Building a Secure Software Supply Chain using Docker

Master's Thesis of

Simon Lipke

at Stuttgart Media University
and ERNW Enno Rey Netzwerke GmbH
in Heidelberg

August 31, 2017

Reviewer:	Prof. Dr. Dirk Heuzeroth
Second reviewer:	M. Sc. Matthias Luft (ERNW)
Matr. Nr.:	30799
Course of Studies:	Computer Science and Media (CSM)
Academic Degree:	Master of Science

Abstract (EN)

Nowadays more and more companies use agile software development to build software in short release cycles. Monolithic applications are split into microservices, which can independently be maintained and deployed by agile teams. Modern platforms like Docker support this process. Docker offers services to containerize such services and orchestrate them in a container cluster. A software supply chain is the umbrella term for the process of developing, automated building and testing, as well as deploying a complete application. By combining a software supply chain and Docker, those processes can be automated in standardized environments. Since Docker is a young technology and software supply chains are critical processes in organizations, security needs to be reviewed. In this work a software supply chain based on Docker is built and a threat modeling process is used to assess its security. The main components are modeled and threats are identified using STRIDE. Afterwards risks are calculated and methods to secure the software supply chain based on security objectives confidentiality, integrity and availability are discussed. As a result, some components require special treatments in security context since they have a high residual risk of being targeted by an attacker. This work can be used as basis to build and secure the main components of a software supply chain. However additional components such as logging, monitoring as well as integration into existing business processes need to be reviewed.

Abstract (DE)

Heutzutage nutzen mehr und mehr Firmen agile Softwareentwicklung, um Software in kurzen Release-Zyklen zu entwickeln. Monotlithische Anwendungen werden in Microservices aufgeteilt, welche unabhängig voneinander erstellt und veröffentlicht werden können. Moderne Plattformen wie Docker unterstützen diesen Prozess. Docker bietet Dienste an, um solche Anwendungen in Container zu verpacken und sie auf Container Clustern zu orchestrieren. Eine Software Supply Chain ist der Überbegriff für den Prozess der Herstellung, des automatisierten Bauens und Testens, sowie der Veröffentlichung von Software. Durch die Kombination aus Software Supply Chains und Docker können diese Prozesse in standardisierten Umgebungen automatisiert werden. Da Docker eine junge Technologie ist und Software Supply Chains einen kritischen Prozess im Unternehmen darstellen, muss zunächst die Sicherheit überprüft werden. In dieser Arbeit wird Bedrohungsmodellierung verwendet, um eine Software Supply Chain auf Basis von Docker zu bauen und abzusichern. Die Hauptkomponenten werden modelliert und Bedrohungen mit Hilfe von STRIDE identifiziert. Daraufhin werden Risiken berechnet und Möglichkeiten diskutiert, die Software Supply Chain auf Basis der Sicherheitsziele Vertraulichkeit, Integrität und

Verfügbarkeit abzusichern. Als Resultat dieser Arbeit stellte sich heraus, dass einige Komponenten eine spezielle Behandlung im Sicherheitskontext benötigen, da sie über ein hohes Restrisiko verfügen, Ziel eines Angriffes zu werden. Diese Arbeit kann als Basis für den Bau und die Absicherung einer Software Supply Chain genutzt werden. Jedoch müssen zusätzliche Komponenten, wie beispielsweise ein Monitoring- und Logging-Prozess, oder die Integration in bestehende Business-Prozesse überprüft werden.

Contents

List of Figures

List of Tables

1 Introduction

Agile software development has been in mainstream for a few years now [120]. According to the survey *The 10th annual State of Agile Report* [116] 62% of the participating companies use this approach to speed up their product delivery chain and increase team productivity. To establish agile processes in the enterprise, containerization is often used [42]. Containerization offers isolation of an application into its own environment based on container technology like LinuX Containers (LXC), as an alternative to hypervisor virtualization [114]. These containers are light weight and easier to rollout due to encapsulation of their dependencies compared to classical virtual machines [114]. Additionally, applications running on the same host can be isolated from each other and from the host, as well as their permissions can be reduced to a minimum, which brings a benefit in security terms [42]. According to the survey *Container Market Adoption Survey 2016* [12] with 310 participating companies, Docker is the leading container technology. The survey shows that 94% of the companies use Docker as container technology, especially to increase efficiency of the development process and to support microservices. For these reasons the Docker ecosystem is used as basis for this work. According to an evaluation from Datadog [20], approximately 10,7% of their clients already adopted Docker in their company, which means that container technology is still in its beginnings and companies just started to deal with this topic. The growth rate of the adoption from May 2015 to May 2016 lies around 30%, which shows the importance of the technology. But at this point security issues could arise, since new processes aren't always build with security in mind. Six of the most used Docker images [20] have been scanned by *Docker Security Scanner* [111], whereat all of them had at least one or more critical vulnerabilities in the *latest* tag [63, 60, 62, 58, 59, 61]. From a companies' view, a survey from 2011 shows that 59% of participating companies are aware of targeted attacks [109]. Docker itself offers an approach for a software supply chain with its enterprise solution with some security features included [110], which have been continuously evolving over the last months and years. Some examples are an identity and access management, consistent builds and tests, automated security scanning, as well as signed images.

1.1 Motivation

Due to the rising interests of companies, the growth rate and the young presence of Docker on the market, this topic offers possibilities to do research. Companies are just beginning to use the technology, therefore not many matured and secured processes (particularly in the area of software supply chains) have been established. Also companies are getting hit by cyber attacks [5] and suffer from data breaches [10], for example Yahoo [92].

Software supply chains are collections of processes and components to continuously test, build and deliver software releases to customers. They are critical processes in the company, since source code and sensitive information are transferred from one component to another. New security concepts and processes are required to protect supply chains from attacks and to ensure confidentiality, integrity and availability of data [77]. In the last months Docker released some new features like secrets management [87] and announced the split in Docker Enterprise and Community Edition [30]. In the past researchers successfully demonstrated attacks on Docker, as for example an attack on host systems by using devices, which have not been namespaced [117]. Papers like *To Docker or not to Docker* [16] show the need of additional research and security concepts by pointing out existing security issues. Another challenge is the integration of new concepts (like for example containerization or microservices) into existing company infrastructures.

1.2 Contribution

Readers of this work will gain knowledge in the field of agile software development, container virtualization, the Docker ecosystem and information security. This work will illustrate possible attacks like for example Denial of Service (DoS), code injection or exploiting vulnerabilities on a software supply chain and discuss security mitigations. The reader will be able to use this work as a guideline on how to reduce the overall risk of an attack on the main components. The problems which will be addressed are as follows:

- Unawareness of companies of possible risks and attacks, as well as consequences of an attack on software supply chains
- Insecurity of critical software supply chains based on Docker
- Complexity of software supply chains
- Lack of guidelines and best practices in secure software supply chains based on Docker

The research questions for this work are as follows:

- How to design and build a secure software supply chain based on Docker?
- How to secure each step and component of the software supply chain to guarantee confidentially, integrity and availability?
- How to secure communication between components?
- Can a Software Supply Chain be completely secure?

1.3 Requirements

The main goals of this work are to build and secure a software supply chain based on Docker. Each requirement has a unique number and will be discussed in the final Chapter *Conclusion*.

#1 Understand the Background and the Docker Ecosystem

The first requirement is to gain basic knowledge which is required to achieve the main goals. This is for example the concept of container virtualization and its use in agile software development, microservices and continuous integration (CI) pipelines as well as the Docker environment. To analyze possible threats and risks, knowledge about information security, security objectives, security principles and attacks on computer systems is required. Also a structured scientific process is required to identify threats, calculate risks and discuss mitigation strategies.

#2 Build a Software Supply Chain

After explaining background information, a typical software supply chain has to be described. The descriptions should be limited to the main components and processes.

#3 Secure the Software Supply Chain

After explaining the software supply chain, a threat modeling process has to be applied to find threats. The threat model should be based on the main security objectives confidentiality, integrity and availability [77]. After the main components and threats have been identified, they need to be assessed and possible countermeasures described to reduce the overall attack surface to a minimum.

#4 Use Docker as base Technology

Since Docker is the leading container technology on the market, it is used as base technology for this work [12]. It should be used to containerize and run components of a software supply chain, as well as the application which is developed.

#5 Use Open Source / Free software

To allow a cost efficient and open software supply chain which can be used by everybody, only open source or free software should be used. This excludes software like Docker Enterprise or other commercial pre-built solutions from the scope of this work. However, proposed concepts are designed to be adaptable to other products as well.

1.4 Related Work

Docker Scan, Claire and Docker Bench for Security

Docker Scan [19], *Claire* [17] and *Docker Bench for Security* [40] are security analysis tools for the Docker ecosystem. *Docker Scan* and *Claire* provide analysis for Docker images and the *Docker Registry*. *Claire* focuses on static analysis of Docker images, while *Docker Scan* also allows to scan registries. *Docker Bench for Security* checks for common best-practices around containers in production. These tools help to analyze sub-aspects of this work, but not the complete software supply chain.

To Docker or not to Docker

The paper *To Docker or not to Docker* [16] analyzes parts of the Docker ecosystem, as for example the *Docker Daemon*, *Docker Hub* or networking in terms of security and shows some vulnerabilities which still need to be addressed. This paper provides a good basis for additional research as it shows some basic weaknesses. This work will inspect all components and processes of a software supply chain, including the subset of this paper.

Understanding and Hardening Linux Containers

The whitepaper *Understanding and Hardening Linux Containers* [33] discusses basic container technologies, evaluates them and discusses security features. This whitepaper provides good basic knowledge about Docker containers and will be used to describe Docker images in context of security and how to reduce the attack surface.

CIS Docker 1.13.0 Benchmark

The *CIS Docker 1.13.0 Benchmark* [22] describes requirements and configurations for a secure Docker environment. This guidelines can be used to harden hosts in a Docker environment.

Securing Jenkins CI Systems

The article *Securing Jenkins CI Systems* [82] describes how to reduce the attack surface of a *Jenkins* build server. It describes general approaches like enabling *Jenkins security*, SSL encryption or disabling the CLI. This article can be used to apply basic hardening to the build server used in this work.

Security Assurance of Docker Containers

The article *Security Assurance of Docker Containers* [84] reviews security aspects of Docker containers. It describes *Notary*, *Docker Security Scanning* and different security scanners. This article helps to get basic knowledge about *Docker Content Trust* and *Container Scanners*, which will be discussed when securing the software supply chain.

1.5 Structure of this Work

In Chapter *Agile Software Development and Docker* background knowledge to agile software development, container virtualization and Docker is explained. In the next Chapter *Methodology* the basic threat modeling process which is used to identify threats and calculate risks is described. Afterwards in Chapter *Modeling a Software Supply Chain* the software supply chain is described and the main components are identified. The following Chapter *Threat Analysis* first creates a list of components, users, data flows and trust boundaries. Afterwards, possible threats and attack vectors are elaborated. These threats are then analyzed, rated and countermeasures discussed in Chapter *Securing the Software Supply Chain*. In the last Chapter *Conclusion* a final conclusion is given and results are discussed.

2 Agile Software Development and Docker

This chapter provides basic knowledge. First, agile software development and microservices are described. It is explained why this approach combined with a continuous integration (CI) pipeline helps to deliver software of higher quality. Afterwards, differences between hypervisor and container virtualization and its integration into agile software development are explained. Finally, an overview of the Docker ecosystem is given.

2.1 Agile Software Development

Agile software development defines values and principles to develop valuable software in an iterative and quick process, while facing continuously changing requirements [90]. An agile software development process aids to help teams to respond to unpredictability by using an iterative workflow with continuous feedback [90]. It uses light-but-sufficient and human- and communication-oriented rules to stay slim and efficient without increasing the risk of mistakes [13]. This has proven to be more successful for specific projects than other approaches, therefore more and more companies are starting to use it [116]. In 2001, the *Agile Alliance* developed a statement of values which are used to work quickly and respond to change. This statement is called *The Manifesto of the Agile Alliance*. It describes four rules (quoted from *Agile software development: principles, patterns, and practices* [90]):

- "Individuals and interactions over processes and tools"
- "Working software over comprehensive documentation"
- "Customer collaboration over contract negotiation"
- "Responding to change over following a plan"

The meaning of the four rules are explained in the following paragraphs. The descriptions are based on sources: [90, 13].

Individuals and interactions over processes and tools

This value means that good processes are relevant for the project, but they don't guarantee success. More important than processes and used tools is the team itself. Good communication and team building increase efficiency and reduce mistakes caused by misunderstandings.

Working software over comprehensive documentation

Documentation is required all over the project to pass information from one person to another. Missing documentation can lead to misunderstandings and reduce quality of the project. On the other hand, if there is too much detailed documentation, efficiency is decreasing. For this reason, short, concise and meaningful documentation is required.

Customer collaboration over contract negotiation

This value describes the relationship between developers and customers. Contracts are useful to mark borders, but software cannot be completely defined from the beginning. Hence regular customer feedback and collaboration needs to be part of the process.

Responding to change over following a plan

Change is an essential part in agile projects. For this reason plans need to be flexible and able to adapt to each change in technology and business.

Multiple processes and frameworks have been developed which follow the agile approach: *Scrum* [106], *Crystal* [14], *Adaptive Software Development* [37] or *Extreme Programming* (XP) [3].

As the survey *Agile Development: Mainstream Adoption Has Changed Agility* from Forrester has found out, Scrum is the most popular agile development process with 10.9% that is used by organizations [120]. It is a software development framework based on interoperable teams which work together to reach a common goal [105]. It defines member roles, such as a product owner, a scrum master and the development team [105]. It also defines artifacts, like for example the product backlog, the sprint backlog and burn-up / -down charts to manage the product and measure results of each work package [105]. All work is done in sprints, which is a planned period of time (for example 14 days) in which items of the sprint backlog are completed [105].

2.2 Microservices

Microservices are an approach to cut down monolithic applications into small parts that work together, called microservices [29]. A monolithic application is one single executable unit which can grow over time [29]. For every change the whole application has to be deployed [96]. Microservices instead are small and lightweight applications, which are independently maintainable and deployable [4]. They often communicate via REST and can be developed in any programming language [29]. The key benefits of microservices are listed in Table 2.1.

Benefit	**Explanation**
Technology Heterogeneity	A system can consist of services based on different technologies, each optimized for its use case and people working on the service.
Resilience	If one system fails, the failure does not cascade and the problem can be isolated.
Scaling	Each service can be scaled independently to fit the requirements.
Ease of Deployment	Services can be deployed independently, which speeds up the deployment process.
Organizational Alignment	Smaller teams are working on smaller codebases, which is more productive and can be better aligned to the architecture of the organization.
Composability	Services can be reused for different purposes since interfaces have to be documented.
Optimizing for Replaceability	Replacing a small microservice is easier and less critical than replacing an old monolith.

Table 2.1: Key Benefits of Microservices ([96])

Agile software development processes, like for example *Scrum* define small interdisciplinary teams which are responsible for a single product [105]. If a product consists of multiple microservices, each service can be managed by a *Scrum* team. By using this approach, features can independently be developed and deployed. Additionally, containerization can be used to package those microservices including all its dependencies into small and isolated containers, which makes them suitable for big container clusters, like for example Docker Swarm or Kubernetes [33].

2.3 Continuous Integration / Deployment

Agile software development processes nowadays use CI to frequently build and test work done by developers. This integration is described in the book *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation* [38], which is used as basis for this chapter. Using CI increases overall software quality and reduces release cycles. CI stands for the collection of techniques, tools and processes to automatically build and test software on each change. A deployment pipeline is the implementation of such an automated build, test, deploy and release process. Figure 2.1 shows a typical deployment pipeline.

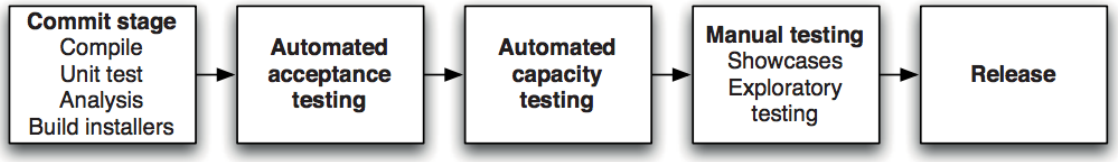

Figure 1.1 *The deployment pipeline*

Figure 2.1: The Deployment Pipeline ([38, p. 4])

It starts with a *Commit state* in which developers compile source code, run tests and analysis tools and build an installer. The following steps (*Automated acceptance testing*, *Automated capacity testing* and *Manual testing*) are a series of tests to prove that the software is ready to be released. The final step is to release the software. This process allows to find errors as quickly as possible because it automatically stops and notifies developers, if a test fails [28]. Typically a build server is required to automatically build software on a regular basis. This server runs tests for each component (unit tests) or tests for the collaboration of different components (integration testing).

CD extends a CI process by automatically deploying each change to the target environment after tests and build succeeded. This step is possible, if the included unit, component and acceptance tests are in high quality and cover a big part of the application.

2.4 Software Supply Chain

The term supply chain management (SCM) is defined as the concatenation of systems and processes to fulfill an order [119]. SCM ranges from the *Source of Supply* to the *Point of Consumption* [119]. The goal of SCM is the supply, removal and recycling of organizational activities [119]. Different components have to be analyzed, like for example quantities, qualities, prices, delivery and storage locations as well as delivery dates [119].

Derived from traditional SCM, a software supply chain (SSC) is a combination of processes and required resources to deliver software. The derived components are software quality, licenses, infrastructure and release dates. The complete process of traditional SCM and a modern SSC can be matched with each other, as shown in Figure 2.2.

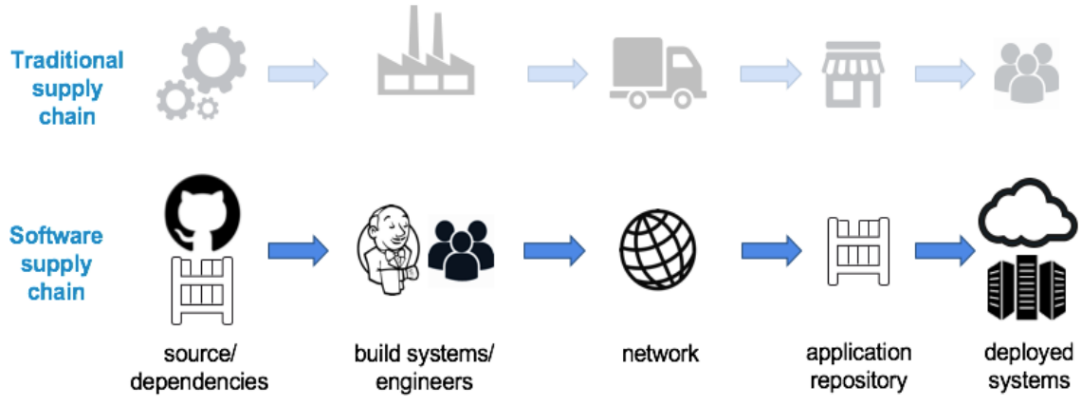

Figure 2.2: Software Supply Chain ([110])

SCM has to identify raw materials (in SSC: sources and dependencies), assemble them (in SSC: build systems and engineers), ship the item (in SSC: network), store the item (in SSC: application repository) and finally sell it (in SSC: deploy) [110]. The build and ship process can be mapped onto a CI pipeline to automatically build and deliver software regularly. The main objectives which are addressed when realizing a SSC are listed in Table 2.2. Those objectives are achieved using the key principles of a supply chain, which are listed in Table 2.3.

Factor	**Explanation**
Efficiency	*Doing the right things* and *Doing the things right* to increase overall efficiency.
Managing Competition Factors	Watching the key factors and knowledge about competition.
Costs	Reduce delivery, storing or hosting costs.
Time	Optimizing throughput time of the software.
Quality	Increasing software quality.
Flexibility	Reacting to changes.

Table 2.2: Objectives of a SSC ([110])

Principle	Explanation
Compression	Reducing the number of steps which are required to build the software.
Cooperation	Usage of partners to achieve the objectives.
Virtualization	Combine competences and build virtual networks to act as a single unit.
Standardization	Use standardized modules to optimize exchange of parameters within the supply chain and reduce delivery times.
Client orientation	Changes are triggered when client need exists (pull principle).
Optimization	Optimization based on experiences and calculations.

Table 2.3: Key Principles of a SSC ([110])

2.5 Virtualization

To create a dynamic SSC, microservices in combination with container virtualization is used, to better use available hardware resources. Virtualization in general is an approach to divide hardware resources into multiple environments [2]. Gartner says that the market has matured over the last years and many organizations have virtualization rates bigger than 75% in their data centers [74]. Cloud providers, like Amazon AWS or Microsoft Azure use virtualization in their data center which helps them to provide infrastructure as a service (IaaS) [7]. Virtualization can be found on both, client side and server side. The server side virtualization can be divided into two classes: hypervisor based virtualization and container based virtualization [7]. Hypervisor based virtualization depends on a piece of software called hypervisor, which abstracts hardware resources for virtual machines. Container based virtualization defines so called containers, which can be used to isolate applications from each other on the same OS kernel [7].

2.5.1 Hypervisor Virtualization

Hypervisor based virtualization allows complete virtual machines (VMs) to run on a hypervisor. Those VMs consist of a complete OS, including a kernel, dependencies and applications [7]. The hypervisor itself is a piece of software which either runs directly on hardware or in an operating system [7]. Two classes of hypervisors exist: *Type 1* and *Type 2* (Figure 2.3).

Type 1 hypervisors (also known as bare metal hypervisors) run directly on hardware. An example for a *Type 1* hypervisor would be VMWare ESXi or Xen Hypervisor [33, 7]. *Type 2* hypervisors (also known as hosted hypervisors) run on top of a host operating system.

