CHAPTER

1

What Is A Container?

Linux containers as we know them today have been realized through a series of incremental innovations, courtesy of a disparate group of protagonists. Destined for a place in the history books, containers have brought significant change to the way that modern software is now developed; this change will be intriguing to look back upon in the coming years ahead.

In simple terms, a *container* is a distinct and relatively isolated unit of code that has a specific purpose. As will be repeated later, the premise of a container is to focus on one key process (such as a web server) and its associated processes. If your web server needs to be upgraded or altered, then no other software components are affected (such as a related database container), making the construction of a technology stack more modular by design.

In this chapter, we will look at how a container is constructed and some of its fundamental components. Without this background information it is difficult to understand how to secure a containerized server estate successfully. We will start by focusing on the way software runs containers; we call that software the *container runtime*. We will focus on the two most prominent runtimes, Docker and Podman. An examination of the latter should also offer a valuable insight into the relatively recent advances in container runtimes.

As we work through the book, we will look at this area again from a more advanced perspective with an eye firmly on security mitigation. Purposely, rather

than studying historical advances of Linux containers, this chapter focuses on identifying the components of a container that security personnel should be concerned about.

Common Misconceptions

In 2014–15, the clever packaging of system and kernel components by Docker Inc. led to an explosion of interest in Linux containers. As Docker's popularity soared, a common misconception was that containers could be treated in the same way as virtual machines (VMs). As technology evolved, this became partially true, but let us consider what that misconception involved to help illustrate some of the security challenges pertinent to containers.

Along the same lines as most VMs, less-informed users trusted that Customer A had no access to Customer B's resources if each customer ran its own containers. This implicit trust is understandable. Hardware virtualization is used often on Linux systems, implemented with tools like the popular Kernel-based Virtual Machine, or KVM (`www.linux-kvm.org`), for example. Virtual machines using such technologies can run on the same physical machine and do indeed share significant levels of segregation, improving their security posture significantly. Helpful information is provided in a white paper by a long-standing commercial brand, VMware, that offers a detailed look at how this works.

```
www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/
whitepaper/techpaper/vmw-white-paper-secrty-vsphr-hyprvsr-uslet-101
.pdf
```

This type of virtualization is not to be confused with *paravirtualization*, utilized by software such as Xen (`xenproject.org`), where guest operating systems (OSs) can share hardware on a modified host OS.

NOTE **Xen is able to support hardware virtualization and paravirtualization. You can find more information on the subject here:**

`wiki.xen.org/wiki/Xen_Project_Software_Overview#PV_.28x86.29`

In Figure 1.1 we can see the difference between containers and virtual machines. The processes shown are those relevant to a running application. Using our web server example again, one process might be running a web server listening on an HTTP port and another for HTTPS. As mentioned, to maintain the desired modularity, containers should service a specific single task (such as a web server). Normally, they will run a main application's process alone, along with any required associated processes.

