option — you'll see many processes that are owned by the root user. This is because these processes have to access some sort of system resource that unprivileged users can't access. However, having services run with full root privileges can be a bit of a security problem. Fortunately, there are some ways to mitigate that.

For example, any web server service, such as Apache or Nginx, needs to start with root privileges in order to bind to ports `80` and `443`, which are privileged ports. However, both Apache and Nginx mitigate this problem by either dropping root privileges once the service has started or by spawning child processes that belong to a non-privileged user. Here, we can see that the main Apache process spawns child processes that belong to the non-privileged `apache` user:

```
[donnie@centos7-tm1 ~]$ ps aux | grep http
root 1015 0.0 0.5 230420 5192 ? Ss 15:36 0:00 /usr/sbin/httpd -DFOREGROUND
apache 1066 0.0 0.2 230420 3000 ? S 15:36 0:00 /usr/sbin/httpd -DFOREGROUND
apache 1067 0.0 0.2 230420 3000 ? S 15:36 0:00 /usr/sbin/httpd -DFOREGROUND
apache 1068 0.0 0.2 230420 3000 ? S 15:36 0:00 /usr/sbin/httpd -DFOREGROUND
apache 1069 0.0 0.2 230420 3000 ? S 15:36 0:00 /usr/sbin/httpd -DFOREGROUND
apache 1070 0.0 0.2 230420 3000 ? S 15:36 0:00 /usr/sbin/httpd -DFOREGROUND
donnie 1323 0.0 0.0 112712 964 pts/0 R+ 15:38 0:00 grep --color=auto http
[donnie@centos7-tm1 ~]$
```

But not all software can do this. Some programs are designed to run with root privileges all the time. For some cases — not all, but some — you can fix that by applying a kernel capability to the program executable file.

Capabilities allow the Linux kernel to divide what the root user can do into distinct units. Let's say that you've just written a cool custom program that needs to access a privileged network port. Without capabilities, you'd either have to start that program with root privileges and let it run with root privileges or jump through the hoops of programming it so that it can drop root privileges once it's been started. By applying the appropriate capability, a non-privileged user would be able to start it, and it would run with only the privileges of that user. (More about that later.)

There are too many capabilities to list here (there's about 40 in all), but you can see the full list by using the following command:

```
man capabilities
```

Returning to our previous example, let's say that we need to use Python to set up a very primitive web server that any non-privileged user can start. (We'll do this with Python 2, because it doesn't work with Python 3.) For now, we'll do this on a CentOS 8 machine.

The names of the Python packages and executable files are different between CentOS 7, CentOS 8, and Ubuntu. I'll show you all three sets of commands when we get to the hands-on lab.

The command for running a simple Python web server is as follows:

```
python2 -m SimpleHTTPServer 80
```

However, this won't work because it needs to bind to port `80`, which is the privileged port that's normally used by web servers. At the bottom of the output from this command, you'll see the problem:

```
socket.error: [Errno 13] Permission denied
```

Prefacing the command with `sudo` will fix the problem and allow the web server to run. However, we don't want that. We'd much rather allow non-privileged users to start it, and we'd much rather have it run without root user privileges. The first step in fixing this is to find the Python executable file, like so:

```
[donnie@localhost ~]$ which python2
/usr/bin/python2
[donnie@localhost ~]$
```

Most times, the `python` or `python2` command is a symbolic link that points to another executable file. We'll verify that with a simple `ls -l` command:

```
[donnie@localhost ~]$ ls -l /usr/bin/python2
lrwxrwxrwx. 1 root root 9 Oct  8 17:08 /usr/bin/python2 -> python2.7
[donnie@localhost ~]$
```

So, the `python2` link points to the `python2.7` executable file. Now, let's see if there are any capabilities assigned to this file:

```
[donnie@localhost ~]$ getcap /usr/bin/python2.7
[donnie@localhost ~]$
```

No output means that there are none. When we consult the capabilities man page, we'll find that the `CAP_NET_BIND_SERVICE` capability seems to be what we need. The one-line description for it is: bind a socket to internet domain privileged ports (port numbers less than `1024`). Okay; that sounds good to me. So, let's set that on the `python2.7` executable file and see what happens. Since we used `getcap` to look at the file capabilities, you can probably guess that we'll use `setcap` to set a capability. (And you'd be correct.) Let's do that now:

```
[donnie@localhost ~]$ sudo setcap 'CAP_NET_BIND_SERVICE+ep'
/usr/bin/python2.7
[sudo] password for donnie:
[donnie@localhost ~]$ getcap /usr/bin/python2.7
/usr/bin/python2.7 = cap_net_bind_service+ep
[donnie@localhost ~]$
```

The `+ep` at the end of the capability name means that we're adding the capability as effective (activated) and permitted. Now, when I try to run this web server with just my own normal privileges, it will work just fine:

```
[donnie@localhost ~]$ python2 -m SimpleHTTPServer 80
Serving HTTP on 0.0.0.0 port 80 ...
```

When I use Firefox on my host machine to connect to this server, I will see a file and directory listing of everything that's in my home directory:

Linux capabilities can also be quite useful in other ways. On any Linux system, the ping utility needs root privileges in order to craft the network packets that it needs to do its job. However, everybody and his brother can use ping as just a normal user. If you look at the ping executable file, you'll see that two capabilities have been assigned to it by the Linux maintainers:

```
[donnie@localhost ~]$ getcap /usr/bin/ping
/usr/bin/ping = cap_net_admin,cap_net_raw+p
[donnie@localhost ~]$
```

As cool as all this seems, there are some downsides:

- Trying to figure out exactly which capabilities a program needs isn't always straightforward. In fact, it can require a good bit of experimentation before you can get things right.
- Setting capabilities isn't a cure-all. A lot of times, you'll see that setting a specific capability still won't allow the program to do what you need it to do. Indeed, there may not even be a capability that will allow the program to function without root privileges as you want it to.
- Performing a system update could replace executable files that you assigned capabilities to. (With ping, we don't have to worry about this since the capabilities were set by the Linux maintainers.)

