University of Bergen
Department of informatics

Security in Docker Swarm: orchestration service for distributed software systems

Author: Didrik Sæther
Supervisors: Tetiana Yarygina, Anya Helene Bagge

UNIVERSITY OF BERGEN
Faculty of Mathematics and Natural Sciences

August, 2018

Abstract

With a constantly increasingly number of services in modern software systems (SOA and micro services), managing such service infrastructure becomes a challenge. Docker Swarm is a popular service orchestration solution that addressed this issue. This makes it a target for attacks, as the orchestrator is entrusted with critical information for a system. This thesis investigates the security of Docker Swarm and the underlying technologies used for providing a secure orchestration service.

Despite the increasing popularity of Docker Swarm, the security properties of it are poorly understood. The security mechanisms that underpin Docker Swarm are not well documented if at all described. The custom protocols used in Docker Swarm for joining a swarm and rotating manager keys lack public security evaluation. This thesis aims to improve our understanding of the high-level security features of Docker Swarm by exploring several attack vectors that are likely to be pursued by a real-world attacker, such as MITM and DoS attacks. Results of investigation show that Docker Swarm provides a secure platform for service orchestration, as it is resilient towards selected high level attacks and follows best security practices.

Acknowledgements

First and foremost I wish to thank my supervisors, Tetiana Yarygina and Anya Helene Bagge. Your knowledge and guidance has provided motivation and insightful conversations during the development of this thesis, and thank you for helpful comments on the text.

I am grateful to all of those with whom I have had the pleasure to work with during this thesis and other related projects. The students and faculty at the Department of Informatics, University of Bergen have created an environment for engagement and excitement for Computer Science. I am grateful for the countless hours spent at the study hall with interesting discussions. A place where I have thrived. A special gratitude to the students at JAFU; We few, we happy few, we band of brothers.

Lastly I want to thank my family, friends and especially Helga, my girlfriend for supporting me during my time as a student, and for always encouraging me.

`Didrik Sæther`
July 31, 2018

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Problem statement and motivation

When preforming large scale software deployment, the problem of governing the software system quickly becomes an issue. In fast-paced agile environments, developers want to spend less time with administration, and more time writing code. For reaching this goal they resort to automation of tasks like scaling, provisioning and deployment. The problem is that there are many pitfalls in automating these tasks, especially in regard to security. This type of automation is called service orchestration.

We can compare it to the actual orchestration of music. Where the developer is the composer, the worker nodes are the musicians, the manager node is the conductor. There are many ways and motives for an adversary to interrupt the sweet music played by the orchestra. An attacker could throw rotten tomatoes. He could sit in the audience with a tape recorder and sell the bootleg recording after the show. Or he could be more Machiavellian and infiltrate the orchestra to play the wrong notes on his instrument with intent. The other musicians would abruptly kick him out after this, but if the attacker can impersonate as the conductor, he could wreak havoc in the orchestra, and the musicians would not dare to question his leadership.

The trust given to an orchestration service makes it a target for attackers. If it was to be successfully attacked, it would compromise a large portion of applications running in a container environment e.g., web services running on *container virtualization* in the cloud. We want to investigate how secure Docker's implementation of an orchestration service is, as it is currently the most popular tool for natively managing a cluster of Docker Engines. It also acts as an application programming interface (API) for Docker and operates with tools like Flynn, Kubernetes and Jenkins.

1.2 Related work

Research on the topic of service orchestration is emerging, as well as in the security of service orchestration. In regard to security analysis of modern container and data centre orchestration services such as Docker Swarm, Kubernetes, and Apache Mesos the work in academia and industry is scarce. What exists of academic work often mention the use of an orchestrator to create a secure system but does not go in detail on the orchestrator itself. Development of these systems are moving rapidly and the field of DevOps security[1] is a small subset of a subset of computer science, this can help explain the lack of formal work. Much of the information about Docker Swarm comes from the official documentation and talks/presentations on the topic, but as these are often as much of a sales pitch as they are informal, we need to verify the authenticity of these presentations. Despite the software being open source there is a lack of formal proof of security, information about the security features and the frameworks they depend on.

Most closely related to our work is the masters thesis *"Security considerations in Docker Swarm networking"* by Brouwers [2] from 2017 which analyses network security in Docker Swarm in respect to data plane traffic and overlay networks. Parts of this analysis are strongly related to what we look at in this thesis, though we have chosen to focus on management- and control plane traffic and the underlying protocols that Docker Swarm uses for creating and managing a swarm. Brouwers was not able to find security issues related to layer 2 (Data link layer) attacks.

Another master thesis, *"Security analysis of Docker containers in a production environment"* by Kabbe [21] also from 2017, reviews security in Docker containers in production environments, and tests known security vulnerabilities, like Dirty COW (CVE-2016-5195),

[1] *DevOps security* is an ambiguous term, as the community still is undecided in terms of calling it: DevOpsSec, SecOps, DevSecOps, SecDevOps or OpSecDev.

Heartbleed (CVE-2014-0160), Shellshock (CVE-2014-6271) and more. The results show that the security of container environments preforms equal or better in comparison with a hypervisor-based environment.

Dino Dai Zovi presented the talk *Datacenter Orchestration Security and Insecurity: Assessing Kubernetes, Mesos, and Docker at Scale* on BlackHat in the summer of 2017 (it was not published for non-attendants of the conference before spring of 2018), where he discussed security of data centre orchestration by showing attack models and evaluating the threats for Kubernetes, Apache Mesos and Docker Swarm. After showing a successful attack towards Docker Swarm and comparing it to the other orchestrators, he presented the security in Docker Swarm as the current "gold standard". For this talk there was not published any paper or article, nor was the source code for the successful attacks published. We have a successful demonstration, but no way to verify what was done. We have unsuccessfully contacted Zovi in an attempt to gain access to the source material.

Yarygina [42] discuss security concerns specific to micro service architecture and possible ways to address them. The paper elaborates why micro service security is a multi-facet problem which requires a layered security solution that is not available at the moment. The paper discuss Docker Swarm and Netflix's solution to mutual authentication of services using mutual transport layer security (MTLS). In this thesis we look deeper into the security of Docker Swarm.

The short research paper *"Analysis of Docker Security"* by Thanh Bui [3], though published in 2015 (before Docker Swarm was revised in 2016), shows that Docker containers are fairly secure, even with the default configuration. This is also relatable to Zovi's work, as it discusses the security implications of Docker as vulnerable to address resolution protocol (ARP) spoofing. Bui states that the security level of Docker containers could be increased if the operator runs them as "non-privileged" and enables additional hardening solutions on the Linux kernel, such as AppArmor or SELinux. We note that today's Docker Swarm is not comparable to the orchestration service in Docker from 2014, as it was rebuilt and split out to a standalone project in 2016 with version 1.12.

1.3 Goals and research questions

The overall goal of this thesis is to evaluate and explore the security features of orchestration services for container-based virtualization. In order to achieve this goal, we attempt to answer the following research questions:

Research questions:

- What security mechanisms are used in Docker Swarm?
- Is Docker Swarm secure against high level attacks?
- If the security is proven strong, what other applications can benefit from applying this model?

In addition to the research questions, we want to provide a set of specific requirements the Docker Swarm should comply with in order for us to have confidence in the security. These requirements are modelled after the SMART principle (Specific, Measurable, Attainable, Realistic, Traceable) for non-functional requirements, as this will provide us with test cases that are not covered by single tests i.e. unit tests. Merkow and Raghavan [26] define the need for non-functional requirements as:

> Gaining confidence that a system *does not do what it's not supposed to do* is akin to proving a negative, and everyone knows that you can't prove a negative. What you can do, however, is subject a system to brutal resilience testing, and with each resistance to an attack, gain increasing confidence that is was developed with a secure and resilience mindset from the very beginning. [26, Chapter 2.2]

Evaluation criteria:

The following requirements should be satisfied, but are not limited to:

- Access to a swarm is limited to an authorised node.
- Nodes in a swarm should have strong identity.
- Management and control traffic sent on the network should be encrypted when possible.

1.4 Chapter outline

Chapter 2 - Background on distributed systems
Introduces all concepts necessary to understand how distributed systems, containerised applications, and orchestration solutions function. continuous integration (CI) and DevOps are discussed as well.

Chapter 3 - Docker, Docker Swarm and related infrastructure technologies
Provides relevant information about Docker, Docker Swarm, Docker networks. We look at related technologies such as etcd from CoreOS, and what it provides for Docker Swarm. We also discusses the languages and frameworks that are involved in creating Docker Swarm.

Chapter 4 - Docker Swarm security
Introduces basic security concepts, and key elements from security relevant to a distributed architecture. In addition, it provides a basic understanding of the most common security threats which a containerised environment is exposed to. A detailed outline of security mechanisms implemented in Docker Swarm. We discuss disclosed security issues and known problems with the technology used to make Docker Swarm.

Chapter 5 - Experiments and results
In this chapter we describe the testing environment for capturing and proxying traffic in a swarm. We outline and explain the different experiments for verifying the goals and research questions defined in Section 1.3, reviews the results of these experiments.

Chapter 6 - Discussion and conclusion
We discuss the findings from Section 5, and give our opinion on the technology and architecture. We then concluding by summarising how Docker Swarm performs in regard to the research questions, and requirements listed in Section 1.3. We end by presenting a list of future work.

Chapter 2

Background on distributed systems

In this chapter we provide the background information necessary to understand how a distributed architecture works, and how containerisation of applications affect development and the administration prossess. The concepts of CI and DevOps practices for a fast moving development life cycle are also presented.

2.1 Distributed computing

The term *distributed computing* refers to a model where networked computers communicate and coordinate their actions only by passing messages. A distributed system is a concurrent system with no global clock and if designed and implemented correctly has independent failures. This means that the system is decomposed into parts that can run independently of each other. There is no notion of a correct time, only the execution order, and a failure is independent and isolated to that system component alone. For example in a client-server application when a client crashes, the server is not affected, and likewise if the server crashes, the client is not affected [6, 36]. A distributed system is not only client-server applications. Examples of other types of distributed systems are *peer-to-peer* that partitions tasks or workloads between peers, and *Network File System* that enable access to files over a computer network much like local storage is accessed [36].

2.1.1 Consensus protocols

One of the problems in fault tolerant distributed systems is *consensus*. Consensus involves multiple servers agreeing on values. Once servers reach a decision on a value, that decision shall be final. Typical consensus algorithms make progress when any majority of their servers is available; for example, a cluster of three servers can continue to operate even if one server fail, and if more servers fail, they stop making progress.

There are a couple of algorithms that try to solve this problem. Paxos is a family of protocols for solving consensus issues in distributed systems. Paxos is quite difficult to understand, in spite of numerous attempts to make it more approachable. Furthermore, its architecture requires complex changes to support real-world systems. As a result, both system builders and students struggle with Paxos [31].

The Raft Consensus Algorithm [31] is an alternative to Paxos. Raft is a fault tolerant distributed key-value store that solves the problem of consensus. Raft is equivalent to Paxos in performance and fault tolerance, but it separates the key elements of consensus (*leader election*, *log replication*, and *safety*) into smaller sub problems, and enforces a stronger degree of coherency to reduce the number of states that must be considered. The fault tolerance in Raft as Ongaro and Ousterhout states handles failure of up to $(N-1)/2$ nodes, but to make progress (agree on values) a quorum of $(N/2)+1$ nodes is required. For example, in a system with five manager nodes; if three of them are unavailable, existing tasks will keep running but the system cannot process requests to schedule new tasks.

An easy way to understand how Raft Distributed Consensus works, we recommend looking at Ben Johnson's website, as he has made an intuitive visualisation of how a cluster works in action[1].

Known implementations of Raft is etcd from CoreOS which is a highly popular tool for distributed key value store (see Section 3.3), and Kudu by The Apache Software Foundation which provides column-oriented data store. Many other applications depend on etcd an Kundu for realising their software, such as Docker Swarm which uses *etcd*. We note that Apache Mesos uses Paxos. Their implementation of a service orchestrator use concepts from Raft for avoiding problems such as livelock[2].

[1] `http://thesecretlivesofdata.com/raft/`

[2] Livelock is (in contrast to deadlock) a condition that takes place when two or more programs change their state continuously, with neither program making progress.

2.1.2 Containerised applications

Containerised applications are applications running in an operating system (OS)-level virtualized environment. Multiple isolated applications or services can run on a single host and access the same OS-kernel. This is in contrast to hypervisor-based virtualization where the OS may share the virtualized hardware resources. When a software system consists of multiple container-based applications it is by definition a distributed system as the applications communicate via the network, has no concurrency or a global clock, and will fail independently (given that the underlying OS itself does fail).

The advantage of using containers instead of hypervisors is higher server density by removing redundant or unnecessary operating system elements from the virtual machines (VMs) themselves. Containers provides high portability, by packaging VM stacks, and a design where the underlying hardware is insignificant. Fast start-up times facilitate a more flexible infrastructure allowing greater latitude to respond to the needs of the moment.

However, the use of a shared kernel does come with security implications. If a shared kernel is compromised, all of the instances that employ that kernel are potentially compromised. This is not unique for containers, as if a hypervisor is compromised the same security implications are true.

Containerisation of applications has a long history, and Docker is not the first to do it. In the early 2000s, FreeBSD Jails allowed administrators to partition a FreeBSD system to smaller systems called *jails*. With this, each jail could get assigned an IP address and configuration [22]. One year later, Linux VServer was introduced, and preformed many of the same tasks as FreeBSD Jails did, but also included partitioning of file systems, network addresses and memory [7]. In 2008, LXC (LinuX Containers) was introduced and was the first complete implementation of Linux container management. Docker was released in 2013 and used LXC in the initial stages before replacing it for its own library later [25, 27].

2.2 Orchestration

The definition of orchestration is the automation of coordination, management and arrangement of computer systems, services and tasks. The orchestrator becomes necessary for providing fast scaling and cluster management in order to coordinate running containers across multiple nodes (computers) in a cluster. For example, orchestration reduces the time and effort for deploying multiple instances of a single application. For software systems at scale a centralised management of the resource pool becomes crucial for maintaining a high server density as stated in Section 2.1.2. Erl [11] and Josuttis [19] ties orchestration closely to service oriented architecture (SOA) as a separate layer in the architecture. The rise of micro services and cloud computing amplified the need for service orchestration based on the ideas from SOA. The orchestration we are concerned about in this thesis depend on SOAs concepts, but rather than defining a separate layer in the application stack, orchestration is a service, and manages containers and the tasks executed by the containers.

Orchestration is closely related to distributed computing. As SOA and micro services are per definition distributed. In SOA, orchestration facilitates the automation of business processes by loosely coupling services across different applications [19]. Micro service is the architectural specialisation of SOA [42]. The architecture describes a particular way of designing software applications as suites of independently deployable services, driven by DevOps practices. We go deeper in detail about DevOps in Section 2.3. To summarise; service orchestration is the combination of service interactions to create higher-level business services.

There are multiple providers of orchestration services for containers. *Kubernetes* is a container orchestration tool, originally designed by Google and made open source in 2014. *Apache Mesos* is also an open source project, but focuses on cluster management designed to scale to very large clusters (In the order of hundreds to thousands of hosts). Mesos itself has no built in orchestrator, and uses *Marathon* as the orchestrator. *Docker Swarm* is an orchestrator for Docker hosts specifically. A swarm is created by by joining multiple machines into a cluster for providing multi-container, multi-machine applications. All these orchestrators have the following in common:

- *Ease of administration*
  Ease of administration provides an extensible architecture, where it is easy to incorporate in an existing infrastructure, with e.g., APIs.
- *Provisioning*
  The tool is able to provision or schedule containers within the cluster and launch them.

- *Configuration-as-text*
  The tool can load an application specification from a schema e.g., javascript object notation (JSON) or YAML.
- *Monitoring*
  The health of a container is to be trackable, and a crash or fail will be recovered by creating a new container or restarting the container.
- *Rolling upgrades and rollback*
  Rolling upgrades ensures that there is no downtime when a new version of the application is rolled out. This is achieved by creating redundancy in the number of healthy containers. Rollback ensures that if a configuration did not work as expected the applied change is reverted without downtime.
- *Policies for scalability*
  The tool is able to automatically scale an application based on e.g. the CPU usage of the containers, or numbers of users for a service.
- *Service discovery*
  In a micro service architecture, it is important that the orchestrator provides automatic detection of services offered by other containers.

2.3 Continuous integration and DevOps

We feel that it is important to mention continuous integration and DevOps as this has been one of the driving forces for creating container orchestration services. As we described in the introduction (Section 1.1) software developers want to spend more time with writing code, and less time with managing it[3]. The practice of CI is about collaborating developers merging their code frequently. Each merge is verified to detect integration errors at the earliest possible time. Building and testing the code reduce integration problems, but also helps develop cohesive software at a greater pace [18].

The two major phases in CI is to make sure the code compiles, and to ensure that the code works as designed. The first part can be achieved by the use of continuous build. Every time code is pushed to the single source repository a build is run to assert that the code

[3]We are aware that not every developer has to manage the code, and that in some cases this is done by system administrators.

compiles. The next part is the use of tests to programmatically validate the quality of the software. This can be achieved by the use of tests, specifically unit, API, and functional tests [34].

On the same topic we have DevOps where software engineering culture meets practice. The aim is at unifying software development (Dev) and software operation (Ops). The practice encourages automation and monitoring of the steps in software construction [30]. The steps in DevOps are illustrated in Figure 2.1. The orchestration service in combination with the underlying container-based virtualization provides us with an easy way to release, configure and monitor applications.

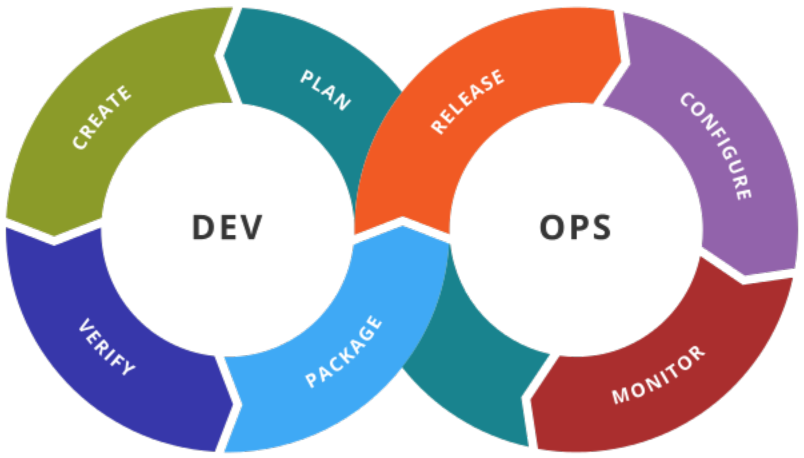


Figure 2.1: DevOps toolchain

Credit: Devops-toolchain by Kharnagy licence: CC BY-SA 4.0

Chapter 3

Docker, Docker Swarm and related infrastructure technologies

This chapter will provide relevant information about Docker and Docker Swarm and related infrastructure technologies. Docker Swarm is built using a variety of technology and frameworks. The use of Golang, gRPC remote procedure calls (gRPC) and Protocol buffers for creating applications is emerging, and a new architectural style that fits the model of cloud computing. We explore and debate the pros and cons of the architecture.

3.1 Docker

Docker is a software technology provider of container-based virtualization that is open source. Docker provides both upstream projects and downstream products as Figure 3.1 depicts. The illustration explains how the components of Docker are assembled into the finished product software that developers can use for running containerised applications. Developers of the Docker stack contribute to the individual components, that are built with the Moby project. Docker community edition (CE) is the product of a Moby build, and the Docker that most people know and use. Docker enterprise edition (EE) is another product of Docker, but not the focus of this thesis, as it is built from Docker CE but is closed source, costs money and focuses on enterprise support and features.

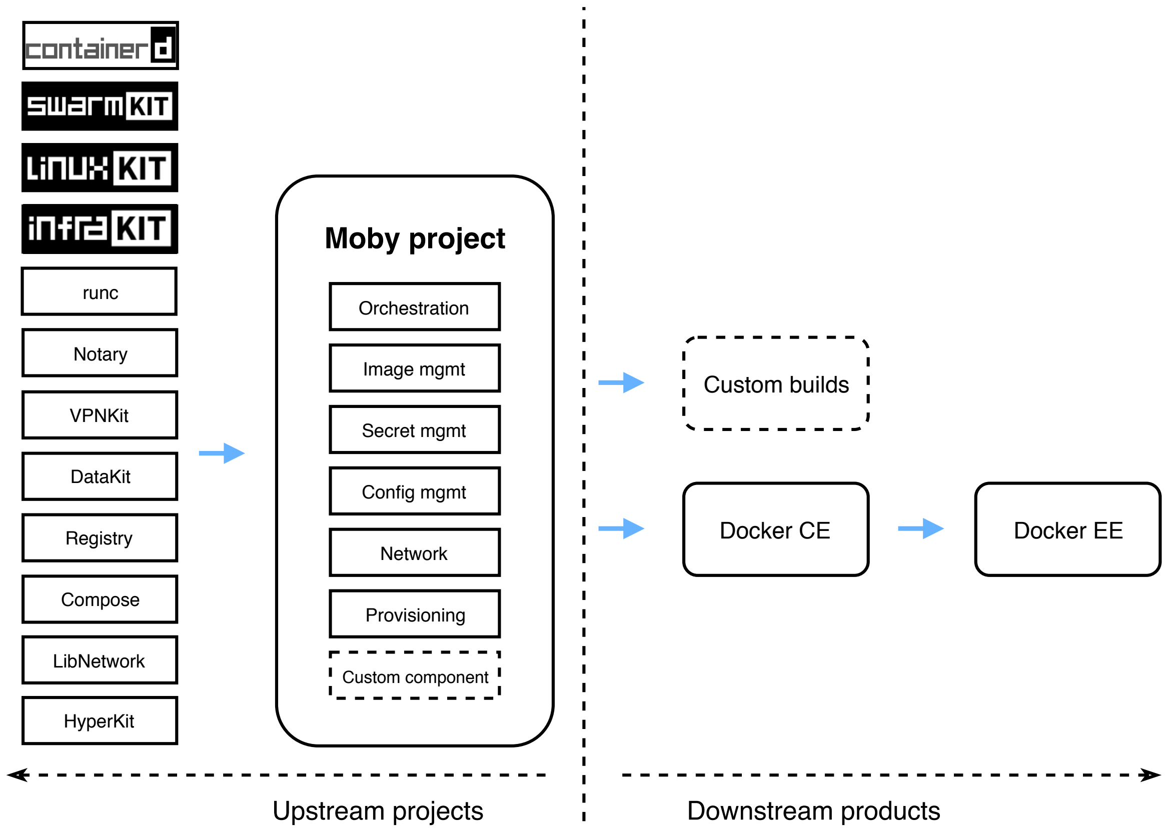


Figure 3.1: Docker upstream projects vs. downstream products
Image is adapted from Coisne [5]

Important upstream components:

- *containerd*
  A container runtime for managing a complete container lifecycle of its host system: Image transfer and storage, container execution and supervision, low-level storage and network attachments.
- *SwarmKit*
  A toolkit for natively managing a cluster of Docker Engines: includes primitives for node discovery, raft-based consensus and task scheduling.
- *LinuxKit*
  A toolkit for building secure, portable and lean operating systems for containers.

- *InfraKit*
  A toolkit for creating and managing declarative, self-healing infrastructure. It provides infrastructure support for higher-level container orchestration systems (e.g., SwarmKit).

3.2 Docker Swarm, SwarmKit, and Swarm mode

Docker Swarm provides the orchestration service for containerised applications. A swarm consists of multiple Docker hosts that run in a cluster. Hosts can be managers that preform membership and delegation tasks, and/or workers that run swarm services. An advantage of swarm services over standalone containers is that you can modify a service's configuration (including the networks and volumes it is connected to), without the need for a restart of the service at once. Docker will update the configuration, stop the service tasks with the out of date configuration, and create new ones matching the desired configuration.

To deploy an application to a swarm, a service definition is submitted to a manager node, which dispatches units of work called tasks to worker nodes. An example of this is illustrated in Figure3.2.

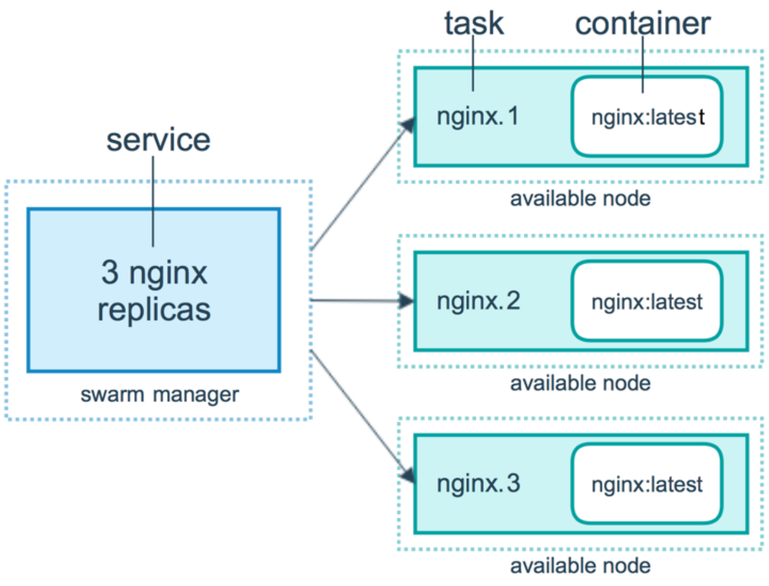


Figure 3.2: Example of a Docker Service illustrating service, tasks and containers

Credit: Docker Inc. `https://docs.docker.com/`

When Docker Swarm was released in 2014, it was an approach to automatically provisioning application clusters using CI and DevOps techniques. One of the objectives of Docker Swarm is to provide an orchestrator with security built-in. By using the principle of least privilege, each participant in a system must only have access to the information and resources that are necessary for its legitimate purpose, and nothing more [28]. We go further into detail about Docker Swarm security in Chapter 4.

