TALLINN UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology

Tambet Paljasma 163516 IVCM

Validating Docker Image and Container Security Using Best Practices and Company Policies

Master's thesis

Supervisors: Margus Ernits

Renno Reinurm

Tallinn 2019

Author's declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Tambet Paljasma

02.01.2019

Abstract

This Master's thesis aims to provide guidelines and a tool that can be used to increase Docker image and container security.

People from different companies that use Docker will be interviewed to understand what do they think about Docker security and following best practices and company policies. Best practices provided by Docker and the community will be gathered along with policies from a company that uses Docker as its main service platform in live and test environments. Best practices are used due to Dockers official documentation lacks in security information. The author will try to provide a solution that could be used to automate the validation of Docker image building and container deployment according to the gathered guidelines. The solution will be tested out with a company that builds and deploys around 100 different containers a day.

The author gathered 37 different policies and best practices that can help with making Docker images building and container or service deployment more secure. The author created a new tool called Dora to automate the validation process of Docker image building and container or service deployment since a solution that included all the wanted features by the company could not be identified. Initially the need for the automated solution came from the company, but the author designed it to be usable by everyone.

The tool was tested out with the company and 267 issues were found. The tool helped to lower the number of issues by 71% in 5 weeks.

The thesis is written in English, contains 80 pages of text, seven chapters, and 12 figures.

Annotatsioon

Selle magistritöö peamine eesmärk on pakkuda juhised ja tööriist, mille abil saaks kasutaja suurendada Docker tõmmiste ja konteinerite turvalisust.

Autor intervjueerib inimesi erinevatest firmadest, et aru saada kuidas suhtutakse Dockeri turvalisusesse ja kui tähtis on nende arvates heade tavade ja firma suuniste järgimine. Dockeri ja selle kogukonna head tavad ja firma, kes kasutab Dockerit kui oma peamist teenuse platvormi toote ja test keskkonnas, reeglid kogutakse kokku. Autor püüab pakkuda lahendust, mille abil saaks automaatselt kontrollida, kas Dockeri tõmmiste ehitamine ja konteinerite kasutuselevõtt on kooskõlas määratud reeglitega.

Leitud lahendust ja kogutud head tavad ning reegleid testitakse koostöös sama firmaga, kelle käest reeglid koguti. Testi tulemusena on võimalik määrata kas lahendus on kasulik või mitte.

Autor korjas kokkus 37 erinevat reeglit, mis võivad aidata teha Docker tõmmiste ehitamist ja konteinerite kasutuselevõttu rohkem turvaliseks. Lahendust, mis rahuldaks kõiki firma vajadusi ja oleks võimeline kontrollima kõiki 37 reeglit. Autor lõi uue tööriista nimega Dora , mis sisaldab kõiki vajalikke omadusi ja on võimeline automaatselt valideerima Dockeri tõmmiste ehitamist ja konteinerite või teenuste kasutuselevõttu. Algselt tuli automaatse kontrollimise vajadus firma käest, kuid autor tahab, et see oleks igaühe poolt kasutatav.

Tööriista testiti koos firmaga ja leiti 267 erinevat viga Docker tõmmiste ehitamises ja konteinerite või teenuste kasutuselevõtus. Tööriist aitas 5 nädalaga alandada vigade arvu 71%.

Töö on kirjutatud inglise keeles ning sisaldab 80 lehekülge teksti, seitset peatükki ja 12 joonist.

Table on content

List of figures

1 Introduction

1.1 Motivation

Docker container enables the user to run multiple systems on the same host operating system. Creating Docker images and deploying containers from them has been made easy and fast by using tools that Docker provides. However it might not be so easy to know what and how should be done to keep them safe. Best practices from Docker or policies from an employer could help with that. These rules could be checked manually for smaller projects but does not scale well when there are hundreds of images created and containers deployed daily.

A company is in need of a solution that could help with validating if Docker best practices and their internal policies have been followed to increase the security of their services running inside containers. It should be stated that even though a company needs the solution then it should still be usable for anyone who uses Docker as well. In this company developers are responsible for deploying their written applications via a container based ecosystem. This involves creating Dockerfiles to build images and compose files to deploy them. The company has around 100 different deployments a day thanks to different automations. Different tools are used to validate the code that is running inside the container for different errors and security issues. But there is no way to verify if the images are built according to the guidelines provided by the security, operations and development operations team. Furthermore, there is not clear indication of what are the best practices provided by Docker that developer should follow. Checking all the Docker images by hand can take a lot of time and therefore lower the deployment time and productivity significantly. Creating an image without following these given practices and guidelines might lead to a serious security issue, broken application or just increase the deployment time.

An automated way to validate if Dockers´ best practices and company policies are taken into account when a Docker image is built or a container deployed is needed.

In the mentioned company's case, the solution should be part of the deployment pipeline. It should be easily modifiable so that different types of images or services could have different rules. The solution should also allow to override rules if necessary for certain services. Finding an issue should result in an educational warning to the developers so that they could fix the issue and learn in the process. Additionally, the solution should be lightweight and easily deployable to the different environments including the developers' computers. Being able to validate Docker image building and container deployment before they reach the deployment pipeline is very important for the company. This lowers the risk of faulty images or container reaching production environments and decreases the deployment queue for other services. Deployment should only be stopped if the risk is too high.

The solution should also be able to map found issues to their Github repositories. Furthermore, machine readable data must be gathered for statistical purposes. This data will be used to validate if the solution helps to lower the number of issues.

It should also be stated that scanning for any kind of different vulnerability in Docker images or containers will not be part of the solution and is out of scope for this paper.

1.2 Importance of following best practices and policies

The author of the thesis asked people from different companies that use Docker daily to provide their thoughts about docker security and how important it is that best practices and company policies are followed.

"*Container technology remains a relatively new technology, though in computer years it is basically ancient. The community that shares knowledge and improves adoption is also somewhat small. As with all things, best practices are guidance by first movers and industry standard bearers that provide a method for using the technology. Best practices are guidance*

in lieu of real knowledge. Because smart people have come together and recommended a way of doing things it is best to heed their advise or the cost of learning may be exorbitant. Internal to your organization, company policies provide a standard for configuration and operation that ensures needless distracting variable are removed from you operation. This limits the unknown variable and allow organisational knowledge, and in time wisdom, to be transferred and maintained between employees. In both cases, as the community or company matures, the best practices or policies can change to reflect progress and knowledge achieved." - Jesse Wojtkowiak Head of Information Security at Pipedrive (Governance, Compliance, Risk) CISSP

"Docker has managed to bundle different operating system isolation and security features together with a relatively simple interface, which so far has been used by big enterprises as a secret advantage. However, Docker started as a small startup and was aiming for rapid growth. In order to gain traction in the developer community, some security best practices were sacrificed and delegated to container operators to harden. But via heavy searching and constantly reading the documentation and following some GitHub conversations, it's possible to use tighten down container security profiles without using additional commercial products. Security, best practices and policies are a proactive approach to enable integrity, stability and continuity into services which are handling users' data. Without trust, the company will lose its customers very quickly." - Renno Reinurm Senior DevOps Engineer at Pipedrive, Docker Captain.

"*We take best practices seriously. It helps keeping docker image sizes manageable and build times under control. Snyk is used to scan against known security vulnerabilities in OS and dependency packages. Due to rapid product growth there are no hard enforced policies. But we train and educate our employees regularly and encourage sharing of best practices. All docker images must originate from one of the companies security-patched base images.*" - Priit Pääsuke DevOps Team Lead at Veriff

"There are many ways how to make docker setup insecure as surface of attack is quite big - it is divided between different levels. To make sure docker infrastructure is secure, standardization and strict control are required. Following best practices is needed because this is aggregated experience, which someone gather by doing mistakes - so to avoid any loss, it is quite reasonable to follow those. In some cases simply following best practices is not enough and thus company policies are introduced. These are usually designed to match specific business needs and required to be followed so that business could expand predictably and securely." - Jevgeni Smirnov Platform Team Lead at Bolt

1.3 Thesis background

In this paragraph, the author of the thesis will give a quick overview of containers, virtual machines, cloud computing and how they differ from each other. The author will also briefly describe why and how popular containers are. It should also be mentioned that systems running on Linux kernel will be taken as an example. This paragraph is meant to give a brief overview of terms that will be used throughout the whole thesis.

1.3.1 Containers

Containers are hosted in a physical or virtual server on top of their operating system or OS. Each container shares the host OS. This makes containers lightweight allowing them to be a few megabytes in size and take a few seconds to start. Containers have less management overhead, thanks to sharing a common operating system. That system may be patched and fixed if an issue arises. [2]

A container is able to contain an application with all its dependencies so that it could run isolated from other processes. Public cloud computing providers such as Amazon Web Service, Microsoft Azure, and Google Cloud Platform have already embraced containers. The popular choices for container management are Docker Swarm, Apache Mesos, and Kubernetes. [12]

Containers get their name from the shipping industry. Different products get to be placed into steel shipping containers, that are designed to be picked up by a crane and fit into a ship to accommodate the containers. The process is standardized, compact and costs less to ship each product by itself. [12]

In computer technology, a problem may arise when moving a piece of software to a different environment, such as installed OS, libraries, storage, security or network. [12]

Containers are not only capable of containing the software but all the dependencies including libraries, binaries and configuration files. Moving a container avoids the differences between environments. [12]

For more information about how containers differ from virtual servers and the rise of container popularity can be found in annex 1 to 3 in this thesis.

1.4 Existing solutions

The author reviewed different tools which check Docker containers and images for security vulnerabilities. Some of the tools do it in runtime some only when the image is created or a container deployed. The cost of these tools varies and some of them are even open-sourced and free. In this thesis, only free open-sourced solutions will be considered. This choice was made due to the fact that it can be difficult to try out payed products since not all of them have a trial period. More importantly this thesis is focusing on free open-sourced tools to provide a solution that anyone could use.

Scanning Docker images and containers for security vulnerabilities is out of the scope of this thesis. The author will focus on how to verify that different company policies and best practices for building Docker images and deploying them with these tools. The tools should be as lightweight as possible to not increase the total time of the deployment pipeline.

1.4.1 The Docker Bench for Security

The Docker Bench for Security [14] is a script that checks for dozens of common best-practices around deploying Docker containers in production. All the tests are automated and inspired by the CIS Docker Community Edition Benchmark v1.1.0. [15]. The Docker Bench for Security is described as a "container that tests containers″ by Docker's security lead, Diogo Mónica. The results have three levels - info, warning and passed. The Docker Bench of Security is written in bash and and can fully run in a container which makes it easy to run on different hosts or different environments.

The Docker Bench of Security is a good tool but does not fully provided a solution to the problem which is being solved as part of this thesis. The result can be written into standard output or into a file. Due to that it may be hard to use this data for future analyses. The fact that The Docker Bench of Security is fully written in bash can have its limitations and it could be hard to run all the needed checks. The Docker Bench for Security is meant for check running container and Docker hosts not how the container are being built or how they are deployed. Checking running containers is a valid approach but not the one which is being looked for in this thesis.

1.4.2 Drydock

Drydock[16] is a Docker audit tool inspired by The Docker Bench for Security. Fully written in python to provide more flexibility. Drydock also allows the creation and use of custom profiles to disable rules. Drydock also saves reports in JSON format for later analyses and parsing. Drydock heavily uses Docker SDK for Python[17] python client to communicate with Docker API.

Drydock is an improvement to The Docker Bench for Security in many ways but also shares some of the shortcomings. Just as The Docker Bench of Security drydock does not provide all the needed tests. The code should be changed to provide everything needed. Drydock does save the test report to a JSON format but additional tooling would be needed to use it with

another tool for statistics and analyses. The development of drydock seems to have stopped and the last change was made in 13.06.2016.

1.4.3 Docker Bench Test

Docker Bench Test [18] is another project inspired by The Docker Bench for Security. It is very similar to The Docker Bench of Security but has improvements in generating machine-readable test results for better analyses for the future. Docker Bench Test also allows to create tests per container and run only selected tests if needed. Docker Bench Test also does not report as much Info level messages as The Docker Bench for Security does. This was done to make the report more readable.

Just as Drydock, the project has gone inactive and the last commit to the Github repository was made on the 12th of September 2016. As stated before Docker Bench Test is very similar to The Docker Bench of Security. It is mostly meant to validate the configuration of Docker hosts and running containers. This is not sufficient for the problem which this thesis attempts to solve. There are not enough tests and the tool is written in bash and shell scripts.

1.4.4 Actuary

Yet another project inspired by The Docker Bench of Security. Actuary [19] is written in go language and has improvements in test configurations and saving results. Results can be outputted in JSON or XML format for later analyses. Actuary runs more tests by default and can also read new rule configurations from a file. The new rule configuration allows the user to use different values or override rules. Actuary has been developed by Diogo Mónica who is the security lead at Docker Inc.

Even though Actuary has improved The Docker Bench of Security functionality in many ways but it still mostly focused on validating rules for Docker daemon on the host. The last commit to the repository of this project was made on the 15th of September 2017 which makes it the most up-to-date tool from the looked at the list so far.

1.4.5 Anchore

Anchore [20] provides different teams like operations, developers or security to perform analysis on container images and executed commands inside the container. Anchore is able to generate reports and also gives the possibility to define policies that can be used in CI/CD pipelines. Anchore also has a plugin for Jenkins [21] to provide a seamless container based CI/CD pipeline.

Anchore is one of the best if not the best tool in this list but is also not completely free of charge. There is an open-source version of Anchore but it does not have all the useful functionality of the paid version. Although Anchore supports many different types of validation that are needed to check all the needed company policies than not all of them. At the time when the thesis creation Anchore did not support checking different rules for Docker container deployments. Different functionality could be added to the open-source version of Anchore, but the author believes it could take the same amount of time as creating a tool that does exactly what is needed for this thesis.