Okay, so there's a good chance that you might never actually have to set any capabilities. However, this is one of the tools that my IoT Security client uses to help lock down IoT devices, so this does have a practical use. And besides, capabilities are a building block for some of the technologies that we'll look at a bit later.

Hands-on lab – setting a kernel capability

For this lab, you'll allow a normal user to run a Python web server. You can use any of your virtual machines. Let's get started:

1. If Apache is installed on your virtual machine, ensure that it's stopped for Ubuntu:

   ```
   sudo systemctl stop apache2
   ```

   For CentOS, do the following:

   ```
   sudo systemctl stop httpd
   ```

2. Install Python 2 for Ubuntu:

   ```
   sudo apt install python
   ```

   For CentOS 7, do the following:

   ```
   sudo yum install python
   ```

   For CentOS 8, do the following:

   ```
   sudo dnf install python2
   ```

3. From within your own home directory, attempt to start the Python `SimpleHTTPServer` with just your normal user privileges, and note the error message on Ubuntu and CentOS 7:

```
python -m SimpleHTTPServer 80
```

   On CentOS 8, you'll see the following:

```
python2 -m SimpleHTTPServer 80
```

4. See if any capabilities are set on the Python executable file on CentOS 7:

```
getcap /usr/bin/python2
```

   On Ubuntu and CentOS 8, do the following:

```
getcap /usr/bin/python2.7
```

5. Set the `CAP_NET_BIND_SERVICE` capability on the Python executable file on CentOS 7:

```
sudo setcap 'CAP_NET_BIND_SERVICE+ep' /usr/bin/python2
```

   On Ubuntu and CentOS 8, do the following:

```
sudo setcap 'CAP_NET_BIND_SERVICE+ep' /usr/bin/python2.7
```

6. Repeat *Steps 3* and *4*. This time, it should work.
7. Ensure that port 80 is open on the virtual machine firewall and use your host machine's web browser to access the server.
8. Shut down the web server using *Ctrl + C*.
9. View the capabilities that have been assigned to the ping executable:

```
getcap /usr/bin/ping
```

10. Review the capabilities of the man page, especially the part about the various capabilities that are there.

That's the end of the lab – congratulations!

So far, you've seen how to set file capabilities and what they can and can't do for you. Next, we'll look at how to control system calls.

Understanding SECCOMP and system calls

Multiple system calls, or syscalls, happen pretty much every time you run any command on a Linux machine. Each syscall takes a command from a human user and passes it to the Linux kernel. This tells the Linux kernel that it needs to perform some sort of privileged action. Opening or closing files, writing to files, or changing file permissions or ownership are just a few of the actions that require making some sort of a syscall. There are approximately 330 syscalls built into the Linux kernel. I can't say exactly how many, because new syscalls get added from time to time. Apart from this, syscalls differ between the various CPU architectures. So, an ARM CPU won't have exactly the same set of syscalls as an x86_64 CPU. The best way to see the list of syscalls that are available on your machine is to use the following command:

```
man syscalls
```

Note that each individual syscall has its own man page.

To get an idea of how this works, here's the `strace` command, which shows the syscalls that get made by a simple `ls` command:

```
[donnie@localhost ~]$ strace -c -f -S name ls 2>&1 1>/dev/null | tail -n +3
| head -n -2 | awk '{print $(NF)}'
access
arch_prctl
brk
close
execve
. . .
. . .
set_robust_list
set_tid_address
statfs
write
[donnie@localhost ~]$
```

In all, 22 syscalls are made from just doing `ls`. (Due to formatting restrictions, I can't show all of them here.)

Secure Computing (**SECCOMP**), originally created for the Google Chrome web browser, allows you to either enable just a certain subset of syscalls that you want for a process to use or disable certain syscalls that you want to prevent a process from using. Unless you're a software developer or a Docker container developer, you probably won't be working with this directly all that much. However, this is yet another building block for the technologies that are used daily by normal humans.

Next, let's put all this cool stuff into perspective by looking at how it's used in real life.

Using process isolation with Docker containers

Container technology has been around for quite some time, but it took Docker to make containers popular. Unlike a virtual machine, a container doesn't contain an entire operating system. Rather, a container contains just enough of an operating system to run applications in their own private sandboxes. Containers lack their own operating system kernels, so they use the kernel of the host Linux machine. What makes containers so popular is that you can pack a lot more of them onto a physical server than you can with virtual machines. So, they're great for cutting the cost of running a data center.

Docker containers use the technologies that we've covered in this chapter. Kernel capabilities, cgroups, namespaces, and SECCOMP all help Docker containers remain isolated from both each other and from the host operating system, unless we choose otherwise. By default, Docker containers run with a reduced set of capabilities and syscalls, and Docker developers can reduce all that even more for the containers that they create.

I can't go into the nitty-gritty details about how all this works in Docker because it would require explaining the development process for Docker containers. That's okay, though. In this section, you'll understand what to look out for if someone wants to deploy Docker containers in your data center.