In 2016 Docker released a major update with version 1.12 that split out the components of Docker into the upstream projects from Figure 3.1. One of these upstream projects is SwarmKit. Some of the new features were automatic scaling, rolling update, service discovery, load balancing, and routing mesh. The legacy functionality of standalone Docker Swarm is still included in the Docker Engine, but will probably be faced in the future according to the documentation [9]. When SwarmKit is integrated into Docker CE it is referred to as Swarm mode as it is cluster management integrated with the Docker Engine. The Docker command line interface (CLI) or API is used to create a swarm, deploy application services and manage swarm behaviour.

For this thesis, when using *"Docker Swarm"*, we refer to Docker SwarmKit as the standalone project integrated into Docker CE. We use SwarmKit to communicate with the Docker Engine. This means that in the examples we use SwarmKit's CLI and not the Docker Engine's own CLI, unless specified.

3.3 Etcd and raft logs in Docker Swarm

Etcd from CoreOS is an open source implementation of the Raft Consensus Algorithm (see Section 2.1.1) used in Docker Swarm. For the case we are concerned about in this thesis, etcd solves the two problems of leader election and log replication. In Docker Swarm the managers are part of a Raft consensus group. Raft elects a leader node that logs all actions, for instance when a new node is added or removed and when a new service is created. This log is then replicated to the other mangers, so that any other manager can overtake the leader role if the current leader becomes unavailable. The log file is a write-ahead log (WAL) file. Juggery [20] shows how it is possible to decrypt the WAL, to see what information is stored in it. An excerpt of what is stored in the log:

- Certificate authority (CA) config
    - Node Certificate expiry
- Root CA
    - CA Key
    - CA Certificate
    - CA Certificate hash
    - Join-tokens
        - worker
        - manager
- Transport layer security (TLS) info
    - Trust root
    - Certificate issuer subject
    - Certificate issuer public key

As we can see from the excerpt, critical security information is stored in the WAL. If this file is compromised, the swarm as a whole would be compromised. From the documentation of Docker Swarm [10] we also see that *Docker secrets* are stored in the WAL. This provides an elegant way to store information that needs to be shared between services (e.g., SSH Keys, Certificates, and usernames/passwords), but not with every service. The documentation states that "secrets are encrypted during transit and at rest in a Docker swarm. A given secret is only accessible to those services which have been granted explicit access to it, and only while those service tasks are running". The documentation does however not explain how they are encrypted.

3.4 Docker networks

For isolation, containers are assigned a network namespace. This is an isolated network stack with its own collection of interfaces, routes and firewall rules. Network namespaces are used to provide isolation between processes. Analog to regular namespaces they ensure that two containers, even if they are on the same host, won't be able to communicate with each other unless explicitly configured to do so. A diagram of the network stack can be seen in Figure 3.3

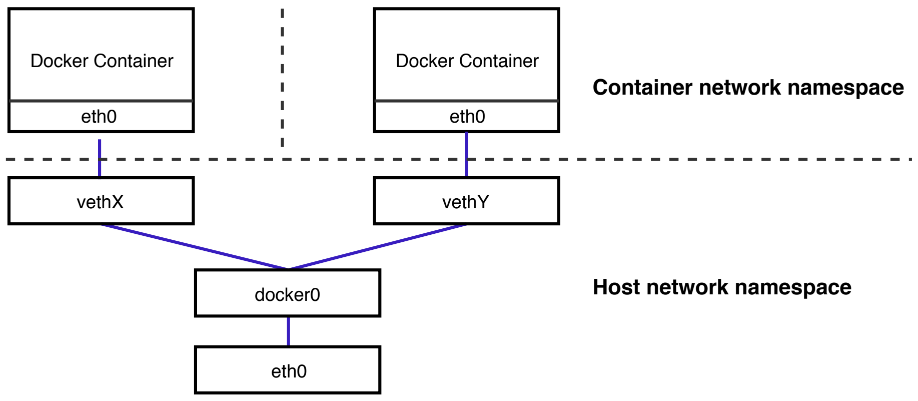


Figure 3.3: Docker single node networking

Image is adapted from Church [4]

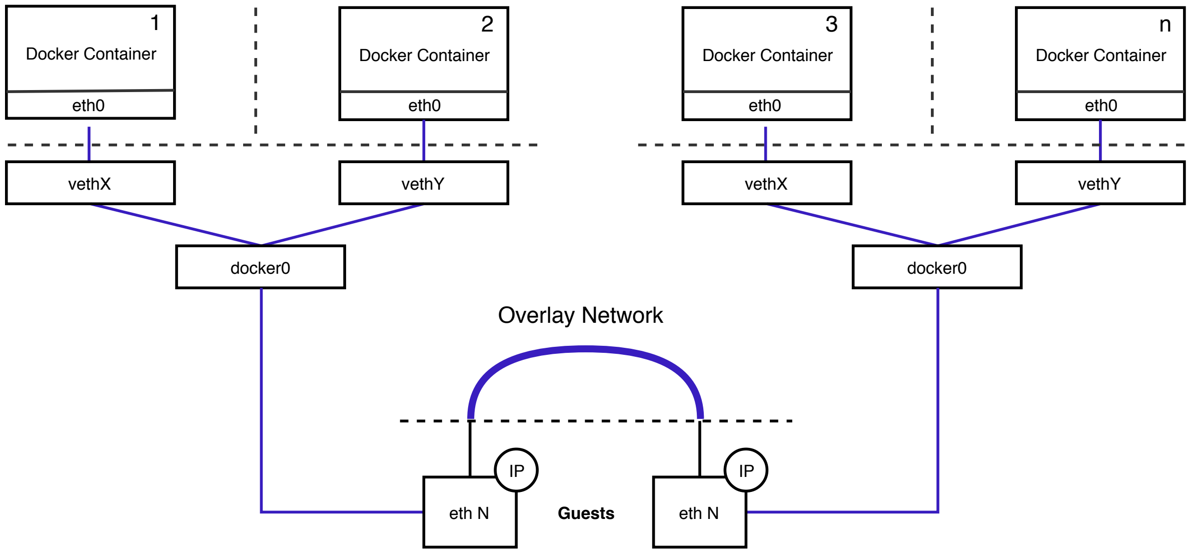


Figure 3.4: Docker multi node networking

Image is adapted from Church [4]

For communication between Docker daemons (for instance in a Swarm) a distributed network is created. This network overlays the host-specific networks and allows containers to connect to it to communicate securely. This is an extension of the single node networking (Figure 3.3), but with overlay network added for communication between the different daemons, as can be viewed in Figure 3.4. We note that these containers are not required to be on the same network. For example the container marked with 1 can reside on one network, and container 2,3 to n on its own subnet.

There are multiple ways of configuring networking in Docker, and it can be tailored to fit most infrastructures. The scope of this thesis will be the default networking stack provided by Docker.

3.5 Golang

When we investigated Docker Swarms security features, we were trying to figure out which implementation of TLS that was used. If an application is using a library or implementations with known errors or that are prone to downgrade attacks, this can compromise the security of the communication. For that reason we investigate the underlying technology of Docker Swarm to investigate if the architecture reliable.

Golang is a fairly young programming language created by Google in 2009. The current version is 1.10.1, almost nine years in the making, and currently at the top of its peak in popularity according to Google Trends as shown in Figure 3.5, of the period 01.10.2009 – 15.06.2018. Google Trends ranks a search result based on the interest over time on a scale from 0-100 where 100 is the point in time when it was most popular[1].

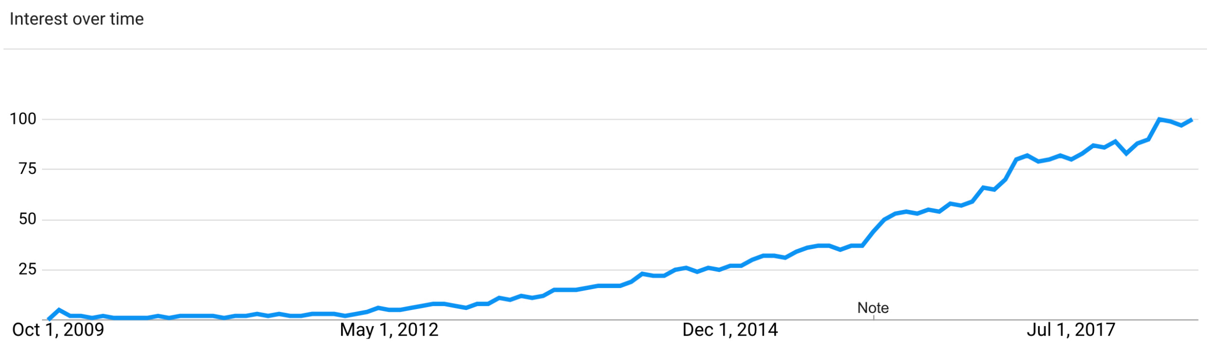


Figure 3.5: Golangs rising popularity on Google Trends with the search term “Golang”. Note: An improvement to the data collection system was implemented on 01.01.2016

[1]`https://trends.google.com/trends/explore?date=2009-10-01%202018-06-15&q=Golang`

The language paradigm is not straightforward as it falls into many categories. The consensus is that it is multi-paradigm: procedural, concurrent, and allows an object-oriented style of programming according to the Golang FAQ and Zhang [14, 44]. The language is made to be simple to learn, as the developers deliberately limited the scope of the language and built something that an experienced developer can learn in a short time. Simplicity does however come with downsides, as the lack of generics and exceptions can discourage developers with background in using those features. The syntax is similar to `C/C++`, and it does not run the code on a VM, and hence will use less resources than e.g. Java, Scala and Kotlin that are all JavaVM languages. The cross-platform support is still better than many non-VM languages.

The built-in concurrency has the same static execution speed as `C/C++` which enables the developers to carry out many processes at the same time. Golang is statically typed and provides type safety, this is an advantage in regard to security. In Go, every variable must have a type associated with it, and the language also requires a developer to dedicate attention to error handling. The compiler is strict, which again leads to a secure and robust code. Another useful safety feature is a garbage collector, which languages like C++ lacks. Built-in testing facilitates the process of testing your code.

A new language is not always a good thing, even if the ideas are good. Languages like Esperanto, Ada and The Magnolia Language (we realise that one of these is a natural language, and the other two are programming languages, but it is still a good comparison), all had applications that would improve the current state, or solve a problem within existing languages. Where they failed was in getting critical mass. There are a number of reasons why a language has low adaptation rate. Either the syntax is too complicated, some language feature is missing or the cost of dealing with the complexity is high.

The number of developers of the language is also very important. We want to mention Golang and Kotlin that are developed by Google and JetBrains respectably, as open source projects. We accredit the backing and development from large corporations with the reason why these languages are currently rated as the some of the most popular according to GitHub [17]. *Octoverse* looks at the number of pull requests for determining the popularity of a language. The need for an active development community is important as the synergy effect will drive popularity for a language.

3.5.1 Golang TLS library

The implementation of TLS in Golang's standard library (`crypto/tls`) is a partially implementation of TLS 1.2, as specified in RFC 5246. Adam Langly (Google Inc.), one of the maintainers of OpenSSL, is in charge or security libraries for Golang, and is an active contributor to the project[2]. One of the reasons for implementing their own security library, and not employ existing libraries such as OpenSSL, was a wish by Google to have Golang written in Go[3]. Rob Pike, co-designer and developer of Golang and co-author of *The Unix Programming Environment* and *The Practice of Programming* summarises the 10th anniversary of the creation of Go with this quote about Adam Langly's work[4]:

> Adam did a lot of things for us that are not widely known, [...] but of course his biggest contribution was in the cryptographic libraries. At first, they seemed disproportionate in both size and complexity, at least to some of us, but they enabled so much important networking and security software later that they become a crucial part of the Go story. Network infrastructure companies like Cloudflare lean heavily on Adam's work in Go, and the internet is better for it. So is Go, and we thank him.

By not using OpenSSL, and with a minimum TLS-version of 1.2, Golang is immune against known attacks such as POODLE (CVE-2014-3566) and Heartbleed (CVE-2014-0160). However, a golden rule is to not create your own encryption scheme, as homemade cryptography is generally considered to be more prone to bugs, and likely hasn't been scrutinised by many other researchers or tested in the wild. This is further discussed in Section 6.

3.6 Protocol buffers

Protocol buffers provide a language-neutral, platform-neutral, extensible mechanism for serialising structured data.

[2]`https://github.com/golang/go/commits?author=agl`

[3]`https://groups.google.com/d/msg/Golang-Nuts/0za-R3wVaeQ/BAE0HbjLdpEJ`

[4]`https://commandcenter.blogspot.com/2017/09/go-ten-years-and-climbing.html`

In the early 2000s, the use of HTTP and XML offered a self-descriptive, language agnostic and platform independent framework for remote communication. This combination resulted in the standardisation of SOAP and Web Services Description Language (WSDL) that promised interoperability among various runtimes and platforms. The SOAP standard with HTTP and XML was for many developers too restrictive. This resulted in the developers moving to JavaScript and JSON, where APIs played a key role, and JSON replaced XML as the preferred wire-format protocol. This combination of HTTP and JSON resulted in the unofficial standard REST. SOAP was confined to large enterprise applications that demanded strict adherence to standards and schema definitions, while REST was a hit among contemporary developers.

As an alternative to the message formats of SOAP and REST, Protocol buffers provide efficient serialisation, a simple interface definition language (IDL), and easy interface updating. All complexity of communication between different languages and environments is handled by gRPC. Messages have to be encoded/decoded as the structure is not self describing without the definition-files. Figure 3.6 show the structure of Protocol buffers and how messages can be encoded/decoded.

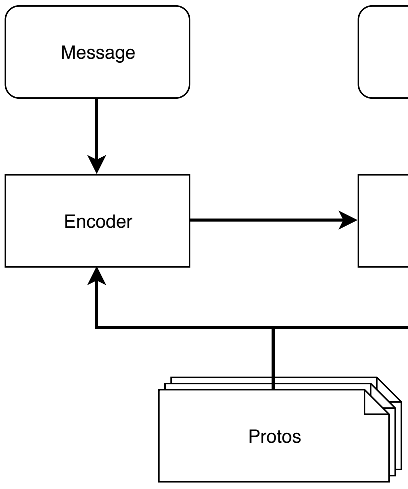


Figure 3.6: Structure of Protocol buffers

With Protocol buffers, the developer creates a `.proto` description (file) of the data structure, see Listing 3.1. From that, the Protocol buffer compiler (`protoc`) creates a class that implements automatic encoding and parsing of the Protocol buffer data in a binary format.

The generated Golang class for Listing 3.1 is shown in Listing A.1 in the Appendix. The generated class provides getters and setters for the fields that make up a Protocol buffer and cater details such as reading and writing the Protocol buffer as a unit.

When defining a `.proto` file for a service, we define the request and response as shown in Listing 3.1. This means that the client and server must agree on guarantees about the safety and interoperability between those architectures. The finer points of platform specific data types should be handled in the target language implementation.

Listing 3.1: *.proto*-file for GCD-application defining the request and response

```
syntax = "proto3";

message GCDRequest {
    uint64 a = 1;
    uint64 b = 2;
}
message GCDResponse {
    uint64 result = 1;
}
service GCDService {
    rpc Compute (GCDRequest) returns (GCDResponse) {}
}
```

When using Protocol buffers for generating the client code for the languages Golang, C++ and Java, there are implemented various safeguards to protect against corrupt or malicious data. For instance there are limits to the overall message size provided by the Protocol buffer library as well as methods like `CodedInputStream::SetTotalBytesLimit;` for C++ that sets the maximum number of bytes that this CodedInputStream will read before refusing to continue. There are also recursion limits to prevent deeply nested messages from overflowing the stack and memory exhaustion prevention (most specifically from receiving messages that indicate a huge length-delimited value).

3.6.1 Comparing Protocol buffers to other message formats

Protocol buffers have many advantages over XML for serialising structured data. According to van Winkel [41] and the documentation [13], Protocol buffers are simpler to use, 3 to 10 times smaller and 20 to 100 times faster, and less ambiguous. The problem with SOAP and WSDL is that they are inextricably tied to XML, so that if a new serialisation format would arise, it would not be able to move over to that format. The Protocol buffer format supports the idea of extending the format over time in such a way that the code can still read data encoded with the old format. XML Schema Definition (XSD) used for validation of XML documents has also seen a lot of criticism for being too complicated and lacks proper support for backwards compatibility. This is not the fact with Protocol buffers.

In our opinion Protocol buffers can replace XML in large enterprise applications that demand strict adherence to standards and schema definitions, as it supports the needs of this domain, but still offers flexibility, a simple to understand message format and high performance. Industry leaders such as Cisco, Square and Google rely heavily on Protocol buffers, which demonstrate that Protocol buffers are capable of performing in enterprise contexts.

Comparing JSON to Protocol buffers, Protocol buffers show many advantages, such as type safety. Analysis of size and speed also suggest that Protocol buffers are much faster in terms of encoding/decoding and size of the data on the wire. Krebs [23] shows the average performance of 500 `HTTP GET` requests issued by one Spring Boot application to another Spring Boot application with JSON and Protocol buffers. Figure 3.7 shows that Protocol buffers outperforms JSON with 78% less time for non-compressed transfers, and 83% for compressed transfers.

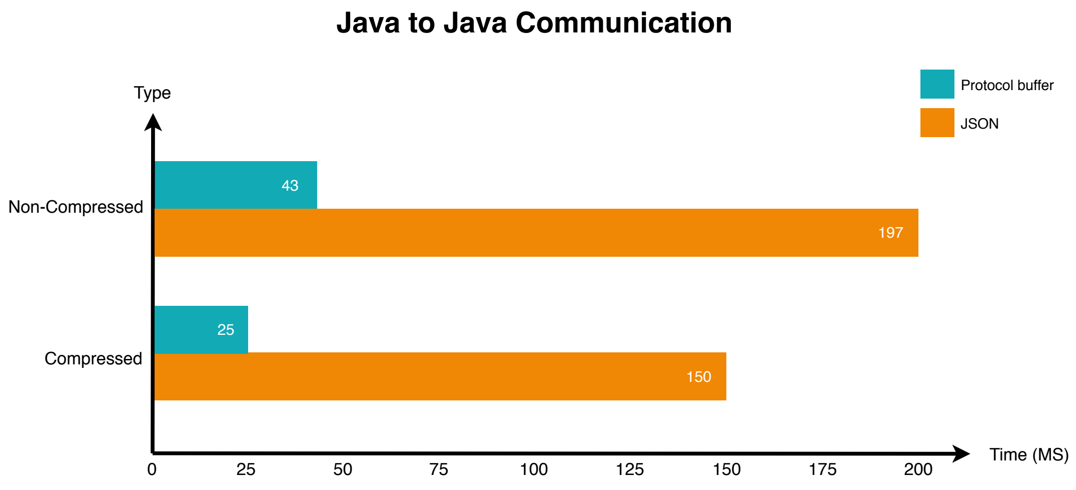


Figure 3.7: Comparison of Protobuf/JSON performance on GET requests Java to Java

Image is adapted from Krebs [23] [23]

In web services, JSON has been used for many years for making RESTful APIs, as it has many advantages as a data interchange format; It is human readable and typically performs well, but it is not without issues in a RESTful context. The specification is not enforced by a machine, which has resulted in confusion amongst developers on what methods, payloads and response codes really mean. This makes the RESTful pattern unpredictable.

Debugging RESTful APIs is not easy either. A complete transaction consists of many pieces (HTTP request method, request address, request payload, response code and response payload). That means that the developer can end up spending valuable time searching for a bug in multiple locations, for just determining where the bug resides.

However most importantly in regard to this thesis; RESTful is not secure. According to Yarygina [43] REST lacks a general agreement on how the REST paradigm addresses security and what web security mechanisms adhere to the REST style. Yarygina also states that message passing is complicated. Protocol buffers are immune to some of the problems that arise with the RESTful style with JSON. Since it provides us with type safety, and a strict message format, it becomes easier for the developer to create secure applications.

Protocol buffers does not provide a perfect substitute for XML and JSON, as there are some disadvantages. For instance, Protocol buffers would not provide a good way to model a text-based document with markup (e.g. HTML) because there is no easy way to interleave structures with text. An advantage of XML and JSON is that it is human-readable and human-editable; Protocol buffers, at least in their native format, are not. Though Protocol buffers are claimed to be self-describing, this is only true if you have the message description (`.proto`-file). Given an arbitrary Protocol buffer, there is no way to know the correct way to deserialise/parse it unless you also have the corresponding descriptor. One could argue that this provides some security by obfuscation. Protocol buffers are also not intended for carrying large amounts of data, like JSON can. The recommended max message length according to documentation is 1 MB.

3.7 gRPC

gRPC is the open source version of Google's internal RPC (Remote Procedure Call) framework used for their infrastructure, called Stubby. Van Winkel [41] describes that gRPC is based around the idea of defining a service, specifying the methods that can be called remotely with their parameters and return types. gRPC is not a standard, so there is no academic paper or article about it. Articles mention using the framework, but does not analyse it, or go further in detail. The documentation shows examples of how to implement functionality but does not document how they work.

By default, gRPC uses Protocol buffers as an IDL for describing the service interface and the structure of the payload message. Protocol buffers provides us with many advantages as described in Section 3.6. For transport gRPC uses HTTP/2, a binary fully multiplexed protocol. We go further in detail about HTTP/2 in Section 3.8. The framework also offers pluggable authentication and load balancing, as we demonstrate with the experiments in Section 5.5.

gRPC provides four types of service methods by leveraging HTTP/2 as the technology underneath, and as such benefits from the efficiencies of HTTP/2:

- Unary RPCs
  Single request from client with a single response from the server.
- Server streaming RPCs
  Single request from the client with a stream in response.
- Client streaming RPCs
  Stream request from the client with a single response from the server.
- Bidirectional streaming RPCs
  Both client and server send a sequence of messages using a read-write stream. Streams operate independently, so clients and servers can read and write in different orders.

The use of bidirectional streaming means that we can reduce the number of TCP-connections to one between two services. This removes the problem with head-of-line blocking, where protocols based on HTTP/1.x is a FIFO queue that suffers from requests getting blocked on high latency requests in the front of the queue. Google boast that by harnessing the power of bidirectional streams and a binary protocol they are able to handle $O(10^{10})$ RPC per. second [38].

The workflow for gRPC is interesting as well. When developing with Protocol buffers, and gRPC the Protocol buffer compiler generates code for the languages that will implement the client and server. This reduces the risk of mismatch of interfaces when developing for several languages (as micro service architectures often are) and provides interoperability between polyglot applications. Figure 3.8 illustrates the gRPC workflow. We see that the following steps corresponds with what we have described in Section 3.6

1. The first step is to define the service endpoint and the structure of the payload messages in the Protocol buffer definition file.
2. By using the compiler `protc`, code is generated for the language of choice. Generating code for Golang is as easy as running the command:
   `protoc -I . --go_out=plugins=grpc:.  *.proto`
3. When the code is generated, the server and client application can be implemented.

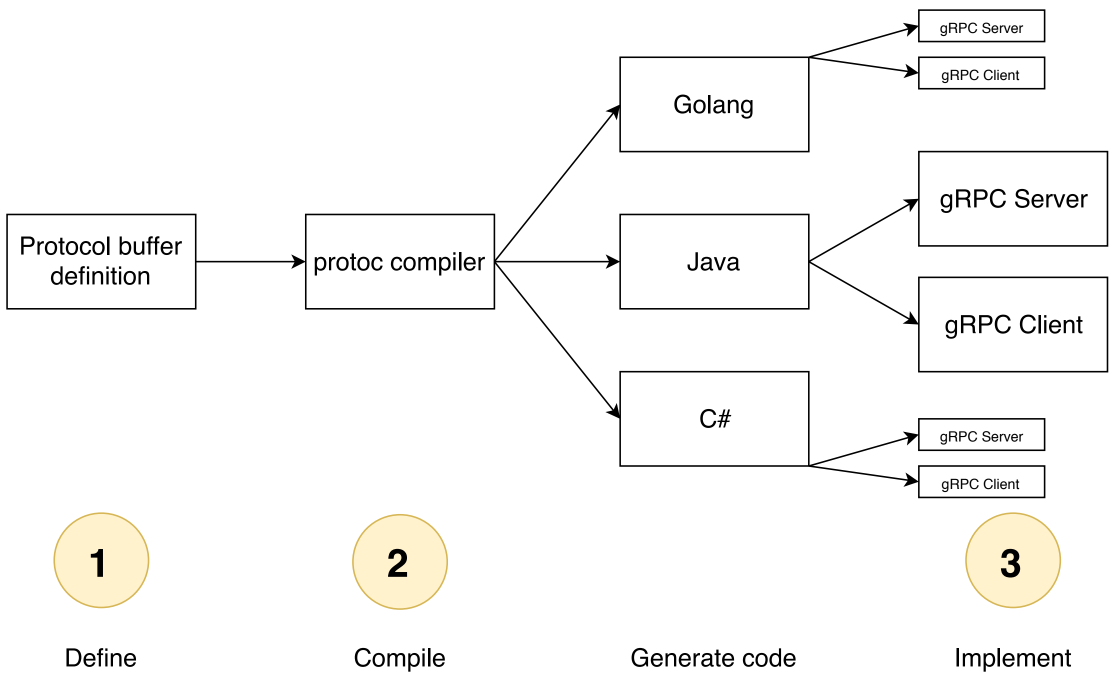


Figure 3.8: Workflow of gRPC

Image is adapted from Varghese [39]

gRPC is not a universal tool that can replace REST in a heartbeat. The fact is that the learning curve is steep, and user-friendly tools such as `curl` for making HTTP requests and testing APIs are not available as of now. The textual nature of JSON/REST gives it an advantage when it comes to human readable data compared to Protocol buffers and gRPC. Load balancing is still a hot topic in the development of gRPC. As for now, it is hard to figure out if load balancing should be done on client or server side, and with both pros and cons of each model [5,6]. We note that Docker Swarm will not generate enough traffic on the management and control plane to need load balancing. OpenAPI and its precursor Swagger is still textual, which makes is easy for developers to make JSON/REST APIs. In contrast, gRPC focuses on delivering the highest performance and quality for APIs by asking developers to follow a tight, well defined methodology that provides powerful streaming API constructs and high-performance implementations.