1.4.6 Existing solutions summary

Most of the five tools which have been looked at follow a similar pattern and should mostly be used to check the environment and the Docker containers running there for a set of rules. These rules are mostly best practices from Docker. Checking the running container is a good approach because then the user can be sure that the same container which would be used in the production environment is tested. Executing tests only against a running container can also make it very difficult to check if everything is done correctly by the companies policies. None of the tools provide the possibility to check how the Docker images deployment will be handled. As an example, there can be a company policy that requires certain types of containers to use a specific Docker network. Checking the connected network for a running container would mean that these networks need to exist in the test environment or on the host where the tests are running. Companies tend to run tests on separated hosta which may not

even be connected to the test environment Docker swarm. This host can, for example, be a Jenkins[21] or any other CI/CD tools´ agent. One requirement for the looked for the solution in this thesis was to be able to execute the tests as part of the deployment pipeline. Anchore provides this possibility but only on a Jenkins pipeline running fully on containers. Checking if a service or a container would be connected to the correct Docker network in a production environment could be done by analyzing the docker-compose file which will be used to deploy a service or the Docker command to run a container.

The author of this thesis strongly suggests using the most fitting tool from the ones reviewed to check if the used Docker environments and containers are coherent with Docker best practices.

1.5 Research methodology and research questions

This paragraph will go over the research methodology that was chosen for this thesis. The chosen research methodology will be presented taking into account all the guidelines for it. A set of research questions based on the research methodology will be answered.

1.5.1 Research methodology

The case study research methodology was chosen by the author of this thesis. Software engineering case study was chosen because it provides the possibility to examine the phenomena in are real-life settings. In the case studies, the phenomenon is expected to change in unanticipated ways. This allows a deeper understanding of the phenomena which are being studied. [24, pg 23]

Case study methodology is feasible when:

- *A case study has research questions set out from the beginning of the study.*
- *Collects data in a planned and consistent manner.*

- *Draws inferences from the data to answer the research question.*
- *Explores a phenomenon or produces an explanation, description, or causal analysis of the phenomenon.*
- *Addresses threats to validity in a systematic way.* [24, pg 19]

In the case study research process, the following five steps should be considered.

- *Case study design – objectives are defined and the case study is planned.*
- *Preparation for data collection – procedures and protocols for data collection are defined.*
- *Collecting evidence – data collection procedures are executed on the studied case.*
- *Analysis of collected data – data analysis procedures are applied to the data.*
- *Reporting – the study and its conclusions are packaged in feasible formats for reporting.* [24]

1.5.2 Case study design

The goal of the study should be a statement of what the researcher and other participants expect to achieve. It should be broken down into a set of research questions. These questions should be answered throughout the whole study by data collection and analysis. The questions are summarized in Figure 1 in annex 4.

- Rationale

  A company is in need of an automated way to validate if building Docker images and deploying the services created from these images is done by Dockers' best practices and the company's' policies. Most of the images are deployed as part of a Docker service. The validation should be done as part of the deployment pipeline. The solution should provide educational information so that employees could learn from them. The solution should be flexible so that different rules which should be checked and sent messages could be configured with ease. The solution should be able to relate found issues to Github repositories for these services. Each Github repository has a team who is responsible for it and must host all the necessary files and configurations for a deployment like dockerfile for image building and docker-compose file for service deployment.

- Purpose

  The purpose of this study to provide a solution that can be configured to check different rules for Docker image building and deploying containers or services.

  To provide more insight to Docker best practices about image building and their deployment.

  Give an overview of the existing solution and if any of them could be used for these needs.

- The case

  The first step is to understand and gather data about different similar solutions, give an overview of Docker and its different features. Followed by an overview of Docker best practices and company policies. Giving an overview of the deployment pipeline that is used by the company under study. A possible solution will be proposed. Data from when the solution was first implemented to the end of the study will be analyzed to figure out if the solution helped or not.

- Units of analysis

  The study will be conducted in cooperation with a company that has over 300 developers who are responsible for building Docker images and deploying them. The company has more than a 100 different deployments a day which makes them a great subject for analysis.

- Theory

  Currently, the company has different tools and solutions to test Docker image vulnerabilities and code inside the image, but no automated way to find if Docker best practices and company policies have been followed. By introducing the solution which does that, the number of issues should be lowered. At first, the number of current issues will be gathered. After the initial gathering process, the findings will be announced to the development teams who are responsible for the service repositories. Found issues should result in educational messages for the developers so that they can learn from them and not make them again. The overall number of issues should decrease by the end of the study.

- Research questions

  Is there an existing solution that can be used in a companies deployment pipeline to automate the validation of Docker images and containers according company policies and best practices provided by Docker .

  What are the best practices provided by Docker for companies that should be checked using the solution?

  Does the solution help to lower the number of existing issues? Issues will be gathered as part of the solution evaluation in chapter six of this thesis.

- Propositions

  Find an existing solution that supports all the needed features that have been set in this thesis or create a new solution.

  Gather data about Docker best practices and company policies for Docker image building and service or container deployment.

  Gather data to find as many existing issues as possible.

  Implement the solution into the companies deployment pipeline.

  Analyze the difference between data gathered at the beginning of the study and in the end of it.

- Define concepts and measures

  Existing issues — found issues by the solution that are not inconsistent with the rules created according to the companies policies and Dockers' best practices.

  Repository with issue — repository that has at least one issue.

Repository issues — identified issues should be mapped to a certain repository for a service, Docker image or container.

Remaining issues — issues that were still found after each step of the study. In the first step remaining issues equals existing issues.

Deployments — the number of deployments done in any step of the study.

When a suitable solution for the stated problem for this thesis is found or created, data about the existing issues will be gathered for 30 days. Since not all Docker services or images are deployed every day or even week then data needs to be gathered for a longer period of time. This will provide the first measurement of existing issues for certain images, services or containers.

Data gathering will continue throughout the whole study to measure the difference of issues in time.

After that every week for 21 days there will be a global announcement for the development teams about the found issues so that they could be solved. The number of issues, number of repositories with issues and remaining issues will be compared to the initially found data.

As the final step the solution will start sending messages directly to the person who started the development of any remaining issues. Data in this step will be compared to the values that were collected in the first and second step of this study. Comparing the data from all step will give the final result.

The number of issues that were gathered for a repository at the beginning of the study, should decrease.

A successful result for the company would be a decrease of 50% for the number of issues.

The total timeframe of the study will be 65 days. 30 days for the initial data gathering, 21 days for global announcements about the issues and 14 days for the direct messages for the developers.

- Methods of data collection

  Data will be collected for every deployment that uses the main deployment pipeline.

  All issues will be mapped for service repositories, so they can be tracked throughout the whole study.

- Methods of data analysis

  Issues related to a Github repository will be analyzed from the beginning of the study to the end of it to examine if they were fixed after the issue was made known to the team responsible for the repository. The decrease of overall issues will be measured to understand if the tool provides good feedback in an understandable way.

- Case selection strategy

  All issues from the initial data gathering done by the solution for the first 30 days of the study will be identified and selected.

- Data selection strategy

  Operations, development operations and information security teams will approve the rules which are then checked by the solution. Data about the issues based on the approved rules found by the solution will be gathered.

- Replication strategy

  This study does not tend to replicate any previous study.

- Quality assurance, validity and reliability

  False positive and the correctness of the gathered data will be check by operations, development operations and information security teams of the company.

1.5.3 Goals of the thesis and expected results

The goal of this thesis is to provide a solution to the raised problem. In short, the solution should provide an easy way to check Docker image building and container or service deployment process against a defined set of rules. Data about the found issues should be stored and educational messages provided to users.

The expected result of the thesis is a solution that helps developers to follow and learn company policies and Dockers' best practices or any other set rules. Doing that the overall number of issues should decrease. The company believes that a 50% decrease in the overall number of issues during the time when the study is conducted means the solution is successful.

2 Docker

In this chapter the author will provide an overview of Docker. What it is, how it came to be, what are the benefits of using Docker and what Docker cannot do. This chapter is meant to provide the knowledge needed to understand the abstract and research of this thesis. Reader who is familiar with Docker can skip this chapter and continue from chapter 3 of this thesis.

Docker is a tool that can be used to encapsulate the process of creating a distributable artifact for any application, deploy it to any environment and scale it. [1, pg 1]

2.1 Birth of Docker

Docker was first introduced to the world by Solomon Hykes, founder, and CEO of dotCloud. It was done in a quick five-minute talk at Python Developer Conference in Santa Clara, California, on March 15, 2013. No pre-announcement was made before the talk and only about 40 people outside dotCloud had been given the opportunity to test Docker. [1, pg 1]

The project was quickly open-sourced and made publicly available on GitHub. More and more people began to hear about Docker and how it was going to revolutionize how software is built, deployed and run. [1, pg 1]

2.2 Docker benefits

When implemented well, Docker can bring benefits to the whole organization, teams, developers and operation engineers in multiple ways. [1, pg 3]

- **Easier software development**: In the past companies used to have different positions for building and releasing software. This was mostly needed to manage all the knowledge and tooling involved. With Docker, everything can be wrapped up into one package and defined by a single file. [1, pg 3]

- **All-in-one approach**: In the past typically applications had to be packaged with different dependencies including libraries and daemons. Environments had to be identical to ensure that the applications will run the same. Docker allows to deploy an application along with every single thing needed to run it. Docker´s image layering also makes this process efficient. [1, pg 4]

- **Easier to test**: Docker images can be built and used to go through the whole testing process. The same image can later be deployed to productions. This reassures that the tested code was deployed. Version control systems, recompiling or repackaging at every step of testing are not needed. [1, pg 4]

- **Smaller resource footprint**: Traditional enterprise virtualization solutions like VMware are typically used when there is a need to create an abstraction layer between the physical hardware and the software application which runs on it. The hypervisors that manage the virtual machines and each kernel that runs the virtual machine use the hardware system's resources. These resources are no longer available for the hosted applications. On the other hand, containers are just another process that communicates to the host machine kernel. Thanks to that container can utilize more resources, up to the system limit. [1, pg 4]

Using Docker can also make architectural decisions simpler because all applications essentially look the same from the outside in a hosting system perspective. It can also improve writing and sharing applications. [1, pg 4]

2.3 What Docker is not?

Docker helps the user to solve many challenges that other tools have traditionally been enlisted to fix. However, the many features provided by Docker often means that it lacks depth in a specific functionality. For example, a configuration management tools for some companies or users can be removed when using Docker. The real power of Docker is that even if it could replace some aspects of a more traditional tool, it is usually compatible with them or even benefits from them. The following list explores some of the tool categories that Docker does not directly replace.

- **Virtualization Platforms**: A container is not a traditional virtual machine. A virtual machine has its own operating system and runs on top of a host. Virtual machine´s advantage comes for the possibility to easily run a totally different operating system on a single host. Containers share the same kernel as the hosts operating system does. Knowing that a container utilizes fewer system resources but must be based on the same underlying operating system. [1, pg 5]

- **Cloud Platform**: Containers and enterprise virtualization workflow may seem to share a lot of similarities with cloud platforms from the outside. Both are able to allow an application to be horizontally scaled to provide a growing demand. However, Docker is not a cloud platform and only be used for deploying, running and managing containers on pre-existing Docker hosts. Docker does not allow a user to create new host systems, storage or any other resource which is typically associated with cloud platforms. [1, pg 5]

- **Configuration management**: Docker can greatly improve the ability to manage applications and their dependencies for an organization, but it does not completely replace more traditional configuration management tools. Dockerfiles are used to specify what a container should look like, what it should have inside at the time it is built, but it does not manage the built containers current state or cannot be used to manage a Docker host system. [1, pg 5]

- **Deployment framework**: Docker can make many aspects of deployment simpler by providing the possibility to create self-contained container images with all the needed dependencies of an application encapsulated in it. These images can be deployed to different environments without any change. However, Docker is not able to automate a complex development pipeline by itself. Other more traditional tools that handle complex deployment processes should be used together with Docker. [1, pg 6]

- **Workflow management**: A Docker host by default does not have any internal concept of a cluster. Additional orchestration tools, like Docker´s own Swarm, must be used. Orchestrators can coordinate work intelligently across a set of Docker hosts. Additionally, they can monitor the resources and states of the hosts and keep track of all the running containers. [1, pg 6]

For more information about Docker images, containers, services, orchestration and their official registry can be found in annex 5 - 9 in this thesis.

3 Docker best practices

This chapter will provide an overview of the best practices provided by Docker and the community for building Docker images and deploying containers. The author chose to review best practices since Docker documentation[1] lacks in security guidelines. CIS Docker 1.13.0 Benchmark v1.0.0 [35] and Docker Security [37] by Adrian Mouat will be largely used. All rules were reviewed and validated by the author. The author also coordinated the rules with Docker Captain [39] Renno Reinurm to be convinced which of them should be followed. Additional information will be provided by the author to the rules if he finds it necessary. Additional information will be gathered from different sources.

3.1 Building Images

- Create a user for the container [35]
    - Rationale:

      Running a container as a non-root user is considered as a good practice. User namespace mapping allows the container to run with the defined user by default without any need for remapping. By default, the user is set to root. [35]

    - Remediation:

      Dockerfile for the container image should include "`USER` <username or ID>". Username or ID should be found in the container base image. [35]

[1] https://docs.docker.com/engine/security/security/

- Addition information:

  To lower the risk in case of a "container breakout" container privileges should be reduced as much as possible [36].

  If a user on the host machine uses the kernel keyring for handling cryptographical keys then a container running with the same UID of that user also has access to these keys. [37]

  Root privileges also allows to change anything inside a running container. Break the service or change it to do something unintentional.