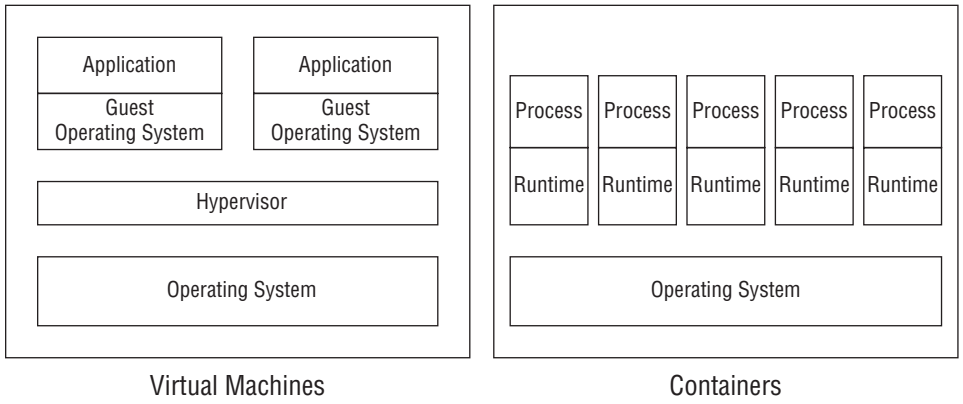

Figure 1.1: How virtual machines and containers reside on a host

It should be clear that a Linux container is an entirely different animal than a VM. A saying that appears to have gained popularity at Red Hat during the explosion of container popularity noted earlier is that fundamentally "containers are Linux." One interpretation of such a statement is that if you can appreciate how a Linux system is constructed at a nuts-and-bolts level and understand how to slice up a system into small segments, each of which uses native Linux components, then you will have a reasonable chance of understanding what containers are. For a more specific understanding of where that phrase comes from, visit this Red Hat blog page that explains the motivation behind the phrase: `www.redhat.com/en/blog/containers-are-linux`.

From the perspective of an underlying host machine, the operating system is not only slicing memory up to share among containers, segmenting the networking stack, dividing up the filesystem, and restricting full access to the CPU; it is also hiding some of the processes that are running in the process table. How are all those aspects of a Linux system controlled centrally? Correct, via the kernel. During the massive proliferation of Docker containers, it became obvious that users did not fully appreciate how many of the components hung together.

For example, the Docker runtime has been improved over time with new security features (which we look at in more detail in Chapter 2, "Rootless Runtimes"); but in older versions, it needed to run as the `root` user without exception. Why? It was because in order to slice up the system into suitable container-shaped chunks, superuser permissions were needed to convince the kernel to allow an application like Docker to do so.

One example scenario (which is common still to this day) that might convey why running as the `root` user is such a problem involves the popular continuous integration/continuous development (CI/CD) automation tool, Jenkins.

TIP **Security in the CI/CD software development pipeline is the subject of the chapters in Part II of this book, "DevSecOps Tooling."**

Imagine that a Jenkins job is configured to run from a server somewhere that makes use of Docker Engine to run a new container; it has built the container image from the Dockerfile passed to it. Think for a second—even the seemingly simplest of tasks such as running a container always used to need `root` permissions to split up a system's resources, from networking to filesystem access, from kernel namespaces to kernel control groups, and beyond. This meant you needed blind faith in the old (now infamous) password manager in Jenkins to look after the password that ran the Jenkins job. That is because as that job executed on the host, it would have `root` user permissions.

What better way to examine how a system views a container—which, it is worth repeating, is definitely not a virtual machine—than by using some hands-on examples?

Container Components

There are typically a number of common components on a Linux system that enable the secure use of containers, although new features, or improvements to existing kernel and system features, are augmented periodically. These are Linux security features that allow containers to be bundled into a distinct unit and separated from other system resources. Such system and kernel features mean that most containers spawned, without adding any nonstandard options to disable such security features, have a limited impact on other containers and the underlying host. However, often unwittingly containers will run as the `root` user or developers will open security features to ease their development process. Table 1.1 presents key components.

Table 1.1: Common Container Components

COMPONENT	DESCRIPTION
Kernel namespaces	A logical partitioning of kernel resources to reduce the visibility that processes receive on a system.
Control croups	Functionality to limit usage of system resources such as I/O, CPU, RAM, and networking. Commonly called *cgroups*.
SElinux/AppArmor	Mandatory Access Control (MAC) for enforcing security-based access control policies across numerous system facets such as filesystems, processes, and networking. Typically, SElinux is found on Red Hat Enterprise Linux (RHEL) derivatives and AppArmor on Debian derivatives. However, SElinux is popular on both, and AppArmor appears to be in experimental phase for RHEL derivatives such as CentOS.
Seccomp	Secure Computing (seccomp) allows the kernel to restrict numerous system calls; for the Docker perspective, see `docs.docker.com/engine/security/seccomp`.

COMPONENT	DESCRIPTION
Chroot	An isolation technique that uses a pseudo root directory so that processes running within the chroot lose visibility of other defined facets of a system.
Kernel capabilities	Checking and restricting all system calls; more in the next section.