3.7.1 gRPC security

Security in gRPC is equivalent with authentication, as the security features are that of the implementation language. Currently the supported authentication mechanisms are SSL/TLS and a generic mechanism to attach metadata based credentials to requests and responses, such as *OAuth2 tokens*. From the documentation [12] gRPC provides an authentication API based on the unified concept of *"Credentials objects"*. These objects can be either *channel credentials* (typically SSL/TLS-credentials) or *call credentials* that are attached to a specific network call. Optional mechanisms are available for clients to provide credentials (i.e. certificates) for mutual authentication. We show in Section 5.5 how to implement authentication for a client-server application in Golang with gRPC.

3.8 HTTP/2

HTTP/2 is a rewrite of the HTTP/1.x protocol and is formed by the specifications RFC 7540 and RFC 7541. Some of the key differences are that HTTP/2 is binary, instead of textual. It is fully multiplexed, instead of ordered and blocking and uses header compression to reduce overhead. The use of a binary protocol and header compression results in less data being

[5]`https://grpc.io/blog/loadbalancing`

[6]`https://blog.bugsnag.com/envoy/`

transferred. The binary protocol provides less overhead when parsing data and is less prone to errors. This eliminates response splitting attacks, that can occur in textual protocols.

The number of TCP-connections are also reduced, where a single TCP-connection is used to ensure effective network resource utilization despite transmitting multiple data streams. Modern browser implementations only support HTTP/2 with TLS (Mozilla Firefox and Google Chrome). This means that HTTPS outperforms HTTP/1.x because of the multiplexing. This is also an advantage for mobile users, where battery life is important [16].

Chapter 4

Docker Swarm Security

In this chapter we focus on some of the key security elements in distributed architectures; trust, public key infrastructure (PKI), and certificate management. We also inspect the security in containerised environments, as there are special considerations for this domain. We provide information about the security features of Docker Swarm, as this information is not easily obtainable, and disorganized. Furthermore, we discuss known security problems with the technology that can affect a swarm, such as disclosed security flaws from CVEs, denial-of-service (DoS) and malicious container images.

4.1 Security basics

4.1.1 Trust on first use

Trust on first use (TOFU) is a security model used by clients that needs to establish a trust relationship with an unknown or not-yet-trusted endpoint. In a TOFU model, the client will try to look up the identifier, often a public key, in a local trust database. If no identifier exists yet for the endpoint, the client will either prompt for the user to determine if the client should trust the identifier, or it will simply trust the identifier which was given and record the trust relationship to its trust database. If a different identifier is received in subsequent connections to the endpoint the client will consider it to be untrusted. This is often solved with a human interaction to verify the identity of the not-yet-trusted endpoint, but this does not scale well. An example of TOFU is found in SSH. When connecting to a new host for the first time, the user is prompted to manually verify key fingerprint, and accept the certificate. This does not scale well because automation would defeat the security of the manual verification.

4.1.2 Mutual authentication and PKI

Mutual authentication and mutual transport layer security is a way to enforce both entities of a communication to authenticate each other and communicate over an encrypted channel. A well-designed mutual authentication solution is a mitigation strategy for man-in-the-middle (MITM) attacks.

HTTPS (TLS over HTTP) is extensively used on the web for providing data confidentiality and integrity, as well as authentication of the communicating parties. The web browser verifies the identity of the web server based on the digital certificate. If the server is to verify the identity of the client (to provide mutual authentication), the most common approach is to verify a user name and password on the application layer. This is the primary way used for verifying that the website a user is visiting is the correct one, the server to authenticate the user, and ensure the communication is encrypted.

The use of certificates as credentials (instead of user names and passwords) allows the authentication to happen on the transport layer instead of application layer. This is useful in a micro service architecture, where it is important that both parties are authenticated, and complexity of user-credentials is undesirable. Section 7.4.6 of the specification of TLS 1.2 (RFC 5246) outlines the use of client certificates for MTLS. It does not however have strict requirements for what information is to be verified, and in which way. The most up-to-date RFC to this topic is the draft for *OAuth 2.0 Mutual TLS Client Authentication and Certificate Bound Access Tokens*[1].

In a PKI a common problem is root of trust. We see this often in hierarchal PKI-designs where commercial root CAs like DigiCert, Symantec and GoDaddy are issuing certificates to end-entities, i.e., `ebay.com` or `uib.no`. A visitor of `uib.no` can trust that he is visiting the

[1]`https://datatracker.ietf.org/doc/draft-ietf-oauth-mtls/`

correct site (and that the connection is encrypted), because he (the web browser) trusts the root CA, in this example: DigiCert. This form of hierarchal trust can have intermediate CAs but is always ending in root of trust. If the root CA gets compromised, all intermediate and end-entities are compromised as well. An example of this happened in 2011 when DigiNotar had a security breach [33]. Not long after the breach DigiNotar filed for bankruptcy. We see how Docker Swarm tries to mitigate this in Section 4.2.2

4.1.3 Certificate management and PKI

Managing certificates and keys is a difficult task with many pitfalls. Without proper lifecycle management the time consumed on rotating certificates manually was, according to a case study done at Cisco, estimated to four hours of management per certificate. They also showed that manual management increased the risk of human error, which in turn resulted in security and operational risks[2].

The agility of the system is also an important topic, where operational changes, such as migration across data centres or responding to unanticipated events (e.g., Heartbleed (CVE-2014-0160)) and accelerated end-of-life of SHA-1 hashing (CVE-2005-4900)) that require a rapid certificate replacement, are almost impossible to respond to by using a decentralised and manual management system.

The disadvantage of using certificates is that when a certificate has been issued, it will have to be rotated at some point. Reasons for this can be that it is near the expiration date, or that a certificate has been compromised and has to be revoked. Revocation is the process of invalidating a certificate before its expiry date. Previous practices for revocation involved using a certificate revocation list (CRL) (RFC 2459). The nature of the list will make it grow in size, to a point where it will have to be cached, rather than downloaded by the client for every lookup. This results in unnecessary large amounts of data being transferred, and if cached it can result in making security decisions on stale data.

Other methods of checking for revocation has been *online certificate status protocol (OCSP)* (RFC 2560), *OCSP stapling* (RFC 6066) and *OCSP must staple* (RFC 7633). These

[2]Case study: Scalable Key and Certificate Lifecycle Management with Cisco Systems. Session ID: SP01-303, RSA Conference 2011, Cisco Systems Inc. https://www.venafi.com/assets/pdf/ss/RSAC2011_Case_Study_Cisco_Systems.pdf

methods still have problems in regard to man-in-the-middle attacks as stated by Langley [24] and are designed for server-to-browser environments rather than service-to-service.

Rivest [35] proposed in 1998 to eliminate certificate revocation. This was revisited by Langley [24] the creator of the cryptographic libraries in Golang (see section 3.5.1) where he refines the idea of certificate revocation. He urges to cease using revocation, and if it must be done, employ *OCSP must staple* (RFC 7633). Instead he advocates the use of short lived certificates. This is also supported by Payne [32] at Netflix, who argues that expiration of a certificate is revocation in itself. Payne also describes why using short lived certificates in a service-to-service environment is the better option than in web browser-to-service environments. Acer et al. [1] at Google reports that a mismatch between the certificates expiration date and client clocks is a widespread problem in Google Chrome. Payne [32] has shown that clocks in a service-to-service environment can often be more in sync as they often reside within the same data centre, and is not affected in the same way as web browsers are to the lack of a global clock.

4.1.4 Security in containerised environments

Security in containerised virtualization is a topic that needs to be addressed with an in depth defence strategy. To prevent container escapement and decrease the attack surface compared to traditional hypervisor virtualization, some of the approaches employed are

- *Kernel namespaces* that provides an isolated view of the system from other containers.
- Each container gets its own network stack to prevent privileged access to sockets or interfaces of other containers.

Systems requiring a higher level of security, Docker images can be extended with an extra layer of security by enabling AppArmor, SELinux and Tomoyo Linux, or other hardening systems [3]. For further hardening, Docker uses digital signature verification of all official repositories to facilitate publisher authentication, image integrity and authorisation. We write more about this in Section 4.4.3.

Managing a cluster of containerised services (e.g., in a micro service architecture) brings new security challenges, as the applications in the containers need to exchange information, and as we saw in Section 2.2, to efficiently manage these large clusters an orchestration service is often required. The orchestrator controls applications running in the containers, and accordingly must be protected. According to Mónica [28], "The distributed nature of orchestrators and the ephemeral nature of resources in this environment makes securing orchestrators a challenging task."

4.1.5 Attack models for containerised environments

Containerised environments have multiple attack vectors for how it can be attacked. Some of these are location-based, and some are classified based on the effort required by the attacker. The following list classifies and describes some of these attacks and mitigation strategies in ordered form of difficulty of mitigation:

1. Location-based attacks

   1.1. *An external attacker*

   An external attacker is someone on the outside of the firewall of the internal network, with the target of acquiring access to the internal network. One way to mitigate this threat is by using a positive security model, where an application is whitelisting ports and protocols, and everything else is denied.

   1.2. *An internal attacker*

   An internal attacker is someone on the inside of the firewall with intent of acquiring privileged information. The attacker can have gained access through other parts of the network, or breached physical security, but can also be an employee. This is mitigated by using fine-grained authorisation when defining services on the network.

2. Based on abilities

   2.1. *A passive attacker*

   A passive listener on the network is someone having access to the internal network and is listening to every communication with intent of acquiring privileged information. This attack is mitigated by encrypting everything on 'the wire'.

2.2. *An active attacker*

Active attacks are often aggressive and highly malicious in nature. Often locking out users, destroying memory or files, or forcefully gaining access to a targeted system or network. Malware, phishing, DoS-attacks and attacks directly towards services are common conventions. Both a passive and active attack employ MITM attacks, but the difference is that the active attacker alteres or spoofs the traffic (tampering). Mitigation for active attacks are the same as for the location-based attacks, but also intrusion detection and prevention systems.

3. Privilege-based attacks

   These attacks are active in the sense that they require an action from the attacker such as tampering.

   3.1. *A compromised non-privileged node*

   In a cluster where an attacker has gained access to and is controlling a non-privileged node (for Docker Swarm a *worker node*), mitigation is done by having privileged nodes pushing tasks to the workers, as opposed to them requesting tasks. In that sense limiting what a compromised node can access of data.

   3.2. *A compromised privileged node*

   A cluster where a privileged node (for Docker Swarm a *manager node*) has been compromised, would be the most difficult task to mitigate, as the privileged nodes control the cluster. One current solution to this is using a signed specification, where the developer signs what he/she wants to happen. The worker nodes do not need to trust the privileged nodes but verifies the signatures from the developers. This way of removing trust from the orchestrator itself is reasonable, as the orchestrator can still perform some orchestration decisions, but it does not get to decide what is running where, which worker node is to perform the task or what resources it can utilise.

The attacks listed here are directly transferable to the orchestra example given in the introduction of this thesis in Section 1.1. The external attacker is comparable to the tomato thrower, the passive listener is the person with the recording equipment, the compromised worker node is the Machiavellian instrument player, and last but not least; the compromised manager node is the rogue conductor.

4.2 Security features in ”Joining the swarm”

Docker Swarm employs a PKI with X.509 certificates for providing security and strong cryptographic identity for nodes in a swarm. As well as a join-token with TOFU built in. The manager node is a CA, and generates a key-pair which is used to sign the X.509 certificate. A node wanting to join the swarm will download the root certificate from the manager, then generate its own key-pair and submit a CSR for the manager to sign. The manager will verify the join-token before issuing the X.509 certificate with the role of the node (*worker* or *manager*) in the organisational unit field of the certificate. This certificate is used to authenticate both parties and secure communications with MTLS. For large scale installations a dedicated/external CA is often preferred, but not necessary. We show a system sequence diagram (SSD) in Figure 4.1 of how a worker node verifies the identity of the manager, and is issued a signed certificate when joining a swarm.

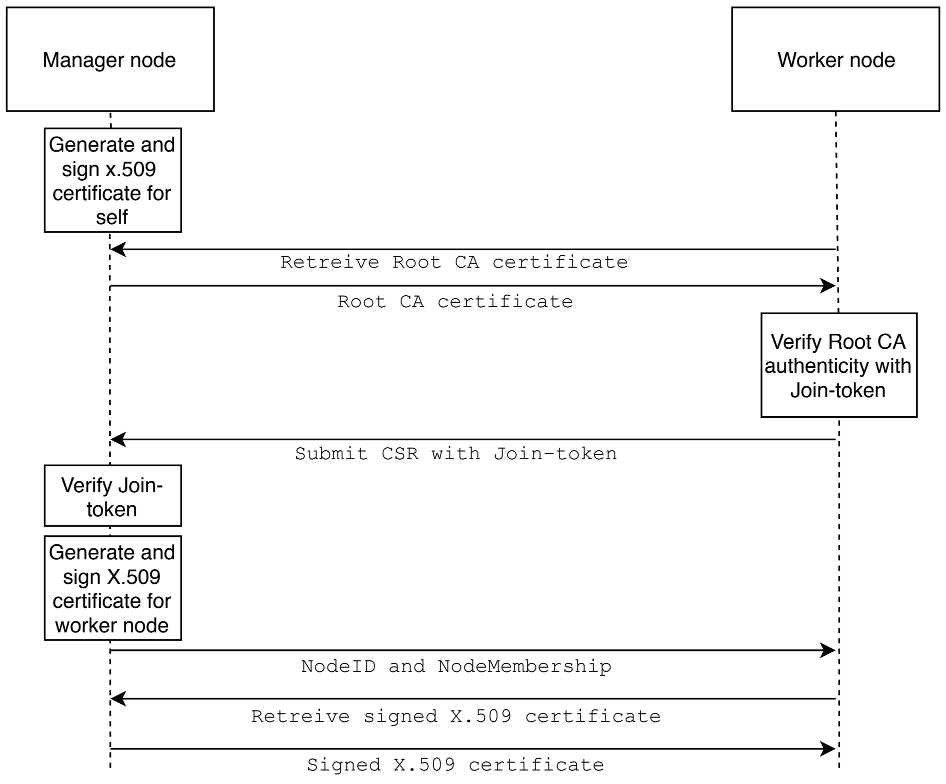

Manager node
Worker node
Generate and sign x.509 certificate for self
Retreive Root CA certificate
Root CA certificate
Verify Root CA authenticity with Join-token
Submit CSR with Join-token
Verify Join-token
Generate and sign X.509 certificate for worker node
NodeID and NodeMembership
Retreive signed X.509 certificate
Signed X.509 certificate

Figure 4.1: System sequence diagram of bootstrapping a new worker node

In Section 4.1.3 we discussed the use of short lived certificates as an alternative to revocation lists. In Docker Swarm a certificate issued to a node with Docker Swarm is short lived. The default is three months, but this can be configured to rotate as often as every hour [8], and the certificate lifecycle is fully automated. This enables scalability and fast moving development but is also a mitigation strategy for compromised certificates and keys. If credentials are compromised or leaked, the window of time an attacker can make use of those credentials is much smaller.

An example of a certificate for a manager node is shown in Listing 4.1 where we in line 8 see three interesting things:

- O (Organisation) is set to be the `Swarm ID`
- OU (Organisational Unit) is the `Node role`: `swarm-manager` or `swarm-worker`
- CN (Common Name) is set to be the `Node ID`

Listing 4.1: Cryptographic node identity for a swarm node in X.509 certificate. Ellipsis (...) indicate truncation. A full version of the certificate can be found in Appendix D.1

```
 1 $ openssl x509 -in swarm-node.crt -text
 2 Certificate:
 3   ...
 4     Issuer: CN=swarm-ca
 5      Validity
 6       Not Before: Jun  4 14:14:00 2018 GMT
 7       Not After : Sep  2 15:14:00 2018 GMT
 8     Subject: O=j8e2jv0sms..., OU=swarm-manager, CN=c52rwgu9uy...
 9       ...
10       X509v3 Subject Alternative Name:
11         DNS:swarm-manager, DNS:c52rwgu9uy..., DNS:swarm-ca
12     ...
13 ----BEGIN CERTIFICATE-----
14 MIICNjCCAdugAwIBAgIUXnx0uzww3B5z
15 ...
```

By using X.509 certificates signed by the CA for distinguishing a node in the swarm, we can assert that the node has a strong cryptographic identity as long as we trust the CA.

4.2.1 Securely joining a swarm

When introducing a node to a swarm (sometimes referred to as bootstrapping), Docker uses a token for joining a node to a cluster. If a joining node presents a valid join-token, the manager verifies the join-token, and authorises the node to be issued a certificate. This join-token is stored in the *Raft store* as part of the encrypted WAL that is handled by etcd (see Section 3.3). Figure 4.2 shows an example of what a token looks like, and what the four pars of a token are:

1. A known prefix. This makes it easy to detect if a token has been compromised by searching for it in a version control system (VCS) or logging system.

2. Token version number. This makes it easy to manage multiple versions of a token.

3. SHA256 hash of the CA's root certificate. The use of this hash means that the joining node can verify that it is joining the correct swarm.

4. Randomly generated secret. Secrets can be rotated at any time and the managers are in sync using the Raft consensus algorithm, so that any manager can verify that a token is valid. We examine the secret in the experiments (Section 5.3).

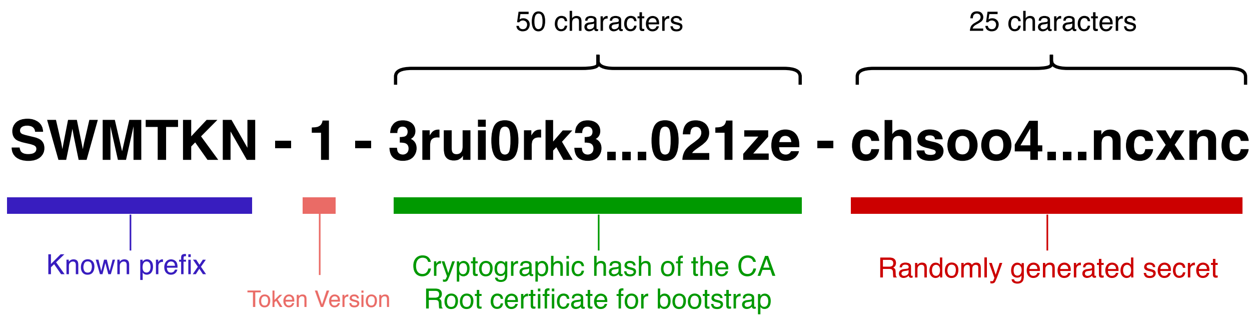

50 characters
25 characters
SWMTKN - 1 - 3rui0rk3...021ze - chsoo4...ncxnc
Known prefix
Token Version
Cryptographic hash of the CA Root certificate for bootstrap
Randomly generated secret

Figure 4.2: Docker Swarm secure token example

Image is adapted from Mónica [29]

As we saw in Section 4.1.1, TOFU can become a problem as it scales poorly. This has been solved by Docker Swarm with the use of the hash of CA's root certificate in the join-token. This enables the client to verify that the manager it is connecting to, is in fact the correct manager.

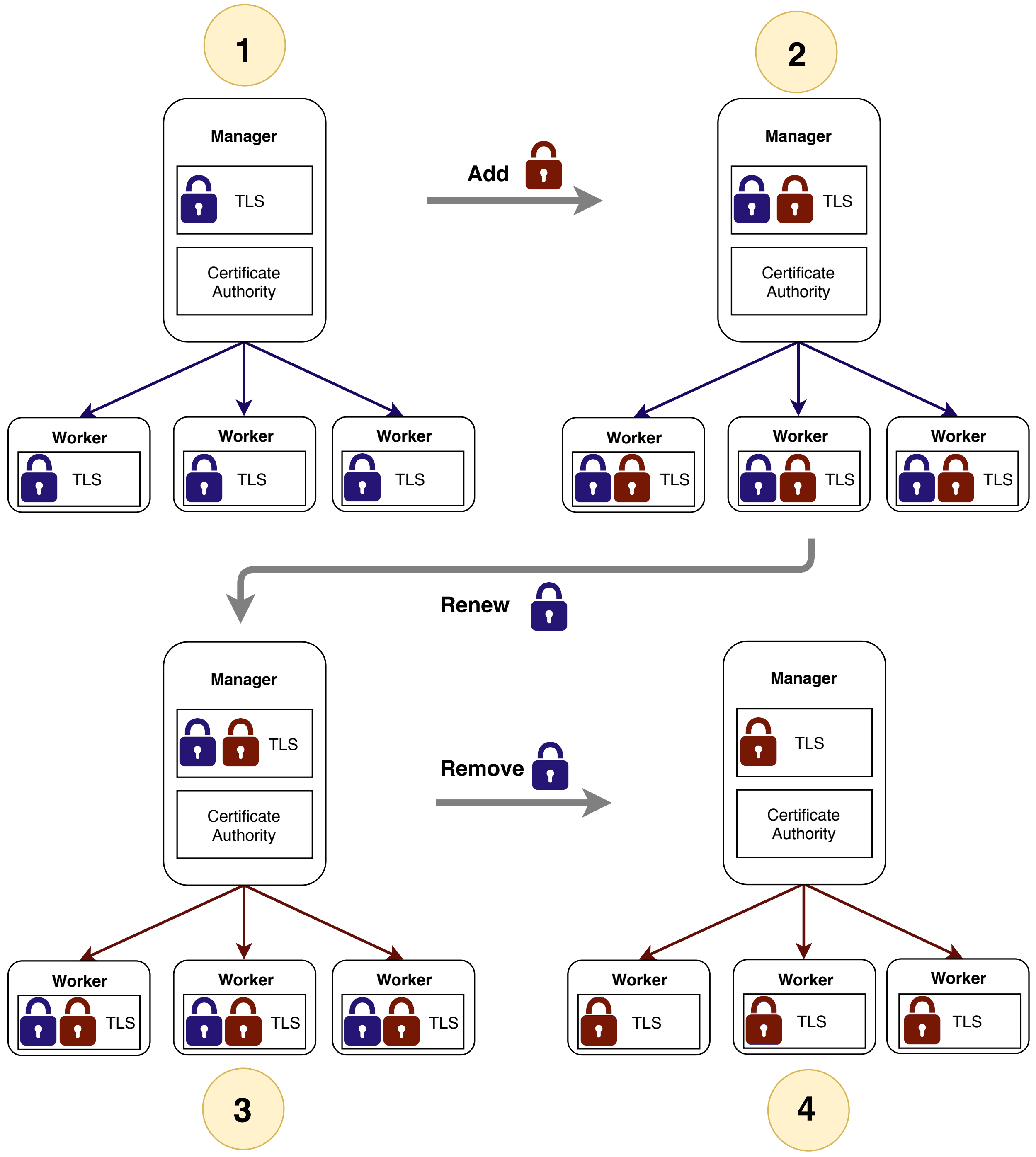


Figure 4.3: Visualisation of transparent root rotation.
Numbers in yellow correspond to the enumeration of steps in 4.2.2 for rotating root CA
Image is adapted from Mónica [29]

4.2.2 Root trust

As described in Section 4.1.2 the root of trust for a PKI is a critical component. For Docker Swarm the root of trust is the Root CA private key and certificate. It is the CA that is the workers root of trust. For an attack scenario where the root of trust (a privileged node) has been compromised, Docker invented transparent root rotation. If there is suspicion or proof that the root of trust has been compromised, a new root of trust can be added, and it is done in four steps. Figure 4.3 visualises the following procedure:

1. Managers and workers trust the compromised root of trust (visualised with the blue lock).
2. A new root of trust is introduced (visualised with the red lock).
   The managers issue certificates for themselves with new root of trust. Trusting both roots.
3. Managers force a certificate rotation of all nodes, enforces new root of trust, but does not remove the trust of the compromised root.
4. When every node has root of trust to the new root, the trust for the compromised root is removed.

We note that it is impossible to guarantee that a malicious attacker with access to a root key cannot wreak havoc, but it is possible to limit the exposure of the system. As we see it, this is a step in the right direction, though we are uncertain how much it would help.

4.3 Communication security

As we saw in Section 3.4, Docker has its own network stack with namespaces for each container. In addition to this Docker Swarm uses three planes for communication, as illustrated in Figure 4.4, *Management plane*, *control plane*, and *data plane*..

