Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

Katrine Wist
Malene Helsem

An Extensive Analysis of the Current Vulnerability Landscape in Docker Hub Images

Master's thesis in Communication Technology
Supervisor: Danilo Gligoroski
June 2020

NTNU
Kunnskap for en bedre verden

Katrine Wist
Malene Helsem

An Extensive Analysis of the Current Vulnerability Landscape in Docker Hub Images

Master's thesis in Communication Technology
Supervisor: Danilo Gligoroski
June 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

NTNU
Norwegian University of
Science and Technology

Title: An extensive analysis of the current vulnerability landscape in Docker Hub images

Students: Katrine Wist and Malene Helsem

Problem description:

Container technology could be seen as a new paradigm shift that started a few years ago. Since then, its popularity has rapidly increased and the use of container technology, especially in enterprises, is replacing virtual machines at a quick rate. Containers offer a lightweight, flexible, and portable virtualization solution. As opposed to virtual machines, containers virtualize at the operating system level. This technique creates an isolated environment for applications to run without the need for a separate virtual machine for each application.

As of today, Docker is the leading container platform with most of the container market. Docker offers an online repository called Docker Hub for distribution of Docker images, where a container is a running instance of an image. One could say that Docker Hub is contributing to the expansion of container technology usage by making it easy to maintain and distribute images between users. However, a drawback is that it makes it hard for Docker to enforce control regarding how often software patching should happen since anyone can maintain images. The result is that outdated and vulnerable software exist in Docker Hub images. Docker Hub is at the time of writing the world's largest platform for sharing container images and hosts over 3 million Docker images.

This massive popularity comes at the cost of a definite need for security to be prioritized. A problem is that most of the previous studies focus on security related to the actual Docker implementation, whereas research done on the present vulnerability landscape in Docker images is less investigated. With that said, the research that has been done revealed that many Docker Hub images contain vulnerabilities. This situation makes it essential to investigate further and look at what the situation is like today.

The aim of our master thesis is, therefore, to investigate the current vulnerability landscape in Docker Hub images by conducting an analysis of approximately 2,500 images. By looking for publicly known vulnerabilities in the images, we want to categorize and collect statistics about the vulnerabilities, inspect what images and packages are affected, and look at the correlation between several different image attributes and vulnerabilities. We are also going to take a more in-depth look at the images, packages and vulnerabilities that are the most exposed. Finally, we will present and discuss our findings in a structured and transparent way in order to contribute to the research community by raising awareness.

Responsible professor:	Danilo Gligoroski, IIK
Supervisor:	Danilo Gligoroski, IIK

Abstract

The use of container technology has skyrocketed during the last few years, with Docker as the leading container platform. Docker's online repository for publicly available container images, called Docker Hub, hosts over 3.5 million images at the time of writing. This makes it the world's largest community of container images. Some of the positive traits of using containers, as advertised by Docker Inc., is that they are standard, lightweight, and *secure*. However, previous research reveals that Docker Hub images actually contain a large number of vulnerabilities on average.

In this thesis, we investigate the current vulnerability landscape of Docker Hub images. By analyzing approximately 2,500 images from Docker Hub, we uncover that Docker containers are not as secure as advertised. Our main findings reveal that (1) the number of newly introduced vulnerabilities on Docker Hub is rapidly increasing; (2) certified images are the most vulnerable; (3) official images are the least vulnerable; (4) there is no correlation between the number of vulnerabilities and image features (i.e., number of pulls, number of stars, and the days since the last update); (5) the most severe vulnerabilities originate from two of the most popular scripting languages, JavaScript and Python; and (6) Python 2.x packages and jackson-databind packages contain the highest number of severe vulnerabilities. We use an open-source vulnerability scanner to perform the analysis of such a large number of images and develop our own scripts and tools. We perceive our study as the most extensive vulnerability analysis of Docker images published during the last couple of years in the open literature.

It is of particular interest to perform this kind of analysis for several reasons. Firstly, the vulnerability landscape is rapidly changing; secondly, the vulnerability scanners are constantly developed and updated, and new vulnerabilities are discovered. Lastly, the volume of images on Docker Hub is increasing every day. Our work contributes to the research community by shining light upon the current status of vulnerabilities in Docker images. The main contribution of this thesis is, thus, to raise awareness and understanding of the vulnerability landscape of Docker Hub, and to provide automated vulnerability scanning software and tools for others in the community to use. Our work is a stepping stone towards the ultimate goal of securing the Docker Hub ecosystem.

Sammendrag

I løpet av de siste årene har bruken av containerteknologi skutt til værs med Docker som den ledende containerplattformen. Dockers plattform for offentlig deling av containerbilder (images), kalt Docker Hub, inneholder i skrivende stund over 3.5 millioner bilder. Dette gjør Docker Hub til verdens største plattform for deling av containerbilder. Som annonsert av Docker Inc., er noen av de positive sidene ved containerteknologi at den er standardisert, lettvektig og *sikker*. Imidlertid kan tidligere forskning avsløre at Docker Hub-bilder faktisk inneholder et gjennomsnittlig høyt antall sårbarheter.

I denne masteroppgaven undersøker vi sårbarhetsbildet knyttet til Docker Hub-bilder. Gjennom en grundig analyse av omtrent 2,500 bilder fra Docker Hub, avslører vi at bilder fra Docker Hub ikke er så sikre som annonsert. Våre hovedfunn er at (1) antall nye introduserte sårbarheter på Docker Hub er raskt økende; (2) sertifiserte bilder er de mest sårbare; (3) offisielle bilder er de minst sårbare; (4) det er ingen korrelasjon mellom antall sårbarheter og bildeattributter (slik som antall nedlastninger, antall sjerner og antall dager siden siste oppdatering); (5) de mest alvorlige sårbarhetene stammer fra to av de mest populære skriptingspråkene, JavaScript og Python; og (6) pakker av typen Python 2.x og jackson-databind inneholder det høyeste antallet alvorlige sårbarheter. For å kunne gjennomføre analysen av såpass mange bilder har vi tatt i bruk en sårbarhetsskanner med åpen kildekode og har videre utviklet våre egne kodeskript og verktøy. Vi ser på vår studie som den mest omfattende sårbarhetsanalysen av Docker-bilder som er publisert de siste par årene.

Det er av spesiell interesse å utføre denne typen analyse av flere grunner. For det første er sårbarhetsbildet i kontinuerlig forandring. For det andre blir sårbarhetsskannere utviklet og oppdatert fortløpende, og nye sårbarheter blir oppdaget. I tillegg øker volumet av bilder på Docker Hub hver eneste dag. Vårt arbeid bidrar til forskningsmiljøet ved å tydeliggjøre den nåværende situasjonen av sårbarheter i Docker-bilder. Hovedbidraget fra denne masteroppgaven er derfor å øke bevisstheten og forståelsen rundt sårbarhetsbildet knyttet til Docker Hub, i tillegg til å levere programvare for en automatisert sårbahetsskanner som kan brukes av andre i forskningsmiljøet. Vårt arbeid bidrar til å komme et steg nærmere det endelige målet om å sikre økosystemet til Docker Hub.

Preface

This thesis is finalizing our Master of Science Degrees in Communication Technology, with a specialization in Information Security, at the Norwegian University of Science and Technology (NTNU). The work on this master thesis started in January 2020 and was completed in June 2020.

We want to thank our responsible professor and supervisor, Danilo Gligoroski, for valuable input and feedback throughout the work on this thesis. Our five years at NTNU has gone with the blink of an eye, and we would like to thank the university for providing us with valuable and rewarding education, and new insights. We would also like to give a special thanks to everyone who has made the time in Trondheim memorable and special.

Katrine Wist and Malene Helsem
Trondheim, June 5th 2020

Contents

List of Figures

List of Tables

List of Listings

List of Acronyms

API Application Programming Interface.

CI/CD Continuous Integration/Continuous Delivery.

CLI Command Line Interface.

CNAs CVE Numbering Authorities.

CPE Common Platform Enumeration.

CPU Central Processing Unit.

CSV Comma Seperated Values.

CVE Common Vulnerabilities and Exposures.

CVSS Common Vulnerability Scoring System.

CWE Common Weakness Enumeration.

ER Entity-Relationship.

GUI Graphical User Interface.

KDE Kernel Density Estimation.

LXC Linux Containers.

NVD National Vulnerability Database.

OS Operating System.

PCC Pearson Correlation Coefficient.

RAM Random-Access Memory.

RDBMS	Relational Database Management System.
REST	Representational State Transfer.
SQL	Structured Query Language.
VM	Virtual Machine.

Chapter 1 Introduction

Container technology has been around for a long time in Linux systems through Linux Containers (LXC), but it was not commonly used until this last decade. It was with the introduction of Docker in 2013 that the containerization popularity exploded. Container technology has revolutionized the development of software and could be seen as a new paradigm shift. Containerization is proven beneficial for Continuous Integration/Continuous Delivery (CI/CD) pipelines, it is providing an effective way of organizing microservices, it is making it easy to move an application between different environments, and, in general, it simplifies the whole system development life cycle. Virtual machines are being replaced by container technology continuously, and the trend is that an increasing number of companies are choosing to containerize their applications. Gartner predicts that more than 70% of global companies will have more than two containerized applications in production by 2023, this is an increase up from less than 20% in 2019 [Hew19].

Software containers got its name from the shipping industry, as the concepts are fundamentally the same. A software container is code wrapped up with all its dependencies so that the code can run reliably and seamlessly in any computer environment, isolated from other processes. This concept tackles the problems that occur when moving software from one environment to another. To give an example, code can run correctly on the developer's machine, but fails to work properly on the test server. Hence, containers are a favorable technology to achieve isolation, portability, and scalability, as well as its characteristics of being lightweight and fast.

There are a few alternatives to Docker, such as LXC, rkt, Apache Mesos and, Vagrant.[1] However, as of today, Docker is the most popular and one of the most acknowledged containerization platforms. There is no doubt that Docker started the container revolution, and its popularity has had a steady growth. Statistics from 2018 show that Docker was running on 20% of hosts, and that 25% of companies

[1]LXC: *https://linuxcontainers.org/*, rkt: *https://coreos.com/rkt/*, Apache Mesos: *http://mesos.apache.org/* and Vagrant: *https://www.vagrantup.com/*

had adopted Docker at that time [Dat18].[2] One of the contributors to Docker's popularity is Docker Hub, which is a registry service for sharing of Docker images.[3] It currently hosts over 3.5 million container images, and the number is continuously increasing.[4] There are four different image repository types: official, verified, certified, and community. Images can be uploaded and maintained by anyone, which creates an innovative environment for anyone to contribute and participate. However, on the downside, this makes it challenging for Docker to ensure that packages and applications are correctly implemented and up to date to avoid outdated and vulnerable software.

The terms cyber security and information security are closely related and often used interchangeably when considering security in computer systems. The work in this thesis is based on the perspective of cyber security rather than information security. Information security is concerning the protection of *information*, while cyber security is about the protection of information *systems*. Cyber security is by Schatz et al. [SBW17] defined as,

> "... the protection of information systems from theft or damage to the hardware, the software, and to the information on them, as well as from disruption or misdirection of the services they provide."

The securing of software is the foundation of this thesis, and thus, we would like to point out that throughout this thesis, the word security is used as a synonym for cyber security.

1.1 Project motivation

When looking into the security of Docker, there are two aspects that need to be considered: the security of the Docker software at the host, and the security of the Docker containers. Docker Inc. claims that "Docker containers are, by default, quite secure; especially if you run your processes as non-privileged users inside the container" [Docc]. However, some people are not aware that Docker (the Docker daemon and container processes) runs with root privileges by default, which exposes a huge attack surface [FO19]. There are multiple scenarios in which a container could be a threat. The first scenario is that a container contains malicious code deliberately placed there by an adversary. Secondly, the container holds instructions that configure insecure settings, potentially making the whole computer system vulnerable. Thirdly, the container contains flaws and vulnerable software, which will be the focus area of this thesis. A single vulnerable container is enough for an adversary to achieve

[2]Numbers based on data from Datadog customers, a monitoring service for cloud-scale applications with thousands of companies from a big span of industries as customers.
[3]Docker Hub: *https://hub.docker.com/*
[4]A Docker container is a running instance of a Docker image.

privilege escalation. Hence, it is essential to raise awareness towards the vulnerability landscape of Docker images. This is because the securing of the Docker ecosystem is highly related to discovering what the vulnerability landscape in Docker images is like.

It does exist some previous research in the field of addressing the vulnerability landscape of Docker Hub. The studies have been conducted with some years apart. In 2015, BanyanOps presented results that revealed that 40% of Docker images contained high rated vulnerabilities [GDT15]. Further, in 2017, Shu et al. found that, on average, an image contains more than 180 vulnerabilities [SGE17]. Finally, in 2019, Socchi and Luu could reveal that the majority of official, community, and certified images hold less than 75 vulnerabilities and that the majority of verified images contain less than 180 vulnerabilities [SL19]. To read about the related work in more detail, see Section 2.4. With the previous findings in mind, it is clear that Docker Hub consists of a vast number of vulnerable images that could expose a serious threat to any computer system. As the Docker Hub ecosystem is continuously changing, it is unquestionable that this also applies to the vulnerability landscape. These observations make it important to address what the vulnerability landscape of Docker Hub is like today. In addition, we will use a different scanner, which could give a new and nuanced view of what the vulnerability landscape is like. With that said, the aim of our master thesis is to provide an important contribution in the field of security in the Docker ecosystem. More specifically, by raising awareness of the situation in today's vulnerability landscape.

1.2 Research scope

In this thesis, we perform an extensive vulnerability analysis of Docker images. We conduct a vulnerability scanning of approximately 2500 Docker images, including most official, and a portion of verified, certified, and community images. Vulnerability data is gathered, systemized, and presented to reveal the current situation.

We do not see it as convenient to replicate previous research, as a direct comparison is rather difficult to do. There are several reasons for this: the vulnerability landscape is rapidly in change, the vulnerability scanners are constantly developed, new vulnerabilities are discovered, and the volume of images on Docker Hub changes from day to day. Thus, we perform an independent analysis instead. We implement our own software for conducting the analysis, and use a well-respected vulnerability scanner not previously used in related research to gather the desired data. However, we do include a discussion regarding the trends on vulnerabilities based on our results compared to the results of previous research, but it will not be included in our main research scope.

1.2.1 Research questions

Based on the introduction, the project motivation, and the previous comments, our research questions are explicitly defined below. The research questions are partially based on the research questions defined in our project report [HW19].

RQ1: How can vulnerabilities found in Docker images be systemized in order to investigate the current vulnerability landscape of Docker Hub?

RQ2: How do image features and the number of vulnerabilities correlate in images?

RQ3: Which types of vulnerabilities are the most severe?

RQ4: Which packages contain the most severe vulnerabilities?