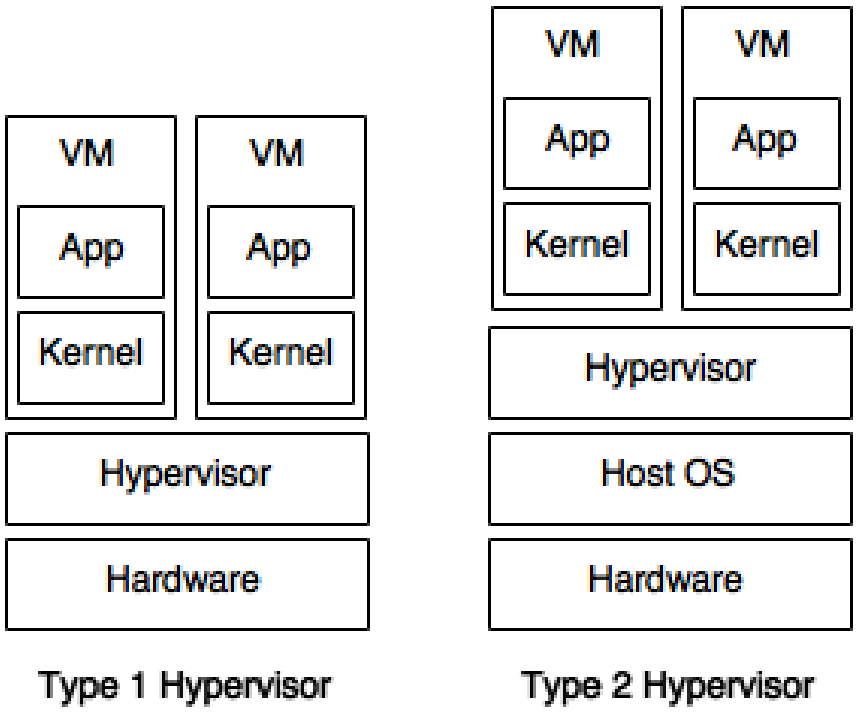

Figure 2.3: Hypervisor based Virtualization

Examples for a *Type 2* hypervisor would be VirtualBox, VMWare Workstation or QEMU [75, 93]. The isolation degree is quite robust, since special CPU instructions provide hardware isolation and the hypervisor itself offers a small attack surface [33]. Nevertheless researches have shown successful attacks (for example VENOM [95] on QEMU), although they are rare [33].

2.5.2 Container Virtualization

Container virtualization (also called *containerization*) uses kernel functionality to isolate groups of processes from each other [33]. This provides the basis for technologies like for example *Docker* or *Rkt*. The isolated environments are called containers. They are created using a combination of multiple kernel features, such as kernel namespaces, cgroups or root capabilities. They share the same OS kernel, so no hypervisor is required [7, 93]. Figure 2.4 shows multiple application containers (boxes with double lines) which run on the same host OS.

Compared to hypervisor virtualization, footprints of applications are smaller, since no additional OS and kernel is required in-between [33]. Container virtualization offers higher performance and faster start up times since the applications run directly on the kernel. This is why it is preferred over hypervisor virtualization when performance is needed [75, 33].

Container technologies as LXC, Docker or Rkt offer platforms and definitions for con-

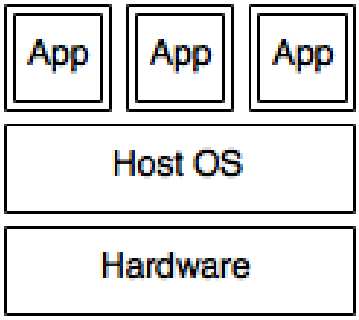

Figure 2.4: Container Virtualization

tainers and images. Those approaches gained popularity in the last years, since the ongoing shift from traditional three tier datacenters to large computers running multiple virtual machine instances [33]. Container technologies abstract many details of how containers are created and managed away from developers [8].

Mechanisms for Container Virtualization

Containers are mainly created by using the following kernel features: Kernel namespaces and Control Groups (cgroups) [33]. These features focus on creating process groups which are isolated from each other (kernel namespaces) and imposing resource limits (control groups) [33].

Kernel Namespaces

Kernel namespaces provide isolation feature of container technology. They isolate multiple processes by logically dividing their kernel space into multiple environments (for example network, processes or file system) [7]. This causes processes to not be able to see or manipulate other processes running on the same host [53]. Isolation is achieved by splitting the global resource identifier table of the kernel and other structures like for example networks into multiple instances (one per process). This creates a per-process view of the kernel [33]. However, not all kernel functionalities as for instance devices, time, syslog or proc and sys pseudo file systems support namespaces [33]. The main namespace features are mount namespaces, inter-process communication (IPC) namespaces, UNIX Timesharing System (UTS) namespaces, process identifier (PID) namespaces, network namespaces and user namespaces [33]. Mount namespaces provide a specific view to the file system. IPC namespaces allows the creation of objects which are visible to members of the same group, but not to others. This is often used to share memory between processes [7]. UTS namespaces allow to set a custom domain or hostname for each member, which is useful for example for hosting a web application or logging [7]. PID namespaces are used to create new processes using a PID starting at 1, which is useful for porting applications

from one host to the other, while maintaining the PIDs of running processes [7].

Control Groups

Control Groups (or cgroups) are used to limit hardware resources like CPU count and usage, disk performance or memory to control performance or security [7]. Those restrictions can be applied to a single process or a collection of processes. Cgroups can be used to ensure that a single container cannot exhaust the system by using all of its resources [53]. The rules are organized in a tree structure and they are inheritable and optionally nestable. Cgroups can be seen as an enhancement to basic ulimits / rlimits and can be used as an additional security mechanism besides kernel namespaces [33]. The configuration is done via a special virtual file system mounted in `/sys/fs/cgroup` and can be changed at any time [33]. The main cgroup subsystems are CPU, memory, BLKIO, devices, network and freezer. If configured wrong, this can also be a security issue, since it can be used for a container escape [36]. In context of container technology, most of cgroups management is abstracted away, with the exception of LXC [33].

Security of Container Virtualization

Compared to hypervisor virtualization, container virtualization offers less isolation since no hypervisor is in between containers [33]. While in hypervisor virtualization an attacker would have to break the OS kernel and additionally the hypervisor, the only layer of security in container virtualization is the OS kernel itself [33]. If an application inside a container contains an exploitable bug, an attacker can get access to it [33]. From there it is possible to either attack the kernel with a kernel vulnerability or scan the network for other containers or hosts who could also be compromisable or contain sensitive data [33]. If a kernel vulnerability exists, all applications sharing the same kernel are affected and isolation mechanisms could be bypassed. This would be a compromise of all containers running on the same kernel and the complete host system [33]. To improve isolation capabilities and prevent container to host escapes, the following kernel features are used: Linux capabilities, Mandatory Access Control (MAC) and Seccomp [33].

Linux Capabilities

Linux Capabilities are attributes which limit privileges of processes run by the root user [7]. They help to enforce namespaces by restricting powers of the root user in containers [7]. This is for example problematic if the setuid bit is used in combination with root owner to execute a binary with root privileges [33]. If the binary contains a memory vulnerability, root access can be obtained by everyone who has access to the binary [33]. By limiting

capabilities of the binary to for example only access to a raw socket (CAP_NET_RAW), damage which can be done by abusing the vulnerability is reduced, since the attacker now has limited access to raw sockets [33]. Linux Capabilities are stored using the extended attributes (xattr) in the security namespace of the binary. Additionally, when starting the binary, a capability bitmap is created for the process and then enforced by the kernel [33]. Some examples for capabilities which were randomly picked for this work are listed in Table 2.4.

Capability	**Explanation**
CAP_SETCAP	Change UIDs / GIDs of files.
CAP_KILL	Send the kill signal to a process.
CAP_SYS_CHROOT	Use chroot to change root directory.
CAP_SYS_MODULE	Load and unload kernel modules.
CAP_SYS_RAWIO	Perform I/O port operations, for example on `/dev/mem`.

Table 2.4: Linux Capabilities ([33])

Mandatory Access Control (MAC)

Mandatory Access Control (MAC) is an optional security feature for containers [33]. It controls access to objects (files, sockets or directories) by using subjects (processes or users) based on security contexts [34]. The default rule is denying each object access, unless not explicitly allowed. This feature is fully integrated into the kernel, so it is possible to reach and control every access made [33]. Due to the complex rules, it can be hard to configure [33]. The most common frameworks are AppArmor or SELinux [7].

Syscall Filtering with Seccomp

Seccomp is a kernel feature which allows the transition of a process into a *secure computing mode* [33]. In this mode the process is only able to make the following system calls: `exit()`, `sigreturn()`, `read()` and `write()` [33]. This mode is also called *SECCOMP_MODE_STRICT* [33]. If the process attempts to make another system call, the kernel will terminate the process with a *SIGKILL* [33]. Another mode is *SECCOMP_MODE_FILTER*, which allows to filter system calls for a process using *Berkeley Packet Filter* (BPF) rules. This mode requires the kernel extension *seccomp-bpf* [33]. BPF is a pseudo-language which was designed to allow performant in-kernel bytecode evaluation in a safe and simple language [33].

2.6 Docker Ecosystem

After describing the core concepts of container virtualization, Docker which is based on those concepts is described in this chapter. It adds an abstraction layer on top of the earlier described container virtualization mechanisms. Docker is a platform to develop, ship and distribute applications [51]. It allows to package applications into containers, which can be run isolated from each other on the same host without the need of a hypervisor [7, 42]. It provides tooling and the platform to manage the lifecycle of containers [51]. Docker also allows to combine multiple hosts into one single cluster and distribute applications onto this cluster [71]. Docker can be used to create standardized environments for applications and integrate those into a CI workflow, to automatically test, deploy and scale them [71]. This helps developers to develop applications in the same environment as used in production [51]. In this work, Docker is used to containerize an exemplary application which is then built and shipped using a SSC. It is also used to run and scale the application in a Docker swarm, as well as for running components of a SSC. The central part of Docker is Docker Engine, as explained in Chapter *Docker Engine* in more detail. The latest release of Docker is used in this work (17.06). An overview of the main services offered by Docker is shown in the following list [51]:

- Docker Engine (core of the Docker ecosystem)
- Docker Compose (definition of one or multiple services in a single file)
- Docker Swarm (orchestration of containers on highly available clusters)
- Docker Registry (storage of Docker images)
- Universal Control Plane (management of containers and container clusters in business environments)
- Docker Secrets (management of secrets in a swarm)
- Docker Content Trust (store and validate signed Docker tags)

2.6.1 Docker Engine

The Docker Engine is the core of the Docker ecosystem. It is a client server application which has three major components, as shown in Figure 2.5: the Docker Daemon which runs on the host, a REST API provided by the Docker Daemon and a command line interface (CLI) (the `docker` command) [51]. The REST API takes commands from the CLI and processes them further [51]. The API can be exposed either via local socket on the same host or a network exposed port [51]. The CLI offers commands to manage

networks, containers and images or volumes for containers [51]. The Docker Engine needs to be installed on each system, which interacts with Docker.

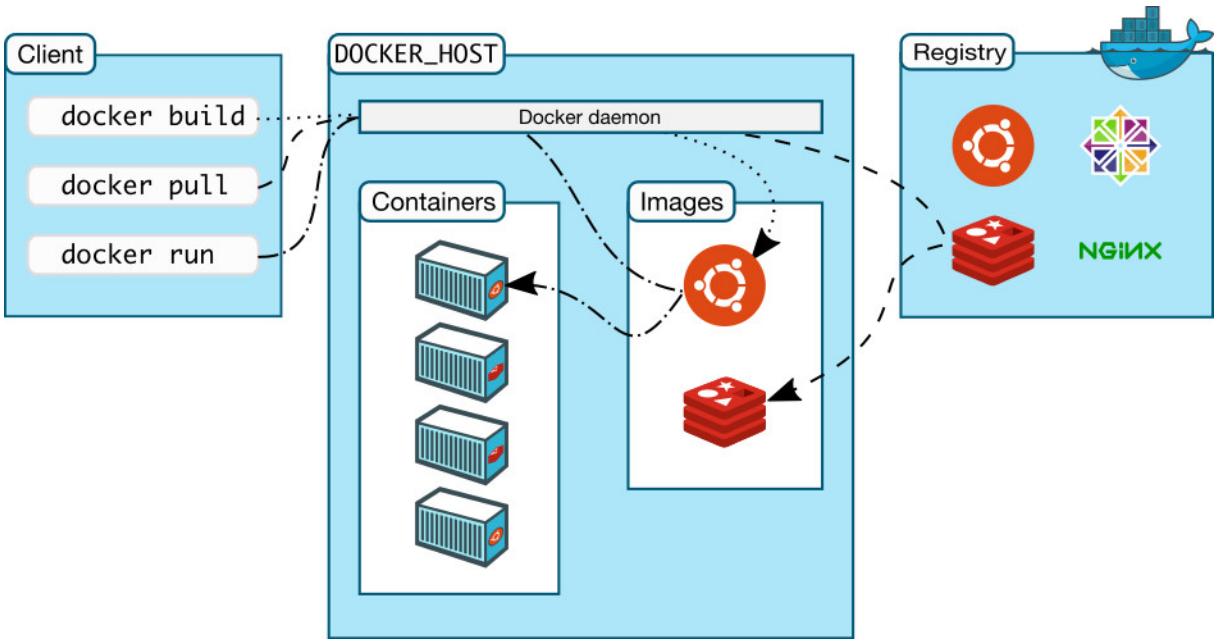

Figure 2.5: Docker Overview ([51])

Docker Daemon

The Docker Daemon runs on the host (as root) and is responsible for listening and processing API requests from the Docker Client [53]. It also manages Docker Objects as for instance images, containers, networks and volumes. To build images it parses a so called `Dockerfile` and executes the instructions [33]. Trusted users should be allowed to control the Docker Daemon, since if someone controls the `dockerd` process, he is able to spawn a privileged container, which is able to mount the root filesystem of the host as writable [33].

Docker Client

The Docker Client (or the `docker` command) is used to communicate with the Docker Daemon. The Docker Client is the primary way to run a command in the Docker world [51]. If a `docker` command is run, the client sends it to the Docker Daemon, which directs all further actions [51].

2.6.2 Docker Images

Docker Images are read-only templates (blueprints) for running a container [51]. They contain the root file system for the container plus some additional parameters and a config-

uration file [51]. To build an image, a Dockerfile with instructions is used in combination with the `docker build` command [51]. This will result in a multi-layered read-only file system representing the Dockerfile. Each instruction in the Dockerfile will create another layer on top of the layered file system [51]. To speed up the build process, each layer is cached [51]. If another `docker build` command is executed, only those layers which have changed are rebuilt [51]. An image is identified by an image record identifier, like for example *mydomain.de:5000/my-image:v1* [45].

2.6.3 Containers

Containers are a runnable instance of an image with a lifecycle, which is shown in Figure 2.6 [51].

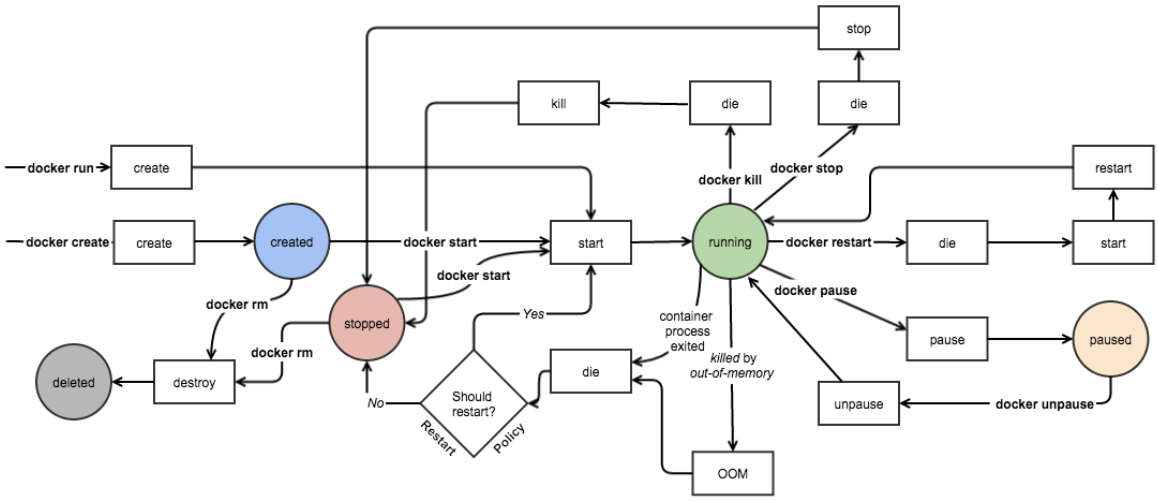

Figure 2.6: Docker Container Events and States ([104])

The most main states of a Docker container are *created*, *running*, *paused*, *stopped* and *deleted* [104]. The Docker CLI offers commands to control the creation, execution, stopping or deletion of containers [51]. To access network or data, it is possible to attach networks or volumes to the container [51].

2.6.4 File Formats

Dockerfile

The Dockerfile is a plaintext file which contains directives on how to build a Docker image, each in a new line [55]. Each instruction (with some exceptions) creates a new layer on the layered file system of the image [55]. The Docker Engine is responsible for parsing and

interpreting those instructions, then building the final Docker image from it [55]. Those directives can for example define the base operating system (`FROM`), run a command in the image (`RUN`), expose a port to the network (`EXPOSE`), add a volume (`VOLUME`) or define the initial command to be executed when the image is started (`CMD`) [33, 55]. Listing 2.1 shows a sample Dockerfile for a Python application, which is based on the official python image ([68]) with the tag *3.4-alpine*. It first copies the current working directory into a path called `/code` in the image and defines this folder as the working directory. Afterwards it runs `pip install -r requirements.txt` in the image to install dependencies, assuming the file `requirements.txt` exists in the `/code` folder. Afterwards, the command which is executed when the container is started is defined. The `Dockerfile` is used in this work to build each component that is based on Docker.

Listing 2.1: Sample Dockerfile

```
FROM python:3.4-alpine

ADD . /code
WORKDIR /code

RUN pip install -r requirements.txt

CMD ["python", "app.py"]
```

Compose file

The Compose file defines an application which is made of one or multiple services [44]. A service is for example a database (communicating on port 3306 and backed by a NFS volume), a frontend (listening on port 80 and 443 and communicating with the backend service) or a backend service. The Compose file describes parameters, such as listening ports, the Docker image, build context or resource limits, which are passed to the containers when they run in a Docker environment [44]. It also defines the networking environment, as well as which service is placed into which network and whether they are able to communicate with each other [44, 49]. To share data between containers, it is possible to define data volumes. Local volumes can only be used in non-swarm mode, whereas in swarm mode shared volume drivers can be used, like for example NFS, SMB or iSCSI [44]. Beginning with version 3 of the Compose file reference, deployment parameters can also be defined, as for instance rolling update policies, replication counters or placement constraints [44]. Listing 2.2 shows a basic Compose file which defines two services: *redis* and *web*. The *redis* service is just a reference to the `redis:alpine` image from Docker Hub, whereas the *web* service is built from the `Dockerfile` in the current directory. Additionally, the *web* service has a volume and an exposed port 5000, which is

mapped to port 5000 on the current host. In this work, the Compose file is used to define services which are required for the components of a SSC.

Listing 2.2: Sample Compose file

```
version: "2"
services:
  web:
    build: .
    ports:
     - "5000:5000"
    volumes:
     - .:/code
  redis:
    image: "redis:alpine"
```

2.6.5 Docker Compose

Docker Compose is a CLI application written in *Python*, that parses a Compose file and builds a multi-container environment from it [67]. Docker Compose parses a given Compose file, translates the service, networking and volume definitions into `docker run`, `docker volume create` and `docker network create` commands and executes them [67].

2.6.6 Docker Registry

An image registry is a central repository for container images [43]. The Docker Registry is an image registry maintained by *Docker Inc* [51]. Typically images are built locally or by a special build server and afterwards pushed into a registry [43]. From here images can be pulled and run in a container cluster like Docker Swarm [43]. *Docker Hub* [41] and *Quay.io* [18] are two examples of cloud-hosted public registries. If no other registry is specified in the *Docker Engine*, *Docker Hub* is used as default [43].

2.6.7 Docker Swarm

Docker Swarm is the functionality to manage and orchestrate services over multiple nodes which are running the Docker Daemon [71]. It allows distributing workloads across multiple nodes which act together as a single unit [51]. As shown in Figure 2.7, a Docker Swarm consists of multiple manager nodes which share a common state using the Raft Consensus Algorithm [97] and worker nodes which can receive tasks [56].

The swarm and its services can be controlled using the Docker Engine CLI and API [71]. Those features are encapsulated in a package called SwarmKit and shipped with the latest version of the Docker Engine (Version $\geq$ 1.12) [48]. In this work Docker Swarm is

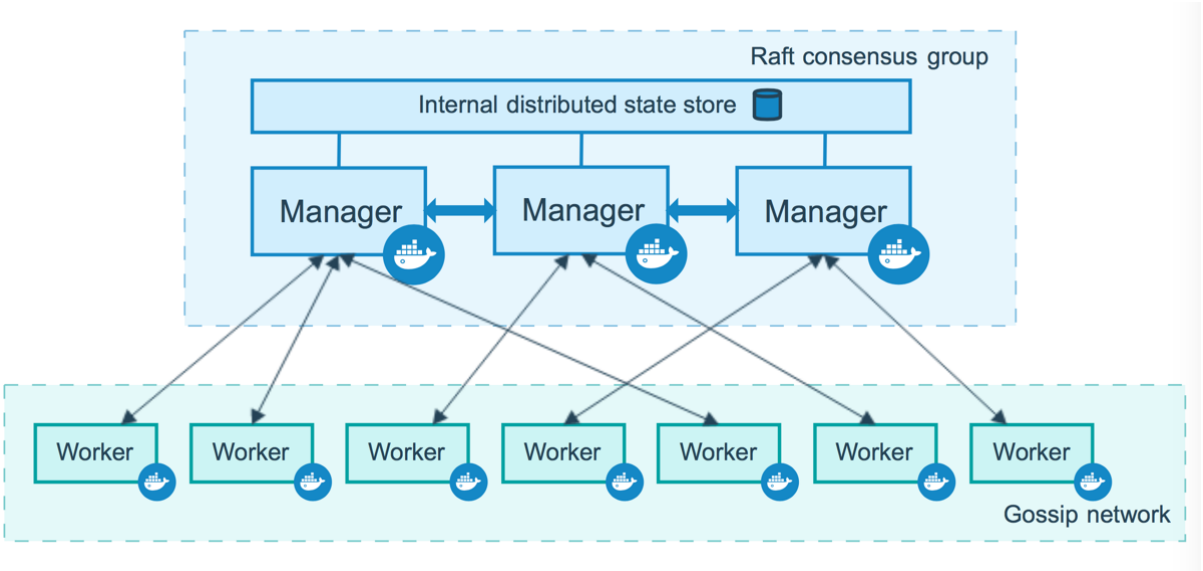

Figure 2.7: Docker Swarm Overview ([56])

used to run components of a SSC. These are for example the development environment including the version control system, the build environment containing the build server and the testing and production clusters, which are used to provide the application to testers and end users.