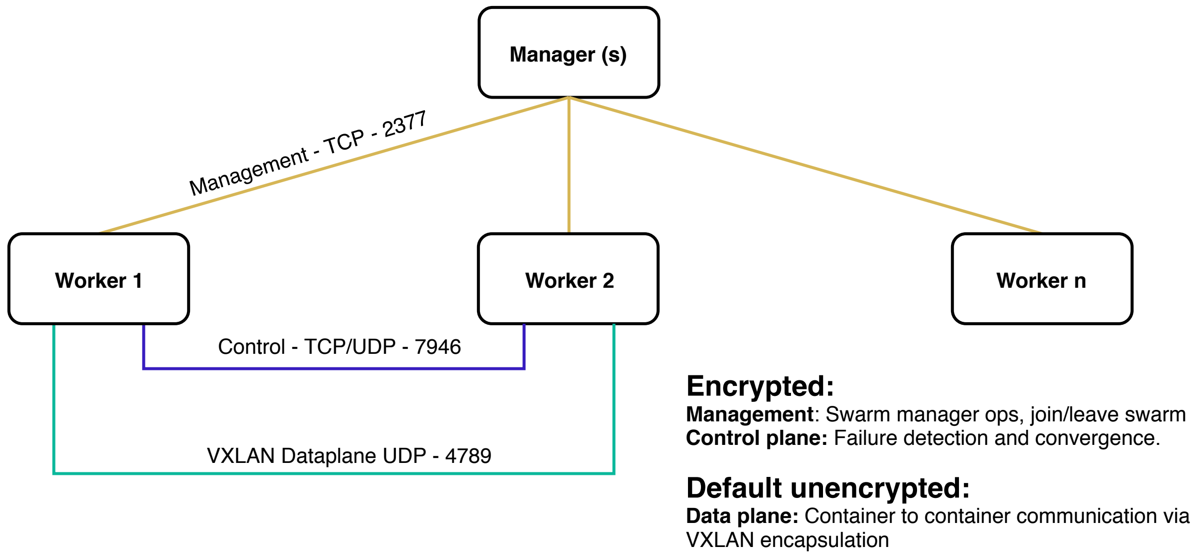

Manager (s)
Management - TCP - 2377
Worker 1
Worker 2
Worker n
Control - TCP/UDP - 7946
VXLAN Dataplane UDP - 4789
Encrypted:
Management: Swarm manager ops, join/leave swarm
Control plane: Failure detection and convergence.
Default unencrypted:
Data plane: Container to container communication via VXLAN encapsulation

Figure 4.4: Docker network planes
Image is adapted from the reference implementation by Church [4]

All swarm service traffic (management and control plane) is encrypted by default, using the AES algorithm in GCM mode. The management traffic uses TCP as transport protocol and the framework gRPC is used for management and control plane traffic. gRPC (see Section 3.7) uses HTTP/2 with mutual authentication and MTLS with X.509 certificates. This means that both parties of a connection is authenticated as stated by Venugopal [40]. This is also addressed by Mónica [28], for securing clusters against an attack model where an attacker controls the underlying communications networks or compromised cluster nodes.

Data plane traffic however is not encrypted by default. When enabled, IPSec encryption at the level of the vxlan is created with the same AES and GCM security as with the swarm service (control and management) traffic. This was confirmed by Zovi et al. [45], and the keys are rotated every 12 hours, according to Brouwers's master thesis on the topic [2]. It is important to mention that enabling encryption for the data plane imposes a non-negligible performance penalty. For enabling encryption on the data layer, Yarygina and Bagge [42] found a penalty of 11% in the experiments. The tests where done with a limited data set and did not test for difference sizes. We predict that bigger chunks of data will provide a higher penalty in performance.

4.4 Known security problems with the technology

Docker Swarm uses a variety of technologies and libraries. Some of these are vulnerable to known attacks. We list some of the problems and discuss them where relevant.

4.4.1 Common Vulnerabilities and Exposures

CVEs provides a reference-method for publicly known information-security vulnerabilities and exposures. Known CVEs for technology used by Docker Swarm is:

- CVE-2017-7860, CVE-2017-7861, CVE-2017-8359 and CVE-2017-9431 are related to gRPC out-of-bounds write. All have been mitigated after release 1.2.2 (current version is 1.11.0 (gorgeous)).
- CVE-2014-7189, CVE-2016-3959, CVE-2017-1000097 and CVE-2018-7187 are related to Golang. Some of them show weaknesses in the implementation of crypto and secure transport layer communications. All CVEs have a common vulnerability score of less than or equal to 5.0 (of a maximum of 10.0). All have been mitigated after release 1.7.4 (current version is 1.10.1).
- CVE-2016-6595 is related to SwarmKit version 1.12.0 (current version of Docker CE is 18.03.0-ce[3]) that allows remote authenticated users to cause a DoS. The CVE has since been disputed by Docker[4].

We discuss the CVEs more in Section 6.1 as they are disclosed and mitigated at the time of writing this. We also investigated the implementation of the custom TLS library in Golang in Section 3.5.1

[3]Starting with release 17.03.0-ce, Docker uses a monthly release cycle with YY.MM versioning scheme.
[4]`https://github.com/moby/moby/issues/25629`

4.4.2 Denial-of-service-attacks on Docker Swarm

Zovi et al.[45] demonstrated a DoS-attack on Docker Swarm by "disabling" the network connectivity of legitimate worker nodes by hijacking their IP-address. The attack is constrained to having root access on the underlying host of the swarm. This can be achieved by container escapement or other security vulnerabilities on the host. As it is necessary for retrieving the join-token from memory on the compromised host. The author is specifying that this is not the most elegant technique, as it involves ARP spoofing and DoS on the network level.

What happens is that by hijacking the IP-address of a legitimate nodes, the nodes are blocked from contacting the manager, and the manager node will set the legitimate nodes *status* as unavailable. The result of this, is that the rogue node is reassigned all work for the services. It is important to mention that the rogue node does not get the cryptographic identity of the nodes they hijack the IP-address from, but instead uses the join-token to get issued its own certificate. Nor that the rouge node can pick task from the orchestrator, it just gets assigned all tasks as it is the only node available. This attack matches the attack of a compromised worker node from Section 4.1.5.

4.4.3 Malicious container images

The images run on Docker hosts can be malicious, as Docker container images can originate from disk or from a remote registry. A popular remote registry is Docker Hub, Docker's own repository. Which hosts over 100 thousand public and private images. There are reported multiple malicious images on Docker Hub[5,6]. These images are obfuscated as legitimate applications, but contain crypto miners, or malware. Examples are: `dockmylife/memorytest/` that mines cryptocurrency under the pretence of being a memory testing software. Three other images: `docker123321/tomcat/` and `docker123321/kk/` had more than 100 thousand pulls and `docker123321/cron/` had 1 million pulls from Docker Hub before they where removed.

Pulling an image uploaded by a malicious user from a large registry is one thing, but the images are also transferred over an untrusted medium, for instance over the internet.

[5]`https://github.com/docker/hub-feedback/issues/1121`

[6]`https://www.fortinet.com/blog/threat-research/yet-another-crypto-mining-botnet.html`

Docker provides a verification system that helps guard against MITM attacks, as it prevents an attacker from secretly forging or tampering with content. To provide trust in the images, Docker Content Trust can be enabled for client-side verification of integrity and publisher for images. This lets publishers sign their images, and consumers ensure that the images they use are signed.

Docker Content Trust is based on Docker Notary tool to publish and manage trusted content and as Honnavalli [15] writes: "The engine behind enforcing and managing this trust is Docker Notary, a Docker service which is implemented based on The Update Framework (TUF)". TUF was created in 2010 as a framework and specification that can be used to secure new and existing software update systems. The project is hosted and developed by The Linux Foundation as part of the Cloud Native Computing Foundation (CNCF).

At some point the system administrator or developer must select an image from a trusted software vendor from a trusted registry, the rest Docker takes care of. This feature is disabled by default, as it will force all registry operations the use of signed and verified images. This thesis will not look into the implementation of Docker Content trust.

Chapter 5

Experiments and results

In this chapter we list and explain the experiments that we perform in order to gain confidence in the requirements, and try to answer the research questions we listed in Section 1.1: What security mechanisms are used in Docker Swarm? Is Docker Swarm secure against high level attacks? If the security is proven strong, what other applications can benefit from applying this model? We also attempt to answer the evaluation criteria:

- Access to a swarm is limited to an authorised node.
- Nodes in a swarm should have strong identity.
- Management and control traffic sent on the network should be encrypted when possible.

We start by defining our testing environment and give a detailed introduction to the various test cases we create from these goals and criteria.

5.1 Defining the experiments

As described in Section 4.1.5, there are multiple attack models which can compromise a swarm. It is therefore important to verify if there are mitigation strategies in place to prevent these attacks. These attack models are permissive and do not provide us with good test cases, but we can provide some test cases that will test one or more of them. We investigate the following procedures and network calls:

1. Evaluate cryptographic strength of the join-token as it provides access to the swarm, but the finer details of the generation are not documented. In neither the official documentation or in the code-base.

2. Investigate what an active attacker can do in a Swarm. Attempt MITM attack to view the request and response sent between nodes in Docker Swarm to verify the security of joining a node to a swarm. For testing this, we use different tools, such as mitmproxy, tcpdump and Wireshark.

3. We develop a custom proxy-tool to act as an intermediate for traffic with Protocol buffers and gRPC, to investigate if it is feasible to proxy the traffic. If this proves fruitful, a custom proxy for Docker Swarm can be developed specifically.

4. Dissect and evaluate the custom protocol for joining the swarm. When a new node joins an existing swarm. How is the new node issued an identity, and what security features are used to securely introduce a new node. For investigating this, it will be necessary to view the documentation and source code.

5. Investigate possibility of input fuzzing on the manger node. Is it possible to consume resources on an existing swarm by crafting a malicious input? We generate a set of misformatted join-tokens and a CSRs with a malformed and a fake payload. This will reveal problems with input validation, or increase confidence in that input validation is done correct.

5.2 Experiment setup

To be able to run through the tests as painless as possible, we prepared by setting up all of the underlying tools beforehand. This section describes which tools we used, which versions and how they were set up. The tests are performed on SwarmKit[1] master-branch from the GitHub-repository of Docker. The repository was cloned on 19.04.2018 [2]. The reason for date and commit to timestamp our cloning is that here is no release schedule for SwarmKit in the same way as there is with Docker CE and Docker EE. Instead, SwarmKit uses branches named after the upcoming release of Docker CE, and merges to the master branch on release.

[1]`https://github.com/docker/swarmkit/`

[2]Commit identification `33d06bf5189881b4d1e371b5571f4d3acf832816`

Current master branch is a bump from version 18.03. The test computer is an MacBook Pro from 2015 with MacOS High Sierra 10.13.3 installed, and the shell installed is fish[3] version 2.7.1.

For modifying and building SwarmKit and creating the proxy-tool described in Section 5.1, a Golang development environment is required. We installed Golang (`version go 1.9.4 Darwin/amd64`) and defined the system variables needed for building the code with

```
$ set -x -U GOPATH $HOME/go
```

5.2.1 Virtual machine

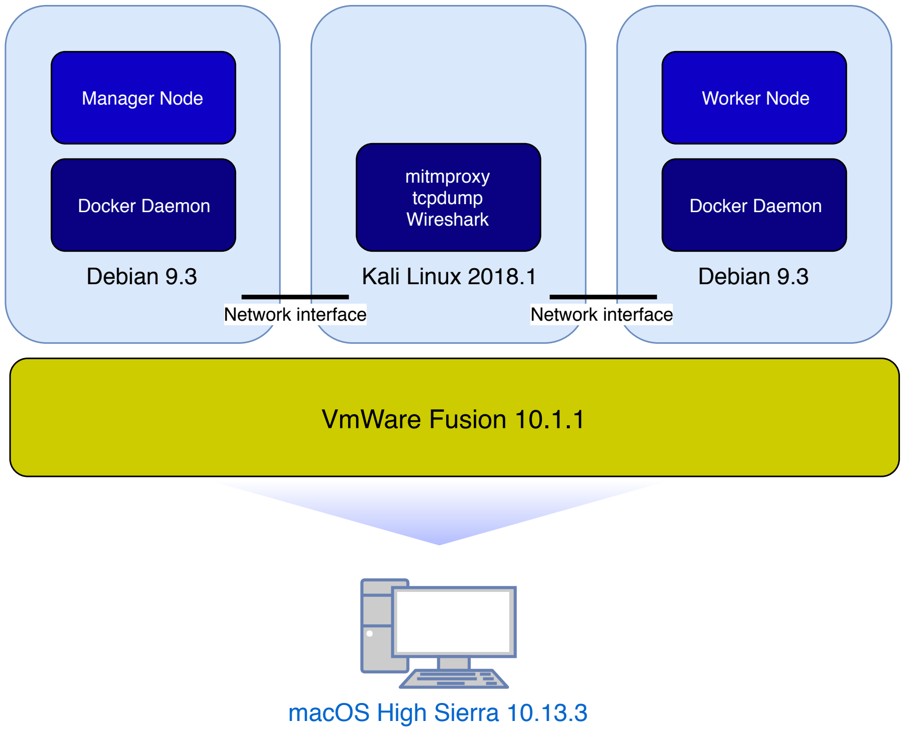


Figure 5.1: Setup of virtual machines for testing MITM attacks

The VMs we used for the setup were created by setting up three VMs in VMWare Fusion. The reason for this is that it provides isolation, and it removes noise (irrelevant network

[3]`https://fishshell.com/`

traffic) generated by other hosts on the network. The virtual environment consists of two Debian 9.3-machines working as a manager and a worker node, and in between them, a machine running Kali Linux 2018.1. The choice of operating system was that the writer of this thesis is familiar with the tools in the operating systems, and they are pretty lightweight, so that they do not slow down the host machine.

The network interface cards (NICs) from the Debian machines were routed through the Kali Linux-machine for easy interception, where the Kali Linux-machine has two NIC; one for each of the Docker hosts. Figure 5.1 shows a visualisation of our setup.

5.3 Cryptographic strength of the join-token

The generation of the secret part of the join-token is done securely using `"crypto/rand"` from the standard library of Golang. We have included the function for generating the join-token in the Appendix C.3. From this function we can see that the generation of the token does not use the whole hash of the certificate from the root CA, but instead it is a hash in base 36, 0-left-padded to 50 characters, and the secret is 16-byte in base 36 0-left-padded to 25 characters. The 0-padding adds no actual zero-values as Base36 encoding requires 6 bits to store a single character. The random generated sequence is 128 bits ($16 * 8$ bits), and the number of characters in Base36 encoding is:

$$\frac{\log 2^{128}}{log 36} = 25\ characters$$

In Section 4.2 and Figure 4.2 we show the join-token and what the different parameters of the join-token are. The first parameters of the token: Known prefix, token version number and the hash from the RootCA, can be guessed or attained, and the only secret part is the randomly generated secret of 25 characters. This secret consists of characters from only two of the categories of characters; *lower case letters* and *numbers* (the two others categories are *upper case letters* and *non-alphanumeric characters*) i.e., Base36.

Using a same approach as for guessing passwords with brute force/exhaustive search, and assuming one hundred trillion (10^{14}) guesses per second[4]. To brute force the random secret the time required will be $\simeq$ 2.63 thousand trillion centuries:

$$\frac{\sum_{n=1}^{25} 36^n}{1\ year\ in\ seconds} \simeq 2,63 * 10^{17}\ years,\ where\ numerator\ is\ search\ space\ \sim 8,31 * 10^{38}$$

We will argue that this number is too high, as we know a lot about this randomly generated secret. Contrariety a password we know the exact length and search space, so that there is no need for testing strings that are longer or shorter, or containing other characters. When taking this into account, we can reduce the search space size from $\simeq 8,31 * 10^{38}$ to $\simeq 8,08 * 10^{38}$. The time to brute force the secret will still take an inconceivable amount of time.

The cryptographic strength of the join-token, and how the randomly generated secret is created has not been disclosed or documented by Docker to the best of this author's knowledge. We are satisfied with the cryptographic strengt of the join-token.

5.4 Man-in-the-middle attack with existing tools

In this section we outline the attempted attack for capturing the packets in a join-sequence between a worker node and a manger node. We know that when a swarm is initialised, the manager node designates itself as a CA and signs a certificate. The hash of this certificate is used in the join-token, so that the joining worker does not have to download a certificate from a non-swarm participant. If the traffic can be decrypted, and tampered with, we could potentially take control of the swarm, by compromising a manager node. To intercept the traffic we used the tool mitmproxy[5] which is an interactive MITM proxy for HTTP and HTTPS with a console interface. mitmproxy is a suite of tools for MITM, with capabilities such as intercepting HTTP and HTTPS requests and responses and modifying them extemporaneous, as well as and replay attacks. In addition to mitmproxy, we also use `tcpdump` which is a popular tool amongst unix users and Wireshark.

[4]We note that to reach that many guesses/second would require a professional/governmental adversary
[5]`https://mitmproxy.org/`

5.4.1 Results

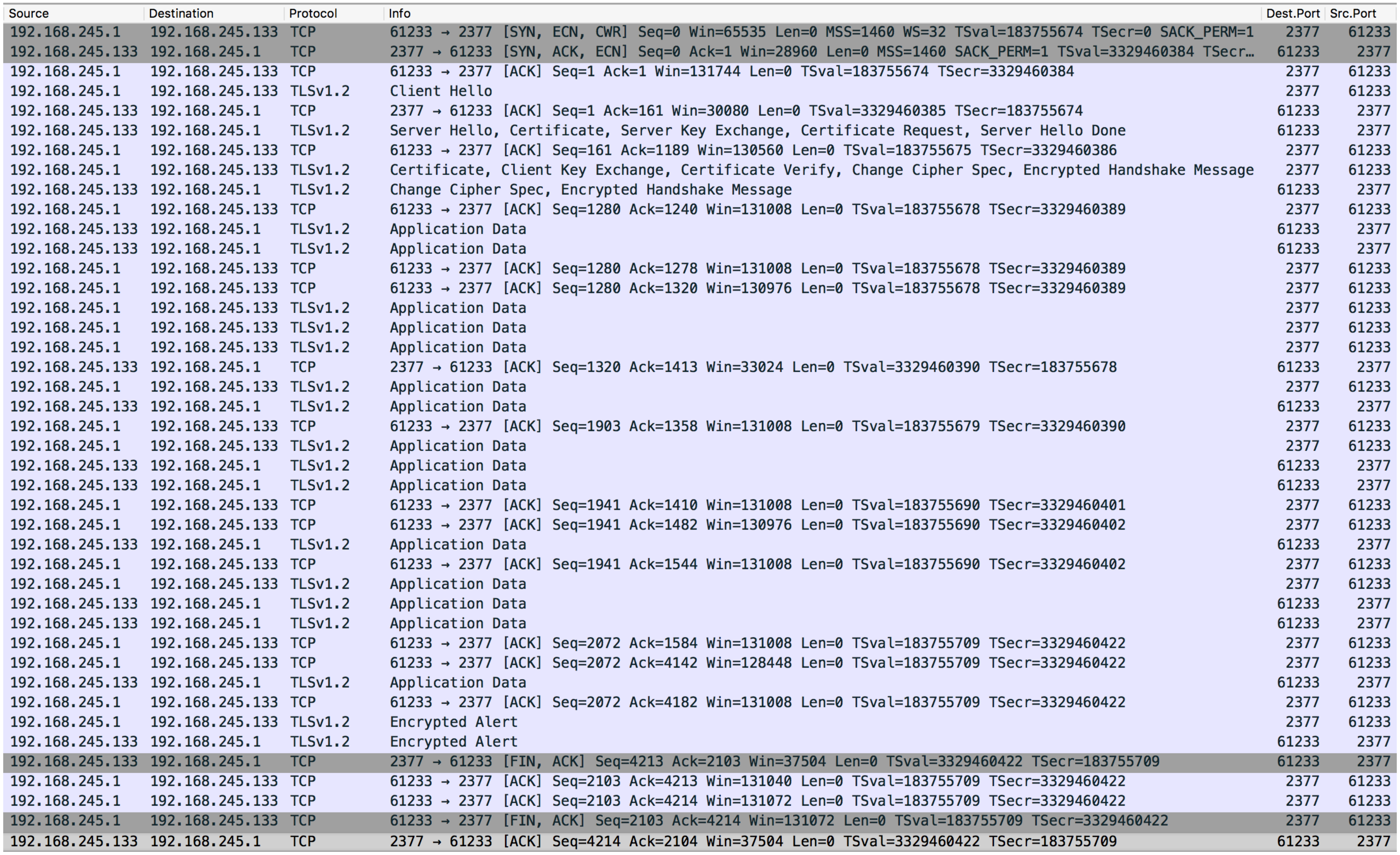
Using mitmproxy for MITM attacking Docker Swarm yielded no results trying to capture a join-sequence. We tested the connection to verify that we can capture other types of traffic between the VMs, but no traffic was recorded using mitmproxy for capturing the join-sequence of Docker Swarm. We tried using tcpdump but with no luck for intercepting traffic on the management plane. We are pretty confident that the reason for this is that mitmproxy and tcpdump currently does not support Protocol buffers and gRPC traffic. This is caused by the binary protocol format used in Protocol buffers and gRPC.

One advantage tpcdump has over the two other tools we tested with, is that for capturing traffic on the data plane, tcpdump has the possibility of listening directly at namespaces, instead of just on the NICs. When debugging distributed applications, this can be especially useful.

In Wireshark we listened to the NIC and saved the traffic to a `.pcap`-file. From there we can confirm that traffic is encrypted. Wireshark does not at this point support Protocol buffers (proto3), but there exist an unofficial dissector for gRPC that uses `proto3`. This dissector invokes the HTTP/2 dissector. This means that the Wireshark source code has to be downloaded and built custom for it to work. We were not able to get this dissector to work, as the traffic needs to be decrypted first.

In Figure 5.2 we see that Wireshark detects that the traffic is TLSv1.2, and when a connection is established between a worker node (`192.168.245.1`), and manager node (`192.168.245.133`). The worker node sends a `client hello`, to initiate the TLS handshake, followed by the expected messages of a handshake. After this the communication becomes encrypted, and we have not found any way to decrypt it in Wireshark. Wireshark does not provide support for elliptic curve cryptography (ECC) certificates for decryption, and any attempt we did at converting the private key to a format that Wireshark does accept, was declined. We have included a view of the entire conversation in Wireshark "Follow TCP Stream" in Appendix B.1.

As these experiments where unsuccessful, the next natural thing for us was to attempt to create a custom proxy. The reason for this was to see if Protocol buffers can be intercepted.



```
Source          Destination     Protocol  Info                                                                                                          Dest.Port Src.Port
192.168.245.1   192.168.245.133 TCP       61233 → 2377 [SYN, ECN, CWR] Seq=0 Win=65535 Len=0 MSS=1460 WS=32 TSval=183755674 TSecr=0 SACK_PERM=1            2377    61233
192.168.245.133 192.168.245.1   TCP       2377 → 61233 [SYN, ACK, ECN] Seq=0 Ack=1 Win=28960 Len=0 MSS=1460 SACK_PERM=1 TSval=3329460384 TSecr…            61233     2377
192.168.245.1   192.168.245.133 TCP       61233 → 2377 [ACK] Seq=1 Ack=1 Win=131744 Len=0 TSval=183755674 TSecr=3329460384                                 2377    61233
192.168.245.1   192.168.245.133 TLSv1.2   Client Hello                                                                                                     2377    61233
192.168.245.133 192.168.245.1   TCP       2377 → 61233 [ACK] Seq=1 Ack=161 Win=30080 Len=0 TSval=3329460385 TSecr=183755674                              61233     2377
192.168.245.133 192.168.245.1   TLSv1.2   Server Hello, Certificate, Server Key Exchange, Certificate Request, Server Hello Done                          61233     2377
192.168.245.1   192.168.245.133 TCP       61233 → 2377 [ACK] Seq=161 Ack=1189 Win=130560 Len=0 TSval=183755675 TSecr=3329460386                            2377    61233
192.168.245.1   192.168.245.133 TLSv1.2   Certificate, Client Key Exchange, Certificate Verify, Change Cipher Spec, Encrypted Handshake Message             2377    61233
192.168.245.133 192.168.245.1   TLSv1.2   Change Cipher Spec, Encrypted Handshake Message                                                                61233     2377
192.168.245.1   192.168.245.133 TCP       61233 → 2377 [ACK] Seq=1280 Ack=1240 Win=131008 Len=0 TSval=183755678 TSecr=3329460389                           2377    61233
192.168.245.133 192.168.245.1   TLSv1.2   Application Data                                                                                               61233     2377
192.168.245.133 192.168.245.1   TLSv1.2   Application Data                                                                                               61233     2377
192.168.245.1   192.168.245.133 TCP       61233 → 2377 [ACK] Seq=1280 Ack=1278 Win=131008 Len=0 TSval=183755678 TSecr=3329460389                           2377    61233
192.168.245.1   192.168.245.133 TCP       61233 → 2377 [ACK] Seq=1280 Ack=1320 Win=130976 Len=0 TSval=183755678 TSecr=3329460389                           2377    61233
192.168.245.1   192.168.245.133 TLSv1.2   Application Data                                                                                                 2377    61233
192.168.245.1   192.168.245.133 TLSv1.2   Application Data                                                                                                 2377    61233
192.168.245.1   192.168.245.133 TLSv1.2   Application Data                                                                                                 2377    61233
192.168.245.133 192.168.245.1   TCP       2377 → 61233 [ACK] Seq=1320 Ack=1413 Win=33024 Len=0 TSval=3329460390 TSecr=183755678                          61233     2377
192.168.245.1   192.168.245.133 TLSv1.2   Application Data                                                                                                 2377    61233
192.168.245.133 192.168.245.1   TLSv1.2   Application Data                                                                                               61233     2377
192.168.245.1   192.168.245.133 TCP       61233 → 2377 [ACK] Seq=1903 Ack=1358 Win=131008 Len=0 TSval=183755679 TSecr=3329460390                           2377    61233
192.168.245.1   192.168.245.133 TLSv1.2   Application Data                                                                                                 2377    61233
192.168.245.133 192.168.245.1   TLSv1.2   Application Data                                                                                               61233     2377
192.168.245.133 192.168.245.1   TLSv1.2   Application Data                                                                                               61233     2377
192.168.245.1   192.168.245.133 TCP       61233 → 2377 [ACK] Seq=1941 Ack=1410 Win=131008 Len=0 TSval=183755690 TSecr=3329460401                           2377    61233
192.168.245.1   192.168.245.133 TCP       61233 → 2377 [ACK] Seq=1941 Ack=1482 Win=130976 Len=0 TSval=183755690 TSecr=3329460402                           2377    61233
192.168.245.133 192.168.245.1   TLSv1.2   Application Data                                                                                               61233     2377
192.168.245.1   192.168.245.133 TCP       61233 → 2377 [ACK] Seq=1941 Ack=1544 Win=131008 Len=0 TSval=183755690 TSecr=3329460402                           2377    61233
192.168.245.1   192.168.245.133 TLSv1.2   Application Data                                                                                                 2377    61233
192.168.245.133 192.168.245.1   TLSv1.2   Application Data                                                                                               61233     2377
192.168.245.133 192.168.245.1   TLSv1.2   Application Data                                                                                               61233     2377
192.168.245.1   192.168.245.133 TCP       61233 → 2377 [ACK] Seq=2072 Ack=1584 Win=131008 Len=0 TSval=183755709 TSecr=3329460422                           2377    61233
192.168.245.1   192.168.245.133 TCP       61233 → 2377 [ACK] Seq=2072 Ack=4142 Win=128448 Len=0 TSval=183755709 TSecr=3329460422                           2377    61233
192.168.245.133 192.168.245.1   TLSv1.2   Application Data                                                                                               61233     2377
192.168.245.1   192.168.245.133 TCP       61233 → 2377 [ACK] Seq=2072 Ack=4182 Win=131008 Len=0 TSval=183755709 TSecr=3329460422                           2377    61233
192.168.245.1   192.168.245.133 TLSv1.2   Encrypted Alert                                                                                                  2377    61233
192.168.245.133 192.168.245.1   TLSv1.2   Encrypted Alert                                                                                                61233     2377
192.168.245.133 192.168.245.1   TCP       2377 → 61233 [FIN, ACK] Seq=4213 Ack=2103 Win=37504 Len=0 TSval=3329460422 TSecr=183755709                     61233     2377
192.168.245.1   192.168.245.133 TCP       61233 → 2377 [ACK] Seq=2103 Ack=4213 Win=131040 Len=0 TSval=183755709 TSecr=3329460422                           2377    61233
192.168.245.1   192.168.245.133 TCP       61233 → 2377 [ACK] Seq=2103 Ack=4214 Win=131072 Len=0 TSval=183755709 TSecr=3329460422                           2377    61233
192.168.245.1   192.168.245.133 TCP       61233 → 2377 [FIN, ACK] Seq=2103 Ack=4214 Win=131072 Len=0 TSval=183755709 TSecr=3329460422                      2377    61233
192.168.245.133 192.168.245.1   TCP       2377 → 61233 [ACK] Seq=4214 Ack=2104 Win=37504 Len=0 TSval=3329460422 TSecr=183755709                          61233     2377
```

Figure 5.2: Screenshot of Wireshark during bootstrap of a new node

5.5 Custom gRPC proxy

Since no existing tools were able to capture the traffic of a join-sequence in Docker Swarm, we build a custom application proxy for decoding Protocol buffers. The implementation is written in Golang, and can be found at GitHub[6]. This implementation is not directly related to Docker Swarm, but can provide valuable insights into the underlying technology that is used for creating Docker Swarm, and perhaps also help us answer the research goal of what other applications can benefit from applying the Docker Swarm-model.