1.2.2 Contributions

Our main contribution is new insights into the vulnerability landscape of Docker Hub images. Our software deliveries are considered as an extra contribution. Thus, all implemented software is provided in an open source repository on GitHub.[5] As such, our implemented tools are publicly available for the community to use and for our research to be reproducible. As an extra contribution, a summary of the work done in this thesis has been written as a conference paper, which is found in Appendix H. The paper was submitted to The 2020 International Conference on Security and Management (SAM2020).[6] It has now been accepted for publication.

1.3 Thesis outline

The following structure is used in this thesis.

Chapter 1 - Introduction introduces the research topic by presenting the project motivation, research scope, and research questions.

Chapter 2 - Background and related work provides necessary theory about the research topic, such as information about container technology, the Docker ecosystem, and vulnerability categorization. The chapter also presents related research.

Chapter 3 - Methodology explains how this project was carried out by describing the research questions in more detail, the research design used, and the applied

[5] Repository containing all implemented software: *https://github.com/katrinewi/Docker-image-analyzing-tools*

[6] SAM2020: *http://sam.udmercy.edu/sam20/*

methodology structured as phases. It also presents the data set we were working with, as well as tools and resources used.

Chapter 4 - Implementation and data acquisition is also a part of the methodology. This chapter thoroughly describes each project phase and how they were conducted in order to obtain the results of our analysis. The first sections explain the implementation of software for data gathering, followed by the process of analyzing the data and visualizing the results.

Chapter 5 - Results presents all findings from the data analysis. First, the obtained data set is introduced. Then, the results are presented for each research question by dividing the results into areas of interest for answering the respective research question.

Chapter 6 - Discussion discusses the results in relation to the research questions, as well as limitations and validity of this study. Further, a comparison between our results and previous studies is presented, and we give our recommendations for future work.

Chapter 7 - Conclusion answers each research question of this thesis and presents the conclusions, including a summary of our research contribution.

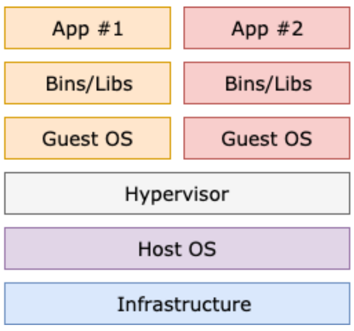
Chapter 2
Background and related work

To get an adequate understanding of the scope of this thesis, this chapter will present the relevant background needed to understand the research area. Container technology will first be presented and compared to virtual machines. Then, all relevant parts in the Docker ecosystem will be explained, followed by aspects related to the categorization method of vulnerabilities. Lastly, this chapter presents the related work previously done in this area of research.

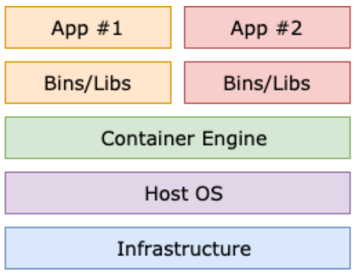
2.1 Virtual machines and container technology

Virtualization is the technique of creating a virtual abstraction of some resources to make multiple instances run isolated from each other on the same hardware [BK10]. There are different approaches to achieve virtualization. One approach is using Virtual Machines (VMs), see Figure 2.1a for VM architecture. A VM is a virtualization of the hardware at the host. Hence, each VM has its own kernel, and in order to manage the different VMs, a software called hypervisor is required. The hypervisor emulates the Central Processing Unit (CPU), storage, and Random-Access Memory (RAM), among others, for each VM. This allows multiple VMs to run as separate machines on a single physical machine.

In contrast to VMs, containers virtualize at the Operating System (OS) level, see Figure 2.1b. In essence, this means that every container running on the same machine share the same underlying kernel, where only bins, libraries and other run time components are executed for a single container exclusively. In short, a container is a standardized unit of software that contains all code and dependencies [Doce]. Thus, containers require less memory and achieve a higher level of portability than VMs. Container technology has simplified the software development process as the code is portable. Hence, what is run in the development environment will be the same as what is run in the production environment [And15].



(a) Virtual machine



(b) Container

Figure 2.1: Architecture of virtual machine and container

2.2 Docker overview

Docker is a container technology platform used to create, deploy, and run applications. The Docker ecosystem consists of several components that, as a whole, delivers a containerization service that is lightweight and offers an isolated and standardized computer environment for execution of applications. In essence, Docker is a capability extension of LXC. LXC is a method for virtualizing the OS and running multiple Linux containers on a single host using the Linux kernel [Arc20]. Docker is a container engine that uses the LXC, as well as the *namespaces* and the *cgroups* features of the Linux kernel to achieve isolation between processes. In short, Docker utilized already existing container technology.

With Docker, multiple components were introduced: a local daemon, a Representational State Transfer (REST) Application Programming Interface (API) for communication between the Docker Command Line Interface (CLI) client and the Docker daemon, an image specification standard, and registries for image distribution. By creating a lightweight and easy-to-use service, Docker has contributed to the rapid growth and usage of the container technology. Docker is written in the Go language and was released in 2013 as an open-source project. As of today, Docker Inc. is responsible for developing Docker.

An overview of the Docker ecosystem architecture is shown in Figure 2.2, where the colored arrows correspond to the color of the commands to the left. The Docker client is interacting with the Docker daemon at the host to run commands. The `docker build` command is called to build a Docker image from a Dockerfile (the purple arrows). When the `docker pull` command is run, the Docker daemon is interacting with the Docker registry to pull an image to the host (the yellow arrows). The image is executed as a container by running the `docker run` command (the green arrows). These concepts will be described in more detail in the next sections.

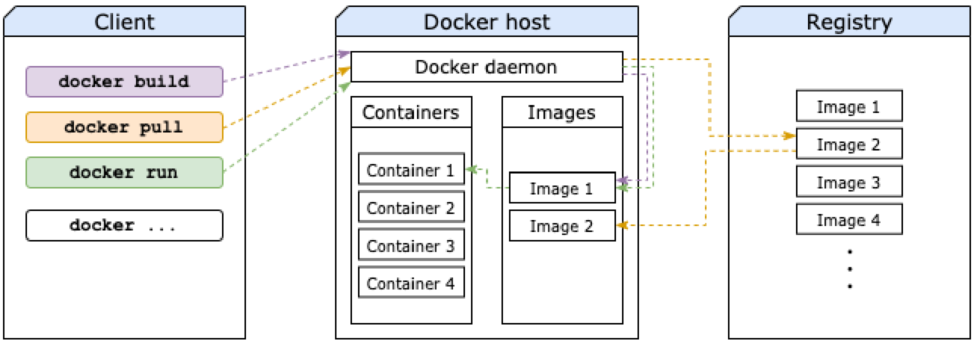


Figure 2.2: Docker ecosystem components. Inspired by figure found on [Docb].

2.2.1 Docker daemon

The Docker daemon is at the core of the Docker ecosystem. By interacting with the host OS, it is responsible for managing the containers, and performing tasks such as launching containers, controlling their isolation level, and monitoring them to trigger required actions. The Docker daemon is also interacting with remote registries to pull or push images, and performs the building of images [CMDP16]. It is by default running with root access on the host and communicates with the Docker client through a REST API.

2.2.2 Docker engine

The Docker engine is an application with server-client architecture. It contains three components: the Docker daemon, a REST API, and a CLI to let the user interact with the Docker daemon [Docb]. The CLI lets the user run commands such as `docker build` to build an image from a Dockerfile, `docker pull` to pull a specific version of an image from a registry and `docker run` to launch a container from an image. The relationship between these components is shown in Figure 2.3.

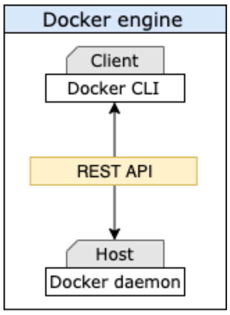


Figure 2.3: Docker engine architecture

2.2.3 Dockerfile

The Dockerfile is a text file containing specific build instructions used to create a Docker image automatically using the API. It contains all dependencies in a humanly readable format. The build process is done using the `docker build` command.

2.2.4 Docker image

A Docker image, from here on out referred to as an image, is created from a Dockerfile. It consists of different layers and metadata, where each command in the Dockerfile will create a new, separate layer. This layer-wise architecture makes it easy for different images to share the same layers between them, where each layer could be containing, for example, a component or a dependency [ZTA+19]. Only the top layer of an image is writable, and the other layers are read-only, see Figure 2.4. This means that the lower layers of the image are unchanged throughout the lifetime of the image. Each time a new container is started or changes to the image is done, a new writable top layer is added to the image. This is called the copy-on-write concept [And15]. In such a way, an image could be seen as a static snapshot of the Dockerfile at a specific time.

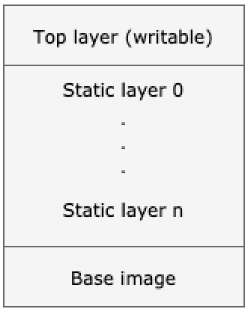


Figure 2.4: Docker image layers

2.2.5 Docker container

A Docker container is a running instance of an image, and hence, the execution environment of Docker. The namespaces and cgroups Linux kernel features are the fundamental concepts of containers, and thus also of Docker containers. Namespaces permits *isolation* between processes. At a conceptual level, it is organizing relevant elements into groups based on identifiers [Mic17]. This way, elements in such a group will only see other elements in that same group. Moreover, the cgroups, or control groups, is what allows for dedicating, limiting, and isolating resources between processes.

2.2.6 Docker registries

To make it easy to store and distribute images between users, Docker introduced Docker registries. Simply put, a Docker registry is a storing and content delivering system for Docker images [Doca]. Images are organized into repositories, where each repository hosts every version of an image with corresponding tags. For example, most images have the *latest tag*, which corresponds to the newest version of the image. As of today, Docker Hub is the most popular Docker registry and is also the default registry that the Docker daemon interacts with. Using the API, users can pull images to download them locally and push images to upload them to the Docker Hub. This is done by using the commands, `docker pull <image>:<tag>` and `docker push <image>:<tag>` , respectively.

On Docker Hub, image repositories are divided into different categories, or types. Repositories are either private or public, and can further be either *official*, *community* or a *verified* repository. Also, repositories can be *certified*, which is a subsection of the verified category. The official image repositories are maintained and reviewed by Docker. The verified ones are reviewed by Docker, but developed by third-party developers. Besides being verified, certified images are also fulfilling additional requirements related to quality, support, and best practices [Mor18]. Community images could be uploaded and maintained by anyone. The distribution of the image repository types on Docker Hub can be seen in Table 2.1, where the numbers were gathered on February 3rd, 2020. The community repository category is by far the most dominant one and makes up ~99% of all Docker Hub repositories.

The different categories of Docker repositories were introduced at different times. The official repositories were included when Docker Hub was introduced in 2014 [Gol14], and verified and certified repositories were included in 2018 when Docker Store and Docker Cloud were merged into Docker Hub [Mor18]. In addition to image type, the image repositories on Docker Hub have additional features, such as the number of times they have been pulled (downloaded), and the date the image was last updated. These can be used as measures to see how popular image repositories are and how frequently they are maintained. Docker Hub also supports a starring system, where users can give stars to image repositories that they like and is often used as a way to bookmark repositories.

2.3 Vulnerability database and categorization method

A vulnerability is defined as a flaw within a computer system that could potentially be exploited and result in unauthorized actions to be performed so that the system is compromised [Nat20a]. The severity of a vulnerability depends on a variety of

Repository type	**Number of images**
Official	160
Verified	250
Certified	51
Community	3,064,454
Total	3,064,915

Table 2.1: Distribution of repository types on Docker Hub (February 3rd, 2020)

variables, and it is highly complex to compare vulnerabilities due to the diversity of different technologies and solutions. Already in 1997, National Vulnerability Database (NVD) started working on a database that would contain publicly known software vulnerabilities to provide a means of understanding future trends and current patterns [ZCO11]. The database is useful in security management when deciding what software is safe to use, and predicting whether or not software contains vulnerabilities that have not yet been discovered.[1]

2.3.1 Common Vulnerabilities and Exposures (CVE)

NVD contains Common Vulnerabilities and Exposures (CVE) entries, and provides details about each vulnerability, like vulnerability overview, Common Vulnerability Scoring System (CVSS) score, references, Common Platform Enumeration (CPE) and Common Weakness Enumeration (CWE) [NKK17].

CVE is widely used as a method for referencing vulnerabilities that are publicly known in released software packages. At the time of writing, it exists over 130,000 entries in the CVE list.[2] The CVE list was created by MITRE Corporation in 1999, whose role is to manage and maintain the list.[3] They work as a neutral and unbiased part in order to serve in the interest of the public. Examples of vulnerabilities found in the CVE list are common errors, faults, flaws, and loopholes that can be exploited by a malicious user to get unauthorized access to a system or server. The loopholes can also be used as propagation channels for viruses and worms that contain malicious software [CZC09]. Over the years, CVE has become a recognized building block for various vulnerability analysis and security information exchange systems, and is considered as the industry standard. This is much because it is

[1]NVD website: *https://nvd.nist.gov*

[2]The number of entries in the CVE list was retrieved on January 28th, 2020 from the official website: *https://cve.mitre.org*

[3]MITRE Corporation is a non-profit US organization with the vision to resolve problems for a safer world: *https://www.mitre.org*

continuously maintained and updated, and because the information is stored with accurate enumeration and orderly naming.

There are three parts that constitute a CVE entry; CVE ID number, description, and references [MIT19]. The CVE ID number has the following structure: CVE-YYYY-NNNNN, for example, CVE-2020-12345. All CVE IDs start with the CVE prefix, followed by the year and a sequence number. The year does not indicate when the vulnerability was discovered; it indicates the former of either the year the CVE ID was assigned, or the year the vulnerability was made public. As of January 1st, 2014, the sequence number can be four or more digits. The original syntax from 1999 only allowed four digits, limiting the number of vulnerabilities to be uniquely identified each year to 9,999. This would not be sufficient today due to the rapid growth of the annual number of reported vulnerabilities [MIT19].

The description part of the CVE should be unique to each vulnerability, and is written by CVE Numbering Authorities (CNAs), the CVE Team or an individual wanting to create a new CVE ID [MIT19]. The description contains information like the specification of the software that is affected (including the products, vendors, and affected versions), the vulnerability type, the possible consequences of the vulnerability, and a description of how an attacker might exploit the vulnerability. However, not all descriptions include all necessary details. The reason is that it has to be reviewed by the CVE Team before publication, and they are only allowed to access publicly available information. Therefore, it might exist details about some vulnerabilities that cannot be officially published in the CVE entry.

The last part of the CVE entry is a list of references where additional information about the vulnerability, and related software, products, and technology can be found, like vulnerability reports, advisories, and other sources.