- Use trusted base images for containers [35]

  - Rationale:

    Public repositories could potentially be unsafe. The images in official repositories in Docker hub are curated and optimized by Docker or the vendors. Extreme caution should be exercised when using an unofficial image. [35]

  - Remediation:

    Only use official images approved by Docker. [35]

  - Addition information:

    Docker Hub was explained more in the Docker chapter of this thesis. But in short Docker Hub is a registry service provided by Docker to which anyone can push their build images for storing and sharing with others. Since anyone around the world can push their images to Docker Hub there is no easy way to be sure if the image is safe to use. It might have some vulnerabilities or even back-doors in it. Fei Huang from Neuvector [32], a company that is one of the

leaders in container network security, wrote an article called *"17 Backdoored Malicious Images Removed From Docker Hub, But Are You Really Any Safer?*" in early 2019. This article talks about 17 different images in Docker Hub which had backdoors in them. These backdoors were mainly hidden inside MySQL and Tomcat images. [33]

Docker Hub official images are curated by Docker. A dedicated team that collaborates with upstream software maintainers, security experts, and Docker community is sponsored by Docker Inc to review, publish and be responsible for all the official images in Docker Hub. [34]

- Add healthcheck instruction to the container image [35]
    - Rationale:

      Availability is an import security triad. Health Checks ensure that the Docker engine will checks periodically if the running container instance is still working according to the give instructions. The Docker engine could exit a non-working container and spawn a new one. [35]

    - Remediation:

      Docker documentation should be followed to add health checks for containers. [35]

    - Addition information:

      Health checks and be added from Dockerfile, docker-compose file or `docker run` command. Having health checks in Dockerfile will include them into the image and they will be executed for every container built from that image. Health checks in docker-compose will be added as part of the service and will be executed for every container that is part of that service. Adding health

checks in the `docker run` command will apply them for the container created with the command. [36]

- Do not use update instructions alone in the Dockerfile [35]
    - Rationale:

      Single line update instructions in the Dockerfile will cache the update layer. This means that any new image with update instructions will use the previously cached update layer. Potentially it can deny any new updates for the new images. [35]

    - Remediation:

      Update and install instructions should be used together. Package versions should be pinned and docker build should be executed with `--no-cache flag`. [35]

- Remove setuid and setgid permissions in the image [35]
    - Rationale:

      Setuid and setgid permissions should be dropped for packages that do not need them within the image. These permissions could be potentially used for elevating privileges. [35]

    - Remediation:

      Setuid and setgid permissions should only be allowed for executables with actually need them. These permissions can be removed during the image build by modifying dockerfile. [35]

- Use COPY instead of ADD in Dockerfile [35]
    - Rationale:

      While `COPY` command in dockerfile just copies the files from the host machine to the container then `ADD` instruction can also be used to retrieve files from remote URLs and perform other operations like unpacking. Therefore `ADD` can potentially be used to add and unpack malicious files from URLs without any scan. [35]

    - Remediation: Use the `COPY` command instead of `ADD` in dockerfile. [35]
- Do not store secrets in Dockerfile [35]
    - Rationale:

      Docker history command together with some tools or utilities can easily be used to backtrack Dockerfiles. For this reason, any secret inside the Dockerfile could be exposed and potentially exploited. [35]

    - Remediation: No secret should be stored inside the Dockerfile. [35]
- Install verified packages only [35]
    - Rationale:

      Packages with unverified authenticity can potentially be malicious or contain vulnerabilities that could be exploited. Hench only authenticated packages should be used when building an image. [35]

Remediation:

GPG keys or any other secure package distribution mechanism should be used to download and verify packages. [35]

3.2 Deploying containers

- Limit container networking [37]
    - Rationale:

      A container should be reachable from as few places as possible to protect the service inside and the traffic from it as much as possible. [37]

    - Remediation:

      Only needed ports for that service should be opened. These ports should only be reachable from a certain network. [37]

- Do not use Docker's default bridge docker0: [35]
    - Rationale:

      Virtual interfaces created in bridge mode are connected to Docker's common network bridge docker0. That bridge is vulnerable to ARP spoofing and MAC flooding attacks due to not having any filtering. [35]

    - Remediation:

      A user-defined network should be used instead of docker0. [35]

- Do not map privileged ports within containers [35]
    - Rationale:

      TCP/IP ports below 1024 are considered essentially privileged. These ports may be used to transfer sensitive and privileged data. Serious implications may follow when allowing a container to use them. [35]

    - Remediation:

      Privileged host ports should not be mapped to a container. [35]

- Limit memory [37]
    - Rational:

      Container with no memory limit is more vulnerable against DoS attacks or application side memory leaks. Container with no memory limit can consume all the host machine memory and therefore bring down the whole host. [37]

    - Remediation:

      Limit the maximum memory a container can use. [37]

    - Addition information:

      Bugs in software, malware or miscalculations in software design can easily can a Denial of Service inside the container if resource limits are not defined.

- Limit CPU [37]
    - Rationale:

      Similar to no memory limit, a container with no CPU limit can starve the host machine and all other containers on that host. This can be a result of an attack or faulty application code. [37]

    - Remediation:

      Limit the maximum CPU a container can use. [37]

    - Addition information:

      Bugs in software, malware or miscalculations in software design can easily cause a Denial of Service if resource limits are not defined.

- Limit restart [37]
    - Rationale:

      Constantly restarting or crashing container wastes a large amount of system time and resources, which can possibly extend to a Denial of Service. [37]

    - Remediation:

      Limit the number of restarts for a container. [37]

- Limit file systems [37]
    - Rationale:

      Preventing the application inside a container to write to files can prevent several attacks and protect the host machine and other containers on that host. [37]

    - Remediation:

      Run containers in read-only mode. [37]

- Do not mount sensitive host system directories on containers [37]
    - Rationale:

      Mounting sensitive directories such as /etc, /sys, /usr or /boot on systems using Linux kernel to a container in read-write mode means these files on the host machine can be changed from the container side. These changes can potentially bring down the entire host or compromise its state. [37]

    - Remediation: Sensitive directories should not be mounted to a container if possible. Especially in read-write mode. [37]

- Do not run ssh within containers [35]
    - Rationale:

      Containers should not be used as SSH servers. It complicates the access policy, key, password and security update management. [37]

    - Remediation:

      SSH server should be uninstalled from containers if not used. [37]

- Do not share the host's process namespace [35]
    - Rationale:

      When container shares the PID namespace with the host then the container can see all the processes running on the host system. This removed the processes isolation between the container and the host system and can potentially lead to numerous problems. [35]

    - Remediation:

      A container should not run with host PID mode. [35]

- Confirm cgroup usage [37]
    - Rationale:

      By default containers run under docker cgroup or a cgroup set by a system administrator. Attaching a container to a different cgroup might lead to granting the container other permissions and resources, which can prove to be unsafe. [35]

    - Remediation:

      Cgroup for containers should not be changed unless explicitly needed. [35]

- Use PIDs cgroup limit [35]
    - Rationale:

      PIDs cgroup limit may prevent a fork bomb attack by restricting the number of forks that can occur in a container at a given time. A fork bomb can potentially crash the entire host. [35]

- Remediation:

  An appropriate value for the PID limit should be set for containers. [35]

- Do not share the host's IPC namespace [35]
  - Rationale: When a container shares the ICP namespace with the host then the container can see all the ICP processes running on the host system. This removed the ICP processes isolation between the container and the host system and can potentially lead to numerous problems. [35]
  - Remediation: Container should not run with host IPC mode. [35]
- Do not share the host's UTS namespace [37]
  - Rationale:

    Sharing the user namespace of inside the container and outside removes the user isolation between the container and the host system. Therefore can potentially be unsafe. [37]

  - Remediation:

    Do not share the same user between the host system and container. [37]

- Do not share the host's user namespace [35]
  - Rationale:

    Limit the number of capabilities or sets of privileges give to a container as much as possible to protect the host machine from attacks or faulty applications. [35]

- Remediation:

  Unnecessary capabilities should not be added a container. [35]

- Do not directly expose host devices to containers [35]
  - Rationale:

    When a host device is exposed to a container by default container has read, write and mknod permissions. For instance, block devices can be removed from a host by a container. Thus, host devices should not be exposed to containers. [35]

  - Remediation:

    Host devices should not be exposed to containers. If it is needed then correct permissions must be used. [35]

- Do not override default ulimit at runtime [35]
  - Rationale:

    The default ulimit set by Docker daemon should be honored. Limiting the open file descriptors per process can help with many issues like fork bombs. [35]

  - Remediation:

    Default ulimit should not be overridden and if needed then done with precaution. [35]

- Do not disable default seccomp profile [35]
    - Rationale:

      Docker seccomp profiles are enabled and should not be turned off. By reducing the number of systems calls the total kernel surface is not as exposed to the container and due to that improves security. [35]

    - Remediation:

      Docker seccomp profiles should not be disabled unless a modified seccomp profile is being used. [35]

- Do not mount the Docker socket inside any container [35]
    - Rationale:

      Mounting the Docker socket inside a container allows the container to execute docker commands which can lead to having full control of the host. [35]

    - Remediation:

      Docker socket should not be mounted to any container if possible. [35]

4 Company

This article will give more information about the company on whose needs this thesis is based. What are the different steps in the deployment pipeline and what tools are used? What different policies and rules should be followed when Docker images are created or containers deployed. The company will also be used to test the proposed solutions.

4.1 Overview

The company has around 300 developers. Jenkins is used as the main tool for building, testing and deploying. Jenkins will be covered in more detail in the Jenkins section in the same paragraph.

Docker is used as the main service platform in live and test environments. Around 100 different deployments are made to live environments daily. Each deployment must have a Github repository with all the needed files and configurations for the whole deployment pipeline and a team that is responsible for the repository.

The company has different rules and policies about building Docker images and deploying Docker containers or Docker services that use them.

The company already has different tools in place to check developed code and created Docker images and containers for known vulnerabilities. As stated before in the Introduction chapter then Docker image and container vulnerability scanning are out of the scope for this thesis.

4.2 Rules and policies

The company has different rules and policies for building Docker images, deploying Docker containers and services. Most of the rules come from Docker best practices and some are our company specific. This paragraph will go over the company-specific rules and policies which are not looked at in the Docker best practices chapter. Since the company's internal rules and policies are not available for the public then the author does not have anything to reference. The author will provide additional information or links to documentation if he finds that it is needed.

4.2.1 Building Images

- Only official or company base images from Docker hub
    - Rationale:

      The company has a private Docker hub repository that houses all the images built, checked, scanned and pushed by the employees. Docker provides official images that curated by them to ensure the security and quality of the image.

    - Remediation:

      Only official or company images must be used as base images.

- Do not use `apk update`
    - Rationale:

      APK update should not be used to keep the images as lightweight as possible and to not pull unwanted packages.

- Remediation:

  `apk add --no-cache` should be used instead.

- Additional information:

  Larger images increased image building, scanning and deployment time.

- Do not use `apk add` without `--no-cache`
  - Rationale:

    `apk add` command is only allowed to be used with `--no-cache` flag to keep the images as lightweight as possible and to not pull unwanted packages.

  - Remediation:

    `apk add --no-cache` should be used instead.

  - Additional information:

    Larger images increased image building, scanning and deployment time.

4.2.2 Deploying containers

- Only official or company images from Docker Hub
  - Rationale:

    The company has a private Docker Hub repository that houses all the images built, checked, scanned and pushed by the employees. Docker provides official images that curated by them to ensure the security and quality of the image.

- Remediation:

  Only official images or company images may be used when deploying a container.

- Do not use static port mapping
  - Rationale:

    Containers are not allowed to map any ports. Only dynamic port mapping from the Docker daemon side is allowed. This ensures that no privileged ports may be used by a container.

  - Remediation:

    Static port mapping should not be used.

  - Additional information:

    When not talking about security, static ports should not be used due to the fact that the chosen port may be already in use on Docker host, which means the container cannot start there.

- Docker compose version higher than 3.2
  - Rationale:

    Companies deployment pipeline and development operations team supports docker-compose versions starting from 3.2

  - Remediation:

    The version in the docker-compose file must be set to 3.2 or higher.

- Additional information:

  Information about different Docker compose versions and their differences can be found from Docker documentation[2].

- Use the correct network for service
  - Rationale:

    Each container should be connected only to the correct network for that service.

  - Remediation:

    The correct network for a service should be used.

  - Additional information:

    Different types of services should be connected to different Docker networks with specific rules set by the network administrators.

- Container health check interval and timeout should not be lower than 30 seconds
  - Rationale:

    Each service should have a health check to ensure that it is working correctly. Running hundreds of health checks simultaneously on the same Docker host creates a large amount of load. That can lead to Docker daemon not answering in time resulting in a faulty failing health check. This can potentially lead to the container being terminated by the daemon.

[2] https://docs.docker.com/compose/compose-file/compose-versioning

- Remediation:

  Container health check interval and timeout should not be lower than 30 seconds.

- Additional information:

  Failing health checks when there is a large number of concurrent health checks running is a bug [3]in Docker daemon.

- Container health check retries should not be lower than three
  - Rationale:

    Services or containers should have at least three health check retries to be sure that the container could not heal itself and it should be restarted.

  - Remediation:

    Container health check retries should be three or higher.

  - Additional information:

    Failing health checks when there is a large number of concurrent health checks running is a bug in Docker daemon.

- Correct Docker host constraint should be used
  - Rationale:

    Each container should be placed on the correct Docker host to ensure the correct configuration and placement. Normal application containers should also not be deployed to Docker swarm managers. This lowers the potential risk

[3] https://github.com/moby/moby/issues/33933

of a faulty container bringing down a manager which can lead to many different problems throughout the whole swarm

- Remediation:

  Use the correct constraint for the container.

- Additional information:

  More information about constraints and labels can be found from a Docker article [4]about labels and constraints.

- Certificates should be mounted in read-only mode

  - Rationale:

    Particular services need to mount certificates to establish secure connections to other tools or services. Certificates mounted to a service should be in read-only mode. This removes the potential risk of a container changing or removing a certificate that applies to all other containers using that certificate.

  - Remediation:

    Certificate mounts should be in read-only mode.

- Reserve correct amount of CPU and memory for the container.

  - Rational:

    Each service or container should reserve the needed amount of CPU and memory. Reserved resources can only be used by the container that reserved them. Reserving too much means not all possible resources are available for

[4] https://success.docker.com/article/using-contraints-and-labels-to-control-the-placement-of-containers

use. Reserving too few might lead to Docker host not having enough resources to fully run all the containers on it.

- Remediation:

  Use correct reservations for each container or service.

- Additional information:

  The correct amount of resources is fetched from the company's internal tool that monitors container and host resource usage.

- Correct UCP collection

  - Rationale:

    Each team should only have access to the services and containers they are responsible for. This removes the risk of somebody by mistake changing configurations for a wrong container or service. Access can be managed by Dockers' Universal Control Plane or UCP shortly.

  - Remediation:

    Use the correct UCP collection for a service or container.

  - Additional information:

    UCP that is the enterprise-grade cluster management solution provided by Docker has its own built-in authentication mechanism. UCP also provides role-based access control which helps to control who can access and make changes to the cluster and applications. [28]

- Use update config for service
    - Rationale:

      Update config can be used to defines how a service should be updated.

    - Remediation:

      Use update config in docker-compose.