Services

Services define how to run an existing Docker image, the command and parameters (such as port, resource limit or volumes) and the desired state in the cluster [57]. A manager node accepts a service definition by an API call and splits this service into tasks, who are scheduled on available worker nodes [57].

Figure 2.8 shows how one replicated nginx service is split into multiple tasks, each running on a different node [57]. The scheduler of the manager is responsible for scheduling tasks onto nodes with available resources [57]. Services marked as *global* are running on each node available [57]. To an end user it looks like he is communicating with a single node of the application.

Nodes

Nodes which are participating in the cluster are running the Docker Daemon in swarm mode [71]. Nodes can be distributed across multiple physical hosts and cloud machines

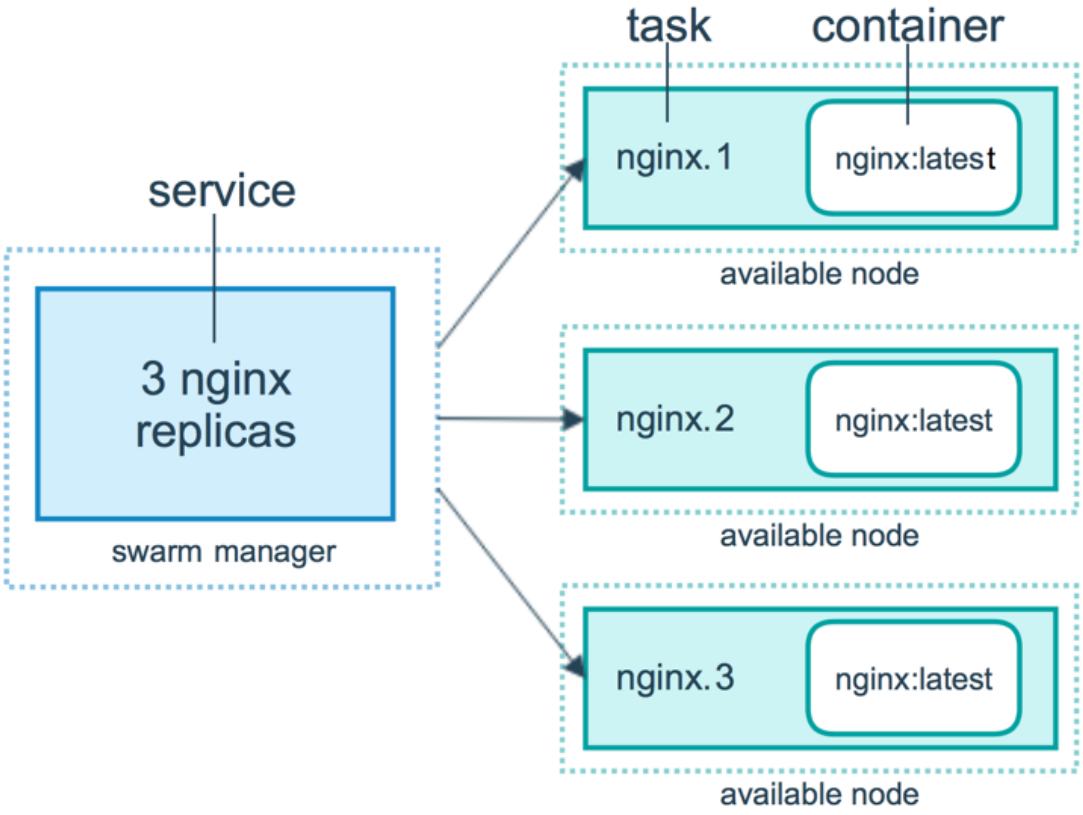

Figure 2.8: Services, Tasks and Containers ([57])

[71].

Manager nodes are responsible for maintaining the status of the cluster, scheduling services and providing the API which is used to control the cluster [56]. Every manager node has access to the state of the cluster by the shared Raft log [56]. By default, manager nodes are also able to execute workloads, this can be disabled in the configuration [56].

Worker nodes receive and execute tasks from the manager nodes [71]. They are running an agent which receives commands from the manager node and reports back the current state of the tasks they execute [56].

Overlay Networks

Overlay networks are used to enable communication between containers in Docker Swarms and to route incoming traffic to the correct container which can be distributed over multiple nodes [49]. Docker Swarm managers are responsible for sharing the same overlay network among all required nodes [49]. Only swarm services are able to connect to overlay networks, no standalone containers [49]. If the user wants to use overlay networks in non-swarm mode, an external key-value storage like etcd is required [49]. When swarm mode is enabled, Docker automatically creates a default overlay network (called *Ingress*) to route incoming traffic to the corresponding container which has published a specific

port [49].

2.6.8 Docker Secrets

Some services need additional credentials, for example a database username and password [66]. Credentials are blobs of data (for example passwords or SSL certificates) which need to be securely provided to the containers [66]. Those secrets should not be transmitted or stored unencrypted in a Dockerfile or the application source code [66].

Docker Secrets is a solution of Docker for secret management which is built into SwarmKit [66]. It was introduced in Docker version 1.13 [66]. Secrets are centrally managed in the encrypted swarm's Raft log, which is replicated across all manager nodes [66]. By running the CLI command `docker secret create`, the secret is sent to a swarm manager over a mutual TLS connection [66]. A secret can be up to 500kb large and can only be used by swarm services, not by single run containers [66]. If a service has been granted access to a secret (for example by using the `--secret` parameter when running a service), a manager pushes it securely to the corresponding Docker Daemon, which then mounts it in an in-memory file system into the container [66]. The container is then able to access the secret in the path `/run/secrets/<my_secret>` [66]. It is possible to grant and revoke secrets to a service at runtime, but all containers of a service are restarted [66]. To change a secret while the service is running, it needs to be rotated [66]. When a container stops, the decrypted secret is automatically unmounted and flushed from the nodes memory [66]. Docker Secrets can also be used to store non-sensitive data, as for instance configuration files [66].

2.6.9 Docker Content Trust

This chapter is based on the Docker documentation [45, 65] and the article *Security Assurance of Docker Containers* [84].

Docker Content Trust (DCT) is a mechanism to sign and verify image tag. When pushing an image into a registry, the image can be signed. After pulling an image out of a registry, this signature can be verified. *Notary* is the client and server utility behind DCT. It is used to verify content which can be distributed over an insecure network. It is based on *The Update Framework* (TUF). Figure 2.9 shows repositories bound to a single person or to an organization. A repository can have multiple tags, both signed and unsigned. A signature of an image is always assigned to a tag.

To force verification of each pulled image on client side, the environment variable `DOCKER_CONTENT_TRUST=1` can be set. This is disabled by default. If DCT on client side is enabled, unsigned images of a repository are ignored.

Figure 2.10 shows different keys used to sign image tags. Each repository has a set of

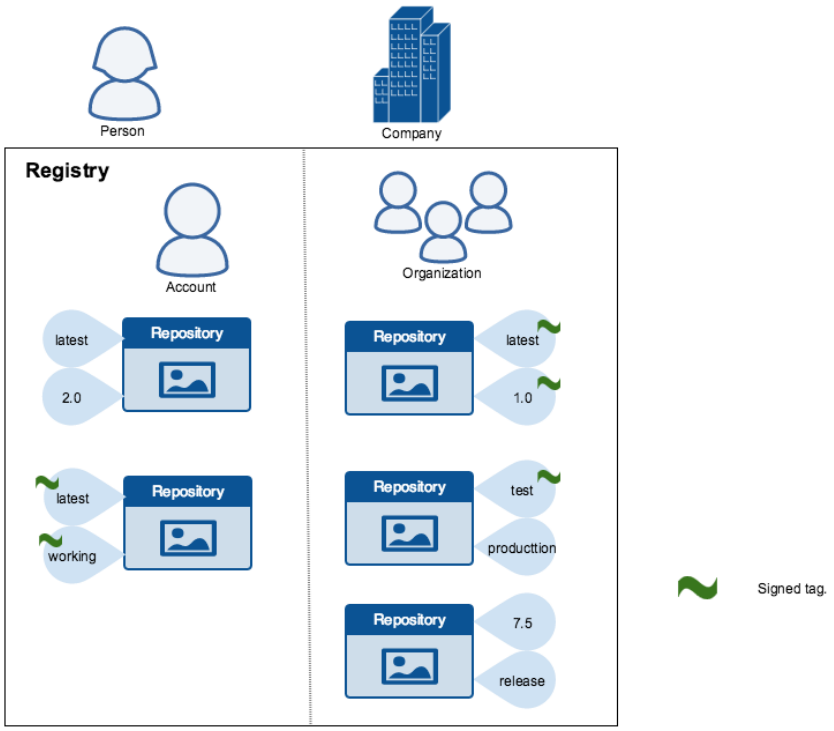

Figure 2.9: Docker Content Trust ([45])

keys which are used by developers to sign their tags. The keys are created in an interactive process when an operation using DCT (`docker push`, `docker build`, `docker create`, `docker pull` or `docker run`) is executed for the first time. The following different keys exist:

Offline / Root Keys

Offline keys are the root keys for creating content trust. They are bound to a specific person or an organization and used to create tagging keys for repositories. They should be stored at a safe place and backed up securely.

Targets Key

The targets key, which resides on client side, is bound to a specific repository and is used by a developer to sign and push image tags.

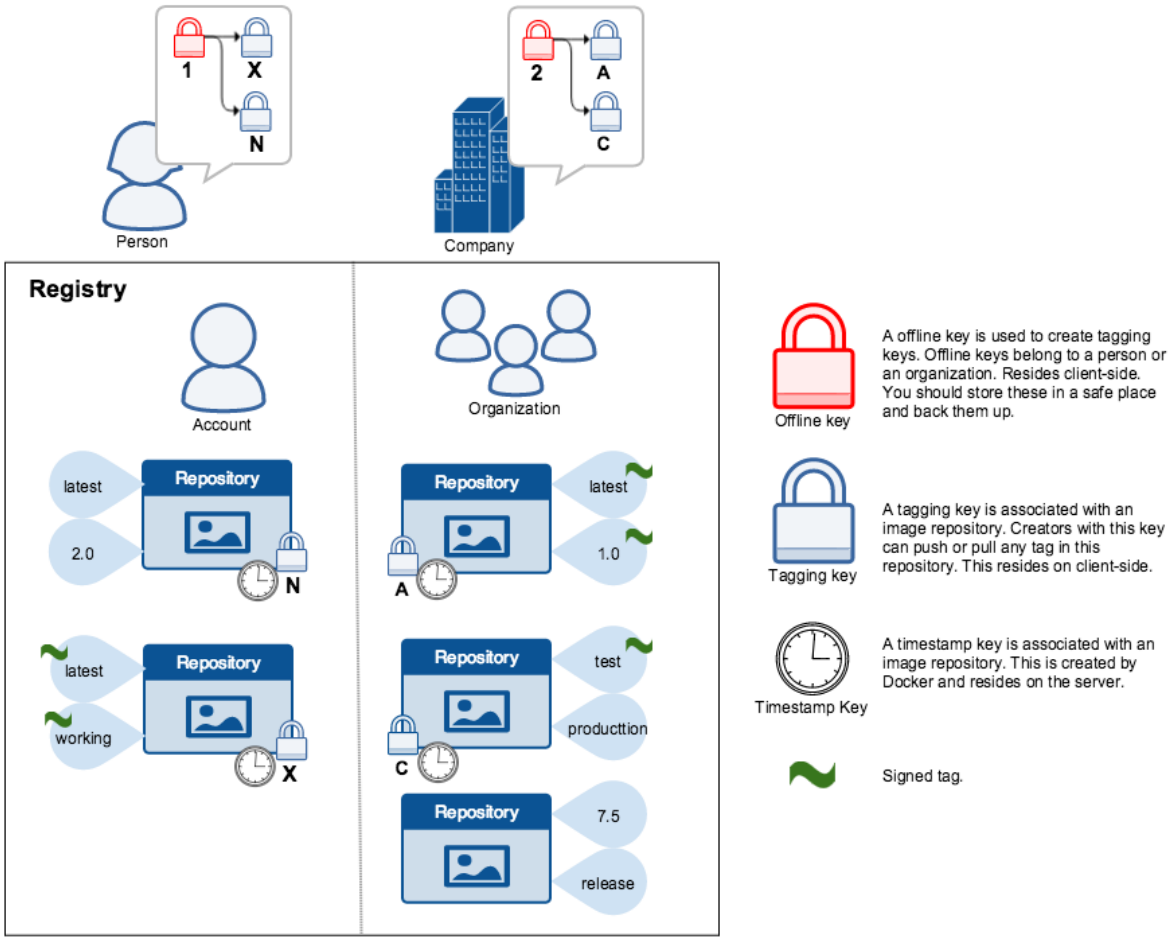

Figure 2.10: Signing Keys ([45])

Snapshot Keys

Those keys allow signing the current collection of image tags to prevent mix and match attacks.

Timestamp Keys

Timestamp Keys are bound to a specific repository and reside on the server. They are generated by Docker and used to guarantee freshness.

Delegation Keys

Those keys are optional and allow to delegate signing to other publishers without sharing the targets key.

2.7 Immutable Infrastructure

The immutable infrastructure paradigm provides stable, efficient and version controlled infrastructure [108]. The main statement of this approach is that once a component (a host or a container) has been started, it should not be manually changed [108]. If some configuration has to be altered, the running instance has to be replaced with a new one [108]. This approach requires full automation and versioning of each component [21], which can be achieved by using technologies as for example ansible [103] or Docker [100].

In this work the immutable infrastructure paradigm helps to avoid manual interaction with a running container or a node of a container cluster. It aids to continuously roll out security patches and reduce the overall risk of vulnerabilities in one of the components.

3 Methodology

Before this work goes into SSCs, the methodology needs to be defined. It is started by explaining why and how information needs to be protected and by defining the main security objectives. Secure design principles and attacks on systems are described to later use them in the threat modeling process. Afterwards the threat modeling process which will be used to build and secure a SSC is explained.

3.1 Information Security

Before talking about threat modeling, an overview about information security in general and security objectives is required. This section provides background knowledge to information security. A definition of information and security objectives is given, the main security principles are described and an overview over the main attacks is shown.

3.1.1 Definitions

Asset

An asset is something of value for an organization [78]. Many types of assets exist, such as machines, facilities, software, services, people, reputation or knowledge [78].

Information

Information is an essential asset in an organization's business which needs to be protected in an appropriate way. They can be stored in many forms, for example in digital or material form [81]. The information to protect in this work is the source code of the application.

Information Security

Information security is the implementation and management of security measures with the aim of ensuring business success and continuity and minimizing impacts of information security incidents [81]. It includes three main objectives: confidentiality, integrity and availability [81]. Those objectives are also referred as *CIA* in this work.

Trust Boundary

A trust boundary is the place where multiple entities interact [107]. Threats often involve actions across trust boundaries [107]. A trust boundary is for instance a network firewall, which filters incoming and outgoing traffic [107].

Attack

Attacks in general are maliciously intended actions against one or multiple components [31]. They are attempts to steal, destroy, manipulate or expose an asset [81]. An attack consists of motivations like stealing money and one or many subgoals [31]. Multiple activities can be engaged which result in events, like for example unauthorized access to a system or halting of an application [31]. As a result, consequences like for example unavailability of a computer system occur and different impacts on business can be measured. A direct impact would be loss of revenue due to the inability to process a business transaction, an indirect impact would be negative impact on the reputation of the company [31].

Attack Vector

An attack vector is a way by which an attacker could be able to gain unauthorized access to a computer or network to do harm [80].

Attack Surface

The attack surface is the sum of places where trust boundaries can be crossed, either on purpose or by accident [107]. It describes how exposed a system is [107]. Applications with more exposed interfaces have a higher attack surface than applications with less exposed interfaces [107].

Threat

Threats describe possible events which could lead to harm of a system or an organization [81]. An example for a threat would be an attacker who is able to bypass the authorization system by abusing a vulnerability in the software.

Risk

A risk in terms of information security is the chance or probability of loss [79]. The magnitude of risk can be expressed by the combination of the likelihood of the occurrence of an event and its impact [79]. The risk level calculation is done using the formula shown in Figure 3.1.

Listing 3.1: Risk Calculation

```
Risk = Likelihood x Impact
```

Residual Risk

The residual risk is the remaining risk after risk treatment [79].

Security Objectives

The three main security objectives are confidentiality, integrity and availability. **Confidentiality** is the property that information has to be protected from unauthorized access [77]. This can be achieved by encrypting the information or keeping it at a safe place. Also authentication and authorization mechanisms can be used to grant access to specific people. **Integrity** is the property of being complete and unmodified [77]. It involves maintaining consistency and trustworthiness over the entire life cycle. Data should be protected from modifications when transferred from one person to another. This can for example be achieved by using checksums or appropriate protocols to transfer data. **Availability** is the property of information being available to authorized entities on demand [77]. An example for providing availability would be to replicate data into multiple data centers or making sure the network has enough bandwidth to serve data on high load.

3.1.2 Security Principles

To protect data and achieve the CIA objectives, multiple design principles exist. They help developers, architects and solution providers to create secure systems by design and avoid common mistakes when working on sensitive data. By using design principles and standards, misunderstandings can be prevented and compliance increased [26]. These principles are later used to design, build and secure the SSC. The following security principles are based on *Writing secure code* and *Security by Design Principles* [86, 26].

Minimize Attack Surface Area

Adding a new feature or component means adding an additional attack surface for attackers. This increases the overall risk of the system being compromised. The goal in designing a secure system is minimizing the overall risk by reducing the attack surface. An example would be adding a search feature to a web application. The search function could include a SQL injection vulnerability, which could lead to information leakage (confidentiality) or data modification (integrity). To reduce the likelihood of an attack, the search functionality could be allowed for authenticated and authorized users only and

input data validation could be added. To completely eliminate the attack surface area, the feature could be removed.

Establish Secure Defaults

When the application or system is shipped to the client, the components should be configured as secure by default. This means that all configurations should be set to values which provide the least attack surface and highest security standards possible. Clients might then be allowed to change configuration to a lower security level, while knowing they increase the risk. Secure defaults also means disabling features that are not commonly used and enabling them when needed. This can also have a positive impact on performance of a component. An example for a secure default would be a password policy, which is by default set to a complex password requirement. Administrators might be able to simplify this policy to improve the login experience for end users, with the cost of increasing the risk of compromised accounts.

Principle of Least Privilege

This principle recommends reducing privileges of each entity to a minimum which is required to complete their work. These privileges could be for example access to resources, other components or to specific files on the filesystem. If a vulnerability was discovered in a component, damage can only be done in this context, not in context of a higher privileged component. To integrate this principle, additional planning is required. It needs to be documented, which component or user can access which resources and why. Another example for this principle would be that an end user is allowed to use a specific component, but is not allowed to change administrative settings.

Principle of Defense in Depth

The *Principle of Defense in Depth* recommends to implement a defense mechanism as if it would be the last instance and no other protection mechanisms are in front. Adding controls to reduce risks in multiple ways is more effective than a single regulating control or even completely relying on external defense mechanisms. This reduces the likelihood of a single point of failure and vulnerabilities are getting more unlikely and harder to exploit. An example would be an administrative page which is protected by an authentication / authorization mechanism, as well as audit logging. This reduces the likelihood of an anonymous attacker gaining access to this page, since he would have to bypass all mechanisms to stay undetected.

Fail Securely

The *Fail Securely* design principle recommends that if a component fails, it should not disclose data which normally would not be disclosed. It should just show an error message and log the rest into a different channel. If a component fails and prints out too much information, this could create new attack vectors for an attacker or leak sensitive information. In Listing 3.2 an example is shown, which makes the user an admin per default. If an attacker could force the functions `codeWhichMayFail` or `isUserInRole` to fail, the attacker would be able to bypass the authorization mechanism.

Listing 3.2: Failing Insecurely ([86])

```
isAdmin = true;
try {
  codeWhichMayFail();
  isAdmin = isUserInRole(  Administrator  );
}
catch (Exception ex) {
  log.write(ex.toString());
}
```

Don't Trust Services

Don't trust services means that all external third party components and partners should be treated as untrustworthy. They most likely have other security policies and cannot be controlled. They could be compromised and send malicious input to the component. In general, every data received from an external system could be an attack. This design principle is related to *Principle of Defense in Depth*, since the assumption that every input is not to be trusted is also a defense mechanism. An example for this principle would be a third party API which provides some kind of reward points. Every input received from the API should be checked and sanitized before displaying it to an end user.

Separation of Duties

A mechanism to prevent fraud is the definition of different roles for different actions. For example the entity who carries out the action should be different from the entity that approves or monitors the action. This means that an administrator who maintains the infrastructure and database of the shop system should not be able to buy from the shop, since he could abuse his privileges to buy items in the shop for free.