But as good as all that sounds, Docker security is far from perfect. As I demonstrated in `Chapter 9`, *Implementing Mandatory Access Control with SELinux and AppArmor*, any non-privileged member of the `docker` group can mount the root filesystem of the host machine in a container of his or her own creation. The normally non-privileged member of the `docker` group has root privileges within the container, and those privileges extend to the mounted root filesystem of the host machine. In the demo, I showed you that only an effective Mandatory Access Control system, specifically SELinux, could stop Katelyn from taking control of the entire host machine.

To address this rather serious design flaw, the developers at Red Hat created their own Docker replacement. They call it `podman-docker`, and it's available in the RHEL 8 and CentOS 8 repositories. Security is much improved in `podman-docker`, and the type of attack that I demonstrated for you doesn't work with it, even without SELinux. Personally, I wouldn't even consider anything other than RHEL 8 or CentOS 8 for running containers. (And no, the Red Hat folk aren't paying me to say this.)

Now that I've given you a high-level overview of how the process isolation technologies are used in Docker, let's look at how they're used in technologies that a mere mortal is more likely to use. We'll begin with Firejail.

Sandboxing with Firejail

Firejail uses namespaces, SECCOMP, and kernel capabilities to run untrusted applications in their own individual sandboxes. This can help prevent data leakage between applications, and it can help prevent malicious programs from damaging your system. It's in the normal repositories for Debian and its offspring, which include Raspbian for Raspberry Pi devices and probably every member of the Ubuntu family. On the Red Hat side, it's in the Fedora repositories, but not in the CentOS repositories. So, for CentOS, you'd have to download the source code and compile it locally. Firejail is meant for use on single-user desktop systems, so we'll need to use a desktop version of Linux. To make things easy, I'll use Lubuntu, which is Ubuntu with the LXDE desktop, instead of the Gnome 3 desktop.

Whenever I can choose between the Gnome 3 desktop and something else, you'll always see me go with something else. To me, Gnome 3 is like the Windows 8 of the Linux world, and you know how everyone liked to hate on Windows 8. However, if you like Gnome 3, more power to you, and I won't argue with you.

Before we get too far along, let's consider some of the use cases for Firejail:

- You want to make doubly sure that your web browser doesn't leak sensitive information when you access your bank's web portal.
- You need to run untrusted applications that you've downloaded from the internet.

To install Firejail on your Debian/Ubuntu/Raspbian machine, use the following command:

```
sudo apt update
sudo apt install firejail
```

This installs Firejail, along with a whole bunch of profiles for different applications. When you invoke an application with Firejail, it will automatically load the correct profile for that application, if one exists. If you invoke an application that doesn't have a profile, Firejail will just load a generic one. To see the profiles, `cd` into `/etc/firejail` and take a look:

```
donnie@donnie-VirtualBox:/etc/firejail$ ls -l
total 1780
-rw-r--r-- 1 root root   894 Dec 21  2017 0ad.profile
-rw-r--r-- 1 root root   691 Dec 21  2017 2048-qt.profile
-rw-r--r-- 1 root root   399 Dec 21  2017 7z.profile
-rw-r--r-- 1 root root  1414 Dec 21  2017 abrowser.profile
-rw-r--r-- 1 root root  1079 Dec 21  2017 akregator.profile
-rw-r--r-- 1 root root   615 Dec 21  2017 amarok.profile
-rw-r--r-- 1 root root   722 Dec 21  2017 amule.profile
-rw-r--r-- 1 root root   837 Dec 21  2017 android-studio.profile
. . .
. . .
```

To easily count the number of profiles, use the following command:

```
donnie@donnie-VirtualBox:/etc/firejail$ ls -l | wc -l
439
donnie@donnie-VirtualBox:/etc/firejail$
```

Subtracting the `total 1780` line from the top of the output gives us a total of 438 profiles.

The simplest way to use Firejail is to preface the name of the application you want to run with `firejail`. Let's start with Firefox:

```
firejail firefox
```

Now, the main problem with Firejail is that it doesn't work well consistently. About a year or so ago, a client had me do a writeup about Firejail, and I got it to mostly work on my Fedora workstation and on my Raspberry Pi with Raspbian. But even with the programs that it did work with, I lost some important functionality. For example, when running a web browser with Firejail on my Fedora machine, I wasn't able to watch videos on several different sites, including YouTube. Dropbox and Keepass didn't work at all under Firejail, even though there are specific profiles for both of them.

And now, on my Lubuntu virtual machine, running Firefox under Firejail just gives me a blank browser page, regardless of where I try to surf. So, I installed `chromium-browser` and tried it. So far, it's working much better, and I can even watch YouTube videos with it. Then, I installed LibreOffice, and so far, it seems to run fine with Firejail.

Among the many options that Firejail offers is the option to ensure that programs run either without any kernel capabilities enabled or with just the capabilities that you specify. Something that the man page recommends is to drop all capabilities for any programs that don't require root privileges. So, for Chromium, we'd do the following:

```
firejail --caps.drop=all chromium-browser
```

So, what if you just want to start your applications from the Start menu, the way that you normally would, but still have Firejail protection? For that, you can use the following command:

```
sudo firecfg
```

This command creates symbolic links in the `/usr/local/bin` directory for each program that has a Firejail profile. They look something like this:

```
donnie@donnie-VirtualBox:/usr/local/bin$ ls -l
total 0
lrwxrwxrwx 1 root root 17 Nov 14 18:14 audacious -> /usr/bin/firejail
lrwxrwxrwx 1 root root 17 Nov 14 18:14 chromium-browser ->
/usr/bin/firejail
lrwxrwxrwx 1 root root 17 Nov 14 18:14 evince -> /usr/bin/firejail
lrwxrwxrwx 1 root root 17 Nov 14 18:14 file-roller -> /usr/bin/firejail
lrwxrwxrwx 1 root root 17 Nov 14 18:14 firefox -> /usr/bin/firejail
lrwxrwxrwx 1 root root 17 Nov 14 18:14 galculator -> /usr/bin/firejail
. . .
. . .
```

If you find that a program doesn't work under Firejail, just go into `/usr/local/bin` and delete the link for it.