    - Additional information:

      Update config allows configuring how many containers can be updated at once or if an existing container should be terminated before a new one is started or not. It also allows specifying what the delay should be and if the new containers should be monitored by Docker or not. Hence, making update config very useful for rolling updates. [39]

- Use restart policy
    - Rationale:

      Restart policies can be used to control what should be done after a container is terminated or crashes.

    - Remediation:

      Use restart policies.

    - Additional information:

      Restart policy provides the possibility to specify when the container should be restarted, what is the delay, the number of attempts and how long to wait before the new container should be running. [39]

- Use stop grace period for service
    - Rationale:

      Stop grace period for service can be used to define the time how long Docker should wait until terminating the service. Some containers that handle bigger requests or serve customers need more time to shut down gracefully.

    - Remediation:

      Use the correct stop grace period for the service.

- Do not use service name longer than 22 characters
    - Rationale:

      A service name cannot be longer than 22 characters. Docker allows a maximum of 63 characters in the fully qualified domain name (FQDN). Due to company naming policy service name must not exceed 22 characters.

    - Remediation:

      A service name should not exceed 22 characters.

- Image and repository name should match
    - Rationale:

      Docker image and Github repository name should match. This is needed in order to ensure that different tooling works correctly.

    - Remediation:

      The same name should be used for image and Github repository.

- Jenkins variable exists.
    - Rationale:

      Jenkins is used as the CI/CD tool by the company. Jenkins takes care of much different automation. Different Docker related files or commands should include needed variables set by Jenkins.

    - Remediation:

      Include Jenkins variable in Docker commands and files.

    - Additional information:

      As an example, Jenkins' deployment pipeline builds a new image and tags it based on build number and Github branch hash. To ensure that the newly built image is tested and deployed the compose file or `docker run` command must have the variable `DOCKER_TAG` which is set by Jenkins.

- Correct user added to the container.
    - Rationale:

      Each service or a container should have an owner who can be contacted if an issue arises.

    - Remediation:

      The owner should be added to the service or container

    - Additional information:

      As an example, Jenkins' deployment pipeline builds a new image and tags it based on build number and Github branch hash. To ensure that the newly built

image is tested and deployed the compose file or `docker run` command must have the variable `DOCKER_TAG` which is set by Jenkins.

4.3 Company deployment pipeline

The whole deployment pipeline consists of many different mostly automated steps which fetch the code, test the code, build Docker images, deploy container, test them and deploy them to a production environment. The overview of the whole pipeline can be found in annex 10. This paragraph can be skipped if the reader is not interested in the company's pipeline.

4.3.1 Github

Github is a web-based hosting service that provides developers easy versioning and source code management for the code which is being developed. [40]

In the company, every service has its own repository. Developers can create branches, commit new code or fetch the existing one. Created branches can be automatically tested and deployed to needed regions by adding a label to the pull request. Branches are by default automatically merged into the master branch when the deployment of this branch was successful.

4.3.2 Jenkins

Jenkins is an open-source continuous integration and delivery or CI/CD tool in short written in Java. Jenkins is mostly used to build, test and deploy software or code continuously. This makes it easier for developers to make changes to existing projects and deploy them to production environments. Jenkins has many plugins that make the whole process even easier or allow the users to execute different stages exactly how they choose. Jenkins allows organizations to accelerate the software development process through automation. [21]

Jenkins uses master and agent hosts to provide scaling, parallelism and distributed builds. Jenkins master can operate alone while managing different builds and executing them by using the resources given to itself. Using a standalone Jenkins master will most likely lead to a lack of resources when the number of projects increases. Jenkins agents are host machines used to offload projects from the master. Jenkins master will manage and schedule build to agent based on agent labels. [22]

5. Proposed solution

The author of this thesis decided to create a solution that presents all the needed features. The author did not identify a solution that satisfies all the company needs and could be used to automate the validation of Docker image creation and container deployment according to company's policies and best practices provided by Docker. None of the tools that were looked at and analyzed provided a full toolset. Some of them could have been modified to provide all the different features that are needed. The author decided to create a new tool due to the fact that changing an already provided solution would most likely take the same amount if not more time as creating a brand new one.

5.1 Overview

Deployment or repository analyzer also known as Dora is a framework created as part of this theses by the author. Dora allows the user to add new rules that will be used to check different type of data. At the moment Dora is able to check a directories content and files or data in YAML, JSON or plain text format. Dora can also fetch data from an external API endpoint. Data is validated by using regular expression search patterns. The report will be put together after all the rules are checked and the result can be presented to the user in standard output or sent via Slack. The report can also be sent to external storage or database for further analyses. For the time being Dora supports Amazon S3 as external storage and MySQL as a database.

Dora is written in Node.js and contains 2585 lines of code. Dora took approximately 12 weeks to develop. Configuration for each rule must be in the JSON format and housed in one or multiple configuration files. A rule file with the needed functionality is used for every rule to validate the data. Each rule can have its own rule file or use the default rule file.

Dora can run as a Docker container or from source code making it easy to run in different environments.

5.2 Architecture of Dora

As stated in the Overview paragraph of this chapter , Dora is written in Node.js.

Dora consists of three different types of files - rule, configuration, and utility. The content of these files will be previewed in the following paragraphs.

5.2.1 Configuration file

All configuration files must be written in JSON format and be placed inside the configs directory in Dora root directory. These files can be named freely but must contain a valid JSON format.

The author encourages to name the files by something meaningful like the rule type. As an example configuration for rules to check docker-compose file could be named docker-compose.json or docker-service.json.

One configuration file can host one or more rule configurations each being a JSON object. Each rule configuration attribute should have the same name as the rule file for that rule. If the rule file cannot be found a default rule file will be used.

Rule files will be discussed more in the next paragraph but as an example, if a rule configuration attribute is named memory_limit then the rule file name must be memory_limit.js.

Different parameters can be set for each rule. Available parameters are the following:

- category - string

  Used to set a category for a rule. Categories are only used for grouping finding in the report. Due to that, this parameter is not mandatory and the default value of Uncategorised will be set if missing.

- file - path to a file

  Used to set the path of a file which content should be checked. This parameter is not mandatory since data could also be gathered from url or dir. File, url, and dir can be set together.

- url - URL to an external endpoint

  URL of an external site or service from which data should be requested. The parameter is not mandatory and can be used together with dir and path parameters.

- dir - path to a directory

  The path to a directory inside the checked repository. If the root directory is being checked then the dir value should be `./` or left empty. The parameter is not mandatory and can be used in conjunction with url or file parameter.

File, url or dir parameters by themselves are not mandatory but one of them should be defined if any data should be gathered.

- active - boolean

  Provides the possibility to turn a rule on or off. This can be used to turn a specific rule off for a specific repository or to deactivate a rule altogether. Each rule by default is active.

- override - boolean

  Allows defining if the rule can be changed or turned off from any repository. Provides the owner of the tool to define rules that can be changed for certain repositories and rules that cannot. Override for each rule is allowed by default.

- slack - boolean

  Provides the possibility to turn direct messaging on or off. The rule will still be reported into the standard output or sent to external storage. This allows the owner to test a rule or run it in the background. Sending direct messages via slack is turned on by default.

- score - integer

  Used to give a rule a score. A score threshold can be set so that a deployment can be stopped if the score reaches the threshold. The score for each rule will be considered as 0 if the parameter is missing.

- docs.name - string

  Allows defining a name for a rule. This name will be used in reporting. If the name parameter is missing “Unnamed” will be used.

- docs.description - string

  Use to set a description of the rule. Description will be used in reporting. “Description not found” will be used if the parameter is not set.

- docs.url

  Provides the possibility to add a URL to external documentation. Defined URL will be added to the report. The default value for this parameter is “No docs found”.

- param - string

  Can be used in two different ways. If the data that is being validated is in plain text the value will be used to find a line that contains it. If the data is in YAML or JSON format then the value will be considered as the path inside the data object. No default value for this parameter is set.

- value - string, integer or boolean

  A list of values that should be compared to the data found using param.

5.2.2 Rule file

Rule file should contain the needed logic and functionality for a rule. Rule name must match the attribute name of the rule that should use it. If the rule file for a rule is not found a default rule file will be used.

The default rule file can be used to check if a line or the object path defined in the rule configuration was found or not. The found data will be compared to all the values specified in rule configuration value attribute. The rule configuration determines if a report should be sent if any data or a specific value in that data was found or should it be sent they were not.

Default rule file is meant for very easy checks where the user wants to make sure a specific parameter or value is set or not.

Each rule that needs more logic behind it must have its own rule file.

5.2.3 Framework utility files

Framework utility files house all the logic, configuration and functionality that are needed for the whole tool to work. Including numeros functionality to transfer data between different

functions and setting correct parameters. The utilities handle data gathering, parsing, analyzing and reporting the results.

- Parser

  The main purpose is to make data understandable to other functions. It has functions to parse data in JSON,YAML or plain text format. Parser tries to understand what kind of data it has been given based on data type and acts accordingly.

- Searcher

  Contains all the functionality needed to find the correct data based on given param and value from the rule configuration. It also takes care of finding all the rule configurations and rule files. Searcher has also the functionality to convert seconds and minutes into hours and bytes and gigabytes into megabytes. This functionality is needed to find and return the correct value.

- Validator

  Houses core functionality and logic for the whole tool. Validator checks if rule configuration is valid and only executes the ones that have not been disabled.

- Reporter

  Takes care of putting together the report, keeping score and publishing it to standard output. It also passes the report data to slack and external utility files.

- Slack

  Has all the functionality to format the report data for slack and sending it. It also checks if user or channel name and slack token have been given in the correct format.

- External

  Takes care of fetching data from external API endpoints if a rule needs it. It also has the functionality to push report data to Mysql and S3.

- Configuration

  Does not contain much functionality but stores Dora general configuration that can be used by any other utility file.

- Initialiser

  Initialises Dora configuration, starts and ends the whole process.

5.2.4 Dora's general logical flow

As the first step after initialization Dora will read and store all the rule configurations. These configurations hold the information where Dora should get the data that should be checked and how to check it. After all the configurations have been loaded Dora will start to check each rule. Dora will check if the rule has its own special rule file or should the default rule file be used. Data will be fetched from the given source that can be a file, directory structure or external data source for every rule that was found in the configuration. Dora will verify the data type and parse it accordingly. When the data is understandable then Dora will look for the given parameter in the configuration. Depending on the configuration a report will be created if the parameter is not found from the data. If the parameter is found and value exists for that rule then it will be validated next. The configuration will determine if the value one or multiple values must exist or should it exist at all. A report will be created if discord is found. When all the rules have been checked, the created reports will be written into standard output, sent to slack or stored externally based on Dora's configuration. The general flow can be seen on Figure 1.

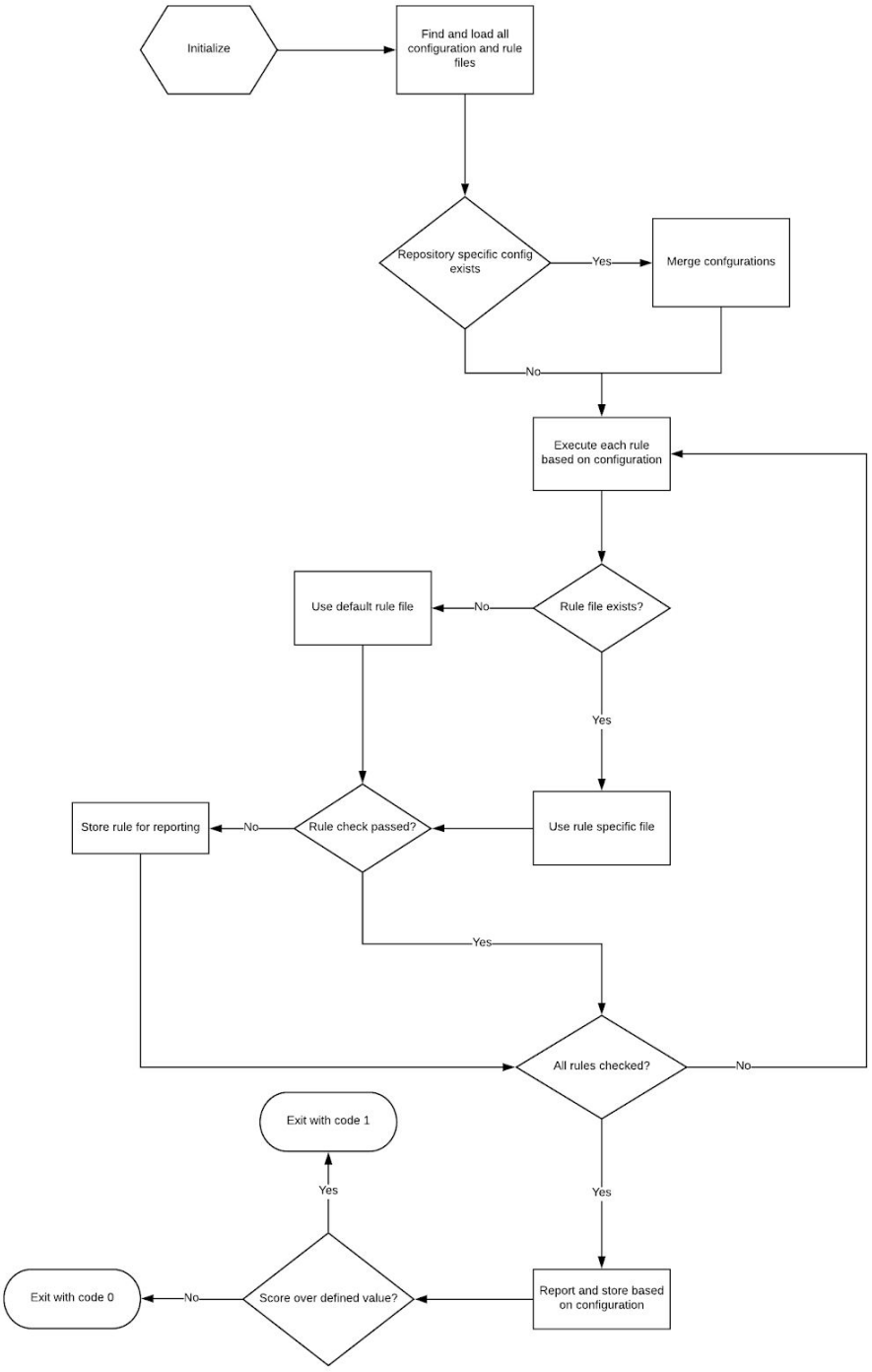

Figure 1 Dora general logical flow

5.3 Solution requirements

Many different requirements were set to favor a solution. Requirements were put together with the company that is in need of the solution and with whom it will be tested with. All the requirements were important to the company to have a full solution for the problem. Each requirement will be reviewed later in this paragraph. Since no suitable solution was found then a proposed tool was created by the author of this thesis. Just as all the reviewed tools so must the created tool cover all the requirements.