Kernel Capabilities

To inspect the innards of a Linux system and how they relate to containers in practice, we need to look a little more at kernel capabilities. The kernel is important because before other security hardening techniques were introduced in later versions, Docker allowed (and still does) the ability to disable certain features, and open up specific, otherwise locked-down, kernel permissions.

You can find out about Linux kernel capabilities by using the command `$ man capabilities` (or by visiting `man7.org/linux/man-pages/man7/capabilities.7.html`).

The manual explains that capabilities offer a Linux system the ability to run permission checks against each system call (commonly called *syscalls*) that is sent to the kernel. Syscalls are used whenever a system resource requests anything from the kernel. That could involve access to a file, memory, or another process among many other things, for example. The manual explains that during the usual run of events on traditional Unix-like systems, there are two categories of processes: any privileged process (belonging to the `root` user) and unprivileged processes (which don't belong to the `root` user). According to the Kernel Development site (`lwn.net/1999/1202/kernel.php3`), kernel capabilities were introduced in 1999 via the v2.1 kernel. Using kernel capabilities, it is possible to finely tune how much system access a process can get without being the `root` user.

By contrast, *cgroups* or *control groups* were introduced into the kernel in 2006 after being designed by Google engineers to enforce quotas for system resources including RAM and CPU; such limitations are also of great benefit to the security of a system when it is sliced into smaller pieces to run containers.

The problem that kernel capabilities addressed was that privileged processes bypass all kernel permission checks while all nonroot processes are run through security checks that involve monitoring the user ID (UID), group ID (GID), and any other groups the user is a member of (known as *supplementary groups*). The checks that are performed on processes will be against what is called the *effective UID* of the process. In other words, imagine that you have just logged in as a nonroot user `chris` and then elevate to become the `root` user with an `su-` command. Your "real UID" (your login user) remains the same; but after you elevate to become the superuser, your "effective UID" is now 0, the UID for the `root` user. This is an important concept to understand for security, because

security controls need to track both UIDs throughout their lifecycle on a system. Clearly you don't want a security application telling you that the `root` user is attacking your system, but instead you need to know the "real UID," or the login user `chris` in this example, that elevated to become the `root` user instead. If you are ever doing work within a container for testing and changing the USER instruction in the Dockerfile that created the container image, then the `id` command is a helpful tool, offering output such as this so you can find out exactly which user you currently are:

```
uid=0(root) gid=0(root) groups=0(root)
```

Even with other security controls used within a Linux system running containers, such as namespaces that segregate access between pods in Kubernetes and OpenShift or containers within a runtime, it is highly advisable never to run a container as the `root` user. A typical Dockerfile that prevents the `root` user running within the container might be created as shown in Listing1.1.

Listing 1.1: A Simple Example Dockerfile of How to Spawn a Container as Nonroot

```
FROM debian:stable
USER root
RUN apt-get update && apt-get install -y iftop && apt-get clean
USER nobody
CMD bash
```

In Listing 1.1, the second line explicitly states that the `root` user is initially used to create the packages in the container image, and then the `nobody` user actually executes the final command. The `USER root` line isn't needed if you build the container image as the `root` user but is added here to demonstrate the change between responsibilities for each `USER` clearly.

Once an image is built from that Dockerfile, when that image is spawned as a container, it will run as the `nobody` user, with the predictable UID and GID of 65534 on Debian derivatives or UID/GID 99 on Red Hat Enterprise Linux derivatives. These UIDs or usernames are useful to remember so that you can check that the permissions within your containers are set up to suit your needs. You might need them to mount a storage volume with the correct permissions, for example.