2.3.2 Common Vulnerability Scoring System (CVSS)

As mentioned in the previous section, the entries in the NVD database include a CVSS score. This is a numerical score indicating the severity of the vulnerability on a scale from 0 to 10, based on a variety of metrics. The following sections that describe CVSS are based on the CVSS v3.1 specification document.[4] The CVSS metrics are divided into three metric groups: the Base Metric Group, the Temporal Metric Group, and the Environmental Metric Group. A *Base Score* is calculated by the metrics in the Base Metric Group, and is independent of the user environment and does not change over time. The Temporal Metrics take in the base score and adjusts it according to factors that do change over time, such as the availability of exploit code. Environmental Metrics adjust the score yet again, based on the type

[4]See the CVSS v3.1 specification document: *https://www.first.org/cvss/specification-document*

of computing environment. This allows organizations to adjust the score related to their IT assets, taking into account existing mitigations and security measures that are already in place in the organization.

In our analysis, it would not be practical to take into account the Temporal or Environmental Metrics as we want to discuss the vulnerability landscape independently of the exact time and environment. Therefore, only the Base Metric group will be described in more detail. It is composed of two sets of metrics: the Exploitability metrics and the Impact metrics, as can be seen in Figure 2.5. The first set takes into account *how* the vulnerable component can be exploited, and includes attack vector and complexity, what privileges are required to perform the attack, and whether or not user interaction is required. The latter set reflects on the *consequence* of a successful exploit and what impact it has on the confidentiality, integrity, and availability of the system.[5] The last metric is *scope*, which considers if the vulnerability can propagate outside the current security scope.

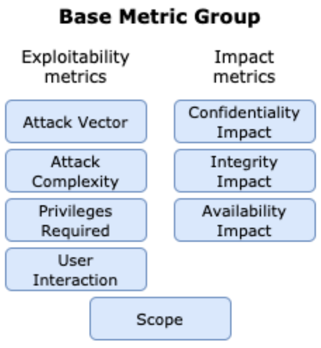


Figure 2.5: CVSS Base Metric Group. Inspired by figure from [FIR19].

When the Base Score of a vulnerability is calculated, the eight different metrics from Figure 2.5 are considered. Each metric is assigned one out of two to four different values that are used to generate a vector string. The vector string is then used to calculate the CVSS score, which is a numerical value between 0 and 10. In many cases, the numerical value is mapped to a textual value for convenience, where the severity is categorized as either critical, high, medium, low, or none, as can be seen in Table 2.2.

[5]Known as the CIA triad

Rating	**CVSS score**
None	0.0
Low	0.1 - 3.9
Medium	4.0 - 6.9
High	7.0 - 8.9
Critical	9.0 - 10.0

Table 2.2: Mapping between severity rating and CVSS score [FIR19]

The above sections are based on CVSS v3.1, which was introduced in June 2019. The second version of CVSS was widely used at its time; however, there was a definite need for improvement. One of the main issues about CVSS v2.0 was that it lacked granularity of metrics such that vendors experienced the CVSS v2.0 score as inaccurate. Therefore, the new version did several changes on the metrics used to calculate the CVSS score, as well as adding more severity categories, and changing the ranges of them. More specifically, CVSS v2.0 only contains the ratings low (0.0-3.9), medium (4.0-6.9) and high (7.0-10.0) [Nat20b], whereas CVSS v3.x contains the ratings as already described as already described in Table 2.2.

2.4 Related work

This section will summarize the most relevant research previously done concerning the vulnerability landscape of Docker Hub images.

One of the first to explore the vulnerability landscape of Docker Hub was BanyanOps [GDT15]. In 2015, they published a technical report revealing that 36% of official images on Docker Hub contained high priority vulnerabilities [GDT15]. Further, they discovered that this number increased to 40% when community images (or general images as they call it in the report) were analyzed. BanyanOps built their own vulnerability scanner based on CVE scores, and analyzed all official images (~75 repositories with ~960 unique images) and 1700 randomly chosen community images. However, at that time, Docker Hub only consisted of ~95,000 images. As of now, Docker Hub hosts over 3.5 million images, which has been a massive increase since the time of their analysis.

In 2017, Shu et al. published a new vulnerability analysis of Docker Hub images [SGE17]. With the aim of revealing the Docker Hub vulnerability landscape, they created their own analysis framework called DIVA (Docker Image Vulnerability Analysis). The DIVA framework discovers, downloads, and analyses official and community images. It is based on the Clair scanner and uses random search strings to discover images on Docker Hub. In total, they analyzed 356,218 unique images.

The analysis revealed that an image (official and community) on average contained more than 180 vulnerabilities. They also found that many images had not been updated for hundreds of days, which is problematic from a security point of view. Further, it was observed that vulnerabilities propagate from parent to child images.

To our knowledge, the most recent vulnerability analysis of Docker Hub images was performed during spring 2019 by Socchi and Luu [SL19]. They investigated whether the security measures introduced by Docker Inc. (more precisely, the introduction of verified and certified image types) improved the security of Docker Hub. In addition, they inspected the distribution of vulnerabilities across repository types, and whether vulnerabilities are still inherited from parent to child images. To perform their analysis, they implemented their own analyzing software using the Clair scanner. They used the results from Shu et al. [SGE17] from 2017 as a comparison. The data set they successfully analyzed consisted of 757 images in total. Whereas 128 were official, 500 were community, 98 were verified, and 31 were certified. They only analyzed the most recent image in each repository and skipped all Microsoft repositories. The thesis concludes that the security measures introduced by Docker Inc. did not improve the overall Docker Hub security. They stated that the number of inherited vulnerabilities had dropped since the analysis of Shu et al.; however, they also found that the average number of new vulnerabilities in child images had highly increased. Moreover, they found that the majority of official, community, and certified repositories contain less than 75 vulnerabilities and that the majority of verified images contain less than 180 vulnerabilities.

Chapter 3 Methodology

In this chapter, the research questions will first be described in detail, and then the requirements for answering them will be pointed out. The specific choice of research design is stated and argued for, and the applied methodology is presented. Figures will be used to explain all project phases and the data flow explicitly. Further, the obtained data set is introduced, and all the required tools and resources are presented.

3.1 Research questions

Our research questions, as defined in Section 1.2.1, formed the basis of this thesis. Several aspects need to be considered to answer each research question sufficiently.

RQ1 *How can vulnerabilities found in Docker images be systemized in order to investigate the current vulnerability landscape of Docker Hub?* is considered as our main research question, and of the most importance. In order to answer this, we first investigated the number of vulnerabilities in each severity category (critical, high, medium, low, negligible, and unknown). Then, we looked at the central tendency (average and median) and other statistical measures describing the data set, such as maximum, minimum, and standard deviation of the number of vulnerabilities in images. We examined how many vulnerabilities that exist in each of the four image types (official, verified, certified, and community), as well as the density distribution of the number of vulnerabilities for each image type. Next, we determined what images that contain the most critical vulnerabilities. We also examined the percentage of images that contain high and critical vulnerabilities, and we determined how vulnerable Microsoft images are compared to other images, as proposed by Socchi and Luu in their thesis' future work section [SL19]. Moreover, the number of images that do not contain any vulnerabilities was inspected. Then, we investigated what the CVE trend is like in all image types, as well as in the number of newly discovered CVE vulnerabilities each year. The final discovery used to answer **RQ1** was how often images on Docker Hub are updated.

When considering **RQ2** *How do image features and the number of vulnerabilities correlate in images?*, we applied Spearman's rank correlation to see whether or not the number of vulnerabilities is affected by features of the images we gathered. The features that were evaluated are the number of pulls, the number of stars, and the number of days since the last update. We also inspected scatter plots to identify other relationships in the data than correlation.

To comply with **RQ3** *Which types of vulnerabilities are the most severe?*, we investigated the most threatening vulnerabilities by looking at the most represented critical vulnerabilities in our data set. Further, we took a more in depth look at the most represented ones by determining what characterizes them and describing their common features.

Finally, the most vulnerable packages in images were identified to answer **RQ4** *Which packages contain the most severe vulnerabilities?*. We determined what packages contain the most critical vulnerabilities and looked at the number of images that use the most vulnerable packages. Lastly, we found out how vulnerable the most used packages on Docker Hub are.

3.2 Research design

This thesis follows a quantitative research design. Quantitative research is a type of research that involves collecting and analyzing data in order to describe the behavior and trends reflected in the data [Suk96]. The data, which is represented as numeric values, has been analyzed by using mathematical and statistical techniques. These will be elaborated on later in this chapter. The results are presented using tables, graphs, and other visuals. The goal of our analysis is to answer the research questions, as explicitly explained in the previous section (Section 3.1). Out of the different categories of quantitative research, our project falls under the descriptive research category and the correlational research category. These terms are further explained in the next section.

3.2.1 Quantitative descriptive and correlational research

All our research questions are related to quantitative research; however, there are differences in the applied research type. **RQ1**, **RQ2** and **RQ4** falls under the category of descriptive research, while **RQ3** is classified as correlational research.

The objective of descriptive research is to describe how something is, rather than to determine the cause and effect. The core of descriptive research is, according to Fox and Bayat, to *describe* what a situation is like through a process of collecting data. It further aims at bringing attention to problems or issues in a more complete way than what was possible without employing this method [FB08]. On the other hand, correlational research focuses on the relationship between two or more variables, without controlling them [McC20]. The goal of correlational research is, thus, to determine the correlation between two or more variables.

3.2.2 Project phases

The methodology applied in this project has enabled us to carry out the work through a systematic approach so that the obtained results were used to answer our research questions successfully. The project was divided into five phases, which are based on the use of quantitative descriptive and correlational research, as already defined. The phases are stated below, and a comprehensive explanation of each phase is given in Chapter 4 Implementation and data acquisition.

Phase 1: Implementation of scripts
Phase 2: Data collection
Phase 3: Import data into database
Phase 4: Data analysis
Phase 5: Visualization of results

The workflow of the project phases is visualized in Figure 3.1 for clarity. As seen in the figure, we carried out the implementation phase in an iterative approach (Phase 1), where we tested the scripts on a subset of the data and implemented necessary changes. We performed the other phases in succession. After the scripts were working as required, the process of collecting the data was started (Phase 2). Then, we imported the collected data into the database (Phase 3), and conducted the data analysis on the data in Phase 4. Lastly, we visualized the results in Phase 5.

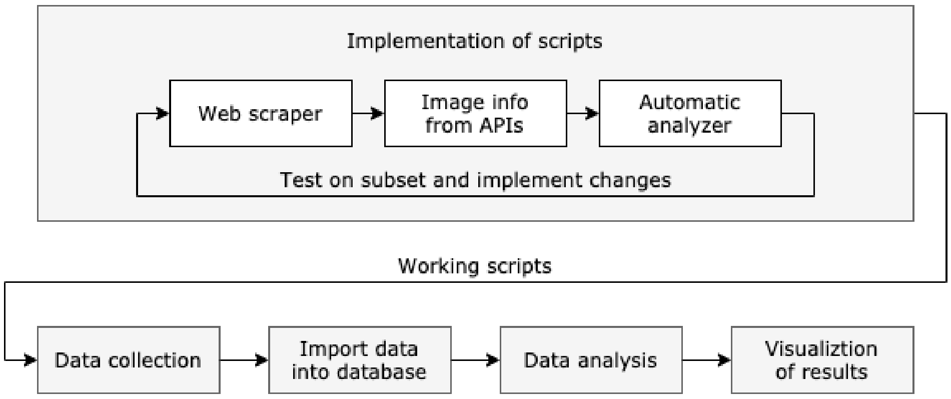


Figure 3.1: Project phases

Figure 3.2 shows the different data sources and how the data traverses the various components. The web scraper utilizes tools specified in Section 3.4.1 and collects 2540 images from Docker Hub. We use the collected image names as input in both API scripts, and in the automatic analyzer. The collected image information is inserted into the MySQL database (see Section 3.4.6). The API scripts gather image metadata from the Docker Registry web APIs (see Sections 3.4.2 and 3.4.3), which is inserted into the database. The automatic analyzer analyzes all images using the Anchore Engine vulnerability scanner (see Section 3.4.4), and returns the results in Comma Seperated Values (CSV) format (see Section 3.4.7). We insert the resulting vulnerability data and the information about failed images into the database. SQL queries are run on the data and the outputted results are then visualized by using the Matplotlib and Seaborn Python libraries (see Section 3.4.5). The machine environment described in Section 3.4.9 was used throughout the project, and the final results are presented in Chapter 5 Results.

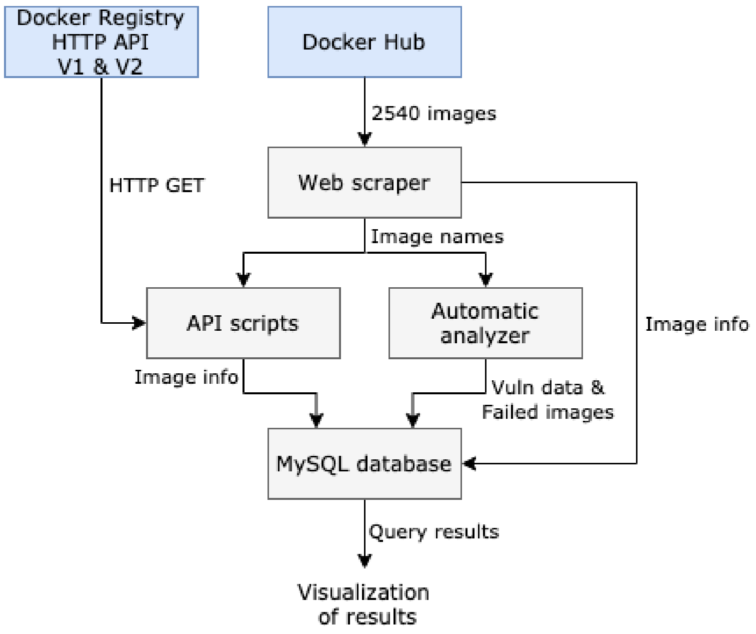


Figure 3.2: Data flow

3.3 Data set

The data set used in this analysis consisted of 2540 images in total, where 2412 were successfully analyzed, and 128 failed. Details regarding the data set can be seen in Table 3.1. We included all official, and a portion of the verified and certified images on Docker Hub in this analysis, in addition to the ~2000 most popular community

Image type	**Successful count**	**Failed count**	**Total count**
Official	157	3	160
Verified	60	57	117
Certified	22	27	49
Community	2,173	41	2,214
Total	2,412	128	2,540

Table 3.1: Distribution of successful and failed images in each image type

images. Due to time constraints, it was decided to analyze one image in each analyzed repository. Images with the latest tag were preferred, as this is the newest and most patched version of the image. Therefore, vulnerabilities found in these versions are more significant because vulnerabilities in previous versions may already be fixed in the latest version. We see it as a better approach to analyze one image from many repositories instead of analyzing all images in a smaller number of repositories. This is because it is likely that there are similarities in the vulnerabilities present in images from the same repository, and choosing to analyze a larger variety of repositories will give a broader perspective of what the vulnerability landscape actually is like. We present the database structure and all data attributes in our data set in Section 4.3.