This paragraph will talk about all the set requirements and how the created tool covers them. In most cases, regular expressions [41] format is used to describing a search pattern for validating if the looked for parameter and value exist or not. Each rule should have a defined destination from where the data should be collected. It can be a path to a file, a directory or a URL to an external service or site.

- Validate Docker image build

  Dora is able to read a Dockerfile on a given path in the rule configuration file. Every command, including multiline commands, will be taken as a new line. Since Dockerfile is a set of commands written in plain text then the most feasible way to find if a command or flag exists, variables and parameters are set is to look for the wanted word, number of characters in the line.

  A good example is the company's policy "Do not use apk add without --no-cache". In this rule configuration param and value are defined as following

  ```
  "param": "apk.* add ",

  "value": ["--no-cache"]
  ```

Meaning a line that contains apk add must also include `--no-cache`. It should be noted that the .* between apk and add can be replaced with any characters unlimited times. Since different flags can be passed to apk add command then this will ensure that the correct line is found.

- Validate service or container deployment

  Docker container or service can be deployed by using Docker CLI [42]. `docker run` or `docker service create` commands must be used respectively. In case of a plain text the regular expression search pattern the rule configuration should be:

  ```
  "param": "docker run",

  "value": ["--ulimit"]
  ```

  When the file is in JSON or YAML format then the path on where the command or value should be searched for is needed. Each object key or YAML item on the path must be separated with a dot. As an example container deployment is defined in a JSON or YAML configuration file.

  YAML:

  ```
  deployment:

      container:

              - docker service create --name busybox-test
                --ulimit 100 busybox
  ```

JSON:

```
{
      deployment:{
            container: docker service create --name
            busybox-test --ulimit 100 busybox
      }
}
```

Then the rule configuration should be the following:

```
"param": "deployment.service",
"value": ["docker service create.*--ulimit.*"]
```

Docker compose file can be used to define how containers should behave. Docker compose file must be written in YAML format as specified by Docker [43]. The rule configuration should be the following when using a docker compose file:

```
"param": "service.*.ulimits"
```

- Easily modifiable rules for different type of images or services

  In Dora configuration, each rule can be allowed to be overridden or not. When the rule is allowed to be changed then a .dora.json configuration file must be added to the repository with the rule configuration. This allows changing any parameter for a rule or disable the rule altogether.

- Part of the deployment pipeline

  Dora can be added to the deployment pipeline as a Docker container or a source code. Deployment pipeline should define the path where Dora can find the repository to validate.

- Reporting issues

  Dora can write the report to standard output, send it directly via Slack or export it to an external database or storage.

- Easily deployable

  Dora can fully run inside a Docker container. The checked repository must be mounted to Dora's container and the path defined.

- Stopping deployments

  The normal exit code for Dora is 0. When a defined score threshold is reached Dora will exit with code 1. The logic of the deployment pipeline could catch the exit code and act accordingly.

- Gathering data for statistical purposes

  Dora is able to send report data to Mysql database or Amazon S3 storage.

- Mapping issue to repo

  Each found issue can be mapped to the repository from which it was found. This is can be done by passing the environment variable `REPO_NAME` to Dora.

5.4 Dora features

Dora has different features that help to validate the rules, produce reports or parse data. In this paragraph the author will introduce them.

- YAML parsing

  Dora is able to parse data in YAML format. This is mostly needed for docker-compose files that are used to deploy Docker services. YAML data format can also be used to define configurations for different tools that may include Docker command that the user wants to check with Dora.

- Wildcard YAML path parsing

  One docker-compose file can hold the deployment configuration for multiple services. Each service has its own name inside the object path. Meaning that if service is called myapp the path where Dora should look for the service memory limit would be `services.myapp.deploy.resources.limits.memory`. This means that different rule and configuration should be created for each service. Dora is able to check each service individually and create reports for each service. This is accomplished by using asterisk (*) in the path. Dora will replace the asterisk symbol with each value on the same level. For example if the user wants to check if memory limit is set for every service inside docker-compose the path inside the rule configurations should be:

  ```
  "param": "services.*.deploy.resources.limits.memory"
  ```

  If the docker-compose has configuration for 2 service myapp and myapp2 then Dora will check them both and create a report about the correct service if a issue is found.

- JSON parsing

  Dora is able to understand data in JSON format. Usually external API endpoints return data in JSON format. JSON format is frequently used to define configuration for different tools. These configurations can contain Docker command that Dora should check.

- Plain text parsing

  Dockerfile that describes how an image should be build is written in plain text. Dora is able to read and understand data written in plain text.

- Reading and checking multiline commands

  Dora is able to read multiline command from the plain text. Dockerfile allows the user split long commands to multiple lines using `/`. As an example the user wants to check if `--no-cache` flag is used with `apk add command`. `--no-cache` flag can be on another line therefore Dora needs to take a multiline command as one to correctly check it.

- Fetching data for external API endpoints

  Dora is able to gather data from external API endpoints not only read information from JSON,YAML and plain text format. This can be used for example to gather memory or cpu usage from a monitoring system. This data can later be used to validate if docker containers have reserved enough memory or not.

- Value converters

  Dora is able to compare values in seconds, minutes, hours and bytes, megabytes or gigabytes. For example if container healthcheck interval in docker-compose is set to

`1m` and the rule configuration defines that healthcheck must be at least `60s`. Dora is able to compare minutes to seconds compare the values correctly.

- Regular expressions for checking values

  Regular expressions can be used to define search parameters in rule configuration. This allows the user to define very specific search patterns. For example if the user want to check if a line in Dockerfile starts with CMD and is followed by an array then the configuration would be:

  ```
  "param": "^CMD"

  "value": "\\[.*\\]"
  ```

- Default rule file

  Default rule file is always used if a special rule file for a rule can not be found. The default rule is meant to be used when the user wants to check if a value exists or not. If a value in a line should not exist then in the rule configuration a exclamation mark (!) must be placed before the value. This can also be used for parameters or objects paths. As an example if a user want to check if none of the services in docker-compose have healthcheck retries defined then the parameter in configuration would be `"param": !services.*.healthcheck.retries`

- Special rule file

  Each rule used by Dora can have its own special rule file. If rule needs more functionality then the default rule file provides a special rule file should be added to the rules directory in Dora root directory. This rule file must be written in Node.js and must have the name of the rule for which it is meant for. Dora has an example special rule file that already has all the needed functions for data gathering and reporting. The

user can write needed functionality or logic into that file and it will be used for that rule.

- Standard output reporting

  Dora can write the report to standard output that can be seen in Figure 2. The report contains the name of the rule, description, where the violation was found, what was the found value, what is the expected value and link to the documentation.

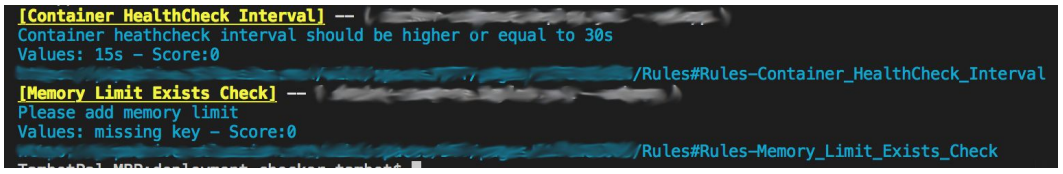

Figure 2 Report in standard output

- Slack messages

  Dora is also able to send Slack messages to a user or channel that can be seen on Figure 3. Slack messages will only be set if slack token and user or channel have been specified. Slack message contains the same information as the standard output message. The only difference is that link to the documentation is provided with a button.

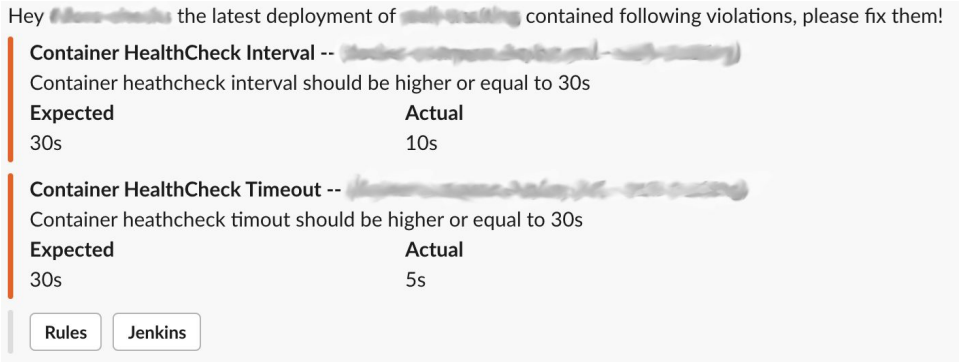

Figure 3 Report in slack

- Storing reports in external storages

  Dora is able to store reports in MySQL database or send them to Amazon S3. For the time being MySQL server address, Amazon S3 bucket name and credentials can be stored in `storage_config.json` fail in Dora root directory or passed as environment variables. These environment variable are `MYSQL_ADDR`, `MYSQL_USER`, `MYSQL_PASS`, `S3_BUCKET`, `S3_USER`, `S3_SECRET`.

- Changing report based on values

  Dora allows the user to use other rule configuration parameters in rule descriptions or documentation url. This help the user to generate more dynamic descriptions and links to the documentation for each rule. For example the user can define description and documentation url as following:

  ```
  "description": "Please use docker-compose version {value}
  or higher"
  "url":
  "https://my.dora.docs.net/wiki/pages/Rules#-{name}"
  ```

  `{value}` will be changed to the value and `{name}` to the rule name specified in the rule configuration.

- Reporting errors

  Dora will report any kind of error in the report. This allows the user to debug any failures more easily. Dora process itself should never crash. All errors and exceptions should be catched and attached to the report.

- Reporting the correct line or service

  Dora will provide the service name or text line from where an issues was found. This helps to find the correct line from a long Dockerfile or the service in docker-compose file.

- Scoring

  Each rule can have a defined score. Rule scores will be summed together and if the value exceeds the score threshold then Dora will exit with code 1. This provides the possibility to cancel a deployment when using Dora as part of a CI/CD tool.

- Multiple values to check

  Dora allows the user to define multiple values for each rule in the configuration. This means that the user can check if multiple or at least one of the values exist. For an example if the user wants to check that each Docker image uses USER guest or nobody then the configuration would be as following:

  ```
  "param": "^USER"

  "value": ["nobody|guest"]
  ```

  If the user want to check that all different values exist for service update_config in docker-compose then the configuration would be the following:

  ```
  "param": "services.*.deploy.update_config"

  "value": ["parallelism", "delay", "monitor"]
  ```

- Dora run options

  `--path={pathToRepo}` path to the repo which should be checked

  `--report` writes report to output

  `--logLevel` specifies the console logging level options ['compact', 'detailed', 'summary', 'all', 'failed'] , defaults to 'failed', 'detailed'

`--slack-user={channel}` sends report to slack channel or to user

`--slack-token={slack token}` specifies the slack token

`--score={score threshold}` activates scoring and exits Dora with code 1 when the threshold is met.

- Environment variables

  Dora is able to also read all the run options from environment variables. The environment variable are the following:

  `PATH, REPORT, LOGLEVEL, SLACK_USER, SLACK_TOKEN, SCORE`

6. Proposed solution evaluation

This chapter provides an overview of the evaluation process of the proposed solution. Data collection methods and principles, calculations and outcome of the evaluation. Since the company's internal data was used for the analysis then the author of this thesis cannot publish any of the actual content. Only the results of different calculations and measurements will be shown.

6.1 Data collection and measures

In this paragraph, the author will explain how and what data was collected.

The company has different tools that track each deployment. Every deployment is written into a Mysql database. In this database, each deployment has data like repository name, the timestamp of deployment start and end, who started the deployment, to what region this deployment is being done and much more. This database will be used to get the total number of deployments in a period of time.

The solution that is being tested as part of this study exports every found issue to Amazon S3. Each issue contains data:

- Timestamp of when the validation was made.
- The repository that was validated.
- Name of the issue.
- Found value.

- Rule category

Later Apache Zeppelin [44] will be used for data analyses.

Different measures are needed to conduct this analysis.

- Name of a repository with at least one issue.
- Number of issues for a repository.
- Number of all deployments in a step of this analysis.
- Names of all repositories used for deployment in a step of this analysis.

6.2 Calculations

In this paragraph, the author will explain how different values will be gathered and calculations made to get the correct results.

- Existing issues

  Gathered by counting all the found issues in the initial data gathering. This value will be static throughout the whole analysis and will be used to compare the different number of issues for every step.

- Repositories with issues

  Calculation is done by counting all the different repositories that have at least one issue. Each step of the analysis will provide the number of repositories. These values will be compared to each other for measurement.

- Deployment

  The number of deployments will be gathered from the company deployment databases. This number is needed to determine if all the repositories that had issues were deployed again. The proposed solution will only report if a issue is found.

- Issues for a repository

  The value will be gathered in every step to calculate the number of unsolved issues for the repositories that were not used in a certain step.

- The difference in issues per step

  The number of issues gathered in one step will be subtracted from the number of issues in the previous stage.

- Difference in repositories with issues per step

  The gathered value in a step will be subtracted from the previous step

- Verifying if all tracked repositories were deployed

  List of reported repository name will be matched to all the repositories deployed in that time period. This will provide the names and the total number of repositories that were not deployed. Not deployed repositories will be considered as a repository with issues.

- Issues of the repositories that were not deployed

  Each issue will be mapped to a repository. Issues from each repository that was not deployed in the time frame of a step will be summed up and added to the number of issues found in that step.

6.3 Initial gathering of issues

In this paragraph, the author presents the process, time frame and findings of the initial data gathering. The initial data gathering about the existing issues was started on 3rd of January 2019 7:00 AM UTC time and ended on 1st of February 2019 16:00 PM UTC time.