Now, you'll want to be aware of a very curious thing with the Firejail documentation. In both the Firejail man page and on the main page of the Firejail website, it says that you can use Firejail to sandbox desktop applications, server applications, and user login sessions. However, if you click on the **Documentation** tab of the Firejail website, it says that Firejail is only meant for single-user desktop systems. That's because, in order to do its job, the Firejail executable has to have the SUID permission bit set. The Firejail developers consider it a security risk to allow multiple users to access a machine with this SUID program.

All right, that's enough talk. Let's get some practice in.

Hands-on lab – using Firejail

For this lab, you'll create a virtual machine with your favorite flavor of desktop Ubuntu. Let's get started:

1. Create a virtual machine with your favorite Ubuntu flavor. To use Lubuntu, as I do, just use this the following download link: `http://cdimage.ubuntu.com/lubuntu/releases/18.04/release/lubuntu-18.04.3-desktop-amd64.iso`.
2. Update the VM using the following command:

```
sudo apt update
sudo dist-upgrade
```

   Then, reboot the machine.

3. Install Firejail, LibreOffice, and Chromium:

```
sudo apt install firejail libreoffice chromium-browser
```

4. In one Terminal window, start Chromium without any kernel capabilities:

```
firejail --caps.drop=all chromium-browser
```

5. Surf to various websites to see if everything works as it should.
6. In another Terminal window, start LibreOffice, also without any capabilities:

```
firejail --caps.drop=all libreoffice
```

7. Create the various types of LibreOffice documents and try out various LibreOffice functions to see how much still works properly.
8. Shut down both Chromium and LibreOffice.
9. Configure Firejail so that it automatically sandboxes every application you start, even if you do this from the normal Start menu:

```
sudo firecfg
```

10. Look at the symbolic links that were created:

```
ls -l /usr/local/bin
```

11. Try to open Firefox from the normal menu. Unless things have been fixed since I wrote this, you should see nothing but blank browser pages. So, shut down Firefox.

12. Okay; so you won't be able to sandbox Firefox. To be able to run Firefox without Firejail, just delete its symbolic link from the `/user/local/bin` directory, like so:

```
sudo rm /usr/local/bin/firefox
```

13. Try to run Firefox again. You should see that it starts normally.

You've completed this lab – congratulations!

There are a lot more Firejail options than what I can show you here. For more information, see the Firejail man page and the documentation on the Firejail website.

So far, you've seen both the good and the bad of using Firejail. Next up, we'll look at a couple of universal packaging systems for Linux.

Sandboxing with Snappy

In the Windows world and the Mac world, operating systems and the applications that they can run are sold independently of each other. So, you buy a computer that runs either Windows or macOS, and then you buy the applications separately. When it comes to doing updates, you have to update the operating system, and then update each application separately.

In the Linux world, most applications that you'll ever need are in the repositories of your Linux distro. To install an application, you just use your distro's package management utility – apt, yum, dnf, or whatever else – to install it. However, this has turned out to be both a blessing and a curse. It does make it easier to keep track of your applications and to keep the latest bug fix and security updates installed. But unless you're running a rolling release distro such as Arch, the application packages will become out of date before your Linux distro's end of life. That's because distro maintainers use application versions that are current when the distro version is released, and they don't upgrade to new application versions until the next version of the distro is released. This also makes things hard for application developers, because each family of Linux distros uses its own packaging format. So, wouldn't it be nice to have a universal packaging format that works across all Linux distros, and that can be kept up to date with ease?

Universal packaging began several years ago with AppImage packages. However, they never really caught on that well, and they don't provide any sandboxing features. So, this is all I'll say about them.

Next came Ubuntu's Snappy system, which allows developers to create snap packages that are supposed to run on any system that the Snappy system can be installed on. Each snap application runs in its own isolated sandbox, which helps protect the system from malicious programs. Since snap applications can be installed without root privileges, there's less of a need to worry about a malicious program using root privileges to do things that it shouldn't do. Each snap package is a self-contained unit, which means you don't have to worry about installing dependencies. You can even create snap packages for servers that contain multiple services. The snapd daemon constantly runs in the background, automatically updating both itself and any installed snap applications.

As good as this all sounds, there are a couple of things about Snappy that make it a bit controversial. First, the Ubuntu folk refuse to release the source code for the Snappy application server. So, it's not possible to look at the source code, and it's not possible to set up your own local Snappy server. If you develop snap packages and want to deploy them, even if this is just on your own local network, you have no choice but to use the central snap package portal that's run by Canonical, Ubuntu's parent company. This does fly in the face of software freedom, which the whole GNU/Linux ecosystem is supposed to represent. However, the Canonical folk do it this way to verify the security of snap packages that get served out.

Secondly, even though snap packages are sandboxed in order to protect the system, other weird things can happen. Soon after the Snappy system came online, a package developer got caught sneaking some Monero mining software into one of his packages. Although he just wanted to monetize his efforts and meant no harm, it's still not good to sneak that sort of thing into your packages without telling your potential customers. After that, the Canonical folk stepped up their efforts to scan packages that get uploaded to the portal, in order to prevent that sort of thing from happening again.

And then, there's the matter of user control. The user has zero control over when packages get updated. The snapd daemon will update your installed packages, whether or not you really want it to.