As described in Section 3.6, the proxy-server must know of the `.proto`-files to deserialise the binary messages sent between the client and server, and implement methods for handling the services. To gain better understanding of Protocol buffers and gRPC, and to test a potential future of a gRPC proxy tool, we started by creating a small client/server application. The application is available at Github[7]. The client application takes two integers as input and sends them via gRPC to the server application. The server calculates the greatest common divisor (GCD) of the numbers, and returns the answer to the client.

Figure 5.3 is a SSD of how the application works. The client sends the `GCDService` an RPC-message with the parameters: `ctx` and
`&pb.GCDRequest{A:a,B:b}`. The `GCDService` computes the result, and returns the RPC message: `&pb.GCDResponse{Result:a}`. The different parameters are listed in Table 5.1.

Table 5.1: Parameters of Figure 5.3

Element	**Type**
`ctx`	`context.WithTimeout(ctx, 5*time.Second)`
`A`	`uint64`
`B`	`uint64`
`Result`	`uint64`

The next step from here is to create a proxy-server to be placed in-between these two applications, and forward the request from the client and response from the server, but most importantly, it will log requests and responses. See Figure 5.4 for SSD of how the application will work.

[6] `https://github.com/Diddern/gIntercept`
[7] `https://github.com/Diddern/gRPC-simpleGCDService`

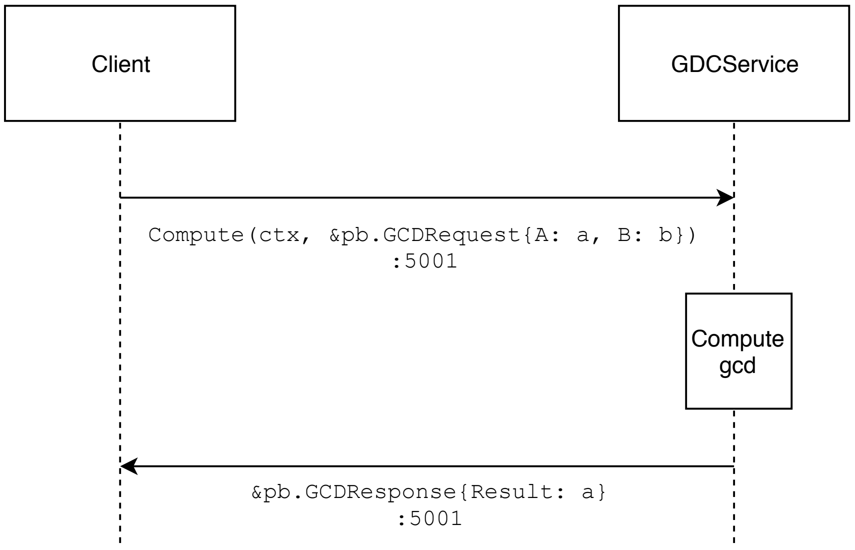

Client
GDCService
Compute(ctx, &pb.GCDRequest{A: a, B: b})
:5001
Compute gcd
&pb.GCDResponse{Result: a}
:5001

Figure 5.3: System sequence diagram of GCD client/server application

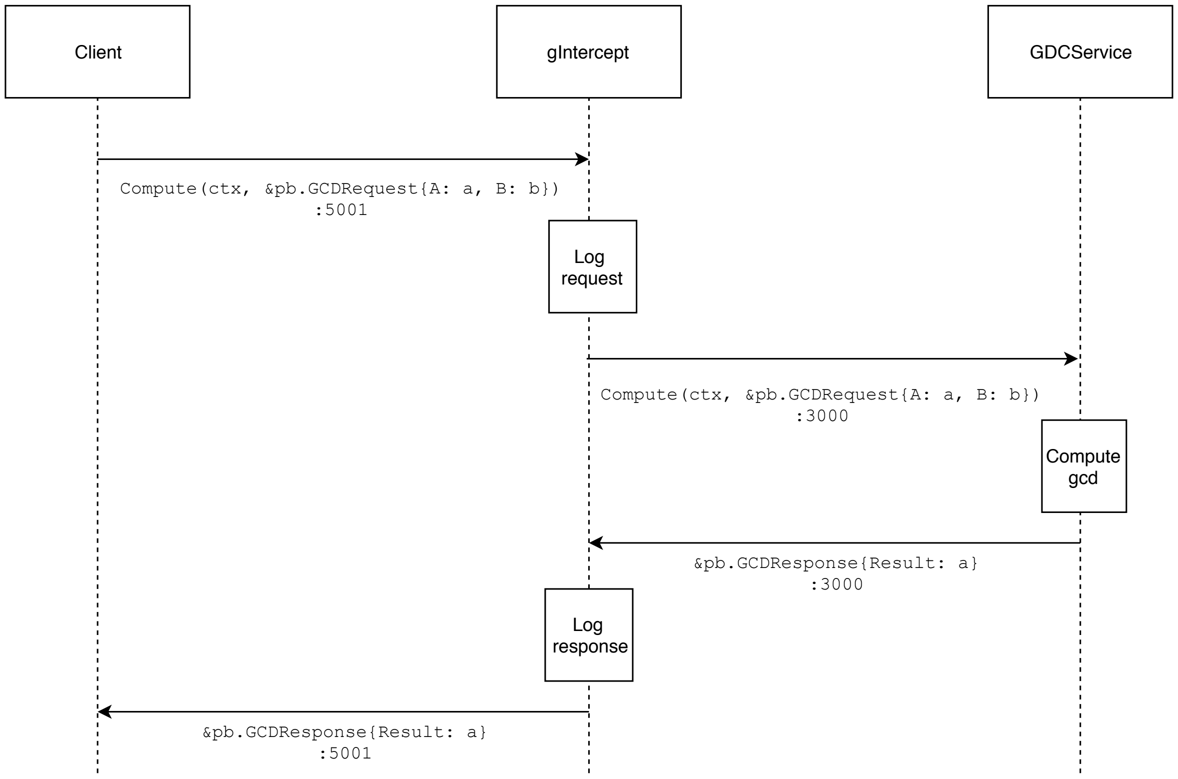

Client
gIntercept
GDCService
Compute(ctx, &pb.GCDRequest{A: a, B: b})
:5001
Log request
Compute(ctx, &pb.GCDRequest{A: a, B: b})
:3000
Compute gcd
&pb.GCDResponse{Result: a}
:3000
Log response
&pb.GCDResponse{Result: a}
:5001

Figure 5.4: System sequence diagram of GCD client/server application with a proxy service

gRPC has options for encrypted and unencrypted communications. Setting up encrypted transport layer communication is a part of the standard features of the gRPC-library. In Listings 5.1 and 5.2 we can see the client and server implementation without encryption enabled, and in listing 5.3 and 5.4 we can see the same client and server with encryption enabled.

Listing 5.1: Client without encryption in gRPC

```go
conn, err := grpc.Dial("localhost:5001", grpc.WithInsecure())
//error handling omitted from listing
client := pb.NewGreeterClient(conn)
//...
```

Listing 5.2: Server without encryption in gRPC

```go
lis, err := net.Listen("tcp", ":5001")
//error handling omitted from listing
server := grpc.NewServer()
```

Listing 5.3: Client with encryption in gRPC

```go
creds, err := credentials.NewClientTLSFromFile("certs/server-cert.pem",
    ↪ "")
//error handling omitted from listing
conn, err := grpc.Dial("localhost:5001",
    ↪ grpc.WithTransportCredentials(creds))
//error handling omitted from listing
client := pb.NewGreeterClient(conn)
//...
```

Listing 5.4: Server with encryption in gRPC

```go
creds, err := credentials.NewServerTLSFromFile("certs/server-cert.pem",
    ↪ "certs/server-key.pem")
//error handling omitted from listing
lis, err := net.Listen("tcp", 5001)
//error handling omitted from listing
server := grpc.NewServer(grpc.Creds(creds))
```

It should be mentioned that our implementation does not use a CA to generate unique certificates for the clients. We generate a certificate with the method in Listing 5.5 with the command line tool `openssl`. If we were to go further with this implementation a dedicated CA would be needed for generating unique client-certificates.

Listing 5.5: Generating certificate for secure communication in gRPC

```
$ openssl req -x509 -newkey rsa:4096 -keyout certs/server-key.pem -out
    ↪ certs/server-cert.pem -days 365 -nodes -subj '/CN=localhost'
```

5.5.1 Results

The use of transport layer encryption in gRPC-applications has been made easy by the developers, and as we can see in the Listings 5.1, 5.2, 5.3 and 5.4 there are only a few lines of code in addition for securing the application. The hard part is securing certificates from unauthorised access, securing the certificate generator from unauthorised use, and rotating certificates in a secure manner. We did not look into creating a secure way of issuing, storing or rotating certificates in this experiment.

We did not go further in creating a proxy tool for inspecting the traffic of Docker Swarm, as that would require us to implement the 32 Protocol buffer files for making a swarm, and this would in principle be considered to reinvent the wheel as it would do the same thing as Docker Swarm does today. The way Docker Swarm handles certificates would also make it difficult for us to create a proxy tool. Doing MITM on MTLS would mean that we would need the certificate of both the worker node, and manager node to inspect traffic in each direction.

5.5.2 Discussion of results

We learned a lot from implementing an application for inspecting protocol buffers over gRPC. Such as that all methods from the `.proto`-files must be implemented for the proxy tool to work. This is for all service calls and messages. That means that for the strongly typed languages that Protocol buffers generate client code for, i.e., C++, Java and Golang, the type must match as well. This provides security by obscurity. Another thing we learned was that the prossess of defining service calls before writing the implementation is a positive for the development flow. It encourages the developer to think about the messages and services before writing the implementation. The fact that gRPC is asynchronous by nature was also a valuable lesson. This is an elegant solution, in contrast to RESTs synchronous nature.

This security of all `proto`-files that must be implemented reminds us about the story of Dan Kaminsky's DNS attack in the summer of 2008 from the book *Secure and Resilient Software Development*. The attack could allow an attacker to redirect clients to alternate servers, leading to misuse. Eight years prior to this, the cryptographer Daniel J. Bernstein had looked at DNS security and decided that Source Port Randomisation was a smart design choice for DNS. When DNS was patched for mitigating Kaminsky's attack, the solution was Source Port Randomisation. Bernstein had not known about the attack but had understood and envisioned a general class of attacks and realised that this enhancement could provide protection. The DNS application he wrote in the year 2000, *djbdns*, did not need patching as it was not only immune to known attacks, but also secure to an unknown attack [26, Chapter 6.1]. This shows that the use of a language and framework that is well implemented can reduce the attack surface of an application.

In an application where the Protocol buffer files are unavailable to the adversary, e.g., in a closed source project, reverse engineering of the protocol buffers is not easy. As we mentioned it is a binary protocol, and without the descriptor it is mostly up to trial and error (or a clever reverse engineer). A related project we found on GitHub[8] called POGOProtos sparked interest, as there was done a lot of work in reverse engineering protocol buffers for cheating in the popular mobile game Pokémon GO. The community[9,10] is very much active in abusing the protocol buffers for capturing rare Pokémon, or gaining power ups by automating tedious tasks, but the development of POGOPrototos has become stale. This is because Niantic, the creators of Pokémon GO has started enforcing security measures like certificate pinning, banning of VPN's, and IP's hosted at data centres for combatting cheaters. The way Docker Swarm use MTLS to authenticate worker and manager nodes would make POGOPrototos ineffective.

We now know that a custom proxy for Docker Swarm is impractical. The next natural step is to inspect the source code of Docker Swarm, as this will provide us with answers to what messages that are sent on the wire. Inspecting the source code, even though a considerable work load will also remove some of the hinders we ran into in the other experiments, such as not being able to decrypt the traffic from Wireshark.

5.6 Dissecting the protocol for joining a swarm

To verify that the security functionality of Docker Swarm, we investigate the creation of a swarm, and the join-sequence of when a a new node joins the swarm by examining the source

[8] `https://github.com/Furtif/POGOProtos`
[9] `https://www.reddit.com/r/pokemongodev/`
[10] `https://talk.pogodev.org/`

code and recompiling it with custom parameters. This is to confirm that the documentation is correct, and to gain confidence in the security requirements from Section 1.3, where we want to investigate that nodes in a swarm are given a strong cryptographic identity, and that access to the swarm is limited to authorised nodes.

From the documentation of SwarmKit [10] we know that certificates are encrypted before stored to disk, the keys are stored in a Secret which is saved to the Raft logs, and rotated between the managers. To obtain a certificate without hacking a container we need to inspect the code-base of SwarmKit. The code-base is currently around a half million lines of code. This made it a very challenging task to find the code that generates the certificate, encrypts it, and handles CSRs.

The project has been suffering from tests failing during build process[11] on MacOS from December 2017. This made the setup, first compilation, and building of SwarmKit a tedious process. When these obstacles were overcome, the build process compiles multiple binaries. For the scope of this thesis we used *swarmd* and *swarmctl* as they are the most important ones.

- *swarmctl*
  This is the client part of SwarmKit, and is the tool for operating SwarmKit clusters, as it is capable of inspecting the cluster, the list of joined nodes and list of services and tasks.
- *swarmd*
  This is the core binary to start a SwarmKit service. It acts as the daemon to create the nodes, and job queue manager for swarm workers.

To introduce the concepts behind the creation of a swarm, and present the modified manager node is somewhat extended, but the meaning is to familiarise the reader with important concepts before presenting the results.

5.6.1 Creating a swarm

In the listings 5.6, 5.7 and 5.8 we show how to create a cluster by starting a manager node, and connect a worker node to that cluster. To explain what is happening, we go through the listings and explain the different options added to the commands run.

[11] https://github.com/docker/swarmkit/issues/2479

In Figure 5.6 we use the flag `-d` to state the directory we want the files to be stored in. And the `--listen-control-api` defines that we want the control API to listen on the socket located in `"/tmp/node-1/swarm.sock"`. We also define the hostname to be `node-1` and the log-level to be set to debug for the most verbose output.

As explained in Section 4.2 and illustrated in Figure 4.1 the manager becomes a CA and has signed a root certificate.

In Listing 5.7 line 1 we export the sockets location to an environment variable, so that the client (*swarmctl*) can access the cluster in line 2. From line 3-13 we see the information about the cluster, and in line 12 we can see the generated worker-token.

In Listing 5.8 we create a node in the same way as we did with the manager node, but with some modifications. We provide it with a `--join-addr` for the manager node, and the `--join-token` described in Figure 4.2 and Section 4.2, and we also define an address for listening for remote API (`--listen-remote-api`). The token provided is the worker-token from Listing 5.7 line 12

Listing 5.6: Starting the manager node from the shell

```
$ swarmd -d /tmp/node-1 --listen-control-api /tmp/node-1/swarm.sock
   ↪ --hostname node-1 --log-level debug
```

Listing 5.7: Inspecting the cluster to see the join-tokens

```
 1 $ export SWARM_SOCKET=/tmp/node-1/swarm.sock
 2 $ swarmctl cluster inspect default
 3     ID           : ly3pczj9ugjamqxokxlxdlkkw
 4     Name         : default
 5     Orchestration settings:
 6         Task history entries: 0
 7     Dispatcher settings:
 8         Dispatcher heartbeat period: 5s
 9     Certificate Authority settings:
10         Certificate Validity Duration: 2160h0m0s
11     Join Tokens:
12         Worker:
             ↪ SWMTKN-1-0fepd7636a2g596d1751y1sl9jqbgc4afmgyj7jzkk8lgqlwgf-
             ↪ 6fum3wg9c15riy9xg43mkkv0z
13         Manager:
             ↪ SWMTKN-1-0fepd7636a2g596d1751y1sl9jqbgc4afmgyj7jzkk8lgqlwgf-
             ↪ 6kst2vj5og9qk0coiq77i7cc9
```

Listing 5.8: Starting the worker node from the shell

```
$ swarmd -d /tmp/node-2 --hostname node-2 --join-addr 127.0.0.1:4242
   ↪ --listen-remote-api 127.0.0.1:4343 --log-level debug --join-token
   ↪ SWMTKN-1-0fepd7636a2g596d1751y1sl9jqbgc4afmgyj7jzkk8lgqlwgf-
   ↪ 6fum3wg9c15riy9xg43mkkv0z
```

We now have a working manager node, and a worker node. These will communicate via MTLS and the certificate will rotate every 2160 hour (90 days). This confirms the documentation from Section 4.2. We then add more worker nodes by applying the same strategy as in Listing 5.8 to prepare a service roll out. Table 5.2 lists the output of the command `swarmctl node ls` that returns a list of active nodes in the swarm. A service is then created with the command in Listing 5.9, where we define that the service is to be named "redis" for convenience, and the Docker image it should host is the latest version of Redis from Docker Hub, and it should be made six replicas of it.

By inspecting the service in Table 5.3 we see that there are six running instances of the image `redis:latest` running distributes amongst the four nodes. The service has a unique ID (`ss9eiks606mj6reqxjrq2rkb5`), and each replica of the instance is assigned a unique task-ID. If we look at running Docker containers by contacting the Docker CLI in Table 5.4, we can see that there are six containers running, and that the containers also have a unique ID, but the name of a container correspond to `<service>.<slot>.<task-id>`.

Table 5.2: Available nodes in swarm

```
$ swarmctl node ls

ID                          Name    Membership  Status  Availability  Manager    Status
lri5mb7wv5rs3hohstvl8gofn   node-2  ACCEPTED    READY   ACTIVE
rkc0er2dspx03dde3a79yl3le   node-1  ACCEPTED    READY   ACTIVE        REACHABLE  *
wfw5fn951p08vmi225vj72w2k   node-3  ACCEPTED    READY   ACTIVE
wglccu7f3jiz06b8ly98ydmut   node-4  ACCEPTED    READY   ACTIVE
```

Listing 5.9: Starting the manager node from the shell

```
$ swarmctl service create --name redis --image redis:latest --replicas 6
ss9eiks606mj6reqxjrq2rkb5
```

Table 5.3: Inspection of service running 6 replicas of Redis

```
$ swarmctl service inspect redis
ID:                              ss9eiks606mj6reqxjrq2rkb5
Name                             redis
Replicas                         6/6
Template
Container
Image                            redis:latest
TaskID                           Service       Slot  Image         Desired State  Last State  Node
80aflj7dkyxqmumfq6f7774ax        redis         1     redis:latest  RUNNING        RUNNING     1
kivm7j95feo6801cmvg0x69ee        redis         2     redis:latest  RUNNING        RUNNING     2
6v3pu4g6if1ya6fwp80of2z85        redis         3     redis:latest  RUNNING        RUNNING     1
ixlqi4yvhk12e5xnamsi24xu6        redis         4     redis:latest  RUNNING        RUNNING     2
18ghiedhux54xsh29pd1zscck        redis         5     redis:latest  RUNNING        RUNNING     3
m75xrz9o8p92gb9htlj1q51nr        redis         6     redis:latest  RUNNING        RUNNING     4
```

Table 5.4: Running Docker containers

```
$ docker ps -a

CONTAINER ID   IMAGE         COMMAND                  CREATED    STATUS     PORTS     NAMES
29e0f51b250b   redis:latest  docker-entrypoint.s...   3 minutes  3 minutes  6379/tcp  redis.6.m75xrz9o8p92gb9htlj1q51nr
6e134b62d0ab   redis:latest  docker-entrypoint.s...   3 minutes  3 minutes  6379/tcp  redis.5.18ghiedhux54xsh29pd1zscck
8c42ed89bcd5   redis:latest  docker-entrypoint.s...   3 minutes  3 minutes  6379/tcp  redis.2.kivm7j95feo6801cmvg0x69ee
89c688181d49   redis:latest  docker-entrypoint.s...   3 minutes  3 minutes  6379/tcp  redis.4.ixlqi4yvhk12e5xnamsi24xu6
785727383b79   redis:latest  docker-entrypoint.s...   3 minutes  3 minutes  6379/tcp  redis.3.6v3pu4g6if1ya6fwp80of2z85
2d94ff97aa65   redis:latest  docker-entrypoint.s...   3 minutes  3 minutes  6379/tcp  redis.1.80aflj7dkyxqmumfq6f7774ax
```

5.6.2 Modified manager node

For confirming what messages are sent on the wire we need to recompile *swarmd* with logging functionality on the method that are being called when the manager is starting, and when a node is joining the swarm. In this section we focus on the manager node, and in Section 5.6.3 we look at how a new node joins the swarm.

Most of the logic for creating a self-signed CA lies in the file `swarmkit/ca/certificates.go`. A pretty large file with 991 lines of code. The methods are fairly high in complexity as well, with some methods exceeding 100 lines of code.

When the manager starts up, the method `CreateRootCA` is called, it takes the parameters `rootCN` and returns a struct of the rootCA. We revisit this struct in Section 5.6.3 (see Table 5.5). The method is shown in Listing 5.10. For viewing the data we attached a logger to the application shown in line 20,21,22. This lets us output to the terminal the private key of the root certificate, which is only stored to disk encrypted under normal circumstances.

Listing 5.10: Creating the root certificate

```go
func CreateRootCA(rootCN string) (RootCA, error) {
    //Create a simple CSR for the CA using the default CA validator and
      ↪ policy
    req := cfcsr.CertificateRequest{
        CN:          rootCN,
        KeyRequest: &cfcsr.BasicKeyRequest{A: RootKeyAlgo, S:
          ↪ RootKeySize},
        CA:          &cfcsr.CAConfig{Expiry: RootCAExpiration},
    }
    //Generate the CA and get the certificate and private key
    cert, _, key, err := initca.New(&req)
    //error handling omitted from listing

    //Convert key to PKCS#8 in FIPS mode
    if fips.Enabled() {
        key, err = pkcs8.ConvertECPrivateKeyPEM(key)
        //error handling omitted from listing
    }
    rootCA, err := NewRootCA(cert, cert, key, DefaultNodeCertExpiration,
      ↪ nil)
    //error handling omitted from listing

    log.G(context.TODO()).Debugf("The certificate is:\n %s",cert)
    log.G(context.TODO()).Debugf("The Elliptic Curve key is:\n %s",key)
    log.G(context.TODO()).Debugf("Resulting in the RootCA:\n %s",rootCA)

    return rootCA, nil
}
```

Listing 5.11: Downloading the root certificate from manager node

```go
func GetRemoteCA(ctx context.Context, d digest.Digest, connBroker
  ↪ *connectionbroker.Broker) (RootCA, error) {
    insecureCreds := credentials.NewTLS(&tls.Config{InsecureSkipVerify:
      ↪ true})
    conn, err := getGRPCConnection(insecureCreds, connBroker, false)
    //error handling omitted from listing
    client := api.NewCAClient(conn.ClientConn)
    ctx, cancel := context.WithTimeout(ctx, 5*time.Second)
    //closing connection omitted from listing
    response, err := client.GetRootCACertificate(ctx,
      ↪ &api.GetRootCACertificateRequest{})
    //error handling omitted from listing
    if d != "" {
        verifier := d.Verifier()
        if err != nil {
            return RootCA{}, errors.Wrap(err, "unexpected error getting
              ↪ digest verifier")
        }
        io.Copy(verifier, bytes.NewReader(response.Certificate))
        if !verifier.Verified() {
            return RootCA{}, errors.Errorf(" remote CA does not match
              ↪ fingerprint. Expected: %s", d.Hex())
        }
    }
    return NewRootCA(response.Certificate, nil, nil,
      ↪ DefaultNodeCertExpiration, nil)
}
```

The logger in line 20,21,22 of Listing 5.10 is the standard logger used for the rest of Docker Swarm, but is non-existent in the file `swarmkit/ca/certificates.go`, as the logger is usually attached to higher order functions. We have implemented it with a dummy-context (`context.TODO()`) which is not important to us at the current moment.