In this time frame, 194 different repositories were used for deployment. Issues were found from 128 of them. This means that around 66% of the deployed repositories had at least one or multiple issues. The result is illustrated on Figure 4.

These results were shown and discussed with companies security, development operations and, operations teams. It should also be noted that 3 rules were added that were not required to be followed by the developers before the analysis started.

False positives were removed from the data in collaboration with the company. The main reason for the false positive results was that some services were allowed to not follow some rules. The total of six rules was whitelisted for specific services. This lowered the overall number of repositories with issues by two.

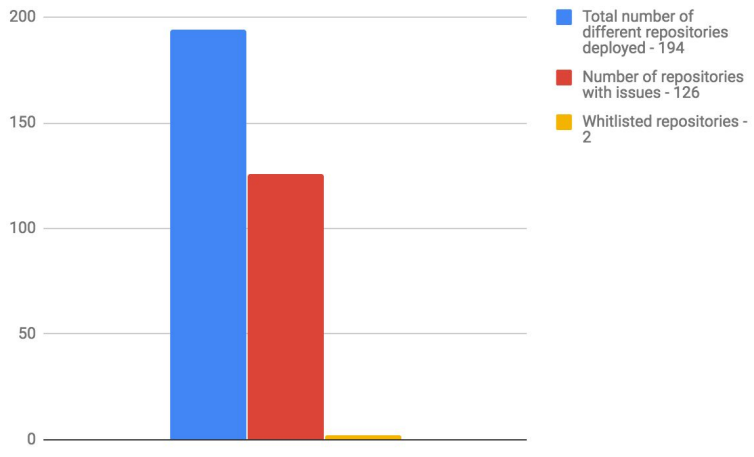

Figure 4 Initial repositories with issues

1707 different deployments were checked and a total of 267 issues were found. 36 different rules were checked for each deployment. Together with the company different rules were grouped based on their importance to low, medium and high level.

- Low - does not pose an immediate threat to the container, Docker host or overall security. Rules related to internal tooling like the name of the owner or not having restart-policy were ranked as low.
- Medium - may cause container or host downtime. May weaken overall security.

  Rules related to resource reservation, limit, health-checks and so on.
- High - poses high threat to the container, Docker host or overall vulnerability.

  Rules about different permissions and unsafe docker images.

From the total of 267 issues 63 or 23,6% were defined as low, 202 or 75,7% as medium and two or 0,7% as high. Owners of the repositories with high-level issues were contacted immediately and the issues were fixed. This changed the total number of issues to 265. The percentage of low, medium and high issues is shown in Figure 5.

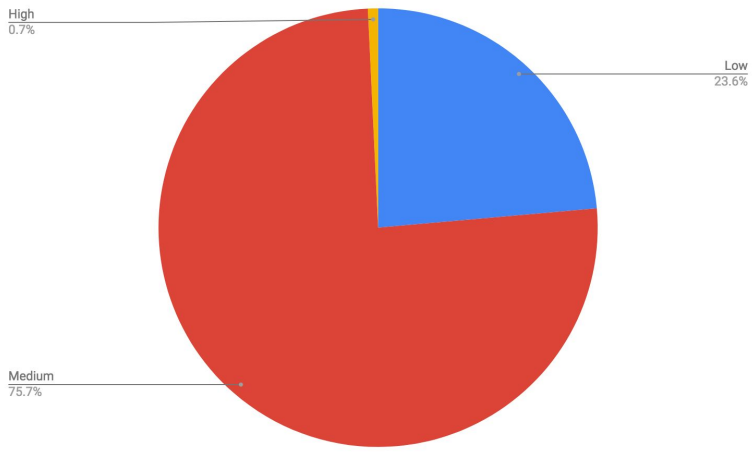

Figure 5 Percentage of repositories with issues from checked repositories

These 126 repositories and 265 issues were used as basis for the measurement and tracked. Each issue for a repository will be tracked separately to verify if it has been solved.

6.4 Global announcements about issues

In this paragraph, the author covers the difference in data gathered initially and after the global announcements to the company's developers.

After the initial data gathering that took place between 3rd of January and 1st of February the next step of the analysis is to announce the findings on Monday 10 AM UTC time for three weeks. Therefore the second step time frame starts on the 4th of February 2019 and ends on the 24th of February 2019. The announcement was made in a public channel that should be used by all developers of the company. The author has no way to make sure how many people saw the announcement and read the initial report. Each announcement was done only once. This was done to measure how many issues will be fixed by developers when the findings are only communicated as a general knowledge not directly to them or their teams.

On the first week, 63 of the tracked repositories were deployed. On the second week, another 23 were added and on the third week 17 more. This makes a total of 103 or 82% of the 126 repositories were checked again. The number of total issues decreased from 267 to 165 which is around 38%. When taking into account the 23 repositories that were not deployed again then the number of existing issues is 191. Most likely the service or container that uses the not deployed repository were not fixed. Thus the number of issues was decreased by 29%. The difference in the number of initially found and remaining issues after this step are shown in Figure 5.

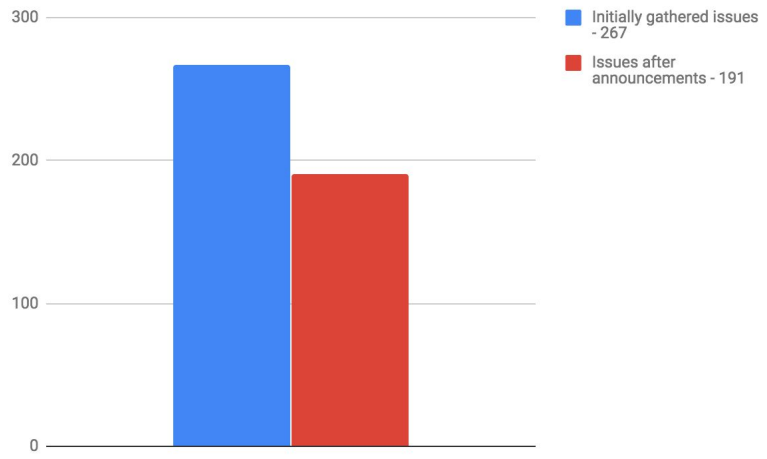

Figure 5 Remaining issues after second step

The number of low-level issues decreased from 63 to 57. More notably the number of medium-level decreased from 202 to 134.

Repositories with issues decreased to 87 which is 31% less than in the previous step. When considering the 23 repositories that were no used for deployments then the number becomes 110 or around 13% decrease. The difference of issues that initially had issues and the repositories that still have issue after this step can be found on Figure 7.

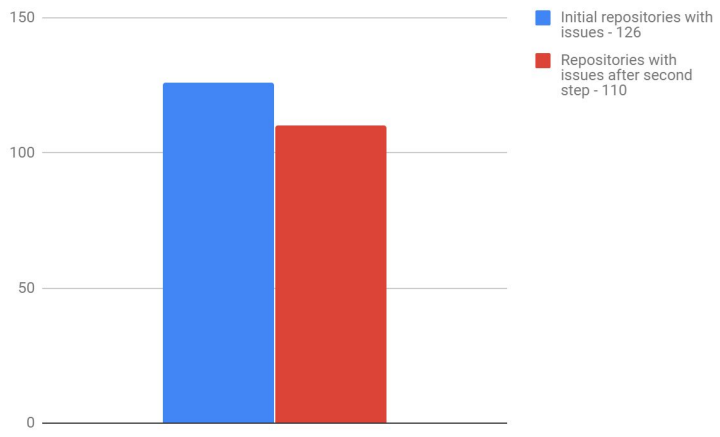

Figure 7 Remaining repositories with issues after second step

This indicates that repositories with more issues were mostly fixed. Each fixed repo had an average of 4 issues.

No new high-level issues were found in that time period. Without including the repositories that were not checked again.

6.5 Direct messaging

Due to unexpected delay, the third step of the analysis had to be delayed to the 4th of March 2019.

In this paragraph the author goes will compare the data that was gathered from the second step to the data gathered after two weeks of sending direct messages.

The third and final step of the analysis is sending direct messages via Slack to the person who started the deployment. A direct message would be sent every time there is a deployment for a repository that has at least one issue and was activated at 10 AM UTC time on 4th of March. Correct person to whom the message should be sent to will be provided by the companies internal tooling.

In the second step, a total of 103 repositories from the 126 that initially had issues were deployed again. Between the 4th and the 18th of March 98 tracked repositories were used for deployments. It should be noted that these 98 repositories had 11 repositories that were not used in the second step. Hence, 114 of the initial 126 repositories between the second and the third step were used again. Also, 8 repositories that were not used in the third step were already fixed in the second step.

The total number of issues decreased to 51. When including the 26 issues from the 20 repositories that were not deployed or fixed the number of issues is 77 or 60% lower than the 191 issues found in the second step and can be seen on Figure 8.

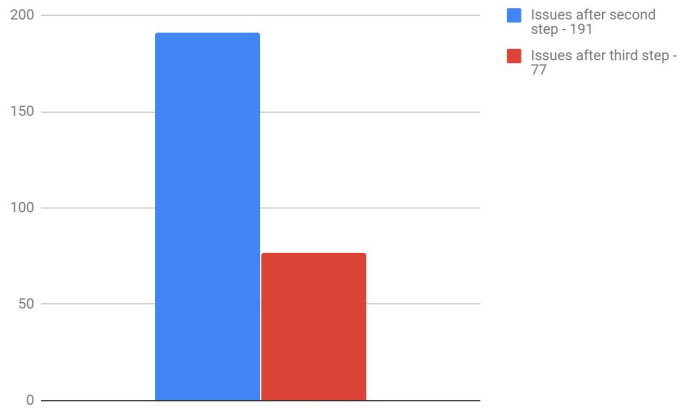

Figure 8 Remaining issues after third step

The number of medium-level issues decreased from 134 to 45 and the number of low-level issues was lowered from 57 to 32. No new high-level issues were found in that time period

The number of repositories with issues was decreased from 110 to 34 which is a 70% difference. When considering the 20 repositories that were not deployed and not fixed the number of repositories with issues becomes 54 or around 51%. The difference of repositories with issues between this and the last step is illustrated on Figure 10.

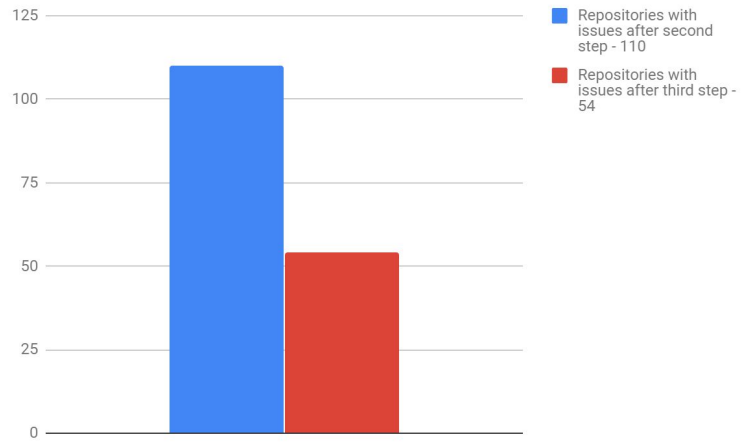

Figure 10 Remaining repositories with issues after third step

6.6 Validation summary

In this paragraph, the author will conclude the analysis that was done in the three steps of the solution validation.

Throughout the three steps, 37 different rules were checked. After each step data was gathered and compared to the previous step. All deployments were tracked using the companies internal tooling to make sure that the same repositories are being compared. This was needed due to the fact that in the time of this analysis the solution only reported repositories that had at least one issue.

In the first, also referred to as, the initial data gathering step that took place between 3rd of January 7:00 AM UTC time and ended on 1st of February 16:00 PM UTC time 1707 deployment using 194 different repositories were checked. The total of 267 issues were found

from 128 different repositories. Due to critical issues, two repositories were fixed immediately. The other 126 repositories were considered as repositories that must be tracked for analysis and comparison between the steps.

In the second step, three global announcements were made to the company's developers about the initially found issues. Each announcement was done once per week on Monday at 10 AM UTC time. Developers had from 4th of February to the 24th of February to resolve the found issues. In the second step, 103 of the 126 tracked repositories were used. Considering the repositories that were not used during the second step the number of issues was lowered to 191 and the number of repositories with issues to 110.

In the third and final step of the validation that took place between the 4th and the 18th of March 98 of the initially used repositories were deployed. The number of remaining issues was 77 and repositories with issues 54 when taking into account the different repositories used in the second and the third step.

Between the second and the third step, a total of 114 different repositories were checked. Hence, 90% of the 126 initially used repositories were checked again. Issues from 72 or 57% of the 126 tracked repositories were resolved during the time of the validation phase. The decrease of repositories with issues throughout the steps can be seen on Figure 10.

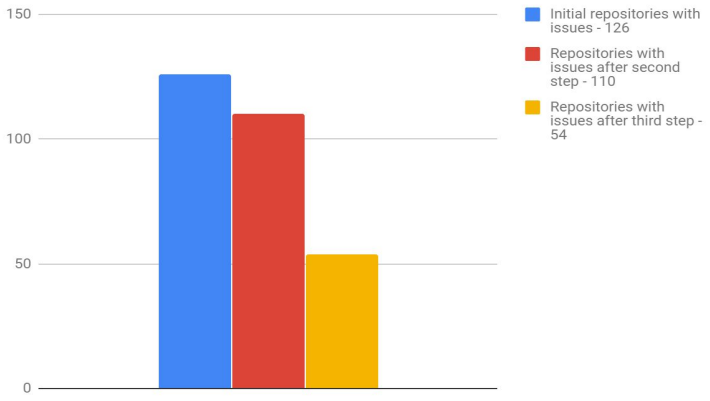

Figure 10 Comparison of repositories with issues in each step

During the validation 190 issues or 71% of the 267 initially found issues were fixed according to the rules and policies of the solution.

In conclusion 1707 deployments for 194 different repositories were checked. From 126 repositories a total of 267 issues were found. The solution helped to resolve 71% or 190 of these issues. The change in total number of issue between the three steps is shown in Figure 11.

