O'REILLY®

2nd Edition

# Container Security

Fundamental Technology Concepts That Protect Containerized Applications

Early Release

RAW & UNEDITED

Liz Rice

# Container Security

SECOND EDITION

Fundamental Technology Concepts that Protect Containerized Applications

**Liz Rice**

## Container Security

by Liz Rice

Published by O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Acquisitions Editor: Megan Laddusaw

Developmental Editor: Rita Fernando

Production Editor: Elizabeth Faerm

Copyeditor: TO COME

Proofreader: TO COME

Indexer: TO COME

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

October 2025: Second Edition

## Revision History for the Early Release

- 2025-06-27: First release

See http://oreilly.com/catalog/errata.csp?isbn=9781492056706 for release details.

979-8-341-62765-9

[LSI]

# Brief Table of Contents (*Not Yet Final*)

*Chapter 1: Container Security Threats* (unavailable)

Chapter 2: Linux System Calls, Permissions, and Capabilities (available)

Chapter 3: Control Groups (available)

Chapter 4: Container Isolation (available)

Chapter 5: Virtual Machines (available)

*Chapter 6: Container Images* (unavailable)

*Chapter 7: Software Vulnerabilities in Images* (unavailable)

Chapter 8: Strengthening Container Isolation (available)

*Chapter 9: Breaking Container Isolation* (unavailable)

*Chapter 10: Container Network Security* (unavailable)

*Chapter 11: Securely Connecting Components with TLS* (unavailable)

Chapter 12: Passing Secrets to Containers (available)

*Chapter 13: Container Runtime Protection* (unavailable)

*Chapter 14: Containers and the OWASP Top 10* (unavailable)

*Conclusions* (unavailable)

*Security Checklist* (unavailable)

# Chapter 1. Linux System Calls, Permissions, and Capabilities

In most cases, containers run within a computer running a Linux operating system, and it’s going to be helpful to understand some of the fundamental features of Linux so that you can see how they affect security, and in particular how they apply to containers. I’ll cover system calls, file-based permissions, and capabilities and conclude with a discussion of privilege escalation. If you’re familiar with these concepts, feel free to skip to the next chapter.

This is all important because *containers run Linux processes that are visible from the host*. A containerized process uses system calls and needs permissions and privileges in just the same way that a regular process does. But containers give us some new ways to control how these permissions are assigned at runtime or during the container image build process, which will have a significant impact on security.

## System Calls

Applications run in what's called *user space*, which has a lower level of privilege than the operating system kernel. If an application wants to do something like access a file, communicate using a network, or even find the time of day, it has to ask the kernel to do it on the application's behalf. The programmatic interface that the user space code uses to make these requests of the kernel is known as the *system call* or *syscall* interface.

There are some 400+ different system calls, with the number varying according to the version of Linux kernel. Here are a few examples:

`read`

: read data from a file

`write`

: write data to a file

`open`

: open a file for subsequent reading or writing

`execve`

: run an executable program

`chown`

: change the owner of a file

`clone`

: create a new process

Application developers rarely if ever need to worry about system calls directly, as they are usually wrapped in higher-level programming abstractions. The lowest-level abstraction you're likely to come across as an app developer is the `glibc` or `musl` libraries, or the Golang `syscall` package. In practice these are usually wrapped by higher layers of abstractions as well.

**NOTE**

If you would like to learn more about system calls, check out my talk “A Beginner’s Guide to Syscalls”, available on O’Reilly’s learning platform.

Application code uses system calls in exactly the same way whether it’s running in a container or not, but as you will see later in this book, there are security implications to the fact that all the containers on a single host share—that is, they are making system calls to—the same kernel.

Not all applications need all system calls, so—following the principle of least privilege—there are Linux security features that allow users to limit the set of system calls that different programs can access. You’ll see how these can be applied to containers in Chapter 5.

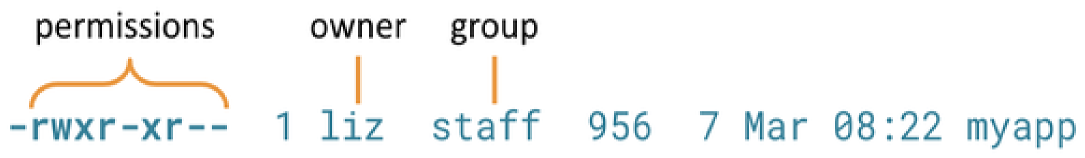
I’ll return to the subject of user space and kernel-level privileges in Chapter 4. For now let’s turn to the question of how Linux controls permissions on files.

## File Permissions

On any Linux system, whether you are running containers or not, file permissions are the cornerstone of security. There is a saying that in Linux, everything is a file. Application code, data, configuration information, logs, and so on—it’s all held in files. Even physical devices like screens and printers are represented as files. Permissions on files determine which users are allowed to access those files and what actions they can perform on the files. These permissions are sometimes referred to as *discretionary access control*, or DAC.

Let’s examine this a little more closely.

If you have spent much time in a Linux terminal, you will likely have run the `ls -l` command to retrieve information about files and their attributes.



```
-rwxr-xr--  1 liz  staff  956  7 Mar 08:22 myapp
```

*Figure 1-1. Linux file permissions example*

In the example in Figure 1-1, you can see a file called *myapp* that is owned by a user called “liz” and is associated with the group “staff.” The permission attributes tell you what actions users can perform on this file, depending on their identity. There are nine characters in this output that represent the permissions attributes, and you should think of these in groups of three:

- The first group of three characters describes permissions for the user who owns the file (“liz” in this example).
- The second group gives permissions for members of the file’s group (here, “staff”).
- The final set shows what any other user (who isn’t “liz” or a member of “staff”) is permitted to do.

There are three actions that users might be able to perform on this file: read, write, or execute, depending on whether the *r*, *w*, and *x* bits are set. The three characters in each group represent bits that are either on or off, showing which of these three actions are permitted—a dash means that the bit isn’t set.

In this example, only the owner of the file can write to it, because the *w* bit is set only in the first group, representing the owner permissions. The owner can execute the file, as can any member of the group “staff.” Any user is allowed to read the file, because the *r* bit is set in all three groups.

**NOTE**

If you’d like more detail on Linux permissions, there is a good article at https://www.linuxjournal.com/content/mastering-linux-file-permissions-and-ownership.

There’s a good chance that you were already familiar with these *r*, *w*, and *x* bits, but that’s not the end of the story. Permissions can be affected by the use of *setuid*, *setgid*, and *sticky* bits. The first two are important from a security perspective because they can allow a process to obtain additional permissions, which an attacker might use for malevolent purposes.

## setuid and setgid

Normally, when you execute a file, the process that gets started inherits your user ID. If the file has the *setuid* bit set, the process will have the user ID of the file’s owner. The following example uses a copy of the sleep executable owned by a non-root user:

```
liz@vm:~$ ls -l `which sleep`
-rwxr-xr-x 1 root root 35336 Apr  5  2024 /usr/bin/sleep
liz@vm:~$ cp /usr/bin/sleep ./mysleep
liz@vm:~$ ls -l mysleep
-rwxr-xr-x 1 liz liz 35336 May  7 10:50 mysleep
```

The `ls` output shows that the copy is owned by the user called `liz`. Run this by executing `./mysleep 100`, and in a second terminal you can take a look at the running process—the `100` means you’ll have 100 seconds to do this before the process terminates (I have removed some lines from this output for clarity):

```
liz@vm:~$ ps ajf
PPID   PID  PGID   SID TTY    TPGID STAT  UID  TIME COMMAND
1351  1352  1352  1352 pts/1   1376 Ss   1001  0:00 -bash
```

```
1352  1376  1376  1352 pts/1   1376 S+   1001  0:00  \_ ./mysleep
10
```

This is running under user ID `1001`, which corresponds to the user `liz` on my VM. Now let's run the same program under root by executing `sudo ./mysleep 100`. In a second terminal the process looks slightly different.

```
PPID   PID  PGID   SID TTY    TPGID STAT  UID  TIME COMMAND
1351  1352  1352  1352 pts/1   2127 Ss   1001  0:00 -bash
1352  2127  2127  1352 pts/1   2127 S+      0  0:00  \_ sudo
./mysleep 10
2127  2128  2128  2128 pts/2   2129 Ss      0  0:00      \_ sudo
./mysleep 10
2128  2129  2129  2128 pts/2   2129 S+      0  0:00          \_
./mysleep 10
```

The UID of 0 shows that both the `sudo` process and the `mysleep` process are running under the root UID.

**NOTE**

If your version of sudo is 1.9.14 or above, as mine is, you'll see two `sudo` processes, but for older versions there will be only one. The man page for `sudo` tells us that in newer versions, it forks one process in a new pseudo-terminal (`pts/2` in my output above) to act as a *monitor* process, before forking a second time to run the command.

Now let's try turning on the *setuid* bit:

```
liz@vm:~$ chmod +s mysleep
liz@vvm:~$ ls -l mysleep
-rwsr-sr-x 1 liz liz 35336 May  7 10:50 mysleep
```

Run `sudo ./mysleep 100` again, and look at the running processes again from the second terminal:

```
  PPID   PID  PGID   SID TTY   TPGID STAT  UID  TIME COMMAND
 1351  1352  1352  1352 pts/1 29543 Ss   1001  0:00 -bash
 1352 29543 29543  1352 pts/1 29543 S+      0  0:00  \_ sudo
./mysleep 10
29543 29544 29544 29544 pts/2 29545 Ss      0  0:00      \_ sudo
./mysleep 10
29544 29545 29545 29544 pts/2 29545 S+   1001  0:00          \_
./mysleep 10
```

The `sudo` processes are still running as root, but this time `mysleep` has taken its user ID from the owner of the file.

This *setuid* bit can be used to give a program privileges that it needs but that are not usually extended to regular users.

**NOTE**

Perhaps the canonical example of the *setuid* bit used to be the executable `ping`, which needed permission to open raw network sockets in order to send its ping message.

An administrator might be happy for their users to run `ping`, but that doesn't mean they are comfortable letting users open raw network sockets for any other purpose they might think of. Instead, the `ping` executable was installed with the *setuid* bit set and owned by the root user so that `ping` can use privileges normally associated with root.

This is no longer needed since the addition of ICMP sockets in kernel version 5.6, which are designed to allow non-privileged processes to open a socket purely for ICMP protocol messages, as used by `ping`.

At the time of writing, most distributions don't yet use this ICMP sockets mechanism for ping. Instead, they give the ping executable permission to access raw sockets using a *capability* called `CAP_NET_RAW`. We'll look into this in more detail shortly, in "Linux Capabilities".

We've already seen *setuid* in action, because it's used by `sudo`, which is an executable owned by root.

```
liz@vm:~$ ls -l `which sudo`
-rwsr-xr-x 1 root root 277936 Apr  8  2024 /usr/bin/sudo
```

The *setuid* bit on sudo means that the executable runs as the root user, which matches what we saw in the output from `ps` earlier.

**NOTE**

Once it's running, `sudo` goes on to check that the real user that invoked it actually has permissions to run `sudo`, by checking the `sudoers` file or some other configured security policy mechanism. If you want to explore this in more detail, `man sudo` has a good explanation.

As you saw when copying `sleep`, when you copy a file, its ownership attributes are set according to the user ID you're operating as, and the *setuid* bit is not carried over. If you want the *setuid* bit you can run `chmod +s` on the file.

Let's explore *setuid* further by taking a copy of `bash`.

```
liz@vm:~$ cp `which bash` ./mybash
liz@vm:~$ ls -l mybash
-rwxr-xr-x  1 liz  liz  1446024 May  7 15:33 mybash
```

This file is owned by my regular user and doesn't have the *setuid* bit set. Now let's change both those things.

```
liz@vm:~$ sudo chown root ./mybash
liz@vm:~$ sudo chmod +s ./mybash
liz@vm:~$ ls -l mybash
-rwsr-sr-x  1 root liz  1446024 May  7 15:33 mybash
```

Since this executable is owned by root and has *setuid*, it seems reasonable to imagine that when you run it, the process will be running as root. And yet, look what happens when you try it.

```
liz@vm:~$ ./mybash
mybash-5.2$ whoami
liz
```

As you can see, the process is *not* running as root, even though the *setuid* bit is on and the file is owned by root. What's happening here? The answer is that in modern versions of `bash` (and several other interpreters like `python`, `node` and `ruby`) the executable might start off running as root, but it explicitly resets its user ID to be that of the original user to avoid potential privilege escalations.

To explore this for yourself in more detail, you can use `strace` to see the system calls that the `bash` (or `mybash`) executable makes. Find the process ID of your shell, and then in a second terminal run the following command:

```
liz@vm:~$ sudo strace -f -p <shell process ID>
```

This will trace out all the system calls from within that shell, including any executables running within it. Look for the `setresuid()` or setuid() system calls being used to reset the user ID..

Not all executables are written to reset the user ID in this way. You can use the copy of `sleep` from earlier in this chapter to see more normal *setuid* behavior. Change the ownership to root, set the *setuid* bit (this gets reset when you change ownership), and then run it as a non-root user:

```
liz@vm:~$ sudo chown root mysleep
liz@vm:~$ sudo chmod +s mysleep
liz@vm:~$ ls -l ./mysleep
-rwsr-sr-x 1 root liz 35336 May  7 10:50 mysleep
liz@vm:~$ ./mysleep 100
```

In another terminal you can use `ps` to see that this process is running under root’s user ID:

```
liz@vm:~$ ps ajf      6646  0.0  0.0   7468   764 pts/2    S+
00:38   0:00 ./mysleep 100
  PPID   PID  PGID   SID TTY   TPGID STAT  UID  TIME COMMAND
35834 35835 35835 35835 pts/0 42509 Ss    1001   0:00 -bash
35835 42509 42509 35835 pts/0 42509 S+       0   0:00  \_
./mysleep 100
```

Now that you have experimented with the *setuid* bit, you are in a good position to consider its security implications.

### Security implications of setuid

The *setuid* bit allows someone to act as if they were a different user, which could give them access to different files, executables, and privileges that they are not supposed to have. As you've seen, modern versions of `bash` and most shells and interpreters reset their user ID to avoid being used for trivial privilege escalations. Because *setuid* provides a dangerous pathway to privilege escalation, some container image scanners (covered in Chapter 7) will report on the presence of files with the *setuid* bit set.

You can also prevent *setuid* from being used within a container using the `--security-opt no-new-privileges` option on a `docker run` command - I'll come back to this in Chapter 3. However, that won't stop an attacker from writing a *setuid* executable owned by root onto mounted directory on the host. You'll find an example of this in the `chapter2/setuid` directory of the GitHub repo that accompanies this book. Host volume mounts can lead to all sorts of attacks, and we'll discuss this more in Chapter X.

The *setuid* bit dates from a time when privileges were much simpler—either your process had root privileges or it didn't. The *setuid* bit provided a mechanism for granting extra privileges to non-root users. Version 2.2 of the Linux kernel introduced more granular control over these extra privileges through *capabilities*.

## Linux Capabilities

There are over 30 different capabilities in today's Linux kernel. Capabilities can be assigned to a thread to determine whether that thread can perform certain actions. For example, a thread needs the `CAP_NET_BIND_SERVICE` capability in order to bind to a low-numbered (below 1024) port. `CAP_SYS_BOOT` exists so that arbitrary executables don't have permission to reboot the system. `CAP_SYS_MODULE` is needed to load or unload kernel modules, and `CAP_BPF` is needed to load eBPF programs.

**NOTE**

Consult man `capabilities` on a Linux machine for detailed information on each individual capabilities.

I mentioned earlier that the ping tool uses the `CAP_NET_RAW` capability so that it can open a raw network socket. T

Capabilities can be assigned to both files and processes. You can see the capabilities for a file using `getcap`, like this:

```
liz@vm:~$ getcap which ping
/usr/bin/ping cap_net_raw=ep
```

You can see the capabilities assigned to a process by using the `getpcaps` command. Many processes typically won’t have capabilities:

```
liz@vm:~$ ps
  PID TTY          TIME CMD
22355 pts/0    00:00:00 bash
25058 pts/0    00:00:00 ps
liz@vm:~$ getpcaps 22355
22355: =
```

In the past, `getpcaps` assumed that if a process was running as root, it had all capabilities, so would report the whole list. These days, `getpcaps` and other tools have been updated not to make this assumption, so processes running as root will typically appear with no capabilities.

We’ve seen that the executable file for ping has `CAP_NET_RAW` associated with it, so let’s see the capabilities assigned to the process when we run it. Leave `ping` running in one terminal:

```
liz@vm:~$ ping 127.0.0.1
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.162 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.188 ms
…
```

In a second terminal, get the process ID and check its capabilities.

```
liz@vm:~$ getpcaps 50394
50394: =
```

How is ping successfully opening a socket if it doesn't have the `CAP_NET_RAW` capability required? And why doesn't it have it, if the executable file has it?