3.4 Tools and resources

We have applied a variety of tools and resources in this thesis. This section will present and describe each one.

3.4.1 Selenium and Geckodriver

Selenium and Geckodriver are tools that were utilized in the web scraper to navigate around on the Docker Hub web page to gather the desired images and their image repository type. Selenium is a testing tool for web applications that support Python through a Python API. Geckodriver is a web browser engine for the Firefox web browser that allows loading and navigating on web pages remotely. Geckodriver is what connects Selenium and the Firefox browser, and thus, it is a necessity in order to make Selenium tests run in the Firefox browser.

3.4.2 Docker Registry HTTP API V1

The Docker Registry API V1 is a REST API which returns JSON data and accepts HTTP requests. The API interacts with the Docker engine and makes image distribution possible [Docd]. Docker Inc. has announced that pushing and pulling to the first version of the API has been deprecated in order to migrate to the second version of the API [San19]. Indeed, it is still possible to use the API. We have used this API to gather metadata about verified and certified images. The following HTTP request shows how to interact with the API to gather metadata about images, such as the number of pulls, the number of stars, and when the image was last updated. Note that our scripts access the web APIs.

```
GET https://<registry-URL>/api/content/v1/products/images/
↪ <repository-name>
```

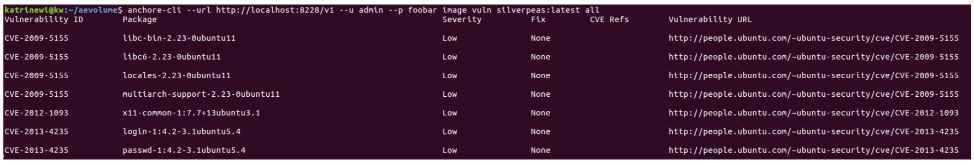
3.4.3 Docker Registry HTTP API V2

The second version of the API was released with the introduction of the Docker Registry 2.0 and is an updated implementation of the first version. This API is similar to the first version, but with some architectural changes. In addition, it has enhancements when it comes to performance and security [CSR17]. This version of the API gives access to official and community images and is used to gather data about them. The HTTP request for retrieving image data is shown below.

```
GET https://<registry-URL>/v2/repositories/<repository-name>
```

3.4.4 Anchore Engine

Out of many currently available container scanners, Anchore Engine is one of the most favored ones in the community.[1] It is an open-source static scanning tool that offers a lot of functionality for inspecting and analyzing container images. Anchore Engine can be used as a stand-alone program, or within another orchestration platform [Off19]. It is either accessed by its CLI, or directly through a REST API. Anchore Engine also allows for custom user-defined policies to be determined in terms of what to accept or reject. A screenshot of the output of a vulnerability scanning with Anchore Engine is shown in Figure 3.3.



```
katrinewi@kw:~/aevolume$ anchore-cli --url http://localhost:8228/v1 --u admin --p foobar image vuln silverpeas:latest all
Vulnerability ID        Package                                         Severity        Fix         CVE Refs        Vulnerability URL

CVE-2009-5155           libc-bin-2.23-0ubuntu11                         Low             None                        http://people.ubuntu.com/~ubuntu-security/cve/CVE-2009-5155

CVE-2009-5155           libc6-2.23-0ubuntu11                            Low             None                        http://people.ubuntu.com/~ubuntu-security/cve/CVE-2009-5155

CVE-2009-5155           locales-2.23-0ubuntu11                          Low             None                        http://people.ubuntu.com/~ubuntu-security/cve/CVE-2009-5155

CVE-2009-5155           multiarch-support-2.23-0ubuntu11                Low             None                        http://people.ubuntu.com/~ubuntu-security/cve/CVE-2009-5155

CVE-2012-1093           x11-common-1:7.7+13ubuntu3.1                    Low             None                        http://people.ubuntu.com/~ubuntu-security/cve/CVE-2012-1093

CVE-2013-4235           login-1:4.2-3.1ubuntu5.4                        Low             None                        http://people.ubuntu.com/~ubuntu-security/cve/CVE-2013-4235

CVE-2013-4235           passwd-1:4.2-3.1ubuntu5.4                       Low             None                        http://people.ubuntu.com/~ubuntu-security/cve/CVE-2013-4235
```

Figure 3.3: Anchore Engine output

Anchore Engine gathers vulnerability information from multiple sources such as RedHat, Debian, and NIST, as illustrated in Figure 3.4. The Feed Service is responsible for collecting the vulnerability data and normalize the data. Anchore

[1]For Anchore Engine source code, see *https://github.com/anchore/anchore-engine*

Engine is periodically fetching vulnerability data from the Feed Service and then saves it into its PostgreSQL database (Anchore Database in the figure) to always stay up-to-date [Hil19]. The vulnerability scanning works by checking image layers up against this vulnerability database. Each found vulnerability is in one of the following severity categories: critical, high, medium, low, negligible, unknown, or no severity. Anchore Engine matches vendor packages with vendor vulnerability records (e.g., Debian packages will map to Debian vulnerability data and RPM packages will map to RedHat vulnerability data), and then draws the severity level from the matched vulnerability record. If a package is not a vendor package (e.g., npm, ruby, pip and java), then the match is made up against NVD data. The severity is again drawn from the vulnerability record that is part of the match. The negligible and unknown categories occur when the vendor has not yet made a decision on the vulnerability and labels the vulnerability as "not yet assigned".

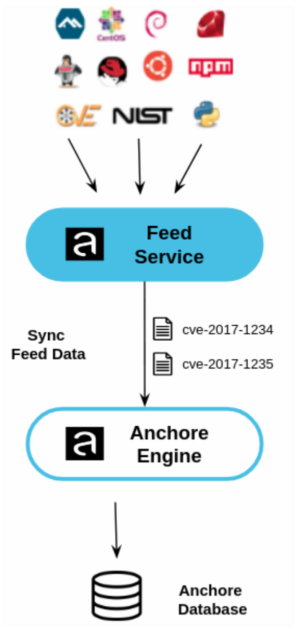


Figure 3.4: Anchore Engine vulnerability data gathering. Figure from [Hil19].

It is trivial that the results from a vulnerability scanning will depend highly on what scanner is used. Both Shu et al. [SGE17] and Socchi and Luu [SL19] used the Clair scanner (see Section 2.4 for related work). As we do not intend to replicate previous research, but rather give a new perspective on the vulnerability landscape in Docker Hub images, we have chosen a *different*, but still a well-respected scanner in the community.

3.4.5 Matplotlib and Seaborn

We used the Python plotting library Matplotlib to generate graphs, bars, and charts so that we could present our results in a transparent way. The Seaborn library is another visualization library based on Matplotlib, which we used when Matplotlib was not sufficient. Matplotlib provides a lot of functionality and makes it possible to conduct changes at a low detail level compared to other considered plotting tools.

3.4.6 MySQL

MySQL is a database system owned by Oracle Corporation that is based on the Structured Query Language (SQL). As our data set is considerably large, it was beneficial to use queries to retrieve results quickly. Thus, the choice was to go for a SQL database, which is also called a Relational Database Management System (RDBMS). A RDBMS is a database where the values in a table are related to each other, and where tables are related to each other through primary and foreign keys. The MySQL database system is one of the most commonly used database systems, and besides, it is open-source and free. Hence, it was a natural choice for this project. MySQL is versatile and can be used with most operating systems, applications, and programming languages. The MySQL database server is well suited for large databases because it is easy to use, very fast and reliable, and also scalable [Ora20a].

There are multiple options for ways of working with MySQL databases. It can be done directly in the machine terminal or using most programming languages, like Python, Java, C, and C++, via different APIs. Another option, which is what we opted for, is MySQL Workbench. It is a Graphical User Interface (GUI) application that provides a graphical tool for communicating with and managing MySQL servers and databases [Ora20b]. In this thesis, we used MySQL Workbench for creating tables in the database, importing data into them, join tables into new ones, and doing queries on the data. It was the option we found the most intuitive, easy to set up, easy to work with, and it provided the required functionality.

3.4.7 CSV file format

The CSV file format is a text file where commas separate the values. Every line in the file constitutes a data record. This file format is commonly used for data exchange between applications, and most databases support it. Due to its seamless integration with the MySQL database system and low overhead, we chose it as the format for saving the gathered data before importing it into the database.

3.4.8 Statistical concepts

As a part of our research design that was described in Section 3.2, we have utilized several mathematical and statistical concepts when analyzing our data. The following sections give a brief description of each one.

Average: The average measure summarizes all values and divide the sum by the total number of values, see Equation 3.1. It was used as a measure of central tendency, for example, when calculating the average number of vulnerabilities in each image.

$$\overline{x} = \frac{1}{N}\sum_{i=1}^{N} x_i \tag{3.1}$$

Median: The median is the middle value in a set where all values are sorted. In cases where the data set has extreme values, the median will be more representative in terms of the central tendency. In our project results, we used the median as another measure to determine the central tendency related to the vulnerability count in images. Formally speaking, the median of a list $\mathbf{x} = \{x_1, \ldots, x_n\}$ that consists of n elements can be seen in Equation 3.2:

$$median(\mathbf{x}) = \frac{\texttt{sorted}(\mathbf{x})_{\lfloor \frac{n+1}{2} \rfloor} + \texttt{sorted}(\mathbf{x})_{\lceil \frac{n+1}{2} \rceil}}{2} \tag{3.2}$$

$\texttt{sorted}(\mathbf{x})$ is a sorted list of the values of the list $\mathbf{x}$, and the notations $\lfloor \frac{n+1}{2} \rfloor$ and $\lceil \frac{n+1}{2} \rceil$ denote the middle index (or the average value of the middle two indexes) of a list of n elements.

Standard deviation: The standard deviation is used to investigate the dispersion of a data set by calculating the average distance to the average value of the data set. A high standard deviation value indicates that the data is varied and spread over a large interval, while a low standard deviation indicates that most values are close to the average. See the mathematical formula for sample standard deviation in Equation 3.3.

$$s = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2} \tag{3.3}$$

Pearson correlation coefficient: Correlation is used to determine how strong two variables are related to each other. The Pearson Correlation Coefficient (PCC) is a number between -1 and 1. A positive value indicates that there is a positive linear correlation, 0 indicates no correlation, and a negative value means that it is a negative linear correlation. A positive correlation means that both variables move in the same direction, and a negative correlation indicates that they move in the opposite direction. PCC should be used on normally distributed data. The PCC formula for a sample is stated below in Equation 3.4, where x and y are two variables (for example, the vulnerability count in images and image star count). $\overline{x}$ and $\overline{y}$ are the average values for these variables.

$$r_{xy} = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \overline{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \overline{y})^2}} \tag{3.4}$$

Spearman's correlation coefficient: In cases where the data set is not related through a linear relationship, the Spearman's correlation coefficient r_s can be used to investigate if there is a monotonic relationship between values [LOHS05]. A positive monotonic relationship means that the values have an increasing relationship, while a negative monotonic relationship means the opposite. Just like the PCC, the correlation coefficient is a value between -1 and 1, where -1 represents a perfect negative correlation, 0 means no correlation, and 1 means perfect positive correlation. The formula for Spearman's correlation is similar to Pearson's; however, before using Equation 3.4 on the data, the relative rank of the data is calculated. In statistics, this approach is normally used when the data is not following a distribution, such as a normal distribution. The rank is calculated by ordering the data and assign them ranked values from 1 to n, where n is the number of values. For cases where multiple values are identical, the average ranking of the values will be used. For example, a data set with distinct values, such as [2, 16, 27, 28, 42], will be ranked [1, 2, 3, 4, 5]. A data set with multiple identical values, such as [2, 2, 6, 6, 6], will be ranked [1.5, 1.5, 4, 4, 4].

Density distribution plot: To see how a variable is distributed in a data set, a density distribution plot can be used. It displays all different values along the x-axis and the probability density along the y-axis. The probability density for a value is the probability that a random value has this exact value. In the case where the data is distinct, for example when looking at the number of vulnerabilities in each image, the density plot needs to be estimated. This will be done by plotting a histogram with the data values, and then use Kernel Density Estimation (KDE) on the data. KDE will smooth out any outliers and extreme values, and give a more even representation of the data set. We used this approach when investigating the density distribution for number of vulnerabilities in each image type.

3.4.9 Machine specification

To perform our analysis, we have used a machine with the following specifications:

– Operating system: Ubuntu 18.04.3 LTS

– CPU: Intel® Core™ i7-8700 CPU @ 3.20GHz $\times$ 12

– RAM: 31.2 GiB

– Storage: 245.1 GB

– OS-type: 64-bit

Chapter 4 Implementation and data acquisition

In this chapter, all project phases, as presented in Section 3.2.2, will be explained in detail. First, the implementation phase is described by presenting all implemented scripts at a conceptual level. Four scripts were implemented to achieve our objectives: a web scraper, two scripts for accessing the Docker Hub APIs, and a script for automating the image analysis process. Then, the rest of the project phases is be presented consecutively. These phases consist of the data collection phase, the import data into database phase, the data analysis phase, and the visualization of results phase. This chapter will also comment on factors that limit the project scope.

4.1 Implemented scripts

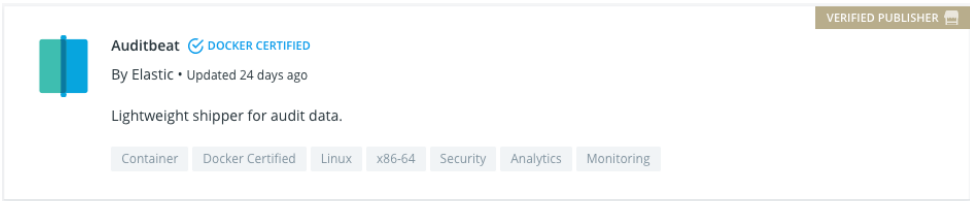
We aim to do an independent analysis of Docker images and not to reproduce previous research. Thus, we have implemented four scripts to perform our analysis. These will be explained in this section, as well as limiting factors. We present the prerequisites needed for our scripts to work are in Appendix A. Our implemented scripts are also provided in a public repository on GitHub: *https://github.com/katrinewi/Docker-image-analyzing-tools*

4.1.1 Web scraper

We found web scraping as the appropriate method to gather images because the APIs (Sections 3.4.2 and 3.4.3) provided by Docker Inc. have shortcomings when it comes to consistency, documentation, and ease of use. Simply put, we see it as beneficial to gather all images from a single source, which would not be possible by using the APIs, and also, to only gather the images that appear as the most popular on the Docker Hub web page. However, we want to mention that the APIs were used to gather additional information about the images by executing independent calls to the APIs, which will be explained later in this chapter.

