Docker Security

An analysis of security threats and recommended practices for building a secure Docker infrastructure

Andreas Pfefferle

Thesis
for the degree of
Bachelor of Science

University of Augsburg
Department of Computer Science
Software Methodologies for Distributed Systems

August 2018

Software Methodologies for Distributed Systems

secomba

Docker Security

Supervisor:	**Prof. Dr. Bernhard L. Bauer**, Department of Computer Science, University of Augsburg, Germany
Advisor:	**Dipl.-Inform. Robert Freudenreich**, Secomba GmbH, Augsburg, Germany

Abstract

English The use of virtualization technologies has increased significantly in recent years. With the introduction of Docker, a container-based approach was established that competes fiercely with traditional virtual machines. Not only are containers significantly more lightweight and resource-efficient in comparison, but they also enable faster development cycles and modern DevOps approaches. However, the question arises whether these advantages go hand in hand with weaker security of the virtualized infrastructure. This thesis tries to address this problem. The analysis consists of three parts: (1) The internal functionalities designed to ensure the security of Docker, (2) the threats to be observed when using Docker, and (3) which measures complicate or even prevent the threats identified in the last step. The analysis of threats covers several layers within a Docker infrastructure, starting with the hardware up to the integration of Docker in an automated deployment pipeline.

Deutsch Der Einsatz von Virtualisierungstechnologien hat in den letzten Jahren stark zugenommen. Mit der Einführung von Docker wurde auch ein containerbasierter Ansatz geschaffen, der traditionellen virtuellen Maschinen starke Konkurrenz macht. Nicht nur sind Container im Vergleich deutlich leichtgewichtiger und ressourceneffizienter, sie ermöglichen auch schneller Entwicklungszyklen und moderne DevOps Ansätze. Dennoch stellt sich die Frage, ob diese Vorteile einhergehen mit einer schlechteren Sicherheit der virtualisierten Infrastruktur. Diese Thesis versucht dieses Problem zu adressieren. Die Analyse umfasst drei Teile: (1) Die internen Funktionalitäten, die die Sicherheit von Docker gewährleisten sollen, (2) welche Gefährdungen beim Einsatz von Docker zu beachten sind, und (3) mit welchen Maßnahmen die im letzten Schritt aufgezeigten Gefährdungen erschwert oder gar vermieden werden. Die Analyse der Gefährdungen umfasst dabei mehrere Schichten innerhalb einer Docker Infrastruktur, beginnend bei der Hardware bis hin zur Integration von Docker in eine automatisierte Deployment-Pipeline.

Contents

1

Introduction

Over the last decade, cloud computing has become ubiquitous. End-user applications such as Dropbox or Google OneDrive have become established as well as platform-as-a-service applications such as Microsoft Azure or infrastructure-as-a-service products such as Amazon Web Services have become fundamental building blocks in the industry. In this context, multi-tenant services are increasingly being used in which several users operate simultaneously on one system, possibly even customers who are independent of one another. Cloud computing uses virtualization technology, whose roots go back to the 1970s [37, 78]. After initially concentrating on hypervisor-based virtualization, a different approach has prevailed with Docker since 2013. Docker containers promise a scalable and efficient improvement compared to hypervisor-based virtualization. Moreover, they are intended to contribute to accelerating development cycles, as they can be used to create reproducible behavior on development systems and in the production environment [67].

Docker was not the first container-based virtualization solution, *jails*, for example, was already available in FreeBSD 4.0 which was released in 2000 [62]. Besides, Linux containers (LXC) existed from 2008, although they were not widely used until Docker was introduced [34]. In fact, LXC was initially used to implement the Docker engine but was eventually replaced by a self-developed component in Docker. Docker is developed by *Docker Inc.* and has since established itself in the containerization market, as can be seen from a few key figures: Over 100,000 third-party projects use Docker, which has been co-created by over 3,000 community developers. Additionally, more than 900,000 applications are now available in the Docker Hub, a solution for the distribution of container images, and more than 29 billion images have been downloaded. The progress of image downloads also allows conclusions to be drawn about the distribution and acceptance of Docker over the years: by the end of 2014, 100 million downloads were announced, 1 billion images were downloaded by November 2014,

4 billion by June 2016 and 18 billion by January 2017 [21].

A high spread of a technology also increases the interest of attackers, as the expected impact grows with it. Furthermore, Docker now has a vibrant ecosystem that includes both its own products and services, but also third-party tools such as Kubernetes or Consul. For an attacker, the attack surface has been increased, making Docker an even more attractive target.

Therefore, it is necessary to examine the dangers of using the containerization technology Docker more in detail. In this thesis, Docker's internal security mechanisms are considered in addition to an analysis of the security threats. Finally, recommendations for increasing security in a Docker infrastructure are provided that are tailored to the analysis of security threats.

1.1 Problem Statement and Goal of the Thesis

In computer science, one goal has always been to achieve efficient utilization of infrastructure. The use of container technologies offers attractive features from the point of view of performance, efficiency and administrative aspects. Nevertheless, it is questionable whether Docker preferred a reduction in security in favor of performance gains. In particular, this question arises when considering that other containerization technologies such as *jails* from FreeBSD or *rkt* from CoreOS, whose focus was on container security, are far less popular than Docker [105].

In total, three questions are addressed in this thesis. First, it will be answered which mechanisms Docker uses for its security. Secondly, it will be investigated to what extent security is threatened despite Docker's built-in security features and how an adversary can attack a Docker system. Third, based on the insights from the previous step, how can a system be strengthened to ward off such attacks.

1.2 Scope of the Thesis

Docker was initially developed as a tendentially monolithic project for Linux containers. The system was gradually dismantled into individual parts. In 2016, Microsoft containers were introduced that can be controlled with the Docker client and the Docker daemon [16]. Linux containers on Mircosoft Windows Professional can also be used, which, however, requires the use of a hypervisor [103]. This thesis deals exclusively with the use of Linux containers on a Linux operating system. This scenario is justified, since almost all modern standard literature on containerization deals with Docker on Linux.

Furthermore, the focus of the threat analysis should not be on orchestrating containers across several different hosts, as is possible with tools such as Docker Swarm, Consul or Kubernetes. However, section 3.2.3 in particular briefly introduces the built-in security mechanisms of Docker Swarm, since it is now installed by default as part of the Docker engine during Docker setup. An analysis of the security threats of such an architecture could be part of a later investigation.

1.3 Structure of the Thesis

In chapter 2, background information on virtualization and the difference between hypervisor-based and container-based virtualization are discussed. Afterward, Docker and its components are introduced, and essential terms and concepts for the following chapters are presented. At the end of the chapter, section 2.3 describes essential security goals which are crucial for the evaluation of Docker in the later course of the thesis. Chapter 3 examines the first of the questions posed in section 1.1. On the one hand, the internal container security features through some Linux functionalities are examined, on the other hand, the security features in the growing Docker ecosystem are discussed. After defining a threat modeling strategy at the beginning of chapter 4, a bottom-up approach is used to deal with security threats and exploitation techniques on different layers. This answers the second research question. Finally, chapter 5 examines security recommendations to increase Docker's security based on the identified threats from chapter 4. Chapter 5, thus, deals with the third research question.

2

Background

This chapter provides background information on virtualization in general, an introduction to Docker, and, finally, an overview of security goals.

To be more precise, section 2.1 provides an overview of the historical development of virtualized systems and a classification of hardware virtualization. Besides, fundamental concepts in the context of virtualization and requirements for such systems are discussed. Afterward, the two most important approaches in this area with hypervisor-based and container-based virtualization will be examined in more detail. After a comparison of these techniques, Docker is classified into one of the two categories.

Section 2.2 introduces the virtualization technology Docker in more detail. First, the architecture of the so-called Docker engine will be presented in detail, and the development of this central component since the initial release in 2013 will be discussed. Afterward, the concepts image, container, and registry are introduced. Finally, some tools for managing multi-container applications will be addressed.

Section 2.3 defines security goals for container-based virtualization. For each of the goals, a short example of their use in the Docker context is given.

2.1 Virtualization

In computing, virtualization can be applied on various levels to simulate virtual instances of, e.g., computer hardware, networks or storage devices. Initially, in the 1960's and 1970's, virtualization was considered a technique of logically dividing mainframes to allow multiple programs to run simultaneously [37]. Today, virtualization can be roughly divided into three categories: First, server or hardware virtualization, where several virtual operating systems run on a single physical server as so-called virtual machines. Secondly, network virtualization reproduces physical networks to run software programs in virtual networks as if they were running in real physical networks. Third, desktop virtualization allows logical desktops to be separated from the physical machine [106, 59]. Since Docker can be classified into the category server virtualization, the following will provide an introduction to this subject in more detail.

A virtual machine can be regarded as an efficient and isolated duplicate of a physical machine. Consequently, a virtual machine can virtualize all hardware resources, including processors, memory, storage or peripheral devices [78]. Generally, an actual machine on which the virtualization is executed is called *host system*, and the virtual machine which runs in the context of a superordinate host system is named *guest system*. A virtual machine is made possible by a software called virtual machine monitor, which creates an abstraction layer between the execution layers [92]. The virtualization requirements of Gerald J. Popek and Robert P. Goldberg, dating from 1974, are still today a recognized instrument for evaluating virtualization solutions and provide guidelines for creating virtualized computer architectures [78]: The *equivalence* or *fidelity* property requires that a program executed by a virtual machine monitor behaves identically to an application executed directly on an equivalent machine. The *resource control* or *safety* requirement means that a virtual machine monitor must have complete control over the virtualized resources. Lastly, the *efficiency* or *performance* characteristic demands that a significant proportion of the machine commands must be executed without virtual machine monitor intervention.

Virtualization addresses a wide range of data center operation issues for organizations and enterprises. The following list provides an overview of the main challenges and possible solutions.

Resource Optimization Virtualizing hardware and assigning parts of it based on the actual needs of users and applications can make much more effec-

tive use of available computing power, storage space and network bandwidth. Instead of buying dedicated physical hardware for each application, resources can be shared via virtualization [75].

Consolidation Consolidating the workloads of several under-utilized servers to fewer machines, hardware, and environmental expenses as well as optimizations of administration of the server infrastructure can be perceived. Taking these facts into consideration, the cost of ownership is reduced significantly, since machines and space needed in data centers decrease [3].

Scalability and Adaptability to Workload Variations Virtual systems are usually hardware-independent, portable and reproducible, which enables scalability and redundancy. Besides, resources can be dynamically relocated or prioritized according to requirements [92, 75].

Security Virtual systems must provide secure, isolated sandboxes for the operation of untrusted applications. The isolation allows other guest systems to remain protected even in the event of a successful attack on another instance [3, 92].

Legacy Applications Legacy software might not run on modern hardware or modern operating systems. With virtualization, legacy applications can continue to run on an old guest operating systems on virtual machines. Such independence also enables an effortless migration between old and new hardware [3, 92].

Virtualization is not strictly limited to a fixed physical-virtual machine scenario, but also virtual nesting within a physical host system is possible and reasonable in practice. For example, in a data center, each physical host system can serve multiple customers by assigning a guest system to each customer. In turn, a customer can operate this guest system as a virtual host system that virtualizes each customer application in further guest systems. In practice, however, deep nesting is rather an exception due to performance losses caused by the virtualization process [92].

In general, a distinction can be drawn between two types of guest systems:

Hypervisor-based A hypervisor is responsible for creating and operating virtual machines. Several guest operating systems can share virtual hardware, so that, for example, it is possible to run Linux, macOS and Win-

dows instances on one physical computer at the same time [92].

Container-based Each guest instance (*container*) uses a shared kernel but can differ in userspace, such as different Linux distributions with the same kernel. [97].

From the point of view of programs running in guest systems, both types give the impression they are running as a stand-alone system directly on the hardware. The way in which both types abstract functions of a higher-level host system is different, which will be investigated in the next sections.

2.1.1 Hypervisor-based Virtualization

Hypervisors create and run virtual machines. The hypervisor represents an intermediate between the guest system and the host system so that one or more independent machines run virtually on physical hardware. The hypervisor's abstraction properties enable several different guest systems to be executed on one host system [101]. Furthermore, hypervisors guarantee isolation and mandatory access control [92]. Although many modern hypervisor techniques meet, besides the equivalence and security condition, Popek's and Goldberg's performance requirement, they have a reputation for working inefficiently compared to, e.g., container-based virtualization [33]. In section 2.1.3, this will be investigated in more detail.

Popek and Goldberg distinguish between two types of hypervisor [78]:

Type-1 hypervisors (also native or bare-metal hypervisors) behave like an operating system because it is running in the most privileged mode [101]. It runs directly on the hardware of the host and provides multiple copies of the actual hardware to the guest systems. Modern Type-1 hypervisors include Microsoft Hyper-V, Xen, Oracle VM Server for SPARC, and VMware ESX/ESXi. The architecture of this type of hypervisor is shown on the left in Fig. 2.1

A Type-2 hypervisor (also hosted hypervisors) does not operate directly on the hardware, but within a host operating system, which in turn directly accesses the hardware. Thus, it runs on top of a host operating system, and the guest runs as a process on the host [101]. Nevertheless, a guest system behaves identically to a non-virtual system. Usually, the additional software layer of the

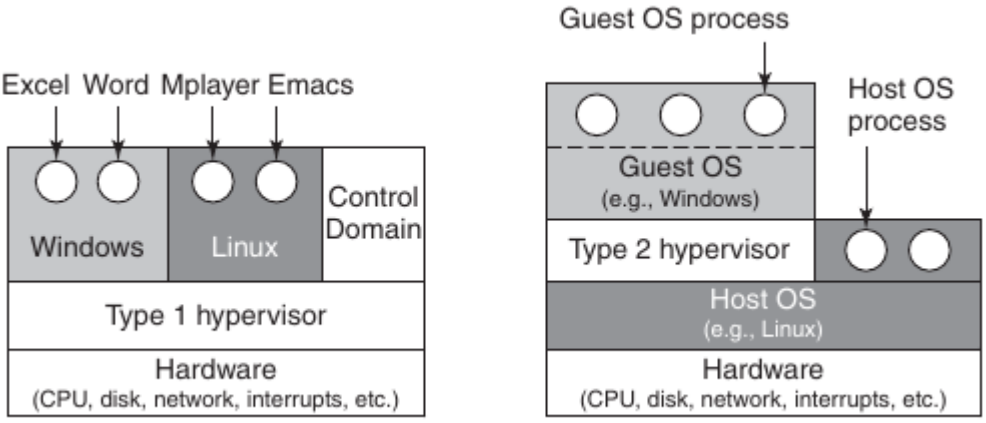

Figure 2.1: The hypervisor Type-1 on the left operates directly on the hardware and is located between hardware and guest systems. The hypervisor Type-2 on the right is a process within the host operating system and is located between host and guest systems [101].

Type-2 hypervisor leads to higher performance losses compared to Type-1, despite various optimization options, e.g., paravirtualization [101]. Amongst others, examples of this type of hypervisor are VirtualBox, VMware Workstation, and QEMU.

Some hypervisor-based virtualization solutions include properties of both categories. For example, Linux's Kernel-based Virtual Machine (KVM) transforms the host operating system to a Type-1 hypervisor, although Linux is still a general purpose operating system [4].

2.1.2 Container-based Virtualization

In container-based virtualization, an operating system of the container software provides certain functionalities for isolating processes from other processes and containers. Often, the Linux kernel is used for this, because it already offers some features for process isolation. But also FreeBSD with *jails* or Solaris with *Zones* have similar mechanisms available [62, 74] . Containers use the hardware of the host system. Thus, software running in containers communicates directly with the host kernel.

Containers are inspired by the Unix command `chroot`. An extended variant of `chroot` was used in FreeBSD to implement *jails*. This mechanism was further improved in the form of *Zones* in Solaris, an operating system for server virtualization developed by Oracle.

2.1.3 Comparison of Hypervisor-based and Container-based Virtualization

The slim design of containers allows almost native performance, as the hypervisor does not exist in this model. A study found that applications in containers have a performance drop of only 4% compared to native performance with the same hardware configuration [108]. In traditional virtualization, the hypervisor alone consumes about 10-20% of the host's computing power [34]. A benchmark test comparing the throughput (operations per second) of a VoltDB setup on hypervisor- and container-based systems concludes that containers have fivefold performance [93].

Containers also offer advantages in the lifecycle of virtual entities: While in traditional virtual machines, a restart takes seconds to minutes, the restart of containers only corresponds to a process restart on the host system that is completed in milliseconds [41]. In the case of VMs, the entire guest operating system must be restarted. Since container images do not require several operating system applications, such as an individual kernel, device drivers or the init system, but instead uses the host system's ones, container images are significantly smaller compared to traditional virtual machine images [108].

From the point of view of security, the lack of a hypervisor can be interpreted controversially: On the one hand, the host system's overhead shrinks because the entire operating system is not virtualized. The fewer host system functions are virtualized, the lower is the security risk that an attacker abuses a function. On the other hand, the concept of a shared kernel is generally less secure, since additional mechanisms must maintain the security objectives [97]. In addition, hypervisor-based virtual machines are commonly deployed to introduce another layer of security by entirely isolating resources exposed to an attacker from resources that need to be protected [41]. Moreover, operators of security-critical infrastructures must consider that one container, which usually runs a single process, can be replaced and updated faster than a traditional virtual machine that hosts multiple applications. Thus, a faster response to security

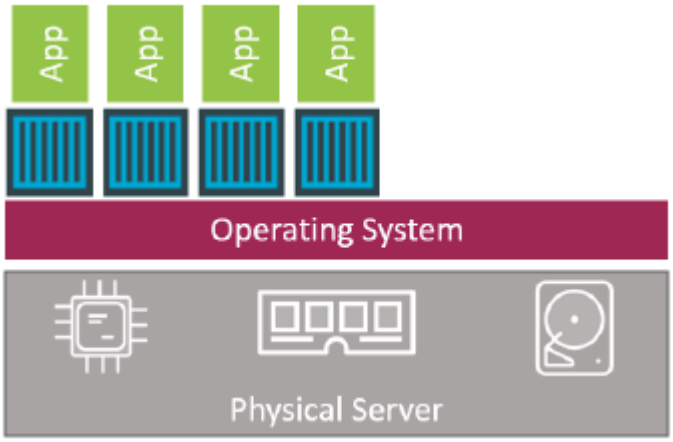

Figure 2.2: Several containerized applications on a host operating system [79].

vulnerabilities in programs can be achieved. For example, in the IT infrastructure of the Financial Times, used components are rebuilt every night so that it is possible to redeploy containers at any time [94].

2.1.4 Classification of Docker

In its initial release, Docker used Linux containers (LXC), which is a project that aims to build a toolkit to isolate processes from other processes [103]. Thus, Docker can be classified as a container-based virtualization technology. In March 2014, LXC was replaced by *libcontainer*.