Now that we have covered some of the theory, we'll move on to a more hands-on approach to demonstrate the components of how a container is constructed. In our case we will not use the dreaded `--privileged` option, which to all intents and purposes gives a container `root` permissions. Docker offers the following useful security documentation about privileges and kernel capabilities, which is worth a read to help with greater clarity in this area:

```
docs.docker.com/engine/reference/run/
#runtime-privilege-and-linux-capabilities
```

The docs describe Privileged mode as essentially enabling "...access to all devices on the host as well as [having the ability to] set some configuration in AppArmor or SElinux to allow the container nearly all the same access to the host as processes running outside containers on the host." In other words, you should rarely, if ever, use this switch on your container command line. It is simply the least secure and laziest approach, widely abused when developers cannot get features to work. Taking such an approach might mean that a volume can only be mounted from a container with tightened permissions onto a host's directory, which takes more effort to achieve a more secure outcome. Rest assured, with some effort, whichever way you approach the problem there will be a possible solution using specific kernel capabilities, potentially coupled with other mechanisms, which means that you don't have to open the floodgates and use Privileged mode.

For our example, we will choose two of the most powerful kernel capabilities to demonstrate what a container looks like, from the inside out. They are `CAP_SYS_ADMIN` and `CAP_NET_ADMIN` (commonly abbreviated without `CAP_` in Docker and kernel parlance).

The first of these enables a container to run a number of sysadmin commands to control a system in ways a `root` user would. The second capability is similarly powerful but can manipulate the host's and container network stack. In the Linux manual page (`man7.org/linux/man-pages/man7/capabilities.7.html`) you can see the capabilities afforded to these `--cap-add` settings within Docker.

From that web page we can see that Network Admin (`CAP_NET_ADMIN`) includes the following:

- Interface configuration
- Administration of IP firewall
- Modifying routing tables
- Binding to any address for proxying
- Switching on promiscuous mode
- Enabling multicasting

We will start our look at a container's internal components by running this command:

```
$ docker run -d --rm --name apache -p443:443 httpd:latest
```

We can now check that TCP port 443 is available from our Apache container (Apache is also known as `httpd`) and that the default port, TCP port 80, has been exposed as so:

```
$ docker ps
IMAGE  COMMAND              CREATED         STATUS PORTS                NAMES
httpd  "httpd-foreground" 36 seconds ago Up 33s 80/tcp, 443->443/tcp apache
```

Having seen the slightly redacted output from that command, we will now use a second container (running Debian Linux) to look inside our first container with the following command, which elevates permissions available to the container using the two kernel capabilities that we just looked at:

```
$ docker run --rm -it --name debian --pid=container:apache \
--net=container:apache --cap-add sys_admin debian:latest
```

We will come back to the contents of that command, which started a Debian container in a moment. Now that we're running a Bash shell inside our Debian container, let's see what processes the container is running, by installing the `procps` package:

```
root@0237e1ebcc85: /# apt update; apt install procps -y
root@0237e1ebcc85: /# ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 15:17 ?        00:00:00 httpd -DFOREGROUND
daemon       9     1  0 15:17 ?        00:00:00 httpd -DFOREGROUND
daemon      10     1  0 15:17 ?        00:00:00 httpd -DFOREGROUND
daemon      11     1  0 15:17 ?        00:00:00 httpd -DFOREGROUND
root        93     0  0 15:45 pts/0    00:00:00 bash
root       670    93  0 15:51 pts/0    00:00:00 ps -ef
```

We can see from the `ps` command's output that `bash` and `ps -ef` processes are present, but additionally several Apache web server processes are also shown as `httpd`. Why can we see them when they should be hidden? They are visible thanks to the following switch on the `run` command for the Debian container:

```
--pid=container:apache
```

In other words, we have full access to the `apache` container's process table from inside the Debian container.

Now try the following commands to see if we have access to the filesystem of the `apache` container:

```
root@0237e1ebcc85: cd /proc/1/root
root@0237e1ebcc85: ls
bin  boot  dev etc  home  lib lib64  media  mnt  opt  proc  root  run  sbin
srv  sys  tmp  usr  var
```

There is nothing too unusual from that directory listing. However, you might be surprised to read that what we can see is actually the top level of the Apache container filesystem and not the Debian container's. Proof of this can be found by using this path in the following `ls` command:

```
root@0237e1ebcc85: ls usr/local/apache2/htdocs
usr/local/apache2/htdocs/index.html
```

As suspected, there's an HTML file sitting within the `apache2` directory:

```
root@0237e1ebcc85:/proc/1/root# cat usr/local/apache2/htdocs/index.html
<html><body><h1>It works!</h1></body></html>
```

We have proven that we have visibility of the Apache container's process table and its filesystem. Next, we will see what access this switch offers us:`--net=container:apache`.