We implemented a scraper in Python 3, which can be seen in Appendix B. The actual implementation happened in multiple iterations, where the code was tested at a small scale before the required changes were done. Since the HTML code on the Docker Hub web page is rendered in the browser using JavaScript code, an approach where the web page was loaded before attempting to scrape it had to be used. To handle this, two different tools were utilized: Selenium and Geckodriver. These tools are explained explicitly in Section 3.4.1.

The script opens the URL of the Docker Hub explore page, and extracts the name of each image along with what type of image repository it belongs to (official, verified, certified or community). The script iterates through all pages by incrementing the page count from 1 to 100 and extracts image information about each image on the page. The reason why the script only iterates through 100 pages is that Docker Hub does only allow navigation to page 100 (as a sidenote, it is not possible to access any more images using the APIs). There are 25 images per page, which makes the total number of repositories accessible through navigation 2500. The script outputs two files: one text file with all image names separated with a line shift and one CSV-file that contains image name and image type for each image. See Figure 3.2 for the general data flow. It needs to be pointed out that all certified images on Docker Hub are also considered as verified images because the certified image type is a subgroup of the verified type, see Figure 4.1 for an example. To clarify, images in the certified category will only be considered as certified images in our database, not as verified. To summarize, the goal of the scraper is to gather the 2500 most popular images along with their image types from Docker Hub.


Auditbeat DOCKER CERTIFIED

VERIFIED PUBLISHER

By Elastic • Updated 24 days ago

Lightweight shipper for audit data.

Container | Docker Certified | Linux | x86-64 | Security | Analytics | Monitoring

Figure 4.1: Screenshot from Docker Hub of the Auditbeat image. Note that it is marked as both a certified and a verified image.

Limitations

It would have been desirable to analyze even more images, but as already explained, Docker Hub does not allow navigation any further than page 100. A solution similar to Shu et al.[SGE17] could have been used, where they search for random strings with lengths between 1 to 20 characters to discover images. However, it was left as future work.

4.1.2 Scripts for accessing the Docker Registry API's

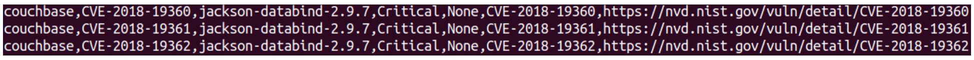
The image metadata that is possible to extract from scraping the Docker Hub web page is inadequate. Hence, there was a need to access the Docker Registry APIs for additional information about each image. We implemented two Python scripts for accessing the first and second versions of the Docker Registry API (see Sections 3.4.2 and 3.4.3, respectively). Both APIs were used because the Docker Registry API V2 only gives access to official and community images. In order to retrieve metadata about verified and certified images, the first version of the API was required. The APIs make it possible to gather information such as the exact number of image pulls, the number of stars, and the date the image was last updated. However, it should be mentioned that the verified and certified images don't contain star-ratings, and many of them lack data about number of pulls. The scripts for accessing the Docker Registry API V1 and V2 can be seen in Appendix C and Appendix D, respectively. The two scripts are similar, so only the script that is accessing the Docker Registry API V2 will be explained in detail. The outputted files from the two scripts were merged into two new files with information from both APIs. The CSV file with image information from both APIs was used in the next phases with the format: image name, pulls, stars, and last update.

The script (API V2) accesses two different URLs depending on the image repository type. If it is a library, the URL *https://hub.docker.com/v2/repositories/library/image-name* has to be used, otherwise the URL *https://hub.docker.com/v2/repositories/image-name* is used. The script takes a txt-file with image names separated with newline as input. The output of the script is a CSV-file with image information as specified in the previous paragraph. Additionally, it outputs a text-file with the images that could not be found.

4.1.3 Automatic analyzer

With the aim of running the analysis automatically, we have implemented a bash script to perform the analysis with the Anchore Engine scanner (see Section 3.4.4 to read about Anchore Engine). The actual code can be further inspected in Appendix E.

The script takes the outputted CSV file from the web scraper that contains all image names as input (see Figure 3.2 to see the general data flow in this project). It iterates through the names and tries to start the Anchore Engine scanning on each image. In some cases, images fail to be analyzed for different reasons. These are identified and written to a separate file that contains only the failed images. Then, if the image scanning is successful, the script checks whether the image contains any vulnerabilities. If it does, these are written to the vulnerability file in the appropriate format for CSV. If the image did not contain any vulnerabilities, it moves on to the next image. There are two output files: one CSV file containing information about all found vulnerabilities, and one CSV file with the images that failed to be analyzed and the failure reason. The format of the vulnerability file is: image name, vulnerability name, package, severity level, fix, CVE-reference and vulnerability URL. An excerpt from the vulnerability file can be seen in Figure 4.2. The format of the failed file is: image name, cause of failure.



```
couchbase,CVE-2018-19360,jackson-databind-2.9.7,Critical,None,CVE-2018-19360,https://nvd.nist.gov/vuln/detail/CVE-2018-19360
couchbase,CVE-2018-19361,jackson-databind-2.9.7,Critical,None,CVE-2018-19361,https://nvd.nist.gov/vuln/detail/CVE-2018-19361
couchbase,CVE-2018-19362,jackson-databind-2.9.7,Critical,None,CVE-2018-19362,https://nvd.nist.gov/vuln/detail/CVE-2018-19362
```

Figure 4.2: Content vulnerability file

In order to perform the scanning of images, we used some of the commands provided by the Anchore Engine CLI in our bash script. Note that these commands in the actual script (Appendix E) contain some extra command line parameters. These are needed to authenticate and connect the Anchore CLI to the Anchore Engine to be able to run the commands. However, they are not needed if the parameters are stored as environment variables instead, and thus, for readability, they are excluded from the following code examples. The relevant commands are listed below, where the <image> attribute indicates the image name.

– `$ anchore-cli image add <image>:latest`
  To pull an image with the latest tag from Docker Hub and start the vulnerability scanning.
– `$ anchore-cli image vuln <image>:latest all`
  To output the vulnerability data of an image.

The output of the last command is in plain text where the different columns are separated with an arbitrary number of spaces, see Figure 3.3. To clean this data and translate it into CSV format, with the same number of commas in each entry, different command-line tools were used. To specify: `sed`, `egrep` and `awk`.

Limitations and skipped images

The script fails on some images, which will be further elaborated in Section 4.2. There are four different failure reasons; permission denied, not found, unknown and time out. The first two are autogenerated by Anchore Engine, while unknown, and time out is added by us. Permission denied occurs when pulling an image is not generic, such as when it requires the user to be authenticated. Not found happens when the image name we try to pull is not in accordance with a real image, or that the latest tag does not exist. To achieve some efficiency, a time out error is triggered when images do not finish analyzing after one hour of running. Lastly, the unknown error happens when the failure reason does not fit in any of the other categories, and hence, is unknown.

4.2 Data collection

The data collection phase constituted of three steps; running the web scraper, extracting image information from the two APIs, and running the automatic analyzer. The web scraper took approximately 15 minutes to run, and created the files *image-names.txt* and *image-info.csv* with the contents as described in Section 4.1.1. Data was collected about 2540 unique images in total. Further, the scripts that extract information from the APIs ran for approximately 10 minutes. Lastly, the automatic analyzer took the *image-names.txt* file as input and analyzed all 2540 images during February 2020. The experienced run time depended highly on the images to be analyzed. For example, an image with no vulnerabilities took a few seconds, an image with many vulnerabilities took tens of minutes, and an image that took more than one hour would be considered failed. The complete analysis of all 2540 images was started on February 25th and took approximately 72 hours. The automatic analyzer outputs the files *failed.csv* with information about the failed images, and *vuln.csv* with vulnerability data about the images that were analyzed. In total, 2412 images were successfully analyzed.

Manual inspection of image tags

After a test run of the automatic analyzer, as many as 583 images failed (22.95%). Due to difficulties in automating the analyzing process for these images, they needed to be manually inspected to find the correct way to pull the image. There are different reasons why the analysis fails on some images. Some of the verified images have a specified way of pulling the images, while other images cost $0.00 and require a registration process, both of which make it hard to pull them automatically. Also, the script only pulls images with the latest tag, which makes some images fail because they are missing this tag. Lastly, we found that some images fail because the image name from the URL (which is what the automatic analyzer uses), is different from the image name used in the pull command.

As pointed out, we observed that the tag values and the way images are pulled from Docker Hub are inconsistent. In Figure 4.3, a screenshot of the *casanode/lnapi* repository on Docker Hub shows how tags are structured in some repositories, which is making it extremely hard to automate the process because image tags are irregular in some repositories. Our approach in cases like this was to choose the uppermost tag in order to get the newest release or to just go with a version. It was decided that it would take too much time to analyze all versions of images, so to choose a tag was better than leaving out the repository as a whole. After manual inspection of the image tags and rerunning the analysis, the number of failed images was reduced from 583 to 128, which is only 5% of the entire image set.

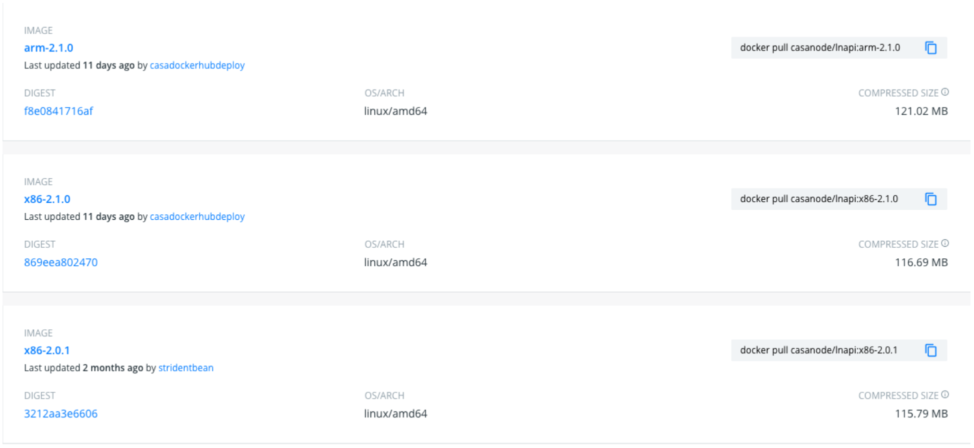


IMAGE
arm-2.1.0
Last updated 11 days ago by casadockerhubdeploy
docker pull casanode/lnapi:arm-2.1.0
DIGEST f8e0841716af
OS/ARCH linux/amd64
COMPRESSED SIZE 121.02 MB

IMAGE
x86-2.1.0
Last updated 11 days ago by casadockerhubdeploy
docker pull casanode/lnapi:x86-2.1.0
DIGEST 869eea802470
OS/ARCH linux/amd64
COMPRESSED SIZE 116.69 MB

IMAGE
x86-2.0.1
Last updated 2 months ago by stridentbean
docker pull casanode/lnapi:x86-2.0.1
DIGEST 3212aa3e6606
OS/ARCH linux/amd64
COMPRESSED SIZE 115.79 MB

Figure 4.3: Some of the available tags in casanode/lnapi repository on Docker Hub. The tag is specified is specified after the docker pull command to the right.

Challenges

It was discovered that the data set from the web scraper included duplicate rows, meaning that one image was included more than one time in the files. This propagated to the other two scripts because they both use the output file of the web scraper as input. Initially, there were 22 duplicate images from the web scraper resulting in 9,574 duplicate rows in the vulnerability file. The scraper created duplicate rows when an image was present on multiple pages of the Docker Hub websites during the execution of the scraper. The images on Docker Hub are ordered by popularity, and when the popularity order is updated, some images will change the page number they are present on. When this happens during the execution of our script, the name of the image will be scraped both times. Because Docker Hub is continuously updated, this issue was inevitable.

As the scraper only accesses 2500 images for each run, the way to overcome this issue was to run the scraper multiple times with some time apart. The result was multiple files with 2500 images each. By removing duplicate rows in these files, the final image file contained 2540 unique images, and this file was then used in the two other scripts.

4.3 Import data into database

When importing the data into the MySQL database in MySQL Workbench, the overall process was first to create appropriate tables and then import the outputted files from the previous project phases into them. Also, we generated a new table by joining two of the tables so that the data was merged. This new table replaced the two old ones. The resulting database structure is presented as an Entity-Relationship (ER) diagram in Figure 4.4. It shows the three tables that we have used in the data analysis phase, including column names and types, primary and foreign keys, and the relation between them. Image_info has a one-to-many relationship with the vuln table, since one image can contain multiple vulnerabilities. With the failed table, image_info has a one-to-one relationship, indicating that one image can only fail once with one failure reason. The primary key in image_info (image_id) is the foreign key in both vuln and failed.

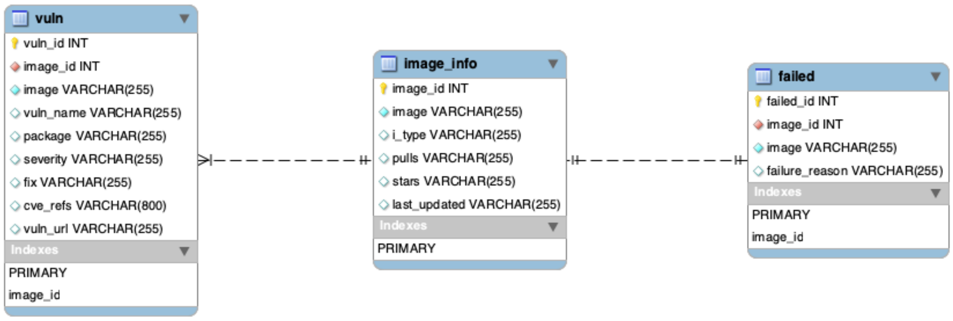


Figure 4.4: ER diagram of the MySQL database

Four database tables were created initially, one for the web scraper data, one for the API data, one for information about vulnerabilities, and one for information about failed images. After merging the tables with web scraper data and API data, the database consisted of three tables, as seen in Figure 4.4. We used SQL queries to create tables, and Listing 1 gives an example of how this was done. The example shows how the vulnerability data table was created, called vuln in Figure 4.4. All other tables were created similarly, see Appendix F for the complete SQL query script used in this phase.

Listing 1 SQL query for creating a table

```
CREATE TABLE vuln (
    vuln_id INT AUTO_INCREMENT PRIMARY KEY,
    image_id INT,
    image VARCHAR(255) NOT NULL,
    vuln_name VARCHAR(255),
    package VARCHAR(255),
    severity VARCHAR(255),
    fix VARCHAR(255),
    cve_refs VARCHAR(800),
    vuln_url VARCHAR(255),
    FOREIGN KEY (image_id) REFERENCES image_info(image_id));
```

After the tables were created, we used the Data Import Wizard tool in MySQL workbench to import the CSV data files and insert the data in appropriate columns as defined when creating the table.