And lastly, the practice of making each snap package a self-contained unit increases disk space usage. Each package contains all the linked libraries that its application uses, which means that you might have multiple packages that all use the same libraries. This isn't necessarily that big of a deal with today's modern, large capacity hard drives, but it could be a problem if you're low on disk space.

If you're running Ubuntu, you might have the Snappy service running already. At some point (I forget when), the Ubuntu folk started including Snappy in a default installation of Ubuntu Server. However, it's not installed on my Lubuntu 18.04 virtual machine. To install it, I would use the following command:

```
sudo apt update
sudo apt install snapd
```

Snappy is also available in the repositories of many non-Ubuntu distros. It's in the normal repositories for Fedora, and in the EPEL repositories for Red Hat and CentOS.

So, how can Snappy be useful to a busy administrator? Well, let's say that your pointy-haired boss has just told you to set up a Nextcloud server so that employees can have a central place to store their documents. However, you're in a time crunch, and you don't want to jump through the hoops of setting up all the individual components of a LAMP system. No problem – just install a snap. First, let's see what's available:

```
donnie@ubuntu1804-1:~$ snap search nextcloud
Name Version Publisher Notes Summary
nextcloud 16.0.5snap3 nextcloud√ - Nextcloud Server - A safe home for all
your data
spreedme 0.29.5snap1 nextcloud√ - Spreed.ME audio/video calls and
conferences feature for the Nextcloud Snap
onlyoffice-desktopeditors 5.4.1 onlyoffice√ - A comprehensive office suite
for editing documents, spreadsheets and presentations
qownnotes 19.11.13 pbek - Plain-text file markdown note taking with
Nextcloud / ownCloud integration
nextcloud-port8080 1.01 arcticslyfox - Nextcloud Server
. . .
. . .
```

There are quite a few choices. We can use the `info` option to narrow things down a bit:

```
donnie@ubuntu1804-1:~$ snap info nextcloud
name: nextcloud
summary: Nextcloud Server - A safe home for all your data
publisher: Nextcloud√
contact: https://github.com/nextcloud/nextcloud-snap
license: AGPL-3.0+
description: |
 Where are your photos and documents? With Nextcloud you pick a server of
your choice, at home, in
 a data center or at a provider. And that is where your files will be.
Nextcloud runs on that
 server, protecting your data and giving you access from your desktop or
mobile devices. Through
 Nextcloud you also access, sync and share your existing data on that FTP
```

```
drive at school, a
 Dropbox or a NAS you have at home.
. . .
. . .
```

It looks like this is what I need. So, let's install it:

```
donnie@ubuntu1804-1:~$ snap install nextcloud
 error: access denied (try with sudo)
 donnie@ubuntu1804-1:~$
```

Hey, now. Wasn't I supposed to be able to do this without root privileges? Well yeah, but only after I log into my Ubuntu One account. To create an account, just go to the registration page at `https://login.ubuntu.com/`.

Then, go back to the command line and use your new credentials to log into the snap store:

```
donnie@ubuntu1804-1:~$ sudo snap login
[sudo] password for donnie:
Personal information is handled as per our privacy notice at
https://www.ubuntu.com/legal/dataprivacy/snap-store
Email address: donniet@any.net
Password of "donniet@any.net":
Login successful
donnie@ubuntu1804-1:~$
```

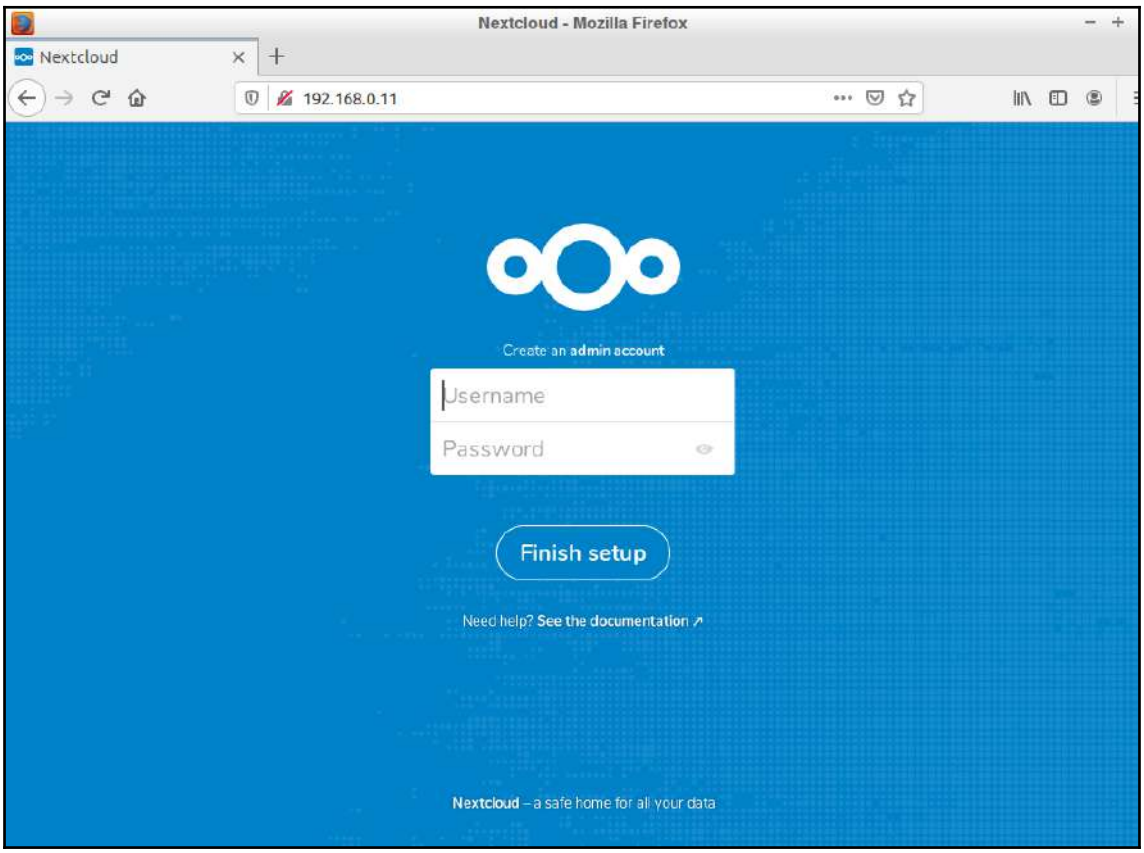
This is the last time that you'll need to use `sudo` with snap commands on this machine. Now, I can install Nextcloud:

```
donnie@ubuntu1804-1:~$ snap install nextcloud
nextcloud 16.0.5snap3 from Nextcloud√ installed
donnie@ubuntu1804-1:~$
```