The answer can be found by using a second terminal to trace the system calls, exactly the same as described earlier when examining how bash behaves. You'll see that ping is now capabilities-aware, and deliberately discards `CAP_NET_RAW` once the socket is open and it has no further use for the capability. (A few irrelevant calls and details have been omitted for clarity.)

```
capget({version=_LINUX_CAPABILITY_VERSION_3, pid=0},
{effective=0, permitted=1<<CAP_NET_RAW, inheritable=0}) = 0
❶
capset({version=_LINUX_CAPABILITY_VERSION_3, pid=0},
{effective=1<<CAP_NET_RAW, permitted=1<<CAP_NET_RAW,
inheritable=0}) = 0     ❷
socket(AF_INET, SOCK_RAW, IPPROTO_ICMP) = 3           ❸
socket(AF_INET6, SOCK_RAW, IPPROTO_ICMPV6) = 4
capget({version=_LINUX_CAPABILITY_VERSION_3, pid=0},
{effective=1<<CAP_NET_RAW, permitted=1<<CAP_NET_RAW,
inheritable=0}) = 0
capset({version=_LINUX_CAPABILITY_VERSION_3, pid=0},
{effective=0, permitted=1<<CAP_NET_RAW, inheritable=0})
❹
```

❶ The process checks that CAP_NET_RAW is in the set of permissions that it is permitted to use. The return code 0 tells it that it is.

❷ The process uses capset to make that capability effective.

❸ It opens sockets for IPv4 and IPv6.

❹ Now that the sockets are open, it uses capset to remove the capability from its effective set.

By the time `getpcaps` inspected the process's capabilities, it was no longer in effect for the process.

**NOTE**

For a more in-depth discussion of the ways that file and process permissions interact, see Adrian Mouat's post on Linux capabilities in practice.

Following the principle of least privilege, it's a good idea to grant only the capabilities that are needed for a process to do its job. When you run a container, you get the option to control the capabilities that are permitted, as you'll see in Chapter 5.

Now that you are familiar with the basic concepts of permissions and privileges in Linux, I'd like to turn to the idea of escalating privileges.

## Privilege Escalation

The term "privilege escalation" means extending beyond the privileges you were supposed to have so that you can take actions that you shouldn't be permitted to take. To escalate their privileges, an attacker takes advantage of a system vulnerability or poor configuration to grant themselves extra permissions.

Oftentimes, the attacker starts as a non-privileged user and wants to gain root privileges on the machine. A common method of escalating privileges is to look for software that's already running as root and then take advantage of known vulnerabilities in the software. For example, web server software might include a vulnerability that allows an attacker to remotely execute code, such as the Struts vulnerabilities[1]. If the web server is running as root, anything that is remotely executed by an attacker will run with root privileges. For this reason, it is a good idea to run software as a non-privileged user whenever possible.

As you’ll learn later in this book, by default *containers run as root*. This means that compared with a traditional Linux machine, applications running in containers are far more likely to be running as root. An attacker who can take control of a process inside a container still has to somehow escape the container, but once they achieve that, they will be root on the host, and there is no need for any further privilege escalation. Chapter 9 discusses this in more detail.

Even if a container is running as a non-root user, there is potential for privilege escalation based on the Linux permissions mechanisms you have seen earlier in this chapter:

- Container images including executable files with the setuid bit
- Additional capabilities granted to a container running as a non-root user

You’ll learn about approaches for mitigating these issues later in the book.

## Summary

In this chapter you have learned (or revised) some fundamental Linux mechanisms that will be essential to understanding later chapters of this book. They also come into play in security in numerous ways; the container security controls that you will encounter are all built on top of these fundamentals.

Now that you have some basic Linux security controls under your belt, it’s time to start looking at the mechanisms that make up containers so that you can understand for yourself how root on the host and in the container are one and the same thing.

---

[1] In the first edition of this book, I was referring to a 2018 critical remote code execution vulnerability that had received a lot of press coverage. It turns out there have been other serious Struts vulnerabilities since then, which is a good case study in how vulnerabilities

continue to be found in widely-used software packages, and why it’s important to keep updating your dependencies!

# Chapter 2. Control Groups

In this chapter, you will learn about one of the fundamental building blocks that are used to make containers: *control groups*, more commonly known as *cgroups*.

Cgroups limit the resources, such as memory, CPU, and network input/output, that a group of processes can use. In containers, they are used to distribute resources across different workloads in a controlled fashion. From a security perspective, well-tuned cgroups can ensure that one process can’t affect the behavior of other processes by hogging all the resources—for example, using all the CPU or memory to starve other applications. You can also limit the total number of processes allowed within a control group - a handy technique to protect against *fork bombs*, which I’ll cover at the end of the chapter.

As you will see in detail in Chapter 3, containers run as regular Linux processes, so cgroups can be used to limit the resources available to each container. Let’s see how cgroups are organized.

**NOTE**

Most Linux distributions today use cgroups version 2, which has some improvements over the original implementation that was widely deployed when containers first became popular. Cgroups v2 is now what's used by Kubernetes and all the popular container runtimes, and it's what is discussed here. However, you might find some references to v1 in older literature.

The main difference is that version 2 uses a single, unified hierarchy for managing all the supported resource types, rather than having separate hierarchies for the different types of resource being managed.

## Control Group Controllers

Control groups are represented in the Linux filesystem under a mount point residing at `/sys/fs/cgroup`. Managing cgroups involves reading and writing to the files and directories under this mount point. Let's take a look at the contents of that directory.

```
root@vm:/sys/fs/cgroup# ls
cgroup.controllers       io.pressure
cgroup.max.depth         io.prio.class
cgroup.max.descendants   io.stat
cgroup.pressure          memory.numa_stat
cgroup.procs             memory.pressure
cgroup.stat              memory.reclaim
cgroup.subtree_control   memory.stat
cgroup.threads           memory.zswap.writeback
cpu.pressure             misc.capacity
cpu.stat                 misc.current
cpu.stat.local           misc.peak
cpuset.cpus.effective    proc-sys-fs-binfmt_misc.mount
cpuset.cpus.isolated     sys-fs-fuse-connections.mount
cpuset.mems.effective    sys-kernel-config.mount
dev-hugepages.mount      sys-kernel-debug.mount
dev-mqueue.mount         sys-kernel-tracing.mount
init.scope               system.slice
io.cost.model            user.slice
io.cost.qos
```

As I'll show you shortly, a new control group can be created by making a new directory, which in turn can have child control groups created within it, building up a hierarchy of control groups.

The `cgroup.controllers` file shows what cgroup *controllers* are available on this machine.

```
root@vm:/sys/fs/cgroup# cat cgroup.controllers
cpuset cpu io memory hugetlb pids rdma misc
```

Each controller manages a type of resource that processes might want to consume. For example, the `cpu` controller manages the CPU usage of the processes in a cgroup, and the `memory` controller manages the memory they can access.

To take effect, controllers have to be enabled by writing the controller name into the `cgroup.subtree_control` file, and a controller can only be enabled for a cgroup if it is enabled in its parent. Every running Linux process is a member of exactly one control group, and you'll find the process IDs of all the group's members listed in the `cgroup.procs` file.

## Creating and configuring cgroups

Creating a subdirectory inside the `/sys/fs/cgroup` directory creates a cgroup, and the kernel automatically populates the directory with the various files that represent parameters and statistics about that group and its resources:

```
root@vm:/sys/fs/cgroup# mkdir liz
root@vm:/sys/fs/cgroup# ls liz
cgroup.controllers
cgroup.events
cgroup.freeze
...
pids.max
pids.peak
rdma.current
rdma.max
```

The details of what each of these different files means are beyond the scope of this book, but some of the files hold parameters that you can manipulate to define limits for the control group, and others communicate statistics about the current use of resources in the control group. You could probably make an educated guess that, for example, *memory.current* is the file that describes how much memory is currently being used by the control group. The maximum that the cgroup is allowed to use is defined by *memory.max*:

```
root@vm:/sys/fs/cgroup/liz# cat memory.max max
```

By default the memory isn't limited, and if a process is allowed to consume unlimited memory, it can starve other processes on the same host. This might happen inadvertently through a memory leak in an application, or it could be the result of a resource exhaustion attack that takes advantage of a memory leak to deliberately use as much memory as possible. By setting limits on the memory and other resources that one process can access, you can reduce the effects of this kind of attack and ensure that other processes can carry on as normal.

To set a limit for a cgroup, you simply have to write the value into the file that corresponds to the parameter you want to limit. Let's set the maximum available memory for the cgroup I just created:

```
root@vm:/sys/fs/cgroup/memory/liz# echo 100000 > memory.max
```

Now you'll find that the `memory.max` parameter is approximately what you configured as the limit—presumably, rounded down to the nearest page size:

```
root@vm:/sys/fs/cgroup/memory/liz$ cat memory.max
98304
```

This illustrates how the limits are set for a group, but the final piece of the cgroups puzzle is to see how processes get assigned into cgroups.

## Assigning a Process to a Cgroup

As mentioned earlier, the set of processes in a cgroup are listed in its `cgroup.procs` file. A new cgroup will start off with no processes, so that file will be empty.

When you start a process, it joins the cgroup of its parent, but you can move it into a new cgroup by simply writing its process ID into the cgroup.procs file of the group you want it to join. In the following example, 29903 is the process ID of a shell:

```
root@vm:/sys/fs/cgroup/memory/liz$ echo 29903 > cgroup.procs
root@vmt:/sys/fs/cgroup/memory/liz$ cat cgroup.procs
29903
root@vm:/sys/fs/cgroup/memory/liz$ cat /proc/29903/cgroup
0::/liz
```

The shell is now a member of the `liz` cgroup, with its memory limited to a little under 100kB. This isn't a lot to play with, so even trying to run `ls` from inside the shell breaches the cgroup limit:

```
$ ls
Killed
```

The process gets killed when it attempts to exceed the memory limit.

## Docker Using Cgroups

You've seen how cgroups are manipulated by modifying the files in the cgroup filesystem for a particular type of resource. It's straightforward to see this in action in Docker.

**NOTE**

To follow along with these examples, you will need Docker running directly on a Linux (virtual) machine. If you’re running Docker for Mac/Windows, it’s running within a virtual machine, which means (as you’ll see in Chapter 4) that these examples won’t work for you, because the Docker daemon and containers are running using a separate kernel within that virtual machine.

Since most Linux distributions use systemd, and because the systemd driver is the recommended default driver for cgroups in Kubernetes, my examples assume the systemd driver. Some readers might encounter the alternative cgroupfs driver, but for the purposes of this book the only relevant difference is that the cgroups are created in a different place in the cgroup file system hierarchy.

Docker automatically creates a cgroup for each container, with a hierarchy that looks like this:

```
/sys/fs/cgroup/system.slice/
└── docker-<container_id>.scope/
    ├── cpu.max
    ├── memory.max
    ├── pids.max
    ├── cgroup.procs
    └── ...
```

The `system.slice` part of the hierarchy indicates that the cgroups are being managed using the systemd driver.

This example runs a container in the background with a limit of 100MB of memory. As you’ll see, Docker uses the cgroup mechanism to enforce this limit. The container will sleep for long enough for you to see its cgroup:

```
root@vm:~$ docker run --rm --memory 100M -d alpine sleep 10000
68fb008c5fd3f9067e1aa245b4522a9f3675720d8953371ecfcf2e9faf91b8a0
root@vm:/sys/fs/cgroup$ ls system.slice/docker-
68fb008c5fd3f9067e1aa245b4522a9f3675720d8953371ecfcf2e9faf91b8a0.
scope
cgroup.controllers
cgroup.events
cgroup.freeze
...
```

Check the memory limit for and current usage by this container :

```
root@vm:/sys/fs/cgroup$ cat system.slice/docker-
68fb...scope/memory.max
104857600
root@vm:/sys/fs/cgroup$ cat system.slice/docker-
68fb...scope/memory.current
462848
```

You can also confirm that the sleeping process is a member of the cgroup:

```
root@vm:/sys/fs/cgroup$ cat system.slice/docker-
68fb...scope/cgroup.procs  19824
root@vagrant:/sys/fs/cgroup$ ps -eaf | grep sleep
root     19824 19789  0 18:22 ?        00:00:00 sleep 10000
root     20486 18862  0 18:28 pts/1    00:00:00 grep --color=auto
sleep
```

## Preventing a fork bomb

A fork bomb rapidly creates processes that in turn create more processes, leading to an exponential growth in the use of resources that ultimately cripples the machine. I’ll show you how to reproduce this, but for caution’s sake, please don’t attempt running the fork bomb on a system that you can’t risk bringing to its knees!

**NOTE**

If you don’t want to risk this yourself, this video of a talk I gave a few years back includes a demonstration.

Earlier in this chapter I created a cgroup called liz and set a memory limit. Let’s remove the memory limit and instead define the maximum number of processes allowed in the cgroup.

```
root@vm:/sys/fs/cgroup/liz# echo max > memory.max
root@vm:/sys/fs/cgroup/liz# echo 20 > pids.max
```

Add the current shell to the cgroup:

```
root@vm:/sys/fs/cgroup/liz# echo $$ > cgroup.procs
```

Inspect the number of processes.

```
root@vm:/sys/fs/cgroup/liz# cat pids.current
2
```

Why are there two processes in this cgroup? The first is the shell, added explicitly by writing its process ID to the `cgroup.procs` file. Since a newly-created process inherits the cgroup of its parent, the second process observed is the `cat` program running within the shell.

Now you can run a fork bomb - this syntax will work if your shell is bash:

```
root@vm:/sys/fs/cgroup/liz# :(){ :|:& };:
```

You should very soon see your terminal filling up with `bash: fork: retry: Resource temporarily unavailable` messages, as processes fail to be started due to the limit imposed by the cgroup. While this might be annoying in your terminal window, other processes on the machine will still be able to operate fine. If the fork bomb were allowed to keep creating new processes, you would see other operations grinding to a halt.

You can use the kill feature of cgroups to terminate the fork bomb. Unfortunately, the shell you started the fork bomb in will also be a casualty of this operation, which kills all the processes in this group:

```
root@vm:/sys/fs/cgroup/liz# echo 1 > cgroup.kill
```

Curious about how the fork bomb works? While this has nothing really to do with container security, it’s a fun bit of syntax:

```
:() {...} defines a function called : (yes, a colon is a valid
name for a function in Bash).
```

The content of the function is `:|:&`. The function calls itself, and pipes the output into another invocation of itself. In bash, piping the output of a function causes it to be run in a new process, as does running a process in the background, which is what the & is for. As a result, each invocation of the : function spawns two processes.

The ; terminates the function definition, and the final : calls the function that has just been defined, kicking off an exponential cascade of process creation - until the cgroup limit is hit.

## Summary

Cgroups limit the resources available to different Linux processes. You don’t have to be using containers to take advantage of cgroups, but Docker and other container runtimes provide a convenient interface for using them: it’s very easy to set resource limits at the point where you run a container, and those limits are policed by cgroups.

Constraining resources provides protection against a class of attacks that attempt to disrupt your deployment by consuming excessive resources, thereby starving legitimate applications. It’s recommended that you set memory and CPU limits when you run your container applications.

Now that you know how resources are constrained in containers, you are ready to learn about the other pieces of the puzzle that make up containers: namespaces and changing the root directory. Move on to Chapter 3 to find out how these work.

# Chapter 3. Container Isolation

This is the chapter in which you’ll find out how containers really work! This will be essential to understanding the extent to which containers are isolated from each other and from the host. You will be able to assess for yourself the strength of the security boundary that surrounds a container.

As you’ll know if you have ever run `docker exec <image> bash`, a container looks a lot like a virtual machine from the inside. If you have shell access to a container and run ps, you can see only the processes that are running inside it. The container has its own network stack, and it seems to have its own filesystem with a root directory that bears no relation to root on the host. You can run containers with limited resources, such as a restricted amount of memory or a fraction of the available CPUs. This all happens using the Linux features that we’re going to delve into in this chapter.

However much they might superficially resemble each other, it’s important to realize that containers *aren’t* virtual machines, and in Chapter 4 we’ll take a look at the differences between these two types of isolation. In my experience, really understanding and being able to contrast the two is absolutely key to grasping the extent to which traditional security measures can be effective in containers, and to identifying where container-specific tooling is necessary.

You'll see how containers are built out of Linux constructs such as namespaces and `chroot`, along with cgroups, which were covered in Chapter 2. With an understanding of these constructs under your belt, you'll have a feeling for how well protected your applications are when they run inside containers.

Although the general concepts of these constructs are fairly straightforward, the way they work together with other features of the Linux kernel can be complex. Container escape vulnerabilities (for example, CVE-2019-5736, a serious vulnerability discovered in both runc and LXC) have been based on subtleties in the way that namespaces, capabilities, and filesystems interact.

## Linux Namespaces

If cgroups control the resources that a process can use, *namespaces* control what it can see. By putting a process in a namespace, you can restrict the resources that are visible to that process.

The origins of namespaces date back to the Plan 9 operating system. At the time, most operating systems had a single "name space" of files. Unix systems allowed the mounting of filesystems, but they would all be mounted into the same system-wide view of all filenames. In Plan 9, each process was part of a process group that had its own "name space" abstraction, the hierarchy of files (and file-like objects) that this group of processes could see. Each process group could mount its own set of filesystems without seeing each other.

The first namespace was introduced to the Linux kernel in version 2.4.19 back in 2002. This was the mount namespace, and it followed similar functionality to that in Plan 9. Nowadays there are several different kinds of namespace supported by Linux:

- Unix Timesharing System (UTS)—this sounds complicated, but to all intents and purposes this namespace is really just about the hostname and domain names for the system that a process is aware of.