As previously mentioned in 2.1.2, other container-based virtualization technologies existed before Docker. However, previous solutions were not as comfortable to use and to integrate into existing ecosystems. Therefore, they were not able to establish themselves in the industry.

2.2 Introduction to Docker

Docker was initially released in March 2013 by *Docker Inc.* under the open source license Apache 2 as a tool for operating containers as well as configuring and automating multi-container systems. It is predominantly implemented in the programming language *Golang*. Currently, over 3000 community developers contributed to the project. As indicated in 2.1.4, Docker aims to be lightweight and easy to use as well as it provides an efficient workflow for software developers and system administrators. To achieve these goals, Docker adds an application deployment engine on top of a virtualized container execution environment [103]. Until March 2014, the execution environment consisted of Linux containers (LXC). Afterward, the *Docker Inc.* in-house developed tool *libcontainer* replaced LXC as execution driver in Docker 0.9 because LXC was Linux-specific, which was a problem for a project that aimed to multi-platform [79]. Docker containers are used as combinable, exchangeable, flexible and portable modules of a system architecture.

The following sections will provide an overview of native components in the Docker ecosystem. First, Docker's internal architecture will be investigated, before exploring images, containers, and registries. Lastly, higher-level orchestration features like Docker Swarm or Compose will be discussed briefly.

2.2.1 Architecture

At a high-level, Docker consists of a Docker client and a Docker daemon, which interact according to a client-server model based on a RESTful API. Client and daemon may run on different physical machines and can communicate via HTTP. In the vast majority of scenarios, Docker's default command line interface client is used [103].

Originally, the Docker daemon was a monolithic binary, which contained, amongst others, the whole Docker client, the Docker API, the container runtime and more components. LXC provided the daemon with access to the fundamental components of containers in the Linux kernel. This monolithic approach impeded the development of the aspired Docker ecosystem. Thus, *Docker Inc.* began to modularize the daemon into smaller specialized and exchangeable tools. Today, the major components, which include the Docker client, the Docker daemon, *containerd*, and *runC*, are summarized as *Docker Engine* [79]. In Fig. 2.3,

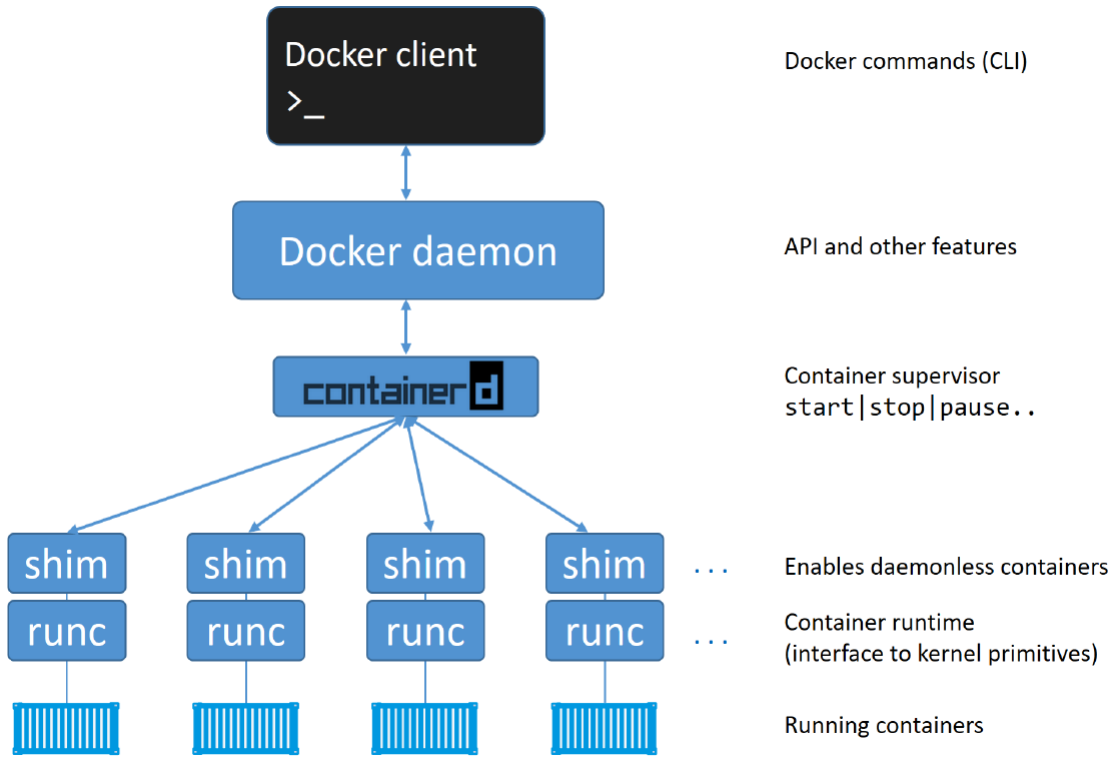

Figure 2.3: Current Docker Engine. Each component can be exchanged if new requirements have to fulfilled [79].

the current Docker Engine architecture can be seen.

While breaking the project apart, the Open Container Initiative (OCI) was founded to define container-related standards. Docker implements the OCI as strictly as possible. For example, the new Docker daemon does not contain any container runtime code. Instead, all container runtime code is implemented in a separate OCI-compliant layer. Together with more than 20 representatives from the industry, including Google, VMWare, and IBM, *Docker Inc.* was heavily involved in creating the *Open Container Format (OCF)* specification [73]. In April 2017, *Docker Inc.* founder Solomon Hykes announced Moby, a project designed to assemble Docker into multiple components [96]. *Docker Inc.* wants to move open source collaborations slowly to Moby and confirms the intention to split the Docker engine into further components. Tools and components that are currently used to assemble the Docker product but may be of interest to the open source community were planned to be released as open source projects. With Moby, *Docker Inc.* introduced a tool that is supposed to serve the compilation of

own container systems and thus promote the spread of containerization techniques.

Modern Docker versions use *runC* as default container-runtime-spec. *runC* is a reference implementation of the OCF and represents a small and lightweight CLI wrapper for *libcontainer*, which, as stated previously, replaced LXC in the early Docker architecture. *libcontainer* is completely written in *Golang* and can independently interact with the Linux kernel [95].

Furthermore, to break down the functionalities of the Docker daemon even further into components, the container execution logic was refactored into a tool called *containerd*. Its purpose is to manage container lifecycle operations like `start`, `stop` or `pause` [79]. *containerd* is lightweight and is also poised to become the most default container runtime wrapper of Kubernetes [67].

Between *containerd* and *runC*, a tool called *shim* enables daemonless containers, which decouples running containers from the daemon which, in turn, allows, e.g., daemon upgrades during container runtime. After a *runC* instance created a container, the process exits, and the corresponding *shim* process becomes the container's new parent. Thus, *shim* is responsible, for example, to keep any STDIN and STDOUT open as well as to report the container's exit status back to the daemon [79].

The daemon in modern Docker versions is responsible for managing images, image builds, the REST API, security, core networking, authentication and orchestration [79].

A typical `docker run container` command is transformed by the Docker client into appropriate API payload and is POSTed to the correct API endpoint. The daemon receives the command and instructs *containerd* over gRPC to create a container. In fact, *containerd* does not actually create containers. Instead, it delegates *runC* to do that by converting the Docker image into an OCF bundle, which can be interpreted by *runC*. Lastly, *runC* uses the operating system's kernel features via *libcontainer* to create a container which will be discussed in more detail in 3.

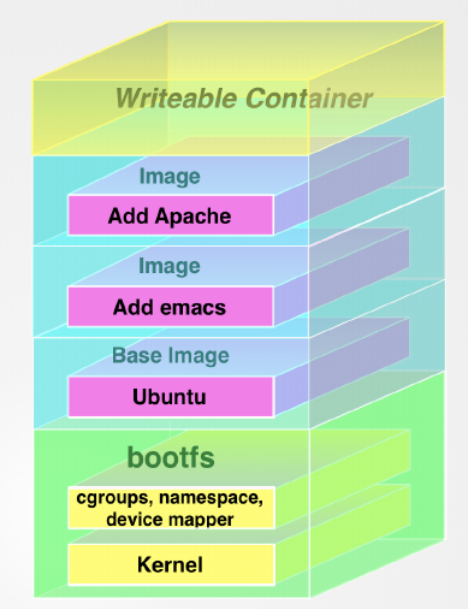

Figure 2.4: The Docker filesystem layers [103].

2.2.2 Images

Docker containers base on images, which can be seen as immutable files that are portable, shareable and easy to store and update. To be precise, images are composed of filesystems layered over each other, i.e., they base on the Copy-On-Write model [103].

The base filesystem *bootfs*, which resembles the typical Linux/Unix boot filesystem, is unmounted after a container has booted to free up the RAM [103]. The next layer on top of the boot filesystem is *rootfs* (also called base image), which is an operating system such as Ubuntu or Alpine. Other than in a traditional Linux boot, where the root filesystem is switched to read-write mode after the boot process and a successful integrity check, in Docker, however, the root filesystem stays in read-only mode [103]. Docker uses *union mount* to add more read-only filesystems onto the root filesystem. Union mounts allow multiple filesystems to be mounted simultaneously, which eventually seem to be a single filesystem. The resulting filesystem contains all filesystems and subdirectories of all underlying layers. In Docker, each of the filesystems is an image. At runtime, when a container is launched from an image, a read-write filesystem on top of any layers below is mounted, where the process running in the container operates [103]. Fig. 2.4 represents the layers of an image.

Images and layers are uniquely identifiable with hash values. An image also contains metadata which includes information on environment variables, port mappings, volumes and other details. Usually, the layers of an image contain the minimal execution environment for one application, including libraries, binaries, packages and the application's source code [67]. The layers enable to build modular images. For example, a developer could use an Ubuntu image as a base to build two further images: one image, that contains a web server application and another image that contains a database management system. Both newly created images base on the same Ubuntu image.

Images can be built with the instructions of a Dockerfile and the `docker build` command. Dockerfiles are simple text files named "Dockerfile" that contain a series of commands [79]. Each instruction is successively executed from top to bottom and produces a new layer in the image in each step. Instructions may copy code from the host machine to the image, define environment variables or define which ports should be opened in a container running the image. Eventually, an image consisting of several layers is generated. Dockerfiles enable a simple, automatic and reproducible way of creating executable images [67].

A simple docker file follows in the listing below:

```
1 FROM ubuntu:18.04
2 MAINTAINER Andreas Pfefferle "andreas.pfefferle@anpfeff.
     com"
3 RUN apt-get update; apt-get install -y nginx
4 RUN echo 'Hello world!' \
5     >/var/www/html/index.html
6 VOLUME /app
7 EXPOSE 80
8 ENTRYPOINT ["/usr/sbin/nginx", "-g", "daemon off;"]
```

In this example, the official Ubuntu image builds the base for the image. It is identified with its hash value. If the host machine does not provide an Ubuntu image, the Docker daemon will contact a registry. Registries will be explored in section 2.2.4. Both RUN statements add a new layer to the image at build time, when the associated command is executed. The VOLUME statement adds a special directory that bypasses the union filesystem [103]. These so-called volumes can be shared and reused between containers and persist even if no containers use them [103]. The EXPOSE instruction opens a port at container runtime. Finally, the ENTRYPOINT command specifies which process to execute when launching a container.

2.2.3 Containers

Section 2.2.2 indicated that images form the basis for containers. Containers are running instances of an image. This can be compared to hypervisor-based virtualization: a traditional virtual machine starts from a virtual machine template, e.g., a VirtualBox VM may boot a Debian template and, thus, virtualizes an entire operating system. In Docker, in contrast, containers share the kernel with their host operating system. In order to create containers, Docker claims resources from the host operating system such as the filesystem, the process tree, and the network stack [79]. This process will be investigated in detail in section 3.1.

Similar to traditional virtual machines, applications running in containers do not detect whether they are running in a virtualized environment [79]. However, unlike traditional virtual machines, Docker was designed to perform exactly one process per container. The container exits automatically when the process finishes [67]. Moreover, Docker containers do not have, in contrast to traditional virtual machines, an init process. Usually, the init process, e.g., *systemd*, on a Linux VM (or Linux OS) runs with a process ID of 1 and a parent process ID of 0 and is responsible for managing all other processes on that operating system. [67]. Although Docker containers are not designed to run multiple processes, tools like *Supervisord* can manage multiple processes per container. For instance, running a *cron* job, that occasionally handles log files of the main application, might be useful. However, *Docker Inc.* recommends running only one application or process per container, since service-oriented and microservices architectures are encouraged [103].

Generally, Docker containers start within milliseconds. Compared to traditional virtual machines, which have to supervise their kernel's locating, decompressing, hardware enumerating and initializing, containers do not have to manage a kernel, since Docker utilizes the host's already running kernel [79].

As already discussed in section 2.2.2 and Fig. 2.4, when starting the container, Docker adds a layer to the image which an application inside the container can read and write to. Each underlying layer can only be accessed with reading permissions. A container's lifecycle can be controlled with several instructions in the Docker client. These instructions include create, start, start, pause, restart and remove. The stop command sends `SIGTERM` signal to the process with the ID 1 inside the container. After this signal, the process has 10 seconds to clean up its filesystem and gracefully shut down itself. If this limit is exceeded, the

process receives a `SIGKILL` [79].

2.2.4 Registry

Docker components are able to communicate with registries via a registry API. A registry is a web application that serves as a storage and distribution platform for images. Registries consist of repositories, which in turn contain images that may contain a name as well as a tag, for example, the version number [103].

When the Docker client receives the command to start an image in a container, the Docker daemon first checks for a local copy of the image. A hash value uniquely identifies the image. If there is no local copy, the daemon contacts a registry. If this registry contains a matching image, the daemon will download the image to the local machine. However, it is not only possible to obtain images from a registry, but *Docker Inc.* also encourages to share locally created images with other persons and organizations via a registry. In fact, when images are created, the Docker engine does not only generate so-called content hashes across the layers. Besides, an extra distribution hash is created over the compressed version of each layer, since each layer is compressed before pushing the image into the registry to save both bandwidth and storage space in the registry. The distribution hash is then used to verify that the image has not been changed on the way to or from the registry [79].

Docker Inc. provides a publicly accessible registry with *Docker Hub*, which is also Docker's default registry. Currently, over 29 billion images have been downloaded from the Docker Hub, and over 900,000 applications with Docker have been implemented (as of July 2018) [21]. Individuals and organizations can create accounts and manage images in their repositories. In addition to the ability to upload locally built images, Docker Hub allows inserting so-called automated builds by connecting a Github or Bitbucket repository that contains a Dockerfile to Docker Hub. When a contributor adds new changes to the git repository, an image build is automatically triggered, adding a new image to Docker Hub [20]. As can be read later in chapter 4, manipulated public images are a significant security problem.

2.2.5 Docker Compose, Docker Swarm and other Tools for managing multi-container applications

In order to manage applications consisting of multiple containers, a tool named *Docker Compose* can be used. In 2014, *Docker Inc.* aquired the *Orchard* team who developed the formerly named *Fig* orchestration tool written in *Python*.

Compose uses a *YAML* file to configure an application's services. A service is a container which interacts with other containers, and that has specific runtime properties [103]. An exemplary *Docker Compose* file is listed below.

```
1  version: '3'
2  services:
3      node:
4          build:
5              context: ../../
6              dockerfile: ./packages/website/Dockerfile
7          command: [ "node",  "/index.js" ]
8      nginx:
9          build:
10             context: ./
11             dockerfile: ./nginx/Dockerfile
12         depends_on:
13             - server
14         ports:
15             - "5432:80"
16     redis:
17         image: redis:alpine
```

The *Docker Compose* file defines three services: First, a *Node.js* application should operate as a server backend. Line 4 to 6 define which Dockerfile is the basis for the resulting *Node.js* image. Secondly, an *Nginx* service should manage the interaction with clients. For this reason, a port mapping from the host's port 5432 to the container's port 80 is initiated. Lastly, a *Redis* service is defined. In this case, no modifications on the public available *Redis* image are necessary. Therefore, no build step is needed and *Docker Compose* can use the default image. When the containers start, an associated network for the three services will be created. Within this network, the services can communicate with each other.

Docker Swarm is a tool for native clustering. It is, like *Docker Compose*, open

source and licensed with the Apache 2.0 license. *Docker Inc.* developed *Swarm* in *Golang* and integrated a *Swarm*-mode in the Docker Engine in Docker 1.12. *Docker Swarm* turns multiple Docker hosts into a single virtual Docker host [103]. It serves the standard Docker API on top of that cluster moves up the abstraction of Docker containers to the cluster level. Such a cluster consists of worker and manager nodes. Managers dispatch and organize the work on the swarm and workers run the tasks dispatched from the manager nodes. Furthermore, *Docker Swarm* also manages load balancing and DNS [32].

Similar to *Docker Swarm* is Google's container orchestration tool *Kubernetes*. It is open source and emerged as the leading orchestrator of containerized apps [79]. Docker is currently the default container runtime of *Kubernetes*. However, *Kubernetes* is built to be pluggable. Therefore, Docker can be easily substituted.

Other container orchestration tools are, e.g., Hashicorp Consul, Centurion, Fleet, Helios or Apache Mesos.

2.3 Security Goals

Depending on the requirements and application area, several security goals exist for a computer-based system or network, which are to be ensured by a wide variety of actions. Several security goals can also be mentioned in the context of Docker and virtualization.

In chapter 4, the STRIDE method will be used to identify potential threats to a Docker infrastructure. STRIDE is an acronym for Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, and Elevation of Privilege [90]. Each of these concepts describes an approach of attacking a system in a specific way and attempts to violate one of the security goals defined in this section. The following subsections are structured in the order of the STRIDE concepts, for example, Authenticity in 2.3.1 is violated by Spoofing, Integrity in 2.3.2 is violated by Tampering, and so on. The STRIDE threats will be examined in section 4.1 in more detail.

2.3.1 Authenticity

Authenticity ensures that data is accurate, unaltered and verifiable. If the communication between two participants is authentic, it can be assumed that an exchanged message originates from the other communication partner and that an unauthorized party did not manipulate it. Technologies for authentication include SSH host keys, TLS with certificates, and digital signatures [90].

In the Docker ecosystem, organizations and individuals exchange images among each other via registries. One of the largest registries is *Docker Hub*, which is public and accessible to everyone. One of the challenges is that both in private and public registries no unauthorized adversary can change images neither in transit nor the registries themselves. In section 3.2.2, the concept of Docker Content Trust will be introduced which intends to ensure the authenticity of an image.

2.3.2 Integrity

Integrity ensures the correctness and consistency of data and its sources, i.e., unauthorized users cannot change data without the owner's permission [101]. Such impermissible modifications also include deleting data or adding incorrect data. Ensuring integrity can be addressed by relying on the system defenses (e.g., Unix file permissions or `.htaccess` files on a web server), using cryptographic methods (e.g., hashes or signatures), and logging and auditing activities on the system [90].