5.6.3 Results

After not being able to decrypt or decode the protocol from the MITM experiments with mitmproxy, tcpdump, Wireshark and custom gRPC proxy, we wanted to confirm what messages that are sent on the wire. The recompilation described in Section 5.6 let us view some logs of what happens when a root certificate is generated. The results of logging parameters from Listing 5.10 are shown in Listing 5.12[12]. The RootCA is a struct (see Glossary for definition), and in the Listing 5.5 we show what the declared fields of the struct are. The explanation is from comments in the code, but we have verified that they are correct.

In Listing 5.12 we were curious to what the `(EXTRA string=)` value in Line 29 is, and after looking at the struct for a `RootCA` we have concluded that it is the list of intermediates as described in Table 5.5.

Joining an existing swarm

A new node joining an existing swarm is sometimes referred to as *secure node introduction*, or *bootstrapping*. We made a communications diagram for this procedure. This is illustrated in Figure 5.5, and Table 2 describes the different parameters of the communication. This is based on analysis of the code from Listings C.1, C.2, 5.11 and 5.12. We see in the communications diagram (Figure 5.5) that the node wanting to join the swarm, first downloads the certificate bundle from the manager node, and verifies the hash of the certificate with the hash of the join-token. When this is done, the worker node creates a CSR and sends the request for getting a signed certificate. The manager node verifies the join-token and signs the certificate. The worker token then sends a request for the singed certificate.

The first method is for downloading the root certificate, as shown in Listing 5.11. We want to point out a clever function in Line 11. The digest covers the entire bundle of

[12]Ellipsis (. . .) indicate truncation such as full certificates.

certificates, and not only only a single certificate. This prevents MITM attacks where a MITMed CA can provide a single certificate which matches the digest, and then provide arbitrary other non-verified root certificates that the manager certificate actually chains up to.

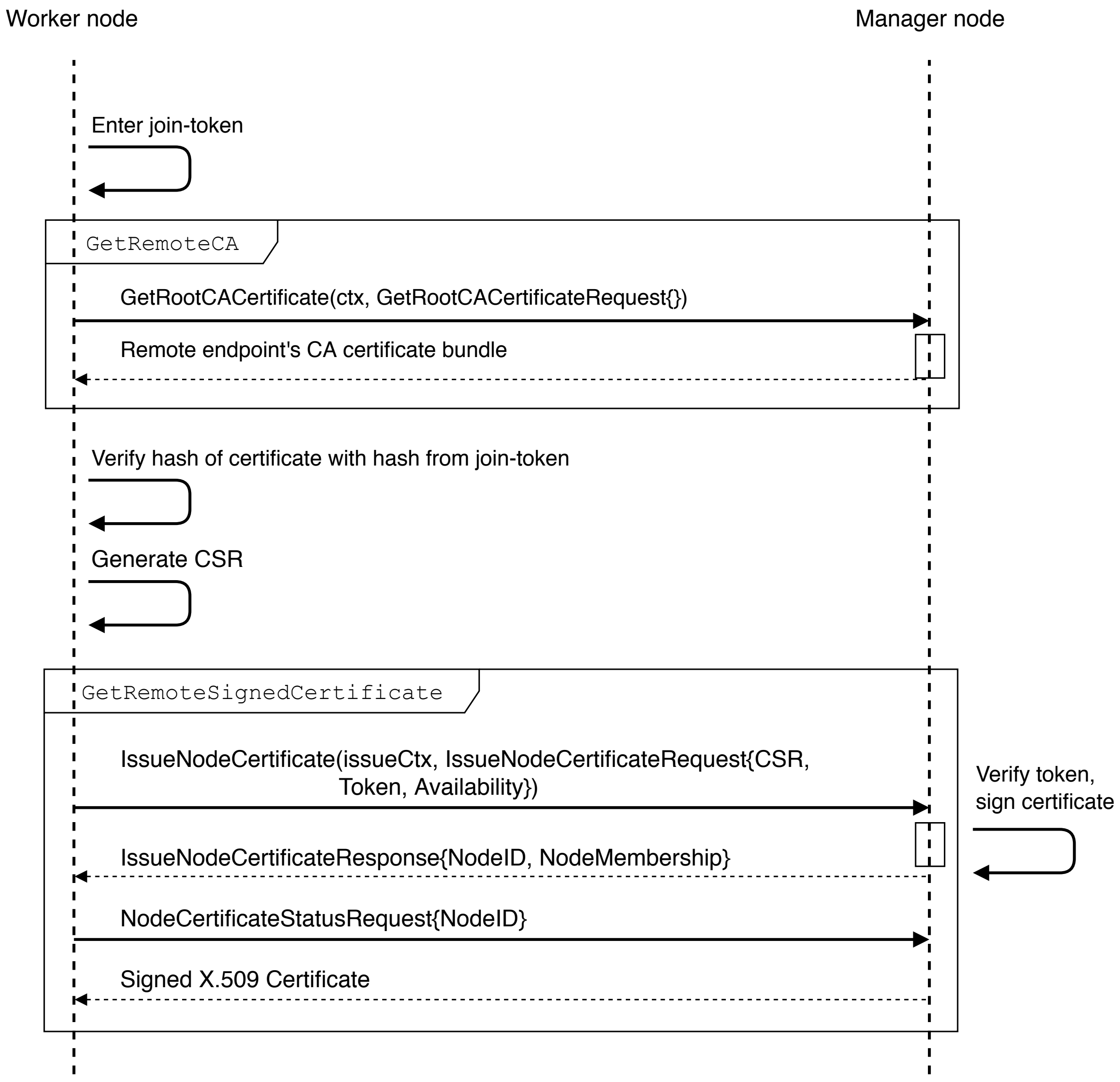

Worker node
Manager node
Enter join-token
GetRemoteCA
GetRootCACertificate(ctx, GetRootCACertificateRequest{})
Remote endpoint's CA certificate bundle
Verify hash of certificate with hash from join-token
Generate CSR
GetRemoteSignedCertificate
IssueNodeCertificate(issueCtx, IssueNodeCertificateRequest{CSR, Token, Availability})
Verify token, sign certificate
IssueNodeCertificateResponse{NodeID, NodeMembership}
NodeCertificateStatusRequest{NodeID}
Signed X.509 Certificate

Figure 5.5: Communications diagram for a worker node joining a swarm

When the certificate is downloaded by the worker node, the worker compares the hash of the certificate with the hash of the join-token, and if they match, the worker sends a CSR to the manager. This is done by the method `GetRemoteSignedCertificate`. We have put this method in the Appendix C.1 due to the size and complexity that surpasses 100 lines of code. This method has the only non-MTLS request that we could find in the source code. It is used for the bootstrapping of the TLS certificates, and by definition cannot use MTLS. After the worker has acquired its signed certificate, all communication is secured by MTLS. This adheres to our requirement for ”management and control traffic sent on the network should be encrypted when possible”.

Table 5.5: RootCA Struct

Element	**Description**
`Certs []byte`	Certs contains a bundle of self-signed, PEM encoded certificates for the Root CA to be used as the root of trust.
`Intermediates []byte`	Intermediates contains a bundle of PEM encoded intermediate CA certificates to append to any issued TLS (leaf) certificates. The first one must have the same public key and subject as the signing root certificate, and the rest must form a chain, each one certifying the one above it, as per RFC 5246 Section 7.4.2.
`Pool *x509.CertPool`	Pool is the root pool used to validate TLS certificates
`Digest digest.Digest`	Digest of the serialised bytes of the certificate(s)
`signer *LocalSigner`	This signer will be nil if the node does not have the appropriate key material

Listing 5.12: Verbose output of creating the root certificate

```
$ swarmd -d /tmp/node-1 --listen-control-api /tmp/node-1/swarm.sock
    ↪ --hostname node-1 --log-level debug

[INFO] generate received request
[INFO] received CSR
generating key: ecdsa-256
[INFO] encoded CSR
[INFO] signed certificate with serial number 4625933...4240060632
DEBU[0000] The certificate is:
 -----BEGIN CERTIFICATE-----
MIIBazCCARCgAwIBAgIUUQdnHaqDoNSEfw9kAwrldpiFuNgwCgYIKoZIzj0EAwIw
...
WBP93a6CVDNtic8vr7TXqCqW7hYdiSfmZjUatcA==
-----END CERTIFICATE-----

DEBU[0000] The Elliptic Curve key is:
-----BEGIN EC PRIVATE KEY-----
MHcCAQEEIMTopYpPy+HQAakz/L/B9If64DRF4jB4e4h9FvNEGGayoAoGCCqGSM49
AwEHoUQDQgAEp7bAqmHJWk+xuwkV27iZzlZC5N8Ljo8nstbUX06DYW1cy0o4o7e1
rZk0N0TBDo43mCnqSlpsi+vyM6zuNJNmNg==
-----END EC PRIVATE KEY-----
```

```

DEBU[0000] Resulting in the RootCA:
-----BEGIN CERTIFICATE-----
MIIBazCCARCgAwIBAgIUUQdnHaqDoNSEfw9kAwrldpiFuNgwCgYIKoZIzj0EAwIw
...
WBP93a6CVDNtic8vr7TXqCqW7hYdiSfmZjUatcA==
-----END CERTIFICATE-----
  %!s(*x509.CertPool=&{map[...:[0]] map[010swarm-ca:[0]] [0xc42016f900]})
      ↪ sha256:12bf30634b...b61a79a439b
  %!s(*ca.LocalSigner=&{0xc42013e4c0 [45 66 69 ... 10] 0xc420352000
      ↪ 0xc420278540})}%!(EXTRA string=)
...
```

Table 5.6: Parameters of Figure 5.5

Element	Description
`ctx`	`context.WithTimeout(ctx, 5*time.Second)`
`GetRootCACertificateRequest`	Empty struct
`issueCtx`	`context.WithTimeout(ctx, 5*time.Second)`
`IssueNodeCertificateRequest`	Struct. See Listing 5.13 for values
`CSR`	[]byte
`Token`	String
`Availability`	NodeSpec_Availability: int32
`NodeID`	String
`NodeMembership`	”PENDING” or ”ACCEPTED”

Listing 5.13: Struct for IssueNodeCertificateRequest

```
struct {
    Role NodeRole `protobuf:"varint,1,opt,name=role,proto3,enum=
        ↪ docker.swarmkit.v1.NodeRole" json:"role,omitempty"`
    CSR []byte `protobuf:"bytes,2,opt,name=csr,proto3"
        ↪ json:"csr,omitempty"`
    Token string `protobuf:"bytes,3,opt,name=token,proto3"
        ↪ json:"token,omitempty"`
    Availability NodeSpec_Availability
        ↪ `protobuf:"varint,4,opt,name=availability,proto3,enum=
        ↪ docker.swarmkit.v1.NodeSpec_Availability"
        ↪ json:"availability,omitempty"`ailability
        ↪ `protobuf:`varint,4,opt,name=availability,proto3,enum=
        ↪ docker.swarmkit.v1.NodeSpec_Availability`
        ↪ json:`availability,omitempty``
```

5.7 Input fuzzing on the manager node

A critical test to perform is to verify what inputs can be sent to a manger node. If the input is processed without authentication, the input can be crafted to consume resources on the manager node. When a new node is joining the swarm, the manager takes data from a potentially unauthenticated service, as seen in the Listing C.1 in the Appendix. We know that the join-token must be parsed by the manager, so we are curious about the input validation that is performed by a manager node.

To test if a bad input can cause problems we use strings from the GitHub-repository *big-list-of-naughty-strings*[13], a popular list of about 500 strings that have a high probability of causing issues when used as user-input data. The list includes strings which contain emojis (should produce the same behaviour as two-byte characters), right-to-left strings, zalgo text, script injection, SQL injection, server code injection and more. We expect the error message `"invalid join token"` for us to have confidence in the join-token validator to be safe from malicious strings. We show a snippet of the code for testing the strings in Listing 5.14. Line 7 is an example of the 500 line long list that we omit from the listing because LaTeX or cannot handle the strings.

By filling the CSR byte array with a random string, we want to see the response from the manager node. In Line 24 of Listing C.1 we see that the request consists of a `CSR`, the `join-token`, and the `availability` of the node joining the swarm. If we change the `CSR` to be `"Hello Mars"` instead of the expected `CSR` from CloudFlare's PKI and TLS toolkit called cfssl[14] as shown in Listing 5.15. Table 5.7 lists the predefined errors for a CSR. If the return value matches on of these inputs, that means that our `"Hello Mars"` input was denied successfully.

CloudFlare's PKI and TLS toolkit is a software suite with tools for building custom TLS PKI tools. The implementation is written in Golang and actively maintained by CloudFlare. After inspecting the source code of Docker Swarm we can see that much of the PKI infrastructure is derived from this toolkit.

Listing 5.14: Join-token from Big-list-of-naughty-strings

```
1 for _, invalidToken := range []string{
2   "LuguberToken", //invalid token
3   "SWMTKN-1-0fepd7636a2g596d1751y1sl9jqbgc4afmgyj7jzkk8lgqlwgf-
      ↪ 6fum3wg9c15riy9xg43mk6", //mistyped
4   "SWMTKN-1-0fepd7636a2g596d1751y1sl9jqbgc4afmgyj7jzkk8lgqlwgf-
      ↪ 6fum3wg9c15riy9xg43mkkv0z", //expiredToken
5   "SWMTKN-1-0fepd7636a2g596d1751y1sl9jqbgc8daertj7jzkk8lgqlwgf-
      ↪ 6fum3wg9c15riy9xg43mkkv0z", //invalid hash valid secret
```

[13]https://github.com/minimaxir/big-list-of-naughty-strings

[14]https://github.com/cloudflare/cfssl/blob/master/csr/

```
  "SWMTKN-1-0fepd7636a2g596d1751y1sl9jqbgc4afmgyj7jzkk8lgqlwgf-
     ↪ 6fum3wg9c15riasdf43mkkv0z", //valid hash invalid secret
   "' OR '1'='1",
   ...
} {
  issueRequest := &api.IssueNodeCertificateRequest{CSR: csr, Token:
     ↪ invalidToken, Availability: config.Availability}
  issueResponse, err := caClient.IssueNodeCertificate(issueCtx,
     ↪ issueRequest)
```

Listing 5.15: Unexpected data as CSR

```
var joinToken =
   ↪ SWMTKN-1-0fepd7636a2g596d1751y1sl9jqbgc4afmgyj7jzkk8lgqlwgf-
   ↪ 6fum3wg9c15riy9xg43mkkv0z
issueRequest := &api.IssueNodeCertificateRequest{CSR: []byte("Hello
   ↪ Mars"), Token: joinToken}
```

Table 5.7: Expected errors from bad CSR in cfssl

Case	**msg**
Unknown	"CSR parsing failed due to unknown error"
ReadFailed	"CSR file read failed"
ParseFailed	"CSR Parsing failed"
DecodeFailed	"CSR Decode failed"
BadRequest	"CSR Bad request"
default:	panic(fmt.Sprintf("Unsupported CF-SSL error reason %d under category APIClientError.", reason))

We also want to check if we can craft a malicious CSR using CloudFlare's PKI and TLS toolkit. This is done by altering the method `RequestAndSaveNewCertificates` in the file `swarmkit/ca/certificates.go` from Appendix C.2 Line 3 to the lines in Listing 5.16. As we can see, we are invoking the generation of a custom CSR to see if this can provoke an alternative behaviour.

Listing 5.16: Generating a malicious CSR

```
request := &cfcsr.CertificateRequest{
        Names: []cfcsr.Name{
            {
                O:  "LuguberOrganisation",
                OU: "LuguberOrganisationalUnit ",
                L:  "LuguberLocality",
            },
        },
        CN:         "LuguberCN",
        Hosts:      []string{"luguber.no"},
        KeyRequest: &cfcsr.BasicKeyRequest{A: "ecdsa", S: 256},
    }
csr, _, err := cfcsr.ParseRequest(request)
```

5.7.1 Results

The results of sending malformed join-tokens to the manager resulted in `"invalid join token"` for all strings except empty string that returns `nil`. For the tokens with invalid hash of root CA and expired tokens the error returned is `"remote CA does not match fingerprint"`. We note that the list of tested strings has more effect on dynamic languages and strings that are consumed in client-to-server applications, as many of them are web browser exploits (XSS, SQL injections and script injections). The experiment shows that Docker Swarm by means of Golang is capable of handling characters from multiple character sets without problems. We can also gain confidence in the requirement listed in Section 1.3 about ”Access to a swarm is limited to an authorised node” as we test with many strings that are known to cause problems for other implementations that takes input from an unauthenticated party. As there are no errors in parsing the misformatted strings, we are confident in stating that the manager node is immune to a DoS-attack by misformatted join-tokens.

The results of sending the false CSR to the manager node with a valid token, results in `"CSR Decode failed"` as is consistent with Table 5.7 for an unexpected error when processing the CSR. We note that in the method `IssueNodeCertificate` of `swarmkit/certificate/ca/server.go` the token is verified before processing the CSR. In Listing 5.17 we show a snippet for this. This prevents an unauthenticated user to sending input to the manager node that will be processed. The method used for comparing the token is in our view secure as well, as an insecure comparison can be used for a timing attack. The method used is from the package `"crypto/subtle"` and uses a `ConstantTimeCompare`. From the language documentation:

> ConstantTimeCompare returns 1 if and only if the two slices, x and y, have equal contents. The time taken is a function of the length of the slices and is independent of the contents.

This is wrapped in a mutex lock that enforces limits on access to a resource (tokens) during the comparison, and we see that if none of the cases match, we return an error message about the need for a valid token to join the cluster.

Listing 5.17: Comparing the Join-tokens of the swarm with new nodes worker token

```
  role := api.NodeRole(-1)
  s.mu.Lock()
  if subtle.ConstantTimeCompare([]byte(s.joinTokens.Manager),
     ↪ []byte(request.Token)) == 1 {
    role = api.NodeRoleManager
  } else if subtle.ConstantTimeCompare([]byte(s.joinTokens.Worker),
     ↪ []byte(request.Token)) == 1 {
    role = api.NodeRoleWorker
  }
  s.mu.Unlock()
  if role < 0 {
    return nil, status.Errorf(codes.InvalidArgument, "A valid join token
       ↪ is necessary to join this cluster")
  }
```

We generated a full CSR for trying to provoke alternative behaviour. As we can see in Listing 5.18 Line 5 and 20 the Luguber-parameters have been applied to the "Subject", and the Hostname has been set in the "DNS"-field. Given that the join-token is valid (for a worker node in our example) the resulting certificate is signed. The certificate for the worker is shown in Listing 5.19. In Lines 12 and 36 we can see that the signed certificate has altered the fields to reflect the worker nodes identity (Subject-fields), and the DNS is set to be `swarm-worker` and the `Node ID`. We also see in Line 8 that the issuer of the certificate is the Root CA.

Listing 5.18: Crafting a malicious CSR

```
1  $ openssl req -in MaliciousCSR.csr -text
2  Certificate Request:
3   Data:
4    Version: 0 (0x0)
5    Subject: L=LuguberLocality, O=LuguberOrganization,
        ↪ OU=LuguberOrganizationUnit, CN=LuguberCN
6    Subject Public Key Info:
7     Public Key Algorithm: id-ecPublicKey
8      Public-Key: (256 bit)
9      pub:
10      04:8e:ea:a2:a0:d9:1e:18:3f:d8:cf:28:da:df:0a:
11      73:d1:2c:10:49:91:8f:28:b0:b3:6f:92:38:fc:b6:
12      34:38:11:2b:e8:4a:03:e3:d8:5d:a6:8d:99:b2:7f:
13      7b:c5:8b:7f:05:66:0d:f7:eb:2a:02:bc:f8:5d:5e:
14      bd:ba:02:c8:fc
15     ASN1 OID: prime256v1
16     NIST CURVE: P-256
17   Attributes:
18   Requested Extensions:
19    X509v3 Subject Alternative Name:
20     DNS:luguber.no
21  Signature Algorithm: ecdsa-with-SHA256
22    30:45:02:20:46:ba:45:a8:8c:da:5b:13:ed:55:54:78:78:b9:
23    91:cf:8d:f7:d6:68:d2:5e:d5:4b:b0:be:b5:35:56:51:11:1c:
24    02:21:00:c1:6b:83:de:e6:5d:03:14:41:d6:2e:96:16:fc:3c:
25    26:98:e2:e7:ba:9b:01:95:84:6c:42:2f:25:02:42:82:ca
26 -----BEGIN CERTIFICATE REQUEST-----
27 MIIBUTCB+AIBADBuMRgwFgYDVQQHEw9MdWd1YmVyTG9jYWxpdHkxHDAaBgNVBAoT
```

```
..
JQJCgso=
-----END CERTIFICATE REQUEST-----
```

Listing 5.19: Resulting certificate of a malicious CSR

```
$ openssl x509 -in CertFromMaliciousCSR.crt -noout -text
Certificate:
  Data:
    Version: 3 (0x2)
    Serial Number:
      7b:50:bc:8b:71:62:4a:83:29:ef:6e:fa:a4:f5:db:8f:1a:70:8f:5b
  Signature Algorithm: ecdsa-with-SHA256
    Issuer: CN=swarm-ca
    Validity
      Not Before: Jun 14 12:58:00 2018 GMT
      Not After : Sep 12 13:58:00 2018 GMT
    Subject: O=qug50tv9c3r5ivrg1e05p4t4m, OU=swarm-worker,
      ↪ CN=z532qsy9y6potlnuubvktexn3
    Subject Public Key Info:
      Public Key Algorithm: id-ecPublicKey
        Public-Key: (256 bit)
        pub:
          04:8e:ea:a2:a0:d9:1e:18:3f:d8:cf:28:da:df:0a:
          73:d1:2c:10:49:91:8f:28:b0:b3:6f:92:38:fc:b6:
          34:38:11:2b:e8:4a:03:e3:d8:5d:a6:8d:99:b2:7f:
          7b:c5:8b:7f:05:66:0d:f7:eb:2a:02:bc:f8:5d:5e:
          bd:ba:02:c8:fc
        ASN1 OID: prime256v1
        NIST CURVE: P-256
    X509v3 extensions:
      X509v3 Key Usage: critical
        Digital Signature, Key Encipherment
      X509v3 Extended Key Usage:
        TLS Web Server Authentication, TLS Web Client Authentication
      X509v3 Basic Constraints: critical
        CA:FALSE
      X509v3 Subject Key Identifier:
        83:FA:7F:36:DD:88:67:F0:5C:B4:2C:B7:72:56:76:43:5C:E4:E8:DF
      X509v3 Authority Key Identifier:
        keyid:7B:E0:4C:3E:43:45:8E:1B:B5:E9:8D:6B:3E:9A:66:12:8D:15:58:A7
      X509v3 Subject Alternative Name:
        DNS:swarm-worker, DNS:z532qsy9y6potlnuubvktexn3
  Signature Algorithm: ecdsa-with-SHA256
     30:46:02:21:00:a1:e6:2d:0d:41:df:9a:ca:1e:46:15:d0:35:
     ..
     10:f4
```

As we can see in Listing 5.19, the CSR ignores the malicious parameters of our CSR and signs a valid certificate for the worker-node, with the only authentication mechanism being the join-token. This is interesting, as we expected the certificate to be rejected. It contains unexpected input, and should in our opinion be be rejected. From a best security practice perspective we suggest that any misformatted request should be rejected.

Chapter 6

Discussion and conclusion

In this chapter we discuss the findings from Chapter 5, and give our opinion on the technology used for creating Docker Swarm and the general security of an orchestration service. We conclude by summarising how Docker Swarm preforms in regard to the research questions, and requirements listed in Section 1.3. We then end by presenting a list of future work in regard to security of orchestration services, Golang and Protocol buffers.

6.1 Discussion

The reason why the MITM attacks with mitmproxy, tcpdump and Wireshark where not successful is that one of the threat models that Docker Swarm protects itself from is both passive and active attacks, where an adversary has control of the underlying network, and can capture network traffic. As long as the certificate is stored securely, it will not be able to eavesdrop on the traffic on the *control* or *management* plane. We are happy with the learning outcome from trying to write a proxy tool for gRPC/Protocol buffers, as it shows that the security of Docker Swarm is well constructed, and the problems we encountered are the parts of overlapping security well embedded in the design.

The idea of Protocol buffers and gRPC is not new. After maturing as internal Google projects from the early 2000s, they were open sourced in 2015, and has been the building blocks of highly successful companies like Square, Netflix, Cisco and Juniper networks.

Bekk Consulting, a reputable technology consulting firm in Norway, publishes every spring a "technology radar" to assess how they see the current landscape in technology by determining potential, maturity and available competence. In the published version from the summer of 2018, Protocol buffers are rated as *"employ"*, and is given high remarks for being language and platform agnostic, fast, and light on the wire[1]. We agree with this and would like to see Protocol buffers becoming the successor to JSON when developing web services.

Many of the ideas and people behind Docker Swarm comes from Square, and Google as they have been using Protocol buffers and gRPC (Stubby) extensively from the beginning of the technology. Docker Swarm's Security lead, Diogo Mónica has a history at Square, as well as Dino Dai Zovi. John Hanke, the CEO of Niantic, Inc. and creator of Pokémon Go has an extensive background at Google where we assume that he understood the potential of gRPC and Protocol buffers, this might be one of the reasons why Pokémon Go is the most popular game ever, with 750 million global downloads in the first year [37]. We believe that the innovations done by the people at Square and Google will make the internet a better place in the future.