Lastly, it was practical to merge the image information gathered from the web scraper and the APIs into one table. This was done to simplify the queries in the next phase. We used the JOIN function in SQL to merge the tables, as seen in Listing 2. The image_info table was created similarly as Listing 1 with the columns as specified in Figure 4.4, and then the result of the join query was inserted into the image_info table.

Listing 2 SQL query for joining two tables

```
INSERT INTO image_info
SELECT
    A.image_id,
    A.image,
    A.i_type,
    COALESCE(B.pulls, ''),
    COALESCE(B.stars, ''),
    COALESCE(B.last_updated, '')
FROM image_info_scraper A
LEFT JOIN image_info_api B USING (image);
#COALESCE is used to change empty values from NULL to ''
```

After this phase, the resulting environment was the base ground for the SQL queries used in the data analysis phase. The complete SQL script used in the data import phase can be found in Appendix F.

4.4 Data analysis

The data analysis phase consisted mainly of writing and executing SQL queries in our MySQL database environment. The goal of this phase was to restructure and analyze our gathered data to answer our research questions. The results are thoroughly presented in Chapter 5. In this section, we present the most important SQL queries that were used and describe their purpose. The complete SQL script can be found in Appendix G. Throughout this section, we will refer to the exact parts of the results chapter where the queries have been used.

To see how many empty values existed in the images that were successfully analyzed, the SQL query in Listing 3 was used. A similar query was used for the vuln table as well. It groups the data by image type (verified, certified, official, and community) and counts the number of non-empty values for each column. The result of this particular query was used in Table 5.1 in Section 5.1.2.

Listing 3 Number of non-empty values for each column in image_info

```sql
SELECT
    i_type,
    COUNT(DISTINCT image_id) AS number_of_images,
    COUNT(DISTINCT IF(NOT image_id='',image_id,Null)) AS image_id,
    COUNT(DISTINCT IF(NOT A.image='',image_id,Null)) AS image,
    COUNT(DISTINCT IF(NOT i_type='',image_id,Null)) AS i_type,
    COUNT(DISTINCT IF(NOT pulls='',image_id,Null)) AS pulls,
    COUNT(DISTINCT IF(NOT stars='',image_id,Null)) AS stars,
    COUNT(DISTINCT IF(NOT last_updated='',image_id,Null)) AS last_updated
FROM (SELECT * FROM image_info WHERE image_id NOT IN (SELECT image_id FROM
↪   failed)) A
LEFT JOIN vuln B USING (image_id)
GROUP BY i_type;
# '' indicates empty field
```

Several of our queries contain the JOIN function because we want to extract information about images across multiple tables. For example, when counting the number of vulnerabilities in each image type, the number of vulnerabilities is in the vuln table, and the image type is in the image_info table. Listing 4 shows a query for extracting the data that was used in Figure 5.3a in Section 5.1.2. It displays the number of analyzed and failed images, and in order to count both in the same query, the image_info and failed tables were joined.

Listing 4 Number of analyzed and failed images of each image type

```
SELECT
    i_type,
    COUNT(DISTINCT IF(NOT A.image_id IN (SELECT image_id FROM failed),
    ↪  A.image_id, Null)) AS analyzed_images,
    COUNT(DISTINCT B.image_id) AS failed_images
FROM image_info A LEFT JOIN failed B USING (image_id)
GROUP BY i_type;
```

For statistical calculations, MySQL contains multiple built-in functions, like average, maximum and minimum values, and standard deviation. However, there are no built-in function for calculating the median. Therefore, we used the query in Listing 5. The query extracts all images and the number of vulnerabilities for each image, then orders this list by the number of vulnerabilities in descending order, and outputs the vulnerability count of the image that is placed in the middle of the list. The median value is calculated by taking the average value of the two middle values to take into account when the number of rows in the data is odd. Median has been used in Sections 5.2.2 and 5.2.3 of the results.

Listing 5 Median value of number of vulnerabilities per image

```
SET @rowindex := -1;
SELECT AVG(D.vuln_count) AS median
FROM
    (SELECT @rowindex:=@rowindex + 1 AS rowindex, C.vuln_count
    FROM
        (SELECT
            A.image_id,
            COUNT(vuln_name) AS vuln_count
        FROM (SELECT * FROM image_info WHERE image_id NOT IN (SELECT image_id
        ↪  FROM failed)) AS A
        LEFT JOIN vuln B USING (image_id)
        GROUP BY A.image_id
        ORDER BY vuln_count DESC) AS C) AS D
WHERE D.rowindex IN (FLOOR(@rowindex / 2) , CEIL(@rowindex / 2));
```

We also used the built-in functions that were mentioned in the previous paragraph. Listing 6 shows how we extracted the data that was used in Table 5.4 in Section 5.2.2. It calculates the average number of vulnerabilities per image, the maximum and the minimum number of vulnerabilities, and the standard deviation for each image type. For this query, we count the vulnerabilities with severity degrees negligible and unknown as zero.

Listing 6 Statistical values for vulnerabilities per image for each image type

```
#average, max, min and standard deviation of number of vulnerabilities in
↪  each image
SELECT
    i_type,
    AVG(vuln_count) AS average,
    MAX(vuln_count) AS max_,
    MIN(vuln_count) AS min_,
    STDDEV_SAMP(vuln_count) AS stddev_
FROM
    (SELECT
        A.image_id,
        MIN(i_type) AS i_type,
        COUNT(IF(NOT vuln_name='' AND (NOT severity='negligible' AND NOT
        ↪  severity='unknown'), 1, Null))  AS vuln_count
    FROM (SELECT * FROM image_info WHERE image_id NOT IN (SELECT image_id
    ↪  FROM failed)) AS A
    LEFT JOIN vuln B USING (image_id)
    GROUP BY A.image_id) AS C
GROUP BY i_type;
```

In some cases, we needed to extract rows based on a substring of a column value. An example of this is found in Listing 7, where we use the SUBSTR() function of SQL to extract substrings from specific positions from the column values. The vuln_name column is on the following format: “[type]-[year]-[seq no]”, where type is either CVE, RHSA, ELSA or ALAS (for example “CVE-2020-12345”). We wanted to group distinct vulnerabilities based on the year they were released, so SUBSTR() was used to extract only the year of that string. In our queries, we also used the INSTR() function of SQL to find a substring inside a specific column value. The result of the query in Listing 7 was used for Figure 5.12b in Section 5.2.9. The INSTR() function was also used in Section 5.2.7 for extracting data about Microsoft images.

Listing 7 Number of unique vulnerabilities in each image type, grouped by year

```
SELECT
    DISTINCT IF(INSTR(vuln_name,'cve'),SUBSTR(vuln_name, 5, 4),
    ↪  SUBSTR(vuln_name,6,4)) AS year,
    COUNT(*) AS total_count,
    COUNT(IF(i_type='verified', vuln_name, Null)) AS verified_count,
    COUNT(IF(i_type='certified', vuln_name, Null)) AS certified_count,
    COUNT(IF(i_type='official', vuln_name, Null)) AS official_count,
    COUNT(IF(i_type='community', vuln_name, Null)) AS community_count
FROM (SELECT DISTINCT vuln_name, i_type FROM vuln A LEFT JOIN image_info B
↪   USING(image_id)) AS A
GROUP BY year
ORDER BY year;
```

Another statistical concept that is not incorporated as a built-in function in SQL is correlation. Correlation was used when answering the second research question in order to detect whether or not there is correlation between image features and the number of vulnerabilities. The SQL queries that we used for calculating the Spearman's correlation coefficient between the number of vulnerabilities, and the number of pulls, stars and days since last update, can be found in Appendix G (from line 398). The query calculates the ranked values of the variables using the RANK() function in SQL, and then uses built-in functions like average and standard deviation for calculating the Pearson correlation coefficient (see Equation 3.4 from Section 3.4.8) on the ranked values. The result is the Spearman correlation coefficient as a value between -1 and 1.

Listing 8 shows a simple query that is used in Section 5.5.1 when finding what packages that are the most vulnerable. The query groups the data by package, counts the number of critical vulnerabilities in each package, and orders the result by the critical count in descending order. Additionally, the query counts the number of images that contain the specific package. Similar queries was used in a number of other sections, such as Sections 5.2.5, 5.4.1 and 5.5.2.

Listing 8 Finding the most vulnerable packages in images

```
SELECT
    package,
    SUM(severity='Critical') AS critical_count,
    (SELECT COUNT(DISTINCT image_id)) AS number_of_images
FROM vuln
GROUP BY package
ORDER BY critical_count DESC;
```

The last SQL query is presented in Listing 9. It groups the data by image, and counts the number of vulnerabilities, pulls stars and days since last update for each image. This query is not used to display any result directly. However, several of our other queries use this query (or parts of it, especially the vulnerability count) as a subquery in the FROM clause. Examples from this section are Listings 5 and 6, however, there are more examples in the full SQL script in Appendix G.

Listing 9 Counting number of vulnerabilities, pulls, stars and days since last update in each image

```sql
SELECT
    A.image,
    COUNT(vuln_name)  AS vuln_count,
    AVG(IF(NOT pulls='',pulls,Null)) AS pulls,
    AVG(IF(NOT stars='',stars,Null)) AS stars,
    AVG(IF(NOT last_updated='',DATEDIFF('2020-02-25', last_updated),Null)) AS
    ↪  days_since
FROM (SELECT * FROM image_info WHERE image_id NOT IN (SELECT image_id FROM
↪  failed)) AS A
LEFT JOIN vuln B USING (image_id)
GROUP BY A.image;
```

In addition to using SQL queries in the data analysis phase, we used Google Sheets and Python for simple calculations on the results from SQL queries. Spreadsheets were used to temporarily store important results from SQL queries, structuring tables and calculating sums, percentage, and ratios that were not necessary to incorporate in the SQL queries.

4.5 Visualization of results

The final project phase was about visualizing the results obtained in the previous phases. We have used numerous methods of visualization, and this section explains them briefly. All tables in Chapter 5 are created directly in Latex, using built-in functionality. The figures in Chapter 5 are all created by using the Matplotlib and Seaborn libraries in Python, which support most types of charts, plots, and graphs. We have utilized line graphs, pie charts, bar charts, histograms, and scatter plots.

Chapter 5 Results

This chapter presents all results needed to answer the research questions of this thesis (see Section 3.1). First, the evolution of images on Docker Hub is investigated, and then, our obtained data set is explained in detail. Further, the results are divided into four sections that present results so that they, as a whole, address each research question extensively.

5.1 Data set

To understand the results of the analysis, it is necessary to describe the data set that has been used. This section will first look at the evolution in the quantity of images on Docker Hub, then explain the content of the three database tables that were used in the data analysis phase (Section 4.4).

5.1.1 Images on Docker Hub

We present Figure 5.1, which displays the number of images on Docker Hub measured one week apart from February 3rd, 2020 to March 30th, 2020. The figure is included to make the reader aware of the continuous growth in the volume of Docker Hub images.

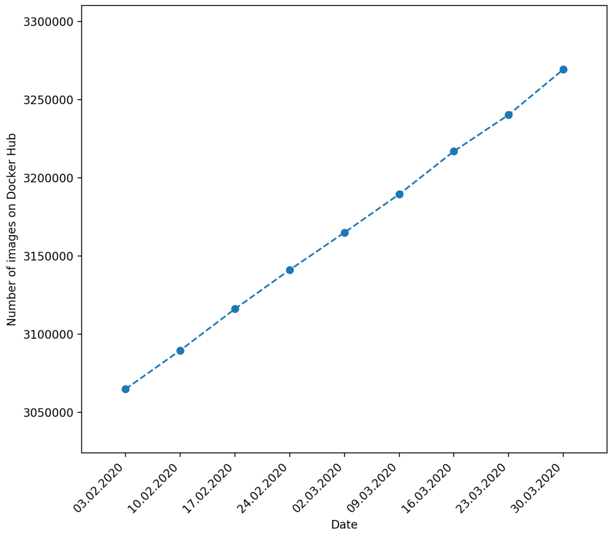


Figure 5.1: The evolution in quantity of images on Docker Hub

The number of images is increasing by approximately 25,000 images each week and follow a linear growth. This means that the Docker Hub ecosystem is continuously changing, especially considering that this increase in images is only reflecting approximately two months. To see the evolution in a longer time perspective, the number of images was only 95,000 at the time of BanyanOps' analysis in 2015 [GDT15].

5.1.2 Image information and failed images

In our database, the image_info table contains 2450 image entries. Out of these, 2,412 images were successfully analyzed. The distribution of images in the four types (verified, certified, official, and community) can be seen in Figure 5.2. It is worth noting that the community images represent by far the most substantial portion of the data set by 90%. Following are official images with 6.5%, verified images with 2.5%, and certified images with 1%.

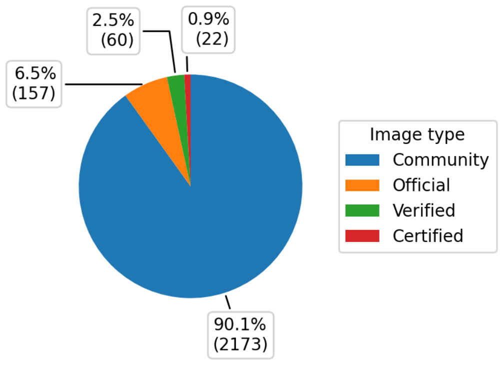


Figure 5.2: Image type distribution

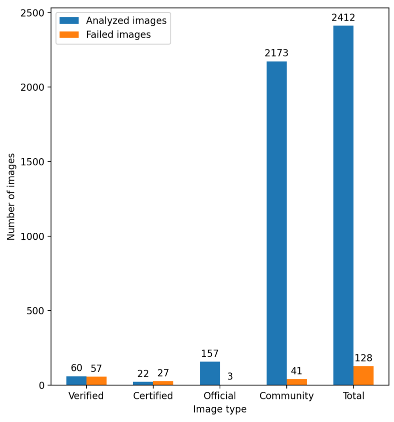
The database table consists of six columns, where some of the columns contain null values because it was not possible to collect data about all images. Table 5.1 shows the number of analyzed images for each image type. For each column in the image_info database table, the table shows how many analyzed images have data in the column and the respective percentage of non-null values. The percentage is based on the total number of analyzed images of each image type (the Images column), not the total number of analyzed images. For instance, looking at the pulls column for verified images, the percentage is based on the total number of analyzed, verified images (60) and how many of these images that have the pulls attribute (53). The columns image_id (unique numerical value), image (name), and i_type (image type) are all complete and contain no null values. The remaining columns have 96-99% data. Note that official and community images have very few entries with null values in pulls and stars compared to verified and certified images. For certified images, there are neither pulls nor stars data, but all images have last_updated data. For verified images, 88% have pulls data, only one has stars data, and all have last_updated data.