Docker is intended to ensure that no data from other containers and the host system can be modified from a malicious container [79]. For example, a hostile container should not be able to modify files from another container, since such an action could change the behavior of the running process in the attacked container.

2.3.3 Non-Repudiation

If a system fulfills the non-repudiation property, a sender of a message cannot deny being the sender. This property is closely related to the authenticity property. The difference can be illustrated in an example. If Alice wants to send a secure message to her communication partner Bob, both can generate a key pair and Alice can create a digital signature of the message with her private keys. When Bob receives the message and the digital signature, he can prove to a third person that only Alice could have sent this message because only she knows her private key. Thus, Alice cannot deny that she wrote the message. By using a symmetrical key which is known to both, Alice could create a MAC from the message and send it to Bob. However, Bob cannot go to a third party and claim that the message originates from Alice, because he could have eventually even created the MAC himself with the symmetric key.

Sometimes repudiation can be a feature, such as in the Signal protocol created by Open Whisper Systems [104]. In the Docker ecosystem, digital signatures are used to sign images if Docker Content Trust is enabled. This will be discussed in section 3.2.2.

2.3.4 Confidentiality

Confidentiality describes that information is not disclosed to unauthorized parties [6]. Moreover, owners of data should also be able to determine which parties can see it. The system should help to define access to data in the smallest possible units, preferably for each individual file [101]. The main threat of the confidentiality property is when data is exposed. To assure confidentiality, mainly access control mechanisms and encryption is used [90].

In the context of Docker, the goal of confidentiality is endangered if an attacker obtains private information. For example, an attacker that gained access to a container should not be able to break out of it and should not be able to read any information from either another container or the host machine. An adversary does not necessarily have to have access to a container through an unauthorized process, but may merely run concurrently on the same host from a cloud service provider. Likewise, it should not be possible for such an attacker to conclude private information from other containers or the host via side-channels such as CPU usage.

2.3.5 Availability

The availability property implies that information or resources can be accessed as desired [6], which indicates attackers may not hold benign stakeholders off from using the application. *Denial of Service* attacks are defined as attacks against availability. For instance, by overloading a network with traffic, the availability of services within the network are affected. If multiple containers run on a Docker host, a single container with too much computing or storage requirements should not cause applications in other containers to become unusable.

2.3.6 Authorization

Authorization includes assigning and checking access permissions to resources such as files or directories. An authorization check usually follows successful authentication. Authorization technologies include, among others, access control lists or role-based access control, such as DAC, which will be investigated in section 3.1.2 [90].

Docker refines DAC used in Unix with Linux/POSIX capabilities. On specific host operating systems there is also the possibility of Mandatory Access Control (MAC) through SELinux or AppArmor, which is discussed in chapter 5.

3

Built-in Security Features of Docker

The security of virtualization products like Docker is essential for acceptance in the industry. The goals set out in section 2.3 must, therefore, be met as a matter of urgency. *Docker Inc.*, which is financed by an enterprise subscription model that includes further features in dealing with the Docker ecosystem in addition to support and certification, also sees security as a high priority, especially since *CoreOS* announced *rkt* 2016 as a competitor product. *rkt* was developed as a container technology with a strong focus on security to fix many of the problems contained in Docker's container model [105]. The major security update for Docker version 1.10 was released on February 4, 2016, a few hours after CoreOS announced version 1.0 of *rkt*, which suggests strong competition between the two vendors [40, 39].

This chapter first examines the Linux features Docker uses to ensure confidentiality, integrity, and availability. Section 3.2 then discusses the features Docker uses to ensure security in its ecosystem.

3.1 Security through Linux Features

Docker's security is pursued according to the well-known philosophies *Principle of least Privilege* and *Defense in Depth* [70, 19].

Principle of Least Privilege Virtualized resources are only allowed to access the information and resources required for their legitimate purpose [85]. Docker is therefore only authorized to perform operations that serve its purpose as container technology. All other privileges are withheld from Docker. Docker 1.10 eliminated the need of running containers as root. However, as will be seen later in the chapter, this requires to set up user namespaces and also limits some Docker features.

Defense in Depth The combination of several independent security measures increases the overall security of a system [8]. The individual mechanisms can - as is also the case in Docker's security architecture - overlap with mechanisms of other layers. Even if an action can be overcome by an attacker, by overlapping other security features, security is still guaranteed. Docker uses many Linux technologies, such as namespaces, control groups, or capabilities, to form a deep defense system.

Docker introduced native Windows containers in 2016 [36]. As explained in section 1.2, this thesis covers only Linux containers. Therefore, security mechanisms of Windows containers are not examined here. This section introduces the security mechanisms made possible by the Linux operating system and its kernel. This includes the isolation of critical areas based on the `chroot` command in the form of namespaces, which will be investigated in depth in section 3.1.1. Afterward, in section 3.1.2, access controls through capabilities are explained. Finally, in section 3.1.3, resource management with control groups will be introduced.

3.1.1 Linux Namespaces for Container Isolation

Under Linux, container isolation is implemented for most critical areas of a system using namespaces. Since 2002, namespaces are part of the Linux kernel. Docker containers are organized collections of namespaces [79].

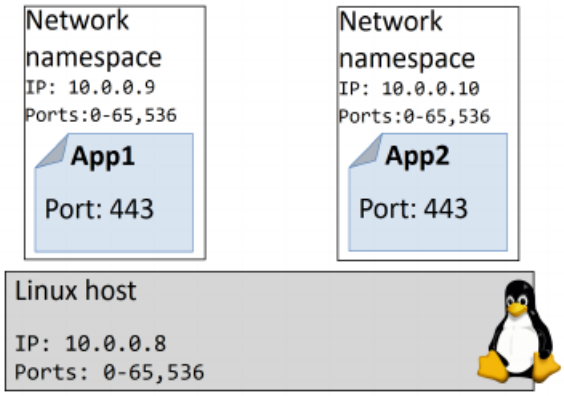

Figure 3.1: A high-level example of two web server applications running on a single host and both using port 443. Each web server app is running inside of its own network namespace. [103].

Since applications within containers must be offered a complete runtime environment, all relevant areas of the host system must be covered by namespaces. It also allows multiple applications to run on the same operating system without conflicts of shared configuration files and shared libraries [79]. For example, if a new process is to be started under Linux, the kernel is told via system calls to provide a new namespace for this. Another example would be network namespaces, which allow running several web servers on port 443. Each of these web servers would then run in its own network namespace, as shown in Fig. 3.1.

Processes within a namespace appear to have their own isolated instance of the global resource, even though the namespace abstracted a global resource [58]. If a global resource is changed, other processes that are members of the namespace notice the modification, which is, however, invisible to processes outside the namespace [100]. Initially, the mount namespace was introduced in the Linux kernel 2.4.19. More namespaces were added in 2006 [1]. In 2016, the latest namespace for cgroup was adjoint in the Linux kernel 4.6 [51]. Currently, other namespaces are discussed and proposed, such as a syslog namespace, which is supposed to help to separate logs from different containers on a host [84].

Since namespaces are used to partition kernel resources, they can be seen as a solution to the security goal of confidentiality (see section 2.3.4). Namespaces conduce fundamentally to isolate containers under Linux. In the following, all namespaces will be illustrated, and their usage in Docker will be shown.

Process isolation through the PID namespace With the PID namespace, it is possible to provide processes with an independent set of process IDs (PIDs) from other namespaces [58]. Since PID namespaces are nested trees, a new process receives a PID for each namespace from its current to the original PID namespace [58]. A parent process is able to kill a child process at any time with the `kill` command or monitor it with the `ps` command. The child process has no knowledge of the overlying process hierarchy and other branches in the process tree [65]. In the Docker context, each container on the host system corresponds to a process. Multiple containers are then isolated using PID namespace, which prevents the containers as child processes to see or access the host that is a parent or other processes that are siblings in the process tree. In turn, the host system has full control over the running containers thanks to its superordinate position in the process tree. Every container is assigned its own PID 1 because each container gets its own process tree branch [79]. Stopping PID 1, every process in its namespace and its descendants will immediately terminate [65]. Thus, the host system can terminate each container, and each of the processes started in it at any time with a single command.

Network Isolation through the network namespace With network namespaces, the network stack can be virtualized. A network namespace provides a private set of IP addresses, a connection tracking table, a custom routing table, socket listing, firewall and other network resources [58]. Each network namespace is assigned its own `/proc/net` directory containing network specific information. Docker provides each container with its own independent network stack, which is implemented by network namespaces. Containers can only communicate with other containers and the hosts via interfaces provided for this purpose [79, 27]. By default, a virtual ethernet bridge named docker0 is used for such communication. Newly started containers are added to this bridge by connecting their network interface eth0 to the bridge. As already mentioned in section 2.2.5, *Docker Compose* uses the capabilities of namespaces to additionally create a network for all started services in which they can easily communicate with each other.

File system isolation through the mount namespace The hierarchical filesystem can be divided using the mount namespace so that processes in different mount namespaces have different views of the filesystem hierarchy [64]. The system calls `mount` and `umount` no longer work on a global set of mount points visible to all processes on the system, but perform op-

erations that only affect the mount namespace associated with the calling process [64]. Compared to `chroot` system call, mount namespaces are considered to be more flexible and secure [64]. For example, it is possible to automatically propagate mount events to other namespaces using nested mount namespaces, which is not possible with `chroot`. This means that an external hard disk mounted in one namespace can also be used in child namespaces. Since it contains essential information for each process, the virtual `/proc` directory of the host system is not affected by mount namespaces [58]. In Docker, mount namespaces ensure that each container has a different view of the host system's directory structure. Processes inside of a container cannot access the mount namespace of the Linux host or other containers - they can only see and access their own isolated mount namespace [79]. Mount namespaces allow applications in containers to write on the filesystem at runtime. The uppermost writable layer (see section 2.2.2) of a container is stored in a directory specially assigned to the container.

Isolation of inter-process communication through the IPC namespace System V IPC objects and POSIX message queues are inter-process communication resources which are identified by mechanisms other than filesystem pathnames [64]. While PID namespaces can limit control over processes in the process hierarchy, IPC namespaces can be used to restrict communication between processes. Objects generated in one IPC namespace can be seen by all other processes that are members of the same namespace but are invisible to processes in outside the IPC namespace [58]. In Docker, one IPC namespace is assigned per container by default. As a result, communication via, for example, IPC Unix sockets, between containers or between containers and docker-independent processes of the host system is impossible [79].

User isolation through the user namespace Isolating the user and group ID number spaces, user namespaces allow processes to have a normal unprivileged user ID outside a user namespace, while simultaneously having a user ID of 0 within the namespace [64]. Thus, a process has full root privileges for actions within the user namespace but is unprivileged for operations beyond the namespace [64]. Creating user namespaces for even unprivileged processes is possible since Linux 3.8, which enables an unprivileged process to access functionality within the user namespace which was previously restricted to root users. A mapping table converts user IDs from the namespace's point of view to the system's point of view [58]. User namespaces were first introduced by Docker with version 1.10.

Usually, containers are started with root privileges so that an attacker who could break out of a container using a security vulnerability could act directly as root on the host system. Such a scenario is one of the worst imaginable security risks. User namespaces can be used to ensure that a root user within the container is mapped to a non-root user of the host system [79]. Currently, user namespaces are optional in Docker must be enabled manually. Even in the latest Docker version 18.03.1-ce user namespaces are not used automatically, but require some steps to use the feature [23]. With user namespaces and the resulting resolution of the containers at a user level, it is possible to limit and charge for the use of services per user [77].

UTS isolation through the UTS namespace "UTS" derives from the name of the structure passed to the `uname` system call, namely `struct utsname`, whose name in turn comes from "UNIX Time-sharing System" [64]. The `uname` system call returns the two system identifiers nodename and domainname. UTS namespaces isolate these system identifiers by using the `sethostname` and `setdomainname` system calls. The UTS namespace capability enables containers to have their own NIS host and domain name. Although UTS namespaces do not have a direct effect on security, this feature can be useful for configuration scripts that customize their operations based on these names [64]. In the listing in section 2.2.5, the services receive hostnames via UTS namespaces according to their descriptor in the *YAML* file. In this case, the hostnames "node", "nginx" and "redis" were assigned at runtime.

Control Group Isolation through the cgroup namespace Control group namespaces are currently not used by Docker [23]. The use of such cgoup namespaces would further advance the isolation of containers, since processes within the containers can currently see the global cgroup landscape [44]. For example, if a task consults the `/proc/self/cgroup` file, it currently sees the full cgroup path from the global cgroup hierarchy, causing information about the host system to be leaked. This information complicates cross-system container migration, as all names must be unique across systems so that no collisions with names occur on the new system. Control groups will be investigated in section 3.1.3.

Security vulnerabilities in the implementations of namespaces, especially the filesystem namespace, the process namespace and the network namespace, are to be regarded as highly critical [103].

3.1.2 Linux Capabilities to restrict Access Rights

Linux always runs a process under the privileges of the user who started the program. Thus, the rights of a user apply to all his actions. However, the simplistic scheme does not work in every case, for example, when a user wants to change the password. The password is usually defined in `/etc/passwd`, which a regular user has read-only access to, but cannot write to. If every user or every application could write to the file, the consequences would be far-reaching. The problem was addressed in Unix with the set-UID bit with which a user creates a process with the rights of the file owner, i.e., another user [45]. Thanks to the set-UID bit, the program `passwd` effectively has the privileges of the root user. Below is the set-UID bit on an Arch Linux system in the fourth position from the left. Note that the owner and the group are both "root".

```
-rwsr-xr-x 1 root root 55416 16.  Dez 2017 /usr/bin/passwd
```

`passwd` could theoretically use all of root's options, such as changing other people's passwords or switching off the firewall. As long as there are no programming errors, `passwd` is limited to changing the password of the calling user. Thus, it can be stated that in case of a bug of such a program extensive damage can be caused. To reduce the risk, Linux distributors have reduced the use of set-UID bits in recent years [63].

The mechanisms of Linux namespaces shown in section 3.1.1 attempt to prevent and, at best, make it impossible for attackers to break out of the virtualized environment of a container. Attackers should not be able to extend the rights initially intended for the container. In the best case, it cannot be determined from a guest's point of view whether they are in a virtual environment at all. However, the regular detection of new vulnerabilities demonstrates that programming bugs in applications and kernels cannot be excluded. Only in 2017, 453 vulnerabilities were classified as Common Vulnerabilities and Exposures (CVE), 56 of which allowed privileges to be obtained [18]. A fast correction of security flaws is only possible if they are known to the developers. If an attacker exploits such bugs and succeeds in breaking the isolation to the host or other containers on the machine, further access controls should ensure that no damage can be done to the host or other guest systems. The mechanisms presented in this section address this challenge.

Linux was developed as a clone of the Unix operating system [107]. Therefore, the central Unix security model in the form of Discretionary Access Con-

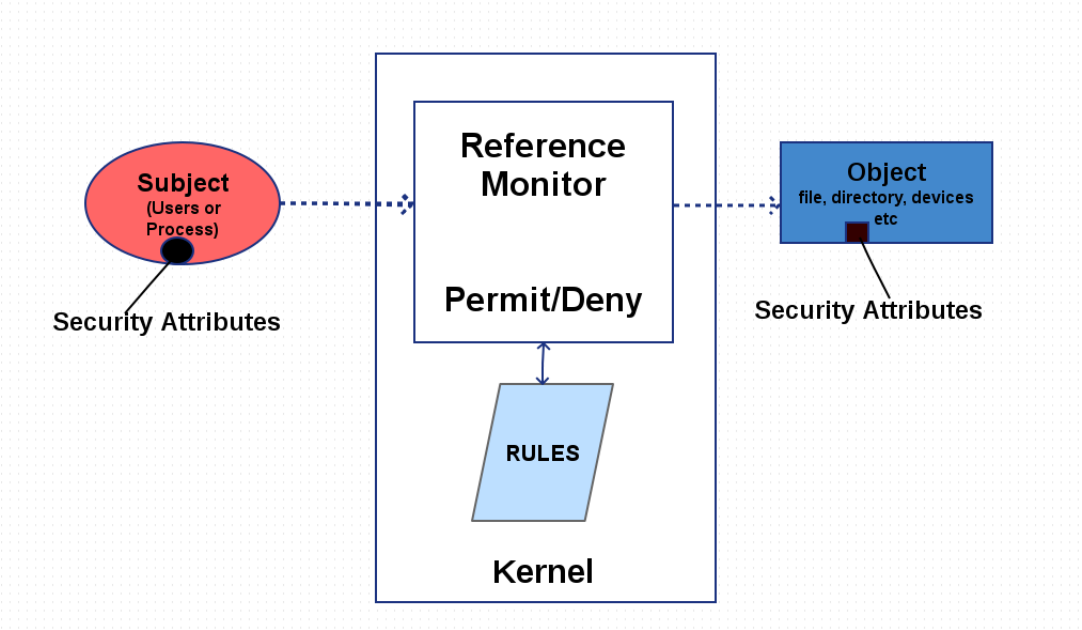

Figure 3.2: Access Control in Linux: The kernel determines whether a requesting user or process has access to a resource [42].

trol (DAC) [76]. While the security features of the Linux kernel have evolved considerably to meet modern requirements, Unix DAC remains the core model [45]. In Unix DAC, the owner of a resource determines the security policy for that object, i.e., a user or a process with access permission is capable of passing that permission on to other subjects. For this reason, it is called a discretionary scheme. When accessing resources, Linux uses the identity of the requesting user to determine whether they can access them. This mechanism is shown in Fig. 3.2.

Access control mechanisms such as DAC attempt to proactively protect the operating system by limiting the potential damage to these dangers on the basis of the Principle Of Least Privilege. The POSIX capabilities released with Kernel 2.2 divide the privileges traditionally associated with superusers into several units that can be activated and deactivated independently [56]. The traditional Unix access control distinguishes between privileged processes and unprivileged processes. Unprivileged processes must pass a complete permission check based on the process's credentials, while privileged processes bypass all kernel authorization checks. Capabilities can, therefore, be seen as a refinement of DAC. The feature allows dividing the root user's rights into currently more than 30 different independent capabilities. Each privileged action is mapped

to a capability. Non-privileged users and processes can use capabilities to perform privileged operations, provided they have the necessary capability to do so [63].

Containers are able to use 14 capabilities under Docker by default. For example, `CAP_CHOWN` is used to change the ownership of a file, or `CAP_NET_BIND_SERVICE` binds a socket to low numbered network ports [79]. Using the `-cap-add` and `-cap-drop` parameters of the Docker run command, capabilities can also be individually enabled or disabled during container startup. Since applications within containers do not require every privilege of a root, a reduced set of capabilities is sufficient. With the reduced capabilities, a container is prohibited from all mount operations, and access to raw sockets to prevent packet spoofing or loading of kernel modules [69]. Reduced privileges impede attackers from taking control of the host since the vectors of attack decreased.