And this time, it works. To start it, just use the following command:

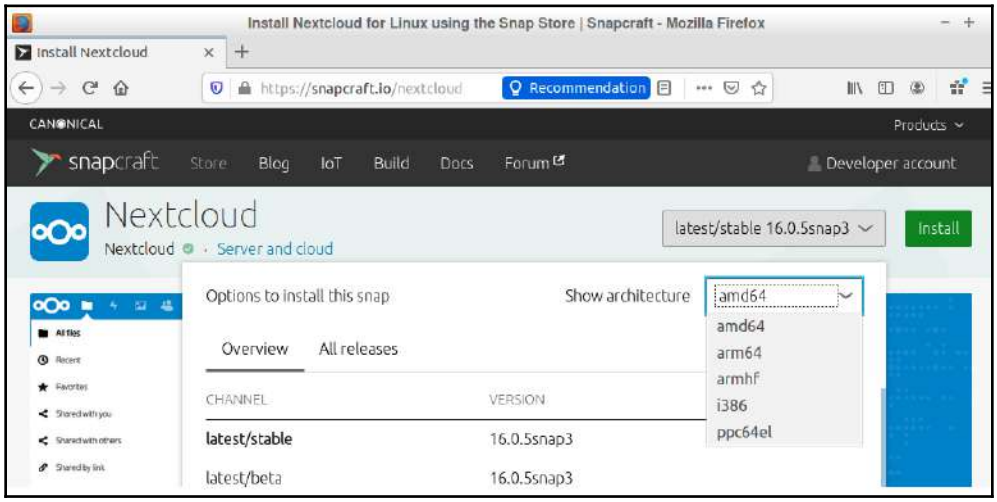
```
donnie@ubuntu1804-1:~$ snap start nextcloud
Started.
donnie@ubuntu1804-1:~$
```

Finally, from a desktop machine, navigate to the IP address of the **Nextcloud** server, click on **Create an admin account**, and once you've filled in all the details, click **Finish Setup**:



Simple, right? Imagine how long that would have taken doing it the old-fashioned way. The only slight catch is that it's running on an unencrypted HTTP connection, so you definitely don't want to expose this to the internet.

The Snapcraft store is Canonical's official repository for snap packages. Anyone who wants to can create an account and upload his or her own snaps. There are plenty of applications there for desktop/workstations, servers, and IoT devices. Several different machine architectures, including x86_64, ARM, and PowerPC, are supported. (So yes, this can even be useful for your Raspberry Pi device.) This can be seen in the following screenshot:



That's pretty much all there is to it. Despite the controversies, it's still a pretty cool concept.

If you have to deploy IoT devices, you might want to look into Ubuntu Core. It's a stripped-down version of Ubuntu that consists wholly of snap packages. Space doesn't permit me to cover it in detail here, but you can read all about it at `https://ubuntu.com/core`.

Now that you've seen how to work with Ubuntu's Snappy system, we'll look at Fedora's Flatpak system.

Sandboxing with Flatpak

The Flatpak system, which was created by the Fedora Linux team, works toward the same goal as Ubuntu's Snappy system, but there are significant differences in their implementation. You can have one or both systems running on any given Linux machine. With either system, you can create a universal package that runs on any machine that has Snappy or Flatpak installed. And both systems run each of their applications in their own security sandbox.

However, as I mentioned previously, there are differences:

- Instead of having each application package entirely self-contained, Flatpak installs shared runtime libraries that the application can access. This helps cut down on disk space usage.
- The Fedora folk operate a central repository that they call Flathub. However, they also made the server code available for anyone who wants to set up his or her own Flatpak repository.
- Flatpak requires just a tiny bit more effort to set up because after you install it, you have to configure it to use the desired repository.
- The Snapcraft store has packages for server, desktop, and IoT use. Flathub mainly has desktop applications.

Depending on which distro you're running, you may or may not already have the Flatpak system installed. On Debian/Ubuntu systems, install it by using the following command:

```
sudo apt update
sudo apt install flatpak
```

On RHEL, CentOS, and Fedora systems, there's a good chance that it's already installed. If it isn't, just install it with the normal `yum` or `dnf` commands. After you've installed Flatpak, go to the Flatpak Quick Setup page to see how to configure it. Click on the icon for the distro that you're running and follow the directions.

You'll find the Quick Setup page here: `https://flatpak.org/setup/`.