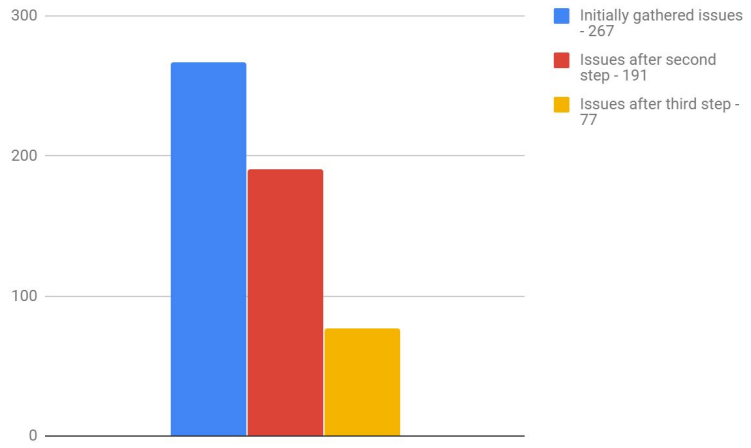

Figure 11 Comparison of issues in each step

7. Future works

At the time of composing this thesis, the presented solution was not yet made public. The author plans to open-source the solution after making modifications to more easily configure different rules into the tool. Open-sourcing provides the possibility for the tool to grow and be used by different users or companies. This may lead to more rules being added or new adaptations the author has not yet considered for this thesis. The tool works well with the company that it was tested with but needs more testing and better support for different use cases to be made public.

Summary

Docker allows users to easily build images or create containers based on them. Even if the creation and deployment process might be easy then following different guidelines about increasing their security can prove to be problematic. Especially when this needs to be done on a company scale. In this thesis a tool called Dora was created by the author. The tool uses best practices and company policies gathered by the author to validate if Docker image building and container or service deployment are done accordingly.

A company needs a solution to help automise the validation of Docker image building and container or service deployment according to Docker's best practices and internal policies. The solution should be lightweight, be able to run in different environments and as part of their deployment pipeline. It should be easily configurable to add, remove or modify rules. Each found issue should result in an educational message to the user. Additionally, the solution should be able to map found issues to Github repositories and be reconfigurable from each repository if needed. Even though the initial need for the tool came from a company the author wanted the tool to be usable for everyone who uses Docker.

In this thesis 37 rules from different sources were gather by the author. These rules were used in a tool created by the author called Dora to help follow them when building Docker images, deploying container or services. The tool is fully automated, lightweight, can run in a Docker container, easily configurable, flexible to cover different use cases and written in Node.js. Regular expressions format is mostly used to describing a search pattern for a specific line, value or a parameter in a command. Dora can read files written in JSON, YAML or plain text format. Configuration for each rule is described in JSON format and each rule can have a file with special functions if needed. When an issue is found a message can be shown in standard output, sent to the user via Slack or to external storage.

In this instance, the rules were best practices provided by Docker and policies from a company that uses Docker as their main service platform. This company has over a hundred deployments using different Docker images a day. In this company, developers are responsible for building Docker images and deploying service or containers. The stated company was also used to validate the usefulness of the proposed solution.

Testing the solution was split into three parts. Initial data gathering, where the existing issues were gathered. Globally announcing the finding and sending direct messages to developers. A total of 267 different issues were found. Announcing the issues globally to the developers lowered the number of issues by 29%. Directly contacting the developers who were responsible for building the image or deploying it proved to be more beneficial as the number of issues was lowered by an additional 42%. Overall the solution helped to solve 71% or 190 of the found issues in 5 weeks.

The solution took approximately 12 weeks to develop and testing it with the company another 9. Creating and testing the solution is the most time consuming part of this thesis.

References

[1] Karl Matthias and Sean P. Kane, O'Reilly 2015, Docker Up & Running

[2] NetApp Blog, Doug Chamberlain March 16, 2018 Containers vs. Virtual Machines (VMs): What's the Difference? [WWW] https://blog.netapp.com/blogs/containers-vs-vms/ (04.01.2019)

[3] Docker Inc. 2017. - Overview of Docker Hub [WWW] https://docs.docker.com/docker-hub/ (15.10.2017)

[4] QuickStart. 2018 - What Is A Cloud Platform? [WWW] https://www.quickstart.com/blog/what-is-cloud-platform/ (12.01.2019)

[5] Google Inc. 2019 - What is cloud computing? [WWW] https://cloud.google.com/what-is-cloud-computing/ (12.01.2019)

[6] Ben Golub, Docker Inc. 2017. - General Session [WWW] https://2017.dockercon.com/, https://www.youtube.com/watch?v=hwkqju_BXEo (15.10.2017)

[7] Steven Signh, Docker Inc. 2018 - General Session [WWW] https://www.youtube.com/watch?v=SGBEq4A8ZHQ (04.01.2019)

[8] Docker Inc. 2018 - Docker Desktop [WWW] https://www.docker.com/products/docker-desktop (04.01.2019)

[9] The Kubernetes Authors. 2017. Kubernetes - [WWW] https://kubernetes.io/ (23.10.2017)

[10] Sarah Novotny. 2017 - Happy Second Birthday: A Kubernetes Retrospective [WWW] http://blog.kubernetes.io/2017/07/ (23.10.2017)

[11] The Kubernetes Authors. 2017. Kubernetes - [WWW] https://github.com/kubernetes/kubernetes (12.01.2019)

[12] Tech Radar, Jonas DeMuro. August 09, 2018 What is container technology? [WWW]

https://www.techradar.com/news/what-is-container-technology

[13] Docker Inc. 2018 - Swarm mode key concepts [WWW] https://docs.docker.com/engine/swarm/key-concepts/ (24.12.2018)

[14] Docker Inc. 2018 - Docker bench security [WWW] https://github.com/docker/docker-bench-security (09.01.2018)

[15] Center for Internet Security 2018 - CIS Benchmarks [WWW] https://www.cisecurity.org/cis-benchmarks/ (09.01.2018)

[16] Github user zuBux 2016 - drydock [WWW] https://github.com/zuBux/drydock

[17] Docker Inc. 2018. - Docker SDK for Python [WWW] https://docker-py.readthedocs.io/en/stable/# (09.01.2018)

[18] Alexei Ledenev. 2016 - Docker Bench Test [WWW] https://github.com/alexei-led/docker-bench-test (12.01.2019)

[19] Docker Inc. Diogo Mónica. 2017 - Actuary [WWW] https://github.com/diogomonica/actuary (12.01.2019)

[20] Anchore Inc. 2019 - Anchore Solutions [WWW] https://anchore.com/solutions/ (09.02.2019)

[21] Jenkins. 2019 - Jenkins [WWW] https://jenkins.io/ (12.01.2019)

[22] Jenkins. 2019 - Jenkins [WWW] https://jenkins.io/doc/ (28.01.2019)

[23] Jenkins. 2019 - Architecting for Scale [WWW] https://jenkins.io/doc/book/architecting-for-scale/ (30.01.2019)

[24] Per Runeson, Martin Host, Austen Rainer and Bjorn Regnell, Case Study Research In Software Engineering

[25] Docker Inc. 2019. - Swarm mode overview [WWW] https://docs.docker.com/engine/swarm/ (24.12.2018)

[26] Docker Inc. 2019. - How nodes work [WWW] https://docs.docker.com/engine/swarm/how-swarm-mode-works/nodes/#manager-nodes (04.01.2019)

[27] Docker Inc. 2019 - How services work [WWW] https://docs.docker.com/engine/swarm/how-swarm-mode-works/services/ (04.01.2019)

[28] Docker Inc. 2019 - Universal Control Plane overview [WWW] https://docs.docker.com/ee/ucp/ (01.03.2019)

[29] Docker Inc. 2019 - Docker Hub [WWW] https://hub.docker.com/ (10.02.2019)

[30] Docker Inc. 2019 - Docker Hub [WWW] https://docs.docker.com/docker-hub/ (10.02.2019)

[31]Erika Heidi. 2016 - What is High Availability [WWW] https://www.digitalocean.com/community/tutorials/what-is-high-availability (24.12.2018)

[32] NeuVector Inc. 2019 - NeuVector [WWW] https://neuvector.com/ (24.01.2019)

[33] NeuVector Inc. Fei Huang. 2019 - 17 Backdoored Malicious Images Removed From Docker Hub, But Are You Really Any Safer? [WWW] https://neuvector.com/docker-security/backdoored-images-removed/ (24.01.2019)

[34] Docker Inc. 2019. - Official Images on Docker Hub [WWW] https://docs.docker.com/docker-hub/official_images/ (10.02.2019)

[35] Center of Internet Security. 2019 - CIS Benchmarks [WWW] https://www.cisecurity.org/cis-benchmarks/ (10.02.2019)

[36] Sysdig Inc. 2019 - 7 Docker security vulnerabilities and threats [WWW] https://sysdig.com/blog/7-docker-security-vulnerabilities/ (10.02.2019)

[37] Adrian Mouat, O'Reilly 2015, Docker Security

[38] Docker Inc. 2019 - Docker Captains [WWW] https://www.docker.com/community/captains (06.05.2019)

[39] Docker Inc. 2019 - Compose file version 3 reference [WWW] https://docs.docker.com/compose/compose-file (01.03.2019)

[40] Github Inc. - Github [WWW] https://github.com/ (30.01.2019)

[41] Jan Goyvaerts, Just Great Software Co. Ltd. 2019 - What Is a Regular Expression, Regexp, or Regex? [WWW] https://www.regexbuddy.com/regex.html (01.03.2019)

[42] Docker Inc 2019. - Child commands [WWW] https://docs.docker.com/engine/reference/commandline/docker/ (03.03.2019)

[43] Docker Inc. 2019 - Get Started, Part 3: Services [WWW] https://docs.docker.com/get-started/part3/#your-first-docker-composeyml-file (01.03.2019)

[44] The Apache Software Foundation. 2019 - Apache Zeppelin [WWW] https://zeppelin.apache.org/docs/0.5.5-incubating/storage/storage.html (01.03.2019)

Annex 1 - Virtual Machines

As server processing power and capacity increases, bare metal applications are not able to profit from all the resources. VMs or virtual machines are designed to run software on top of a physical server to emulate a particular hardware system. This is mostly done by using a hypervisor. A hypervisor is a software, firmware, or hardware that creates and runs VMs. It is stationed between the physical hardware of a server and the virtual machine and is necessary for virtualizing the physical part. [2]

Each virtual machine has the ability to run a unique guest operating system. This allows VMs with different operating systems to run on the same physical server. Each VM can have its own binaries, libraries, and applications. This provides benefits like consolidate different applications onto a single host and save hosting costs by using resources more efficiently. It also makes the server provisioning faster with the use of VM images, improves disaster recovery since the backup hardware does not have to be exactly the same anymore. [2]

These improvements over bare metal might not be enough for all use cases. Each VM has its own operating system, which adds overhead in memory, CPU and storage usage. This may add complexity to different stages in a software deployment lifecycle since VMs must be able to house all the images of different environments. Virtualization may also limit the portability of applications between public and private clouds or traditional data centers. [2]

Cloud platforms/computing

A Cloud platform assumes the responsibility of providing a customer or a user with a data center on the fly. Cloud hosting platforms resources are usually shared between many tenants. [4]

Cloud computing is an alternative to building and maintaining data centers by providing resources like storage, computing power, networking, data processing, analytics, machine learning, monitoring and even fully managed services to customers. In the past cloud computing was related to startups and visionary enterprise users. Today cloud computing is part of the whole IT enterprise and mainstream across organizations of any type. [5]

Cloud computing benefits are innovation and the economics of businesses overall by providing the opportunity to improve flexibility, reduce costs, and focus on their products, not on the infrastructure. [5]

Annex 2 - Difference between containers and virtual machines

Container and virtual machines might look similar from the outside but are very different in the inside and how they run or use resources. The primary difference is that containers use the host machine operating system and resources provided to it to run multiple workloads. Virtual machines, on the other hand, use virtualized host machine hardware to give the possibility to run different operating systems on the same hardware.

Containers can provide more flexibility, portability, and speed to help the software development cycle. [2]

In more detail, virtual machines allow the user to run a complete kernel and operating system on top of a virtualization layer, which is mostly known as a hypervisor. Virtual machines are strongly isolated from each other even when they are running on the same host. This is due to the fact that each hosted kernel is sitting in a separate memory space and has defined entry points to the actual hardware. [1, pg 63]

Containers share a single kernel and operating system and the isolation is done within the same kernel. Containers advantage comes from using the resources of a single host more efficiently by not needing a whole operating system for each isolated function. [1, pg 63]. The talked about differences are illustrated in Figure 12.

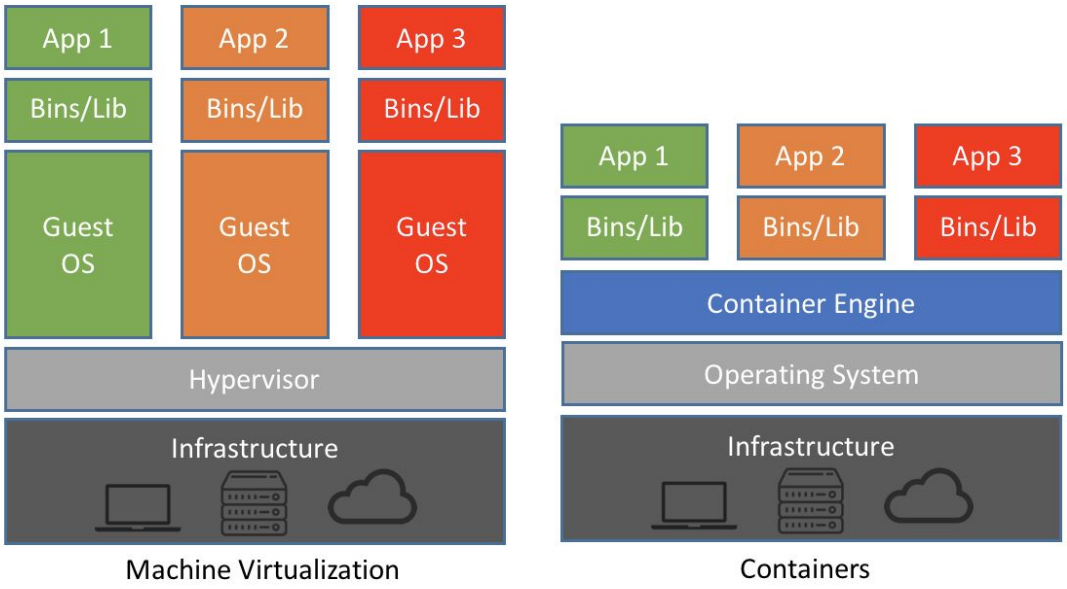

Figure 12 Difference between virtual machines and containers[2]

Annex 3 - Container popularity

Docker and containers popularity has been rising every year and is not slowing down but even increasing.