In addition to Linux capabilities, other technologies for restricting the access rights are also offered by Docker. Seccomp, for example, is used to restrict access of processes to system calls [31]. However, this feature is only available under specific kernels. Therefore, as with the mandatory access control systems SELinux and AppArmor, which are also only available to the Docker host operating system under certain conditions, this is discussed in Chapter 5 as a tool to improve container security.

3.1.3 Resource Management through Control Groups

In section 2.3.5, it was explained that so-called Denial of Service attacks could violate the availability of services. Server applications or environments that serve multiple users simultaneously and which are separated by virtualization are affected even if they are not targeted themselves, but are hosted on the same host as the attacked application. For example, an attack may overload a service to such an extent that it uses all resources of the host system so that other services of the same host cannot be supplied with sufficient resources themselves. To prevent such a scenario, the Linux kernel offers control groups (`cgroups`) [107].

Initially started by Google engineers in 2006, the control group feature was integrated into the Linux kernel in version 2.6.24 in 2008. Linux control groups allow processes to be organized into hierarchical groups whose use of different

resource types can be restricted and monitored [44]. A pseudo file system called cgroupfs provides the kernel interface to control groups. The grouping is implemented in the cgroup kernel code, while resource tracking and limiting are implemented in a number of resource-like subsystems [57]. In addition to versatile and granular functions for managing a host's resources, control groups can be used to implement complex procedures for correcting processes that exceed limits. Within a control group, all processes are bound to a set of limits defined by the cgroup file system. Resource controllers are kernel components that modify the behavior of processes in a cgroup. They are the components responsible for assigning resources to control groups, for example, by limiting the amount of CPU time and memory available to a cgroup or freezing and resuming the execution of processes in a cgroup [57]. The hierarchical arrangement of groups in controllers is defined by creating, removing and renaming subdirectories within the cgroup file system. Usually, the limits, controls, and settlements provided by cgroups affect the entire subhierarchy below the cgroup in which the attributes are defined. For example, the limits set at a higher hierarchy level cannot be exceeded by progeny groups [107]. The development of the resource controllers, however, was mostly uncoordinated, resulting in many inconsistencies between the controllers and the management of the group hierarchies. Therefore, control groups were reimplemented under the name `cgroup v2` and released in Linux kernel 4.5. However, Docker's default container runtime *runC* still does not use the second version of cgroup which is due to the missing support of cgroups v2 as default setup with `systemd` [28]. Otherwise, using version 2 on a host that is using the first version cannot be applied because moving all of the processes in the system to the v2 equivalent would be required. Since such a solution would change a Linux distribution's policies, the integration of cgroups v2 is postponed until also group namespaces implement the usage of version 2.

Docker uses cgroups v1 to individually adjust resources such as CPU and memory to each container. In other words, cgroups ensures that a single container cannot use all CPU computation time, space in the memory or storage I/O of the host [79]. Control groups are based on the idea of resource limits that define soft and hard limits assigned to each process but which were limited to a single process. Many container technologies have therefore added their own features to resource limits. For example, Solaris offers the use of resource pools that implement resource partitioning. FreeBSD developers have added hierarchical resource restrictions for jails [44].

By default, a container receives 1024 CPU shares. CPU shares determine the

amount of relative computing time a process gets on a processor. The relative proportions of the shares are decisive, i.e., two processes with 1000 CPU shares each receive the same computing power. How much time that takes on the actual processor depends on the underlying model [67]. The run command in the Docker client allows specifying the resource usage of a container. For example, executing the following commands provides the *node* container twice as much CPU power compared to the *redis* container.

```
andreas@host:$ docker run -cpuset-cpus=0 -c 1000 node:alpine
andreas@host:$ docker run -cpuset-cpus=0 -c 500 redis:alpine
```

3.2 Security through Features in the Docker Ecosystem

After discussing the native security mechanisms of Docker containers based on Linux functionalities in section 3.1 the security of the Docker ecosystem is examined in the following sections. The Docker ecosystem encompasses the entirety of all components and interaction possibilities that exist in connection with Docker. The focus of this investigation is on the application level. Therefore, it is now considered which security technologies are provided by the Docker platform so that tools and services such as Docker Hub can be used securely by developers and administrators.

3.2.1 Docker Security Scanning for Image Inspection

As will be shown later in section 4.2, vulnerabilities in existing public images are not uncommon. For example, a study in 2015 found that more than 40% of the images shared by users in the Docker Hub contained severe security vulnerabilities. Moreover, 36% of the official images, which were considered to be well maintained, had very critical security flaws [61]. Official images can be found in official repositories of the Docker Hub. Official repositories are a curated set of repositories designed to provide essential basic operating system images (e.g., Ubuntu, CentOS, or Alpine) and drop-in solutions for popular programming language runtimes, data stores, and other services. These images were specially created according to best practices for Dockerfiles and included detailed documentation. Besides, a team paid by *Docker Inc.* will ensure maintenance and regular security updates in official repositories. According to information from *Docker Inc.*, this team works closely with the software manufacturers of the respective products as well as with security experts and the Docker community [26]. The team's work is public, as the code underlying the official images is always publicly available through GitHub.

To automatically detect vulnerabilities in images of the Docker Hub, the former project Nautilus was presented as Docker Security Scanning in May 2016 [102]. For proactive risk management, scanning provides a detailed security profile of scanned images. Scanning is performed at the binary level before the images are deployed. It creates a bill of materials (BOM) with a detailed report of all layers and components [79]. Additionally, the system regularly searches for

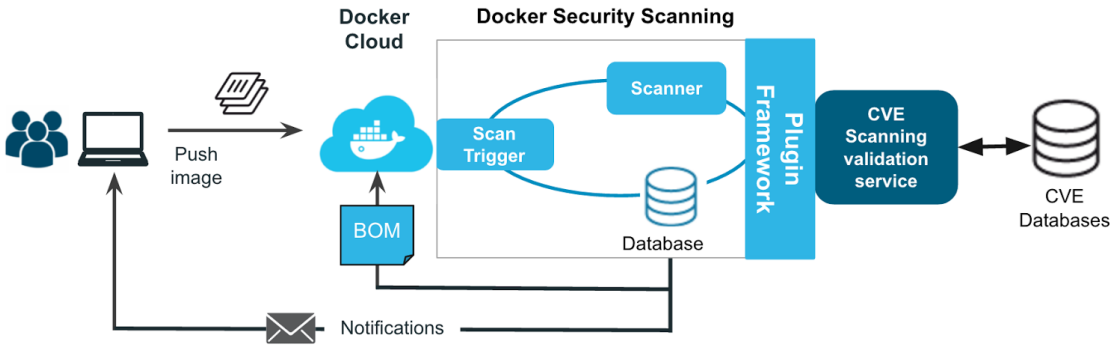

Figure 3.3: Pushing an image to a repository triggers a series of events. Docker Security Scanning comprises a scan trigger, the scanner, a database, a plugin framework and CVE validation services [102].

new security vulnerabilities and informs users if a security flaw occurs.

After a user or organization pushes an image into a repository, a series of events is triggered [26]. First, the scanner separates the images into the individual layers and components, which is possible due to the metadata for building into images which were presented in section 2.2.2. An image layer can potentially contain multiple packages or components. Their name and version numbers identify those. After finding out the packages and components of each layer, a binary level scan of the contents of the packages is also performed [102]. For both partial procedures, a check using the validation service against several CVE databases follows. The Common Vulnerabilities and Exposures (CVE) is an industry standard whose goal is to introduce a uniform naming convention for security vulnerabilities and other vulnerabilities in computer systems [2]. After querying many CVE databases, a detailed BOM is generated on this basis and stored in the Docker Security Scanning database [102]. The results can then be viewed in the repository through a web user interface. Fig 3.3 illustrates the procedure.

The individual layers and components of the images contained in the BOMs enable the Docker Security Scanning service to check them at a later date easily. If a new vulnerability is found in a component of an image and is reported in a central CVE database, Docker Security Scanning can query its own database for affected images and tags with corresponding corrupted packages and report it to the repository administrator [102]. The transmitted information can then

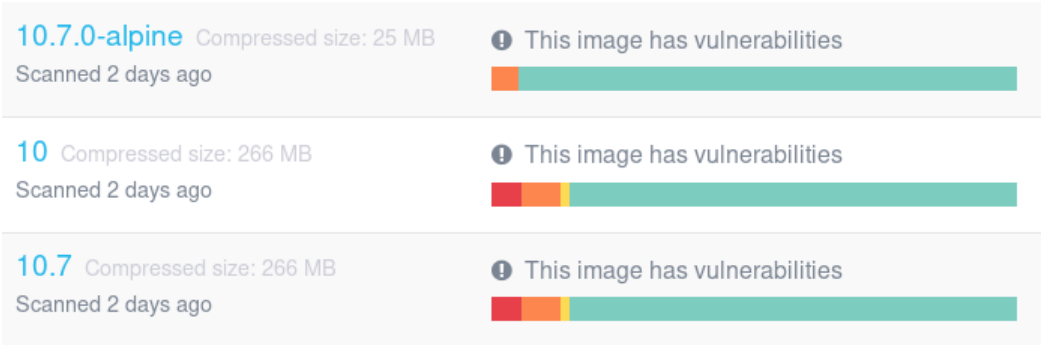

Figure 3.4: Multiple images in the official *node* repository. Each image is identified by a tag. Apparently, the Alpine Linux-based image has fewer and less dangerous vulnerabilities.

be used to make informed decisions on further action to combat the security vulnerability [102].

Docker Security Scanning was originally available for official images of the Docker Hub as well as for images from private repositories of Docker Cloud [79]. Docker Cloud leverages Docker Hub and includes advanced services for managing infrastructure, builds, testing, and deployment. For activating the scans, only one adjustment in the web user interface was sufficient [102]. By the end of March 2018, the scanning feature in private repositories of Docker Cloud was discontinued [26]. Besides the official images, Docker Security Scanning is currently only available for paying customers of the Docker Enterprise Edition in the form of the Docker Trusted Registry [30]. Docker Trusted Registry will be examined in section 3.2.2.

The security of official images can be checked in the Docker Hub web interface in the tab "Tags" of a repository. For each tag, a bar is displayed with different colors to indicate the severity of vulnerabilities contained in an image. If an image has no security weaknesses, a short information text is displayed along with a completely green bar. This view is only visible to registered users of the Docker Hub. Fig. 3.4 shows an exemplary snippet of the official *node* repository.

3.2.2 Verification and Distribution of Images

At the end of 2014, vulnerabilities in the verification and distribution of images were discovered that endangered both data integrity and the availability of a host system by running unverified and manipulated files [83]. Since Docker 1.8, mechanisms have been implemented to improve the verification and distribution model of images.

One of the improvements was the introduction of SHA-256 hash values over each layer for their unique referencing [99]. The previously used randomly generated UUIDs could not guarantee this unique referencing because they are not deterministic. However, SHA-256 is considered cryptographically secure, which means that the generated layer IDs are collision-proof and therefore unique.

The hash function ensures the integrity of the layers of an image. The correctness of the data in the layers is validated by comparing a calculated hash of a layer with the referenced hash entry in the image metadata in the manifest. In the manifest, the referenced hash values of the layer are structured in the form of a tree. Since version 2 of the manifest, the manifest file can be digitally signed to ensure the integrity of the metadata [29]. More detailed information on the procedure is given in section 2.2.4.

Furthermore, with Docker Content Trust, a package and distribution model to prevent manipulated images, replay attacks and man in the middle attacks was implemented. Docker Content Trust allows developers to sign their images when they are pushed to Docker Hub or Docker Trusted Registry [79]. Digital signatures of data sent to and received by remote Docker registries can be used to perform client-side integrity checks and to verify the publisher of certain image tags [22]. Since this feature is currently switched off by default, the environment variable `DOCKER_CONTENT_TRUST` must be set manually to 1 on the Docker host machine to activate it. [22]. Using this setting ensures that only trusted images are pulled from a repository or built and run on the host.

Docker Content Trust can also provide valuable context, such as ensuring that an image has been signed for use in a production environment or whether an image has been replaced with a newer version and is therefore out of date [79].

With Content Trust enabled, commands operating on tagged images of the Docker

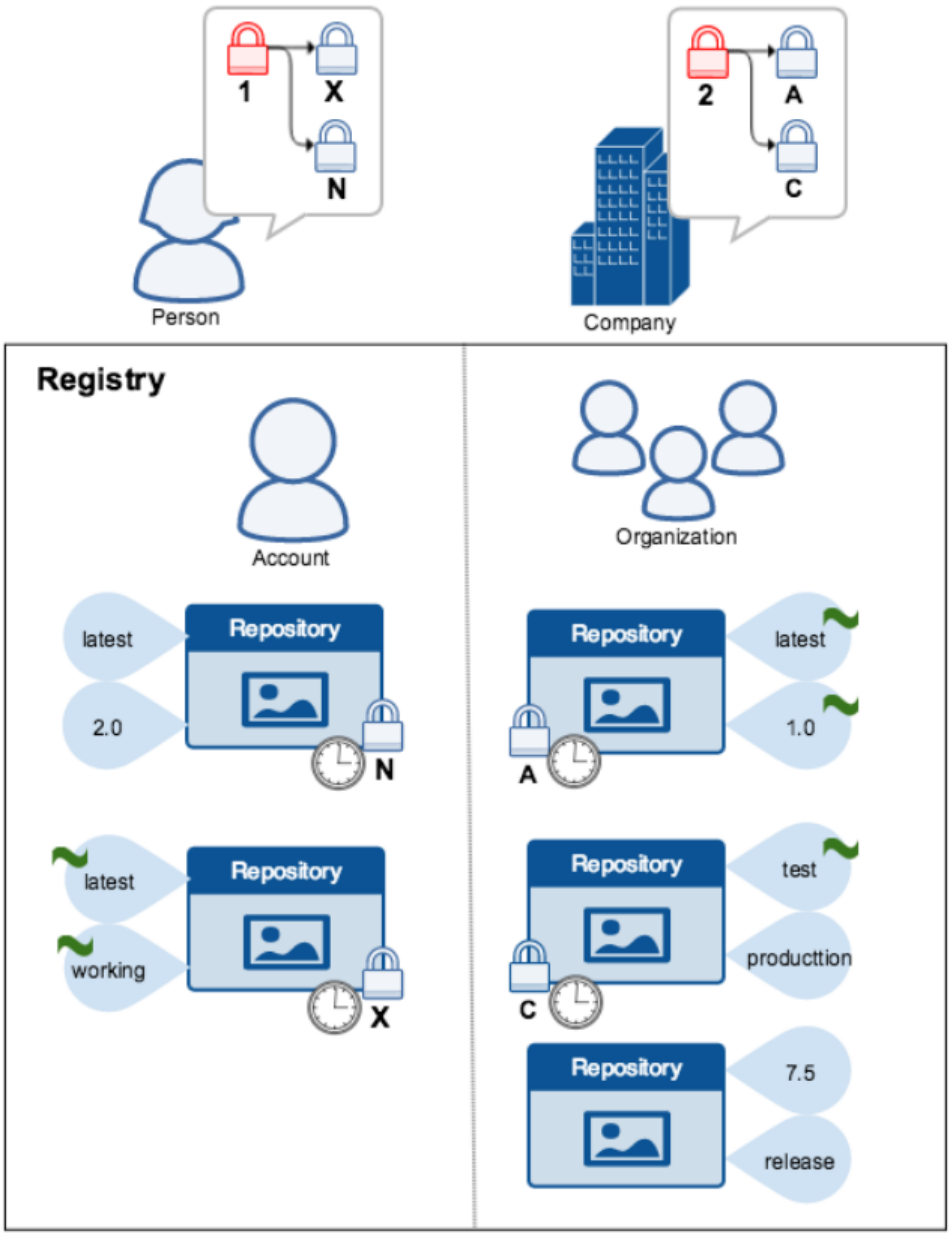

Figure 3.5: An illustration of the relationships between the various signing keys. The red padlock symbolizes an offline key, and blue ones show tagging keys. The clock stands for timestamp keys. Image tags that are signed are highlighted with a green mark. This points out that developers might have deliberately disabled Content Trust for image tags without a mark, for example, if further development is required and they should therefore not be used by a client equipped with Docker Content Trust [22].

client must have either content signatures or explicit content hashes [22]. Currently, this includes the commands push, pull, build, create and run. For instance, the command `docker pull someimage:latest` succeeds only if `someimage:latest` is signed. Solely an explicit content hash like in `docker pull myimage@sha256:1234abcd` would also work.

Trustworthiness is ensured with signing keys, which are created the first time an operation using Content Trust is performed [22]. A key set consists of an offline key, tagging keys and server-managed keys such as timestamp keys [22]. The offline key is the basis for Content Trust of an image tag since it is used to create tagging keys. It belongs to a person or organization and always remains on the client side. The offline key is derived from a password. Since it plays a crucial role, it should be safely stored and preferably backed up. With a tagging key, it is possible to sign tags of an image. It is connected to a repository so that only participants with a tagging key can push into this repository. Timestamp keys provide freshness security guarantees for a repository. These keys are created and managed on the server. Fig. 3.5 presents the relationships between the keys.

3.2.3 Docker Swarm Mode

Although the focus of this thesis is not on network security or the interaction of several Docker hosts, the security of Docker Swarm should be considered briefly here, because Swarm mode is now integrated as a native component of the Docker engine. Before version 1.12, Swarm was available as a standalone product [79].

Docker Swarm was already presented in section 2.2.5. It enables to cluster multiple Docker hosts and to deploy applications in a declarative way. Each swarm consists of managers and workers. Manager nodes form the control level of the cluster and are responsible for cluster configuration and dispatching jobs to worker nodes. Worker nodes then execute the application code as a container.

Swarm mode enables several security features out-of-the-box, including mutual authentication via TLS, secure join tokens, certificate authority configuration with automatic certificate rotation, and an encrypted cluster store [79].

Swarm join tokens Every swarm maintains one token for joining new man-

agers and one token for joining new workers. Join tokens are required to add new nodes to an existing cluster. Every join token is comprised of 4 distinct fields separated by dashes, namely `PREFIX-VERSION-SWARM ID-TOKEN`. The swarm ID is a hash of a swarm's certificate. The token portion determines whether it can be used for manager or worker joins. If an adversary has compromised a join token, it can be revoked with a single command, which, in turn, also issues a new one. Join tokens are stored in the cluster database [79].

TLS and mutual authentication Every node of a cluster receives a client certificate which is used for mutual authentication. Using the certificate, the cluster and the role of a node can be found out. Besides, the certificate contains two dates to define the period of validity [79].

Certificate authority configuration An administrator of a cluster can adjust the certificate rotation period. A short validity period has the advantage that in case of a compromise a certificate is only valid for a certain period of time [79].