- Process IDs
- Mount points
- Network
- User and group IDs
- Inter-process communications (IPC)
- Control groups (cgroups)
- Time

A process is always in exactly one namespace of each type. When you start a Linux system it has a single namespace of each type, but as you'll see, you can create additional namespaces and assign processes into them. You can easily see the namespaces on your machine using the `lsns` command:

```
liz@vm:~$ lsns
        NS TYPE   NPROCS    PID USER COMMAND
4026531834 time        3 848409 liz  -bash
4026531835 cgroup      3 848409 liz  -bash
4026531836 pid         3 848409 liz  -bash
4026531837 user        3 848409 liz  -bash
4026531838 uts         3 848409 liz  -bash
4026531839 ipc         3 848409 liz  -bash
4026531840 net         3 848409 liz  -bash
4026531841 mnt         3 848409 liz  -bash
```

This looks nice and neat, and there is one namespace for each of the types I mentioned previously. Sadly, this is an incomplete picture! The man page for `lsns` tells us that it "reads information directly from the */proc* filesystem and for non-root users it may return incomplete information."

The additional namespaces you might see as root are not terribly interesting until you start creating some of your own, but I mentioned it to point out that when we are using `lsns`, we should run as root (or use `sudo`) to get the complete picture.

Let's explore how you can use namespaces to create something that behaves like what we call a "container."

> **NOTE**
>
> The examples in this chapter use Linux shell commands to create a container. If you would like to try creating a container using the Go programming language, you will find instructions at *https://github.com/lizrice/containers-from-scratch*.

## Isolating the Hostname

Let's start with the namespace for the Unix Timesharing System (UTS). As mentioned previously, this covers the hostname and domain names. By putting a process in its own UTS namespace, you can change the hostname for this process independently of the hostname of the machine or virtual machine on which it's running.

If you open a terminal on Linux, you can see the hostname:

```
liz@myhost:~$ hostname
myhost
```

Most (perhaps all?) container systems give each container a random ID. By default this ID is used as the hostname. You can see this by running a container and getting shell access. For example, in Docker you could do the following:

```
liz@myhost:~$ docker run --rm -it --name hello ubuntu bash
root@cdf75e7a6c50:/$ hostname
cdf75e7a6c50
```

Incidentally, you can see in this example that even if you give the container a name in Docker (here I specified `--name hello`), that name isn't used for the hostname of the container.

The container can have its own hostname because Docker created it with its own UTS namespace. You can explore the same thing by using the unshare command to create a process that has a UTS namespace of its own.

As it’s described on the man page (seen by running `man unshare`), `unshare` lets you “run a program with some namespaces unshared from the parent.” Let’s dig a little deeper into that description. When you “run a program,” the kernel creates a new process and executes the program in it. This is done from the context of a running process—the *parent*—and the new process will be referred to as the *child*. The word “unshare” means that, rather than sharing namespaces of its parent, the child is going to be given its own.

Let’s give it a try. You need to have root privileges to do this, hence the `sudo` at the start of the line:

```
liz@myhost:~$ sudo unshare --uts sh
$ hostname
myhost
$ hostname experiment
$ hostname
experiment
$ exit
liz@myhost:~$ hostname
myhost
```

This runs a `sh` shell in a new process that has a new UTS namespace. Any programs you run inside the shell will inherit its namespaces. When you run the `hostname` command, it executes in the new UTS namespace that has been isolated from that of the host machine.

If you were to open another terminal window to the same host before the `exit`, you could confirm that the hostname hasn’t changed for the whole (virtual) machine. You can change the hostname on the host without affecting the hostname that the namespaced process is aware of, and vice versa.

This is a key component of the way containers work. Namespaces give them a set of resources (in this case the hostname) that are independent of the host machine, and of other containers. But we are still talking about a process that is being run by the same Linux kernel. This has security implications that I'll discuss later in the chapter. For now, let's look at another example of a namespace by seeing how you can give a container its own view of running processes.

## Isolating Process IDs

If you run the `ps` command inside a Docker container, you can see only the processes running inside that container and none of the processes running on the host:

```
liz@myhost:~$ docker run --rm -it --name hello ubuntu bash
root@cdf75e7a6c50:/$ ps -eaf
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 18:41 pts/0    00:00:00 bash
root        10     1  0 18:42 pts/0    00:00:00 ps -eaf
root@cdf75e7a6c50:/$ exit
liz@myhost:~$
```

This is achieved with the process ID namespace, which restricts the set of process IDs that are visible. Try running unshare again, but this time specifying that you want a new PID namespace with the `--pid` flag:

```
liz@myhost:~$ sudo unshare --pid sh
$ whoami
root
$ whoami
sh: 2: Cannot fork
$ whoami
sh: 3: Cannot fork
$ ls
sh: 4: Cannot fork
$ exit
liz@myhost:~$
```

This doesn't seem very successful—it's not possible to run any commands after the first `whoami`! But there are some interesting artifacts in this output.

The first process under sh seems to have worked OK, but every command after that fails due to an inability to fork. The error is output in the form `<command>: <process ID>: <message>`, and you can see that the process IDs are incrementing each time. Given the sequence, it would be reasonable to assume that the first whoami ran as process ID 1. That is a clue that the PID namespace is working in some fashion, in that the process ID numbering has restarted. But it's pretty much useless if you can't run more than one process!

There are clues to what the problem is in the description of the `--fork` flag in the man page for `unshare`: "Fork the specified program as a child process of unshare rather than running it directly. This is useful when creating a new pid namespace."

You can explore this by running `ps` to view the process hierarchy from a second terminal window:

```
liz@myhost:~$ ps fa
    PID TTY      STAT   TIME COMMAND
...
 924537 pts/0    Ss     0:00 -bash
 924718 pts/0    S+     0:00  \_ sudo unshare --pid sh
 924719 pts/1    Ss     0:00      \_ sudo unshare --pid sh
 924720 pts/1    S+     0:00          \_ sh
```

As you saw in Chapter 2, sudo forks itself to provide a monitor process, and then goes on to execute `unshare`. The `sh` process is not a child of `unshare`; it's a child of the (second) `sudo` process.

Now try the same thing with the `--fork` parameter:

```
liz@myhost:~$ sudo unshare --pid --fork sh
$ whoami
root
$ whoami
root
```

This is progress, in that you can now run more than one command before running into the "Cannot fork" error. If you look at the process hierarchy again from a second terminal, you’ll see an important difference:

```
liz@myhost:~$ ps fa
    PID TTY      STAT   TIME COMMAND
...
 924537 pts/0    Ss     0:00 -bash
 925113 pts/0    S+     0:00  \_ sudo unshare --pid --fork sh
 925114 pts/1    Ss     0:00      \_ sudo unshare --pid --fork sh
 925115 pts/1    S      0:00          \_ unshare --pid --fork sh
 925116 pts/1    S+     0:00              \_ sh

...
```

With the `--fork` parameter, the `sh` shell is running as a child of the `unshare` process, and you can successfully run as many different child commands as you choose within this shell.

Given that the shell is within its own process ID namespace, the results of running `ps` inside it might be surprising:

```
liz@myhost:~$ sudo unshare --pid --fork sh
$ ps
  PID TTY          TIME CMD
      1 ?        00:03:27 systemd
      2 ?        00:00:00 kthreadd
      3 ?        00:00:00 pool_workqueue_release

...many more lines of output about processes……
root      925540  925539  0 16:55 pts/1    00:00:00 unshare --pid
--fork sh
root      925541  925540  0 16:55 pts/1    00:00:00 sh
root      925543  925541 99 16:56 pts/1    00:00:00 ps -eaf
$ exit
liz@myhost:~$
```

As you can see, `ps` is still showing all the processes on the whole host, despite running inside a new process ID namespace. If you want the `ps` behavior that you would see in a Docker container, it’s not sufficient just to use a new process ID namespace, and the reason for this is included in the man page for `ps`: “This ps works by reading the virtual files in /proc.”

Let's take a look at the `/proc` directory to see what virtual files this is referring to. Your system will look similar, but not exactly the same, as it will be running a different set of processes:

```
liz@myhost:~$ ls /proc
1      29      492628  64      acpi           loadavg
10     29375   492642  65      bootconfig     locks
10927  29451   492656  7       buddyinfo      mdstat
1181   3       492664  72      bus            meminfo
...many more lines...
```

Every numbered directory in `/proc` corresponds to a process ID, and there is a lot of interesting information about a process inside its directory. For example, `/proc/<pid>/exe` is a symbolic link to the executable that's being run inside this particular process, as you can see in the following example:

```
liz@myhost:~$ ps
  PID TTY          TIME CMD
28441 pts/0    00:00:00 bash
28558 pts/0    00:00:00 ps
liz@myhost:~$ ls /proc/28441
arch_status         fdinfo              ns              smaps_rollup
attr                gid_map             numa_maps       stack
autogroup           io                  oom_adj         stat
auxv                ksm_merging_pages   oom_score       statm
cgroup              ksm_stat            oom_score_adj   status
clear_refs          latency             pagemap         syscall
cmdline             limits              patch_state     task
comm                loginuid            personality
timens_offsets
coredump_filter     map_files           projid_map      timers
cpu_resctrl_groups  maps                root
timerslack_ns
cpuset              mem                 sched           uid_map
cwd                 mountinfo           schedstat       wchan
environ             mounts              sessionid
exe                 mountstats          setgroups
fd                  net                 smaps


liz@myhost:~$ ls -l /proc/28441/exe
lrwxrwxrwx 1 liz liz 0 Oct 10 13:32 /proc/28441/exe ->
/usr/bin/bash
```

Irrespective of the process ID namespace it's running in, `ps` is going to look in `/proc` for information about running processes. In order to have `ps` return only the information about the processes inside the new namespace, there needs to be a separate copy of the `/proc` directory, where the kernel can write information about the namespaced processes. Given that `/proc` is a directory directly under root, this means changing the root directory.

## Changing the Root Directory

From within a container, you don't see the host's entire filesystem; instead, you see a subset, because the root directory gets changed as the container is created.

You can change the root directory in Linux with the chroot command. This effectively moves the root directory for the current process to point to some other location within the filesystem. Once you have done a chroot command, you lose access to anything that was higher in the file hierarchy than your current root directory, since there is no way to go any higher than root within the filesystem, as illustrated in Figure 3-1.

The description in `chroot`'s man page reads as follows: "Run COMMAND with root directory set to NEWROOT. […] If no command is given, run *${SHELL} -i* (default: */bin/sh -i*)."

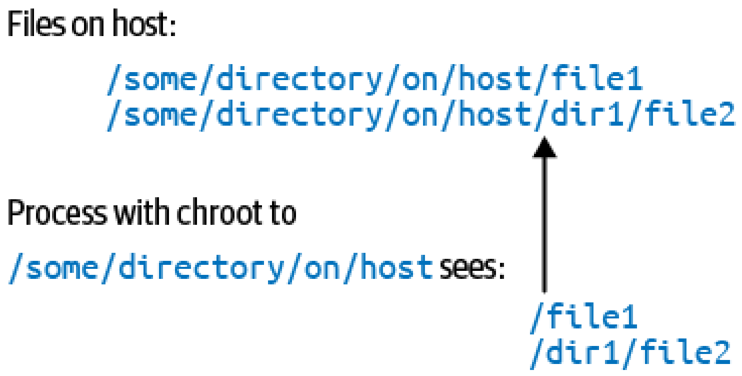

Files on host:
/some/directory/on/host/file1
/some/directory/on/host/dir1/file2
Process with chroot to
/some/directory/on/host sees:
/file1
/dir1/file2

*Figure 3-1. Changing root so a process only sees a subset of the filesystem*

From this you can see that `chroot` doesn't just change the directory, but also runs a command, falling back to running a shell if you don't specify a different command.

Create a new directory and try to `chroot` into it:

```
liz@myhost:~$ mkdir new_root
liz@myhost:~$ sudo chroot new_root
chroot: failed to run command '/bin/bash': No such file or
directory
liz@myhost:~$ sudo chroot new_root ls
chroot: failed to run command 'ls': No such file or directory
```

This doesn't work! The problem is that once you are inside the new root directory, there is no `bin` directory inside this root, so it's impossible to run the `/bin/bash` shell. Similarly, if you try to run the `ls` command, it's not there. You'll need the files for any commands you want to run to be available within the new root. This is exactly what happens in a "real" container: the container is instantiated from a container image, which encapsulates the filesystem that the container sees. If an executable isn't present within that filesystem, the container won't be able to find and run it.

Why not try running Alpine Linux within your container? Alpine is a fairly minimal Linux distribution designed for containers. You'll need to start by downloading the filesystem[1]:

```
liz@myhost:~$ mkdir alpine
liz@myhost:~$ cd alpine
liz@myhost:~/alpine$ curl -o alpine.tar.gz https://dl-
cdn.alpinelinux.org/
alpine/v3.21/releases/x86_64/alpine-minirootfs-3.21.3-
x86_64.tar.gz
  % Total    % Received % Xferd  Average Speed   Time    Time
Time  Current
                                 Dload  Upload   Total   Spent
Left  Speed
100 3425k  100 3425k    0     0  22.1M      0 --:--:-- --:--:-- -
-:--:-- 22.3M
liz@myhost:~/alpine$ tar xvf alpine.tar.gz
```

At this point you have a copy of the Alpine filesystem inside the `alpine` directory you created. Remove the compressed version and move back to the parent directory:

```
lizt@myhost:~/alpine$ rm alpine.tar.gz
liz@myhost:~/alpine$ cd ..
```

You can explore the contents of the filesystem with `ls alpine` to see that it looks like the root of a Linux filesystem with directories such as `bin`, `lib`, `var`, `tmp`, and so on.

Now that you have the Alpine distribution unpacked, you can use `chroot` to move into the `alpine` directory, provided you supply a command that exists within that directory's hierarchy.

It's slightly more subtle than that, because the executable has to be in the new process's path. This process inherits the parent's environment, including the `PATH` environment variable. The `bin` directory within `alpine` has become `/bin` for the new process, and assuming that your regular path includes `/bin`, you can pick up the `ls` executable from that directory without specifying its path explicitly:

```
liz@myhost:~$ sudo chroot alpine ls
bin    etc    lib    mnt    proc   run    srv    tmp    var
dev    home   media  opt    root   sbin   sys    usr
liz@myhost:~$
```

Notice that it is only the child process (in this example, the process that ran `ls`) that gets the new root directory. When that process finishes, control returns to the parent process. If you run a shell as the child process, it won't complete immediately, so that makes it easier to see the effects of changing the root directory:

```
liz@myhost:~$ sudo chroot alpine sh
/ $ ls
bin    etc    lib    mnt    proc   run    srv    tmp    var
dev    home   media  opt    root   sbin   sys    usr
/ $ whoami
root
```

```
/ $ exit
liz@myhost:~$
```

If you try to run the `bash` shell, it won't work. This is because the Alpine distribution doesn't include it, so it's not present inside the new root directory. If you tried the same thing with the filesystem of a distribution like Ubuntu, which does include `bash`, it would work.

To summarize, `chroot` literally "changes the root" for a process. After changing the root, the process (and its children) will be able to access only the files and directories that are lower in the hierarchy than the new root directory.

**NOTE**

In addition to `chroot`, there is a more sophisticated version called `pivot_root`. For the purposes of this chapter, whether `chroot` or `pivot_root` is used is an implementation detail; the key point is that a container needs to have its own root directory. I have used `chroot` in these examples because it is simpler and more familiar to many people.

There are security advantages to using `pivot_root` over `chroot`, so in practice you should find the former if you look at the source code of a container runtime implementation. The main difference is that `pivot_root` takes advantage of the mount namespace; the old root is no longer mounted and is therefore no longer accessible within that mount namespace. The `chroot` system call doesn't take this approach, leaving the old root accessible via mount points.

You have now seen how a container can be given its own root filesystem. I'll discuss this further in Chapter 6, but right now let's see how having its own root filesystem allows the kernel to show a container just a restricted view of namespaced resources.

## Combine Namespacing and Changing the Root

So far you have seen namespacing and changing the root as two separate things, but you can combine the two by running `chroot` in a new namespace:

```
liz@myhost:~$ sudo unshare --pid --fork chroot alpine sh
/ $ ls
bin    etc    lib    mnt    proc   run    srv    tmp    var
dev    home   media  opt    root   sbin   sys    usr
```

If you recall from earlier in this chapter (see “solating Process IDs”), giving the container its own root directory allows it to create a `/proc` directory for the container that’s independent of `/proc` on the host. For this to be populated with process information, you will need to mount it as a pseudofilesystem of type `proc`. With the combination of a process ID namespace and an independent `/proc` directory, `ps` will now show just the processes that are inside the process ID namespace:

```
/ $ mount -t proc proc proc
/ $ ps
PID   USER     TIME  COMMAND
    1 root      0:00 sh
    5 root      0:00 ps
/ $ exit
liz@myhost:~$
```

Success! It has been more complex than isolating the container’s hostname, but through the combination of creating a process ID namespace, changing the root directory, and mounting a pseudofilesystem to handle process information, you can limit a container so that it has a view only of its own processes.

If this seems complex, you might like to know that the `unshare` command has a `--mount-proc` option to simplify it.

```
liz@liz-tetragon:~$ sudo unshare --pid --fork --mount-proc bash
root@liz-tetragon:/home/liz# ps
    PID TTY          TIME CMD
      1 pts/4    00:00:00 bash
      8 pts/4    00:00:00 ps
```

There are more namespaces left to explore. Let’s see the mount namespace next.

## Mount Namespace

Typically you don’t want a container to have all the same filesystem mounts as its host. Giving the container its own mount namespace achieves this separation.