Avoid Security by Obscurity

Securing a system by hiding implementation details or implementing own mechanisms which are already covered by standards is bad security practice, since they are more likely to fail. The principle *Avoid Security by Obscurity* is similar to the *Kerckhoffs's principle* [83], which states that a cryptosystem has to be secure if everything is known about the system, except the key. Key system's security should not rely on hiding information such as source code but instead use principles like *Principle of Defense in Depth*, fraud and audit controls or secure password policies. It should be assumed that an attacker knows everything an administrator knows and has access to all source code and designs. By using this strategy, additional mechanisms are implemented to secure the component which reduces the overall risk.

Keep it Simple

By keeping implementations and architectures simple, failures and errors can be reduced. It is easier to keep an overview of a simple architecture and things are easier to maintain and control. Complex approaches also increase the attack surface, which increases the overall risk of an attack.

Fix Security Issues Correctly

When a security issue is identified, it needs to be fixed correctly to prevent further misuse. If the issue is involved in multiple components, all other related components need to be tested as well. For example it is assumed that a flaw has been found that one user can hijack another user session by modifying the session cookie. If the cookie handling code is also used in other components, all components need to be tested after a fix has been issued.

3.1.3 Attacks

This section explains the main attacks on architectures and operations. These attacks are later used to identify potential threats in the threat analysis process. The attacks are based on *Secure coding: principles and practices* [31].

Man-in-the-middle Attack

A man-in-the-middle (MITM) attack is possible if an attacker is able to intercept network traffic between two hosts. He is then able to masquerade as one of the parties and manipulate data exchanged between the two hosts. As a defense mechanism, compo-

nents should implement cryptographic algorithms such as transport layer security (TLS), authentication, session checksums or shared secrets, as for instance cookies.

Race Condition Attack

A race condition attack, also known as Time-of-Check-to-Time-of-Use-Problem (TOCTTOU) involves multiple processes which run in parallel. It could be for example possible to replace an existing file which is already validated and used by another process to bypass initial security checks by the first process. To defend against a race condition attack, developers need to be aware of differences between atomic an non-atomic operations and avoid non-atomic operations if possible.

Replay Attack

A replay attack is a network attack which repeats or delays data transmission. It tries to fool participants into thinking they have successfully completed a protocol transaction. An example would be the transmission of a password hash, which was used for authentication and has been eavesdropped by an attacker in the network. After saving the transmission packets, the attacker can start another authentication request and resend the saved packets containing the password hash to successfully authenticate. Possible countermeasures are session identifiers, nonces or timestamps which should be integrated in the authentication process.

Sniffer Attack

Sniffers are tools which allow to monitor network traffic. They can be used by administrators to diagnose networks, but also by attackers to record sensitive information which is transmitted in clear text. To defend against a sniffer attack, switches and routers need to be configured correctly. On application level, TLS can be used to transmit sensitive data in encrypted form.

Session Hijacking Attack

A session hijacking attack exploits session control mechanisms. For example a web service needs cookies to identify a user across multiple HTTP requests. If an attacker is able to steal this cookie, he could use it to establish a valid session and gain access to the web server.

Denial-of-service Attack

By sending a high amount of traffic to an application, host or network, an attacker could make a service unavailable for legitimate users. To defend against a *Denial-of-service*

Attack (DoS), architecture and network need to be planned to moderately use resources. CPU, file and memory limits should be applied so that one application is not able to overload the whole system. A process is needed to monitor resource usage and block specific incoming traffic if a DoS attack is happening.

Default Accounts Attack

Components are often shipped with default credentials to simplify initial configuration and installation. Default credentials are an attack vector for attackers, since many lists with default credentials for all kinds of software already exist. To prevent *Default Accounts Attack*, default accounts need to be removed in the initial configuration process. Also processes which automatically test the architecture for default credentials can be implemented.

Password Cracking Attack

*Password Cracking Attack*s are a possibility to get access to protected systems which require credentials. Tools can be used to guess username and password combinations, either by randomly generating passwords (brute force) or using lists with predefined passwords (dictionary attack). To prevent this attack, users need to be trained to use strong passwords or password policies need to be defined. Also using additional factors, like for instance biometric characteristics or additional hardware can help to reduce the risk of this kind of attack.

3.2 Threat Modeling

The basic security objectives and attacks have been defined in the last chapter. In this chapter a process is defined to find the main threats and to protect information against those threats.

Threat Modeling is a structured approach to identify threats, risks and mitigations for a given component or system [107]. It uses abstractions which aid to help thinking about risks [107]. The main reason for threat modeling is the fact, that secure systems cannot be built until potential threats for the system are identified and understood [86]. In general a threat model consists of two models: a model of what is built and a model of threats [107]. The first model is a detailed documentation of the product itself, containing its external dependencies, the assets and entry points [107]. This information can be used to determine weak spots, vulnerabilities or risks early and to better understand security requirements [107]. A threat model can be integrated into an existing software development lifecycle (SDLC) to increase overall security from the beginning [25]. The model of threats is a

detailed list of what can go wrong, once the product has been built [107]. The list of threats can also be categorized and ranked, which can be used to implement countermeasures and mitigate those risks [25]. Multiple types of threat modeling exist [27]:

- Software Centric
- Security Centric
- Risk Centric

A software centric approach prioritizes threats upon their effect on functional use cases or impact on for example reliability of the software [27]. Security centric threat modeling ranks upon how easy it is to exploit a threat or the technical impact on the product [27]. A risk centric threat model (like for example PASTA) prioritizes after information owners, business or other stakeholders [25]. In this work a security centric threat modeling process is used to identify threats on a SSC, since it has a technical focus. A threat modeling process can be divided into multiple steps:

3.2.1 Step 1: Model System

The first step is necessary to gain an overall understanding of the system and how it interacts with external entities [107]. It is a structured approach to gain as much information as possible about the product [107]. To identify how, by whom and in which circumstances the product is used, use cases can be created as a first step [25]. Also entry points, assets and trust levels have to be identified and described in an overall threat model documentation [107]. To easily see how data is passed through the application and through trust boundaries, data flow diagrams (DFD) can be created [25]. To improve the models further and to get an overview of potential attack vectors, trust boundaries should be added [107].

3.2.2 Step 2: Identify Threats

The second step uses a methodology, as for instance STRIDE from the attacker's point of view or *Application Security Frame* (ASF) from the defender's point of view, in combination with the created models from the previous step to find possible targets and threats [25]. Afterwards a ranking methodology, like DREAD can be applied to calculate the level of risk those threats impose [25].

3.2.3 STRIDE

SRIDE is used in this work to find potential threats. It is an acronym which aims to help to identify threats imposed to a product. The description of STRIDE and the explanation of each letter is based on the sources [107, 86]. The acronym STRIDE stands for:

- **S**: Spoofing (pretending to be someone else)
- **T**: Tampering with Data (data is manipulated)
- **R**: Repudiation (actions are audited and a user can be identified)
- **I**: Information Disclosure (user gets more information than needed)
- **D**: Denial of Service (data is not available)
- **E**: Elevation of Privilege (user is able to gain more privileges)

It was invented by Loren Kohnfelder and Praerit Garg in 1999 [107]. It was designed to aid people identifying and enumerating types of attacks which threaten systems and things that could go wrong [107]. It is not intended to be a categorization method, since many threats cannot be assigned to a single category [107]. In the following paragraphs, the meaning of each letter in the acronym is explained in detail.

Spoofing

In general, spoofing means an attacker pretends to be something or someone different than himself. *Spoofing* can be divided into three basic categories: spoofing a file or process, spoofing a machine or spoofing a person. **Spoofing a file** or process can be achieved by creating a malicious file which has the same name and attributes as the original file, tricking an application or person into executing it. An example would be renaming a file to a common name, such as `sshd`. **Spoofing a machine** is possible on multiple layers of a network stack, as for example spoofing ARP requests on layer 2, IP addresses on layer 3, or DNS packets on layer 7. After a machine has been spoofed, it is possible to act as a MITM instance and modify network communication. **Spoofing a person** can be achieved by using someone's credentials which have for example been stolen due to a phishing attack. This could then be used to initiate further attacks with using the privileges of the spoofed person.

Tampering with Data

Tampering with data means that something is modified in a malicious way. Typically this happens on local disk, in memory or in the network. An example of a local file is

a configuration file, which could be modified to lower the encryption strength or allow anonymous access to a component. If a bug in an application is found, sensitive data in memory could be modified or stolen. A network attack could involve manipulating data on the transport way or redirecting traffic to another host machine which is in control of the attacker.

Repudiation

Repudiation is the act of claiming that a person (not limited to attackers) did not do something even if they did. This often appears in the business layer, which is above the application layer. An example would be that someone claims he did not click on the email attachment after a malware infection. Another example would be, a person claims that he or she did not accept a package from UPS, although it has been delivered by the postman. Nonrepudiation could be achieved by using processes to log, retain and analyze all events which happen throughout the system.

Information Disclosure

Information disclosure happens when an attacker receives information which he is not authorized to see. This could occur if he has access to processes, data stores or is able to analyze data flows in the network. Processes can leak information like memory addresses, which could be used in a later attack to bypass security mechanisms, such as address space layout randomization (ASLR). Also sensitive data could be leaked in error messages and stack traces, for example credentials to access a database. Data stores as for instance databases, swap files, temporary files or hardware devices like USB devices could also contain sensitive information. They should be protected by setting correct permissions and adding additional security mechanisms, like for example encryption. Data which is transmitted unencrypted over the network could also lead to information disclosure.

Denial of Service

As already described in Paragraph *Denial-of-service Attack*, a denial of service (DoS) attack is taking all resources of a service to make it unavailable for others. Those resources could be memory, CPU, disk space or network resources. Two categories of DoS attacks exist: persistent and distributed denial of service (DDoS). Persistent attacks are for instance cronjobs which survive reboots and create endless loops to consume all CPU resources. DDoS attacks are done by sending as much traffic as possible to an application or host to make it inaccessible for others.

Elevation of Privilege

An *Elevation of Privilege* attack means an attacker is able to do something he is not authorized to do. This could be for example executing code as admin user being logged in as a standard user. Two main ways exist to achieve a privilege escalation: corrupting processes or bypassing authorization checks. Corrupting a process means sending invalid input to a process which cannot be interpreted correctly and leads to a buffer overflow. This could give an attacker control of the application flow and allow him to run custom code on the host. Bypassing authorization could be possible due to missing authorization checks or bugs in the authorization component.

3.2.4 Step 3: Address Threats

In step three, the found threats from the previous step are addressed and mitigations are defined [107]. Different mitigation strategies are listed in Table 3.1 ([107]). The decision which mitigation strategy should be used for which threat is based on multiple factors, such as costs of transferring the threat to another party, the likelihood of its occurrence or costs for avoiding the threat [107]. As a result, a complete list with all threats and calculated risk levels mapped onto the mitigation strategies is created [107]. The final resulting document is the threat model for the given product [25].

Method	**Explanation**
Mitigating Threats	Make it harder to take advantage of a threat, for example by adding an additional layer of security.
Eliminating Threats	Eliminate the threat completely, for example by removing the feature which is involved.
Transferring Threats	Transfer risk to someone else, for example an insurance.
Accepting Threats	Do nothing.

Table 3.1: Mitigation Strategies

3.2.5 Step 4: Validate

The last step is to validate the work which was done in the previous steps [107]. The initial model and the threats have to be reviewed and updated in an iterative process [107]. For example complete data flows can be added based on the additional information gained in step two or three [107].

3.2.6 Detailed Approach

The approach in this work follows the described threat modeling process from the last chapters. First, a SSC is described and models are created to show the overall attack surface and trust boundaries. Afterwards, threats are identified using attacks described earlier as an aid. Based on the likelihood and impact the final risk levels are calculated afterwards. To get an overview of the risks exposed to each component, the values *low*, *medium* and *high* can be assigned to the likelihood and impact. The meaning of the values for likelihood are explained in Table 3.2. The values for impact are explained in Table 3.3. The final risk value can be calculated using the 3x3 matrix shown in Figure 3.1. This matrix is used to illustrate the level of risk which can then be discussed. The meaning of the calculated risk values are defined in Table 3.4. After calculating the risk, treatments are assigned, countermeasures and methods are explained and the residual risk is discussed. Step four is done implicitly in this work by iteratively reviewing the found threats and comparing them to the countermeasures which have been discussed.

Value	**Explanation**
Low	Unlikely, there is a small possibility it might occur.
Medium	It is likely to occur as there is a history of casual occurrences.
High	The threat is expected to occur, since it happened frequently in the past.

Table 3.2: Values for Likelihood

Value	**Explanation**
Low	Minor financial or reputational impact.
Medium	Moderate financial or reputational impact.
High	High financial or reputational impact.

Table 3.3: Values for Impact

Value	**Explanation**
Low	The risk is acceptable as it is unlikely to occur or cause damage.
Medium	The risk can cause damage, and should be addressed.
High	Not acceptable, likely to cause damage and it needs to be addressed.

Table 3.4: Risk Level Values

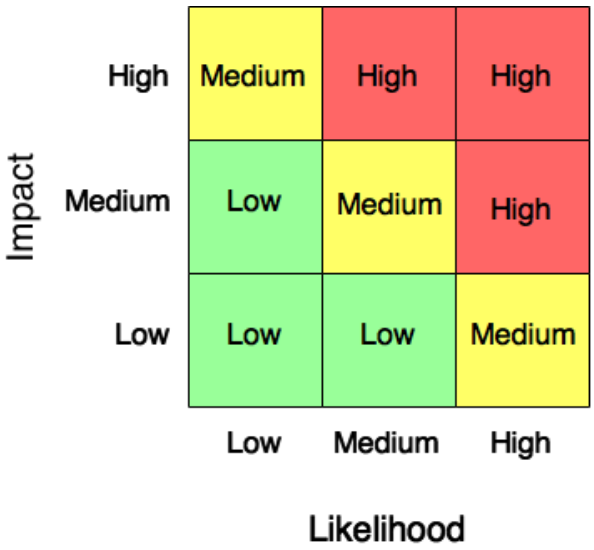

Figure 3.1: Risk Matrix

4 Modeling a Software Supply Chain

This chapter describes and models a SSC. Since the SSC is built on Docker as base technology, this work focuses on technical aspects of a SSC. Therefore topics as for instance integration into business processes, key management strategies or compliance rules are out of scope. Topics such as logging and monitoring processes are also out of scope, since they should be implemented organization-wide, not limited to SSCs. This chapter is the first step of the threat modeling process. As described in Chapter *Software Supply Chain*, a SSC consists of the following parts: *sources and dependencies*, *build systems and engineers*, *network*, *application repository* and *deployed system*. This chapter's structure is based on the parts of the SSC.

4.1 Overview

A general overview of environments of a SSC and its borders can be found in Figure 4.1.

The local environment is the laptop or stationary machine for each developer, which provides tools to maintain the application. The development environment contains the central VCS, which is used to share code among developers. To ensure high code quality and continuous builds, a CI pipeline is required, which is mainly located in the build environment. The central part of the CI pipeline is the build server running an automation software. The testing environment runs the application in a Docker Swarm which is similar to the production environment, but access is limited to the testing teams (labeled as *Tester*). It is a flattened clone of the production environment and is used to test the latest features within a production-like environment. The database data is copied from the production database, but typically anonymized. This means that the real data cannot be reconstructed. The production environment is the main container cluster which provides access to the application for the end user. It uses real data and is typically scaled across multiple nodes behind a load balancer to handle incoming traffic bursts.

4.2 Sources / Dependencies

In this chapter all topics related to source code and dependencies are explained. These are the source code of the application itself and the Docker images which are used to run

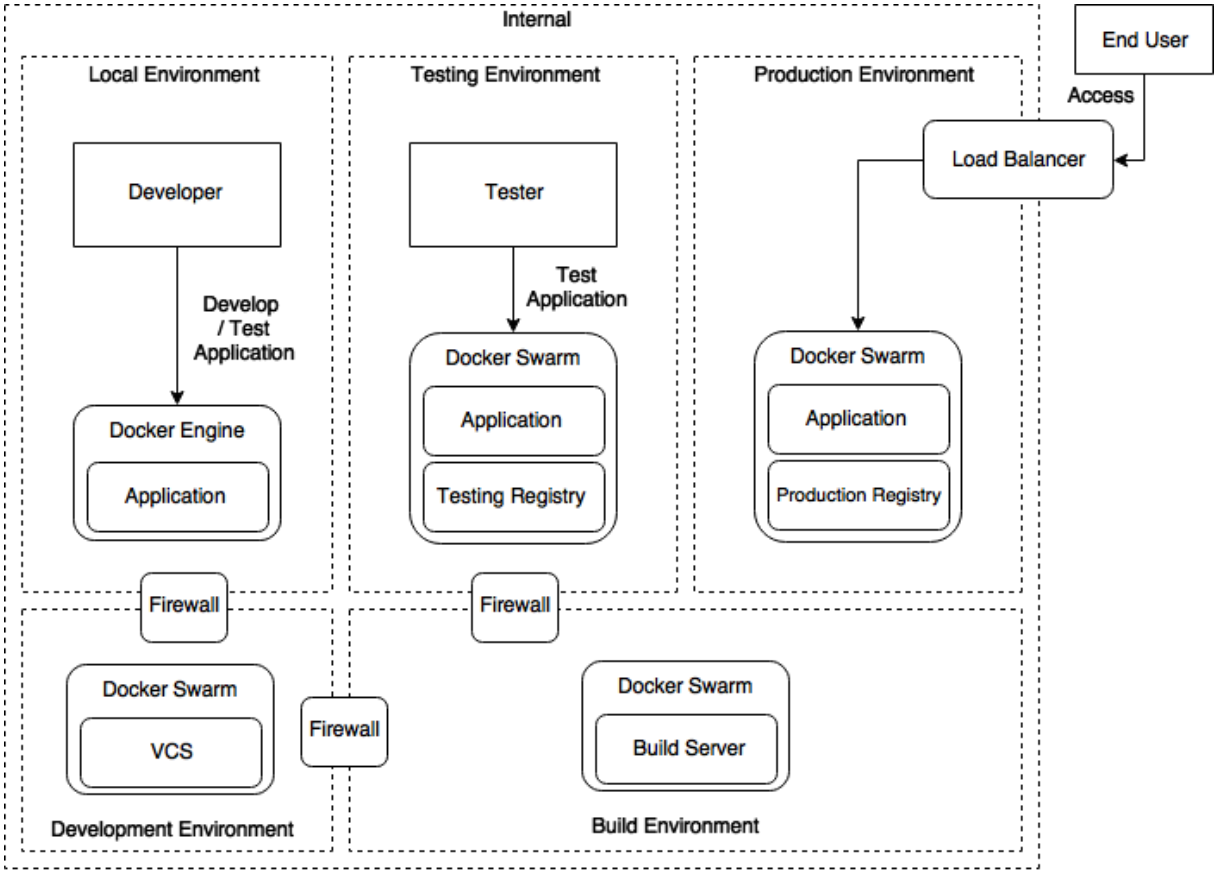

Figure 4.1: General Overview

the application in a container cluster.

4.2.1 The Source Code

The source code is the main asset in the SSC and needs to be protected using the CIA objectives. The application which is exemplary used is shown in Figure 4.2. It consists of a single container which provides a web application on port 80. For this work a small HTML document is used, which shows a simple *Hello World* (Attachment 1). As shown in Figure 4.2, the end user can access the application container on port 80. The application Docker image installs some additional dependencies, such as `nginx` onto the OS base image.

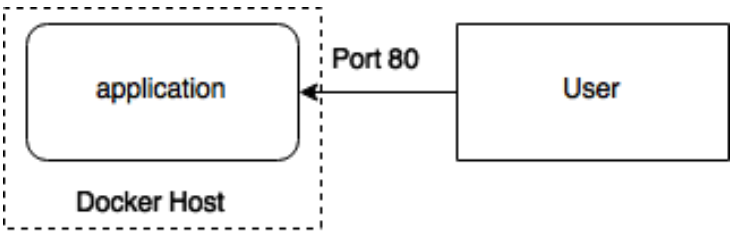

Figure 4.2: Application Overview

The complete application can be found in the attached source code in the `application/` folder. To build and run the application, Docker Compose can be used. The file `docker-compose.yml` (Attachment 9) describes a single service called *application*. The code within the containers is located in the folder `/var/www/app`. The `docker-compose.yml` file is used for development, this is why the local code folder is mounted as a volume into both services. This allows the developer to change the source code and see the changes directly in the browser, without the need to rebuild the containers. The `docker-compose.prod.yml` (Attachment 8) has the same structure as the `docker-compose.yml`, but uses the `Dockerfile` which resides in the `docker/production/application/` folder (Attachment 4). Those *Dockerfile*s additionally copy the whole `code/` folder into the final image, to make sure it has access to the source code when running in the testing or production environment.

4.2.2 Docker Images

Docker images provide a consistent environment for an application or service and bundle all required dependencies. These images can be used to run a complete application on different infrastructures, such as a local *Docker Engine* installation for development or a productive *Docker Swarm*.

OS Base Image

The OS base image is used as basis for all further Docker images. It is used for the following components:

- The sample application
- The build server
- The VCS server
- The Registries

It provides the basic OS layer which is comparable to for example the *Ubuntu* [73] or *Debian* base image [46]. It needs to provide basic OS functionality, as for instance a network stack. It should be as small as possible to save disk space and contain only the required dependencies to reduce the overall attack surface. The base image should be built from *SCRATCH* ([69]) in a separate CI pipeline, which is out of scope of this work since this image should be built organization-wide and not only for this SSC. Instead of the OS base image, this work uses images from *Docker Hub* to run the application.

Application Docker Image

The application image is used to containerize the sample application. It is based on the OS base image and contains additional dependencies like `nginx` to run the frontend. `Nginx` runs in background and listens on port 80.