Encrypted cluster store This is where configuration files and the status of a cluster are stored. The implementation is based on *etcd* and is automatically configured to replicate itself to all managers in the Swarm [79]. Many critical components such as Docker networking leverage the cluster store. The Docker engine maintains the cluster store automatically, but in production environments, users are encouraged to handle backup and recovery solutions proactively.

In addition to the automatically enabled security features of Swarm shown above, there is also Docker Secrets. Since Docker 1.13, it is possible to distribute and manage passwords, TLS certificates, SSH keys and other secrets to clusters. Secrets are encrypted at rest, encrypted on transport, mounted in in-memory filesystems, and operate under a least-privilege model where they are only made available to services that have been explicitly granted access to them [79]. Secrets are stored in the encrypted cluster store, of which each manager has a replica. When a worker receives a secret, it is stored as an unencrypted file on `/run/secrets/`, which is an in-memory filesystem. If a service that has used a secret is terminated, the in-memory filesystem is torn down and the secret flushed from the node [109].

4

Docker Security Threats

In the last chapter, it was demonstrated that Docker provides some inherent features to ensure the security goals defined in section 2.3. Various mechanisms are also used in the Docker ecosystem to ensure the safety of infrastructures. Today's containers are embedded in a complex system that involves much more than running programs in a virtualized environment. Instead, Docker communicates with various repositories and third-party vendors such as Github. Besides, containers are embedded in automated development and deployment chains, which further increases complexity. Through images from public registries, third-party code is inevitably executed, which is frequently subject to security vulnerabilities [91, 38].

A security analysis, therefore, requires a multi-layered consideration of the Docker components and their integration into an overall system. This chapter analyses and evaluates the Docker components layer by layer for possible security threats and their exploitation. By using a bottom-up approach starting at low-level attack points up to the integration of Docker into a deployment pipeline, this analysis provides a comprehensive view of security threats in a Docker infrastructure.

4.1 Approach

Before analyzing a complex system like Docker, a suitable threat modeling approach must be chosen. However, it must be clear that there is never a perfect approach and that it is always possible that certain risks are not taken into account. Subsection 4.1.1 briefly presents and discusses some strategies. Afterward, the further procedure for the analysis in this thesis will be described.

4.1.1 Discussion of several Threat Modelling Strategies

In order to analyze the threats for a specific system, several approaches are possible and are also applied in practice. The most traditional way to enumerate threats is brainstorming [90]. A group of experts collects potential dangers and assesses them according to the respective risk. The quality of threat modeling in this variant depends strongly on the experience and domain-specific knowledge of the experts. Furthermore, there is a risk that unstructured discussions may also produce unstructured, incomplete or unrealistic results. Structured strategies for threat modeling have gradually emerged, including the three most common asset-focused, attacker-focused, and software-focused. Such structured and consistent strategies help to achieve the necessary countermeasures for the respective risks. A combination of the three strategies is rarely beneficial, as it tends to be confusing [90]. The three strategies are briefly presented and evaluated below.

4.1.1.1 Focusing on Assets

The asset-focused approach includes identifying a system. An asset is something of value [90]. In threat modeling, usually, assets are either something an attacker wants, something that the operator of a system wants to protect, or stepping stones to either of these. These three types of assets often overlap when looking at them from a different perspective. For example, the payment information of users may be of value for an attacker. Protecting a company's reputation may be a goal of an operator of a system. Finding out the password for a database management system may be a stepping stone for an attacker.

Starting with a list of assets, a set of experts then considers attack scenarios for

each asset. Each asset should be assigned to a particular computer system, and their connection to other assets and computer systems should be modeled in order to show interconnections [90]. The threats and attack scenarios can then be found with techniques like attack trees or STRIDE.

By focusing on assets, non-technical stakeholders may be able to contribute to the process by helping to prioritize on certain assets. However, in most cases, it is not possible to derive a direct threat or a set of steps of an attack starting with an asset [90]. Moreover, prioritizing the found assets also does not provide a benefit in finding or fixing threats.

4.1.1.2 Focusing on Attackers

The attacker-focused strategy involves defining profiles of different adversaries. Such profiles include the capabilities, resources, and motivation of an attacker. In practice, attacker profiles range from simple lists to data-derived personas [90]. Given an adversary's profile, security experts can use it to make a structured brainstorming approach.

Attacker-focused threat modeling tends to result in human-centered attacks rather than technical attacks [90]. On the one hand, human-centered attacks may be easier to understand by non-technical stakeholders who are keen to publish a product quickly. On the other hand, other attacks may not be considered in the analysis. Furthermore, the attacker profiles tend to be incomplete and often do not provide sufficient structure for the analysis.

4.1.1.3 Focusing on Software

By understanding software and data flows in a system, the software-focused approach helps to expose accumulated complexity [90]. In large software projects, understandings of individual components and their interaction differ. A shared understanding of software within a development team can be beneficial even before identifying the threats. Thus, a comprehensive model of the software or the system can result in a substation improvement of the security.

Including software developers in the threat modeling process increases their awareness of security as well as the quality of the software. Moreover, by un-

derstanding different assumptions of, e.g., a component, developers can build improved solutions.

4.1.2 Realization of the Software-Focused Approach in this Thesis

The last section briefly presented and discussed the three most common strategies of threat modeling. It has emerged that a software-focused approach helps to understand the system better and find solutions to problems. For this reason, this thesis pursues such a strategy. The implementation is loosely oriented to the approach of Myagmar et al. [72].

More precisely, the next step is to identify the different layers in a Docker infrastructure. Afterward, STRIDE is introduced, with the help of which the threats for each layer are found out in the following analysis. Subsection 4.1.2.3 briefly explains how possible exploits for each identified threat are found in this thesis. Finally, it is explained how the risk assessments were made in the analysis in section 4.2.

4.1.2.1 Identifying the Layers

A Docker infrastructure can be very intricate. For example, by using an orchestration tool such as Docker Swarm, a cluster over several Docker hosts can be created. However, as stated in section 1.2, this thesis does not focus on such a scenario. Nevertheless, a Docker infrastructure has to investigated from different layers. In this thesis, by starting at the hardware level of a production machine, several aspects of a production environment will be analyzed. An overview of all layers is shown in Fig. 4.1.

The selection of layers is based on previous work that analyzed the security of Docker, e.g., [53, 91, 11, 71, 61, 82, 38, 9]. However, most of them focused on one layer and did not conduct structured threat modeling but mainly concentrated on exploiting security vulnerabilities.

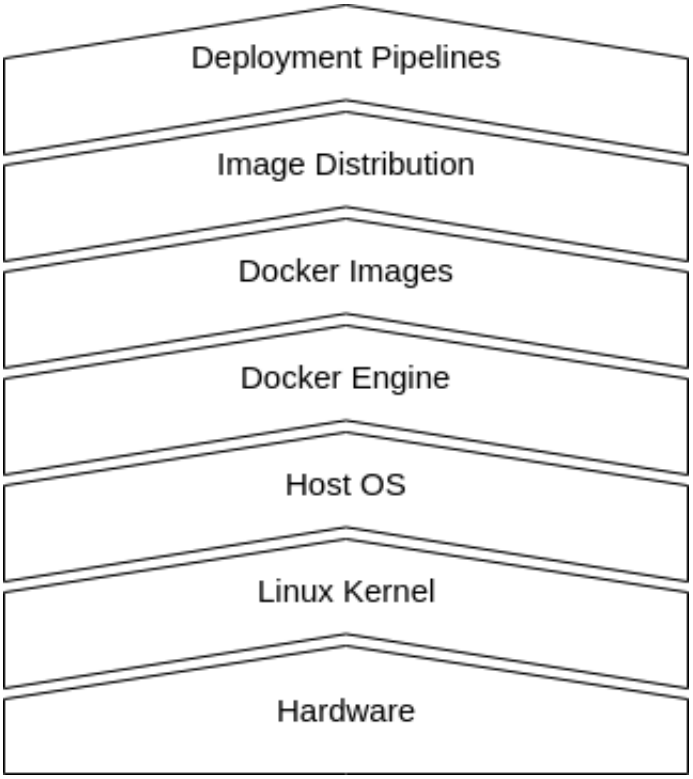

Figure 4.1: Overview of the investigated layers. Each layer will be analyzed with STRIDE to identify the threats.

4.1.2.2 Using STRIDE to identify the Threats for each Layer

In order to find threats for a system, several approaches are possible. For example, attack trees, which were introduced by Bruce Schneier at the end of the 1990's, provide a structured way of describing an attack. Each attack is represented in a tree structure with the attack goal as the root node and ways of achieving the goal as leaf nodes [86]. However, since the STRIDE approach was designed to help developers, the analysis in this thesis will use this framework.

STRIDE is an acronym for Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, and Elevation of Privilege [90]. It was developed by Loren Kohnfelder and Praerit Garg around the same time when Schneier introduced attack trees. Each of the threat classes violates one of the security goals set out in section 2.3. The following list will briefly define each threat category [90].

Spoofing Pretending to be another person, role, machine, file or process. Spoofing violates the goal of authenticity.

Tampering Altering, adding or removing data on a storage device, a network or in memory. Tampering violates the goal of integrity.

Repudiation Claiming to be not responsible for something, e.g., an action such as clicking on a specific link. Repudiation violates the goal of non-repudiation.

Information Disclosure Disclosing information to unauthorized processes, data stores, networks or stakeholders. Information disclosure goal the property of confidentiality.

Denial of Service Utilization of resources that would be necessary to maintain the service. Denial of service violates the goal of availability.

Elevation of Privilege Allowing a user or a process to do unauthorized actions through corrupting a process, missed or erroneous authorization checks. Elevation of privilege violates the goal of authorization.

4.1.2.3 Finding Exploits for the identified Threats

For each identified threat, possible exploits will be presented in the analysis in section 4.2. In order to find suitable exploitation techniques, this thesis considers already presented attacks from related papers, attacks from practice that have made it into the media, as well as attacks that haven't been considered yet.

4.1.2.4 Assessing the Risk for the identified Threats

It is difficult to assess the risk of a threat because comparative values or statistics are often missing at all. This concerns both the probability and the dangerousness of an occurrence of a threat. Nevertheless, this thesis tries to make an estimation. The assessment is then usually based on the discussions in the respective papers, which have, for example, analyzed the effects of a particular security vulnerability. Overall, the risk assessment will include both likelihood and potential damage.

4.2 Analysis of Docker Security Threats

The attacked layer can distinguish attacks on Docker and its ecosystem. In this way, an entire attack surface should be captured as accurately as possible. In the following sections, the individual layers found in 4.1.2.1 are examined in more detail using a bottom-up procedure. Starting at the hardware level, vulnerabilities in the Linux kernel, the host setup, Docker respective *libcontainer*, inside images and in the image distribution process are then systematically examined one after the other. Finally, as the last layer, the integration of the Docker ecosystem into a production pipeline as recommended by the Docker vendors is investigated. As described in subsection 4.1.2, the analysis is divided into three steps for each layer: First, the threats of a layer are identified. Secondly, it will be shown how these points of attack could be exploited or how they have been applied in the past. Thirdly, the severity and likelihood are evaluated and discussed briefly. Detailed suggestions to increase security for the respective attack scenarios will be shown in chapter 5.

4.2.1 Hardware

The hardware vulnerabilities Meltdown and Spectre, disclosed at the beginning of 2018, have revealed a new class of vulnerabilities [48, 52]. They are based on speculative jump prediction for faster execution of programs and endanger the majority of modern processors. A new Spectre attack was released in July 2018 under the name NetSpectre. Instead of executing local code, it can read arbitrary memory over a network [88]. Eight other Spectre variants were already presented as Spectre Next Generation in May [46]. Since Meltdown and Spectre were the first security vulnerabilities of this kind, this section focuses on these flaws.

4.2.1.1 Threats

Spoofing -

Tampering -

Repudiation -

Information Disclosure Unautorized processes can read arbitrary memory

Denial of Service -

Elevation of Privilege -

4.2.1.2 Exploitation

The Meltdown vulnerability is listed under CVE number CVE-2017-5754. The specific security bug is that the processor temporarily reads and processes the contents of a memory cell as part of out-of-order execution, although the calling process has no rights for this memory section. This changes the runtime behavior when accessing the cache and allows the memory contents to be traced back using an exact time measurement, even after the result of speculative execution has been discarded [52]. Meltdown is relatively easy to exploit. An example code shown in the corresponding paper comprises only a few lines. Basically, a memory location, to which an adversary actually has no access, is read in a loop. However, further operations with the value from the accessed memory location are performed speculatively before the kernel exception is raised. For example, the read value is used as an index for an array. This loads a memory page into the cache whose address depends on the content of the memory location to be read. Using this information, an attacker could retrieve the complete memory step by step, even if it was not authorized to do so [52].

As with the Meltdown flaw, the prerequisite for an attack using the Spectre vulnerability is that the processor supports out-of-order execution. According to the idea of out-of-order execution, the state of the processor is as if the corresponding instruction had never been executed, even in the event of a misspeculation. The Spectre attack takes advantage of the fact that the state of the system still changes at specific points, even if the result of speculative execution is discarded [48]. These changes, e.g., loading a memory page into the cache, serve as a hidden channel to eject information from the address space of the attacked process. The receiving process decodes the transmitted information from the changes in the system and can thus read it [48]. For Spectre, executing the attack requires the attacked process to execute particular instructions given by the attacker speculatively. This can be achieved, for example, by Return Oriented Programming using a buffer overflow or in the form of a program provided by the attacker and executed by the user (e.g., in an interpreted script language such as JavaScript) [48]. In this form, the Spectre vulnerability can be used to

extract information from the memory space of the executing interpreter (e.g., web browser). Thus, an attacker might be able to read passwords from the web browser's password store via a malicious JavaScript loaded from a web server [48]. In the context of Docker, an adversary may use a bug of a service running inside a container. For instance, a flaw in a web server allowing remote code execution can be used to run Spectre scripts to retrieve information of the host's memory, including data from other containers running on the same host.

In addition to Meltdown and Spectre, other attacks are also conceivable to restore secrets via side channels such as power consumption, electromagnetic leaks or sound [53, 87]. Already in 2017, an attack on Intel SGX showed that 96% of an RSA private key could be reconstructed with a side channel [87]. The attack worked in a native and a Docker container environment. The attack is called "Prime+Probe" attack and uses SGX specific CPU instructions as well as a timer to detect cache hits.

4.2.1.3 Risk

Potentially all Intel processors implementing out-of-order execution were affected by Meltdown, i.e., all processors since 1995, except the Itanium and Atom processors produced before 2013. Also, all common operating systems were generally vulnerable to Meltdown. However, countermeasures in the form of Linux kernel patches have been proven to prevent the attack [80].

The spread of Spectre is considered even more significant than of Meltdown, as processors from other manufacturers than Intel are also affected. The regularly disclosed variants of Spectre show that it is more advanced than Meltdown. As a result, Intel announced that in the second half of 2018 processors would be delivered that would provide additional shielding between running applications and between processes with different access rights by redesigning parts of the processor architecture, protecting them against branch target injection [7, 60].

Although hypervisor-based virtualization and bare-metal usage are also affected by the hardware security flaws, it must be emphasized at this point that the isolation of containers can be broken with Meltdown, Spectre, and Prime+Probe. Overall, the risk can be classified as moderate. Although concrete attacks on hardware were demonstrated, NetSpectre, for example, can only extract about 15 bits per hour [88].

4.2.2 Linux Kernel

All containers use the same kernel of the host. The security mechanisms of Docker for isolation, restriction of access rights and resource management depend significantly on kernel functionalities as shown in chapter 3.1.1.

4.2.2.1 Threats

Spoofing -

Tampering Using a memory corruption vulnerability in the kernel, arbitrary code can be executed. Thus, an attacker could modify files, which violates the integrity of the files.

Repudiation -

Information Disclosure Kernel bugs may lead to a situation where non-root users can read the memory of the kernel space.

Denial of Service If the kernel has a memory corruption vulnerability, arbitrary code can be executed by an adversary. For example, the attacker may run computing-intensive programs to block the CPU of a machine.

Elevation of Privilege Programming errors in the Linux kernel can lead to missing authorization checks.

4.2.2.2 Exploitation

A study analyzed all 141 Linux kernel vulnerabilities reported to CVE databases between January 2010 and March 2011 and categorized the vulnerabilities by the type of attacks that can exploit the vulnerability [12]. The study identified ten categories of vulnerabilities based on the kind of programming mistake the developers made [12]. The most essential five of these are briefly discussed in this section.

Missing pointer checks If the kernel does not perform the `access_ok` security

check or replaces it with supposedly faster operations such as `__get__user`, it does not correctly validate whether the value of a user-provided pointer or variable points to only user-space memory. With this error class, it was possible to read or write as an unprivileged process in arbitrary kernel locations. Six of eight of such bugs led to memory corruption [12]. Missing pointer checks lead to elevation of privilege, but with memory corruption, each of the other threats can be realized by an attacker.

Missing permission checks The kernel performs a privileged operation without checking if the calling process is allowed to do so. In general, such errors led to a violation of a kernel security policy, namely 15 out of 17 reported in the investigation period [12]. Thus, an elevation of privilege can be expected with such a vulnerability in the kernel.

Buffer overflow This error class occurs when the kernel incorrectly checks the upper or lower bound when accessing a buffer, allocates a smaller buffer than it is supposed to, uses unsafe string manipulation functions, or defines local variables which are too large for the kernel stack [71, 12]. Attacks with write access to memory are usually memory corruption (13 out of 15), reading attacks involve information disclosure (2 out of 15).

Integer overflow Performing an integer operation incorrectly results in an integer overflow, underflow, or sign error. Attacks similar to those allowed by buffer overflow vulnerabilities can be achieved by an exploit to trick the kernel into using incorrect values to allocate or access memory. For instance, an overflow after multiplication may cause the kernel to assign a smaller buffer than required. An underflow after subtraction can cause memory corruption beyond the end of a buffer [12]. Sign errors in the comparison can circumvent the limits of the check and lead to the disclosure of information.

Uninitialized data Copying contents of a kernel buffer to the user space without setting unused fields to zero will result in potentially leaking sensitive information to user processes, such as variables on the kernel stack. With 29 vulnerabilities, this category has most of all. 28 of these led to information disclosure. Based on published information, however, further attacks are possible, which require, for example, the exact address of a key [12].

4.2.2.3 Risk

Security flaws cannot be excluded in large and complex projects such as the Linux kernel. It now contains more than 25 million lines of code [66]. In the Docker context, an adequate exploit can allow complete root privileges in the container and finally force the breakout from the container and the takeover of the host [9]. Kernel exploits can violate almost any of the security goals set out in section 2.3. However, virtual machines or native environments that use Linux are equally affected [82]. Overall, the impact Linux kernel bugs can be considered as high, since many different vulnerabilities are possible and the security of the Docker container runtime is strongly related to the security of the kernel.