At DockerCon 2017 that is a conference hosted by Docker held in Austin, Texas, the current CEO of Docker Inc. Ben Golub stated - there are over 900 000 Docker apps on Docker Hub [3] and over 14 million unique IP addresses have accessed Docker hub. At 2014, when the first DockerCon was held, there were only about 1500 Docker apps on Docker Hub. This

means that the amount of new Docker apps that have been uploaded to Docker Hub has increased by 600%. [6]. At DockerCon 2018 held in San Francisco, California, Docker CEO and Chairman Steven Signh said that more than a million new developers started using Docker Desktop [8], over a million new applications have been added to Docker Hub in 2018 alone and more a billion containers are downloaded from Docker Hub every two weeks. [7]

It is clear to see that the usage of Docker has grown over the 4 years. People are interested in containers and how they work or how to use them themselves.

Not only is Docker gaining more and more popularity but so are the other containers' orchestrators like for example Kubernetes.

Kubernetes is a system that provides automated deployment, scaling, and management for containers like Docker does [9]. Sarah Novotny a Program Manager in Kubernetes Community says in their blog post which was written on 27.07.2017 that the Kubernetes project has had 50,685 commits in GitHub[40] over the last 12 months [10]. At the end of 2018, the Kubernetes GitHub commits have grown well over 73 000. [11]

Increasing popularity means that the security must increase and the way containers are used should be thoroughly thought through.

Annex 4 - Case study design elements

Element	Example Questions Describing the Element
Rationale	Why is the study being done?
Purpose	What is expected to be achieved with the study?
The case	Overall, what is being studied?
Units of analysis	In more detail, what is being studied?
Theory	What is the theoretical frame of reference?
Research questions	What knowledge will be sought or expected to be discovered?
Propositions	What particular (causal) relationships are to be investigated?
Define concepts and measures	How are entities and attributes being defined and measured?
Methods of data collection	How will data be collected?
Methods of data analysis	How will data be analyzed?
Case selection strategy	How will cases (and units of analyses) be identified and selected?
Data selection strategy	How will data be identified and selected? For example, who will be interviewed? What electronic data sources are available for use in the study? What nonelectronic, naturally occurring data sources are available for use in the study?
Replication strategy	Is the study intended to literally replicate a previous study, or theoretically replicate a previous study; or is there no intention to replicate?
Quality assurance, validity and reliability	How will the data collected be checked for quality? How will the analysis be checked for quality?

Figure 13 Case study design elements [24]

Annex 5 - Docker images

All Docker containers are based on a Docker image. Docker image provides the bases to everything that is deployed or run on Docker. Docker images can be created or downloaded from a public registry. All Docker images have one or more file system layers which in most cases have a one-to-one mapping to each individual build step that is used to create an image. [1, pg 43]

Docker relies heavily on its underlying filesystem to build and manage the multiple layers that are combined into a single usable image. [1, pg 43]

Docker images can be named and tagged to provide a quick way to determine the purpose and version of that image. [1, pg 43]

Dockerfile

Dockerfile lets the user create a Docker image with default tools provided by Docker. Dockerfile describes all the steps which are required to create a Docker image. Dockerfile is usually placed in the root directory of the source code repository or directory which makes it easier to include the wanted files to the image. [1, pg 43-46]

A new layer is stored into the Docker image from every line in Dockerfile. This gives the possibility to only build layers that are not part of any previous builds that have been cached. Any Docker image could be for instance built based on Linux image, but also a previously built and fully working image could be taken as the base. A base for an image can be locked down to a specific version by using a name and a tag for the base image. [1, pg 43-46]

Dockerfile allows the user to specify different metadata, variables, and commands to an image:

- **ENV**: Usable for the application or code running inside the container built from that image.
- **USER**: Operating system user that will be used inside the container. By default, the user is set to root.
- **RUN**: Can be used to run a specified command or commands in a Docker image. Run is often used to install different packages, libraries or software.
- **ENTRYPOINT**: Is used to determine the first command which should be executed when the container starts. Entrypoint can be used to set parameters, run a script inside the container or even start an application.
- **CMD**: Is used to note the command and/or parameters that will be executed after the entrypoint and can be used as an addition to the entrypoint.
- **COPY** - Copies files or directories from the specified path from the local storage to the given path inside the Docker image.
- **WORKDIR**: Used for specifying the default path inside the container. In many cases set to the path of the application inside.
- **MAINTAINER**: Provides the possibility to add contact information of the Dockerfile's author. The maintainer will populate the Author field in that Docker image and images which use it as a base.
- **LABEL**: Gives the ability to add labels to images and containers which use that image. Labels are key-value metadata pairs that can be later used to search for or identify Docker images or containers. [1, pg 43-46]

Annex 6 - Docker container

Containers, as discussed earlier in Containers paragraph of this thesis, are self-contained execution environments that share the kernel of the host machine. Containers can be isolated from each other if needed. Since containers are using the host's kernel means that only containers compatible with that kernel can be run on that host. Meaning a Windows application container cannot run on a host with Linux kernel. [1, pg 63]

Containers are not a new idea or something that Docker came up with. Containers provide a way to encapsulate and isolate a part of the running system. Docker just makes creating and managing containers easy and fast. [1, pg 63]

In late 2013, after Docker was announced many other container technologies and engines were shown to the public. As an example, Google open-sourced its internally used container engine which they had been using for some years already. [1, pg 63]

Creating a Docker container

Starting a Docker container is easy. `Docker run` command must be executed on a host with a Docker engine and a container is spawned. The `docker run` command, in reality, wraps two separate steps into one. First, a container is created from the given Docker image and then the created container is started. The second part can also be done with the `docker start` command. [1, pg 63]

Docker container configuration

Many arguments can be passed to run or start a container. Some of them are mandatory like for instance the image from which the container should be created. Others are optional that

allow the user to configure the container. Even though a container is always built from an underlying image the following arguments can affect the final set. [1, pg 66 - 78]

- **Container name:** By default, Docker will name a container randomly. Docker will combine an adjective with a famous person name. The argument `--name` can be used to give the container a specific name.

- **Labels**: Labels are containers or image metadata key-value pairs. All the labels from the underlying image will be carried on to the newly built image. Containers can be given additional labels by using `--label` argument. Labels can be later used for filtering or searching containers based on a label.

- **Hostname:** By default, some system files will be copied to a container configuration directory by Docker on start. After that, it will be bind mounted to the container. One of these files is for example /etc/hostname on Linux systems. The hostname can be changed with `--hostname`.

- **Domain Name Service or DNS**: DNS configuration just as the hostname will be copied and mounted to the container. If DNS needs to be changed `--dns` or `--dns-search` can be used.

- Media Access Control or MAC address: By default, the container will get a MAC address which starts with 02:42:ac:11 prefix. If this needs to be changed then `--mac-address` can be used.

- **Storage volumes**: Sometimes additional volumes need to be attached to the container or localhost disk space is not enough. To mount a volume to a container `-v` should be used.

- **Resource quotas**: The term “noisy neighbor” is often used when talking about different problems about cloud or microservice architectures. That term refers to

applications or application running on the same physical system which have a noticeable impact on the performance and resource availability.

In traditional virtual machines, CPU and memory allocation can be controlled easily. Since it is an important topic then Docker also supports CPU share, CPU, I/O, memory and memory swap limits. CPU limit can be modified with `--cpu`, CPU share with `--cpu-share`, I/O with `--IO`, memory limit with `--memory-limit` and swap with `--memory-swap`.

More arguments for resource management for Docker containers can be found in Docker documentation[5]. [1, pg 66 - 78]

Annex 7 - Docker orchestration with the swarm

The main purpose of Docker swarm[25] is to provide high-availability[31] to services running inside a Docker containers.

Docker swarm has two types of nodes. Ones for hosting services and others to manage the swarm.

Docker swarm managers

Managers are responsible for management tasks like - maintaining cluster state, scheduling services and providing swarm mode HTTP API endpoints. [26]

Managers maintain the internal state of the entire swarm and all the running services. Docker recommends running an odd number of managers to provide high-availability and fault-tolerance. [26]

[5] https://docs.docker.com/config/containers/resource_constraints/

Docker swarm workers

Workers are instances of the Docker Engine to run and maintain containers. A single manager swarm can be created where the node acts as a manager and hosts containers. But at least one manager is needed to use workers through the swarm HTTP API. Docker swarm workers do not make scheduling decisions or server swarm HTTP API. [26]

Docker Swarm Mode features

Docker swarm provides many features to provide high-availability, scaling, networking and more.

- **Multi-host networking**: Multi-host networking feature provides the possibility to create the same network across different hosts. Containers connected to that network will be assigned an address and can communicate with each other freely. [25]

- **Scaling**: Scaling provides high-availability for the containers. So basically a container with the same purpose can run on multiple hosts at the same time. This means that if there is a security hole in the container then not one but all the hosts the container is running on can potentially be affected. So providing as much protection to the containers is a must. [25]

- **Cluster management integrated with Docker engine**: Docker engine CLI can be used to create a swarm that can host different services. No other orchestration is needed for that. [25]

- **Decentralized design**: Entire swarm can be built from a single disk image. There is no difference between swarm managers and workers deployment. This allows

changing node roles at runtime. A manager can be turned into a worker and vice versa. [25]

- **Declarative service model**: Gives the user the possibility to define different services into a service stack to provide better visibility, filtering, and deployment to applications comprised of many services. [25]
- **Desired state reconciliation**: By constantly monitoring the cluster state managers are able to create new replicas of crashed containers. [25]
- **Service discovery**: Docker Swarm managers assign unique DNS names and provide load balancing for running containers. Every container running in a swarm can be contacted by using the embedded DNS server of the swarm. [25]
- **Load balancing**: Service ports can be exposed to an external load balancer. Internally service container distribution can be specified between nodes. [25]
- **Secure by default**: Docker Swarm nodes enforce TLS authentication and encryption for communication between other nodes or itself. Docker also supports self-signed or custom root certificates. [25]
- **Rolling updates**: Docker Swarm Managers allow to delay service container deployment to different nodes. [25]

Annex 8 - Docker service

Docker service is used to deploy an image for a microservice to swarm. Usually, this image is part of a larger application. For example a front-end and back-end application with a database to store data or any other executable application which should run in a distributed

environment. Similar to Docker container when running a service the user must define what image to run and different arguments about how to do that or what configuration to use. These parameters can, for example, be to set:

- Number of containers that should be running at once.
- Network to which the containers should be connected to and which ports the containers should be using.
- Different reservations and limits
- Update policies about what to do when the service is updated.
- Restart policies about what to do when a container crashes or is restarted. [27]

Docker service, containers tasks, and scheduling.

Deploying a service to Docker swarm means that the swarm manager accepts the service configuration. The manager will schedule the containers for that service to nodes in the swarm as tasks. Tasks are independent of each other on different nodes in the swarm. Since a container is an isolated process then each task takes care of one container. If the container is running then the task will be changed to running state. If the health check of the container fails or the container is terminated the task terminates with it. [27]

When declaring a desired state for service by creating, removing or updating it, a new task will be scheduled for each container for that service. Each task will be a slot that the scheduler will fill or clears depending on the given task. [27]

Annex 9 - Docker hub

"Docker Hub is the world's largest library and community for container images" [30]

It has over 100,000 different Docker container images for community members, software vendors or open-source projects. [29]

Docker hub is provided by Docker with the purpose of sharing and finding Docker container images. It provides repositories to push and pull images to or from. Teams or organizations can manage their private repositories. High-quality official images from different software vendors provided by Docker. Webhooks and builds that can be used with tools like GitHub or Bitbucket to automate image building, pushing and verifications. [30]

Annex 10 - Deployment pipeline overview

The company's deployment pipeline has many internal tools that handle the communication between different tools, test and evaluate the developed code, monitor the whole process and report the outcome. In this paragraph, the author will not dive deeply into different tooling and processes but will give a general overview and logic behind the deployment pipeline.

- Creating a branch in Github repository

  A newly developed or changed code will be added to the branch by the developers.

- Creating a pull request in Github

  The branch that was created must be reviewed by the owners of this repository.

- Adding a label to the pull request

  Different labels in a pull request trigger a different kind of automation.

- Git Checkout

  Wanted branch for the specified service will be downloaded to a Jenkins agent in a test environment. Each build executed by Jenkins has its workspace where the data is saved and where by default the commands are executed.

- Validation

  The downloaded code will be checked and validated by different tools. The solution will be part of this step.

- Dependency installation

  Code dependencies will be installed into the Jenkins workspace so that the code could be executed on the Jenkins agent if needed. Dependencies will be installed differently based on the language in which the code is written in.

- Post installation

  Used to run custom scripts or commands which are needed for that specific code or repository.

- Pre Docker build tasks

  This stage is used to for instance build the code before a Docker image is created.

- Docker image building

  Docker image is built by using the Dockerfile from the downloaded branch of a Github repository.

- Code unit testing

Unit tests for the downloaded code will be executed and results will be reported.

- Code functional testing

  Functional tests will be executed against the build Docker image and results will be reported.

- Docker image push to Docker hub

  Created Docker image will be pushed to Docker hub so that it could be downloaded to Docker swarm or any other Jenkins agent.

- Pulling Docker image to the correct region

  Docker image that was created and pushed to Docker hub in the previous stage will be downloaded to the Jenkins slave in the correct environment from which it could be deployed to Docker swarm.

- Pre-deployment tasks

  Used to run any needed commands or scripts before the Docker service using the previously pulled image is deployed to a Docker swarm.

- Docker service deployment

  Docker service using the built image will be deployed to the correct region.

- Post-deployment tasks

  This step is used in case something additional needs to be done for the service that was deployed. For example database migrations.

- Post-deployment tests

Tests will be executed against the already deployed code or service to ensure that everything works as intended.

- Docker service scaling

  This step is used in case the service scale needs to be changed.