Build Server Docker Image

The build server image is based on the OS base image and contains the automation software plus additional dependencies, as for example *Apache Ant* or *Python*. In this work, *Jenkins* is used, as explained in Chapter *Build Server*. More dependencies can be required if for example additional security checks are applied on the source code. To build and deploy the testing image, access to a *Docker Engine* is required. In this work the Docker socket (`/var/run/docker.sock`) from the build server is mounted into the *Jenkins* container, to allow the container to get access to the host's *Docker Engine*. The `docker-compose.yml` file for this example (Attachment 12) can be found in the `buildserver/` folder in the attachment. To run the Jenkins server, `docker stack deploy --compose-file docker-compose.yml jenkins` needs to be run.

VCS Docker Image

The VCS image is also based on the OS base image and contains additional software and dependencies to provide a VCS server. Software that might be used is for example *gitolite* ([11]) or *GitLab* ([9]). As described in Chapter *VCS*, *gitolite* is used in this work. The `docker-compose.yml` file for the *gitolite* repository can be found in the attached source code in the `vcs/` folder (Attachment 13). To start the sample *gitolite* repository, the command `docker stack deploy --compose-file docker-compose.yml gitolite` can be run in a Docker Swarm cluster.

Registry Docker Image

In this work the *Docker Registry* is used as described in Chapter *Registry*. The `docker-compose.yml` file for the registry can be found in the attached folder `registry/` (Attachment 14). To run the registry, `docker stack deploy --compose-file docker-compose.yml registry` has to be executed in a Docker Swarm.

4.3 Build Systems / Engineers

This section describes all topics related to build systems or engineers. These are for example the general development, build and deployment processes, local environments which are used to develop applications, or the build server.

4.3.1 The CI Pipeline

Development Process

Developers regularly push their changes into the central VCS, as shown in the general development workflow in Figure 4.3. The VCS then notifies the build server to trigger a pull, test and build. This is described in more detail in Chapter *Build Process*. After the tests and build have been finished, the developer receives a notification whether it was successful or not. This is done by a notification service, such as be Slack [76] or email. To provide a better overview, the deployment step was omitted in Figure 4.3.

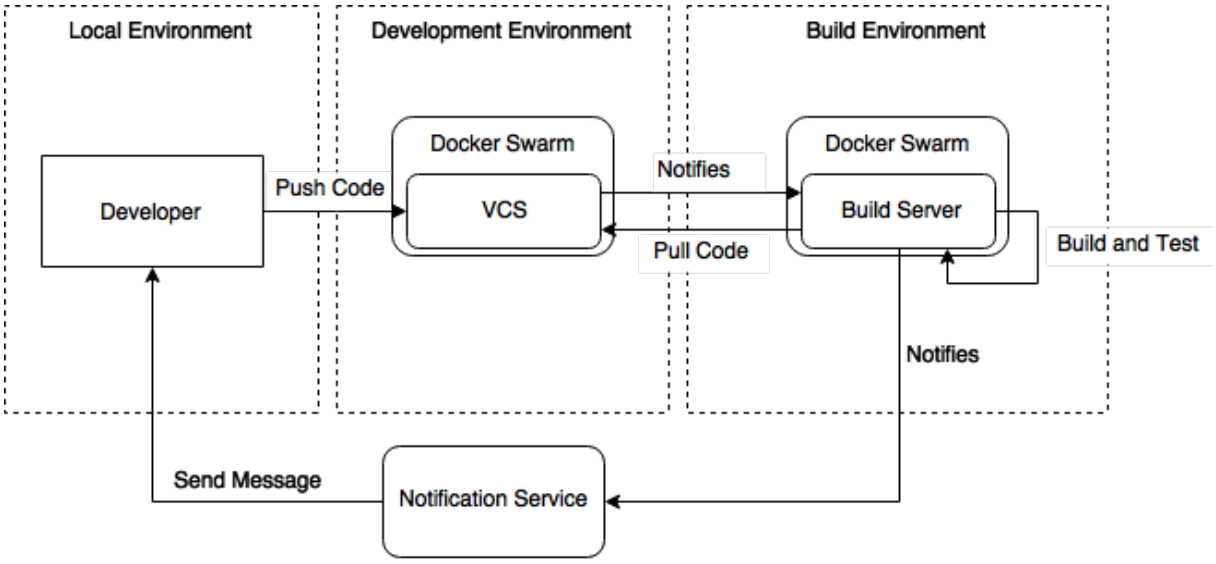

Figure 4.3: Development Process

Build Process

When a push into the VCS happens, the build server is notified by the VCS. When receiving such a notification, the build process is triggered, as shown in Figure 4.4. It starts by pulling the latest changes out of the VCS. Next, tests are executed, which could for example be unit, integration or smoke tests. Afterwards, source code analysis tools can be run, as for instance *phpcs* ([85]) or *phpmd* ([102]) in case of a PHP application. After running the tests and checks, the results are parsed and interpreted by the automation software. If one of the tests failed or the source code analysis has thrown warnings above a certain threshold, the build server stops and sends a notification. This notification is sent to the developer who was responsible for the last push. He then needs to fix the source code until the tests work again.

If the tests and source code analysis tools successfully finished their work, the Docker images for the application are built. In general, `docker build -t <tag> -f <Dockerfile>` can be executed to build a Docker image. This command parses the `Dockerfile`, builds

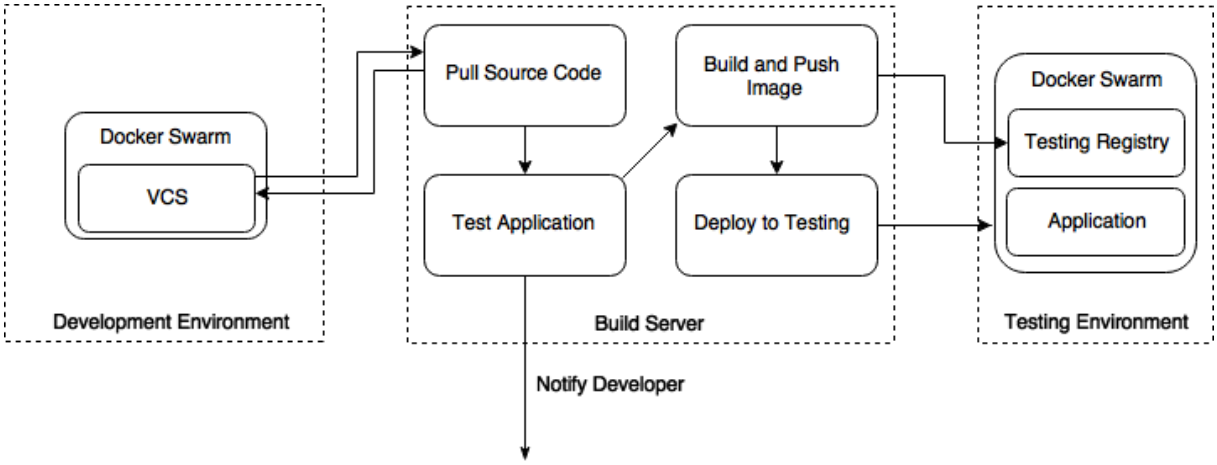

Figure 4.4: Build Process

a new Docker image and copies the source code into the new image. *Docker Compose* simplifies the build process by providing all necessary information in a single `docker-compose.yml` file. To build the application, `docker-compose build` can be run in the `application/` folder. To build the image for the testing environment, the file `docker-compose.prod.yml` is required. It extends the regular `docker-compose.yml` file by copying the source code into the image. The necessary commands are bundled in the attached `build.sh` file (Attachment 6), which automatically uses the `docker-compose.prod.yml` file.

Deployment to Testing

When the build process has been completed, the image can be pushed into the testing registry. This step is included in Figure 4.4. Pushing the image into the testing registry is achieved by running `docker-compose push` and using the `docker-compose.prod.yml` file. The combined command can be found in the `push.sh` file (Attachment 11). If automated deployment is enabled, the automation software calls the deployment script, which tells the testing container cluster to pull and run the latest image version. In general, a deployment works by running `docker stack deploy --compose-file <docker-compose file> <app_name>` on the host machine. To execute the command, the build server needs SSH access to the testing swarm. The deployment command is included in the `deploy.sh` script (Attachment 7). It copies the current `docker-compose.prod.yml` onto the testing host and runs `docker stack deploy --compose-file docker-compose.prod.yml application`. After deploying the latest image to the testing cluster, the testers are now able to test this version.

Deployment to Production

After the application image has been tested by the testing team, it needs to be deployed to the production environment. This process is shown in Figure 4.5. An operator pulls the image from the testing registry and pushes it into the production registry. After the image has been pushed, the operator needs to run `docker stack deploy --compose-file docker-compose.prod.yml <app_name>` on the production swarm to pull and roll out the new containers.

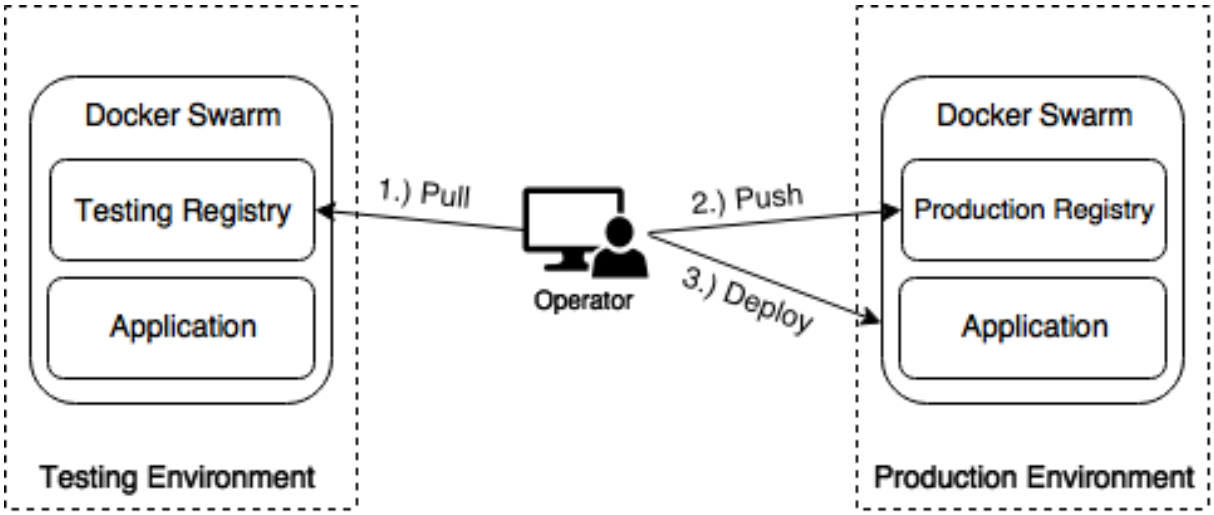

Figure 4.5: Deployment to Production

4.3.2 Local Environment

The local environment is used to work on the source code of the application. It represents the computer or laptop of each developer. It needs to support developers at their daily work and make development as easy as possible. If possible, all developers should work on the same operating system and use the same toolset to make the whole process as consistent as possible. To build and run images locally, *Docker Engine* needs to be installed. The software versions used to run the application have to be the same as in testing / production to avoid errors based on different versions of the server software. This is achieved by using `Dockerfile`s which are similar to those which are used for production (Attachment 2). On the contrary to the testing / production environment, the local environment should offer additional features like debugging or syntax checking. Debug information of the application should be limited to this environment to avoid unwanted information leaks. The database data used while developing should be sample data or a fully anonymized production dump.

4.3.3 Build Server

The build server plays a central role in the SSC. As the name implies, it is used to regularly build, test and deploy the application. Its central part is an automation software which runs as a container in a Docker Swarm and listens to changes in the VCS. If a change occurs, it automatically pulls the latest source code and processes it further. Some exemplary automation servers are *Jenkins* [24], *Travis CI* [113], *Buildbot* [94] or *Bamboo* [1]. In this work *Jenkins* is used as the automation server, as it is open source and one of the most distributed automation software with over 100,000 installations [82].

4.4 Network

In this section the communication of the components in the SSC is described. It is based on the general overview, which is shown in Figure 4.1. A developer needs to access the VCS to push the source code. The build server needs to communicate with the VCS to exchange notifications and the source code. It also needs to push the resulting Docker images into the testing registry. The end user needs to be able to access the application in the production environment. The complete and secured network layout is described in Chapter *Network Overview*, after the threat modeling process.

4.5 Application Repository

The application repository section contains all topics which are related to providing access to source code or to Docker images. These are the central VCS and different image registries.

4.5.1 VCS

The VCS is the central code repository for developers. It is run as a Docker container in a Docker Swarm. In this work, *gitolite* [11] (On Docker Hub: [50]) is used as a sample repository, as it provides the required functionalities like for example public key authentication.

4.5.2 Registry

In this work *Docker Registry* is used, as it provides the basic functionalities. Multiple registries are required:

- Testing Registry

- Production Registry
- Internal Registry

The testing and production registries hold Docker images for their environments. They run in the corresponding Docker swarm to limit access to the registries. In addition to the swarm, the testing registry should be accessible for the build server and operators to allow pushes. The production registry access should be limited to operators for deployment. *Internal* is a general purpose registry, which is used to save additional images like for the VCS or build server. Since the internal registry does not only contain images for the SSC and can be used organization-wide, it is out of scope of this work.

4.6 Deployed Systems

In this section the deployed systems are described. These are mainly container clusters which are used to run application containers. Since Docker abstracts away the infrastructure and *Docker Engine* runs on different operating systems, the base infrastructure is out of scope of this work.

4.6.1 Docker Swarm

Container clusters are required to run and schedule an application image onto multiple nodes. An overview of required container clusters and the applications which run in those clusters is shown in Table 4.1.

Environment	**Application**
Development	VCS
Build	Build Server
Testing	Sample Application, Testing Registry
Production	Sample Application, Production Registry

Table 4.1: Required Container Clusters

Since this work is based on Docker, Docker Swarm is used as container cluster (both denominations are used as synonyms, as well as *swarm*). *Docker Engine* needs to be installed on all nodes to form a swarm.

5 Threat Analysis

This chapter is step two in the threat modeling process. First, the main components, users, dataflows and trust boundaries are listed based on Chapter *Modeling a Software Supply Chain*. Afterwards the main components of the SSC are analyzed and threats are identified using STRIDE.

5.1 Components, Users and Trust Boundaries

This chapter lists the main components, users, trust boundaries and dataflows which are required for a SSC. All are labeled with a unique ID in ascending order to later reference them.

5.1.1 Exclusions

Table 5.1 shows components which are excluded as well as the reasons for excluding them.

Excluded Component	**Reason**
Sample Application	As the name states, this is an exemplary application which does not contain sensitive data. The threat model of the sample application is out of scope of this work, since analyzing the sample application does not provide an additional value for the SSC.

Table 5.1: Excluded Components

5.1.2 List of Components

In Table 5.2 the main components of the SSC are listed and a unique identifier is assigned.

5.1.3 List of Users

In Table 5.3 the main users which are part of the SSC are listed and a unique identifier is assigned.

Component	ID
Docker Images	C1
Local Environment	C2
VCS	C3
Build Server	C4
Registry	C5
Docker Swarm	C6

Table 5.2: List of Components

User	ID
Developer	U1
Tester	U2
End User	U3
Infrastructure Operator	U4

Table 5.3: List of Users

5.1.4 List of Dataflows

In Table 5.4 all dataflows of a SSC are listed and a unique identifier is assigned. It is based on the description of Chapter *Network*.

Source	Destination	Protocol	Description	ID
Local Environment	VCS	SSH	Push and pull source code	D1
VCS	Build Server	HTTP / HTTPS	Notify change	D2
Build Server	VCS	SSH	Pull source code	D3
Build Server	Testing Registry	Docker Registry HTTP API V2	Push image	D4
Build Server	Testing Docker Swarm	SSH	Deploy	D5
Tester	Application (Testing)	Any	Test application	D6
End User	Application (Production)	Any	Use application	D7
Infastructure Operator	All	SSH	Build and maintain infrastructure, manage credentials.	D8

Table 5.4: List of Dataflows

5.1.5 List of Trust Boundaries

In Table 5.5 relevant trust boundaries of a SSC are listed and a unique identifier is assigned. In Figure 5.1 the boundaries are shown visually. Trust boundaries are defined according to the data they contain. The local and development environment contain the source code of the application but no real user data. The build server processes the source code, and additionally has access to the testing registry and testing swarm. The testing environment hosts the application for the testers. The production environment is a completely separated trust boundary since real data is used.

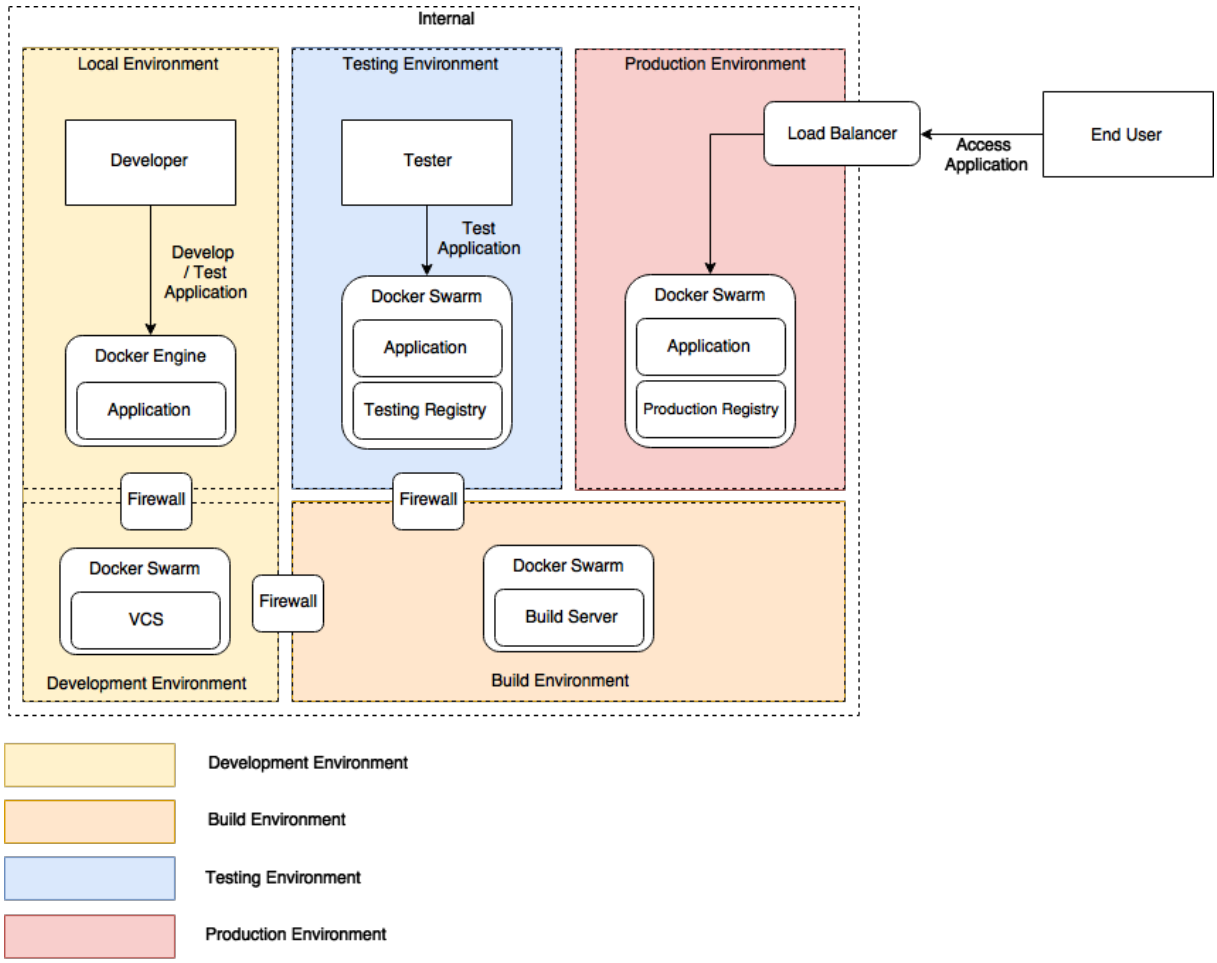

Figure 5.1: Trust Boundaries

Components and Users	ID
Development Environment	TB1
Build Environment	TB2
Testing Environment	TB3
Production Environment	TB4
External / End User	TB5

Table 5.5: List of Trust Boundaries

5.1.6 Data Flow Diagram

The data flow diagram for a SSC is shown in Figure 5.2. All data flows from Table 5.4, trust boundaries from Table 5.5 and users from Table 5.3 are included, except dataflow D8, which accesses all environments and user U4 for the same reason.

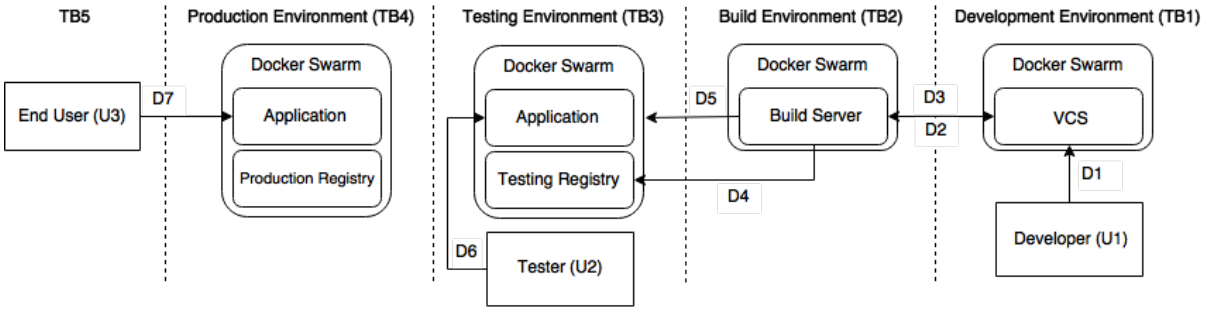

Figure 5.2: Data Flow Diagram

5.2 Threat Modeling

This chapter analyzes the components listed in Chapter *Components, Users and Trust Boundaries* and identifies threats. To find threats, STRIDE in combination with the defined attacks from Chapter *Attacks* is used. The base layout for the threat analysis for each component is a table consisting of rows representing each letter of the word STRIDE. Additionally, each threat is labeled with a unique identifier (starting with T), which will be used as a reference in later chapters.