4.2.3 Docker Host

Docker provides solid isolation between containers and limits access of the container to the host. As shown in chapter 3.1, the kernel functionalities namespaces, cgroups and capabilities ensure that. Docker also offers features in its ecosystem that contribute to the protection of container infrastructures. However, the default settings can be adjusted relatively quickly, which also has an impact on security. Using some options given to either the Docker daemon at startup or the command to start a container can allow containers to have extended access to the host [61].

4.2.3.1 Threats

Spoofing If one changes the TLS configuration for communication with remote registries, an attacker using a man-in-the-middle attack can pretend to be a trustworthy communication partner, e.g., the Docker Hub.

Tampering Extended access to the host when starting containers allows containerized applications to perform extensive interactions between the host and the kernel, which might include operations on the host's filesystem.

Repudiation Missing log files impede to trace an attacker.

Information Disclosure Mounting sensitive host directories in containers enables applications running inside containers to read files of the host.

Denial of Service Utilization of resources that would be necessary to maintain the service. Denial of service violates the goal of availability.

Elevation of Privilege Mounting the host's root filesystem with write permissions enables the execution of highly privileged executable program files.

4.2.3.2 Exploitation

Mounting `/var/run/docker.sock` into a container allows processes to any command that the Docker daemon can run, which generally provides access to the whole host system as the Docker service runs as root [81]. What might be useful in a development environment, is considered a bad practice for production environments. For example, in a development environment, one container should be responsible for extracting information about other the behavior of other containers on the host. However, despite all warnings, even some popular non-official repositories make use of mounting `docker.sock` into a container, e.g., `nginx-proxy` which has over ten million pulls in the Docker Hub. `nginx-proxy` enables therewith the automatical creation of reverse proxy entries for other containers [81]. Attackers could not only read the secrets of other containers, for example from environment variables but could also start new containers and give them root privileges.

Moreover, the Docker daemon can be exposed to a network. By making `/var/-run/docker.sock` accessible over HTTP, a remote Docker client can access and manage the Daemon over a network. However, Docker does not use TLS or authentication by default. As shown in a report about an attempted attack, the adversary was about to install a cryptocurrency miner [13]. First, the attacker identified the version of the running Docker daemon. Using error message, the adversary was able to install a corresponding Docker client version. Secondly, by using the command `docker import`, a compressed file with an image of the cryptocurrency Monero could be downloaded to the target machine. Only because the researchers, who deliberately set up the honeypot as a lure trap, explicitly forbade the `docker run` command, the image was not executed.

Using `-insecure-registry`, an insecure registry can be registered in the Docker daemon. It allows communication with a registry via HTTP without TLS or via

HTTPS with an unknown certificate. This setting paves the way for man-in-the-middle attacks. Furthermore, the option `-uts=host` allocates the container in the same UTS namespace of the host which allows the container to see and change the host's name and domain [61].

Since capabilities divide the root privileges into over 30 individual privileges and Docker uses only a subset of them by default, `-cap-add=<CAP>` allows adding additional capabilities for a container. More root privileges also make a container more harmful to the host. For example, with `-cap-add=SYS ADMIN` it is even possible to remount `/proc` to read-write mode, which is used to store information about processes [61]. Moreover, the host's kernel settings can be modified.

4.2.3.3 Risk

The displayed scenarios can lead to data leakage or denial of service, as they give the container more rights to access the host. However, this requires a proactive decision by an administrator [61]. By default, Docker has settings that represent a tradeoff between usability and security. If in doubt, the settings should only be changed to reduce privileges further. To do this, it is possible to use `-cap-drop` when starting a container. For example, if only root privileges are used to open ports, all default privileges except `NET_BIND_SERVICE` can be dropped. Overall, the risk of misconfiguring the Docker host can be considered as low. If the best practices and the recommendations, which will be presented in chapter 5, are complied with, many threats can be mitigated.

4.2.4 Docker Engine

This section mainly covers vulnerabilities linked to Docker or *libcontainer*. Since the beginning of the project, Docker and its components have had 17 security vulnerabilities that were recorded in CVE databases [17, 25]. Given the size of the project, this is a small number of CVE vulnerabilities compared to other projects [71].

4.2.4.1 Threats

Spoofing -

Tampering Processes inside containers can bypass the isolation and, therefore, escalate the container, such as in CVE-2014-6408. It has been shown that containers obtain read and write access on the entire host file system in the event of an outbreak [71, 9].

Repudiation -

Information Disclosure Vulnerabilities in the Docker engine can lead to access to special directories on the host, e.g., in CVE-2015-3630, CVE-2016-8887, or CVE-2016-8867.

Denial of Service -

Elevation of Privilege Bugs in the Docker engine allow container processes to execute unauthorized operations, such as in CVE-2016-3697.

4.2.4.2 Exploitation

There are currently three CVE vulnerabilities that have not yet been mitigated. CVE-2015-3290 and CVE-2015-5157 are based on bugs in the kernel's non-maskable interrupt handling and allow privilege escalation. They can be used in Docker because *libcontainer* does not prevent `modify_ldt` system calls. CVE-2016-5195 is based on a race condition in the Linux kernel's memory subsystem for handling copy-on-write breakage of private read-only memory mappings. This vulnerability allowed unprivileged local users to gain write access to read-only memory [25]. Furthermore, a study explored the application of penetration testing tools in the Docker context and tested the environment against common attacks [53, 87]. It has been shown that such tools also help the attackers in the context of Docker.

4.2.4.3 Risk

CVE-2016-5195 has not been sufficiently mitigated on all platforms, but for some host operating systems this is prevented by the combination of *seccomp* filtering of `ptrace` and the fact that `/proc/self/mem` is write-protected [25]. The other bugs have not been fixed yet.

At this point, it should be mentioned that CVE databases only list public security vulnerabilities. It is entirely possible that other vulnerabilities have been found, but they have never been disclosed to the public, and have been used to one' s advantage. This also includes the sale of security vulnerabilities on the underground market.

4.2.5 Docker Images

Using images from repositories of public registries like the Docker Hub directly in the production environment or indirectly as base images for building own images, one trusts implicitly in the source code of other parties. This is unavoidable in today's use of modern computers since every operating system is based on code that is not written by oneself. Based on this insight, it seems unreasonable at first glance that the security evaluation distinguishes between foreign source code of a Docker image and, for example, code of a software library. However, it should be noted that images always contain a combination of several programs and settings. Even the indirect use of a base image such as Alpine Linux, which is only a few megabytes in size as a minimal distribution, is not immune to security vulnerabilities [71].

4.2.5.1 Threats

Spoofing Code obtained from Git repositories during the build process can originate from an adversary instead of the producer of the software. Thus, an attacker pretended to be another person.

Tampering A debug tool that was accidentally not removed after the development process allows an attacker to modify data in a container.

Repudiation -

Information Disclosure Outdated software in images allows attackers to compromise the application inside the container. For example, an attacker may have access to a database with customer information due to an obsolete software package that allows attackers to read data from the database.

Denial of Service A maliciously prepared image from a public repository uses the computing power of the machine to mine cryptocurrencies.

Elevation of Privilege A maliciously prepared Dockerfile instructs the root directory to be mounted into the container at runtime (e.g., via `VOLUME / /my-escape`), which a process within the container can use for privilege escalation.

4.2.5.2 Exploitation

A study in 2015 revealed that 40% of all images in the Docker Hub contained high-priority CVE vulnerabilities [38]. Even 36% of the official images included high-priority, and 64% of the official images contained medium- or high-priority vulnerabilities. Thus, containers are vulnerable to remote exploits of vulnerabilities contained in the container images. Furthermore, images tagged "latest" also performed disappointingly with 23% and 47% respectively. Official images are the most popular in the Docker Hub, and many have several million downloads. Another recent work on the vulnerabilities in Docker's images analyzed the Docker Hub images using the DIVA (Docker Image Vulnerability Analysis) framework [91]. Analysis of 356,218 images showed that both official and unofficial images have an average of more than 180 vulnerabilities. Besides, many images have not been updated for hundreds of days, and the vulnerabilities often tend to transfer from parent images to child images [61].

When installing image build code from external sources, for example via Git, the security of the image depends significantly on the underlying code and the security of the connection to fetch the data. These remote repositories are an entry point for code injection. The attacker can either modify the code in the repository itself or exploit a possible insecure connection between the repository and the Docker daemon which is responsible for running the build process [103].

Docker Inc. promotes the DevOps movement, which allows developers to package their applications themselves to mix development and production environments, potentially creating vulnerabilities [24]. Development versions of packages or development tools can be included in the final version of the Docker image, which increases its attack surface. For instance, a debugging tool may have been added to the image for development purposes [61]. Besides, a Dockerfile instruction may have opened a port for debugging, through which the debugging tool could communicate with a development environment (IDE). An increased attack surface through the DevOps approach offers an attacker more options. Besides, a forgotten development tool points out the incautiousness in the development process, which allows concluding on further errors of a developer. Debugging tools sometimes allow direct intervention in a running program, so that the security property availability is violated.

Another attack vector is outdated software packages in an image. An outdated base image may be the reason why newer versions of a software package are not offered, and thus potentially vulnerable packages are installed by default during the Docker build process. For example, if within a Dockerfile that uses an old Debian version as base image, the `RUN apt-get install -y mysql` instruction may install an outdated version. Indeed, obsolete base images can already contain security flaws themselves and then become a serious issue at runtime.

Obsolete software dependencies in images can also be added to an image through the build process. It is not uncommon that during the build process, where an image is created layer by layer with the Dockerfile instructions, code is pulled from remote repositories like Github or Bitbucket. This code is also potentially outdated. In many Docker registry repositories, obsolete images are still available due to the multiplication of image structures. They are differentiated by tags. Fast development cycles usually focus on the latest versions [61].

If obsolete software dependencies are used for an image, classic methods for exploiting application vulnerabilities are possible if the container exposes an entry point [61]. This can be an involuntarily opened debugging port, but also input data. It is feasible that an outdated image could cause a web application with vulnerabilities to run in a production environment. If this vulnerability allows remote code execution, the security of the container or the host is at risk, for example, in combination with a vulnerability in the Linux kernel [61].

Besides, intentionally prepared images disguised as regular services in the Docker

Hub are a recurring problem. An example is the images of the Docker Hub repository "docker123321" that were examined in June 2018 [50]. A total of 17 images were found, which verifiably started a process to mine the cryptocurrency Monero at container runtime. The images were widely distributed as they were downloaded more than 5 million times. By executing the images in containers, Monero Coins worth about 90,000 dollars were mined. *Docker Inc.* was repeatedly informed about this repository on Twitter and Github, but it was only deleted more than half a year after the first notification. The images were named like popular services such as Tomcat and, in addition to the mining process, also performed the correct services to cover up the malicious intentions. However, such an attack can be considered a violation of availability, as mining partially blocked resources within the container.

By modifying a Dockerfile on, e.g., Github, an attacker is able to insert instructions that lead to elevation of privilege. Adding a root directory or a Unix socket into the container, processes within the container may use this to escalate the container.

4.2.5.3 Risk

The impact of security vulnerabilities in images can be described as moderate if one adheres to Docker's suggested best practices since by default Docker exposes only a limited attack surface. Otherwise, security vulnerabilities in images can have serious consequences. For example, if instead of using microservices-oriented images like Alpine Linux, as recommended, large Linux distributions like Ubuntu with a larger attack surface are used, the probability of exploitation is considerably larger. If containers are used contrary to the one-container-runs-one-service principle, one can speak of casting a container's usage as VM [68]. At this point, it should not be forgotten that larger containers also generate more complexity.

4.2.6 Image Distribution

Registries like the Docker Hub are one of the central points of attack for adversaries, as many are publicly accessible. Besides, they are used by many different actors so that an attack on the broad scope can scale. Docker Hub is comparable to package repositories of Linux distributions. Both provide clients

with executable software, in the case of Docker Hub it is images, in the case of Linux package repositories, it is program packages. In the Docker ecosystem, the Docker daemon acts as package manager on the host [61]. Thus, the Docker daemon is subject to the same class of security threats as usual package manager.

4.2.6.1 Threats

Spoofing Man-in-the-middle attacks between a registry and Docker daemon can inject malicious images. Thus, the attacker claimed to be one of the two entities.

Tampering Unpacking incoming packages can lead to overwriting specific files and directories of the host machine since the Docker daemon runs with root privileges.

Repudiation By publishing unsigned images, a software vendor cannot be held responsible in the case of a bug in the image. Users of such an image cannot prove that the vendor is to blame for problems.

Information Disclosure -

Denial of Service Unpacking untrusted zipped archives may cost a lot of computing power and, eventually, lead to full memory.

Elevation of Privilege -

4.2.6.2 Exploitation

As further studies show, attacks on package managers are entirely feasible [10]. An attacker must control part of the network between the client and the repository. Applied to Docker, this is a man-in-the-middle attack between the Docker daemon and Docker Hub.

A man-in-the-middle attack aims to inject selected data. Between the Docker Hub and the Docker daemon, an attacker wants to exchange an image. It is possible to take advantage of the fact that images are always compressed before

transport. Zip bombs are specially prepared files containing multiple nested compressed files [89]. A well-known example is 42.zip, which contains a few terabytes of zeros. It has a size of only 42 kB when compressed several times. Unpacking such a file requires a lot of computing power and, of course, memory, which can ultimately lead to a denial of service [61]. If the malicious image is prepared according to this pattern, the Docker daemon can also be overloaded.

A vulnerability in Docker found in 2014 and named as CVE-2014-9356 made so-called path traversals possible [25]. The bug allowed the decompression to be performed on absolute paths. Since the Docker daemon is privileged as root and the images are unpacked on the host file system, this bug could be used to exchange binaries of the host with selected binaries of the adversary.

4.2.6.3 Risk

CVE-2014-9356 was fixed in Docker 1.3.3. A similar vulnerability has not been found since then. Nevertheless, especially Zip Bomb attacks are theoretically still possible. Docker Content Trust makes such attacks almost impossible because images are signed there like with Linux package managers. However, Docker Content Trust requires to be activated manually. Overall, the risk can be considered moderate.

4.2.7 Deployment Pipelines

Docker Inc. offers numerous services for the integration of Dockers in automatic build and deployment pipelines. Each node can completely access the underlying code. An exemplary integration pipeline can be seen in Fig. 4.2. In the example, a push to an external git repository triggers an automatic build in the Docker Hub. The Docker Hub then sends an HTTP request to a Docker host that pulls the newly built image. The container is then restarted with the new image.

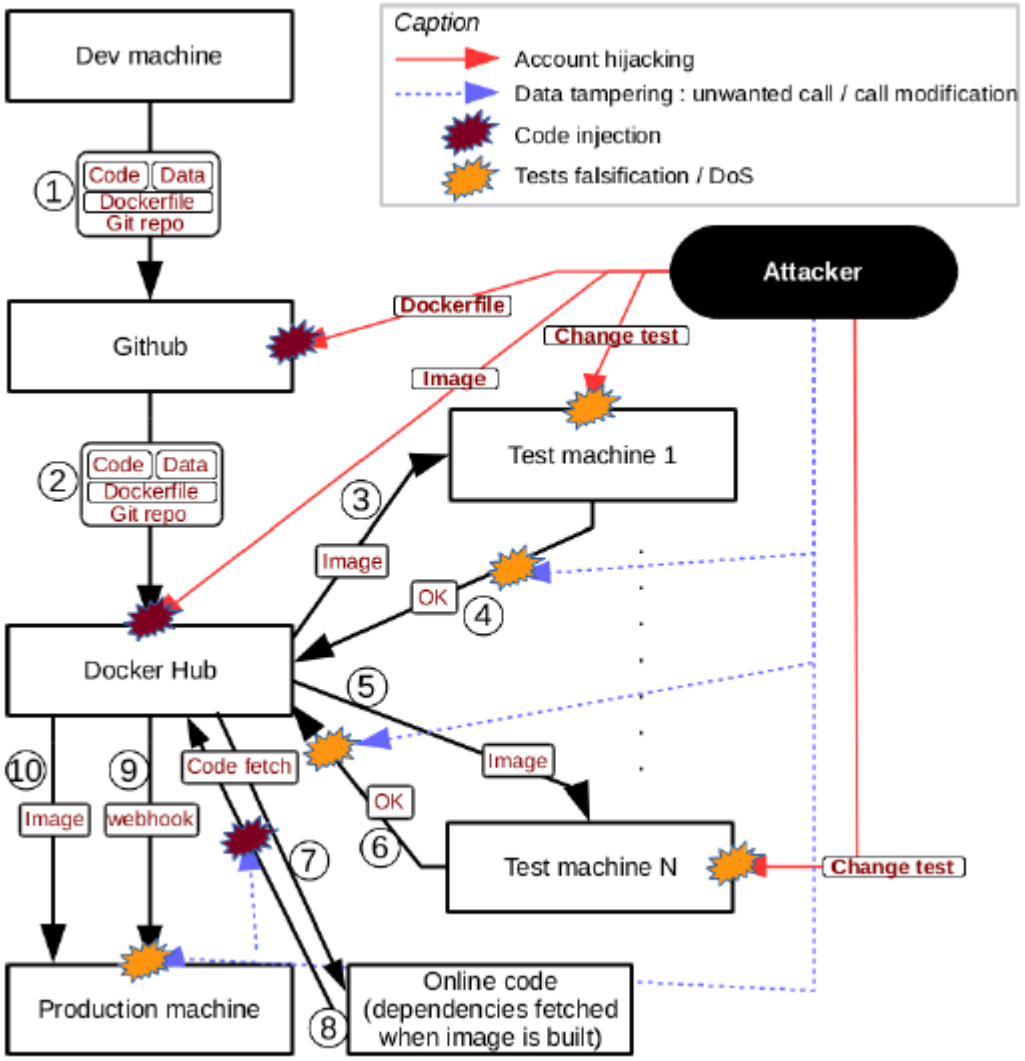

Figure 4.2: Automated deployment setup in the public cloud using Github, the Docker Hub, external test machines and repositories from where code is downloaded during build process [61].

4.2.7.1 Threat

Spoofing Hijacking, e.g., a Github account allows attackers to execute arbitrary code in the production environment if the pipeline is fully automated.

Tampering Unencrypted communication between entities of the deployment

pipeline allows an attacker to modify network traffic.

Repudiation Large and complex deployment pipelines impede to trace the cause of an attack. If the attack is due to a bad password of a developer or administrator, the reason is difficult to trace afterward.

Information Disclosure If attackers can exchange an image in the production process, they can modify it so that persistent data from a connected database is sent to a target of their choice.

Denial of Service Replacing the image on the production machine with an image for mining cryptocurrencies.

Elevation of Privilege An infiltrated image can expose certain directories of the root to the container in the production environment, which can then be used for privilege escalation.