Golang is from our point of view a robust and fast programming language with great security features. Its immaturity has good and bad sides; As a positive, since it is fairly new, it does not support older protocols. This makes Golang immune to downgrade attacks previous to TLS version 1.2. A negative is that the community and adoption rate is currently not big enough to provide us with good documentation and guides. Flaws or zero days found in other implementations of security features can potentially stay undetected. We will critique their own implementation of TLS, mentioned in Section 3.5.1, as there may exist flaws or bugs that can potentially compromise infrastructures by not using well established and rigorously tested frameworks for transport layer security.

On the subject of the disputed CVE (CVE-2016-6595) registered for Docker SwarmKit, the developing community responded quickly to the incident, and we agree with Diogo Mónica that this is not CVE worthy, as the DoS-attack cannot be performed by an adversary without access to a valid join-token. In this thesis we showed that sending non-valid join-tokens to a manager node, did not exhaust resources on the manager, and that input from an unauthenticated user was handled in a way that satisfies the security requirement. Zovi

[1]`https://radar.bekk.no/tech2018/sprak-og-rammeverk/protobuf`

et al. [45] discusses the fact that *least privilege orchestration* model minimises the attack surface exposed to a compromised container. He describes the experiment with Shellshock (CVE-2014-0160) as ”pretty uninteresting”, because the attacker would get a shell in the container, and the attack surface is only a highly constrained container on a single system. This results in few avenues for lateral movement, with remaining attacks possible being for example traditional data plane attacks, container escapes and OS privilege escalation.

The big cloud providers, Amazon, Google and Microsoft, all heavily depend on containers. Containers provide the infrastructure for creating the next generation of cloud services. Server-less computing (also referred to as ”Backend as a service” or ”Function as a service”) is on the rise as the next big thing. The developer writes the code, sets a few parameters and uploads the code to e.g., AWS Lambda or Google Cloud Functions, and when the code is called the code is deployed to a container. When the function is done executing, the container disappears.

6.2 Conclusion

After working through all parts of this thesis we are able to answer the research questions presented in Chapter 1:

- *”What security mechanisms are used in Docker Swarm?”*
  We uncover that Docker Swarm depends on a variety of existing frameworks for providing a secure orchestration service. The use of fairly new technology stack limits the attack surface as we see with Golang not supporting older versions of TLS for encrypting network traffic. The use of Protocol buffers and gRPC empowers Docker Swarm to leave a small footprint on the network by using a binary protocol and few TCP-connections. Ergo the security features pose a negligible loss in performance.
  As nodes show to have strong identity with the use of X.509-certificates, the join-token limits access to a swarm in an acceptable way, and the traffic for control and management is encrypted where possible. Docker is not vulnerable to timing attacks as it uses `ConstantTimeCompares` that in a mutex lock enforces limits on access to a resource during the comparison. The use of good security practices such as secure defaults, strong cryptographic identities and namespaces results in us having gained

confidence in that Docker Swarm was developed with a secure and resilience mindset from the very beginning. We are however concerned by the fact that Docker Content Trust is not enabled by default, even though it is not a part of Docker Swarm. This is in our view in direct contradiction with the principle of secure by default.

- *"Is Docker Swarm secure against high level attacks?"*
  Docker Swarm has also shown resilience towards high level attacks, and we found no vulnerabilities for the cases we tested on the current version. The join-tokens length provides enough entropy with 25 characters that we do not need to recommend adding upper case and non-alphanumeric characters to increase the search space. No misformatted requests were accepted by the manager, for the join-token or for tampering with X.509-CSRs. We consider the security satisfactory.
- *"If the security is proven strong, what other applications can benefit from applying this model?"*
  The architecture and ideas are better than many existing solutions and can be adopted in other settings for distributed systems. For service-to-service communication the use of Protocol buffers and gRPC provides a better infrastructure for service calls in micro service architecture than REST does, as security and type safety has been incorporated and enforced at compilation.

The experiments creating a GCD application and custom proxy server for that application shows that the use of Protocol buffers and gRPC is a secure choice as it reduces the attack surface of an application. Our implementations can be found at `github.com/Diddern/gRPC-simpleGCDService` and `github.com/Diddern/gIntercept`. In addition to the implementations, our contribution is a comprehensive investigation to the security features of Docker Swarm. It provides documentation and insight to undisclosed and dispersed information.

Apart from the security of Docker networking, and the underlying technology of Docker, much of the security will rely on the trust we can have in the veracity of the container images, as discussed in Section 4.4.3. The developers must be cautious when picking images for hosting their applications.

Docker Swarm is built of existing language components and frameworks and does not reinvent the wheel where it is not needed. It provides a simple to use, secure by default set of best practices for managing a cluster of nodes. It has quickly become the de facto standard for orchestration services and provides a secure platform and API for developers to manage their applications. With larger applications that cannot run as a one-time-function, a scalable container cluster demands a secure and reliable orchestration service. We are confident that Docker Swarm is currently a secure and good solution for administration of such a cluster. Both in regard to flexibility, and functionality, but also in terms of creating a secure tool for employing faster moving development in DevOps spirit. The splitting of Dockers components into upstream projects, and downstream products is consistent with the Unix philosophy of *"do one thing well"*, and makes it easy to contribute to the open source project as it will be less overwhelming.

6.3 Future work

The presented research in this thesis is a good starting point for further security analysis and development of service orchestration solutions. The test cases we looked into point to the fact that it is possible to create a proxy tool for inspecting swarm traffic and provide us with a way to efficiently decode the gRPC traffic. When all gRPC calls for Docker Swarm can be decrypted, a formal proof of security can be made to verify the security of the service. In no particular order we suggest the following list as possible future work for strengthening the trust in Docker Swarm:

- Implement gRPC proxy that will automatically parse provided protocol buffer description. Alternatively, create a Wireshark plugin for Protocol buffers version 3 (`proto3`) type files. This will provide a good basis for a debugging tool for gRPC calls, as there is no such tool equivalent to Postman or Curl for REST.
- Apply formal verification techniques to prove (or disprove) the correctness of the security protocols used in Docker Swarm. Since the protocol for a new node joining an existing swarm was extensively examined in this thesis, this protocol is a good first candidate.
- Compare other orchestration services (Apache Mesos and Kubernetes) to Docker Swarm in a security perspective.
- Security analysis of Golang's TLS libraries. We predict that investigation has been done, but the results have not been disclosed to the public. Multiple studies are needed to provide confidence in the security of a TLS library.

- As we see in this thesis, trusting the publisher of images for the containers can be difficult. An investigation of the difficulty of activating Docker Content Trust as the default configuration and validating the security should be performed. The impact of only being able to use signed images should also be measured.
- Inspect the use of ASLR (Address space layout randomisation) in Golang. Golang is stated to be memory safe but does not use ASLR. If libraries written in C/C++ are imported, the application can suffer from memory corruption vulnerabilities. Golang binaries can be compiled with PIE (Position-independent code), why is not Docker using it?
- Improvement of the verbose logging for Docker Swarm. This can be very useful in debugging of Swarm problems.

Glossary

AES	Advanced Encryption Standard is a specification for the encryption of electronic data. The algorithm described by AES is a symmetric-key algorithm, meaning the same key is used for both encrypting and decrypting the data.
CVE	Common Vulnerabilities and Exposures (CVE) system provides a reference-method for publicly known information-security vulnerabilities and exposures. The maintainer of the system is Mitre Corporation, with funding from the National Cyber Security Division of the United States Department of Homeland Security.
DevOps	Combination of "Development" and "operations". Set of practices intended to reduce the time from committing code to the code being live in production. Spawns from the Agile software development methods, and has common goals as continues delivery practices.
GCM	Galois/Counter Mode is a block cipher mode of operation for symmetric key cryptography. It uses a universal hash function over a binary Galois field to provide authenticated encryption.
struct	Golang structs are typed collections of fields. Golang supports empty structs as well as nested structs, and importantly structs are mutable.
Zero day	A zero-day (or 0-day) vulnerability refers to a security flaw in software that is unknown to the software maker or to antivirus vendors until they are disclosed.

List of Acronyms and Abbreviations

API	application programming interface.
ARP	address resolution protocol.
CA	certificate authority.
CE	community edition.
CI	continuous integration.
CLI	command line interface.
CRL	certificate revocation list.
CSR	certificate signing request.
DoS	denial-of-service.
ECC	elliptic curve cryptography.
EE	enterprise edition.
GCD	greatest common divisor.
gRPC	gRPC remote procedure calls.
IDL	interface definition language.
JSON	javascript object notation.
MITM	man-in-the-middle.
MTLS	mutual transport layer security.
NIC	network interface card.
OCSP	online certificate status protocol.
OS	operating system.
PKI	public key infrastructure.
SOA	service oriented architecture.
SSD	system sequence diagram.
TLS	transport layer security.
TOFU	trust on first use.
VCS	version control system.
VM	virtual machine.
WAL	write-ahead log.

Bibliography

[1] Mustafa Emre Acer, Emily Stark, Adrienne Porter Felt, Sascha Fahl, Radhika Bhargava, Bhanu Dev, Matt Braithwaite, Ryan Sleevi, and Parisa Tabriz. Where the wild warnings are: Root causes of chrome https certificate errors. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 1407–1420, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4946-8. doi: 10.1145/3133956.3134007.

[2] Marcel Brouwers. Security considerations in Docker Swarm networking. Master's thesis, University of Amsterdam, Netherlands, 2017.
URL: `http://work4.delaat.net/rp/2016-2017/p53/report.pdf`.

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

[4] Mark Church. Docker - Docker Reference Architecture: Designing Scalable, Portable Docker Container Networks.
URL: `https://success.docker.com/article/networking`. Last visited 2018-06-20.

[5] Victor Coisne. Get involved with the Moby Project by attending upcoming Moby summits!, 2017.
URL: `https://blog.docker.com/2017/05/get-involved-moby-project-attending-upcoming-moby-summits/`. Last visited 2018-06-20.

[6] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed Systems: Concepts and Design*. Addison-Wesley Publishing Company, USA, 5th edition, 2011. ISBN 0132143011, 9780132143011.

[7] Benoit des Ligneris. Virtualization of Linux based computers: the Linux-VServer project. In *19th International Symposium on High Performance Computing Systems and Applications, (HPCS 2005)*, pages 340–346. IEEE, 2005.

[8] DockerInc. *Manage swarm security with public key infrastructure (PKI) — Docker Documentation*, 2018.
URL: `https://docs.docker.com/engine/swarm/how-swarm-mode-works/pki/`. Last visited 2018-06-20.

[9] DockerInc. *Docker Swarm (standalone) release notes — Docker Documentation*, 2018.
URL: `https://docs.docker.com/release-notes/docker-swarm/#04-2015-08-04`. Last visited 2018-02-09.

[10] DockerInc. *Manage sensitive data with Docker secrets — Docker Documentation*, n.d.
URL: `https://docs.docker.com/engine/swarm/secrets/#how-docker-manages-secrets`.
Last visited 2018-05-06.

[11] Thomas Erl. *Service-oriented architecture (SOA): concepts, technology, and design.* Prentice Hall Englewood Cliffs, 2005. ISBN 0131858580.

[12] Cloud Native Computing Foundation. *Authentication*, n.d.
URL: `https://grpc.io/docs/guides/auth.html`. Last visited 2018-07-02.

[13] Google Inc. Developer guide - why not just use xml? Technical report, 2017.
URL: `https://developers.google.com/protocol-buffers/docs/overview#whynotxml`.
Last visited 2018-05-10.

[14] Google Inc. *Frequently Asked Questions (FAQ) - The Go Programming Language*, 2018.
URL: `https://golang.org/doc/faq#Is_Go_an_object-oriented_language`. Last visited 2018-06-10.

[15] Ravi Honnavalli. Docker notary: Very TUF, but devil is in the detail!, 2017.
URL: `https://medium.com/walmartlabs/docker-notary-very-tuf-but-devil-is-in-the-detail-5e643ea0aa16`. Last visited 2018-05-25.

[16] Troy Hunt. I wanna go fast: HTTPS' massive speed advantage, 2016.
URL: `https://www.troyhunt.com/i-wanna-go-fast-https-massive-speed-advantage/`.
Last visited 2018-06-18.

[17] GitHub Inc. Github octoverse 2017 — highlights from the last twelve months, 2017.
URL: `https://octoverse.github.com/`. Last visited 2018-06-27.

[18] Nic Jackson. *Building Microservices with Go.* Packt Publishing Ltd, 2017. ISBN 139781786468666.

[19] Nicolai M Josuttis. *SOA in practice: the art of distributed system design.* O'Reilly Media, Inc., 2007. ISBN 0596529554.

[20] Luc Juggery. Raft logs on swarm mode, 2017.
URL: `https://medium.com/lucjuggery/raft-logs-on-swarm-mode-1351eff1e690`. Last visited 2018-07-02.

[21] Jon-Anders Kabbe. Security analysis of Docker containers in a production environment. Master's thesis, Norwegian University of Science and Technology, 2017.
URL: `https://brage.bibsys.no/xmlui/handle/11250/2451326`.

[22] Poul-Henning Kamp and Robert NM Watson. Jails: Confining the omnipotent root. In *Proceedings of the 2nd International SANE Conference*, page 116, May 2000.

[23] Bruno Krebs. Beating JSON performance with protobuf, 2017.
URL: `https://auth0.com/blog/beating-json-performance-with-protobuf/`. Last visited 2018-05-10.

[24] Adam Langley. No, don't enable revocation checking, 2014.
URL: `https://www.imperialviolet.org/2014/04/19/revchecking.html`. Last visited 2018-04-20.

[25] Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), March 2014. ISSN 1075-3583.
URL: `http://dl.acm.org/citation.cfm?id=2600239.2600241`.

[26] Mark S. Merkow and Lakshmikanth Raghavan. *Secure and Resilient Software Development.* Auerbach Publications, Boston, MA, USA, 1st edition, 2010. ISBN 9781439826966.

[27] Adrian Mouat. *Using Docker: Developing and Deploying Software with Containers.* O'Reilly Media, Inc., 2016. ISBN 9781491915769.

[28] Diogo Mónica. Least privilege container orchestration, 2017.
URL: `https://blog.docker.com/2017/10/least-privilege-container-orchestration/`.
Last visited 2018-02-09.

[29] Diogo Mónica. Secure Substrate: Least Privilege Container Deployment, 2017.
URL: `https://www.youtube.com/watch?v=iHQCVFMBdCA`. Last visited 2018-03-20.

[30] Sam Newman. *Building microservices: designing fine-grained systems.* O'Reilly Media, Inc., 2015. ISBN 9781491950319.

[31] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 305–320, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-10-2.

[32] Bryan Payne. PKI at scale using short-lived certificates. San Francisco, CA, 2016. USENIX Association.

[33] J Ronald Prins and Business Unit Cybercrime. DigiNotar certificate authority breach 'operation black tulip'. *Fox-IT, November*, 2011.

[34] Dan Radigan. Continuous integration, n.d.
URL: `https://www.atlassian.com/agile/software-development/continuous-integration`.
Last visited 2018-02-09.

[35] Ronald L. Rivest. Can we eliminate certificate revocations lists? In *Proceedings of the Second International Conference on Financial Cryptography*, FC '98, pages 178–183, London, UK, UK, 1998. Springer-Verlag. ISBN 3-540-64951-4.

[36] Andrew S Tanenbaum and Maarten Van Steen. *Distributed systems: principles and paradigms.* CreateSpace Independent Publishing Platform, 2nd edition, 2016. ISBN 9781530281756.

[37] The Pokémon GO team. Celebrating the first anniversary of Pokémon GO!, 2017.
URL: `https://pokemongolive.com/en/post/anniversary2017`. Last visited 2018-06-01.

[38] Ray Tsang. Fast & efficient microservices with HTTP/2 and gRPC, 2017.
URL: `https://vimeo.com/233788634`. Last visited 2018-06-03.

[39] Shiju Varghese. Inter-Process Communication in Microservices using gRPC, 2017.
URL: `https://www.slideshare.net/shijucv/interprocess-communication-in-microservices-using-grpc`. Last visited 2018-06-20.

[40] Madhu Venugopal. Dcus17 : Docker networking deep dive, 2017.
URL: `https://www.slideshare.net/MadhuVenugopal2/dcus17-docker-networking-deep-dive`.
Last visited 2018-05-11.

[41] JC van Winkel. The production environment at google, from the viewpoint of an SRE. In Betsy Beyer, editor, *Site Reliability Engineering: How Google Runs Production Systems*, chapter 2. O'Reilly Media, Inc., 2016.
URL: `http://landing.google.com/sre/book.html`.

[42] Tetiana Yarygina and Anya Helene Bagge. Overcoming security challenges in microservice architectures. In *12th IEEE Symposium on Service-Oriented System Engineering (SOSE 2018)*, pages 11–20. IEEE, March 2018. ISBN 978-1-5386-5207-7. doi: 10.1109/SOSE.2018.00011.

[43] Yarygina, Tetiana. Restful is not secure. In Lynn Batten, Dong Seong Kim, Xuyun Zhang, and Gang Li, editors, *Applications and Techniques in Information Security*, pages 141–153, Singapore, 2017. Springer Singapore. ISBN 978-981-10-5421-1.

[44] Keyi Zhang. The paradigms — The Go Programming Language Report, 2015.
URL: `https://kuree.gitbooks.io/the-go-programming-language-report/content/2/text.html`. Last visited 2018-06-10.

[45] Dino Dai Zovi, Brandon Edwards, Hrushikesh Kalburgi, and Kent Ma. Datacenter orchestration security and insecurity: Assessing Kubernetes, Mesos, and Docker at scale, 2017.
URL: `https://www.youtube.com/watch?v=lXggHTqznOI`. Last visited 2018-05-27.

Appendix A

Generated code from Protocol buffers

Listing A.1: Generated class in Golang for descriptor in Listing 3.1

```go
/* Package pb is a generated protocol buffer package.

It is generated from these files:
    gcd.proto

It has these top-level messages:
    GCDRequest
    GCDResponse
*/
package pb

import proto "github.com/golang/protobuf/proto"
import fmt "fmt"
import math "math"

import (
    context "golang.org/x/net/context"
    grpc "google.golang.org/grpc"
)

//Reference imports to suppress errors if they are not otherwise used.
var _ = proto.Marshal
var _ = fmt.Errorf
var _ = math.Inf

//This is a compile-time assertion to ensure that this generated file is
   ↪ compatible with the proto package it is being compiled against. A
   ↪ compilation error at this line likely means your copy of the proto
   ↪ package needs to be updated.
const _ = proto.ProtoPackageIsVersion2 //please upgrade the proto package

type GCDRequest struct {
    A uint64 `protobuf:"varint,1,opt,name=a" json:"a,omitempty"`
    B uint64 `protobuf:"varint,2,opt,name=b" json:"b,omitempty"`
}

func (m *GCDRequest) Reset()                    { *m = GCDRequest{} }
func (m *GCDRequest) String() string            { return
   ↪ proto.CompactTextString(m) }
func (*GCDRequest) ProtoMessage()               {}
func (*GCDRequest) Descriptor() ([]byte, []int) { return fileDescriptor0,
    ↪ []int{0} }

func (m *GCDRequest) GetA() uint64 {
    if m != nil {
        return m.A
    }
    return 0
}

func (m *GCDRequest) GetB() uint64 {
    if m != nil {
        return m.B
    }
    return 0
}

type GCDResponse struct {
    Result uint64 `protobuf:"varint,1,opt,name=result"
        ↪ json:"result,omitempty"`
}

func (m *GCDResponse) Reset()                    { *m = GCDResponse{} }
func (m *GCDResponse) String() string            { return
    ↪ proto.CompactTextString(m) }
func (*GCDResponse) ProtoMessage()               {}
func (*GCDResponse) Descriptor() ([]byte, []int) { return
    ↪ fileDescriptor0, []int{1} }

func (m *GCDResponse) GetResult() uint64 {
    if m != nil {
        return m.Result
    }
    return 0
}

func init() {
    proto.RegisterType((*GCDRequest)(nil), "pb.GCDRequest")
    proto.RegisterType((*GCDResponse)(nil), "pb.GCDResponse")
}

//Reference imports to suppress errors if they are not otherwise used.
var _ context.Context
var _ grpc.ClientConn

//This is a compile-time assertion to ensure that this generated file is
    ↪ compatible with the grpc package it is being compiled against.
const _ = grpc.SupportPackageIsVersion4

//Client API for GCDService service

type GCDServiceClient interface {
    Compute(ctx context.Context, in *GCDRequest, opts ...grpc.CallOption)
        ↪ (*GCDResponse, error)
}

type gCDServiceClient struct {
    cc *grpc.ClientConn
}

func NewGCDServiceClient(cc *grpc.ClientConn) GCDServiceClient {
    return &gCDServiceClient{cc}
}
```

```

func (c *gCDServiceClient) Compute(ctx context.Context, in *GCDRequest,
    ↪ opts ...grpc.CallOption) (*GCDResponse, error) {
    out := new(GCDResponse)
    err := grpc.Invoke(ctx, "/pb.GCDService/Compute", in, out, c.cc,
        ↪ opts...)
    if err != nil {
        return nil, err
    }
    return out, nil
}

//Server API for GCDService service

type GCDServiceServer interface {
    Compute(context.Context, *GCDRequest) (*GCDResponse, error)
}

func RegisterGCDServiceServer(s *grpc.Server, srv GCDServiceServer) {
    s.RegisterService(&_GCDService_serviceDesc, srv)
}

func _GCDService_Compute_Handler(srv interface{}, ctx context.Context,
    ↪ dec func(interface{}) error, interceptor
    ↪ grpc.UnaryServerInterceptor) (interface{}, error) {
    in := new(GCDRequest)
    if err := dec(in); err != nil {
        return nil, err
    }
    if interceptor == nil {
        return srv.(GCDServiceServer).Compute(ctx, in)
    }
    info := &grpc.UnaryServerInfo{
        Server:     srv,
        FullMethod: "/pb.GCDService/Compute",
    }
    handler := func(ctx context.Context, req interface{}) (interface{},
        ↪ error) {
        return srv.(GCDServiceServer).Compute(ctx, req.(*GCDRequest))
    }
    return interceptor(ctx, in, info, handler)
}

var _GCDService_serviceDesc = grpc.ServiceDesc{
    ServiceName: "pb.GCDService",
    HandlerType: (*GCDServiceServer)(nil),
    Methods: []grpc.MethodDesc{
        {
            MethodName: "Compute",
            Handler:    _GCDService_Compute_Handler,
        },
    },
    Streams:  []grpc.StreamDesc{},
    Metadata: "gcd.proto",
}

func init() { proto.RegisterFile("gcd.proto", fileDescriptor0) }

var fileDescriptor0 = []byte{
    //144 bytes of a gzipped FileDescriptorProto
    0x1f, 0x8b, 0x08, 0x00, 0x00, 0x00, 0x00, 0x00, 0x02, 0xff, 0xe2,
        ↪ 0xe2, 0x4c, 0x4f, 0x4e, 0xd1,
    0x2b, 0x28, 0xca, 0x2f, 0xc9, 0x17, 0x62, 0x2a, 0x48, 0x52, 0xd2,
        ↪ 0xe0, 0xe2, 0x72, 0x77, 0x76,
    0x09, 0x4a, 0x2d, 0x2c, 0x4d, 0x2d, 0x2e, 0x11, 0xe2, 0xe1, 0x62,
        ↪ 0x4c, 0x94, 0x60, 0x54, 0x60,
    0xd4, 0x60, 0x09, 0x62, 0x4c, 0x04, 0xf1, 0x92, 0x24, 0x98, 0x20,
        ↪ 0xbc, 0x24, 0x25, 0x55, 0x2e,
    0x6e, 0xb0, 0xca, 0xe2, 0x82, 0xfc, 0xbc, 0xe2, 0x54, 0x21, 0x31,
        ↪ 0x2e, 0xb6, 0xa2, 0xd4, 0xe2,
    0xd2, 0x9c, 0x12, 0xa8, 0x7a, 0x28, 0xcf, 0xc8, 0x0a, 0x6c, 0x60,
        ↪ 0x70, 0x6a, 0x51, 0x59, 0x66,
    0x72, 0xaa, 0x90, 0x0e, 0x17, 0xbb, 0x73, 0x7e, 0x6e, 0x41, 0x69,
        ↪ 0x49, 0xaa, 0x10, 0x9f, 0x5e,
    0x41, 0x92, 0x1e, 0xc2, 0x2e, 0x29, 0x7e, 0x38, 0x1f, 0x62, 0xa2,
        ↪ 0x12, 0x43, 0x12, 0x1b, 0xd8,
    0x5d, 0xc6, 0x80, 0x00, 0x00, 0x00, 0xff, 0xff, 0x00, 0x5f, 0x20,
        ↪ 0xdc, 0xa4, 0x00, 0x00, 0x00,
}
```

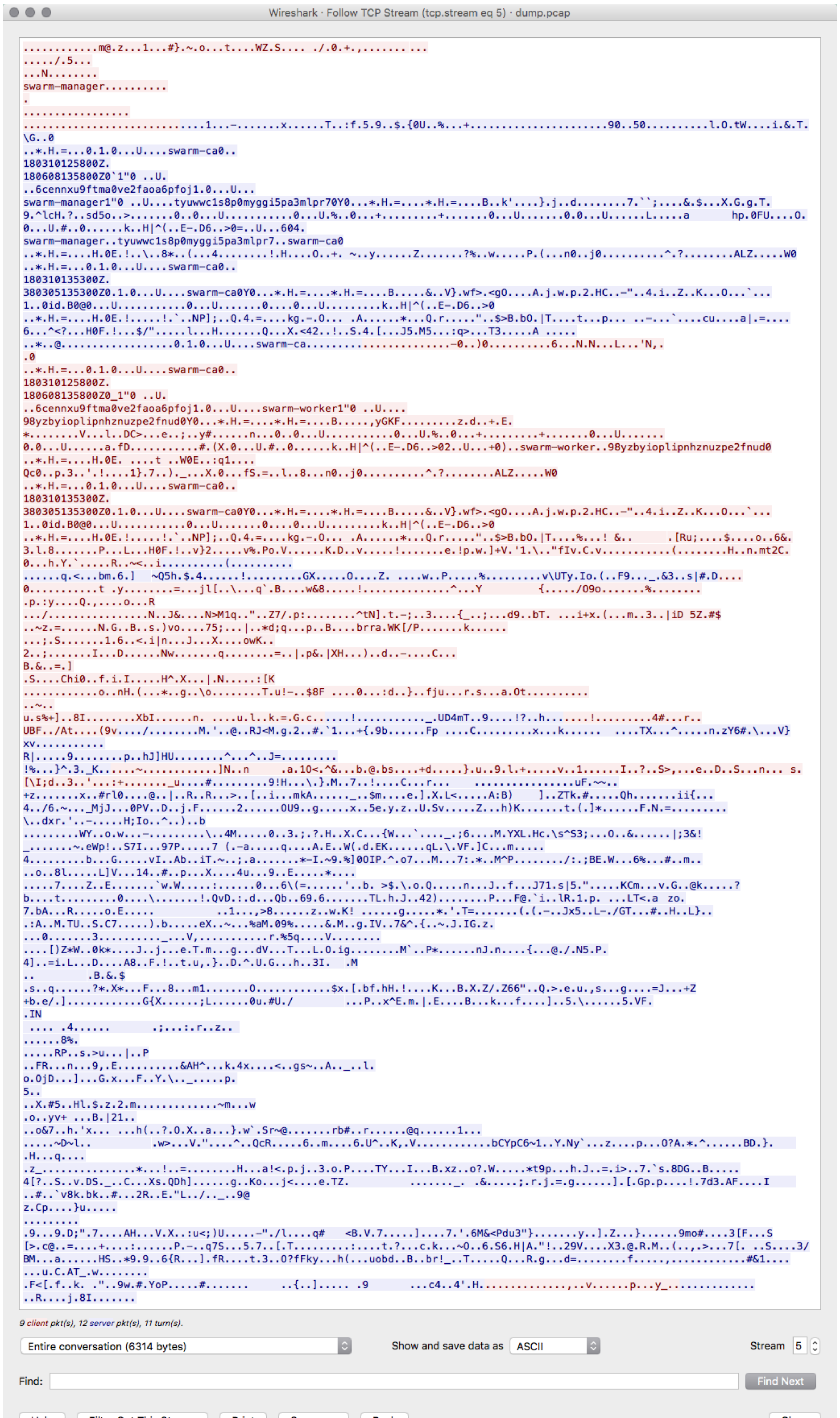
Appendix B

Wireshark output

Wireshark has a feature to follow a TCP stream. In Figure B.1 we show a printscreen of the join-sequence captured by Wireshark. Text in red is the joining worker-node, and the blue is the manager. This ”conversation” is encrypted, as Wireshark does not support decryption of this stream as we see in Section 5.4.1.