5.2.1 Exclusions

To simplify the threat models, common threats, as for example default credentials are excluded. A list of excluded threats and the reasons are listed in Table 5.6.

5.2.2 Components

Docker Images (C1)

The threat model for the Docker images is shown in Table 5.7.

Local Environment (C2)

The threat model for the local environment is shown in Table 5.8. Threats such as malware or local privilege escalation vulnerabilities are not listed, since they belong to the excluded category *OS level threats*.

Excluded Threat	**Reason**
Default credentials	Default credentials are excluded, because this threat can be addressed by setting up the infrastructure in an automated process and should be taken care of in each component.
OS level threats	OS level threats like for example SSH or bash vulnerabilities are excluded to reduce the focus on the main components of a SSC. On each OS, base hardening mechanisms should be applied.
Disabled (audit) logs	Logs should be enabled everywhere and a central logging service should be established. Since a central logging infrastructure is a separate mechanism, it is out of scope of this work.
Rogue internal	Rogue internals, as for instance a rogue operator are not a threat limited to SSCs. They should be mitigated organization-wide.

Table 5.6: Excluded Threats

S	-
T	* Backdoored images in public registries (T1.1) * Image forgery while transmitting (T1.2)
R	-
I	* Vulnerabilities (T1.3) * Backdoors (T1.1) * Hardcoded information (for example credentials or IPs) (T1.4)
D	* Vulnerabilities (T1.3) * Untested *latest* tag (T1.5)
E	* Privilege escalation vulnerability (T1.3)

Table 5.7: Threat Analysis for Docker Base Image

S	-
T	* Source code manipulation (T2.1)
R	-
I	* Credential leak (T2.2) * Source code leak (T2.3)
D	* Loss of credentials (T2.4)
E	* Vulnerability in development tools (T2.5) * Misconfigured Docker Engine (Socket exposed to network) (T2.6)

Table 5.8: Threat Analysis for Local Environment

VCS (C3)

The threat model for the VCS is shown in Table 5.9.

S	* Authentication / authorization bypass (T3.1)
T	* Unauthorized push (T3.2) * MITM while pulling / pushing source code (T3.3) * Destroying source code (T3.6)
R	-
I	* Vulnerability in software (T3.4)
D	* Too many interactions at the same time (T3.5)
E	* Vulnerability in software (T3.4) * Authentication / authorization bypass (T3.1)

Table 5.9: Threat Analysis for the VCS

Build Server (C4)

The threat model for the build server is shown in Table 5.10. The *OWASP Top 10 security risks for web applications* is used to analyze the web interface [99].

S	* Broken authentication or session management (T4.1) * Leaked credentials (T4.2) * Cross-Site Scripting (XSS) (T4.3)
T	* Unauthenticated access (T4.4) * MITM (T4.5)
R	-
I	* Injection attack (T4.6) * MITM (T4.5) * XSS (T4.3) * Leaked credentials (T4.2)
D	* Too many parallel builds (T4.7) * Vulnerability in software (T4.8)
E	* Broken authentication or session management (T4.1) * Vulnerability in software (T4.8) * Injection attack (T4.6) * Cross-Site Request Forgery (CSRF) (T4.9) * XSS (T4.3) * Host compromise with Docker socket (T4.10)

Table 5.10: Threat Analysis for the Build Server

Registry (C5)

The threat model for image registries in testing and production environment is shown in Table 5.11.

S	* Broken authentication (T5.1)
T	* Unauthenticated push (T5.2) * Image forgery while pushing / pulling (T5.3)
R	-
I	* Vulnerabilty in registry software (T5.4) * MITM while pushing or pulling images (T5.5)
D	* Too many interactions (T5.6) * Image use too much space (T5.7)
E	* Vulnerability in registry software (T5.4)

Table 5.11: Threat Analysis for the Registry

Docker Swarm (C6)

The threat model for the Docker Swarm which runs in testing and production is shown in Table 5.12.

S	-
T	* Network exposed Docker socket (T6.1)
R	-
I	* Network exposed Docker socket (T6.1) * MITM between master and nodes (T6.2) * Leaked credentials (for example for DB) (T6.3)
D	* Network exposed Docker socket (T6.1) * Container memory / CPU exhaustion (T6.4)
E	* Network exposed Docker socket (T6.1) * Container to host breakout (T6.5)

Table 5.12: Threat Analysis for the Docker Swarm

6 Securing the Software Supply Chain

Based on the identified threats, this chapter calculates the risks, discusses security mechanisms to mitigate, eliminate, transfer or accept those risks and assesses the residual risk. The discussion includes security design principles which were introduced in Chapter *Security Principles*. The general goal of this chapter is to show the residual risk of an attack on each component and possible solutions to secure information which is passed through the SSC regarding the CIA objectives.

6.1 Network Overview

Before evaluating the components, an overview of the network communication is given. The network layout is shown in Figure 6.1.

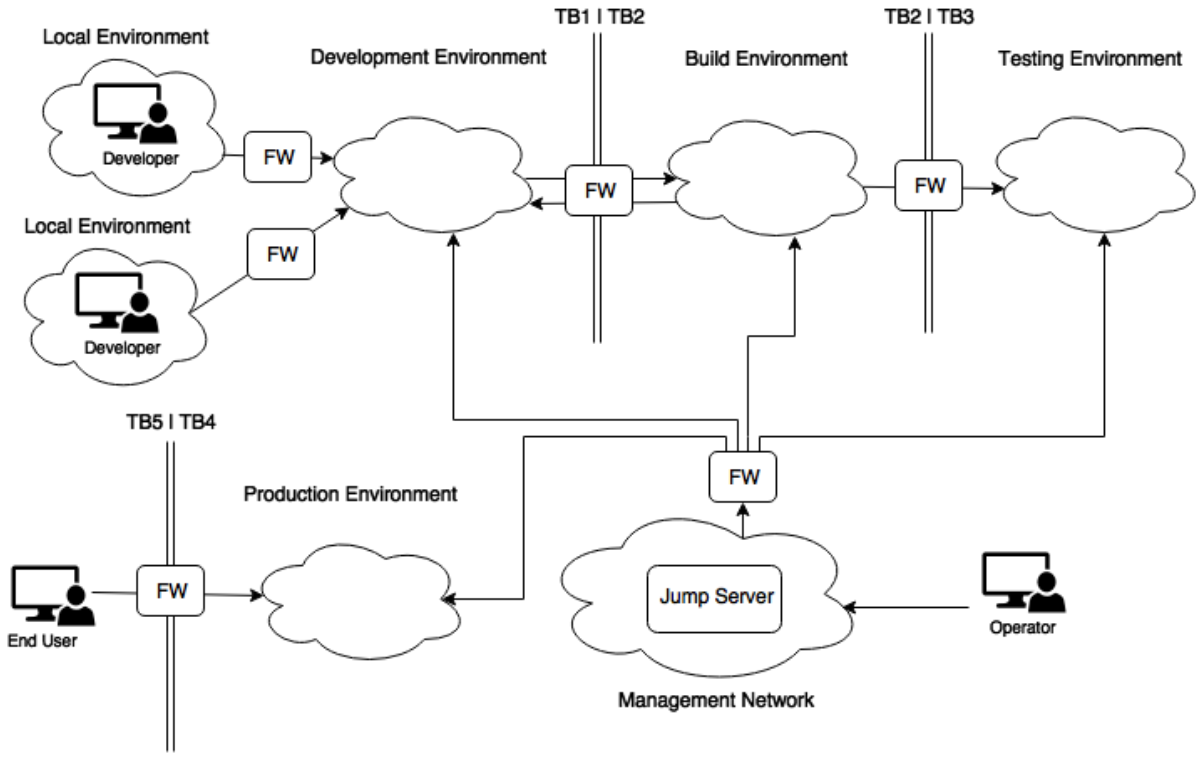

Figure 6.1: Network Overview

By creating logically separated subnets, each component can be isolated from each other and trust boundaries can be protected (*Defense in Depth*). Due to sensitive data, the

production environment should additionally be physically separated. The subnets should be able to communicate with each other through firewalls (labeled as *FW* in Figure 6.1). Those firewalls contain access control lists (ACLs), which allow minimal traffic which is defined in the dataflows in Chapter *List of Dataflows* (*Principle of Least Privilege*). The default rule is to deny all other traffic (*deny-all*). By using this concept, it is harder for an attacker to move from one component to the other (*lateral movement* [121]), even if he managed to infiltrate one of the components. Each cloud in Figure 6.1 represents a subnet. The trust boundaries from Chapter *List of Trust Boundaries* are marked with double lines. Each crossing from one trust boundary into another is secured by a firewall. The communication from the local environment to the development environment can be unidirectional, since the developers only need to exchange source code with the VCS. The VCS does not need to initiate a connection back. The communication between development and build environment has to be bidirectional, since the VCS needs to connect to the build server and vice versa. The communication from the build server to the testing environment is unidirectional, since the build server has to push the final Docker image. No connection back should be allowed, in case the testing environment has been compromised.

Additionally, a management network is required to allow operators to maintain the infrastructure (*Defense in Depth*). It needs unidirectional access to all environments. All management interfaces, as for instance SSH ports, should be able to communicate with the management network only. The management network contains jump servers, which are accessible by operators and contain keys to access the different environments. These keys are for example SSH private keys.

6.2 Components

6.2.1 Docker Images (C1)

Docker images form the base layer for many components of the SSC (for example the VCS, the build server or the testing and production application). To provide an overview, each found threat mapped to its likelihood, impact, risk and mitigation strategy is shown in Table 6.1.

Explanations

T1.1: The likelihood of backdoors is low if developers are constrained to use public image from official repositories only. The impact is high since if a backdoor stays undetected, an attacker would be able to gain access to production data.

T1.2: Image forgery is limited to attackers which have access to the internal network.

Threat	Likelihood	Impact	Risk	Mitigation
T1.1 (backdoor)	low	high	medium	eliminate
T1.2 (image forgery)	low	high	medium	eliminate
T1.3 (vulnerabilities)	high	high	high	mitigate
T1.4 (hardcoded information)	medium	medium	medium	mitigate
T1.5 (untested *latest* tag)	low	low	low	eliminate

Table 6.1: Risk Calculation and Mitigation (C1)

The impact is high since manipulating an image and for example adding a backdoor would give an attacker access to production data.

T1.3: Vulnerabilities are very likely to occur since every dependency can be affected. If an attacker is able to exploit the vulnerability, he could gain access to production data.

T1.4: Hardcoded credentials are likely to occur, especially in new teams or if a new developer joins a team. The impact is medium since an attacker needs access to a registry to read out the credentials of an image.

T1.5: An untested latest tag is unlikely to occur, since container clusters can be configured to not use them. If it still happens, an untested feature could be deployed to testing or production, which exposes a medium risk.

Mitigation Strategies

To eliminate threat T1.1 (backdoor) and mitigate T1.3 (vulnerabilities), a separate internal CI pipeline is required, which builds the Docker base image. The resulting image is then used as base image for further Docker images, as for example the VCS. Developers should be constrained to use this image as basis to build their custom images. If possible, the base Docker image should be built using *SCRATCH* ([69]) and only bundle the necessary dependencies, to reduce the overall attack surface (*Minimize Attack Surface Area*). Tools such as SSH are not required in the image, due to the immutable infrastructure paradigm. The initial effort to build a custom base Docker image is high, since all required dependencies have to be identified, installed and tested. Nevertheless it addresses two basic threats (T1.1 and T1.3) and gives control over the installed software (*Don't Trust Service*, in this case *Docker Hub* images). To further reduce vulnerabilities in the Docker base image, additional security scanners, like for example *Clair* [17], can be integrated into the CI pipeline. The results of the static analysis have to be analyzed and depending on the risk, patches have to be applied or dependencies removed.

Threat T1.3 can also be addressed by hardening the base OS image after all required dependencies have been installed. Possible strategies to harden images and containers are described in detail in the paper *Understanding and Hardening Linux Containers* [33].

T1.4 (hardcoded credentials) is mainly a mistake by developers. To mitigate hardcoded credentials, a proper secret management strategy needs to be introduced. This involves using *Docker Secrets* as secret management platform, which allows to secretly store information in the encrypted raft log. Since developers are not able to manage the Docker Swarms in testing and production environment, an operator has to configure the swarms and make sure the required credentials are mounted to the correct path in the running containers. Secrets which are required for a service are configured by developers in the final `docker-compose.prod.yml`, but this configuration does not contain the values of the secrets itself. The process looks as follows: developers configure the secrets in the `docker-compose.prod.yml` file and ask operators to create or update the corresponding secrets in the testing or production swarm. After the secrets have been created, the application can be deployed and automatically receives the required credentials when started.

Threat T1.5 (untested *latest* tag) should be taken care of in the complete development and deployment process (it is also seen as an anti-pattern [35]). The *latest* tag is used as default tag, if no other tag has been specified. If a developer or the build server accidentally tags an untested or unstable image as *latest*, it is possible that this image is executed in production environment. This threat can be eliminated by configuring the target environment to always use a specific version tag.

Additionally, to mitigate threat T1.4 and T1.5, compliance rules can be introduced which developers have to follow. Those rules could for example be to avoid any confidential data in Docker images or to use a secure coding process.

The elimination of T1.2 (image forgery) by using DCT will be explained in chapter *Registry (C5)*.

Residual Risk

Three out of five threats could be eliminated by the described mechanisms. The remaining threats (T1.3 with risk *high* and T1.4 with risk *medium*) can be mitigated but leave a residual risk for Docker images.

6.2.2 Local Environment (C2)

To create a secure baseline for local environments, basic OS hardening should be applied, like for instance account passwords, encrypted hard drive or firewalls (*Establish Secure Defaults*). The details to basic hardening is out of the scope of this work, since hardening

guides for different operating systems already exist, such as the *CIS Ubuntu Linux 16.04 LTS Benchmark* ([23]). An overview of identified threats mapped to likelihood, impact, risk and a mitigation strategy can be found in Table 6.2.

Threat	Likelihood	Impact	Risk	Mitigation
T2.1 (source code manipulation)	low	high	medium	mitigate
T2.2 (credential leaks)	medium	high	high	mitigate
T2.3 (source code leak)	low	medium	low	mitigate
T2.4 (loss of credentials)	low	low	low	accept
T2.5 (vulnerabilities)	high	high	high	mitigate
T2.6 (misconfiguration)	low	high	medium	eliminate

Table 6.2: Risk Calculation and Mitigation (C2)

Explanations

T2.1: The likelihood of source code manipulation is low, since the attacker needs to gain access to the local environment and the developers should review their changes before pushing into the VCS. The impact is high, since the attacker could place backdoors which have access to production data.

T2.2: Credential leaks are likely as humans are the weakest point of a system. Credentials could be stolen for example by a phishing attack or exploiting a vulnerability in the local environment. The impact is high, since an attacker could gain access to the VCS and push manipulated source code.

T2.3: A source code leak happens if an attacker gains access to the local environment, therefore they are unlikely. The impact is medium, since an attacker could easier find bugs which he could exploit in production.

T2.4: The loss of credentials is unlikely, since a developer should create backups of his local machine. The impact is also low, since new credentials and keys can be generated by an operator.

T2.5: Vulnerabilities on the local system are to be expected. The impact is high, since an attacker could gain root access on the local machine.

T2.6: A misconfigured Docker Engine is unlikely, since Docker does not expose the socket by default. The impact of an exposed socket is high, since the attacker would gain root on the local machine.

Mitigation Strategies

If an attacker gained access to a local environment, he is able to manipulate the source code which then could be pushed into the VCS (T2.1). This threat can be mitigated by manually reviewing each source code change before pushing into the VCS. This could be achieved by using a code review process, such as pair programming ([118]) or working with pull requests, which have to be reviewed by other developers.

To protect local credentials and mitigate T2.2 (credential leak), a password manager, like for example *1Password* ([39]) or *KeePassX* ([112]) should to be used. Additionally, the private SSH keys should be protected with a password, which must be entered on each key access (*Defense in Depth*). To avoid phishing attacks, a developer should be careful when opening emails or clicking on links. Additionally, security awareness seminars can be realized for developers.

T2.3 (source code leak) can be mitigated by hardening the base system and carefully handling the source code. This means that the source code should never leave the local environment anywhere else than to the VCS.

To mitigate T2.4 (credential loss), regular backups should be created and a restore process has to be defined.

To reduce the threat of a privilege escalation caused by vulnerabilities in the development tools (T2.5) or misconfiguration of the local environment (T2.6), regular security patches need to be applied. Also the configuration of the local development tools need to be reviewed and hardenings should be applied. This could for instance be the configuration of the Docker Engine to not expose the socket to the network (*Establishing Secure Defaults*). To eliminate threat (T2.6), default configuration can be provided by the IT security department.

Residual Risk

Threat T2.6 could be eliminated by providing secure configurations to the developers. T2.4 can be accepted since likelihood, impact and risk are *low*. Threat T2.2 and T2.5 with risk *high*, T2.1 with risk *medium* and T2.3 with risk *low* present a residual risk which should be mitigated using the solutions describe above.

6.2.3 VCS (C3)

The VCS contains the source code of the application. An overview of identified threats mapped to likelihood, impact, risk and a mitigation strategy can be found in Table 6.3.

Threat	Likelihood	Impact	Risk	Mitigation
T3.1 (authentication bypass)	low	high	medium	mitigate
T3.2 (unauthorized push)	low	high	medium	eliminate
T3.3 (MITM)	medium	medium	medium	eliminate
T3.4 (vulnerabilities)	high	high	high	mitigate
T3.5 (interactions)	low	medium	low	mitigate
T3.6 (destroying source code)	low	high	medium	mitigate

Table 6.3: Risk Calculation and Mitigation (C3)

Explanations

T3.1: The likelihood of authentication bypasses is medium, since they could be achieved using vulnerabilities in the software. The impact of an authentication bypass is high, since an attacker could push manipulated source code.

T3.2: Unauthorized pushes are unlikely to happen, since authentication and authorization should be enabled. The impact is high, since an attacker could push manipulated source code.

T3.3: MITM attacks are likely to happen since if an attacker has access to the network traffic, he can easily sniff it. The impact is medium since sensitive information could be leaked.

T3.4: Vulnerabilities are very likely to occur and have a high impact since they could be abused to gain unauthorized access to the VCS or the host system.

T3.5: The likelihood of too many interactions happening in parallel is low, since developers push their changes irregularly. The impact is medium, since the build server would not be triggered and deployments could not be made.

T3.6: The likelihood of source code being destroyed is low, since regular backups should be created. The impact is high, since the main asset would be lost.

Mitigation Strategies

To prevent unauthorized attackers from pushing manipulated source code (T3.2) into the VCS, authentication and authorization has to be enabled. This can be achieved by using public key authentication. Each developer generates a pair of a private and a public

key and hands over the public key to the operator. The operator configures the VCS to allow authentication with the private key corresponding to the public key. Now the developer is able to push and pull source code using his private key. Each private key should be held in a safe place and protected by a password, to prevent a credential leak (T2.2) (*Principle of Defense in Depth*). Alternatively a username password combination can be used, which has to be entered on each authentication. The password should be secure ([32]) and kept in a safe place, by using for example a password manager. To control access to each repository, authorization should be enabled in the VCS. The access of each developer should limited to the repositories on which he is currently working (*Principle of Least Privilege*).

To mitigate vulnerabilities (T3.4), which could also allow an authentication bypass (T3.1), regular security patches to the VCS have to be applied.

Another threat is a MITM attack when transferring source code (T3.3). To eliminate this attack, an encrypted channel has to be used to push and pull source code. This could be for example SSH, which also comes with public key authentication. If a connection is initially established, the fingerprint of the server's public key has to be validated to ensure the communication is done with the correct server. To further reduce the threat of source code forgery, transmission over insecure protocols, as for instance HTTP should be disabled (*Minimize Attack Surface Area*).

To reduce the denial of service threat by having too many interactions at the same time (T3.5), a resource limit for the server and an access limit for the developer should be configured.

To mitigate the threat of destroyed source code (T3.6), regular backups of the source code and the VCS have to be created.

Residual Risk

Threats T3.2 and T3.3 could be eliminated. T3.4 with risk *high* leaves a high residual risk which should be addressed by using additional processes or compliance rules for patch management. T3.1 and T3.6 with risk *medium* also leave a residual risk and should be watched. Threat T3.5 leaves a small residual risk, due to the *low* risk rating.

6.2.4 Build Server (C4)

The build server is a central component, crosses two trust boundaries (TB1 / TB2 and TB2 / TB3), and operates on sensitive source code. An overview of identified threats mapped to likelihood, impact, risk and a mitigation strategy can be found in Table 6.4.

Threat	Likelihood	Impact	Risk	Mitigation
T4.1 (broken auth)	medium	high	medium	mitigate
T4.2 (leaked credentials)	low	high	medium	mitigate
T4.3 (XSS)	high	medium	high	mitigate
T4.4 (unauthenticated access)	low	high	medium	eliminate
T4.5 (MITM)	medium	medium	medium	eliminate
T4.6 (injection attack)	high	high	high	mitigate
T4.7 (parallel builds)	low	medium	low	mitigate
T4.8 (vulnerabilities)	high	high	high	mitigate
T4.9 (CSRF)	medium	high	high	mitigate
T4.10 (host compromise)	low	high	medium	accept

Table 6.4: Risk Calculation and Mitigation (C4)

Explanations

T4.1: The likelihood of a broken authentication mechanism is medium, since vulnerabilities could lead to such a failure. The impact is high, since an attacker could get access to source code and the testing environment.

T4.2: Leaked credentials are unlikely, since the build server should be separated from other environments. The impact is high, since an attacker could gain access to the testing environment and manipulate source code.