Here’s an example that creates a simple bind mount for a process with its own mount namespace:

```
liz@myhost:~$ sudo unshare --mount sh
$ mkdir source
$ touch source/HELLO
$ ls source
HELLO
$ mkdir target
$ ls target
$ mount --bind source target
$ ls target
HELLO
```

Once the bind mount is in place, the contents of the `source` directory are also available in `target`. If you look at all the mounts from within this process, there will probably be a lot of them, but the following command finds the target you created if you followed the preceding example:

```
$ findmnt target
TARGET       SOURCE                      FSTYPE OPTIONS
/home/liz/target
             /dev/sda1[/home/liz/source] ext4
rw,relatime,discard,
                                                errors=remount-ro,
commit=30
```

From the host’s perspective, this isn’t visible, which you can prove by running the same command from another terminal window and confirming that it doesn’t return anything.

Try running `findmnt` from within the mount namespace again, but this time without any parameters, and you will get a long list. You might be thinking that it seems wrong for a container to be able to see all the mounts on the host. This is a very similar situation to what you saw with the process ID namespace: the kernel uses the `/proc/<PID>/mounts` directory to communicate information about mount points for each process. If you create a process with its own mount namespace but it is using the host's `/proc` directory, you'll find that its `/proc/<PID>/mounts` file includes all the preexisting host mounts. (You can simply cat this file to get a list of mounts.)

To get a fully isolated set of mounts for the containerized process, you will need to combine creating a new mount namespace with a new root filesystem and a new `proc` mount, like this:

```
liz@myhost:~$ sudo unshare --mount chroot alpine sh
/ $ mount -t proc proc proc
/ $ mount
proc on /proc type proc (rw,relatime)
/ $ mkdir source
/ $ touch source/HELLO
/ $ mkdir target
/ $ mount --bind source target
/ $ mount
proc on /proc type proc (rw,relatime)
/dev/root on /target type ext4
(rw,relatime,discard,errors=remount-ro,commit=30)
```

Alpine Linux doesn't come with the `findmnt` command, so this example uses `mount` with no parameters to generate the list of mounts. (If you are cynical about this change, try the earlier example with `mount` instead of `findmnt` to check that you get the same results.)

You may be familiar with the concept of mounting host directories into a container using `docker run -v <host directory>:<container directory> ...`. To achieve this, after the root filesystem has been put in place for the container, the target container directory is created and then the source host directory gets bind mounted into that target. Because each container has its own mount namespace, host directories mounted like this are not visible from other containers.

> **NOTE**
>
> If you create a mount that is visible to the host, it won't automatically get cleaned up when your "container" process terminates. You will need to destroy it using `umount`. This also applies to the `/proc` pseudofilesystems. They won't do any particular harm, but if you like to keep things tidy, you can remove them with `umount proc`. The system won't let you unmount the final `/proc` used by the host.

## Network Namespace

The network namespace allows a container to have its own view of network interfaces and routing tables. When you create a process with its own network namespace, you can see it with `lsns`:

```
liz@myhost:~$ sudo lsns -t net
        NS TYPE NPROCS PID USER    NETNSID NSFS COMMAND
4026531840 net     126   1 root unassigned
/usr/lib/systemd/systemd --system --deserialize=80

liz@myhost:~$ sudo unshare --net bash
root@myhost:/home/liz$ lsns -t net
        NS TYPE NPROCS    PID USER    NETNSID NSFS COMMAND
4026531840 net     125      1 root unassigned
/usr/lib/systemd/systemd --system --deserialize=80
4026532277 net       2 927734 root unassigned      bash
```

> **NOTE**
>
> You might come across the `ip netns` command, but that is not much use to us here. Using `unshare --net` creates an anonymous network namespace, and anonymous namespaces don't appear in the output from `ip netns list`.

When you put a process into its own network namespace, it starts with just the loopback interface:

```
liz@myhost:~$ sudo unshare --net bash
root@myhost:~$ ip a
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN group default
qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```

With nothing but a loopback interface, your container won’t be able to communicate. To give it a path to the outside world, you create a virtual Ethernet interface—or more strictly, a pair of virtual Ethernet interfaces. These act as if they were the two ends of a metaphorical cable connecting your container namespace to the default network namespace.

In a second terminal window, as root on the host, you can create a virtual Ethernet pair by specifying the anonymous namespaces associated with their process IDs, like this:

```
root@myhost:~$ ip link add ve1 netns 28586 type veth peer name ve2
netns 1
```

`ip link add`

: indicates that you want to add a link.

`ve1`

: is the name of one “end” of the virtual Ethernet “cable.”

`netns 28586`

: says that this end is “plugged in” to the network namespace associated with process ID 28586 (which is shown in the output from `lsns -t net` in the example at the start of this section).

`type veth`

: shows that this a virtual Ethernet pair.

`peer name ve2`

: gives the name of the other end of the “cable.”

`netns 1`

> specifies that this second end is “plugged in” to the network namespace associated with process ID 1.

The `ve1` virtual Ethernet interface is now visible from inside the “container” process:

```
root@myhost:~$ ip a
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN group default
qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: ve1@if3: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN
group ...
    link/ether 7a:8a:3f:ba:61:2c brd ff:ff:ff:ff:ff:ff link-
netnsid 0
```

The link is in “DOWN” state and needs to be brought up before it’s any use. Both ends of the connection need to be brought up.

Bring up the `ve2` end on the host:

```
root@myhost:~$ ip link set ve2 up
```

And once you bring up the `ve1` end in the container, the link should move to “UP” state:

```
root@myhost:~$ ip link set ve1 up
root@myhost:~$ ip a
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN group default
qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: ve1@if3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
noqueue state UP ...
    link/ether 7a:8a:3f:ba:61:2c brd ff:ff:ff:ff:ff:ff link-
netnsid 0
    inet6 fe80::788a:3fff:feba:612c/64 scope link
       valid_lft forever preferred_lft forever
```

You can see that an IPv6 address has automatically been assigned to this interface in the container. Let's tell the host that the route to reach this address is through its ve2 interface, and then you can perform IPv6 ping from the host to the container:

```
root@myhost:~# ip -6 route add fe80::788a:3fff:feba:612c dev ve2
ping6 fe80::788a:3fff:feba:612c
PING fe80::788a:3fff:feba:612c (fe80::788a:3fff:feba:612c) 56 data
bytes
64 bytes from fe80::788a:3fff:feba:612c%ve2: icmp_seq=1 ttl=64
time=0.160 ms
64 bytes from fe80::788a:3fff:feba:612c%ve2: icmp_seq=2 ttl=64
time=0.052 ms
64 bytes from fe80::788a:3fff:feba:612c%ve2: icmp_seq=3 ttl=64
time=0.072 ms
```

Unlike IPv6, addresses are not automatically added to IPv4-capable interfaces, so if you want to send IPv4 traffic over the virtual ethernet connection, you'll need to define the IPv4 address at either end. In the container:

```
root@myhost:~$ ip addr add 192.168.1.100/24 dev ve1
```

And on the host:

```
root@myhost:~$ ip addr add 192.168.1.200/24 dev ve2
```

This will also have the effect of adding an IP route into the routing table in the container:

```
root@myhost:~$ ip route
192.168.1.0/24 dev ve1 proto kernel scope link src 192.168.1.100
```

As mentioned at the start of this section, the network namespace isolates both the interfaces and the routing table, so this routing information is independent of the IP routing table on the host. At this point the container can send traffic only to `192.168.1.0/24` addresses. You can test this with a ping from within the container to the remote end:

```
root@myhost:~$ ping 192.168.1.200
PING 192.168.1.200 (192.168.1.200) 56(84) bytes of data.
64 bytes from 192.168.1.200: icmp_seq=1 ttl=64 time=0.355 ms
64 bytes from 192.168.1.200: icmp_seq=2 ttl=64 time=0.035 ms
^C
```

We will dig further into networking and container network security in Chapter 10.

## User Namespace

The user namespace allows processes to have their own view of user and group IDs. Much like process IDs, the users and groups still exist on the host, but they can have different IDs. The main benefit of this is that you can map the root ID of 0 within a container to some other non-root identity on the host. This is a huge advantage from a security perspective, since it allows software to run as root inside a container, but an attacker who escapes from the container to the host will have a non-root, unprivileged identity. As you'll see in Chapter 9, it's not hard to misconfigure a container to make it easy escape to the host. With user namespaces, you're not just one false move away from host takeover.

**NOTE**

User namespace support is enabled by default from Kubernetes 1.33 onwards, although you need a Linux kernel version 6.3 or newer. It's also supported in recent version of container runtimes like containerd and runc, and can be enabled in Docker using the `--userns-remap` flag on the daemon.

Generally speaking, you need to be root to create new namespaces (which is why the Docker daemon runs as root, but the user namespace is an exception:

```
liz@myhost:~$ unshare --user bash
nobody@myhost:/home/liz$ id
uid=65534(nobody) gid=65534(nogroup) groups=65534(nogroup)
```

```
nobody@myhost:/home/liz$ echo $$
31196
```

I'll use the process ID returned by `echo $$` in a moment. First, let's notice that inside the new user namespace the user has the `nobody` ID. You need to put in place a mapping between user IDs inside and outside the namespace, as shown in Figure 3-2.

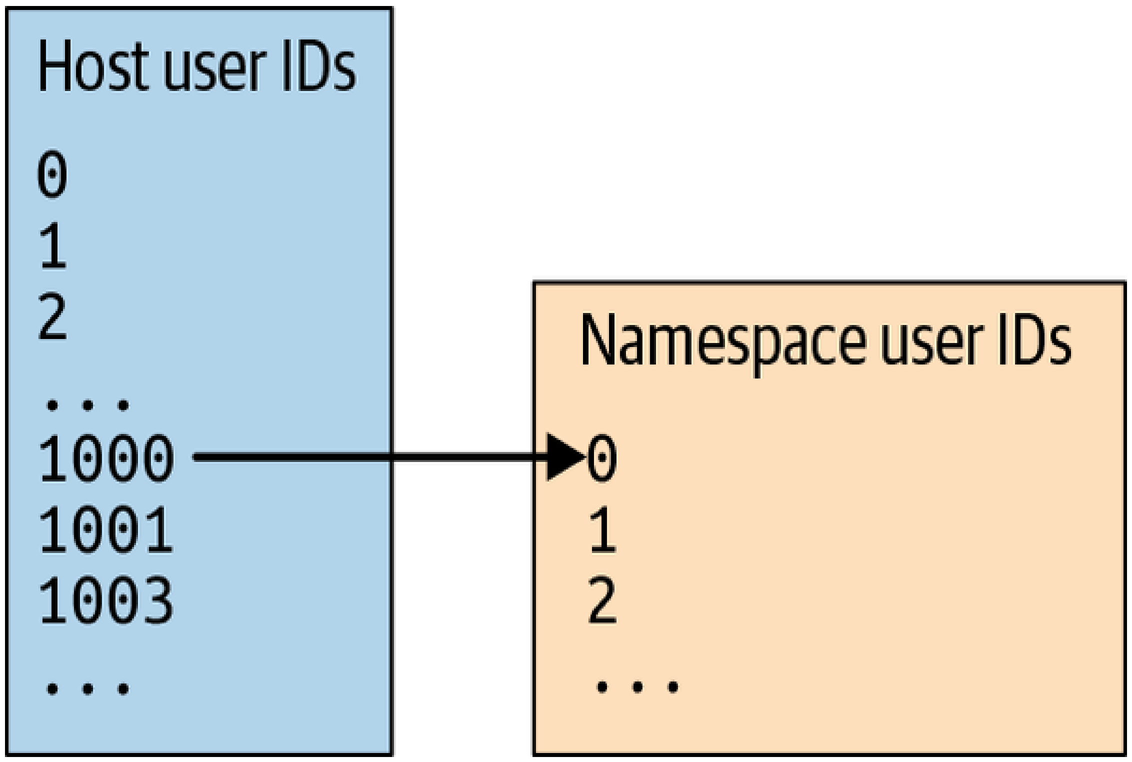


*Figure 3-2. Mapping a non-root user on the host to root in a container*

This mapping exists in `/proc/<pid>/uid_map`, which you can edit as root (on the host). There are three fields in this file:

- The lowest ID to map from the child process's perspective
- The lowest corresponding ID that this should map to on the host
- The number of IDs to be mapped

As an example, on my machine, the `liz` user has ID 1001. In order to have `liz` get assigned the root ID of 0 inside the child process, the first two fields are 0 and 1001. The last field can be 1 if you want to map only one ID (which may well be the case if you want only one user inside the container). Here's the command I used in a second terminal window to set up that mapping:

```
liz@myhost:~$ sudo echo '0 1001 1' > /proc/31196/uid_map
```

Inside its user namespace, the process has taken on the root identity. Don't be put off by the fact that the bash prompt still says "nobody"; this doesn't get updated unless you rerun the scripts that get run when you start a new shell (e.g., `~/.bash_profile`):

```
nobody@myhost:/home/liz$ id
uid=0(root) gid=65534(nogroup) groups=65534(nogroup)
```

A similar mapping process using `/proc/<pid>/gid_map` can be used to map the group(s) used inside the child process.

So now the process is running under root's user ID, and in older versions of Linux this used to be sufficient to get the full set of root's capabilities. In kernel 5.8 this was changed in important ways so that root in the child process no longer automatically gets the privileges of root across the whole host machine - it's merely a "namespace root". Let's explore what that ID looks like from the host's perspective. Start by running a sleep command:

```
nobody@myhost:/home/liz$ sleep 100 # Remember prompt has not
updated to show “root”
```

In a second terminal, let's see what this process looks like:

```
liz@myhost:~$ ps -eaf | grep sleep
liz        84714   84272  0 17:33 pts/0    00:00:00 sleep 100
liz        84716   82632  0 17:34 pts/1    00:00:00 grep --
color=auto sleep
```

The sleep command is being run under the unprivileged `liz` identity from the host’s perspective, even though this looks like root inside the user namespace.

This unprivileged user doesn’t have the `CAP_SYS_ADMIN` capability required to create, say, a new UTS namespace.

```
nobody@myhost:~$ unshare --uts
unshare: unshare failed: Operation not permitted
```

Earlier in this chapter when we looked at Isolating the Hostname I mentioned that you need to be root to run `unshare --uts` successfully. It would fail with “Operation not permitted” for exactly the same reason as in this case - no `CAP_SYS_ADMIN` capability. However, the kernel does permit a one-shot approach to creating other namespaces along with the user namespace. A regular, unprivileged user on the host can run a command like this:

```
liz@myhost:~$ unshare --user --uts sleep 100
```

Find the process ID for this sleep command in another terminal, and inspect its namespaces:

```
liz@myhost:~$ lsns -p 87982
        NS TYPE   NPROCS   PID USER COMMAND
4026531834 time        4 15244 liz  -bash
4026531835 cgroup      4 15244 liz  -bash
4026531836 pid         4 15244 liz  -bash
4026531839 ipc         4 15244 liz  -bash
4026531840 net         4 15244 liz  -bash
4026531841 mnt         4 15244 liz  -bash
4026532267 user        1 87982 liz  ├─sleep 100
4026532306 uts         1 87982 liz  └─sleep 100
```

You can see that the sleep process has inherited most of the namespaces, but it has its own user and UTS namespaces.

So, an unprivileged user can create other namespaces after all! That seems great - except done like this it’s not terribly useful. No extra capabilities are given to the user, so trying to change the hostname won’t be permitted.

```
liz@myhost:~$ unshare --user --uts hostname hello
hostname: you must be root to change the host name
```

Let’s see what happens if you try this as a privileged user.

```
liz@myhost:~$ sudo unshare --user --uts bash
nobody@myhost:/home/liz$ hostname new
hostname: you must be root to change the host name
nobody@myhost:/home/liz$ id
uid=65534(nobody) gid=65534(nogroup) groups=65534(nogroup)
```

The user inside the new namespace is `nobody`. There needs to be an explicit mapping to set the UID 0 on the host to UID 0 inside the user namespace. In a second terminal you could achieve this by writing a uid_map file for the process similar to how we did earlier, or you can use a convenient `--map-root-user option` on the `unshare` command.