Still inside the Debian container we will run this command:

```
root@0237e1ebcc85:/proc/1/root# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
10: eth0@if11: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
       valid_lft forever preferred_lft forever
```

The slightly abbreviated output from the `ip a` command offers us two network interfaces, `lo` for the loopback interface and `eth0`, which has the IP address 172.17.0.2/16.

Let's exit the Debian container by pressing Ctrl+D and return to our normal system prompt to run a quick test. We named the container `apache`, so using the following `inspect` command we can view the end of the output to get the IP address for the Apache container:

```
$ docker inspect apache | tail -20
```

Listing 1.2 shows slightly abbreviated output from that command, and lo and behold in the `IP Address` section we can see the same IP address we saw from within the Debian container a moment ago, as shown in Listing 1.2: `"IPAddress": "172.17.0.2".`

Listing 1.2: The External View of the Apache Container's Network Stack

```
"Networks": {
                "bridge": {
                    "IPAMConfig": null,
                    "Links": null,
                    "Aliases": null,
                    "NetworkID": [...snip...]
                    "Gateway": "172.17.0.1",
                    "IPAddress": "172.17.0.2",
                    "IPPrefixLen": 16,
                    "IPv6Gateway": "",
```

```
                    "GlobalIPv6Address": "",
                    "GlobalIPv6PrefixLen": 0,
                    "MacAddress": "02:42:ac:11:00:02",
                    "DriverOpts": null
                }
            }
        }
    }
]
```

Head back into the Debian container now with the same command as earlier, shown here:

```
$ docker run --rm -it --name debian --pid=container:apache \
--net=container:apache --cap-add sys_admin debian:latest
```

To prove that the networking is fully passed across to the Debian container from the Apache container, we will install the `curl` command inside the container:

```
root@0237e1ebcc85:/# apt update; apt install curl -y
```

After a little patience (if you've stopped the Debian container, you'll need to run `apt update` before the `curl` command for it to work; otherwise, you can ignore it) we can now check what the intertwined network stack means from an internal container perspective with this command:

```
root@0237e1ebcc85:/# curl -v http://localhost:80
<html><body><h1>It works!</h1></body></html>
```

And, not straight from the filesystem this time but served over the network using TCP port 80, we see the HTML file saying, "It works!"

As we have been able to demonstrate, a Linux system does not need much encouragement to offer visibility between containers and across all the major components of a container. These examples should offer an insight into how containers reside on a host and how easy it is to potentially open security holes between containerized workloads.

Again, because containers are definitely not the same as virtual machines, security differs greatly and needs to be paid close attention to. If a container is run with excessive privileges or punches holes through the security protection offered by kernel capabilities, then not only are other containers at serious risk but the host machine itself is too. A sample of the key concerns of a "container

escape" where it is possible to "break out" of a host's relatively standard security controls includes the following:

- Disrupting services on any or all containers on the host, causing outages
- Attacking the underlying host to cause a denial of service by causing a stress event with a view to exhaust available resources, whether that be RAM, CPU, disk space capacity, or I/O, for example
- Deleting data on any locally mounting volumes directly on the host machine or wiping critical host directories that cause system failure
- Embedding processes on a host that may act as a form of advanced persistent threat (APT), which could lie dormant for a period of time before being taken advantage of at a later date

Other Containers

A little-known fact is that serverless technologies also embrace containerization, or more accurately lightweight virtualization when it comes to AWS Lambda. Making use of KVM as mentioned earlier, AWS uses Firecracker to provide what it calls MicroVMs. When launched, AWS explicitly stated that security was its top priority and ensured that multiple levels of isolation were introduced to provide defense in depth. From a performance perspective, remarkably the MicroVMs can apparently start up in about an eighth of a second. An active Open Source project, Firecracker is an intriguing technology:

```
github.com/firecracker-microvm/firecracker
```

As mentioned earlier, the security model is a familiar one, according to the AWS site: "The Firecracker process is jailed using cgroups and seccomp BPF, and has access to a small, tightly controlled list of system calls."