T4.3: XSS attacks in the webinterface are very likely, since *OWASP Top 10 security risks for web applications* defines them as very widespread [99]. Attackers could for example steal cookies and hijack a user session, which makes this a medium impact.

T4.4: Unauthenticated access is unlikely to happen, since authentication and authorization should be enabled. The impact is high, since an attacker could gain access to sensitive data.

T4.5: A MITM attack is likely to happen if the attacker has access to the network traffic. The impact is medium, since an attacker could sniff sensitive data which is sent unencrypted over the network.

T4.6: Injection attacks are very likely to happen since they are easy to exploit and a common threat [99]. The impact is high, since an attacker could gain access to sensitive information or even get access to the complete host.

T4.7: Too many parallel builds are unlikely to happen, since developers commit and

push infrequently, which spreads the workload. The impact is medium, since it could postpone deployments.

T4.8: Vulnerabilities are very likely to occur and have a high impact, since an attacker could gain root access on the host machine.

T4.9: CSRF attacks are common and they are easy to detect [99], which makes them likely. The impact is medium, since an attacker can do what the current logged in person is privileged to do.

T4.10: A host compromise is unlikely, since per default the Docker socket is not exposed to the network. The impact is high, since it leads to an instant compromise of the host.

Mitigation Strategies

To enable authentication and authorization and eliminate T4.4, global security in Jenkins should be activated and anonymous access disabled (*Establish Secure Defaults*). As user realm multiple options exist, such as *Jenkin*'s own database, LDAP or Unix users and groups [82]. Access to *Jenkins* configuration should be limited to operators, to avoid misconfiguration of the build processes (*Principle of Least Privilege*). This also prevents developers from bypassing integrated security mechanisms in the pipeline, like for example security or compliance checks could lead to the build failing in case a security requirement has not been met.

The threat of a broken authentication or session management (T4.1) can be reduced by regularly installing the latest security patches to the automation software and disabling unneeded features, as for instance the integrated CLI which exposes vulnerabilities like java deserialization bugs [82] (*Minimize Attack Surface Area*). This also mitigates the threat of vulnerabilities in the software (T4.8), XSS (T4.3), injection attacks (T4.6) and CSRF (T4.9).

The build server requires credentials for the following actions:

- **VCS**: Pull source code
- **Testing Registry**: Push Docker image
- **Testing Docker Swarm**: Access to deploy the application using SSH
- **Email / Slack**: Send notifications to developers

To reduce the threat of leaked credentials (T4.2), they should be managed using *Jenkin*'s *Credentials* plugin which encrypts the keys using the combination of a master and intermediate keys [115]. To further reduce the threat, access to managing credentials should be limited to operators (*Principle of Least Privilege* and *Separation of Duties*).

To eliminate T4.5 (MITM) and mitigate T4.2 (leaked credentials), TLS encryption should be enabled. This can be achieved by using either self-signed certificates or a valid domain certificate. Both methods ensure the encryption of data between client and server. Domain certificates additionally are able to eliminate MITM (T4.5) attacks, since certificates can be validated on connection setup.

A detailed guide on how to lock down Jenkins can be found in *Securing Jenkins CI Systems* ([82]).

Another topic is the Docker socket which has been mounted into the *Jenkins* container. This is required to enable Docker builds in a Docker container, as described in *jpetazzo/Using Docker-in-Docker for your CI or testing environment? Think twice.* ([101]). If the *Jenkins* container has been compromised, this leads to an instant host compromise (T4.10), since the *Docker Engine* is able to spawn privileged containers which can mount the root filesystem writable. To reduce the impact of this compromise, the build server should be isolated in the network and no other application should be run in the same Docker swarm or on the host machine (*Principle of Least Privilege*).

The remaining threat T4.7 (too many parallel builds) can be mitigated by limiting the number of concurrent builds in the pipeline. Also resource limits for each Jenkins slave container can be set to avoid memory exhaustion.

Residual Risk

Threats T4.4 and T4.5 could be eliminated. The residual risk for the build server remains high, since the threats T4.1, T4.2, T4.3, T4.6, T4.8 and T4.9 have a *high* or *medium* risk value and could not be eliminated completely. Those risks make the build server a highly valuable target for an attacker. The overall risk should be reduced by completely locking away the build server into a separated network with nothing else running and strict access controls. Additional security reviews for the build server software should be created to reduce the number of vulnerabilities. This reduces the likelihood of an incident happening. If a SSC based on Docker is used, which requires the Docker socket to be mounted into the Jenkins container, T4.10 rated as *high* exposes a high residual risk due to the fact, that it has to be accepted.

6.2.5 Registry (C5)

The testing and production registries hold Docker images which are needed by the Docker Swarms. Table 6.5 provides overview of identified threats mapped to likelihood, impact, risk and a mitigation strategy.

Threat	Likelihood	Impact	Risk	Mitigation
T5.1 (broken auth)	medium	high	medium	mitigate
T5.2 (unauthenticated push)	low	high	medium	eliminate
T5.3 (image forgery)	low	high	medium	eliminate
T5.4 (vulnerability)	high	high	high	mitigate
T5.5 (MITM)	medium	medium	medium	eliminate
T5.6 (too many interactions)	low	medium	low	mitigate
T5.7 (missing disk space)	medium	medium	medium	mitigate

Table 6.5: Risk Calculation and Mitigation (C5)

Explanations

T5.1: The likelihood of a broken authentication mechanism is medium, since vulnerabilities could be responsible for this to happen. The impact is high, since an attacker could get access to all images.

T5.2: An unauthenticated push is unlikely to happen, since authentication and authorization should be enabled. The impact is high, since an attacker could push a tampered image including a backdoor to gain access to production data.

T5.3: Image forgery while pushing an image into the registry is unlikely to occur, since an attacker would need access to the internal network. The impact is high, because the attacker would be able to push backdoored images.

T5.4: Vulnerabilities should be expected to occur and have a high impact, since they could be used to gain access to all Docker images in the registry.

T5.5: MITM attacks are likely to happen if an attacker has access to the internal network. They have medium impact since an attacker could gain access to unencrypted sensitive information.

T5.6: Too many interactions are unlikely since access is limited to the build server and operators. Deployments could be postponed, which makes this a medium impact.

T5.7: Exhausted disk space is likely to occur, especially if the infrastructure is new and no experiences have been gained yet. The impact is medium, since deployments could be postponed.

Mitigation Strategies

To make sure images cannot be pushed by untrusted entities (T5.2), authentication and authorization have to be used. *Docker Registry* supports native basic authentication ([47]) (which requires TLS to be configured) and delegating authentication to a trusted token server ([72]). To enable more advanced authentication and authorization, a proxy is recommended in front of the registry ([47]) (*Principle of Defense in Depth*). Access should be limited to the responsible entities like the build server and operators (*Principle of Least Privilege*).

To mitigate the threat of vulnerabilities (T5.4) and a broken authentication mechanism (T5.1), regular security patches to the registry should be applied.

The threat of a MITM attack (T5.5) can be eliminated by encrypting traffic using TLS in combination with certificate pinning. Additionally the `insecure-registries` key in the `daemon.json` which allows unencrypted traffic should be avoided (*Establish Secure Defaults*). To further reduce the threat of image forgery (T5.3), DCT should be used and enabled by default (*Establish Secure Defaults*). To use DCT, an additional *Notary* server is required. It is responsible for handling image tag signatures. Before pushing an image into the testing registry, the build server has to sign the image tag. The testing swarm is then able to check the signature of the build server (by setting `DOCKER_CONTENT_TRUST=1`) before running the application. On deployment to production, the operator has to sign the image from the testing registry and push it into the production registry. The production swarm is then able to make sure the image has been signed by the build server and additionally by an operator.

The threat of a DoS attack by too many interactions at the same time (T5.6) or disk space exhaustion (T5.7) can be mitigated by defining resource limits or by replicating the registry across multiple nodes which have enough resources available. To further increase the overall security, regular backups need to be created.

To reduce the overall risk of a vulnerability in a Docker image, additional image security scannings can be integrated into the registry. Docker Registry by default does not offer an integrated security scanner, but commercial products are available, like for example *Xray* from JFrog ([89]) which integrates into *Artifactory* ([88]).

Residual Risk

Threats T5.2, T5.3 and T5.5 could be eliminated by using mechanisms discussed in this section. T5.4, T5.1 and T5.7 could be mitigated but expose a residual risk which has to be addressed by using a patch management process and defining resource quotas. T5.6

has a *low* residual risk and can be accepted.

6.2.6 Docker Swarm (C6)

Docker Swarms are used to run and scale the application across multiple nodes. To create a secure baseline, the underlying OS should be hardened. The detailed hardening process is out of scope of this work, since hardening guides for different operating systems already exist, like for instance the *CIS Ubuntu Linux 16.04 LTS Benchmark* ([23]). Table 6.6 provides an overview of identified threats mapped to likelihood, impact, risk and a mitigation strategy.

Threat	Likelihood	Impact	Risk	Mitigation
T6.1 (exposed docker socket)	low	high	medium	eliminate
T6.2 (MITM)	high	high	high	eliminate
T6.3 (leaked credentials)	medium	high	high	mitigate
T6.4 (resource exhaustion)	medium	high	high	mitigate
T6.5 (container to host breakout)	medium	high	high	mitigate

Table 6.6: Risk Calculation and Mitigation (C6)

Explanations

T6.1: The likelihood of an exposed Docker socket is low, since the socket is not exposed by default. The impact is high, because the attacker could gain an instant host compromise.

T6.2: A MITM attack is expected to occur, if the Docker swarm is extended over multiple datacenters. The impact is also high, since sensitive information could be leaked if an attacker sniffs the network.

T6.3: Leaked credentials are likely to occur, especially if error messages or stack traces are misconfigured. The impact is high, since sensitive data could be accessed by using the leaked credentials.

T6.4: Resource exhaustion is likely to occur, especially if an attacker uses a DDoS attack. The impact is high, as the application could become unavailable to other users.

T6.5: A container to host breakout is likely to occur, since an attacker could use a vulnerability in the application in combination with a kernel exploit to gain access to the host system. The impact is high, since the attacker would gain access to production data.

Mitigation Strategies

Access to Docker Swarm management should be given using an encrypted authentication method, like for example SSH. This eliminates the threat of an exposed Docker socket (T6.1), since using SSH makes exposing the socket to the network obsolete and it can be bound to *localhost* (*Establish Secure Defaults*). Access to managing the swarm should be reduced to either the build server (to deploy on testing swarm), or an operator (to manage the testing and production swarms) (*Principle of Least Privilege*).

To eliminate MITM attacks (T6.2) between master and worker nodes, Docker Swarm automatically creates a public key infrastructure (PKI). This encrypts control traffic between managers and workers by default (*Establish Secure Defaults*) ([52]). To encrypt data which is sent from container to container (data plane), overlay encryption should be enabled by adding the `--opt encrypted` flag when creating an overlay network. This is especially relevant, if container clusters are extended over multiple datacenters (which is not recommended due to the increasing network delay).

To mitigate T6.3 (leaked credentials), credentials need to be stored safely. *Docker Secrets* can be used to store and encrypt credentials in the raft log and mount them into containers on demand. To make sure no unauthorized entity can read them, access should be limited to operators (*Principle of Least Privilege*). Also error messages of the application have to be adjusted to not leak information (*Fail Securely*).

Resource exhaustion is a problem in container clusters. Per default, no resource limits for containers are set. This means that a single container is able to block the whole swarm by using all memory or CPU resources (T6.4). This threat can be mitigated by setting resource constraints when starting a container. This is not possible on Docker swarm level, so every application has to define their own runtime limits [64]. This should be enforced and validated in the CI pipeline by running checks on the `docker-compose.prod.yml` file. This could also be enforced by setting parameters for the *dockerd* process while configuring the infrastructure [54]. One solution would be to create a cgroup with a defined memory limit. This cgroup is assigned as parent to all containers on the host by using the parameter `--cgroup-parent`.

One concern of using Docker compared to traditional virtualization is the isolation capability of Docker containers. Container management solutions use a set of kernel features to isolate process groups from each other. Containers share the same kernel, which means that using a kernel exploit (T6.5, container to host breakout) in a compromised container could lead to a compromise of all other containers on the same host. This threat can be mitigated by using patches from *Grsecurity* [98] to harden the Linux kernel against exploits. Using grsecurity patches on the other side is hard to maintain, since the kernel

has to be compiled by hand.

Additionally, `seccomp-bpf` filtering can be used to limit the amount of system calls which can be issued by a process (*Minimize Attack Surface Area*). The default Docker `seccomp` profile disables around 44 system calls [70]. The attack surface can be reduced further by removing allowed system calls one by one and testing, if the application still works.

To further reduce the permissions of a Linux container, all Linux capabilities should be removed (*Principle of Least Privilege*). Typical web applications do not require Linux capabilities to run, since they are dynamically parsed and executed. To find out which capabilities an application requires, utility scripts as described in *Find what capabilities an application requires to successful run in a container* [15] are available.

To add an additional layer of security, kernel extensions such as *SELinux* or *AppArmor* can be used.

Containers which run using the `--privileged` flag should be avoided at any time, since if such a container is compromised, the host is instantly compromised as well (*Principle of Least Privilege*).

Residual Risk

Threat T6.1 and T6.2 could be eliminated by using correct Docker Engine configurations and encryption. T6.3-T6.5 expose a *high* risk which can be mitigated using mechanisms described above. The overall residual risk is still high, especially in production environments, due to the sensitiveness of data used. This is the reason why especially production environments should add additional security layers, as for instance a physical separation to other environments.

6.3 Deployment Process

The deployment should be divided into two separate processes: deployment to testing and deployment to production environment (*Separation of Duties*).

Deployment to a testing environment is less critical than to production. The reason is that the data used in the testing environment database is no real data. Also the end users are not affected if a testing environment has been compromised. This is why this step can be done in an automated process, like for example by using a build server. To access the testing environment, the build server has to store SSH credentials, which means that if the build server is compromised, the attacker also has access to testing.

Deployments to production environment should be a separate process. If the build server would be responsible for deploying to production and save the access credentials, a compromise of the build server would also mean getting access to real client data.

Additionally, it would create a single point of failure. This is the reason why an operator should be responsible for deploying the application to production. The operator can use the management network to log into the testing environment, pull the latest tested Docker image, sign it using DCT and push it into the production registry. By using this workflow, the operator can also make sure that each image is tested before pushing it into the production registry. After pushing the image, he can log into the production environment using SSH and is then able to update the production swarm.

6.4 Security Scanning

Automated CI pipelines in combination with Docker allow the integration of additional security scanning in the process. This helps to reduce the overall risk of manipulated or insecure source code being deployed into production environment. Two main integration points were found in this work: the image registries and the automated test and build steps.

Image registries can be extended by plugins which automatically scan the pushed images, as for example *Docker Security Scanning* [111]. They are able to scan images for installed packages and compare the results with security feeds from vendors. This is one use case in the so called *Deep Container Inspection (DCI)* ([91]). It is possible to develop a workflow in which the registries automatically deny the push if a certain threshold of vulnerabilities or compliance fails have been found in the image.

Additionally, security checks can be integrated into the automated test and build process. These checks can lead from simple source code analysis to individual Docker security checks. Simple source code analysis could for example include checks for typical programming mistakes or the usage of insecure libraries or functions. Docker specific checks could be analyzing the `docker-compose.yml` for insecure parameters, such as a network port which should not be open. This step could also be used to enforce compliance rules before deploying an application to testing or production. If a compliance check fails, the build will fail and inform the developers.

6.5 Patch Management

Patch management in automated container environments which follow the *Immutable Infrastructure* paradigm requires only little manual interaction.

To patch a running application container, a rebuild and redeployment can be triggered. A new build is triggered on the separate build server for the base OS image and on the build server for the application (in the SSC). This creates a new application image based on a new base OS image, including the latest security patches. After building is done,

the resulting image can be tested on the testing environment and afterwards deployed onto production. After running the deploy command, all old containers are replaced with newly patched containers.

Docker Swarm allows to add and remove new nodes while keeping the cluster alive. If the infrastructure orchestration has been automated, a newly patched base OS image can be created and rolled out into the infrastructure. The infrastructure tools shut down the nodes one by one, while replacing them with new ones. The swarm manager is responsible for delivering workload onto the newly added instances while keeping the swarm alive.

7 Conclusion

In the first chapters, background knowledge to agile software development, CI pipelines, container virtualization and the Docker ecosystem was explained. A SSC was defined and it was explained how it integrates with the other topics. In the next chapter the threat modeling methodology including basic knowledge about information security was defined. Afterwards a SSC was built and modeled using this methodology. The main components, processes and users and how they interact with each other were identified and described afterwards. Based on this information trust boundaries were defined and a threat model was created using STRIDE. In the last chapter *Securing the Software Supply Chain* the identified threats were used to calculate the risk and mitigation strategies were defined. Also possible solutions to secure the SSC were discussed for each component and the residual risk was explained. This chapter also answered the first three research questions from the beginning of this work. The last research question *Can a Software Supply Chain be completely secure?* can be answered as follows: It cannot. The results from Chapter *Securing the Software Supply Chain* show, that there is always a residual risk which can be mitigated, but not completeley be elmiminated.

The requirements from Chapter *Requirements* could be fulfilled. Requirement #1 (Understand the Background and the Docker Ecosystem) was achieved by explaining background knowledge in Chapter *Agile Software Development and Docker*. Requirement #2 (Build a Software Supply Chain) was carried out by defining what a SSC is and explaining the most relevant components in Chapter *Modeling a Software Supply Chain*. Requirement #3 (Secure the Software Supply Chain) could be partially fulfilled by creating a threat model, calculating risks and discussing security mechanisms in Chapter *Securing the Software Supply Chain*. STRIDE helped to elaborate threats, but it is not able to find all threats for a component. It always remains a residual risk which cannot be completely addressed, since there is no *total security*. Requirements #4 (Use Docker as base Technology) and #5 (Use Open Source / Free software) were implicitly fulfilled by building the SSC based on those requirements.

As summary it can be said that a SSC is a collection of processes and components in an organization which handles sensitive data and needs to be protected regarding the CIA objectives. The Docker ecosystem provides a good basis since most components, as for example the VCS or build server can be containerized and run using the same technology.

Especially when it comes to running an application in a testing or production container cluster, Docker Swarm takes over a lot of work by orchestrating containers over multiple nodes. In terms of security Docker provides a good baseline by adding an additional level of isolation between applications running on the same host. Nevertheless, additional hardening in form of resource limits (cgroups), seccomp profiles, MAC policies or overlay network encryption should be applied.

By using a structurized threat modeling process in combination with STRIDE, this work could identify the main threats for the most relevant components of a SSC and calculate risks based on the likelihood and impact. Based on these risks security mechanisms could be discussed and critical components identified. After threat modeling the SSC this work found out, that each component has the residual risk of vulnerabilities in the software. This makes a patch management process essential. Due to the immutable infrastructure paradigm, the containerized components using Docker can be updated by rebuilding and redeploying the images. This work also found out, that the build server and the production environments are critical components due to a high residual risk. The build server handles sensitive source code and should be isolated using networks with firewalls. The production swarm contains real user data and should be physically separated from the other components to avoid lateral movement.

This work can be used as basis to build and secure a SSC. However, remaining components and processes, such as license and dependency management, logging and monitoring processes or business integration (for example threat intelligence [6]) have to be reviewed, evaluated and integrated.

Glossary

ACL Access Control List. 62

API Application Programming Interface. 17, 18, 21–23, 33, 54

ARP Adress Resolution Protocol. 38

ASF Application Security Frame. 37

ASLR Address Space Layout Randomization. 39

BPF Berkeley Packet Filter. 16

CD Continuous Deployment. 10

CI Continuous Integration. 3, 7, 10, 11, 17, 43, 45, 63, 75, 77, 79

CIA Confidentiality, Integrity, Availability. 31, 44, 61, 79

CLI Command Line Interface. 5, 17, 19, 21, 24, 70

CPU Central Processing Unit. 15, 75

CSRF Cross-Site Request Forgery. 58, 69, 70

DCT Docker Content Trust. 24, 25, 64, 73, 77

DDoS Distributed Denial of Service. 39, 74

DFD Data Flow Diagram. 37

DNS Domain Name System. 38

DoS Denial of Service. 36, 39, 73

GID Group ID. 16

HTML Hypertext Markup Language. 44

HTTP Hypertext Transfer Protocol. 35, 54, 68

HTTPS Hypertext Transfer Protocol Secure. 54

IaaS Infrastructure as a Service. 12

IP Internet Protocol. 38

IPC Inter-Process Communication. 14

iSCSI internet Small Computer System Interface. 20

LDAP Lightweight Directory Access Protocol. 70

MAC Mandatory Access Control. 15, 16, 80

MITM Man in the Middle. 34, 38, 58, 59, 67–69, 71–75

NFS Network File System. 20

OS Operating System. 13, 15, 45, 46, 64, 74, 77, 78

PASTA Process for Attack Simulation and Threat Analysis. 37

PID Process ID. 14, 15

PKI Public Key Infrastructure. 75

REST Representational State Transfer. 9, 17

SCM Supply Chain Management. 10, 11

SDLC Software Development Lifecycle. 36

SMB Server Message Block. 20

SSC Software Supply Chain. 11, 12, 17, 21, 22, 29, 31, 37, 41, 43–45, 49–51, 53–57, 61, 62, 71, 77, 79, 80

SSH Secure Shell. 48, 54, 57, 62, 63, 66, 68, 70, 75–77

SSL Secure Socket Layer. 5

TLS Transport Layer Security. 35, 71, 73

TOCTTOU Time-of-Check-to-Time-of-Use. 35

TUF The Update Framework. 24

UID User ID. 16

UTS UNIX Timesharing Service. 14

VCS Version Control System. 43, 45–47, 49–51, 54, 58, 62, 63, 65–68, 79

VM Virtual Machine. 12

XP Extreme Programming. 8

XSS Cross-Site Scripting. 58, 69, 70

Bibliography

[1] Atlassian. Erstellen, testen, deployen. `https://de.atlassian.com/software/bamboo`, 2017. Accessed: 14.07.2017.