```
liz@myhost:~$ sudo unshare --user --uts --map-root-user bash
root@myhost:/home/liz# id
uid=0(root) gid=65534(nogroup) groups=65534(nogroup)
root@myhost:/home/liz# cat /proc/$$/uid_map
         0          0          1
root@myhost:/home/liz# hostname new
root@myhost:/home/liz# hostname
new
```

If you’re running containers as a root user, you’ve seen that it’s easy to get root privileges inside the container, and it’s also easy to set up the container with a user namespace so that it doesn’t automatically get root privileges. This is a security benefit because fewer containers need to run as “real” root (that is, root from the perspective of the host).

If you want to run containers as an unprivileged user *and* get root privileges inside the container, that’s a bit more tricky. This concept is called *rootless containers* and we’ll come back to this in Chapter 9.

## Inter-process Communications Namespace

In Linux it’s possible to communicate between different processes by giving them access to a shared range of memory, or by using a shared message queue. The two processes need to be members of the same inter-process communications (IPC) namespace for them to have access to the same set of identifiers for these mechanisms.

Generally speaking, you *don’t* want your containers to be able to access one another’s shared memory, so they are given their own IPC namespaces.

You can see this in action by creating a shared memory block and then viewing the current IPC status with `ipcs`:

```
$ ipcmk -M 1000
Shared memory id: 0
$ ipcs

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch
status
0x74e9655a 0          root       644        1000       0

------ Semaphore Arrays --------
key        semid      owner      perms      nsems
```

In this example, the newly created shared memory block (with its ID in the `shmid` column) appears in the “Shared Memory Segments” block. A process with its own IPC namespace does not see any of these IPC objects:

```
$ sudo unshare --ipc sh
$ ipcs

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch
status
```

```
------ Semaphore Arrays --------
key        semid      owner      perms      nsems
```

## Cgroup Namespace

Since the introduction of the cgroup namespace in Linux kernel 4.6, and the adoption of cgroups v2 (discussed in Chapter 1), the use of the cgroup namespace has been key for containers and container management tools, because the cgroup has become a way to identify the parent container for a process .

The cgroup namespace is a little bit like a chroot for the cgroup filesystem; it stops a process from seeing the cgroup configuration higher up in the hierarchy of cgroup directories than its own cgroup.

**NOTE**

This section assumes that you're using cgroups v2. If you need to revisit how they work, you'll find them discussed in Chapter 1.

You can see the cgroup namespace in action by comparing the contents of `/proc/self/cgroup` outside and then inside a cgroup namespace:

```
lizt@myhost:~$ cat /proc/self/cgroup
0::/user.slice/user-1001.slice/session-357.scope
@myhost:~$
liz@myhost:~$ sudo unshare --cgroup bash
root@myhost:/home/liz# cat /proc/self/cgroup

0::/
```

The process sees a root-level cgroup. However, this process has full access to the root filesystem so looking at `/sys/fs/cgroup` shows the hosts cgroup hierarchy. For example, looking at the contents of `/sys/fs/cgroup/cgroup.procs` would show a lot of processes that are nothing to do with this process and its own control group. Similarly to how a container needs its own view of `/proc` to get a correct view of the processes inside its process ID namespace, so it needs its own version of `/sys/fs/cgroup`. As before, you'll need to create a mount namespace and change the root directory - in this example I am using the `alpine` root filesystem that we used earlier:

```
liz@myhost:~$ sudo unshare --cgroup --mount chroot alpine sh
/ # mkdir -p /sys/fs/cgroup
/ # mount -t cgroup2 none /sys/fs/cgroup
/ # ls /sys/fs/cgroup
cgroup.controllers      cpu.stat.local          memory.reclaim
cgroup.events           cpu.uclamp.max          memory.stat
cgroup.freeze           cpu.uclamp.min
memory.swap.current
cgroup.kill             cpu.weight              memory.swap.events
cgroup.max.depth        cpu.weight.nice         memory.swap.high
cgroup.max.descendants  io.pressure             memory.swap.max
cgroup.pressure         memory.current          memory.swap.peak
cgroup.procs            memory.events
memory.zswap.current
cgroup.stat             memory.events.local     memory.zswap.max
cgroup.subtree_control  memory.high
memory.zswap.writeback
cgroup.threads          memory.low              pids.current
cgroup.type             memory.max              pids.events
cpu.idle                memory.min              pids.events.local
cpu.max                 memory.numa_stat        pids.max
cpu.max.burst           memory.oom.group        pids.peak
cpu.pressure            memory.peak
cpu.stat                memory.pressure
```

The cgroup pseudo filesystem has been populated with all the information you might expect, but there is still one problem - the process IDs reflected here reflect the host's view of processes. You'll need a process ID namespace too for this to work completely as expected.

```
liz@myhost:~$ sudo unshare --mount --pid --fork --cgroup bash
root@myhost:/home/liz# mount -t proc proc alpine/proc
root@myhost:/home/liz# mount -t cgroup2 none alpine/sys/fs/cgroup
root@myhost:/home/liz# chroot alpine sh
/ # ps
PID   USER     TIME  COMMAND
    1 root      0:00 bash
   22 root      0:00 sh
   23 root      0:00 ps
```

```
/ # cat /sys/fs/cgroup/cgroup.procs
0
0
0
0
0
0
1
22
26
```

This looks pretty consistent with what you might expect, but what are those 0 entries in the `cgroup.procs` file? The answer is that these are processes in this cgroup that are outside the process ID namespace. The child process has its own view of cgroups, but it is still a member of the cgroup of its parent. The parent process can create a cgroup for the child by creating a new directory in `/sys/fs/cgroup` and write the child's process ID into the cgroup.procs.

## Time Namespace

Using the time namespace, a process can adjust its own CLOCK_MONOTONIC and CLOCK_BOOTTIME, making it seem as if the system booted at a different time. It's intended for seamless process migration between systems, for example allowing timers and sleeps to pick up where they left off, and it can be used to reproduce issues that are time-dependent (for example, if a variable's value is generated based on time).

But can you imagine the confusion caused if different containers in a distributed system all have a different view of time? For starters, trying to co-ordinate logs and metrics across different containers would get really complex! It could also open up opportunities for an attacker to obfuscate malicious activity by making it appear to happen in the past or the future. For this reason I'm not aware of any container systems that make use of the time namespace.

You have now explored all the different types of namespace and have seen how they are used along with `chroot` to isolate a process's view of its surrounding. Combine this with what you learned about cgroups in the previous chapter, and you should have a good understanding of everything that's needed to make what we call a "container."

Before moving on to the next chapter, it's worth taking a look at a container from the perspective of the host it's running on.

## Container Processes from the Host Perspective

Although they are called containers, it might be more accurate to use the term "containerized processes." A container is still a Linux process running on the host machine, but it has a limited view of that host machine, and it has access to only a subtree of the filesystem and perhaps to a limited set of resources restricted by cgroups. Because it's really just a process, it exists within the context of the host operating system, and it shares the host's kernel as shown in Figure 3-3.

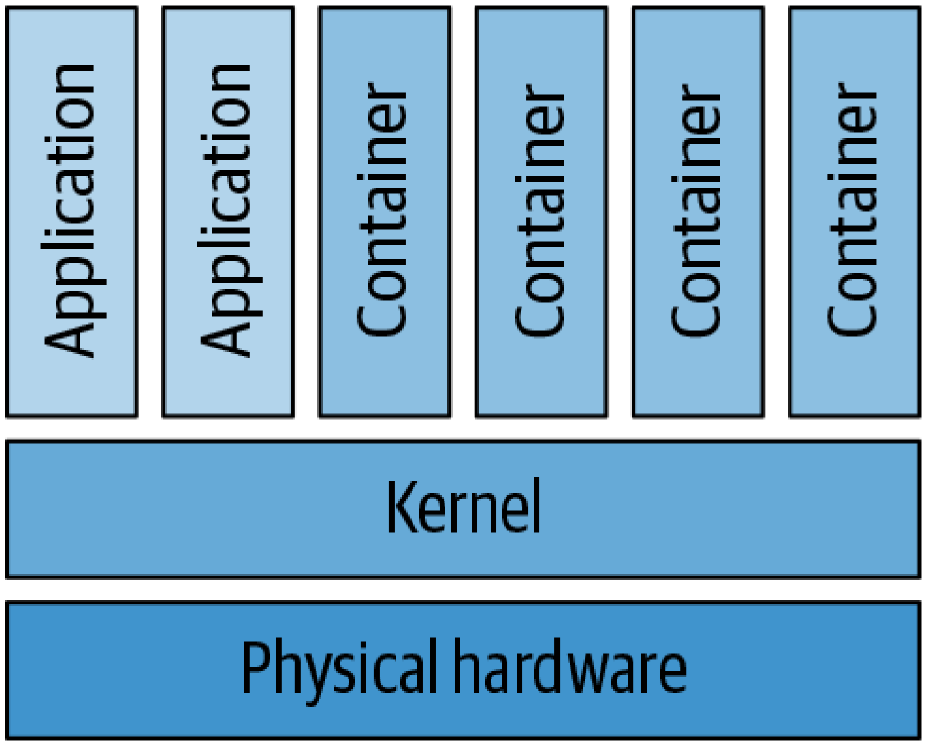


*Figure 3-3. Containers share the host's kernel*

You'll see how this compares to virtual machines in the next chapter, but before that, let's examine in more detail the extent to which a containerized process is isolated from the host, and from other containerized processes on that host, by trying some experiments on a Docker container. Start a container process based on Ubuntu (or your favorite Linux distribution) and run a shell in it, and then run a long `sleep` in it as follows:

```
$ docker run --rm -it ubuntu bash
root@1551d24a $ sleep 1000
```

This example runs the `sleep` command for 1,000 seconds, but note that the `sleep` command is running as a process inside the container. When you press Enter at the end of the `sleep` command, this triggers Linux to clone a new process with a new process ID and to run the `sleep` executable within that process.

You can put the sleep process into the background (`Ctrl-Z` to pause the process, and `bg %1` to background it). Now run `ps` inside the container to see the same process from the container's perspective:

```
me@myhost:~$ docker run --rm -it ubuntu bash
root@ab6ea36fce8e:/$ sleep 1000
^Z
[1]+  Stopped                 sleep 1000
root@ab6ea36fce8e:/$ bg %1
[1]+ sleep 1000 &
root@ab6ea36fce8e:/$ ps
  PID TTY          TIME CMD
    1 pts/0    00:00:00 bash
   10 pts/0    00:00:00 sleep
   11 pts/0    00:00:00 ps
root@ab6ea36fce8e:/$
```

While that `sleep` command is still running, open a second terminal into the same host and look at the same sleep process from the host's perspective:

```
me@myhost:~$ ps -C sleep
  PID TTY          TIME CMD
30591 pts/0    00:00:00 sleep
```

The `-C sleep` parameter specifies that we are interested only in processes running the `sleep` executable.

The container has its own process ID namespace, so it makes sense that its processes would have low numbers, and that is indeed what you see when running ps in the container. From the host's perspective, however, the sleep process has a different, high-numbered process ID. In the preceding example, there is just one process, and it has ID 30591 on the host and 10 in the container. (The actual number will vary according to what else is and has been running on the same machine, but it's likely to be a much higher number.)

To get a good understanding of containers and the level of isolation they provide, it's really key to get to grips with the fact that although there are two different process IDs, they both refer to *the same process*. It's just that from the host's perspective it has a higher process ID number.

The fact that container processes are visible from the host is one of the fundamental differences between containers and virtual machines. An attacker who gets access to the host can observe and affect *all the containers running on that host*, especially if they have root access. And as you'll see in Chapter 9, there are some remarkably easy ways you can inadvertently make it possible for an attacker to move from a compromised container onto the host.

## Container Host Machines

As you have seen, containers and their host share a kernel, and this has some consequences for what are considered best practices relating to the host machines for containers. If a host gets compromised, all the containers on that host are potential victims, especially if the attacker gains root or otherwise elevated privileges (such as being a member of the `docker` group that can administer containers where Docker is used as the runtime).

It's highly recommended to run container applications on dedicated host machines (whether they be VMs or bare metal), and the reasons mostly relate to security:

- Using an orchestrator to run containers means that humans need little or no access to the hosts. If you don't run any other applications, you will need a very small set of user identities on the host machines. These will be easier to manage, and attempts to log in as an unauthorized user will be easier to spot.

- You can use any Linux distribution as the host OS for running Linux containers, but there are several "Thin OS" distros specifically designed for running containers. These reduce the host attack surface by including only the components required to run containers. Examples include Flatcar, Talos and Bottlerocket. With fewer components included in the host machine, there is a smaller chance of vulnerabilities (see Chapter 7) in those components.

- All the host machines in a cluster can share the same configuration, with no application-specific requirements. This makes it easy to automate the provisioning of host machines, and it means you can treat host machines as immutable. If a host machine needs an upgrade, you don't patch it; instead, you remove it from the cluster and replace it with a freshly installed machine. Treating hosts as immutable makes intrusions easier to detect.

I'll come back to the advantages of immutability in Chapter 6.

Using a Thin OS reduces the set of configuration options but doesn't eliminate them completely. For example, you will have a container runtime (perhaps containerd) plus orchestrator code (perhaps the Kubernetes kubelet) running on every host. These components have numerous settings, some of which affect security. The Center for Internet Security (CIS) publishes benchmarks for best practices for configuring and running various software components, including Docker, Kubernetes, and Linux.

In an enterprise environment, look for a container security solution that also protects the hosts by reporting on vulnerabilities and worrisome configuration settings. You will also want logs and alerts for logins and login attempts at the host level.

## Summary

Congratulations! Since you've reached the end of this chapter, you should now know what a container really is. You've seen the three essential Linux kernel mechanisms that are used to limit a process's access to host resources:

- Namespaces limit what the container process can see—for example, by giving the container an isolated set of process IDs.

- Changing the root limits the set of files and directories that the container can see.

- Cgroups control the resources the container can access.

As you saw in Chapter 1, isolating one workload from another is an important aspect of container security. You now should be fully aware that all the containers on a given host (whether it is a virtual machine or a bare-metal server) share the same kernel. Of course, the same is true in a multiuser system where different users can log in to the same machine and run applications directly. However, in a multiuser system, the administrators are likely to limit the permissions given to each user; they certainly won't give them all root privileges. With containers—at least at the time of writing—they all run as root by default and are relying on the boundary provided by namespaces, changed root directories, and cgroups to prevent one container from interfering with another.

In Chapter 5 we'll explore options for strengthening the security boundary around each container, but next let's delve into how virtual machines work. This will allow you to consider the relative strengths of the isolation between containers and between VMs, especially through the lens of security.

---

[1] In this example I have used the version built for an x86-based architecture. If you're running on an ARM machine, you'll want to replace `x86_64` with `aarch64`. The latest releases of Alpine are available at *https://alpinelinux.org/releases/*.

# Chapter 4. Virtual Machines

Containers are often compared with virtual machines (VMs), especially in terms of the isolation that they offer. Let’s make sure you have a solid understanding of how VMs operate so that you can reason about the differences between them and containers. This will be particularly useful when you want to assess the security boundaries around your applications when they run in containers, or in different VMs. When you are discussing the relative merits of containers from a security perspective, understanding how they differ from VMs can be a useful tool.

This isn’t a black-and-white distinction, really. As you’ll see in Chapter 5, there are several sandboxing tools that strengthen the isolation boundaries around containers, making them more like VMs. If you want to understand the security pros and cons of these approaches, it’s best to start with a firm understanding of the difference between a VM and a “normal” container.

The fundamental difference is that a VM runs an entire copy of an operating system, including its kernel, whereas a container shares the host machine’s kernel. To understand what that means, you’ll need to know something about how virtual machines are created and managed by a Virtual Machine

Monitor (VMM). Let’s start to set the scene for that by thinking about what happens when a computer boots up.

## Booting Up a Machine

Picture a physical server. It has some CPUs, memory, and networking interfaces. When you first boot up the machine, an initial program runs that’s called the BIOS, or Basic Input Output System. It scans how much memory is available, identifies the network interfaces, and spots any other devices such as displays, keyboards, attached storage devices, and so on.

In practice, a lot of this functionality has been superseded nowadays by UEFI (Unified Extensible Firmware Interface), but for the sake of argument, let’s just think of this as a modern BIOS.

Once the hardware has been enumerated, the system runs a bootloader that loads and then runs the operating system’s kernel code. The operating system could be Linux, Windows, or some other OS. As you saw in Chapter 1, kernel code operates at a higher level of privilege than your application code. This privilege level allows it to interact with memory, network interfaces, and so on, whereas applications running in user space can’t do this directly.

On an x86 processor, privilege levels are organized into *rings*, with Ring 0 being the most privileged and Ring 3 being the least privileged. For most operating systems in a regular setup (without VMs), the kernel runs at Ring 0 and user space code runs at Ring 3, as shown in Figure 4-1.

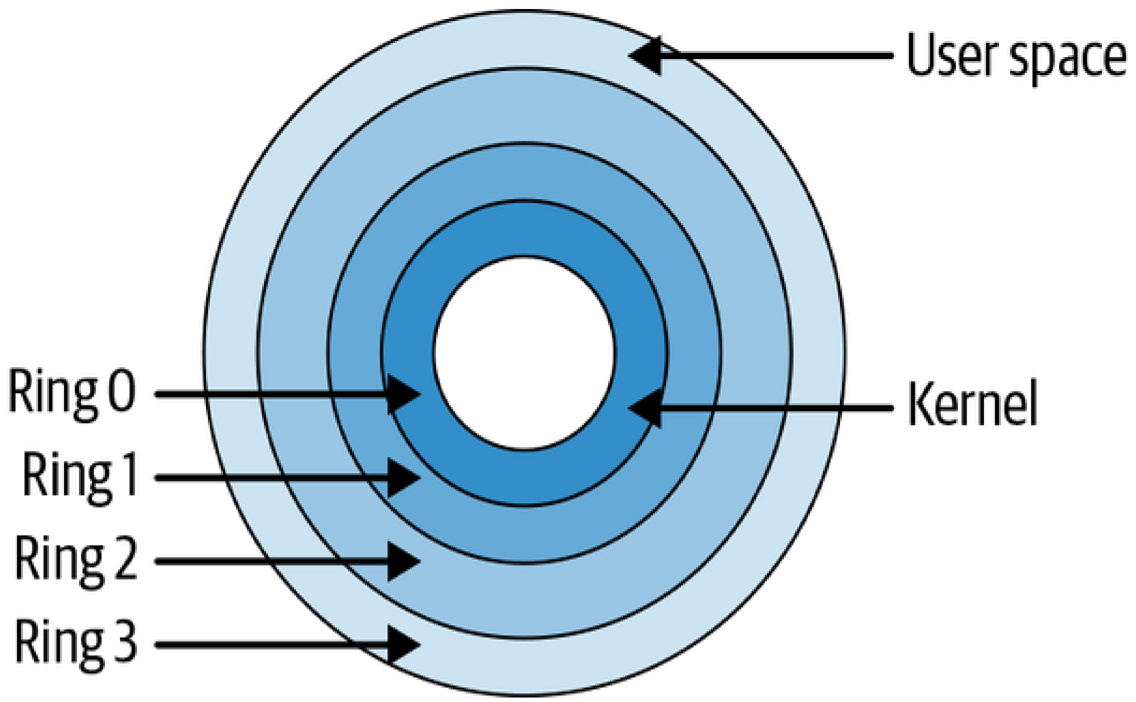


*Figure 4-1. Privilege rings*

Kernel code (like any code) runs on the CPU in the form of machine code instructions, and these instructions can include privileged instructions for accessing memory, starting CPU threads, and so on. The details of everything that can and will happen while the kernel initializes are beyond the scope of this book, but essentially the goal is to mount the root filesystem, set up networking, and bring up any system daemons. (If you want to dive deeper, there is a lot of great information on Linux kernel internals, including the bootstrap process, on GitHub.)

Once the kernel has finished its own initialization, it can start running programs in user space. The kernel is responsible for managing everything that the user space programs need. It starts, manages, and schedules the CPU threads that these programs run in, and it keeps track of these threads through its own data structures that represent processes. One important aspect of kernel functionality is memory management. The kernel assigns blocks of memory to each process and makes sure that processes can't access one another's memory blocks.

## Enter the VMM

As you have just seen, in a regular setup, the kernel manages the machine's resources directly. In the world of virtual machines, a Virtual Machine Monitor (VMM) does the first layer of resource management, splitting up the resources and assigning them to virtual machines. Each virtual machine gets a kernel of its own.

For each virtual machine that it manages, the VMM assigns some memory and CPU resources, sets up some virtual network interfaces and other virtual devices, and starts a guest kernel with access to these resources.

In a regular server, the BIOS gives the kernel the details of the resources available on the machine; in a virtual machine situation, the VMM divides up those resources and gives each guest kernel only the details of the subset that it is being given access to. From the perspective of the guest OS, it thinks it has direct access to physical memory and devices, but in fact it's getting access to an abstraction provided by the VMM.

The VMM is responsible for making sure that the guest OS and its applications can't breach the boundaries of the resources it has been allocated. For example, the guest operating system is assigned a range of memory on the host machine. If the guest somehow tries to access memory outside that range, this is forbidden.

There are two main forms of VMM, often called, not very imaginatively, Type 1 and Type 2. And there is a bit of gray area between the two, naturally!

### Type 1 VMMs, or Hypervisors

In a regular system, the bootloader runs an operating system kernel like Linux or Windows. In a pure Type 1 virtual machine environment, a dedicated kernel-level VMM program runs instead.

Type 1 VMMs are also known as *hypervisors*, and examples include Hyper-V, Xen, and ESX/ESXi. As you can see in Figure 4-2, the hypervisor runs directly on the hardware (or “bare metal”), with no operating system underneath it.

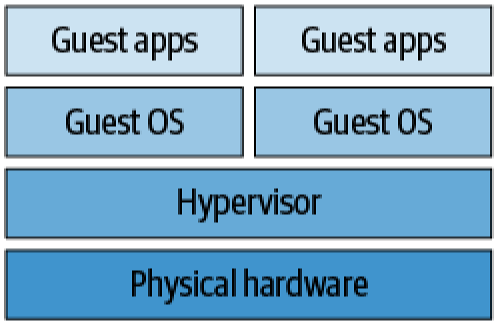


*Figure 4-2. Type 1 Virtual Machine Monitor, also known as a hypervisor*

In saying “kernel level,” I mean that the hypervisor runs at Ring 0. (Well, that’s true until we consider hardware virtualization later in this chapter, but for now let’s just assume Ring 0.) The guest OS kernel runs at Ring 1, as depicted in Figure 4-3, which means it has less privilege than the hypervisor.

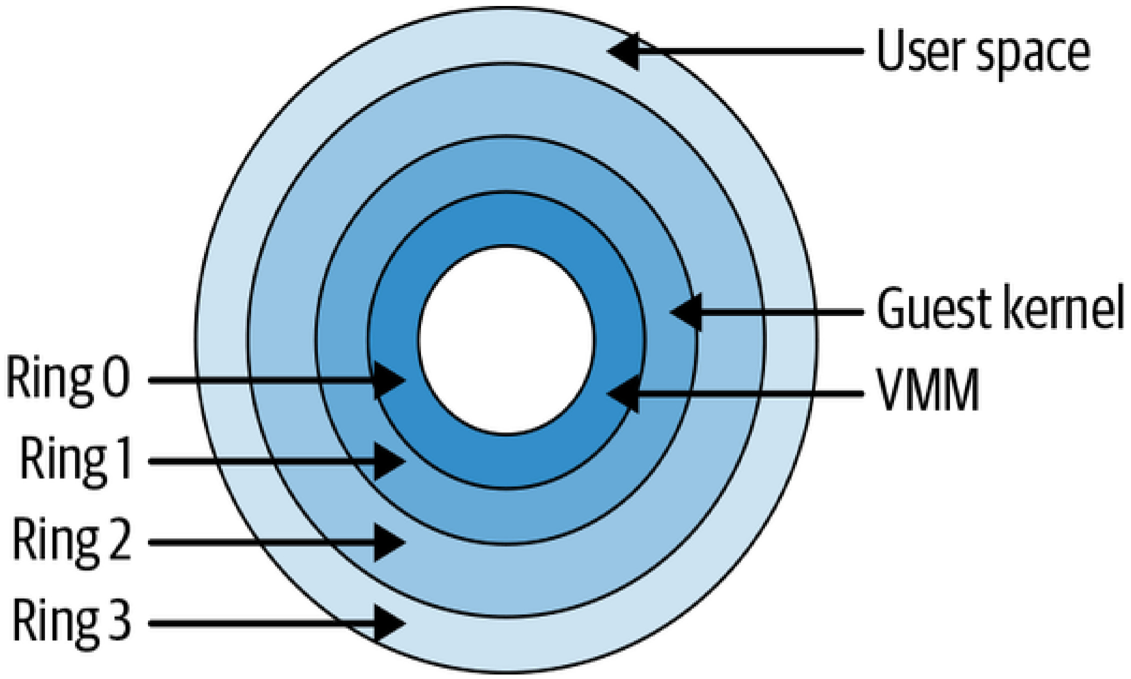


*Figure 4-3. Privilege rings used under a hypervisor*

## Type 2 VMM

When you run virtual machines on your laptop or desktop machine, perhaps through something like VirtualBox, they are "hosted" or Type 2 VMs. Your laptop might be running, say, macOS, which is to say that it's running a macOS kernel. You install VirtualBox as a separate application, which then goes on to manage guest VMs that coexist with your host operating system. Those guest VMs could be running Linux or Windows. Figure 4-4 shows how the guest OS and host OS coexist.

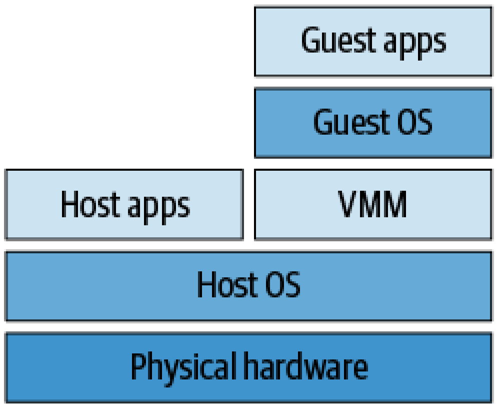


*Figure 4-4. Type 2 Virtual Machine Monitor*

Consider that for a moment and think about what it means to run, say, Linux within a macOS. By definition this means there has to be a Linux kernel, and that has to be a different kernel from the host's macOS kernel.

The VMM application has user space components that you can interact with as a user, but it also installs privileged components allowing it to provide virtualization. You'll see more about how this works later in this chapter.

Besides VirtualBox, other examples of Type 2 VMMs include Parallels and QEMU.

## Kernel-Based Virtual Machines

I promised that there would be some blurred boundaries between Type 1 and Type 2. In Type 1, the hypervisor runs directly on bare metal; in Type 2, the VMM runs in user space on the host OS. What if you run a virtual machine manager within the kernel of the host OS?

This is exactly what happens with a Linux kernel module called KVM, or Kernel-based Virtual Machines, as shown in Figure 4-5.

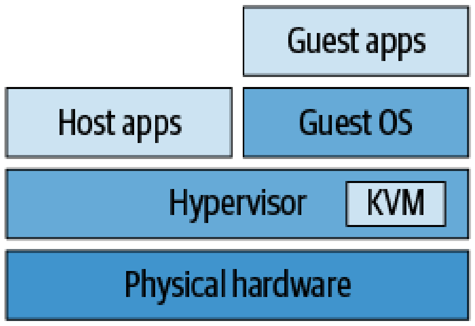


*Figure 4-5. KVM*

Generally, KVM is considered to be a Type 1 hypervisor because the guest OS doesn't have to traverse the host OS, but I'd say that this categorization is overly simplistic.

KVM is often used with QEMU (Quick Emulation), which I listed earlier as a Type 2 hypervisor. QEMU dynamically translates system calls from the guest OS into host OS system calls. It's worth a mention that QEMU can take advantage of hardware acceleration offered by KVM.

Whether Type 1, Type 2, or something in between, VMMs employ similar techniques to achieve virtualization. The basic idea is called "trap-and-emulate," though as we'll see, x86 processors provide some challenges in implementing this idea.

## Trap-and-Emulate

Some CPU instructions are *privileged*, meaning they can be executed only in Ring 0; if they are attempted in a higher ring, this will cause a *trap*. You can think of the trap as being like an exception in application software that triggers an error handler; a trap will result in the CPU calling to a handler in the Ring 0 code.

If the VMM runs at Ring 0 and the guest OS kernel code runs at a lower privilege, a privileged instruction run by the guest can invoke a handler in the VMM to emulate the instruction. In this way the VMM can ensure that the guest OSs can’t interfere with each other through privileged instructions.

Unfortunately, privileged instructions are only part of the story. The set of CPU instructions that can affect the machine’s resources is known as *sensitive*. The VMM needs to handle these instructions on behalf of the guest OS, because only the VMM has a true view of the machine’s resources. There is also another class of sensitive instructions that behaves differently when executed in Ring 0 or in lower-privileged rings. Again, a VMM needs to do something about these instructions because the guest OS code was written assuming the Ring 0 behavior.

If all sensitive instructions were privileged, this would make life relatively easy for VMM programmers, as they would just need to write trap handlers for all these sensitive instructions. Unfortunately, not all x86 sensitive instructions are also privileged, so VMMs need to use different techniques to handle them. Instructions that are sensitive but not privileged are considered to be “non-virtualizable.”

## Handling Non-Virtualizable Instructions

There are a few different techniques for handling these non-virtualizable instructions:

- One option is *binary translation*. All the non-privileged, sensitive instructions in the guest OS are spotted and rewritten by the VMM in real time. This is complex, and newer x86 processors support hardware-assisted virtualization to simplify binary translation.
- Another option is *paravirtualization*. Instead of modifying the guest OS on the fly, the guest OS is rewritten to avoid the non-virtualizable set of instructions, effectively making system calls to the hypervisor. This is the technique used by the Xen hypervisor.
- Hardware virtualization (such as Intel’s VT-x) allows hypervisors to run in a new, extra privileged level known as *VMX root mode*,

which is essentially Ring –1. This allows the VM guest OS kernels to run at Ring 0 (or VMX non-root mode), as they would if they were the host OS.

**NOTE**

If you would like to dig deeper into how virtualization works, Keith Adams and Ole Agesen provide a useful comparison and describe how hardware enhancements enable better performance.

Now that you have a picture of how virtual machines are created and managed, let's consider what this means in terms of isolating one process, or application, from another.

## Process Isolation and Security

Making sure that applications are safely isolated from each other is a primary security concern. If my application can read the memory that belongs to your application, I will have access to your data.

Physical isolation is the strongest form of isolation possible. If our applications are running on entirely separate physical machines, there is no way for my code to get access to the memory of your application.

As we have just discussed, the kernel is responsible for managing its user space processes, including assigning memory to each process. It's up to the kernel to make sure that one application can't access the memory assigned to another. If there is a bug in the way that the kernel manages memory, an attacker might be able to exploit that bug to access memory that they shouldn't be able to reach. And while the kernel is extremely battle-tested, it's also extremely large and complex, and it is still evolving. Even though we don't know of significant flaws in kernel isolation as of this writing, I wouldn't advise you to bet against someone finding problems at some point in the future.

These flaws can come about due to increased sophistication in the underlying hardware. In recent years, CPU manufacturers developed “speculative processing,” in which a processor runs ahead of the currently executing instruction and works out what the results are going to be ahead of actually needing to run that branch of code. This enabled significant performance gains, but it also opened the door to the famous Spectre and Meltdown exploits.

You might be wondering why people consider hypervisors to give greater isolation to virtual machines than a kernel gives to its processes; after all, hypervisors are also managing memory and device access and have a responsibility to keep virtual machines separate. It’s absolutely true that a hypervisor flaw could result in a serious problem with isolation between virtual machines. The difference is that hypervisors have a much, much simpler job. In a kernel, user space processes are allowed some visibility of each other; as a very simple example, you can run ps and see the running processes on the same machine. You can (given the right permissions) access information about those processes by looking in the /proc directory. You are allowed to deliberately share memory between processes through IPC and, well, shared memory. All these mechanisms, where one process is legitimately allowed to discover information about another, make the isolation weaker, because of the possibility of a flaw that allows this access in unexpected or unintended circumstances.

There is no similar equivalent when running virtual machines; you can’t see one machine’s processes from another. There is less code required to manage memory simply because the hypervisor doesn’t need to handle circumstances in which machines might share memory—it’s just not something that virtual machines do. As a result, hypervisors are far smaller and simpler than full kernels. There are well over 20 million lines of code in the Linux kernel; by contrast, the Xen hypervisor is around 50,000 lines.

Where there is less code and less complexity, there is a smaller attack surface, and the likelihood of an exploitable flaw is less. For this reason, virtual machines are considered to have strong isolation boundaries.

That said, virtual machine exploits are not unheard of. Darshan Tank, Akshai Aggarwal, and Nirbhay Chaubey describe a taxonomy of the different types of attack, and the National Institute of Standards and Technology (NIST) has published security guidelines for hardening virtualized environments.

## Disadvantages of Virtual Machines

At this point you might be so convinced of the isolation advantages of virtual machines that you might be wondering why people use containers at all! There are some disadvantages of VMs compared to containers:

- Virtual machines have start-up times that are several orders of magnitude greater than a container. After all, a container simply means starting a new Linux process, not having to go through the whole start-up and initialization of a VM. The relatively slow start-up times of VMs means that they are sluggish for auto-scaling, not to mention that fast start-up times are important when an organization wants to ship new code frequently, perhaps several times per day. (However, Amazon’s Firecracker, discussed in “Firecracker” and Intel-backed Cloud Hypervisor offer VMs with very fast start-up times, of the order of 100ms as of this writing.)

- Containers give developers a convenient ability to “build once, run anywhere” quickly and efficiently. It’s possible, but very slow, to build an entire machine image for a VM and run it on one’s laptop, but this technique hasn’t taken off in the developer community in the way containers have.

- In today’s cloud environments, when you rent a virtual machine you have to specify its CPU and memory, and you pay for those resources regardless of how much is actually used by the application code running inside it.

- Each virtual machine has the overhead of running a whole kernel. By sharing a kernel, containers can be very efficient in both resource use and performance.

When choosing whether to use VMs or containers, there are many trade-offs to be made among factors such as performance, price, convenience, risk, and the strength of security boundary required between different application workloads.

## Container Isolation Compared to VM Isolation

As you saw in Chapter 3, containers are simply Linux processes with a restricted view. They are isolated from each other by the kernel through the mechanisms of namespaces, cgroups, and changing the root. These mechanisms were created specifically to create isolation between processes. However, the simple fact that containers share a kernel means that the basic isolation is weaker compared to that of VMs.

However, all is not lost! You can apply additional security features and sandboxing to strengthen this isolation, which I will explain in Chapter 5. There are also very effective security tools that take advantage of the fact that containers tend to encapsulate microservices, and I will cover these in Chapter 13.

## Summary

You should now have a good grasp of what virtual machines are. You have learned why the isolation between virtual machines is considered strong compared to container isolation, and why containers are generally not considered suitably secure for hard multitenancy environments. Understanding this difference is an important tool to have in your toolbox when discussing container security.

Securing virtual machines themselves is outside the scope of this book, although I touched on hardening container host configuration in "Container Host Machines".

Later in this book you will see some examples in which the weaker isolation of containers (in comparison to VMs) can easily be broken through misconfiguration. Before we get to that, let’s make sure you are up to speed on what’s inside a container image and how images can have a significant bearing on security.

# Chapter 5. Strengthening Container Isolation

Back in Chapters 2 and 3, you saw how containers create some separation between workloads even though they are running on the same host. In this chapter, you'll learn about some more advanced tools and techniques that can be used to strengthen the isolation between workloads.

Suppose you have two workloads and you don't want them to be able to interfere with each other. One approach is to isolate them so that they are unaware of each other, which at a high level is really what containers and virtual machines are doing. Another approach is to limit the actions those workloads can take so that even if one workload is somehow aware of the other, it is unable to take actions to affect that workload. Isolating an application so that it has limited access to resources is known as *sandboxing*.

When you run an application as a container, the container acts as a convenient object for sandboxing. Every time you start a container, you know what application code is supposed to be running inside that container.

If the application were to be compromised, the attacker might try to run code that is outside that application's normal behavior. By using sandboxing mechanisms, we can limit what that code can do, restricting the attacker's ability to affect the system.

Several of these sandboxing approaches involve applying a profile when you start a container, where that profile defines operations that the container can or can't perform. There are also eBPF-based tools that can apply sandboxing capabilities dynamically, so that the profile can be updated or applied while a container is running. We'll cover these in Chapter X.

The first sandboxing mechanism we'll consider is *seccomp*.

## Seccomp

In "System Calls", you saw that system calls provide the interface for an application to ask the kernel to perform certain operations on the application's behalf. Seccomp is a mechanism for restricting the set of system calls that an application is allowed to make.

When it was first introduced to the Linux kernel back in 2005, seccomp (for "secure computing mode") meant that a process, once it had transitioned to this mode, could make only a very few system calls:

- `sigreturn` (return from a signal handler)
- `exit` (terminate the process)
- `read` and `write`, but only using file descriptors that were already open before the transition to secure mode

Untrusted code could be run in this mode without being able to achieve anything malicious. Unfortunately, the side effect is that lots of code couldn't really achieve anything at all useful in this mode. The sandbox was simply too limited.

In 2012, a new approach called *seccomp-bpf* was added to the kernel. This uses Berkeley Packet Filters to determine whether or not a given system call is permitted, based on a seccomp profile applied to the process. Each process can have its own profile

**NOTE**

Berkeley Packet Filters are a precursor to eBPF, which we'll cover in Chapter X.

The BPF seccomp filter can look at the system call opcode and the parameters to the call to make a decision about whether the call is permitted by the profile. In fact, it's slightly more complicated than that: the profile indicates what to do when a syscall matches a given filter, with possible actions including return an error, terminate the process, or call a tracer. But for most uses in the world of containers, the profile either permits a system call or returns an error, so you can think of it as whitelisting or blacklisting a set of system calls.

This can be very useful in the container world because there are several system calls that a containerized application really has no business trying to make, except under extremely unusual circumstances. For example, you really don't want any of your containerized apps to be able to change the clock time on the host machine, so it makes sense to block access to the syscalls `clock_adjtime` and `clock_settime`. It's unlikely that you want containers to be making changes to kernel modules, so there is no need for them to call `create_module`, `delete_module`, or `init_module`. There is a keyring in the Linux kernel, and it isn't namespaced, so it's a good idea to block containers from making calls to `request_key or keyctl`.

The Docker default seccomp profile is part of the Moby open-source project and blocks more than 40 of the 400+ syscalls (including all the examples just listed) without ill effects on the vast majority of containerized applications. Unless you have a reason not to do so, it's a good default profile to use.

Kubernetes has supported the ability to configure a `seccompProfile` setting in the `podSecurityContext` for a workload since version 1.22, and the `RuntimeDefault` option for this setting uses the default profile for the container runtime (for example `containerd` uses the Moby/Docker profile). You might want to go even further and limit a container to an even smaller group of syscalls—in an ideal world, there would be a tailored profile for each application that permits precisely the set of syscalls that it needs. There are a few different possible approaches to creating this kind of profile:

- You can use `strace` to trace out all the system calls being called by your application. Jess Frazelle described how she did this to generate and test the default Docker seccomp profile in this blog post.
- For Kubernetes deployments there is a Security Profiles Operator that can record the syscalls used by an application and then apply them as a profile. This tool can generate AppArmor and SELinux profiles as well as seccomp - we’ll discuss those shortly.
- If creating seccomp profiles yourself seems like a lot of effort, you may wish to look at commercial container security tools, some of which have the ability to observe individual workloads in order to automatically generate custom seccomp profiles.

One thing to be aware of with seccomp profiles is that system calls continue to evolve as Linux develops over time. Since writing the first edition of this book, around 100 syscalls have been added to the kernel. Generally, application developers don’t program directly to syscalls, as they are abstracted by programming language libraries, and upgrading those libraries can potentially mean a change to the underlying system calls that are used, without this change being obvious to the application developers. A strict seccomp profile might deny access to a new system call being legitimately used, so whenever the host operating system is upgraded to a new kernel version that includes new system calls, profiles might need to be updated accordingly.

NOTE

If you are interested in the underlying technology behind `strace`, you might like to watch this talk in which I create a very basic `strace` implementation in a few lines of Go.

## AppArmor

AppArmor (short for “Application Armor”) is one of a handful of Linux security modules (LSM) that can be enabled in the Linux kernel. You can check the LSMs available on a machine by looking at the contents of the `/sys/kernel/security/lsm` file.

In AppArmor, a profile can be associated with an executable file, determining what that file is allowed to do in terms of capabilities and file access permissions. You’ll recall that these were both covered in Chapter 1.

Various container runtimes include support for AppArmor, including Docker, containerd and crio.

AppArmor and other LSMs implement *mandatory access controls*. A mandatory access control is set by a central administrator, and once set, other users do not have any ability to modify the control or pass it on to another user. This is in contrast to Linux file permissions, which are *discretionary access controls*, in the sense that if my user account owns a file, I could grant your user access to it (unless this is overridden by a mandatory access control), or I could set it as unwritable even by my own user account to prevent myself from inadvertently changing it. Using mandatory access controls gives the administrator much more granular control of what can happen on their system, in a way that individual users can’t override.

AppArmor includes a “complain” mode in which you can run your executable against a profile and any violations get logged. The idea is that you can use these logs to update the profile, with the goal of eventually seeing no new violations, at which point you start to enforce the profile.Once you have a profile, you install it under the `/etc/apparmor` directory and run a tool called `apparmor_parser` to load it. See which profiles are loaded by looking at `/sys/kernel/security/apparmor/profiles`.

Running a container using `docker run --security-opt="apparmor:<profile name>"` ... will constrain the container to the behaviors permitted by the profile. By default Docker will apply a default AppArmor profile which blocks various operations such as using `ptrace` within a container. You can see the AppArmor profile applied to a running container in the output from `docker inspect <container ID>`, which shows output like this:

```
"AppArmorProfile": "docker-default"
```

You can add annotations to apply an AppArmor profile on a container in a Kubernetes pod. The Security Profile Operator mentioned earlier can build and apply AppArmor profiles specific to a workload.

## SELinux

SELinux, or “Security-Enhanced Linux,” is another type of LSM. History (or at least Wikipedia) relates that it has its roots in projects by the US National Security Agency, and it’s now an open source project primarily maintained by Red Hat. If you’re running a Red Hat distribution (RHEL, Fedora or Centos Stream) on your hosts, there is a good chance that SELinux is enabled already.

SElinux lets you constrain what a process is allowed to do in terms of its interactions with files and other processes. Each process runs under an SELinux *domain*—you can think of this as the context that the process is running in—and every file has a type. You can inspect the SELinux information associated with each file by running `ls -lZ`, and similarly you can add `-Z` to the `ps` command to get the SELinux detail for processes.

A key distinction between SELinux permissions and regular DAC Linux permissions (as seen Chapter 1) is that in SELinux, permissions have nothing to do with the user identity—they are described entirely by labels. That said, they work together, so an action has to be permitted by both DAC and SELinux.

Every file on the machine has to be labeled with its SELinux information before you can enforce policies. These policies can dictate what access a process of a particular domain has to files of a particular type. In practical terms, this means that you can limit an application to have access only to its own files and prevent any other processes from being able to access those files. In the event that an application becomes compromised, this limits the set of files that it can affect, even if the normal discretionary access controls would have permitted it. When SELinux is enabled, it has a mode in which policy violations are logged rather than enforced (similar to what we saw in AppArmor).

Manually creating an effective SELinux profile for an application takes in-depth knowledge of the set of files that it might need access to, in both happy and error paths, so that task may be best left to the app developer. Some vendors provide profiles for their applications.

SELinux is tightly integrated with Red Hat-maintained container runtimes `podman` and CRI-O. Under these runtimes, each container runs in its own SELinux domain, and file volumes can be marked with the :z or :Z flags to automatically relabel content for container access.

**NOTE**

If you are interested in learning more about SELinux, there is a good tutorial on the subject by DigitalOcean, or you might prefer Dan Walsh's visual guide.

The security mechanisms we have seen so far—seccomp, AppArmor, and SELinux—all police a process's behavior at a low level. Generating a complete profile in terms of the precise set of systems calls or capabilities needed can be a difficult job, and a small change to an application can require a significant change to the profile in order to run. The administrative overhead of keeping profiles in line with applications as they change can be a burden, and human nature means there is a tendency either to use loose profiles or to turn them off altogether. The default Docker seccomp and AppArmor profiles provide some useful guardrails if you don't have the resources to generate per-application profiles.

It's worth noting, however, that although these protection mechanisms limit what the user space application can do, there is still a shared kernel. A vulnerability within the kernel itself, like Dirty COW, would not be prevented by any of these tools.

So far in this chapter you have seen security mechanisms that can be applied to a container to limit what that container is permitted to do. Now let's turn to a set of sandboxing techniques that fall somewhere between container and virtual machine isolation, starting with gVisor.

## gVisor

Google's gVisor sandboxes a container by intercepting system calls in much the same way that a hypervisor intercepts the system calls of a guest virtual machine.

According to the gVisor documentation, gVisor is a "user-space kernel," which strikes me as a contradiction in terms but is meant to describe how a number of Linux system calls are implemented in user space through paravirtualization. As you saw in Chapter 4, paravirtualization means reimplementing instructions that would otherwise be run by the host kernel.

To do this, a component of gVisor called the Sentry intercepts syscalls from the application. Sentry is heavily sandboxed using seccomp, such that it is unable to access filesystem resources itself. When it needs to make system calls related to file access, it off-loads them to an entirely separate process called the Gofer.

Even those system calls that are unrelated to filesystem access are not passed through to the host kernel directly but instead are reimplemented within the Sentry. Essentially it's a guest kernel, operating in user space.

The gVisor project provides an executable called `runsc` that is compatible with OCI-format bundles and acts very much like the regular `runc` OCI runtime that we met in Chapter 6. Running a container with `runsc` allows you to easily see the gVisor processes. In the following example I am running the same bundle for Alpine Linux that I used in "OCI Standards":

```
$ cd alpine-bundle
$ sudo runsc run sh
```

In a second terminal you can use `runsc list` to see containers created by `runsc`:

```
$ runsc list
ID  PID    STATUS   BUNDLE                        CREATED
OWNER
sh  32258  running  /home/liz/alpine-bundle   2019-08-26T13:51:21
root
```

Inside the container, run a `sleep` command for long enough that you can observe it from the second terminal. The `runsc ps <container ID>` shows the processes running inside the container:

```
$ runsc ps sh
UID       PID       PPID      C         STIME     TIME      CMD
0         1         0         0         14:06     10ms      sh
0         15        1         0         14:15     0s        sleep
```

So far, so much as expected, but things get very interesting if you start to look at the processes from the host's perspective (the output here was edited to show the interesting parts):

```
$ ps fax
  PID TTY      STAT   TIME COMMAND
  ...
3226 pts/1    S+     0:00  |      \_ sudo runsc run sh
```

```
3227 pts/1    Sl+    0:00  |         \_ runsc run sh
3231 pts/1    Sl+    0:00  |             \_ runsc-gofer --
root=/var/run/runsc
3234 ?        Ssl    0:00  |             \_ runsc-sandbox --
root=/var/run/runsc
3248 ?        tsl    0:00  |                 \_ [exe]
3257 ?        tl     0:00  |                     \_ [exe]
3266 ?        tl     0:00  |                     \_ [exe]
3270 ?        tl     0:00  |                     \_ [exe]
  ...
```

You can see the `runsc run` process, which has spawned two processes: one is for the Gofer; the other is `runsc-sandbox` but is referred to as the Sentry in the gVisor documentation. Sandbox has a child process that in turn has three children of its own. Looking at the process information for these child and grandchild processes from the host's perspective reveals something interesting: all four of them are running the `runsc` executable. For brevity the following example shows the child and one grandchild:

```
$ ls -l /proc/3248/exe
lrwxrwxrwx 1 nobody nogroup 0 Aug 26 14:11 /proc/3248/exe ->
/usr/local/bin/runsc
$ ls -l /proc/3257/exe
lrwxrwxrwx 1 nobody nogroup 0 Aug 26 14:13 /proc/3257/exe ->
/usr/local/bin/runsc
```

Notably, none of these processes refers to the `sleep` executable that we know is running inside the container because we can see it with `runsc ps`. Trying to find that `sleep` executable more directly from the host is also unsuccessful:

```
liz@myhost:~$ sudo ps -eaf | grep sleep
liz  3554 3171  0 14:26 pts/2    00:00:00 grep --color=auto sleep
```

This inability to see the processes running inside the gVisor sandbox is much more akin to the behavior you see in a regular VM than it is like a normal container. And it affords extra protection for the processes running inside the sandbox: even if an attacker gets root access on a host, there is still a relatively strong boundary between the host and the running processes. Or least there would be, were it not for the `runsc` command itself! It offers an `exec` subcommand that we can use, as root on the host, to operate inside a running container:

```
$ sudo runsc exec sh ps
PID   USER     TIME  COMMAND
    1 root      0:00 /bin/sh
   21 root      0:00 sleep 100
   22 root      0:00 ps
```

While this isolation looks very powerful, there are limitations:

- The first is that not all Linux syscalls have been implemented in gVisor. The project has a compatibility guide showing which system calls have been implemented on Intel and ARM architectures. As noted in that guide, many languages examine the available system calls and can call back to alternatives at runtime, so the majority of applications will function within gVisor. However, if your application needs access to GPUs, this isn't supported.

- The second is performance which will likely be impacted. The gVisor project published a performance guide to help you explore this in more detail. Essentially, gVisor deliberately chooses an improved security model, sacrificing performance to achieve those security goals.

Because gVisor reimplements large parts of the kernel, it's complex, and that complexity suggests a relatively high chance of including some vulnerabilities of its own (like this privilege escalation discovered by Max Justicz).

If you're running on Google's Cloud Platform, gVisor is readily available, and you can also use it on self-managed, vanilla Kubernetes, but I'm not aware of any other managed Kubernetes services that offer it.

As you have seen in this section, gVisor provides an isolation mechanism that more closely resembles a virtual machine than a regular container.

However, gVisor affects only the way that an application accesses system calls. Namespaces, cgroups, and changing the root are still used to isolate the container.

The rest of this chapter discusses approaches that use virtual machine isolation for running containerized applications.

## Kata Containers

As you've seen in Chapter 3, when you run a regular container, the container runtime starts a new process within the host. The idea with Kata Containers is to run containers within a separate virtual machine. This approach gives the ability to run applications from regular OCI format container images, with all the isolation of a virtual machine.

For each container, Kata creates a separate virtual machine using a "micro-VMM" - a lightweight virtual machine monitor such as Firecracker, QEMU or Cloud Hypervisor - we'll consider these technologies shortly.

Like gVisor, Kata Containers make a trade-off between security and performance. For many deployments, especially where workloads are essentially trusted (for example, they are all created and operated by the same business) the additional isolation is an unnecessary cost, requiring additional memory, CPU, and impacting performance, and features like shared volumes or GPU support may not be available.

## Micro-VMMs

As you saw in "Disadvantages of Virtual Machines", conventional virtual machines are slow to start, making them unsuitable for the ephemeral workloads that typically run in containers. But what if you had a virtual machine that boots extremely quickly? Firecracker and Cloud Hypervisor are both. minimal virtual machine monitors offering the benefits of secure isolation through a hypervisor and no shared kernel, but designed specifically for containers.

These projects are all written in Rust (which as a language provides memory-safety guarantees that help to avoid vulnerabilities) and achieve startup times around 100ms. Firecracker and Cloud Hypervisor are both supported by Kata Containers for running in Kubernetes environments.

These “micro-VMMs” are able to start VMs so fast because they strip out functionality that is generally included in a kernel but that isn’t required in a container. Enumerating devices is one of the slowest parts of booting a system, but containerized applications rarely have a reason to use many devices. The main saving comes from a minimal device model that strips out all but the essential devices.

There are some differences in philosophy and background between these projects:

- Firecracker originated in AWS and is used at scale for running Lambda workloads. It is designed to provide the minimal necessary functionality for running and isolating container workloads with the fastest start-up times.
- Cloud Hypervisor supports more complex workloads such as nested virtualization, Windows as a guest OS, and GPU support.

There is one last approach to isolation that I’d like to mention in this chapter. It’s rarely used in practice but I think it’s an interesting approach that takes an even more extreme approach to reducing the size of the guest operating system: Unikernels.

## Unikernels

The operating system that runs in a virtual machine image is a general-purpose offering that you can reuse for any application. It stands to reason that apps are unlikely to use every feature of the operating system. If you were able to drop the unused parts, there would be a smaller attack surface.

The idea of Unikernels is to create a dedicated machine image consisting of the application and the parts of the operating system that the app needs.

This machine image can run directly on the hypervisor, giving the same levels of isolation as regular virtual machines, but with a lightweight startup time similar to what we see in Firecracker.

Every application has to be compiled into a Unikernel image complete with everything it needs to operate. The hypervisor can boot up this machine in just the same way that it would boot a standard Linux virtual machine image.

IBM's Nabla project is mostly inactive now, but it made use of Unikernel techniques for containers. Nabla containers use a highly restricted set of just seven system calls, with this policed by a seccomp profile. All other system calls from the application get handled within a Unikernel library OS component. By accessing only a small proportion of the kernel, Nabla containers reduce the attack surface. The downside is that you need to rebuild your applications in Nabla container form.

## Summary

In this chapter, you have seen that there are a variety of ways to isolate instances of application code from one another, which look to some degree like what we understand as a "container":

- Some options use regular containers, with additional security mechanisms applied to bolster basic container isolation: seccomp, AppArmor, SELinux. These are proven and battle-tested but also renowned for how hard they are to manage effectively.
- Where stronger boundaries are needed between containers, micro-VMMs can provide the isolation of a virtual machine, but can come with performance penalties.
- There is a third category of sandboxing techniques such as gVisor that fall somewhere between container and virtual machine isolation.

What’s right for your applications depends on your risk profile, and your decision may be influenced by the options offered by your public cloud and/or managed solution. You should also consider runtime security tools (which we’ll come to in Chapter 13) as they offer a more flexible and dynamic approach to sandboxing that might be more appropriate for your deployments. These might be an alternative to static sandboxing profiles, or they could be combined to provide defence in depth.

Regardless of the container runtime you use and the isolation it enforces, there are ways that a user can easily compromise this isolation. Move on to Chapter 9 to see how.

# Chapter 6. Passing Secrets to Containers

Application code often needs certain credentials to do its job. For example, it may need a password to access a database, or a token giving it permission to access a particular API. Credentials, or secrets, exist specifically to restrict access to resources—the database or the API in these examples. It's important to make sure that the secrets themselves stay "secret" and, in compliance with the principle of least privilege, are accessible only to people or components who really need them.

This chapter starts by considering the desirable properties of secrets and then explores the options for getting secret information into containers. It ends with a discussion of native support for secrets in Kubernetes.

## Secret Properties

The most obvious property of a secret is that it needs to be secret—that is, it must be accessible only to the people (or things) that are supposed to have access. Typically you ensure this secrecy by encrypting the secret data and sharing the decryption key only with those entities that should have permission to see the secret.

The secret should be stored in encrypted form so that it’s not accessible to every user or entity that can access the data store. When the secret moves from storage to wherever it’s used, it should also be encrypted so that it can’t be sniffed from the network. Ideally, the secret should never be written to disk unencrypted. When the application needs the unencrypted version, it’s best if this is held only in memory.

It is perhaps tempting to imagine that once you have encrypted a secret, that is the end of the matter, because you can pass it safely to another component. However, the receiver would need to know how to decrypt the information it received, and that entails a decryption key. This key is in itself a secret, and the receiver would need to get hold of that somehow, leading us back to the original question of how we can pass this next-level secret safely.

You need to be able to *revoke* secrets—that is, make them invalid in the event that the secret should no longer be trusted. This could happen if you know or suspect that an unauthorized person has been able to access the secret. You might also want to revoke secrets for straightforward operational reasons, such as someone leaving the team.

You also want the ability to *rotate* or change secrets. You won’t necessarily know if one of your secrets has been compromised, so by changing them every so often you ensure that any attacker who has been able to access some credentials will find that they stop working. It’s now well-recognized that forcing humans to change passwords regularly is a bad idea, but software components can cope fine with frequently changing credentials.

The life cycle of a secret should ideally be independent of the life cycle of the component that uses it. This is desirable because it means you don’t have to rebuild and redistribute the component when the secret changes.

The set of people who should have access to a secret is often much smaller than the set of people who need access to the application source code that will use that secret, or who can perform deployments or administration on (parts of) the deployment. For example, in a bank, it’s unlikely that developers should have access to production secrets that would grant access to account information. It’s quite common for secret access to be write-only for humans: once a secret is generated (often automatically and at random), there may never be a reason for a person to legitimately read the secret out again.

It’s not just people who should be restricted from having access to secrets. Ideally, the only software components that can read the secret should be those that need access to it. Since we are concerned with containers, this means exposing a secret only to those containers that actually need it to function correctly.

Now that we have considered the preferred qualities of a secret, let’s turn to the possible mechanisms that could be used to get a secret into the application code running in a container.

## Getting Information into a Container

Bearing in mind that a container is deliberately intended to be an isolated entity, it should be no surprise that there is a limited set of possibilities for getting information—including secret data—into a running container:

- Data can be included in the container image, as a file in the image root filesystem.
- Environment variables can be defined as part of the configuration that goes along with the image (see Chapter 6 for a reminder of how the root filesystem and config information make up an image).
- The container can receive information over a network interface.
- Environment variables can be defined or overridden at the point where the container is run (for example, including `-e` parameters on a `docker run` command).

- The container can mount a volume from the host, and read information out of volumes on that host.

Let’s take each of these options in turn.

## Storing the Secret in the Container Image

The first two of these options are unsuitable for secret data because they require you to hardcode the secret into the image at build time. While this is certainly possible, it is generally considered a bad idea:

- The secret is viewable by anyone who has access to the source code for the image. You might be thinking that the secret could be encrypted rather than in plain text in the source code—but then you’ll need to pass in another secret somehow so that the container can decrypt it. What mechanism will you use to pass in this second secret?
- The secret can’t change unless you rebuild the container image, but it would be better to decouple these two activities. Furthermore, a centralized, automated system for managing secrets (like CyberArk or Hashicorp Vault) can’t control the life cycle of a secret that is hardcoded in the source.

Unfortunately, it is surprisingly common to find secrets baked into source code. One reason is simply that developers don’t all know that it’s a bad idea; another is that it’s all too easy to put the secrets directly into the code as a shortcut during development or testing, with the intention of removing them later—and then simply forget to come back and take them out.

Several image scanning tools (discussed in Chapter X) can help you spot when secrets have been hard-coded into a container image, so you can remove them and use a better mechanism instead!

If passing the secret at build time is off the table, the other options all pass the secret when the container starts or is running.

## Passing the Secret Over the Network

The third option, passing the secret over a network interface, requires your application code to make the appropriate network calls to retrieve or receive the information, and as a result it is the approach I have seen least often in the wild.

In addition, there is the question of encrypting the network traffic that carries the secret, which necessitates another secret, typically in the form of an X.509 certificate (see Chapter 11). You could offload this part of the problem to a service mesh, which can be configured to ensure that network connections use encryption for security. We'll discuss service meshes further in Chapter X.

If you use managed services like AWS Secrets Manager or Hashicorp Vault to hold secrets, your applications can retrieve secrets from them using an interface that abstracts an underlying network communication with the secrets service. Accessing these services also requires secret credentials which have to be passed into the container using some other mechanism.

## Passing Secrets in Environment Variables

The fourth option, passing secrets via environment variables, is generally frowned upon for a couple of reasons:

- In many languages and frameworks, a crash will result in the system dumping debug information that may well include all the environment settings. If this information gets passed to a logging system, anyone who has access to the logs can see secrets passed in as environment variables.
- If you can run `docker inspect` (or an equivalent) on a container, you get to see any environment variables defined for the container, whether at build or at runtime. Administrators who have good reasons for inspecting properties of a container don't necessarily need access to the secrets.

Here's an example of extracting the environment variables from a container image:

```
liz@myhost:~$ docker image inspect --format '{{.Config.Env}}'
nginx
[PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bi
n NGINX_VERSION=1.27.5 NJS_VERSION=0.8.10 NJS_RELEASE=1~bookworm
PKG_RELEASE=1~bookworm DYNPKG_RELEASE=1~bookworm]
```

You can also easily inspect environment variables at runtime. This example shows how the results include any definitions passed in on the `run` command (`EXTRA_ENV` here).

```
liz@myhost:~$ docker run -e EXTRA_ENV=HELLO --rm -d nginx
13bcf3c571268f697f1e562a49e8d545d78aae65b0a102d2da78596b655e2f9a
liz@myhost:~$ docker container inspect --format '{{.Config.Env}}'
13bcf
[EXTRA_ENV=HELLO
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
NGINX_VERSION=1.27.5 NJS_VERSION=0.8.10 NJS_RELEASE=1~bookworm
PKG_RELEASE=1~bookworm DYNPKG_RELEASE=1~bookworm]
```

The 12-factor app manifesto encouraged developers to pass configuration through environment variables, so in practice you may find yourself running third-party containers that expect to be configured this way, including some secret values. You can mitigate the risk of environment variable secrets in a few ways (which may or may not be worthwhile, depending on your risk profile):

- You could process output logs to remove or obscure the secret values.
- You can modify the app container (or use a sidecar container) to retrieve the secrets from a secure store (like Hashicorp Vault, CyberArk Conjur, or cloud provider secret/key management systems). Some commercial security solutions will provide this integration for you.

AWS Fargate is an example of a managed container service that supports passing secrets using environment variables. Instead of including the secret value directly in the configuration for the Fargate task, the task definition can reference secrets held safely and in encrypted form in AWS Secrets Manager. This means the task definition itself doesn’t include sensitive data (which would be similar to holding sensitive data in the source code for a container image). Still, by the time the containerized application running in Fargate sees the value retrieved from the Secrets Manager service, it will be an unencrypted environment variable.

One last thing to note about secrets configured through environment variables is that the environment for a process is configured only once, and that’s at the point where the process is created. If you want to rotate a secret, you can’t reconfigure the environment for the container from the outside.

### Passing Secrets Through Files

A better option for passing secrets is to write them into files that the container can access through a mounted volume. Ideally, this mounted volume should be a temporary directory that is held in memory rather than written to disk - as an example, Docker Swarm secrets are mounted into containers using an in-memory filesystem. Combining this with a secure secrets store ensures that secrets are never stored “at rest” unencrypted.

Because the file is mounted from the host into the container, it can be updated from the host side at any time without having to restart the container. Provided the application knows to retrieve a new secret from the file if the old secret stops working, this means you can rotate secrets without having to restart containers. The requirement for applications to be aware of updated secrets has been made easier through Linux’s *inotify* mechanism, where the filesystem can send an event to let a process know when a file has changed.

## Kubernetes Secrets

If you’re using Kubernetes, the good news is that it has native secrets support that meets many of the criteria I described at the start of this chapter:

- Kubernetes Secrets are created as independent resources, so they are not tied to the life cycle of the application code that needs them.
- Kubernetes secrets are stored (along with other resource data) as base64-encoded values in etcd. Data at rest in etcd is not encrypted by default, but Kubernetes has built-in support that you can enable for encrypting Secrets (and any other resources of your choosing that you might consider sensitive). If you’re using a managed Kubernetes service you’ll very likely find that this encryption is either on by default or easily configurable. (It’s also possible to encrypt the entire etcd data store, but this is rarely done since Kubernetes started offering resource encryption at the API server level, which is usually easier to manage.)
- Secrets are encrypted in transit between components. This requires that you have secure connections between Kubernetes components (for example, a TLS connection between the API Server and etcd data store) though this is generally the case by default in most distributions.
- Kubernetes Secrets support the file mechanism as well as the environment variable method, mounting secrets as files in a temporary filesystem that is held in-memory and never written to disk.
- You can set up Kubernetes RBAC (role-based access control) so that users can configure Secrets resources but can’t access them again, giving them write-only permissions.

In addition to the native Secrets support, Kubernetes now has an optional Secrets Store CSI (Container Storage Interface) Driver that eliminates the need to use native Kubernetes Secrets.

## Secrets Store CSI Driver

This extension allows secrets to be pulled directly from a secure secret management service (like Vault or a cloud provider Key Management Service) at runtime, and mounted into pods as files. These secrets are never stored in etcd and never exposed as environment variables.

To start using this approach you might need to update your applications to read secrets from the right files, and they need to be restarted on key rotation (unless they can watch for updates using inotify).

There is the option to sync these resources to native Kubernetes Secrets, though this would seem to defeat the whole point of using the Secrets Store driver! However, the ability to sync can help during a migration in which certain apps need the legacy Secrets approach, for example because they read from environment variables. Additionally, some resources might refer to Secrets (for example the Ingress resource can look for TLS certifiates by reference to a Secret resource).

## Rotating secrets in Kubernetes

When it comes to rotating secrets, there are two aspects to consider:

- Rotating the value of a Kubernetes Secret being passed to an application
- Rotating the keys in the EncryptionConfig resource used to encrypt Secret resources in etcd

If you're using plain Secret objects, you can update their values with `kubectl`, and then you will generally need to restart pods that use those secrets to get the application to use the new values. Secret managers (like Vault or AWS Secrets Manager) can make this process easier.

Rotating the keys used for encrypting Secret resources is a multi-step process that involves modifying the EncryptionConfig resource, and restarting the API Server(s) at least twice:

1. Add the new key as a second entry in the EncryptionConfig resource.
2. Restart the API Servers to read the new EncryptionConfig. They have access to the new key and can use it for decryption if they encounter a resource that they can’t decrypt with the old key.
3. Swap the keys so that the new key is the first entry in the EncryptionConfig resource.
4. Restart the API Servers to re-read the EncryptionConfig, so they start using the new key for encryption.
5. Replace all the existing Secret resources so they are encrypted with the new key.
6. It’s a good idea to update the EncryptionConfig to remove the old key.

In my experience most enterprises choose a third-party commercial solution for secret storage, either from their cloud provider (such as the AWS Key Management System, or its Azure or GCP equivalents), or from a vendor such as Hashicorp or CyberArk. These offer several benefits:

- A dedicated secrets management system can be shared with multiple clusters. Secret values can be rotated, irrespective of the life cycle of the application cluster(s).
- These solutions can make it easier for organizations to standardize on one way of handling secrets, with common best practices for management and consistent logs and auditing of secrets.

NOTE

The public cloud providers all document their recommendations for encrypting Kubernetes secrets:

- aAWS documentation for using Key Management Service with EKS
- Microsoft documentation for using Key Management Service with AKS
- Google documentation for using a

## Secrets Are Accessible by Root

Whether a secret is passed into a container as a mounted file or as an environment variable, it is going to be possible for the root user on the host to access it.

If the secret is held in a file, that file lives on the host's filesystem somewhere. Even if it's in a temporary directory, the root user will be able to access it. As a demonstration of this you can list the temporary filesystems mounted on a Kubernetes node, and you'll find something like this:

```
root@myhost:/$ mount -t tmpfs
...
tmpfs on /var/lib/kubelet/pods/f02a9901-8214-4751-b157-
d2e90bc6a98c/volumes/kuber
netes.io~secret/coredns-token-gxsqd type tmpfs (rw,relatime)
tmpfs on /var/lib/kubelet/pods/074d762f-00ed-48e6-a22f-
43fc673df0e6/volumes/kuber
netes.io~secret/kube-proxy-token-bvktc type tmpfs (rw,relatime)
tmpfs on /var/lib/kubelet/pods/e1bad0db-8c0b-4d7b-8867-
9fc019de258f/volumes/kuber
netes.io~secret/default-token-h2x8p type tmpfs (rw,relatime)
...
```

Using the directory names included in this output, the root user has no difficulty accessing the secret files held within them.

Extracting the secrets held in environment variables is almost as simple for the root user. Let's demonstrate this by starting a container with Docker on the command line, passing in an environment variable:

```
liz@myhost:~$ docker run --rm -it -e SECRET=mysecret ubuntu sh
$ env
...
SECRET=mysecret
...
```

This container is running `sh`, and from another terminal you can find the process ID for that executable:

```
liz@myhost:~$ ps -C sh
  PID TTY          TIME CMD
17322 pts/0    00:00:00 sh
```

In Chapter 3 you saw that lots of interesting information about a process is held in the `/proc` directory. That includes all its environment variables, held in `/proc/<process ID>/environ`:

```
liz@myhost:~$ sudo cat /proc/17322/environ
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=2cc99c98ba5aTERM=xtermSECRET=mysecretHOME=/root
```

As you can see, any secret passed in through the environment can be read in this way. Are you wondering whether it wouldn't be better to encrypt the secret first? Think about how you would get the decryption key—which also needs to be kept secret—into the container.

I can't overemphasize that anyone who has root access to a host machine has carte blanche over everything on that machine, including all its containers and their secrets. This is why it's so important to prevent unauthorized root access within your deployment, and why running as root inside a container is so dangerous: since root inside the container is root on the host, it is just one step away from compromising everything on that host.

## Summary

If you have worked through the book to this point, you should have a good understanding of how containers work, and you know how to send secret information safely between them. You have seen numerous ways in which containers can be exploited, and many ways in which they can be protected.

The last group of protection mechanisms we shall consider relates to *runtime protection*, coming up in the next chapter.

## About the Author

**Liz Rice** is the chief open source officer with eBPF specialists at Isovalent, creators of the Cilium cloud native networking, security, and observability project. She sits on the CNCF Governing Board and on the Board of OpenUK. She was chair of the CNCF's Technical Oversight Committee in 2019–2022 and cochair of KubeCon + CloudNativeCon in 2018. She is also the author of *Container Security*, published by O'Reilly.