Image type	**Images**	**Column name**			
		`image_id`, `image`, `i_type`	`pulls`	`stars`	`last_updated`
Verified	60	60 (100%)	53 (88.3%)	1 (1.7%)	60 (100%)
Certified	22	22 (100%)	0 (0.0%)	0 (0.0%)	22 (100%)
Official	157	157 (100%)	156 (99.4%)	156 (99.4%)	156 (99.4%)
Community	2173	2173 (100%)	2170 (99.9%)	2170 (99.9%)	2169 (99.8%)
Total	2412	2412 (100%)	2379 (98.6%)	2327 (96.5%)	2407 (99.8%)

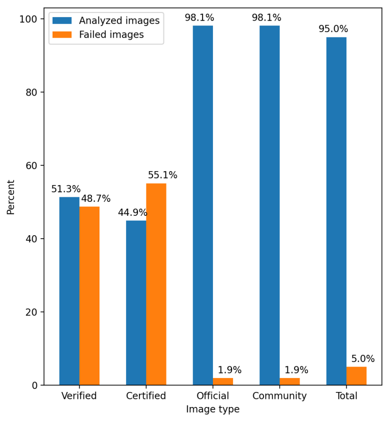
Table 5.1: Number of analyzed images of each type and column information with number of images with data. The percentages are based on the total number of images of each image type from the Images column.

Failed images

Out of the 2540 images that were collected, 128 images failed. Figure 5.3a shows the distribution between analyzed and failed images in the four image types, and in total. As the majority of the images are community images, this bar is prominent in the chart. Note that, despite the large difference in the number of images between community and verified images, there are fewer community images that failed to be analyzed than verified images (41 failed community images versus 57 failed verified images). Also, note that only three of the official images failed, and that certified is the only image type where the number of failed images is higher than the number of analyzed images. To take the large variations in the number of images in each type into account, Figure 5.3b is presented. It shows the percentage of analyzed versus failed images in the four image types, and in total. A little over half of the verified images were analyzed, and less than half of the certified images were analyzed. For official and community images, 98.1% were analyzed. In total, 95% were analyzed, and 5% failed.



(a) Number of analyzed and failed images in each image type, and in total.



(b) Percentage of analyzed and failed images each image type, and in total.

Figure 5.3: Analyzed and failed images

The failed table in the database contains 128 entries where all entries have a failed_id, image_id, image (name) and failure_reason, and no null values. There are four different failure reasons; permission denied, not found, time out, and unknown error. The distribution of images in the different failure categories can be seen in Figure 5.4. Almost half of the images fail due to permission denied error, which is likely to correspond with the high number of failed verified and failed certified images. Indeed, many verified and certified images require authentication, which will lead to a permission denied failure, as explained in Section 4.1.3.

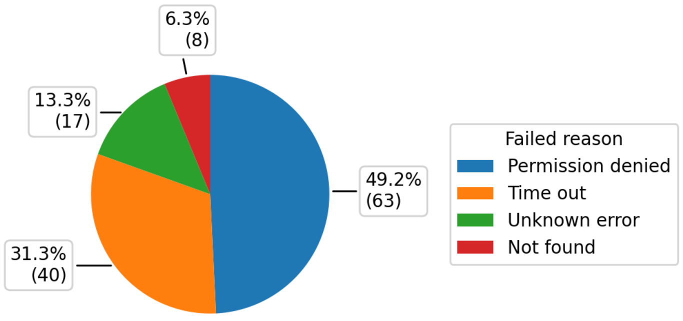


Figure 5.4: Distribution of failed images

As an additional delivery, we have created a separate document called "Additional appendices: An Extensive Analysis of the Current Vulnerability Landscape in Docker Hub Images". It includes two appendices. Appendix A includes a list of all analyzed images grouped by each image type. Each entry in the list have the image name and the number of detected vulnerabilities. Appendix B includes a list of the failed images, with image name and failure reason. The document is an attachment to our master thesis delivery on the NTNU submission page.

5.1.3 Vulnerability information

The last table in the database is the vuln table, which contains all vulnerability data. It consists of as many as 918,792 entries, and there are nine different columns. Table 5.2 shows the column names, the number of values, the number of unique values, and the number of null values for each column. Vuln_id is a unique integer for each entry in the table. Image_id is a unique integer for each image, and image is the name of the image. Our data contains 1,982 images with at least one vulnerability. The vuln_name values are the names of the vulnerabilities found, meaning that the data set includes 9,408 unique vulnerabilities. There are four categorizations of vulnerabilities in the data set; CVE, RHSA, ALAS, and ELSA. However, 98% of the entries are CVE vulnerabilities.

Further, the package column states in which package the vulnerability is found, where the data set consists of 13,381 unique package values. There are six severity levels (critical, high, medium, low, negligible, and unknown), as explained in Section 3.4.4. The fix value typically indicates the name of a newer version of the package that the vulnerability is found in. 35% of the fix values are "none", meaning there is no known fix to the vulnerability. The cve_refs column works as a mapping between the vulnerability names and CVEs from NVD. So, for every entry with a none-CVE vuln_name, the reference is a CVE value or a list of CVE values. For every entry with a CVE vulnerability as vuln_name, the reference is the same as the vulnerability name or a null value. That is why this is the only column with null values. Lastly, the vuln_url column contains a URL to a web page with more information about the vulnerability.

Column name	Number of values	Number of unique values	Number of null-values
`vuln_id`	918,792	918,792	0
`image_id`	918,792	1,982	0
`image`	918,792	1,982	0
`vuln_name`	918,792	9,408	0
`package`	918,792	13,381	0
`severity`	918,792	6	0
`fix`	918,792	4,661	0
`cve_refs`	913,285	9,085	5,507
`vuln_url`	918,792	14,233	0

Table 5.2: Column information for the vulnerability table in the database

Throughout the rest of Chapter 5 Results, it is essential to state the difference between a vulnerability and a *unique* vulnerability. From now and throughout, a vulnerability is an entry from the vuln table. There are 918,792 such values, meaning this is the total number of vulnerabilities in the data set. A unique vulnerability is a vulnerability with a unique value in the vuln_name column, which means that a unique vulnerability can be found in multiple images. There are 9,408 unique vulnerabilities in the data set.

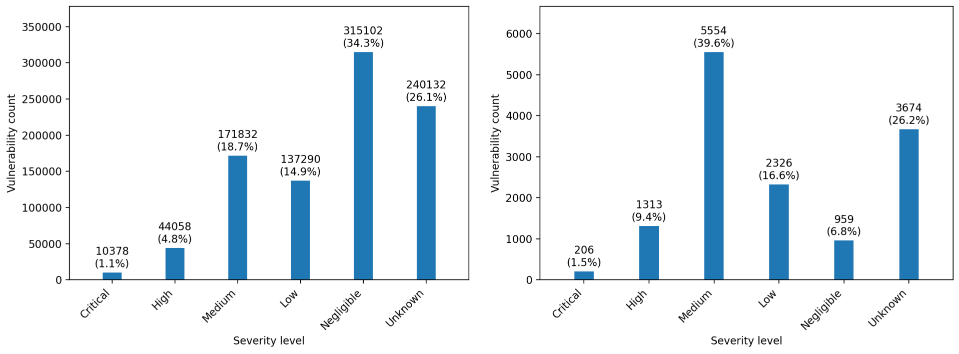
5.2 The vulnerability landscape of Docker Hub

This section will present the results gathered to answer ***RQ1:*** *How can vulnerabilities found in Docker images be systemized in order to investigate the current vulnerability landscape of Docker Hub?*

5.2.1 Distribution of vulnerabilities in each severity category

The number of vulnerabilities found in each severity category is presented in Figure 5.5. The vulnerabilities found in total and the number of unique vulnerabilities are both defining the vulnerability landscape, and therefore, both of these perspectives are investigated.

In Figure 5.5a, all found vulnerabilities are included. Thus, in this data set, the same vulnerability could potentially have multiple entries in the result. This is, for instance, because a particular vulnerability could be found in multiple images, and a single image could contain the same vulnerability in multiple packages. In Figure 5.5b, only unique vulnerabilities are shown. It should be pointed out that some



(a) Distribution of all 918,792 vulnerabilities

(b) Distribution of 14,032 unique vulnerabilities

Figure 5.5: Vulnerability distribution in severity levels

vulnerabilities are present in several severity categories, depending on which image it is found in. In cases like this, all versions of the vulnerability are included, which makes up a total of 14,032 vulnerabilities. This is why the total number of unique vulnerabilities here is higher than what is described in Section 5.1.3.

The negligible and unknown categories clearly stand out in Figure 5.5a, with a total of 315,102 and 240,132 vulnerabilities, respectively. When considering unique vulnerabilities (Figure 5.5b), the medium category is the most dominant one with 5,554 unique vulnerabilities. By examining the relation between Figure 5.5a and Figure 5.5b, one can observe the ratio of vulnerabilities in each severity category. The negligible category contains few unique vulnerabilities represented in many Docker images, whereas the medium category has many unique vulnerabilities represented at a lower ratio. The vulnerability ratio will be explained in detail in the next paragraph.

Severity	**Number of vulnerabilities (A)**	**Number of unique vulnerabilities (B)**	**Ratio (A/B)**
Critical	10,378	206	50.4
High	44,058	1,313	33.6
Medium	171,832	5,554	30.9
Low	137,290	2,326	59.0
Negligible	315,102	959	328.6
Unknown	240,132	3,674	65.4
Total	918,792	14,032	65.5

Table 5.3: Vulnerability ratio in severity levels

Table 5.3 shows the total number of vulnerabilities, the number of unique vulnerabilities, and the ratio, which is measured as the total number of vulnerabilities divided by the number of unique vulnerabilities. So, for each unique vulnerability, there are a certain number of occurrences of the specific vulnerability in the data set. For example, for each unique vulnerability in the critical category, there are 50 occurrences of this vulnerability in the data set on average. For each unique negligible vulnerability, there are as many as 329 occurrences on average. This is significantly larger than the other values. Despite medium having the highest number of unique vulnerabilities, it has the lowest ratio. To read about how the Anchore Engine determines the different severity categories, see Section 3.4.4.

5.2.2 Central tendency of the vulnerability distribution

We present the central tendency of our data set to emphasize the high variations of vulnerabilities in images. Figure 5.6 shows the number of vulnerabilities (y-axis) found in each image (x-axis) in descending order. Less than 50 images have more than 4,000 vulnerabilities, and approximately 1,000 images have less than 50 vulnerabilities. Therefore, we take a look at both the average number of vulnerabilities and the median. The average is 381 vulnerabilities per image, while the median is 82. In cases like this, where the distribution is skewed, the median will give a better representation of the central tendency than the average. This is because the extreme values pull the average away from the center.

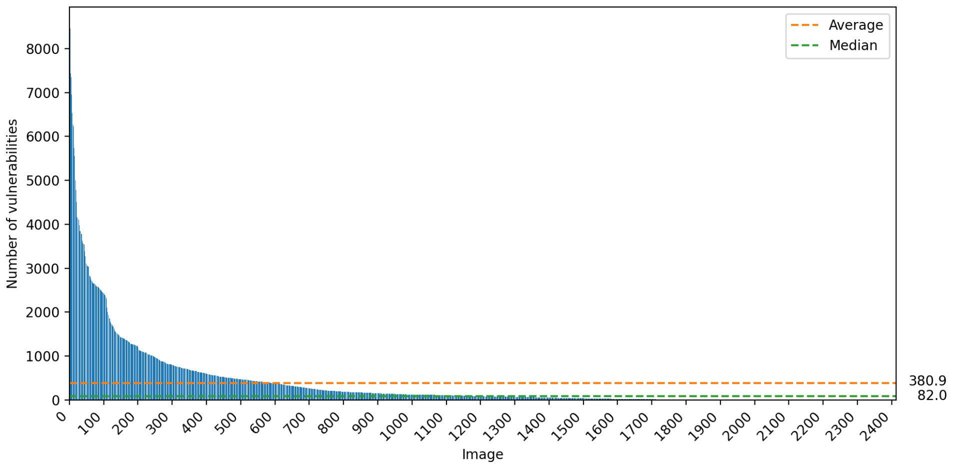


Figure 5.6: Average and median of the number of vulnerabilities in each image

We have looked at the average and median values of the number of vulnerabilities in images when disregarding the vulnerabilities that are categorized as negligible and unknown. Looking at Table 5.3 from the previous section, one can see that negligible and unknown vulnerabilities together make up 555,234 out of the 918,792 vulnerabilities (which is 60% of all vulnerabilities). As vulnerabilities in these two categories are considered to contribute with little threat when investigating the current vulnerability landscape, it gives a more accurate result to exclude these. Also, they have to be excluded to comply with CVSS v3.1. Therefore, we calculated the average and the median value of vulnerabilities in images when disregarding negligible and unknown vulnerabilities (counting them as zero). The result was 150.7 for the average and 25.5 for the median, which is significantly lower.

To investigate the data when disregarding the negligible and unknown vulnerabilities further, we created Table 5.4 that shows statistical values of the number of vulnerabilities for each image type. The results show that community images have the highest average, maximum and standard deviation values (158, 6,509 and 391, respectively). A high standard deviation value means that the average distance from the center of the data to each data point is big, indicating that the data is sparse and varied. This is also reflected in the fact that the maximum value for community images is significantly larger than the average and the median, which is the case for the other three image types as well.

The image type that is considered as the least vulnerable is official. It has the lowest average of 73 and the lowest median value of 9. Further, the maximum and standard deviation values for official images are the second lowest. The lowest

Image type	**Number of analyzed images**	**Number of vulnerabilities**	**Number of vulnerabilities**				
			Average	Median	Max	Min	Standard deviation
Verified	60	6,073	101.2	13	1,128	0	225.9
Certified	22	1,987	90.3	37	428	0	121.3
Official	157	11,489	73.2	9	1,615	0	214.4
Community	2,173	344,009	158.3	28	6,509	0	391.1

Table 5.4: Statistical values for vulnerabilities per image type, disregarding negligible and unknown vulnerabilities. Inspiration from [SGE17].

maximum value belongs to certified and is only 428. Although certified has the lowest maximum value, it has the highest median value. This indicates that a larger portion of the images has many vulnerabilities. The standard deviation of 121 for this image type is the lowest, meaning that the data is less sparse. As a final note, all four image types contain at least one image with zero vulnerabilities.

5.2.3 Vulnerabilities in each image type

There are significant differences in the number of analyzed images in each image category. To mention an example, there are 2,173 analyzed community images, and only 22 analyzed certified images (Figure 5.3a). This has to be accounted for in order to compare the number of vulnerabilities in each image type. Hence, the result is presented as the median value of vulnerabilities per image for each image type (see Figure 5.7). As described in the previous section, the median describes the central tendency better than the average when the data is skewed and is therefore chosen. Note that only critical, high, medium, and low vulnerabilities are included in the figure. The negligible and unknown vulnerabilities are not included.