The text with red background is sent from the worker node (`192.168.245.1`) and the text with the blue background is sent from the manager node (`192.168.245.133`).

Figure B.1: Following the encrypted TCP Stream with Wireshark of a join-sequence



Wireshark · Follow TCP Stream (tcp.stream eq 5) · dump.pcap

```
............m@.z...1...#}.~.o...t....WZ.S.... ./.0.+.,........ ...
...../.5...
...N........
swarm-manager..........
.
.................
..............................1...-.......x......T..:f.5.9..$.{0U..%...+.....................90..50..........l.O.tW....i.&.T.
\G..0
..*.H.=...0.1.0...U....swarm-ca0..
180310125800Z.
180608135800Z0`1"0 ..U.
..6cennxu9ftma0ve2faoa6pfoj1.0...U...
swarm-manager1"0 ..U....tyuwwc1s8p0myggi5pa3mlpr70Y0...*.H.=....*.H.=....B..k'....}.j..d........7.``;....&.$...X.G.g.T.
9.^lcH.?..sd5o..>.......0..0...U...........0...U.%..0...+.........+.......0...U.......0.0...U......L.....a       hp.0FU....O.
0...U.#..0......k..H|^(..E-.D6..>0=..U...604.
swarm-manager..tyuwwc1s8p0myggi5pa3mlpr7..swarm-ca0
..*.H.=....H.0E.!..\..8*..(...4........!.H....O..+. ~..y......Z.......?%..w.....P.(...n0..j0..........^.?........ALZ.....W0
..*.H.=...0.1.0...U....swarm-ca0..
180310135300Z.
380305135300Z0.1.0...U....swarm-ca0Y0...*.H.=....*.H.=....B.....&..V}.wf>.<gO....A.j.w.p.2.HC..-"..4.i..Z..K...O...`...
1..0id.B0@0...U...........0...U.......0....0...U.........k..H|^(..E-.D6..>0
..*.H.=....H.0E.!.....!.`..NP];..Q.4.=....kg.-.O... .A......*...Q.r....."..$>B.bO.|T....t...p... ..-...`....cu....a|.=....
6...^<?...H0F.!...$/".....l...H.......Q...X.<42..!..S.4.[...J5.M5...:q>...T3.....A .....
..*..@..................0.1.0...U....swarm-ca..........................-0..)0..........6...N.N...L...'N,.
.0
..*.H.=...0.1.0...U....swarm-ca0..
180310125800Z.
180608135800Z0_1"0 ..U.
..6cennxu9ftma0ve2faoa6pfoj1.0...U....swarm-worker1"0 ..U....
98yzbyioplipnhznuzpe2fnud0Y0...*.H.=....*.H.=....B.....,yGKF.........z.d..+.E.
*........V...l..DC>...e..;..y#......n...0..0...U...........0...U.%..0...+.........+.......0...U.......
0.0...U......a.fD...........#.(X.0...U.#..0......k..H|^(..E-.D6..>02..U...+0)..swarm-worker..98yzbyioplipnhznuzpe2fnud0
..*.H.=....H.0E. ....t ..W0E..:q1....
Qc0..p.3..'.!....1}.7..)._...X.0...fS.=..l..8...n0..j0..........^.?........ALZ.....W0
..*.H.=...0.1.0...U....swarm-ca0..
180310135300Z.
380305135300Z0.1.0...U....swarm-ca0Y0...*.H.=....*.H.=....B.....&..V}.wf>.<gO....A.j.w.p.2.HC..-"..4.i..Z..K...O...`...
1..0id.B0@0...U...........0...U.......0....0...U.........k..H|^(..E-.D6..>0
..*.H.=....H.0E.!.....!.`..NP];..Q.4.=....kg.-.O... .A......*...Q.r....."..$>B.bO.|T....%...! &..      .[Ru;....$....o..6&.
3.l.8.......P...L...H0F.!..v}2.....v%.Po.V......K.D..v.....!.......e.!p.w.]+V.'1.\.."fIv.C.v...........(........H..n.mt2C.
0...h.Y.`.....R..~<..i..........(..........
......q.<...bm.6.]  ~Q5h.$.4......!.........GX.....O....Z. ....w..P.....%.........v\UTy.Io.(..F9..._.&3..s|#.D....
0...........t .y........=...jl[..\...q`.B....w&8.....!..............^...Y         {...../O9o.......%........
.p.:y....Q.,.....o...R
.../................N..J&....N>M1q.."..Z7/.p:........^tN].t.-;..3....{_..;...d9..bT. ...i+x.(...m..3..|iD 5Z.#$
..~z.=......N.G..B..s.)vo....75;...|..*d;q...p..B....brra.WK[/P.......k......
...;.S.......1.6..<.i|n...J...X....owK..
2..;.......I...D......Nw.......q........=..|.p&.|XH...)..d..-....C...
B.&..=.]
.S....Chi0..f.i.I.....H^.X...|.N.....:[K
............o..nH.(...*..g..\o........T.u!-..$8F ....0...:d..}..fju...r.s...a.Ot..........
..~..
u.s%+]..8I........XbI......n. ....u.l..k.=.G.c......!..........._.UD4mT..9....!?..h........!.........4#...r..
UBF../At....(9v..../........M.'..@..RJ<M.g.2..#.`1...+{.9b......Fp ....C.........x...k......  ....TX...^.....n.zY6#.\...V}
xv...........
R|.....9........p..hJ]HU........^...^..J=.........
!%...}^.3._K......~............]N..n      .a.10<.^&...b.@.bs....+d.....}.u..9.l.+.....v..1......I..?..S>,...e..D..S...n... s.
[\I;d..3..'...:+......._u....#.........9!H...\.}.M..7..!....C...r...    .................uF.~~..
+z.......x..#rl0....@..|..R..R...>..[..i...mkA......_..$m....e.].X.L<.....A:B)    ]..ZTk.#.....Qh.......ii{...
4../6.~..._MjJ...0PV..D..j.F......2......OU9..g......x..5e.y.z..U.Sv.....Z...h)K.......t.(.]*......F.N.=.........
\..dxr.'..-.....H;Io..^..)..b
.........WY..o.w...-.........\..4M.....0..3.;.?.H..X.C...{W...`...._.;6....M.YXL.Hc.\s^S3;...O..&......|;3&!
_.......~.eWp!..S7I...97P.....7 (.-a.....q....A.E..W(.d.EK......qL.\.VF.]C...m.....
4.........b...G.....vI..Ab..iT.~..;.a.......*-I.~9.%]0OIP.^.o7...M...7:.*..M^P......../:.;BE.W...6%...#..m..
..o..8l.....L]V...14..#..p...X....4u...9..E.....*....
.....7....Z..E.......`w.W.....:......0...6\(=......'..b. >$.\.o.Q.....n...J..f...J71.s|5.".....KCm...v.G..@k.....?
b....t.........0....\.......!.QvD.:.d...Qb..69.6.......TL.h.J..42).........P...F@.`i..lR.1.p. ...LT<.a zo.
7.bA...R.....o.E.....           ..1...,>8......z..w.K! ......g.....*.'.T=.......(.(.-..Jx5..L-./GT...#..H..L}..
.:A..M.TU..S.C7.....).b.....eX..~...%aM.09%.....&.M..g.IV..7&^.{..~.J.IG.z.
...0.......3.........._...V,...........r.%5q....V........
....[)Z*W..0k*....J..j...e.T.m...g...dV...T...L.O.ig........M`..P*......nJ.n....{...@./.N5.P.
4]..=i.L...D....A8..F.!..t.u,.}..D.^.U.G...h..3I.   .M
..        .B.&.$
.s..q......?*.X*...F...8...m1.......O............$x.[.bf.hH.!....K...B.X.Z/.Z66"..Q.>.e.u.,s...g....=J...+Z
+b.e/.]............G{X......;L......0u.#U./         ...P..x^E.m.|.E....B...k...f....]..5.\......5.VF.
.IN
 .... .4......       .;...:.r..z..
......8%.
.....RP..s.>u...|..P
..FR...n...9,.E..........&AH^...k.4x....<..gs~..A.._..l.
o.OjD...]...G.x...F..Y.\.._.....p.
5..
..X.#5..Hl.$.z.2.m.............~m...w
.o..yv+ ...B.|21..
..o&7..h.'x... ...h(..?.O.X..a...}.w`.Sr~@.......rb#..r......@q......1...
.....~D~l..          .w>...V."....^..QcR.....6..m....6.U^..K,.V............bCYpC6~1..Y.Ny`...z....p...O?A.*.^......BD.}.
.H...q....
.z_...............*...!..=........H...a!<.p.j..3.o.P....TY...I...B.xz..o?.W.....*t9p...h.J..=.i>..7.`s.8DG..B.....
4[?..S..v.DS._..C...Xs.QDh]......g..Ko...j<....e.TZ.         ......._. .&.....;.r.j.=.g......].[.Gp.p....!.7d3.AF....I
..#..`v8k.bk..#...2R..E."L../.._..9@
z.Cp....}u.....
.........
.9...9.D;".7....AH...V.X..:u<;)U.....-"./l....q#   <B.V.7.....]....7.'.6M&<Pdu3"}.......y..].Z...}......9mo#....3[F...S
[>.c@..=....+....:......P.-..q7S...5.7..[.T.........:....t.?...c.k...~0..6.S6.H|A."!..29V....X3.@.R.M..(..,.>...7[. ..S....3/
BM...a......HS..*9.9..6{R...].fR....t.3..O?fFky...h(...uobd..B..br!_..T.....Q...R.g...d=........f....,............#&1....
...u.C.AT_.w........
.F<[.f..k. ."..9w.#.YoP.....#.......      ..{..]..... .9       ...c4..4'.H..............,..v......p...y_..............
..R....j.8I.......
```

9 client pkt(s), 12 server pkt(s), 11 turn(s).

Entire conversation (6314 bytes)

Show and save data as ASCII

Stream 5

Find:

Find Next

Appendix C

Snippets of code from Docker Swarm source code

Listing C.1: `GetRemoteSignedCertificate` from `swarmkit/ca/certificates.go`

```go
func GetRemoteSignedCertificate(ctx context.Context, csr []byte,
    ↪ rootCAPool *x509.CertPool, config CertificateRequestConfig)
    ↪ ([]byte, error) {
    if rootCAPool == nil {
        return nil, errors.New("valid root CA pool required")
    }
    creds := config.Credentials

    if creds == nil {
        //This is our only non-MTLS request, and it happens when we are
            ↪ boostraping our TLS certs We're using CARole as server
            ↪ name, so an external CA doesn't also have to have
            ↪ ManagerRole in the cert SANs
        creds = credentials.NewTLS(&tls.Config{ServerName: CARole,
            ↪ RootCAs: rootCAPool})
    }

    conn, err := getGRPCConnection(creds, config.ConnBroker,
        ↪ config.ForceRemote)
    if err != nil {
        return nil, err
    }

    //Create a CAClient to retrieve a new Certificate
    caClient := api.NewNodeCAClient(conn.ClientConn)

    issueCtx, issueCancel := context.WithTimeout(ctx, 5*time.Second)
    defer issueCancel()

    //Send the Request and retrieve the request token
    issueRequest := &api.IssueNodeCertificateRequest{CSR: csr, Token:
        ↪ config.Token, Availability: config.Availability}
    issueResponse, err := caClient.IssueNodeCertificate(issueCtx,
        ↪ issueRequest)

    if err != nil {
        conn.Close(false)
        return nil, err
    }

```

```
	statusRequest := &api.NodeCertificateStatusRequest{NodeID:
	    ↪ issueResponse.NodeID}
	expBackoff :=
	    ↪ events.NewExponentialBackoff(events.ExponentialBackoffConfig{
		Base:   time.Second,
		Factor: time.Second,
		Max:    30 * time.Second,
	})

	//Exponential backoff with Max of 30 seconds to wait for a new retry
	for {
		timeout := 5 * time.Second
		if config.NodeCertificateStatusRequestTimeout > 0 {
			timeout = config.NodeCertificateStatusRequestTimeout
		}
		//Send the Request and retrieve the certificate
		stateCtx, cancel := context.WithTimeout(ctx, timeout)
		defer cancel()
		statusResponse, err := caClient.NodeCertificateStatus(stateCtx,
		    ↪ statusRequest)
		switch {
		case err != nil && grpc.Code(err) != codes.DeadlineExceeded:
			conn.Close(false)
			//Because IssueNodeCertificate succeeded, if this call failed
			    ↪ likely it is due to an issue with this particular
			    ↪ connection, so we need to get another.  We should try a
			    ↪ remote connection - the local node may be a manager
			    ↪ that was demoted, so the local connection (which is
			    ↪ preferred) may not work.
			config.ForceRemote = true
			conn, err = getGRPCConnection(creds, config.ConnBroker,
			    ↪ config.ForceRemote)
			if err != nil {
				return nil, err
			}
			caClient = api.NewNodeCAClient(conn.ClientConn)

		//If there was no deadline exceeded error, and the certificate
		    ↪ was issued, return
		case err == nil && (statusResponse.Status.State ==
		    ↪ api.IssuanceStateIssued  statusResponse.Status.State ==
		    ↪ api.IssuanceStateRotate):
			if statusResponse.Certificate == nil {
				conn.Close(false)
				return nil, errors.New("no certificate in
				    ↪ CertificateStatus response")
			}

			//The certificate in the response must match the CSR we
			    ↪ submitted. If we are getting a response for a
			    ↪ certificate that was previously issued, we need to
			    ↪ retry until the certificate gets updated per our
			    ↪ current request.
			if bytes.Equal(statusResponse.Certificate.CSR, csr) {
				conn.Close(true)
				return statusResponse.Certificate.Certificate, nil
			}
		}

		//If NodeCertificateStatus timed out, we're still pending, the
		    ↪ issuance failed, or the state is unknown let's continue
		    ↪ trying after an exponential backoff
		expBackoff.Failure(nil, nil)
```

```go
        select {
        case <-ctx.Done():
            conn.Close(true)
            return nil, err
        case <-time.After(expBackoff.Proceed(nil)):
        }
    }
}
```

Note: In Line 30 of Listing C.2 we have translated a comment in Cyrillic script to english using Google Translate. The reason for this is that LaTeX does not treat unicode characters in listings well.

Listing C.2: `RequestAndSaveNewCertificates` from `swarmkit/ca/certificates.go`

```go
1  func (rca *RootCA) RequestAndSaveNewCertificates(ctx context.Context, kw KeyWriter, config CertificateRequestConfig) (*tls.Certificate, *IssuerInfo, error) {
2      //Create a new key/pair and CSR
3      csr, key, err := GenerateNewCSR()
4      if err != nil {
5          return nil, nil, errors.Wrap(err, "error when generating new node certs")
6      }
7  
8      //Get the remote manager to issue a CA signed certificate for this node. Retry up to 5 times in case the manager we first try to contact isn't responding properly (for example, it may have just been demoted).
9      var signedCert []byte
10     for i := 0; i != 5; i++ {
11         signedCert, err = GetRemoteSignedCertificate(ctx, csr, rca.Pool, config)
12         if err == nil {
13             break
14         }
15 
16         //If the first attempt fails, we should try a remote connection. The local node may be a manager that was demoted, so the local connection (which is preferred) may not work. If we are successful in renewing the certificate, the local connection will not be returned by the connection broker anymore.
17         config.ForceRemote = true
18 
19         //Wait a moment, in case a leader election was taking place.
20         select {
21         case <-time.After(config.RetryInterval):
22         case <-ctx.Done():
23             return nil, nil, ctx.Err()
24         }
25     }
26     if err != nil {
27         return nil, nil, err
28     }
29 
30     //Trust, but verify.
31     //Before we overwrite our local key + certificate, let's make sure the server gave us one that is valid. Create an X509Cert so we
```

```
        ↪ can .Verify(). Check to see if this certificate was signed by
        ↪ our CA, and isn't expired

    parsedCerts, chains, err := ValidateCertChain(rca.Pool, signedCert,
        ↪ false).
    //TODO(cyli): - right now we need the invalid certificate in order to
        ↪ determine whether or not we should download a new root, because
        ↪ we only want to do that in the case of workers.  When we have a
        ↪ single codepath for updating the root CAs for both managers and
        ↪ workers, this snippet can go.
    if _, ok := err.(x509.UnknownAuthorityError); ok {
        if parsedCerts, parseErr :=
            ↪ helpers.ParseCertificatesPEM(signedCert); parseErr == nil
            ↪ && len(parsedCerts) > 0 {
            return nil, nil, x509UnknownAuthError{
                error:          err,
                failedLeafCert: parsedCerts[0],
            }
        }
    }
    if err != nil {
        return nil, nil, err
    }

    //ValidateChain, if successful, will always return at least 1 parsed
        ↪ cert and at least 1 chain containing at least 2 certificates:
        ↪ the leaf and the root.
    leafCert := parsedCerts[0]
    issuer := chains[0][1]

    //Create a valid TLSKeyPair out of the PEM encoded private key and
        ↪ certificate
    tlsKeyPair, err := tls.X509KeyPair(signedCert, key)
    if err != nil {
        return nil, nil, err
    }

    var kekUpdate *KEKData
    for i := 0; i < 5; i++ {
        //ValidateCertChain will always return at least 1 cert, so
            ↪ indexing at 0 is safe
        kekUpdate, err = rca.getKEKUpdate(ctx, leafCert, tlsKeyPair,
            ↪ config)
        if err == nil {
            break
        }

        config.ForceRemote = true

        //Wait a moment, in case a leader election was taking place.
        select {
        case <-time.After(config.RetryInterval):
        case <-ctx.Done():
            return nil, nil, ctx.Err()
        }
    }
    if err != nil {
        return nil, nil, err
    }

    if err := kw.Write(NormalizePEMs(signedCert), key, kekUpdate); err !=
        ↪ nil {
        return nil, nil, err
    }

    return &tlsKeyPair, &IssuerInfo{
        PublicKey: issuer.RawSubjectPublicKeyInfo,
        Subject:   issuer.RawSubject,
    }, nil
}
```

Listing C.3: Generating the join-token for a swarm from `swarmkit/ca/config.go`

```
func GenerateJoinToken(rootCA *RootCA) string {
    var secretBytes [generatedSecretEntropyBytes]byte

    if _, err := cryptorand.Read(secretBytes[:]); err != nil {
        panic(fmt.Errorf("failed to read random bytes: %v", err))
    }

    var nn, digest big.Int
    nn.SetBytes(secretBytes[:])
    digest.SetString(rootCA.Digest.Hex(), 16)
    return fmt.Sprintf("SWMTKN-1-%0[1]*s-%0[3]*s", base36DigestLen,
       ↪ digest.Text(joinTokenBase), maxGeneratedSecretLength,
       ↪ nn.Text(joinTokenBase))
}
```

Appendix D

Certificate for a manager node

Disclaimer: We have put full certificates in this appendix, but they have all been revoked, and are not valid anymore, as they would pose a security risk to the writer of this thesis if they where.

Listing D.1: Full certificate of a manager node

```
$ openssl x509 -in swarm-node.crt -text
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            5e:7c:74:bb:3c:30:dc:1e:73:6d:b0:fa:8e:66:f7:94:80:cd:c5:54
    Signature Algorithm: ecdsa-with-SHA256
        Issuer: CN=swarm-ca
        Validity
            Not Before: Jun  4 14:14:00 2018 GMT
            Not After : Sep  2 15:14:00 2018 GMT
        Subject: O=j8e2jv0smsnmuiha3yjysm2sa, OU=swarm-manager,
          ↪ CN=c52rwgu9uy7mxdvvn98jzhiqf
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
                Public-Key: (256 bit)
                pub:
                    04:1b:80:56:92:d0:64:df:c1:59:34:22:98:03:dc:
                    d6:ad:94:ae:4a:45:cd:38:f7:61:4f:fb:eb:9b:f5:
                    ef:1a:57:8d:c4:e3:59:e5:5d:1b:6a:6d:7d:d6:b9:
                    60:4a:e5:9f:68:29:cb:6f:3d:c0:79:2f:f7:af:ba:
                    52:68:f2:75:c8
                ASN1 OID: prime256v1
                NIST CURVE: P-256
        X509v3 extensions:
            X509v3 Key Usage: critical
                Digital Signature, Key Encipherment
            X509v3 Extended Key Usage:
                TLS Web Server Authentication, TLS Web Client
                  ↪ Authentication
            X509v3 Basic Constraints: critical
                CA:FALSE
            X509v3 Subject Key Identifier:
                07:E4:2F:A6:3D:A9:08:37:72:DC:43:D5:6B
                   ↪ :95:D2:98:21:06:91:88
            X509v3 Authority Key Identifier:
                keyid:0A:9F:F5:72:54:A2:26:E7:E1:08:BB
                   ↪ :59:74:10:C8:E2:27:DE:6D:83

            X509v3 Subject Alternative Name:
                DNS:swarm-manager, DNS:c52rwgu9uy7mxdvvn98jzhiqf,
                   ↪ DNS:swarm-ca
    Signature Algorithm: ecdsa-with-SHA256
         30:46:02:21:00:d3:10:84:90:14:46:c7:e0:39:ac:98:21:81:
         6e:82:b6:5c:68:96:bf:4f:5c:b0:e9:0c:1c:f9:02:f5:b0:f6:
         bb:02:21:00:83:6b:96:b9:fc:8a:ee:40:76:b3:5f:4a:f8:e2:
         73:53:2a:ca:f6:82:14:f8:98:f7:c7:50:fe:83:22:26:c9:58
-----BEGIN CERTIFICATE-----
MIICNjCCAdugAwIBAgIUXnx0uzww3B5zbbD6jmb3lIDNxVQwCgYIKoZIzj0EAwIw
EzERMA8GA1UEAxMIc3dhcm0tY2EwHhcNMTgwNjA0MTQxNDAwWhcNMTgwOTAyMTUx
NDAwWjBgMSIwIAYDVQQKExlqOGUyanYwc21zbm11aWhhM3lqeXNtMnNhMRYwFAYD
VQQLEw1zd2FybS1tYW5hZ2VyMSIwIAYDVQQDExljNTJyd2d1OXV5N214ZHZ2bjk4
anpoaXFmMFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEG4BWktBk38FZNCKYA9zW
rZSuSkXNOPdhT/vrm/XvGleNxONZ5V0bam191rlgSuWfaCnLbz3AeS/3r7pSaPJ1
yKOBvzCBvDAOBgNVHQ8BAf8EBAMCBaAwHQYDVR0lBBYwFAYIKwYBBQUHAwEGCCsG
AQUFBwMCMAwGA1UdEwEB/wQCMAAwHQYDVR0OBBYEFAfkL6Y9qQg3ctxD1WuV0pgh
BpGIMB8GA1UdIwQYMBaAFAqf9XJUoibn4Qi7WXQQyOIn3m2DMD0GA1UdEQQ2MDSC
DXN3YXJtLW1hbmFnZXKCGWM1MnJ3Z3U5dXk3bXhkdnZuOThqemhpcWaCCHN3YXJt
LWNhMAoGCCqGSM49BAMCA0kAMEYCIQDTEISQFEbH4DmsmCGBboK2XGiWv09csOkM
HPkC9bD2uwIhAINrlrn8iu5AdrNfSvjic1MqyvaCFPiY98dQ/oMiJslY
-----END CERTIFICATE-----
```