[2] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *ACM SIGOPS operating systems review*, volume 37, pages 164–177. ACM, 2003.

[3] Kent Beck. *Extreme programming explained: embrace change*. addison-wesley professional, 2000.

[4] Jim Bird. Devopssec: Delivering secure software through continuous delivery. 2016.

[5] Bitcom. War ihr unternehmen in den letzten 2 jahren von datendiebstahl, industriespionage oder sabotage betroffen? `https://de.statista.com/statistik/daten/studie/150885/umfrage/anteil-der-unternehmen-die-opfer-von-digitalen-angriffen-wurden/`, 2015. Accessed: 14.03.2017.

[6] Matt Bromiley. Threat intelligence: What it is, and how to use it effectively. *SANS Institute InfoSec Reading Room*, 2016.

[7] Thanh Bui. Analysis of docker security. *arXiv preprint arXiv:1501.02967*, 2015.

[8] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *Communications of the ACM*, 59(5):50–57, 2016.

[9] GitLab B.V. The platform for modern developers. `https://about.gitlab.com`, 2017. Accessed: 14.07.2017.

[10] Identity Theft Resource Center and CyberScout. Data breaches increase 40 percent in 2016, finds new report from identity theft resource center and cyberscout. `http://www.idtheftcenter.org/2016databreaches.html`, 2017. Accessed: 14.03.2017.

[11] Sitaram Chamarty. Hosting git repositories. `http://gitolite.com`, 2017. Accessed: 03.08.2017.

[12] ClusterHQ and DevOps.com. Container market adotpion survey 2016. `https://clusterhq.com/assets/pdfs/state-of-container-usage-june-2016.pdf`, 2016. Accessed: 14.03.2017.

[13] Alistair Cockburn. *Agile software development*, volume 177. Addison-Wesley Boston, 2002.

[14] Alistair Cockburn. Crystal methodologies. `http://alistair.cockburn.us/Crystal+methodologies`, 2017. Accessed: 26.08.2017.

[15] William Cohen. Find what capabilities an application requires to successful run in a container. `https://developers.redhat.com/blog/2017/02/16/find-what-capabilities-an-application-requires-to-successful-run-in-a-container/`, 2017. Accessed: 18.08.2017.

[16] Theo Combe, Antony Martin, and Roberto Di Pietro. To docker or not to docker: A security perspective. *IEEE Cloud Computing*, 3(5):54–62, 2016.

[17] CoreOS. Vulnerability static analysis for containers. `https://github.com/coreos/clair`. Accessed: 14.03.2017.

[18] CoreOS. Quay [builds, analyzes, distributes] your container images. `https://quay.io`, 2017. Accessed: 20.06.2017.

[19] Roberto Munoz Daniel Garcia. Docker security analysis & hacking tools. `https://github.com/cr0hn/dockerscan`. Accessed: 14.03.2017.

[20] Datadog. 8 surprising facts about real docker adoption. `https://www.datadoghq.com/docker-adoption/`, 2016. Accessed: 14.03.2017.

[21] Brian Fitzgerald, Nicole Forsgren, Klaas-Jan Stol, Jez Humble, and Brian Doody. Infrastructure is software too! 2015.

[22] Center for Internet Security. Cis docker 1.13.0 benchmark. Technical report, 2017.

[23] Center for Internet Security. Cis ubuntu linux 16.04 lts benchmark. Technical report, 2017.

[24] Eclipse Foundation. Jenkins. `https://jenkins.io`, 2017. Accessed: 14.07.2017.

[25] OWASP Foundation. Application threat modeling. `https://www.owasp.org/index.php/Application_Threat_Modeling`, 2017. Accessed: 29.06.2017.

[26] OWASP Foundation. Security by design principles. `https://www.owasp.org/index.php/Security_by_Design_Principles`, 2017. Accessed: 10.08.2017.

[27] OWASP Foundation. Threat risk modeling. `https://www.owasp.org/index.php/Threat_Risk_Modeling`, 2017. Accessed: 29.06.2017.

[28] Martin Fowler and Matthew Foemmel. Continuous integration. `https://martinfowler.com/articles/continuousIntegration.html`, 2006. Accessed: 29.08.2017.

[29] Martin Fowler and James Lewis. Microservices. `https://martinfowler.com/articles/microservices.html`, 2014. Accessed: 29.08.2017.

[30] Michael Friis. Announcing docker enterprise edition. `https://blog.docker.com/2017/03/docker-enterprise-edition/`, 2017. Accessed: 14.03.2017.

[31] Mark Graff and Kenneth R Van Wyk. *Secure coding: principles and practices.* " O'Reilly Media, Inc.", 2003.

[32] Paul A Grassi, Michael E Garcia, and James L Fenton. Digital identity guidelines. *National Institute of Standards and Technology, Los Altos, CA*, 2017.

[33] Aaron Grattafiori. Understanding and hardening linux containers. *Whitepaper, NCC Group*, 2016.

[34] Larry H. Linux kernel heap tampering detection. `http://phrack.org/issues/66/15.html`, 2009. Accessed: 25.05.2017.

[35] Ben Hall. Dockerfile and the :latest tag anti-pattern. `http://blog.benhall.me.uk/2015/01/dockerfile-latest-tag-anti-pattern/`, 2017. Accessed: 16.08.2017.

[36] Jesse Hertz. Abusing privileged and unprivileged linux containers. *Whitepaper, NCC Group*, 2016.

[37] Jim Highsmith. *Adaptive software development: a collaborative approach to managing complex systems.* Addison-Wesley, 2013.

[38] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation.* Pearson Education, 2010.

[39] AgileBits Inc. Go ahead. forget your passwords. `https://1password.com`, 2017. Accessed: 04.08.2017.

[40] Docker Inc. Docker bench for security. `https://dockerbench.com`. Accessed: 14.03.2017.

[41] Docker Inc. Explore - docker hub. https://hub.docker.com/explore/. Accessed: 14.03.2017.

[42] Docker Inc. Introduction to container security: Understanding the isolation properties of docker. https://www.docker.com/sites/default/files/WP_IntrotoContainerSecurity_08.19.2016.pdf, 2016. Accessed: 14.03.2017.

[43] Docker Inc. About registry. https://docs.docker.com/registry/introduction/, 2017. Accessed: 23.07.2017.

[44] Docker Inc. Compose file version 3 reference. https://docs.docker.com/compose/compose-file/, 2017. Accessed: 20.06.2017.

[45] Docker Inc. Content trust in docker. https://docs.docker.com/engine/security/trust/content_trust/, 2017. Accessed: 29.06.2017.

[46] Docker Inc. debian. https://hub.docker.com/_/debian/, 2017. Accessed: 18.07.2017.

[47] Docker Inc. Deploy a registry server. https://docs.docker.com/registry/deploying/, 2017. Accessed: 18.08.2017.

[48] Docker Inc. Docker 1.12: Now with built-in orchestration! https://blog.docker.com/2016/06/docker-1-12-built-in-orchestration/, 2017. Accessed: 20.06.2017.

[49] Docker Inc. Docker container networking. https://docs.docker.com/engine/userguide/networking/, 2017. Accessed: 20.06.2017.

[50] Docker Inc. Docker image for gitolite. https://hub.docker.com/r/jgiannuzzi/gitolite/, 2017. Accessed: 15.08.2017.

[51] Docker Inc. Docker overview. https://docs.docker.com/engine/docker-overview/, 2017. Accessed: 03.06.2017.

[52] Docker Inc. Docker reference architecture: Designing scalable, portable docker container networks. https://success.docker.com/Architecture/Docker_Reference_Architecture%3A_Designing_Scalable%2C_Portable_Docker_Container_Networks, 2017. Accessed: 18.08.2017.

[53] Docker Inc. Docker security. https://docs.docker.com/engine/security/security/, 2017. Accessed: 27.04.2017.

[54] Docker Inc. dockerd. https://docs.docker.com/engine/reference/commandline/dockerd/, 2017. Accessed: 18.08.2017.

[55] Docker Inc. Dockerfile reference. `https://docs.docker.com/engine/reference/builder/`, 2017. Accessed: 08.06.2017.

[56] Docker Inc. How nodes work. `https://docs.docker.com/engine/swarm/how-swarm-mode-works/nodes/`, 2017. Accessed: 23.07.2017.

[57] Docker Inc. How services work. `https://docs.docker.com/engine/swarm/how-swarm-mode-works/services/`, 2017. Accessed: 20.06.2017.

[58] Docker Inc. library/elasticsearch - docker hub. `https://hub.docker.com/r/library/elasticsearch/tags/latest/`, 2017. Accessed: 14.03.2017.

[59] Docker Inc. library/mysql - docker hub. `https://hub.docker.com/r/library/mysql/tags/latest/`, 2017. Accessed: 14.03.2017.

[60] Docker Inc. library/nginx - docker hub. `https://hub.docker.com/r/library/nginx/tags/latest/`, 2017. Accessed: 14.03.2017.

[61] Docker Inc. library/postgres - docker hub. `https://hub.docker.com/r/library/postgres/tags/latest/`, 2017. Accessed: 14.03.2017.

[62] Docker Inc. library/redis - docker hub. `https://hub.docker.com/r/library/redis/tags/latest/`, 2017. Accessed: 14.03.2017.

[63] Docker Inc. library/registry - docker hub. `https://hub.docker.com/r/library/registry/tags/latest/`, 2017. Accessed: 14.03.2017.

[64] Docker Inc. Limit a container's resources. `https://docs.docker.com/engine/admin/resource_constraints/`, 2017. Accessed: 18.08.2017.

[65] Docker Inc. Manage keys for content trust. `https://docs.docker.com/engine/security/trust/trust_key_mng/`, 2017. Accessed: 29.06.2017.

[66] Docker Inc. Manage sensitive data with docker secrets. `https://docs.docker.com/engine/swarm/secrets/`, 2017. Accessed: 04.06.2017.

[67] Docker Inc. Overview of docker compose. `https://docs.docker.com/compose/overview/`, 2017. Accessed: 20.06.2017.

[68] Docker Inc. python. `https://hub.docker.com/_/python/`, 2017. Accessed: 30.06.2017.

[69] Docker Inc. scratch. `https://hub.docker.com/_/scratch/`, 2017. Accessed: 04.08.2017.

[70] Docker Inc. Seccomp security profiles for docker. https://docs.docker.com/engine/security/seccomp/#passing-a-profile-for-a-container, 2017. Accessed: 18.08.2017.

[71] Docker Inc. Swarm mode key concepts. https://docs.docker.com/engine/swarm/key-concepts/, 2017. Accessed: 20.06.2017.

[72] Docker Inc. Token authentication specification. https://docs.docker.com/registry/spec/auth/token/, 2017. Accessed: 18.08.2017.

[73] Docker Inc. ubuntu. https://hub.docker.com/_/ubuntu/, 2017. Accessed: 18.07.2017.

[74] Gartner Inc. Gartner says worldwide server virtualization market is reaching its peak. http://www.gartner.com/newsroom/id/3315817, 2017. Accessed: 08.08.2017.

[75] Parallels Inc. Introduction to virtualization. https://openvz.org/Introduction_to_virtualization, 2017. Accessed: 18.07.2017.

[76] Slack Technologies Inc. Where work happens. https://slack.com, 2017. Accessed: 31.07.2017.

[77] Leitfaden Informationssicherheit. It-grundschutz kompakt. Technical report, Technical Report BSI-Bro10/311, Referat 114 Sicherheitsmanagement und IT-Grundschutz, Bundesamt für Sicherheit in der Informationstechnik, 2010.

[78] ISO ISO and IEC Std. Iso/iec 27000:2005(e). *Information technology Security techniques Information security management systems Requirements*, 2005.

[79] ISO ISO and IEC Std. Iso 31000:2009(e). *Risk management Principles and guidelines*, 2009.

[80] ISO ISO and IEC Std. Iso/iec 27032:2012(en). *Information technology Security techniques Guidelines for cybersecurity*, 2012.

[81] ISO ISO and IEC Std. Iso/iec 27000:2014(e). *Information technology Security techniques Information security management systems Overview and vocabulary*, 2014.

[82] Allen Jeng. Securing jenkins ci systems. *SANS Institute InfoSec Reading Room*, 2016.

[83] Auguste Kerckhoffs. La cryptographie militaire. *Journal des sciences militaires*, pages 5–83, 1883.

[84] Adam Kliarsky. Security assurance of docker containers. *SANS Institute InfoSec Reading Room*, 2016.

[85] Squiz Labs. About. `https://github.com/squizlabs/PHP_CodeSniffer`, 2017. Accessed: 11.07.2017.

[86] David LeBlanc and Michael Howard. *Writing secure code*. Pearson Education, 2002.

[87] Ying Li. Introducing docker secrets management. `https://blog.docker.com/2017/02/docker-secrets-management/`, 2017. Accessed: 14.03.2017.

[88] JFrog Ltd. Artifactory. `https://www.jfrog.com/artifactory/`, 2017. Accessed: 18.08.2017.

[89] JFrog Ltd. Jfrog xray - universal artifact analysis. `https://www.jfrog.com/xray/`, 2017. Accessed: 18.08.2017.

[90] Robert C Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall, 2002.

[91] Scott McCarty. What is deep container inspection (dci) and why is it important? `http://rhelblog.redhat.com/2015/09/03/what-is-deep-container-inspection-dci-and-why-is-it-important/`, 2015. Accessed: 20.08.2017.

[92] Cara McGoogan. Yahoo hack: What you need to know about the biggest data breach in history. `http://www.telegraph.co.uk/technology/2016/12/15/yahoo-hack-need-know-biggest-data-breach-history/`, 2016. Accessed: 14.03.2017.

[93] Shannon Meier. Ibm systems virtualization: Servers, storage, and software.

[94] Dustin J. Mitchell. Buildbot. `http://buildbot.net`, 2017. Accessed: 14.07.2017.

[95] MITRE. Cve-2015-3456. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3456`, 2017. Accessed: 30.06.2017.

[96] Sam Newman. *Building microservices: designing fine-grained systems*. " O'Reilly Media, Inc.", 2015.

[97] Diego Ongaro and John K Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference*, pages 305–319, 2014.

[98] Inc Open Source Security. Grsecurity adds confidence to containers. `https://grsecurity.net`, 2017. Accessed: 18.08.2017.

[99] Top OWASP. Top 10-2013. *The Ten Most Critical Web Application Security Risks*, 2013.

[100] Jrme Petazzoni. Immutable infrastructure with docker and containers (gluecon 2015). GlueCon 2015, 2015.

[101] Jrme Petazzoni. jpetazzo/using docker-in-docker for your ci or testing environment? think twice. `https://jpetazzo.github.io/2015/09/03/do-not-use-docker-in-docker-for-ci/`, 2017. Accessed: 18.08.2017.

[102] Manuel Pichler. Phpmd - php mess detector. `https://phpmd.org`, 2017. Accessed: 11.07.2017.

[103] Inc. Red Hat. Automation for everyone. `https://www.ansible.com`, 2017. Accessed: 26.08.2017.

[104] Docker Saigon. Docker internals. `http://docker-saigon.github.io/post/Docker-Internals/`, 2016. Accessed: 23.07.2017.

[105] Ken Schwaber and Mike Beedle. *Agile software development with Scrum*, volume 1. Prentice Hall Upper Saddle River, 2002.

[106] Scrum.org. The home of scrum. `https://www.scrum.org`, 2017. Accessed: 26.08.2017.

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

[108] Josh Stella. An introduction to immutable infrastructure. `https://www.oreilly.com/ideas/an-introduction-to-immutable-infrastructure`, 2015. Accessed: 04.08.2017.

[109] Symantec. Wie vertraut sind sie mit dem gefahrenpotenzial folgender sicherheitsrisiken für ihr unternehmen? `https://de.statista.com/statistik/daten/studie/209376/umfrage/vertrautheit-mit-gefahren-durch-computerkriminalitaet/`, 2011. Accessed: 14.03.2017.

[110] Docker Security Team. Securing the enterprise software supply chain using docker. `https://blog.docker.com/2016/08/securing-enterprise-software-supply-chain-using-docker/`, 2016. Accessed: 14.03.2017.

[111] Docker Security Team. Docker security scanning. `https://docs.docker.com/docker-cloud/builds/image-scan/`, 2017. Accessed: 14.03.2017.

[112] KeePassX Team. The official keepassx homepage. `https://www.keepassx.org`, 2017. Accessed: 04.08.2017.

[113] GmbH Travis CI. Test and deploy with confidence. `https://travis-ci.org`, 2017. Accessed: 14.07.2017.

[114] James Turnbull. *The docker book*. Lulu. com, 2014.

[115] Unknown. Credentials storage in jenkins. `http://xn--thibaud-dya.fr/jenkins_credentials.html`, 2017. Accessed: 18.08.2017.

[116] VersionOne. The 10th annual state of agile report. `https://versionone.com/pdf/VersionOne-10th-Annual-State-of-Agile-Report.pdf`, 2015. Accessed: 14.03.2017.

[117] Daniel J Walsh. Are docker conatiners really secure? `https://opensource.com/business/14/7/docker-security-selinux`, 2014. Accessed: 14.03.2017.

[118] Don Wells. Pair programming. `http://www.extremeprogramming.org/rules/pair.html`, 2017. Accessed: 16.08.2017.

[119] Hartmut Werner. *Supply chain management*. Springer, 2000.

[120] Dave West, Tom Grant, M Gerush, and D D'silva. Agile development: Mainstream adoption has changed agility. *Forrester Research*, 2(1):41, 2010.

[121] Wade Williamson. Lateral movement: When cyber attacks go sideways. `http://www.securityweek.com/lateral-movement-when-cyber-attacks-go-sideways`, 2016. Accessed: 23.08.2017.

Appendices

.1 Application

Listing 1: application/code/index.html

```
<!DOCTYPE html>
<html>
        <body>
                <h1>Hello World</h1>
        </body>
</html>
```

Listing 2: application/docker/development/application/Dockerfile

```
FROM nginx:1.13

RUN rm /etc/nginx/conf.d/default.conf

COPY docker/development/application/html.conf /etc/nginx/conf.d/html.
   conf

WORKDIR /var/www/app/
```

Listing 3: application/docker/development/application/html.conf

```
server {
    listen 80 default_server;
    server_name _;
    index index.html;
    root /var/www/app;
}
```

Listing 4: application/docker/production/application/Dockerfile

```
FROM nginx:1.13

RUN rm /etc/nginx/conf.d/default.conf

COPY docker/production/application/html.conf /etc/nginx/conf.d/html.conf

COPY code/ /var/www/app

RUN chown -R nginx:nginx /var/www/app

WORKDIR /var/www/app
```

Listing 5: application/docker/production/application/html.conf

```
server {
```

```
    listen 80 default_server;
    server_name _;
    index index.html;
    root /var/www/app;
}
```

Listing 6: application/build.sh

```
#!/usr/bin/env bash

# Include exports
. ./exports

docker-compose -f docker-compose.yml -f docker-compose.prod.yml build
```

Listing 7: application/deploy.sh

```
#!/usr/bin/env bash

# Include exports
. ./exports

scp docker-compose.prod.yml ubuntu@swarmhost:/home/ubuntu/
ssh ubuntu@swarmhost 'docker stack deploy --compose-file /home/ubuntu/
   docker-compose.prod.yml application'
```

Listing 8: application/docker-compose.prod.yml

```
version: '3'

services:
    application:
        build:
            dockerfile: docker/production/application/Dockerfile
            context: .
        image: ${REPOSITORY}agilesec${VERSION}
        ports:
            - 80:80
```

Listing 9: application/docker-compose.yml

```
version: '3'

services:
    application:
        build:
            dockerfile: docker/development/application/Dockerfile
            context: .
```

```
        image: agilesec
        ports:
            - 80:80
        volumes:
            - ./code:/var/www/app
```

Listing 10: application/exports

```
export REPOSITORY="127.0.0.1:5000/"
#export REPOSITORY=""
export VERSION=":0.2"
export STACKNAME="application"
```

Listing 11: application/push.sh

```
#!/usr/bin/env bash

# Include exports
. ./exports

docker-compose -f docker-compose.yml -f docker-compose.prod.yml push
```

.2 Build Server

Listing 12: buildserver/docker-compose.yml

```
version: '3'

services:
    jenkins:
        image: slipke/jenkins:latest
        ports:
            - "8081:8080"
            - "50000:50000"
        volumes:
            - jenkins_home:/var/jenkins_home
            - /var/run/docker.sock:/var/run/docker.sock

volumes:
    jenkins_home:
```

.3 VCS Server

Listing 13: vcs/docker-compose.yml

```
version: '3'
services:
    gitolite:
        image: jgiannuzzi/gitolite:latest
        ports:
            - "2222:22"
        volumes:
            - gitolite-sshkeys:/etc/ssh/keys
            - gitolite-git:/var/lib/git
        environment:
            SSH_KEY: "<ssh_key>"

volumes:
    gitolite-sshkeys:
    gitolite-git:
```

.4 Registry

Listing 14: registry/docker-compose.yml

```
version: '3'

services:
    registry:
        image: registry:2
        ports:
            - "5000:5000"
```

Eidesstattliche Erklärung

Hiermit versichere ich, Simon Lipke, ehrenwörtlich, dass ich die vorliegende Masterarbeit mit dem Titel: *Building a Secure Software Supply Chain using Docker* selbstständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden. Ich habe die Bedeutung der ehrenwörtlichen Versicherung und die prüfungsrechtlichen Folgen (§26 Abs. 2 Bachelor-SPO (6 Semester), §24 Abs. 2 Bachelor-SPO (7 Semester), §23 Abs. 2 Master-SPO (3 Semester) bzw. §19 Abs. 2 Master-SPO (4 Semester und berufsbegleitend) der HdM) einer unrichtigen oder unvollständigen ehrenwörtlichen Versicherung zur Kenntnis genommen.

Stuttgart, den 31. August 2017

Simon Lipke