Apparently, at least according to this page on the AWS forums (`forums.aws.amazon.com/thread.jspa?threadID=263968`), there are restrictions applied to the containerized service such as limitations on varying kernel capabilities. These are dropped for security purposes and might include various syscalls like PTRACE, which allow the monitoring of and potentially the control of other processes. Other more obvious services, such as SMTP, are disallowed to prevent spam from leaving a function. And removing the ability to use the `CAP_NET_RAW` capability makes it impossible to spoof IP addresses or use raw sockets for capturing traffic.

Another approach to running containers in a more secure fashion is to lean on hardware virtualization to a greater degree. One of the earlier pioneers of containerization was CoreOS (known for a number of other products, such as `etcd`, which is prevalent in most modern Kubernetes distributions). They created a container runtime called `rkt` (which was pronounced "rock-it"), that is sadly now deprecated. The approach from `rkt` was to make use of KVM as a hypervisor. The premise (explained at `coreos.com/rkt/docs/latest/running-kvm-stage1.html`) was to use KVM, which provides efficient hardware-level virtualization, to spawn containers rather than `systemd-nspawn` (`wiki.debian.org/nspawn`), which can create a slim namespaced container. The sophisticated `rkt` offered what might be called *hard tenancy* between containers. This strict isolation enabled true protection for Customer B if Customer A was compromised; and although containers are, again, not virtual machines, `rkt` bridged a gap where previously few other security innovations had succeeded.

A modern approach being actively developed, similar to that of `rkt`, is called Kata Containers (`katacontainers.io`) via the Open Stack Foundation (OSF). The marketing strapline on the website confidently declares that you can achieve the "speed of containers" and still have the "security of VMs." Along a similar vein to `rkt`, MicroVMs are offered via an Open Source runtime. By using hardware virtualization the isolation of containerized workloads can be comfortably assured. This post from Red Hat about SElinux alerations for Kara Containers is informative: `www.redhat.com/sysadmin/selinux-kata-containers`. Its customers apparently include internet giants such as Baidu, which uses Kata Containers in production, and you are encouraged to investigate their offering further.

Finally, following a slight tangent, another interesting addition to this space is courtesy of AWS, which, in 2020, announced the general availability of an Open Source Linux distribution called Bottlerocket (`aws.amazon.com/bottlerocket`). This operating system is designed specifically to run containers with improved security. The premise for the operational side of Bottlerocket is that creating a distribution that contains only the minimal files required for running containers reduces the attack surface significantly. Coupled with SElinux, to increase isolation between containers and the underlying host, the usual suspects are present too: cgroups, namespaces, and seccomp. There is also device mapper functionality from `dm-verity` that provides integrity checking of block devices to prevent the chances of advanced persistent threats taking hold. While time will tell if Bottlerocket proves to be popular, it is an interesting development that should be watched.

Summary

In this chapter, we looked at some of the key concepts around container security and how the Linux kernel developers have added a number of features over the

years to help protect containerized workloads, along with contributions from commercial entities such as Google.

We then looked at some hands-on examples of how a container is constructed and how containers are ultimately viewed from a system's perspective. Our approach made it easy to appreciate how any kind of privilege escalation can lead to unwanted results for other containers and critically important system resources on a host machine.

Additionally, we saw that the USER instruction should never be set to `root` within a container and how a simple Dockerfile can be constructed securely if permissions are set correctly for resources, using some forethought. Finally, we noted that other technologies such as serverless also use containerization for their needs.