Curiously, though, if you click on the icon for either CentOS or Red Hat, you'll see the directive to install the Flathub repository, but it doesn't say how to do it. That's okay, though. Just click on the icon for any other distro and you'll see the proper command, which looks like this:

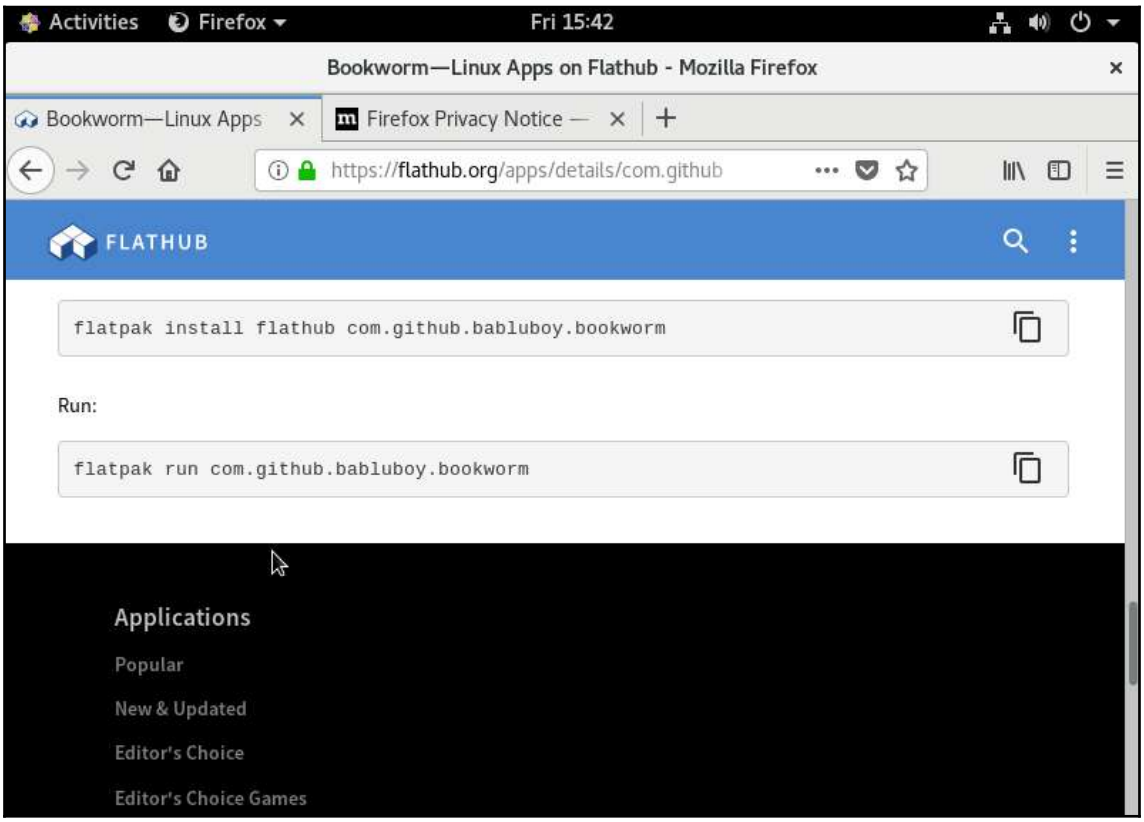
```
sudo flatpak remote-add --if-not-exists flathub
https://flathub.org/repo/flathub.flatpakrepo
```

Supposedly, this is the only `flatpak` command that requires sudo privileges. (Supposedly, I say, because, in practice, certain runtime library packages do require sudo privileges to install.)

After installing the repository, restart the machine. After the machine has rebooted, you'll be ready to install some applications. To pick one, go to the Flathub website and browse until you find something you want.

You'll find Flathub here: `https://flathub.org/home`.

Let's say that you've browsed through the Productivity apps and found the Bookworm e-book reader. Click on the link to go to the **Bookworm** app page. You'll see an **Install** button at the top of it. If you click on that button, you'll download the install file for **Bookworm**. To install it, you'll still need to type a command at the command line. Your best bet is to scroll down to the bottom of the page, where you'll see the command that will both download and install the app at the same time:



There's also the command to run the app, but you might not need it. Depending on which distro you're running, you might or might not have an icon created for you in your Start menu. (The only distro that I haven't seen do that is Debian.)

As I mentioned previously, `flatpak` commands sometimes require sudo privileges, and sometimes they don't. Try running your `flatpak` command without `sudo` first. If it doesn't work, just do it again with `sudo`.

Unlike Snappy, Flatpak doesn't automatically update its apps. You'll have to do that yourself by periodically implementing the following command:

```
flatpak update
```

In this section, you looked at the basics of using the Snappy and Flatpak universal packaging systems. With each one, developers can package their applications just once, instead of doing it multiple times with multiple types of packages. End users can use them so that their applications that are always up to date, instead of having to stick with the more outdated versions that are in their distro repositories. And, in keeping with the overall context of this book, understand that by running applications in their own isolated sandboxes, both of these systems provide an extra measure of application security.

Summary

So, another big chapter is behind us, and we've seen lots of cool stuff. We started by looking at the `/proc` filesystem and at how to configure some of its settings for the best security possible. After that, we looked at how cgroups, namespaces, kernel capabilities, and SECCOMP can be used to isolate processes from each other. We wrapped this chapter up with some examples of utilities and package management systems that use these cool technologies.

In the next chapter, we'll talk about the different ways you can scan, audit, and harden your systems. I'll see you there.

Questions

1. Which of the following is true?
   A. `/proc` is just like any other directory in the Linux filesystem.
   B. `/proc` is the only pseudo-filesystem in Linux.
   C. `/proc` is one of several pseudo-filesystems in Linux.
   D. You can set values for `/proc` parameters with the `systemctl` command.