4.2.7.2 Exploitation

The architecture shown in Figure 4.2 allows account hijacking, tampering with network communications (depending on the use of TLS), and insider attacks [61]. In general, it must be assumed that each of the pipeline nodes is vulnerable, which makes the security of the overall system dependent on the weakest element. In an experiment, a compromised Github account was used to demonstrate the execution of malicious code in a broad set of production machines. In addition to the compromised Github account, the test setup also included a Docker Hub account, a developer machine, and a production machine, each of which was considered non-compromised [61]. The purpose of the attack was to use the Docker ecosystem to execute arbitrary code in a container in the production environment. The compromise of the Github account could have succeeded, for example, through a phishing attack.

The target was reached in an impressive 5 minutes and 30 seconds after committing in the Github repository [61]. Although the takeover of the Github account is independent of Docker, the attempt shows that the automatic execution of the damaged images within minutes is a big challenge. The same consequences would result in a takeover of the Docker Hub account. For this reason, account hijacking should be regarded as an increasing danger, especially as the number of accounts with different providers is increasing due to test services, for

example.

A similar problem is posed by poorly configured container orchestration tools. In July 2018, 22,000 management interfaces of tools like Kubernetes, Docker Swarm or Consul were found, which were publicly accessible via the Internet [47]. More than 300 of them were protected without a password or were still in the setup stage. Attackers were able to access the complete container management and execute arbitrary code, for example.

In addition to code injection, Denial of Service attacks are also possible [11]. For example, if an attacker takes over test machines, intentionally erroneous tests can be created. Depending on the configuration, this can also affect the production machines if the execution of specific images in containers is linked to functioning tests.

4.2.7.3 Risk

The probability of a vulnerable node in the pipeline increases with the number and complexity of the overall system. For example, if multiple production machines or containers running images from the same Github repository are included, then multiple containers are also affected by an attack. Not to forget the pressure of innovation, as increasingly shorter development and deployment cycles are demanded. Careful development and high coverage of the tests should, therefore, be in the foreground. It should be mentioned that Docker Content Trust is not compatible in a complex architecture as shown in Figure 4.2 [67]. Code from external entities is processed, which is not possible with Content Trust. With Content Trust, only one instance, namely the developer (person or organization), needs to be trusted. Thus Docker Content Trust can be considered as a possible single point of failure. In a complex multi-actor architecture, each entity must be trusted, each capable of compromising images. This does not create any security improvements compared to a single point of failure but instead increases the attack surface with equally weak nodes.

Overall, automatic pipelines must be regarded as critical. The brief attack time of a few minutes indicates that an immediate detection of developers or administrators is almost impossible. The complexity of such an architecture also makes it difficult to identify the cause of the problem.

5

Recommendations for Security Improvements

This chapter evaluates procedures to improve protection for the attack scenarios set out in chapter 4. A list of over 90 individual measures to ensure the safety of Docker can be found in the CIS report. However, these are not adapted to the respective points of attack as in this thesis and also do not include some of the components shown in section 4.2. For example, the CIS report does not cover hardware security or the integration of Docker in a deployment pipeline.

Generally, Docker default settings offer a fair trade-off between usability and security. Nevertheless, individual components can be strengthened explicitly against attacks, whereby certain error classes from section 4.2 can be avoided entirely. For example, the use of Docker Content Trust avoids man-in-the-middle attacks. Some error classes cannot be circumvented, but at least they can be mitigated. Therefore, the suggestions made in this chapter to improve the security should be taken into account when Docker is used in a production environment.

5.1 CIS Docker Benchmark

The Center for Internet Security (CIS) is an association of organizations and individuals to provide materials and resources on internet security. Particular attention is paid to the so-called benchmarks which are thematic compilations of instructions and tips for securing specific computer systems against threats [14]. For several years now, a Docker Security Benchmark has also been published regularly, which offers instructions for setting up a secure configuration for Docker Community Edition. The latest report refers to Docker CE 17.06 [15]. A benchmark for Docker Enterprise Edition has been announced for 2018 but has not yet been published [43].

The Community Edition report is divided into seven sections, two of which deal with the Docker daemon. Each chapter contains specific guidelines for action and provides an assessment. Accordingly, a rating can be either "Scored" or "Not Scored". "Scored" recommendations increased the benchmark score while "Not Scored" recommendations did not increase the resulting benchmark score, which does not mean that a measure evaluated as "Not Scored" reduces the benchmark score [15].

On Github, *Docker Inc.* provides a tool to check Docker best-practices for deploying containers in production [15]. The CIS benchmark inspired the tests which enable to check hosts and Docker containers against the benchmark.

In the following sections, this work will be oriented at some points in the evaluation of CIS, since CIS is regarded as a recognized organization and a reliable source. However, the benchmark does not cover all the attack scenarios in section 4.2. Therefore, other sources will be used to prove the conclusions.

5.2 Protection against Docker Security Threats

The attack scenarios found in section 4.2 can be prevented or at least made more difficult by some countermeasures. As in all security-critical applications, 100% security can never be guaranteed, despite measures to strengthen it. However, according to the Defense in Depth principle presented in section 3.1, the overall security of a system can be increased by combining several security concepts. The ones shown in the next sections are also not strictly distinct to each other. For example, hardening of the host can also improve the security of the Docker engine. The following sections are structured using the same bottom-up approach as in chapter 4.2. For instance, section 4.2.5 can be interpreted as a reaction to the problems found in section 5.2.5.

5.2.1 Hardware

As a user, there are few opportunities to protect against hardware vulnerabilities. Instead, one is dependent on the manufacturers of processors until they develop new reliable models. Intel, for instance, has announced its intention to release Cascade Lake processors in the second half of 2018 which are said not to be vulnerable to Meltdown and Spectre variant 2 [7]. Since Intel did not comment on other Spectre variants, such as the ones published in May and July 2018 [46, 88], it must be waited for security analyses at least until the release of the next processor generation. However, compared to traditional virtual machines, a container-based virtualization solution can be updated more easily. If a hardware security flaw can be fixed with a software update, such as Meltdown, only the kernel of the host needs to be updated. Virtual machines have each an individual kernel which needs to be updated. Overall, the complete approach of multi-tenancy architectures may need to be reconsidered. Although dedicated hardware for each application would eliminate the benefits of virtualization and containerization, the attack surface would be significantly reduced.

5.2.2 Linux Kernel

Despite the vast number of vulnerabilities in the Linux kernel shown in section 4.2.2, it is an essential software product used in the industry. For this reason,

many volunteers and corporations are continually working on maintaining the kernel. Since around 57% of changes to the kernel come from various companies, it can be concluded that it is also in the interest of corporations to maintain security in the kernel [35]. Therefore, it is recommended to always provide the Linux kernel and the host operating system with security patches. However, it must always be checked whether updates restrict the functionality of other software applications running on the host. The tradeoff between security and more functionality must then be decided in each individual case.

5.2.3 Docker Host

In addition to the kernel, Docker and other software on the host must also be kept up to date. Docker regularly releases patches to fix security bugs. With newer Docker versions, the exploitation of known vulnerabilities can be avoided [15]. At this point, too, third-party vendors may be affected by updates, so patches should always be evaluated first.

The Docker daemon runs on the host OS with root privileges [67]. Its actions should, therefore, be audited. In addition to the Docker daemon, auditing should also be enabled for all critical files, directories and components, such as `/etc/docker` or `/usr/bin/docker-runc`. A complete list can be found in the CIS benchmark [15]. The Linux tool *auditd* can be used for auditing. It creates large log files, which give information about possible misuse of Docker. Since these log files take up a lot of storage space quickly, they should be regularly analyzed and archived to detect any longer-term trends in the system.

Depending on the host operating system, Mandatory Access Control (MAC) mechanisms are used in addition to the standard DAC features of the kernel. MACs offer additional security through an identity independent set of rules. Several security mechanisms can thus be implemented consecutively in order to implement Defense In Depth, which was explained in section 3.1. One solution that implements MAC is AppArmor, which is used by default in OpenSuse, Debian, and Ubuntu, for example [5]. Application-specific rules are defined on the basis of text-based profiles that define access to objects via paths in the file system and various combinable authorization types. For example, AppArmor allows a profile for a specific container to specify individually that it may not access a specific file, or that it may access it but only with read permissions. Thus, AppArmor enforces security policies that go beyond DAC. A similar MAC solu-

tion is SELinux, which is considered more challenging to configure [98]. Overall, it should be considered that MAC mechanisms add additional layers of host operating system security. Therefore, when choosing the host OS, it is advised to consider the integration of such a system.

Docker uses namespaces to implement the isolation between containers and with the host system. This isolation should not be compromised by improper configuration of the production environment, for example by sharing the host's various namespaces with containers. Sharing the network namespace would allow the container to access network services such as the D-bus on the Docker host, allowing a container process to switch off the Docker host, for example. Sharing other namespaces would have similarly drastic consequences, which are not discussed here in detail, but can be found in the CIS report [15]. It should only be noted that sharing the namespaces of the host namespaces with the containers in the critical production machines almost always contributes to the deterioration of security.

Docker containers receive by default a reduced set of root privileges in the form of Linux kernel capabilities. However, a container may be granted additional capabilities as well as specific capabilities can be removed. With fewer privileges of a root user, an attacker has fewer abilities that he could exploit to his advantage. Therefore, it is highly recommended to remove all capabilities from a container except those that are explicitly required [15]. In particular, the option `-privileged`, which provides a container with all capabilities, should not be used.

A secure Docker production environment should also guarantee the use of *seccomp*. Seccomp is a tool to filter incoming system calls using a whitelist, limiting the kernel surface exposed to applications [31]. With less attack surface, an adversary can use fewer kernel functionalities, which contributes to an overall improvement in security.

In section 4.2.3, it was shown that mounting the Docker daemon in the container leads to a deterioration of security. The same applies to sensitive host directories. Therefore, this should be avoided and at the very most only be carried out with reading permissions.

5.2.4 Docker Engine

By default, the Docker daemons binds to a non-networked Unix socket, `unix:///var/run/docker.sock`, on the host on which it is running [103]. However, the daemon runs with root privileges to have the necessary access to the corresponding resources. If the Docker daemon were bound to a TCP port or another Unix socket, anyone with access to that port or socket would have full access to the Docker daemon [15]. However, if the daemon has to be accessible via a network (e.g., for Docker Swarm), such a step is necessary. Therefore it must be ensured that only clients with access rights can communicate with the daemon over the network. For this purpose, a TLS authentication can be configured for Docker, which limits the connections to the Docker daemon to a restricted number of clients. Then, only clients that have authorized certificates deposited at the daemon are allowed to interact with it via TLS.

As shown above, any user with access rights to the Docker daemon can run any Docker client command. The same applies to callers using Docker's Remote API to interact with the daemon. If access permissions for Docker client commands should be more granular, an authorization plugin can be used. The container orchestration tool Kubernetes, for example, has such a function integrated [15]. Each incoming docker command is therefore first checked by the daemon to see if the sender has the necessary permissions.

Docker containers of a host can exchange information using the default network bridge which means that each container potentially listens for all packets between the container network on the same host. The security goal of confidentiality could be threatened. However, networks can be easily adapted with Docker. It is recommended that networks should be configured so that only containers that need to communicate with each other are on a network [67]. With the command `dockerd -icc=false` the inter-container communication is suspended. Only containers that are explicitly included in a network can then communicate.

By default, the usage of user namespaces in Docker is optional. As explained in section 3.1.1, user namespaces allow containers to provide a unique range of user and group IDs that are outside the traditional user and group range used by the host system. This is useful if containers do not have an explicit container user in the container image. With the user namespaces, the root user has administrator privileges within the container but is effectively mapped to an unprivileged UID on the host system. This would significantly increase the

security of a running container, but it would reduce other docker functionalities [79]. Therefore, the application of user namespaces must be individually evaluated according to purpose and scenario.

Otherwise, the CIS report shows that most of the other settings are secure by default. For example, the ownership and permissions of most configuration files or directories like `/etc/docker` are very strictly assigned. Moreover, so-called experimental features are also disabled and a non-obsolete storage driver is used [15].

5.2.5 Docker Images

In the last section, the challenges of the user namespace were briefly addressed. Though, to run a container as a non-root user, the `USER` instruction can be utilized in the Dockerfile. User namespace remapping is then not necessary. If an attacker succeeds in breaking out, they are only non-root users on the host system, which significantly reduces the potential for damage [67]. For this reason, the Dockerfile should always contain a `USER` statement, provided that root privileges are not required in the container. Additionally, the Dockerfile instruction HEALTHCHECK is useful because it triggers the Docker engine to check the running container periodically [15]. If a container has changed to an undesired status, it can be terminated, and a new container will be started.

Using minimal images reduces the attack surface. Smaller base images like Alpine Linux (2-3MB compared to 49MB of Debian) do not only save storage capacity. They also offer fewer tools and paths to escalate privileges. Furthermore, removing the package installer as a last step in the Dockerfile for production image further impedes adversaries.

It is often recommended that only official images from the Docker Hub should serve as a basis for own images and containers, primarily because they are subject to security scanning as well as unofficial images often do not receive updates for months [91]. Generally, one can agree to this, but studies proved that even official images sometimes contain serious security flaws [38, 91]. Therefore, the use of tools such as Anchore or Clair is recommended, which perform a similar security scan as Docker Hub and images against CVE errors. They also allow scanning of unofficial images. Overall, it must be noted that the high density of vulnerabilities in both official and unofficial images indicates a

fundamental problem of software development. Despite security teams, some technologies are prone to vulnerabilities, so their use should be reconsidered in the first place. For example, each image in the official repository of the content management system WordPress currently contains several vulnerabilities classified as very severe. Therefore, the use of such a product should be reconsidered in the first place.

5.2.6 Image Distribution

Man-in-the-Middle attacks between the Docker engine and a registry can be prevented by using Docker Content Trust because it provides data to and from the registry with digital signatures. On the client side, the publisher can be verified, and the integrity of the data can be checked. However, integration in build and deployment pipelines is no longer possible [67]. Without Content Trust, Docker Hub could automatically build an Image after pushing into a git repository. Docker Hub lacks the keys needed to sign the image. Consequently, a production machine with activated content trust would not trust the image. Finally, the use of Content Trust is to be evaluated as a well-functioning security solution, which, however, is accompanied by less flexibility in the automation of development processes.

5.2.7 Deployment Pipelines

Section 4.2.7 showed an attack wherein an automated deployment pipeline, a push from a compromised Github account resulted in a malicious image executing on a production machine within minutes. Although account takeovers can never be prevented entirely, they can be significantly reduced by a second factor during authentication, e.g., an app (2FA) or physical security keys (U2F) can be used. Google, for instance, has not had a single successful phishing attack in over a year and a half by using U2F [49]. Besides, fewer automated deployment pipelines can also contribute to increased security. For example, if a second person always has to confirm the deployment, taking over a Github account cannot trigger such an attack.

6

Conclusion

The security of virtualization and containerization technologies is essential for its acceptance in the industry. As shown in chapter 3, Docker uses the Principle of Least Privilege and Defense in Depth principles to implement the security goals of section 2.3 with a multi-layered security architecture. Nevertheless, security threats are still possible for several reasons. These range from design and implementation errors in the hardware and Linux kernel to configuration errors in a fully automated deployment pipeline.

Docker's built-in security mechanisms allow the isolation of different containers and the host operating system using Linux namespaces. However, by default, the user namespace used to map a root user within a container to a non-privileged user is not used. Such a measure would significantly increase security, as an outbreak of a container could cause drastically less damage.

The Linux capabilities for restricting access rights and the control groups for ensuring the goal of availability are further Linux kernel functionalities that contribute to the protection of containers. However, also in the now vast Docker ecosystem, there are some mechanisms to make the use of Docker safe. This includes first and foremost the use of Docker Security Scanning. However, this feature is only available for official repositories of the Docker Hub and for paying customers of the Docker Enterprise Edition. This decision by *Docker Inc.* shows that for economic reasons the focus in future development will be on the paid version. Further indications are, among others, that with the open container format OCF an industry standard has been initiated and implemented, which can also be taken up by other projects. This shows that *Docker Inc.* does not want to depend on the innovative potential of container technology so that the focus will be much more on orchestration tools and additional services, like Docker Swarm and the Docker Enterprise Edition.

The security analysis from chapter 4 was carried out on several levels and followed a bottom-up approach. With the help of STRIDE, a recognized method was used to find threats. Several potential security threats were found in each of the layers. Besides, attack scenarios were identified for the respectively found threats. From the user's point of view, the configuration of the Docker host, the selection, and distribution of images and the integration of automated deployment chains are particularly important. The basic Docker configuration offers a fair trade-off between security and usability, but the security can be fundamentally violated with custom settings. Images are especially a gateway for attackers, as they are shared with other parties via partially public registries. The integration of Docker into a deployment pipeline that restarts a container within minutes of a change in the code repository is also very critical. Only the compromise of one of the services involved in the pipeline endangers the security of the overall system. Therefore, it should always be considered whether the integration of Docker Content Trust to increase security should not be valued higher than the productivity increases of an automated pipeline.

The recommendations in chapter 5 offer measures to increase the security of a docker infrastructure and are to be understood as countermeasures against the previously identified hazards. Although the measures in the CIS benchmark cover a wide range, even more layers were considered in this thesis, including the integration of Docker into a deployment pipeline.

An interesting further study could be the analysis of Docker competitors such as *rkt* from CoreOS as well as a comparison with other containerization solutions. *rkt* in particular was developed as a project with a strong security focus, which is why the measures used there to guarantee security are indeed worth a closer look. In addition to traditional virtualization and containerization products, so-called Unikernels have recently also been further researched. Although they are not yet widely used in the industry, their concept is fascinating, especially in the background of the problems of using a shared kernel covered in this thesis. When using Unikernels, the isolation is guaranteed by non-shared minimal kernels. They achieve almost the same or even better performance than containers and, moreover, do not have to struggle with the security threats that result from the use of a shared kernel. Recently it was shown that Unikernels could become a serious alternative to containers [55, 54].

In conclusion, Docker reduces much complexity from the developer's point of view. Once Docker is installed on the machine, Dockerfiles enable relatively easy packaging of an application into an image that can then be efficiently and

reproducibly ported to other machines. However, from the system's point of view, some complexity has been added. In contrast to a bare-metal approach, an application depends on specific Linux kernel functionalities and is also exposed to some dangers in the Docker ecosystem. Overall, therefore, the use of Docker is only recommended after thorough consideration. Besides, an automatic deployment pipeline should only be used in a production environment with appropriate logging and auditing measures, or automatic chains should be averted entirely. Overall, critical docker infrastructures should only be used with additional measures to increase security.

Bibliography

[1] *[PATCH] namespaces*. [Git commit 071df104f808b8195c40643dcb4d060681742e29]. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=071df104f808b8195c40643dcb4d060681742e29/.

[2] *About CVE*. [Online; accessed June 02, 2018]. URL: https://cve.mitre.org/about/.

[3] Amit Singh. *Reasons to Use Virtualization*. [Online; accessed June 30, 2018]. URL: http://www.kernelthread.com/publications/virtualization/.