2. Which of the following commands would you use to set a value for a `/proc` parameter?
   A. `sudo systemctl -w`
   B. `sudo sysctl -w`
   C. `sudo procctl -w`
   D. `sudo sysctl -o`
   E. `sudo systemctl -o`
3. You need a program executable to run with one specific root privilege, without having to grant any root privileges to the person who will be running it. What would you do?
   A. Add a namespace.
   B. Create a SECCOMP profile.
   C. Add the SUID permission.
   D. Add a kernel capability.
4. Where would you find information about user processes?
   A. In the numbered subdirectories of the `/proc` filesystem.
   B. In the alphabetically named subdirectories of the `/proc` filesystem.
   C. In the `/dev` directory.
   D. In each user's home directory.
5. What is a syscall?
   A. It tells the Linux kernel to perform a privileged action on behalf of a user.
   B. It calls new system information into the kernel.
   C. It keeps track of everything that the system kernel is doing.
   D. It isolates calls to system resources from each other.
6. What is the best way to allow users to only see information about their own processes?
   A. Add the `hidepid=2` option to the kernel startup parameters in the GRUB configuration.
   B. Add the `nopid=1` option to the kernel startup parameters in the GRUB configuration.
   C. Add the `nopid=1` option to the `/etc/fstab` file.
   D. Add the `hidepid=1` option to the `/etc/fstab` file.
7. Which of the following commands would you use to see which kernel parameters need to be changed for the best security?
   A. `sudo audit system`
   B. `sudo lynis audit system`
   C. `sudo system audit`
   D. `sudo lynis system audit`

8. Which of the following commands would allow a non-privileged user to start a Python web server on Port 80 without using root privileges?
   A. `sudo setcap 'CAP_NET_SERVICE+ep' /usr/bin/python2.7`
   B. `sudo setcap 'CAP_NET_BIND_SERVICE+ep' /usr/bin/python2.7`
   C. `sudo getcap 'CAP_NET_BIND_SERVICE+ep' /usr/bin/python2.7`
   D. `sudo setcap 'CAP_NET_SERVICE+ep' /usr/bin/python2.7`
9. What is a major difference between the Snappy and Flatpak systems?
   A. There are none.
   B. Flatpak packages are completely self-contained, but Snappy packages have you install separate runtime packages.
   C. Snappy packages are completely self-contained, but Flatpak packages have you install separate runtime packages.
   D. Flatpak packages run in their own sandbox, but Snappy packages don't.
   E. Snappy packages run in their own sandbox, but Flatpak packages don't.
10. You need to limit the number of syscalls that your Docker container can make. How would you do that?
    A. Create the container in its own cgroup and configure the syscall limits for that cgroup.
    B. Create the container in its own namespace and configure the syscall limits for that namespace.
    C. Run the container under Firejail.
    D. Create the container with a SECCOMP profile.

Answers

1. C.
2. B.
3. D.
4. A.
5. A.
6. D.
7. B.
8. B.
9. C.
10. D.

Further reading

- Exploring the `/proc` filesystem: `https://www.tecmint.com/exploring-proc-file-system-in-linux/`
- Linux kernel security subsystem wiki: `https://kernsec.org/wiki/index.php/Main_Page`
- Linux Magic System Request Key Hacks: `https://www.kernel.org/doc/html/latest/admin-guide/sysrq.html`
- Linux to get kernel "lockdown" feature: `https://www.zdnet.com/article/linux-to-get-kernel-lockdown-feature/`
- Protect hard and symbolic links in RHEL/CentOS: `https://www.tecmint.com/protect-hard-and-symbolic-links-in-centos-rhel/https://www.tecmint.com/protect-hard-and-symbolic-links-in-centos-rhel/`
- Log suspicious martian packets: `https://www.cyberciti.biz/faq/linux-log-suspicious-martian-packets-un-routable-source-addresses/`
- Protect against the usage of ptract: `https://linux-audit.com/protect-ptrace-processes-kernel-yama-ptrace_scope/`
- Introduction to Linux Control Groups (cgroups): `https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/resource_management_guide/chap-introduction_to_control_groups`
- RHEL 7: How to get started with cgroups: `https://www.certdepot.net/rhel7-get-started-cgroups/`
- Limit CPU Usage on Ubuntu 19.10 with systemd cgroups: `https://youtu.be/_AODvcO5Q_8`
- Control Groups versus Namespaces: `https://youtu.be/dTOT9QKZ2Lw`
- How to manage Linux file capabilities: `https://www.howtoforge.com/how-to-manage-linux-file-capabilities/`
- getcap, setcap, and file capabilities: `https://www.insecure.ws/linux/getcap_setcap.html`
- Docker security features – SECCOMP profiles: `https://blog.aquasec.com/new-docker-security-features-and-what-they-mean-seccomp-profiles`
- Docker documentation – Docker Security: `https://docs.docker.com/engine/security/security/`
- Firejail website: `https://firejail.wordpress.com/`
- Firejail documentation: `https://firejail.wordpress.com/documentation-2/`
- Introduction to App Packages on Linux: `https://fosspost.org/education/app-packages-linux-snap-flatpak-appimage`

- The Ubuntu snap Usage Tutorial: `https://tutorials.ubuntu.com/tutorial/basic-snap-usage#0`
- The Official Canonical Snapcraft Store: `https://snapcraft.io/`
- The future of Linux desktop application delivery is Flatpak and Snap: `https://www.zdnet.com/article/the-future-of-linux-desktop-application-delivery-is-flatpak-and-snap/`
- Flathub: `https://flathub.org/home`
- Three approaches to secrets management for Flatpak applications: `https://opensource.com/article/19/11/secrets-management-flatpak-applications`