[4] Andrea Arcangeli. "Linux as a Hypervisor". In: *linuxplanet* (2008). [Online; accessed July 1, 2018]. URL: http://www.linuxplanet.com/linuxplanet/reports/6503/3/.

[5] Mick Bauer. "Paranoid penguin: an introduction to Novell AppArmor". In: *Linux Journal* 2006.148 (2006), p. 13.

[6] Matt Bishop. *Computer security: art and science*. Addison-Wesley Professional, 2003.

[7] Brian Krzanich. *Advancing Security at the Silicon Level*. [Online; accessed June 04, 2018]. URL: https://newsroom.intel.com/editorials/advancing-security-silicon-level/.

[8] Bruce Schneier. *Security in the Cloud*. [Online; accessed June 17, 2018]. URL: https://www.schneier.com/blog/archives/2006/02/security_in_the.html.

[9] Thanh Bui. "Analysis of docker security". In: *arXiv preprint arXiv:1501.02967* (2015).

[10] Justin Cappos et al. "A look in the mirror: Attacks on package managers". In: *Proceedings of the 15th ACM conference on Computer and communications security*. ACM. 2008, pp. 565–574.

[11] Jeeva Chelladhurai, Pethuru Raj Chelliah, and Sathish Alampalayam Kumar. "Securing docker containers from denial of service (dos) attacks". In: *Services Computing (SCC), 2016 IEEE International Conference on*. IEEE. 2016, pp. 856–859.

[12] Haogang Chen et al. "Linux kernel vulnerabilities: State-of-the-art defenses and open problems". In: *Proceedings of the Second Asia-Pacific Workshop on Systems*. ACM. 2011, p. 5.

[13] Yehuda Chikvashvili. *Cryptocurrency Miners Abusing Containers: Anatomy of an (Attempted) Attack*. [Online; accessed June 25, 2018]. URL: https://blog.aquasec.com/cryptocurrency-miners-abusing-containers-anatomy-of-an-attempted-attack.

[14] CIS. *About us*. [Online; accessed June 19, 2018]. URL: https://www.cisecurity.org/about-us/.

[15] Docker Inc. CIS. "CIS Docker CE 17.06 Benchmark". In: *Center for Internet Security.< https://benchmarks. cisecurity. org/tools2/docker/CIS_Docker_1* 11 (2017).

[16] Microsoft Corporation. *Windows Containers*. [Online; accessed June 07, 2018]. URL: https://docs.microsoft.com/en-us/virtualization/windowscontainers/about/.

[17] CVE Details. *Docker: Vulnerability Statistics*. [Online; accessed June 19, 2018]. URL: https://www.cvedetails.com/product/28125/Docker-Docker.html.

[18] CVE Details. *Linux Kernel: Vulnerability Statistics*. [Online; accessed July 22, 2018]. URL: https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33.

[19] Docker Inc. *AppArmor*. [Online; accessed June 22, 2018]. URL: https://github.com/docker/labs/tree/master/security/apparmor.

[20] Docker Inc. *Automated builds*. [Online; accessed July 12, 2018]. URL: https://docs.docker.com/docker-cloud/builds/automated-build/.

[21] Docker Inc. *Company*. [Online; accessed July 19, 2018]. URL: https://www.docker.com/company/.

[22] Docker Inc. *Content trust in Docker*. [Online; accessed June 12, 2018]. URL: https://docs.docker.com/engine/security/trust/content_trust/.

[23] Docker Inc. *Docker CE release notes*. [Online; accessed July 16, 2018]. URL: https://docs.docker.com/release-notes/docker-ce/.

[24] Docker Inc. *Docker Datacenter Enables DevOps*. [Online; accessed June 04, 2018]. URL: https://www.docker.com/use-cases/devops.

[25] Docker Inc. *Docker security non-events*. [Online; accessed June 23, 2018]. URL: https://docs.docker.com/engine/security/non-events/.

[26] Docker Inc. *Docker Security Scanning*. [Online; accessed June 13, 2018]. URL: https://docs.docker.com/v17.12/docker-cloud/builds/image-scan/.

[27] Docker Inc. *Docs: Docker security*. [Online; accessed July 19, 2018]. URL: https://docs.docker.com/engine/security/security/.

[28] Docker Inc. *GitHub Issue: support cgroup v2 (unified hierarchy)*. [Online; accessed July 22, 2018]. URL: https://github.com/opencontainers/runc/issues/654.

[29] Docker Inc. *Image Manifest V 2, Schema 2*. [Online; accessed June 09, 2018]. URL: https://docs.docker.com/registry/spec/manifest-v2-2/.

[30] Docker Inc. *Scan images for vulnerabilities*. [Online; accessed June 13, 2018]. URL: https://docs.docker.com/ee/dtr/user/manage-images/scan-images-for-vulnerabilities/#the-docker-security-scan-process.

[31] Docker Inc. *Seccomp security profiles for Docker*. [Online; accessed July 18, 2018]. URL: https://docs.docker.com/engine/security/seccomp/.

[32] Docker Inc. *Swarm mode overview*. [Online; accessed July 19, 2018]. URL: https://docs.docker.com/engine/swarm/.

[33] Michael Eder. "Hypervisor-vs. container-based virtualization". In: *Future Internet (FI) and Innovative Internet Technologies and Mobile Communications (IITM)* 1 (2016).

[34] Wes Felter et al. "An updated performance comparison of virtual machines and linux containers". In: *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*. IEEE. 2015, pp. 171–172.

[35] The Linux foundation. *The Top 10 Developers and Companies Contributing to the Linux Kernel in 2015-2016*. [Online; accessed July 13, 2018]. URL: https://www.linuxfoundation.org/blog/2016/08/the-top-10-developers-and-companies-contributing-to-the-linux-kernel-in-2015-2016/.

[36] Michael Friis. *Build and run your first Docker Windows Server container*. [Online; accessed June 16, 2018]. Docker Inc., 2016. URL: https://blog.docker.com/2016/09/build-your-first-docker-windows-server-container/.

[37] Charles David Graziano. "A performance analysis of Xen and KVM hypervisors for hosting the Xen Worlds Project". In: (2011).

[38] Jayanth Gummaraju, Tarun Desikan, and Yoshio Turner. "Over 30% of official images in docker hub contain high priority security vulnerabilities". In: *Technical Report*. BanyanOps, 2015.

[39] Hacker News. *Docker 1.10.0 is out*. [Online; accessed June 25, 2018]. URL: https://news.ycombinator.com/item?id=11037543.

[40] Hacker News. *The Security-Minded Container Engine by CoreOS: rkt Hits 1.0*. [Online; accessed June 25, 2018]. URL: https://news.ycombinator.com/item?id=11035955.

[41] Major Hayden and Richard Carbone. "Securing linux containers". In: *GIAC (GCUX) Gold Certification, Creative Commons Attribution-ShareAlike 4.0 International License* 19 (2015).

[42] IBM. *About RHEL/AIX Access Control (DAC, MAC and RBAC)*. [Online; accessed July 13, 2018]. URL: https://www.ibm.com/developerworks/community/blogs/mhhaque/entry/About_RHEL_AIX_Access_Control_DAC_MAC_RBAC.

[43] Docker Inc. *CIS Docker EE benchmark*. [Online; accessed June 19, 2018]. URL: https://docs.docker.com/compliance/cis/docker_ee/.

[44] Jake Edge. *Control group namespaces*. [Online; accessed July 1, 2018]. URL: https://lwn.net/Articles/621006/.

[45] James Morris. *Overview of Linux Kernel Security Features*. [Online; accessed July 13, 2018]. URL: https://www.linux.com/learn/overview-linux-kernel-security-features.

[46] Jürgen Schmidt. *Super-GAU für Intel: Weitere Spectre-Lücken im Anflug*. [Online; accessed May 13, 2018]. URL: https://www.heise.de/ct/artikel/Super-GAU-fuer-Intel-Weitere-Spectre-Luecken-im-Anflug-4039134.html.

[47] Kristian Kißling. *Mehr als 22.000 Containerverwaltungen öffentlich zugänglich*. [Online; accessed July 19, 2018]. URL: https://www.golem.de/news/lacework-mehr-als-22-000-containerverwaltungen-oeffentlich-zugaenglich-1807-135285.html.

[48] Paul Kocher et al. "Spectre attacks: Exploiting speculative execution". In: *arXiv preprint arXiv:1801.01203* (2018).

[49] Brian. Krebs. *Google: Security Keys Neutralized Employee Phishing*. [Online; accessed July 22, 2018]. URL: https://krebsonsecurity.com/2018/07/google-security-keys-neutralized-employee-phishing/.

[50] Kromtech. *Cryptojacking invades cloud. How modern containerization trend is exploited by attackers*. [Online; accessed June 22, 2018]. URL: https://kromtech.com/blog/security-center/cryptojacking-invades-cloud-how-modern-containerization-trend-is-exploited-by-attackers.

[51] Linus Torvalds. *Linux kernel mailing list: cgroup namespace support for v4.6-rc1*. [Online; accessed June 24, 2018]. URL: https://lkml.org/lkml/2016/3/26/132.

[52] Moritz Lipp et al. "Meltdown". In: *arXiv preprint arXiv:1801.01207* (2018).

[53] Tao Lu and Jie Chen. "Research of Penetration Testing Technology in Docker Environment". In: (2017).

[54] Anil Madhavapeddy et al. "Jitsu: Just-In-Time Summoning of Unikernels." In: *NSDI*. 2015, pp. 559–573.

[55] Anil Madhavapeddy et al. "Unikernels: Library operating systems for the cloud". In: *Acm Sigplan Notices*. Vol. 48. 4. ACM. 2013, pp. 461–472.

[56] man7.org. *capabilities - overview of Linux capabilities*. [Online; accessed July 02, 2018]. URL: http://man7.org/linux/man-pages/man7/capabilities.7.html.

[57] man7.org. *cgroups - Linux control groups*. [Online; accessed June 30, 2018]. URL: http://man7.org/linux/man-pages/man7/cgroups.7.html.

[58] man7.org. *namespaces - overview of Linux namespaces*. [Online; accessed June 25, 2018]. URL: http://man7.org/linux/man-pages/man7/namespaces.7.html.

[59] Margaret Rouse. *Desktop Virtualization*. [Online; accessed June 30, 2018]. URL: https://searchvirtualdesktop.techtarget.com/definition/desktop-virtualization.

[60] Martin Fischer. *Spectre und Meltdown: Intel-Prozessoren mit vollem Hardwareschutz bereits 2018*. [Online; accessed June 04, 2018]. URL: https://www.heise.de/security/meldung/Spectre-und-Meltdown-Intel-Prozessoren-mit-vollem-Hardwareschutz-bereits-2018-3995993.html.

[61] A Martin et al. "Docker ecosystem–Vulnerability Analysis". In: *Computer Communications* 122 (2018), pp. 30–43.

[62] Matteo Riondato. *FreeBSD Handbook - Jails*. [Online; accessed July 1, 2018]. URL: https://www.freebsd.org/doc/handbook/jails.html.

[63] Michael Bacarella. *Taking Advantage of Linux Capabilities*. [Online; accessed June 13, 2018]. URL: https://www.linuxjournal.com/article/5737.

[64] Michael Kerrisk. *Namespaces in operation, part 1: namespaces overview*. [Online; accessed June 25, 2018]. URL: https://lwn.net/Articles/531114/.

[65] Michael Kerrisk. *Namespaces in operation, part 3: PID namespaces*. [Online; accessed June 25, 2018]. URL: https://lwn.net/Articles/531419/.

[66] Michael Larabel. *The Linux Kernel Gained 2.5 Million Lines Of Code, 71k Commits In 2017*. [Online; accessed May 29, 2018]. URL: https://phoronix.com/misc/linux-2017/index.html.

[67] Ian Miell and Aidan Hobson Sayers. *Docker in practice*. [Second Edition, Manning Early Access Program, Version 6]. Manning Publications Co., 2018.

[68] Mike Coleman. *Containers are not VMs*. [Online; accessed June 11, 2018]. URL: https://blog.docker.com/2016/03/containers-are-not-vms/.

[69] Moby project. *default_template_linux.go*. [Online; accessed June 17, 2018]. URL: https://github.com/moby/moby/blob/2210b15e08b8a81bb7c22b34036a5d0336fdaemon/execdriver/native/template/default_template_linux.go.

[70] Diogo Mónica. *Least Privilege Container Orchestration*. [Online; accessed June 18, 2018]. Docker Inc., 2017. URL: https://blog.docker.com/2017/10/least-privilege-container-orchestration/.

[71] A Mouat. *Docker Security - Using Containers Safely in Production*. 2015.

[72] Suvda Myagmar, Adam J Lee, and William Yurcik. "Threat modeling as a basis for security requirements". In: *Symposium on requirements engineering for information security (SREIS)*. Vol. 2005. Citeseer. 2005, pp. 1–8.

[73] Open Container Initiative. *About*. [Online; accessed July 02, 2018]. URL: https://www.opencontainers.org/about.

[74] Oracle Corporation. *System Administration Guide: Oracle Solaris Containers-Resource Management and Oracle Solaris Zones*. [Online; accessed July 1, 2018]. URL: https://docs.oracle.com/cd/E19044-01/sol.containers/817-1592/zones.intro-1/.

[75] Oracle Oracle Corporation. *Reasons to Use Virtualization*. [Online; accessed June 30, 2018]. URL: https://docs.oracle.com/cd/E35328_01/E35332/html/vmusg-virtualization-reasons.html.

[76] Sylvia Osborn, Ravi Sandhu, and Qamar Munawer. "Configuring role-based access control to enforce mandatory and discretionary access control policies". In: *ACM Transactions on Information and System Security (TISSEC)* 3.2 (2000), pp. 85–106.

[77] Phil Estes. *Rooting out Root: User namespaces in Docker*. [Online; accessed July 16, 2018]. URL: https://events.static.linuxfound.org/sites/events/files/slides/User%20Namespaces%20-%20ContainerCon%202015%20-%2016-9-final_0.pdf.

[78] Gerald J Popek and Robert P Goldberg. "Formal requirements for virtualizable third generation architectures". In: *Communications of the ACM* 17.7 (1974), pp. 412–421.

[79] Nigel Poulton. *Docker Deep Dive*. [Version from February 11, 2018]. Leanpup, 2016.

[80] Andrew Prout et al. "Measuring the Impact of Spectre and Meltdown". In: *arXiv preprint arXiv:1807.08703* (2018).

[81] raesene. *The Dangers of Docker.sock*. [Online; accessed June 13, 2018]. URL: https://raesene.github.io/blog/2016/03/06/The-Dangers-Of-Docker.sock/.

[82] Elena Reshetova et al. "Security of OS-level virtualization technologies". In: *Nordic Conference on Secure IT Systems*. Springer. 2014, pp. 77–93.

[83] Jonathan Rudenberg. *Docker Image Insecurity*. 2014.

[84] Rui Xiang. *Add namespace support for syslog*. [Online; accessed June 24, 2018]. URL: https://lwn.net/Articles/562389/.

[85] Jerome H Saltzer and Michael D Schroeder. "The protection of information in computer systems". In: *Proceedings of the IEEE* 63.9 (1975), pp. 1278–1308.

[86] Bruce Schneier. "Attack trees". In: *Dr. Dobb's journal* 24.12 (1999), pp. 21–29.

[87] Michael Schwarz et al. "Malware guard extension: Using SGX to conceal cache attacks". In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. 2017, pp. 3–24.

[88] Michael Schwarz et al. "NetSpectre: Read Arbitrary Memory over Network". In: (2018).

[89] Chithra Selvaraj and Sheila Anand. "A survey on security issues of reputation management systems for peer-to-peer networks". In: *Computer Science Review* 6.4 (2012), pp. 145–160.

[90] Adam Shostack. *Threat modeling: Designing for security*. John Wiley & Sons, 2014.

[91] 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*. ACM. 2017, pp. 269–280.

[92] Jim Smith and Ravi Nair. *Virtual machines: versatile platforms for systems and processes*. Elsevier, 2005.

[93] *Softlayer Benchmark, Data Sheet*. [Online; accessed July 1, 2018]. URL: https://voltdb.com/sites/default/files/voltdb_softlayer_benchmark_0.pdf.

[94] Software Engineering Daily. *Container Security*. [Online; accessed July 19, 2018; podcast transcript]. URL: https://softwareengineeringdaily.com/wp-content/uploads/2018/05/SED592-Container-Security.pdf.

[95] Solomon Hykes. *Docker 0.9: Introducing execution drivers and libcontainer*. [Online; accessed July 10, 2018]. URL: https://blog.docker.com/2014/03/docker-0-9-introducing-execution-drivers-and-libcontainer/.

[96] Solomon Hykes. *Introducing Moby Project: a new open-source project to advance the software containerization movement*. [Online; accessed July 05, 2018]. URL: https://blog.docker.com/2017/04/introducing-the-moby-project/.

[97] Stephen Soltesz et al. "Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors". In: *ACM SIGOPS Operating Systems Review*. Vol. 41. 3. ACM. 2007, pp. 275–287.

[98] Ralf Spenneberg. *SELinux & AppArmor: Mandatory Access Control für Linux einsetzen und verwalten;[CD: Fedora Core 6-VMware-Image zum experimentellen Einsatz von SELinux]*. Pearson Deutschland GmbH, 2008.

[99] Stephen Day. *A new model for Docker image distribution*. [Online; accessed June 09, 2018]. URL: https://www.slideshare.net/Docker/docker-48351569.

[100] systutorials. *namespaces: overview of Linux namespaces*. [Online; accessed June 24, 2018]. URL: https://www.systutorials.com/docs/linux/man/7-namespaces/.

[101] Andrew S Tanenbaum. *Modern operating system*. Pearson Education, Inc, 2009.

[102] Toli Kuznets. *Docker Security Scanning safeguards the container content lifecycle*. [Online; accessed July 02, 2018]. URL: https://blog.docker.com/2016/05/docker-security-scanning/.

[103] James Turnbull. *The Docker Book*. [Version: v17.12.0-ce (ec75703), February 25, 2018]. Lulu. com, 2014.

[104] Nik Unger et al. "SoK: secure messaging". In: *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE. 2015, pp. 232–249.

[105] UpGuard. *Docker vs CoreOS Rkt*. [Online; accessed July 12, 2018]. URL: https://www.upguard.com/articles/docker-vs-coreos.

[106] VMware Inc. *Virtualization*. [Online; accessed June 29, 2018]. URL: https://www.vmware.com/solutions/virtualization.html.

[107] Brian Ward. *How Linux works: What every superuser should know*. No Starch Press, 2014.

[108] Miguel G Xavier et al. "Performance evaluation of container-based virtualization for high performance computing environments". In: *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*. IEEE. 2013, pp. 233–240.

[109] Ying Li. *Content trust in Docker*. [Online; accessed June 17, 2018]. URL: https://blog.docker.com/2017/02/docker-secrets-management/.

List of Figures