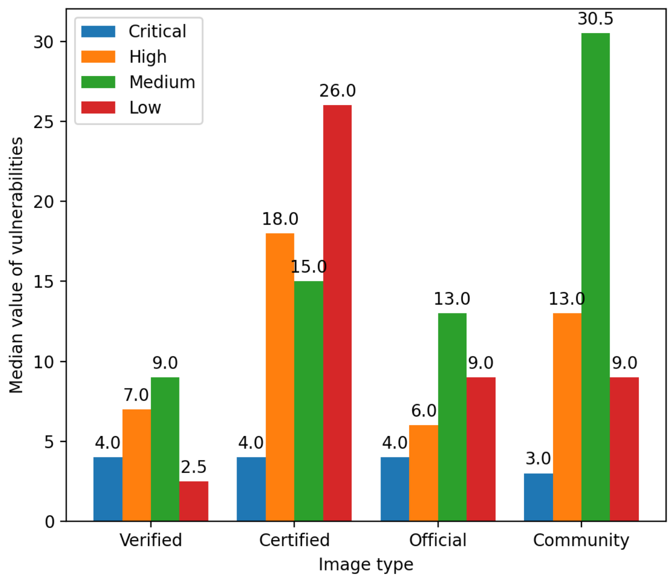
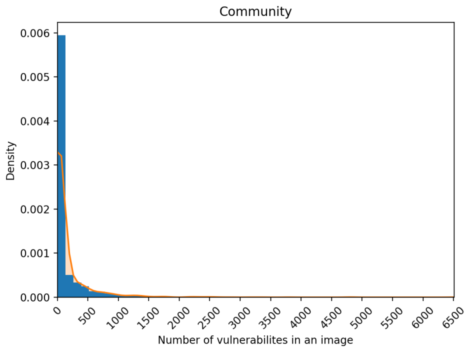


Figure 5.7: Median values of vulnerabilities for each severity category and image type

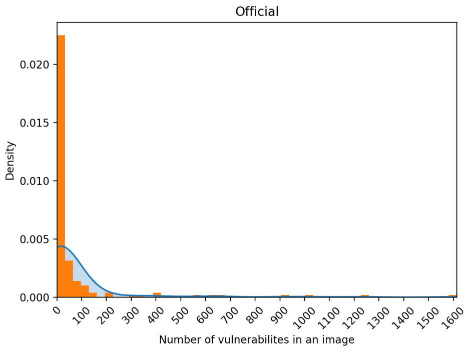
The results show that the median of critical vulnerabilities is almost the same for all four image types. For the other severity categories, the median is more varied across the image types. The high severity category is the most represented in certified images, while the medium category is the most represented in the community images. For verified, official and community images, the medium severity has the highest median, while the certified images have the most low vulnerabilities. Overall, the certified images are the most vulnerable.

5.2.4 Density distribution

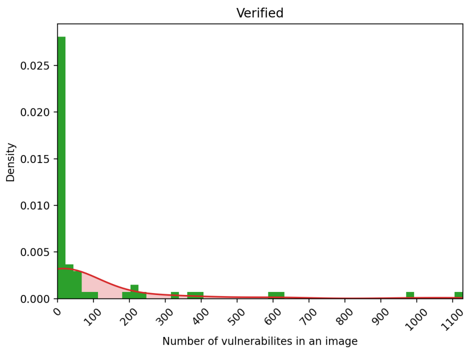
As the vulnerability count in the images in our data set is varied, we present the density distribution for each image type. Figure 5.8 shows the density distribution plots for all four image types. We have disregarded negligible and unknown vulnerabilities in these plots.



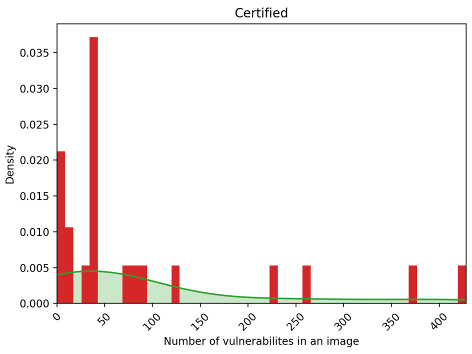
(a) Community images



(b) Official images



(c) Verified images



(d) Certified images

Figure 5.8: Density distribution plots for number of vulnerabilities in each image type

The density of community images (Figure 5.8a) is highly left-skewed, and the majority of images have between none and 1,000 vulnerabilities. Despite that the maximum number of vulnerabilities is 6,500, there are practically no images with over 2,000 vulnerabilities. The density is the highest for values under 250 vulnerabilities. We want to point out that the density range along the y-axis in community images (from 0 to 0.006) is much smaller than for the other three image types (from 0 to around 0.03). This means that the overall density of official images is significantly lower and that the values are more spread over the value range along the x-axis. For official images (Figure 5.8b), the histogram is also highly right-skewed, with a significantly higher density of values below 50. Almost all values range from 0 to 200 vulnerabilities.

When looking at the verified images in Figure 5.8c, the highest density is below 100 vulnerabilities, and the density estimation line is more leveled compared to community and official. There are higher densities around 200 vulnerabilities, around 350-400 vulnerabilities, and on 600, 990, and 1100 vulnerabilities, indicating that the distribution of vulnerabilities is spread. Lastly, the certified images (Figure 5.8d) are the most even out of the four image types. The majority of images hold less than 50 images, while the other values are spread across the x-axis with equal densities.

5.2.5 Images that contain the most critical vulnerabilities

Out of all 2,412 successfully analyzed images, this section presents the most vulnerable ones. Table 5.5 displays the top 10 most vulnerable images based on the number of critical vulnerabilities in each image. In cases where the critical count is the same, the image with the highest number of high rated vulnerabilities is considered as the most vulnerable one. The number of pulls column denotes the total number of pulls (downloads) for each image.

	Image	**Critical**	**High**	**Medium**	**Low**	**Number of pulls**
1	pivotaldata/gpdb-pxf-dev	822	698	576	132	139,246,839
2	cloudera/quickstart	571	2,155	1,897	158	6,892,856
3	silverpeas	341	264	397	226	828,743
4	microsoft-mmlspark-release	184	428	264	252	1,509,541
5	anchorfree/hadoop-slave	168	636	797	107	5,375,424
6	saturnism/spring-boot-helloworld-ui	133	217	112	2	12,686,987
7	pantsel/konga	133	39	169	0	12,431,685
8	renaultdigital/runner-bigdata-int	127	335	691	103	4,787,745
9	springcloud/spring-pipeline-m2	125	293	2,027	1,357	8,359,973
10	raphacps/simpsons-maven-repo	122	271	399	2	36,136,733

Table 5.5: Top 10 most vulnerable images (sorted by critical count)

Out of the top 10 most vulnerable images, there are 8 community images, 1 official image (*silverpeas*), and 1 verified image (*microsoft-mmlspark-release*). There are big variations in the number of vulnerabilities in all presented severity levels. The most vulnerable image, *pivotaldata/gpdb-pxf-dev*, has ~250 more critical vulnerabilities than the second most vulnerable image. However, the second most vulnerable image, *cloudera/quickstart*, contains as many as 2,155 high rated vulnerabilities, which is ~1500 more vulnerabilities than the one rated as the most vulnerable image. We decided to focus on the critical vulnerabilities in the ranking of the most vulnerable images. This is because it is the highest possible ranking, and hence, the most severe vulnerabilities are found in this category. The other severity categories are included in the table as extra information to give a clear view of the distribution of vulnerabilities. From the number of pulls column, one can observe that the most vulnerable image is also the most downloaded one out of the top 10, with almost 140 million pulls. This is approximately 100 million more pulls than the second most pulled image on this list (the *raphacps/simpsons-maven-repo* image). There is no immediate correlation that could be observed between the number of pulls and the number of vulnerabilities in these images.

5.2.6 Percentage of images with critical and high vulnerabilities

It is enough with a single vulnerability for a system to be compromised. Thus, we determine what percentage of images that contain at least one high or critical rated vulnerability for each image type, as shown in Figure 5.9.

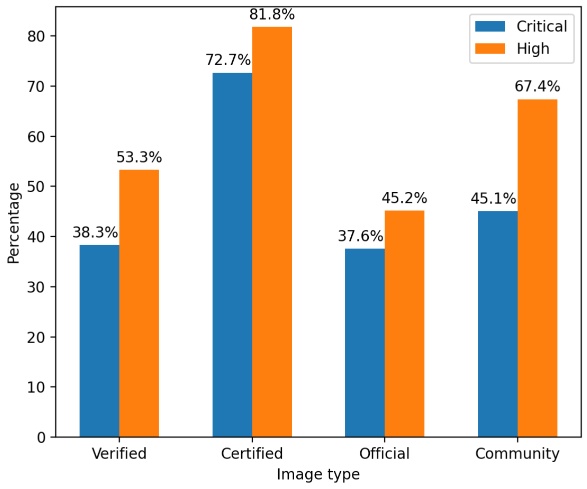


Figure 5.9: Percentage of images that contain at least one high or critical rated vulnerability

Our results in Figure 5.9 reveal that the certified image type, which is a subsection of the verified image type, is the most vulnerable with this measure. 82% of all certified images contain at least one vulnerability with high severity level, and 73% of them contain at least one critical vulnerability. Community images come out as the second most vulnerable image type. 67% have high vulnerabilities, and 45% have critical vulnerabilities. The third most vulnerable image type is verified, followed by official.

When combining these results, to investigate what amount of the image types that contain *either* at least one critical or high rated vulnerability, the results are as follows: 82% for certified images, 68% for community images, 57% for verified images and 46% for official images.[1] Hence, the official images are the least vulnerable. Still,

[1]Vulnerabilities that are given both critical and high severity in the same image are only included once in this calculation.

it should be emphasized that almost half of the official images contain critical or high rated vulnerabilities, as presented in this section.

5.2.7 Vulnerabilities in Microsoft images

There are variations in vulnerabilities that are present in different operating systems; thus, we seek to discover how vulnerable Microsoft images are compared to other images. In total, 115 Microsoft images were gathered.[2] Out of these, 67 were successfully analyzed, and 48 failed to be analyzed, as shown in Figure 5.10. Out of the successfully analyzed images, 52 were verified, and 15 were community images. The high number of failures is a combination of permission denied errors, special procedures for downloading, and the fact that many Microsoft repositories contain references to other repositories, instead of containing any actual images.[3]

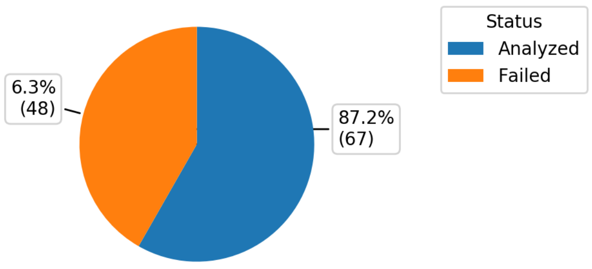


Figure 5.10: Distribution of analyzed and failed Microsoft images

The distribution of vulnerabilities in each severity category

Table 5.6 displays the number of vulnerabilities and the unique vulnerabilities found in Microsoft images in each severity category, as well as the vulnerability ratio. By comparing this table with Table 5.3 that shows the equivalent numbers for the whole data set, we observe that Microsoft images have a much lower vulnerability ratio in general. This means that the number of unique vulnerabilities in Microsoft images are represented fewer times in the data. To directly compare, unique vulnerabilities found in Microsoft images are on average found 10 times each, and this number increases to 66 when all images are considered. Most of the vulnerabilities are found

[2]Includes all found Microsoft based images, both verified images maintained by Microsoft, and community images.

[3]E.g. the Windows Base OS images repository: *https://hub.docker.com/_/microsoft-windows-base-os-images?tab=description*

in the negligible and unknown categories, similar to the results when considering the whole data set, as presented in Section 5.2.1.

Severity	**Number of vulnerabilities (A)**	**Number of unique vulnerabilities (B)**	**Ratio (A/B)**
Critical	368	39	9.4
High	915	99	9.2
Medium	3,151	419	7.5
Low	3,308	466	7.1
Negligible	8,676	415	20.9
Unknown	4,732	634	7.5
Total	21,150	2,072	10.2

Table 5.6: Vulnerabilities found in Microsoft images

By investigating the number of vulnerabilities in Microsoft images further, there is an average of 315.7 vulnerabilities per analyzed image. This number includes the negligible and unknown categories; however, it is of more interest to look at the critical and high rated vulnerabilities separately. When only considering the critical and high rated vulnerabilities, there are, on average, 19.2 vulnerabilities found in each analyzed Microsoft image. When considering all other analyzed images, excluding Microsoft images, this number is 22.7. This means that, on average, Microsoft images contain a lower number of critical and high rated vulnerabilities than other images.

In order to compare the number of vulnerabilities per image in each of the six severity levels between Microsoft images and other images, we present Figure 5.11. The vulnerabilities per image value (y-axis) is computed by dividing the total number of found vulnerabilities in that category by the total number of analyzed images (67 for Microsoft images and 2,345 for all other images). On average, Microsoft images contain slightly more critical vulnerabilities, while the other images, on average, have a higher number of vulnerabilities in all other severity categories.

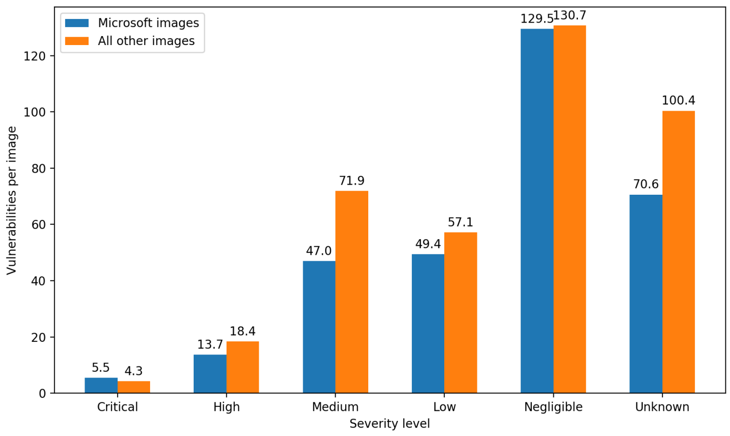


Figure 5.11: Number of vulnerabilities per image in each severity level

5.2.8 Images with no vulnerabilities

From our analysis, 430 images did not contain any vulnerabilities. This makes up 17.8% of the total number of analyzed images. 393 images were community, 29 images were official, only 6 were verified, and 2 were certified. For these images, the average number of pulls is 90,766,525, the average number of stars is 93.6 and the average number of days since the last update is 279 (approximately nine months). In Table 5.7, the average number of pulls is almost twice as big for images without vulnerabilities, while the average number of stars is lower. The average number of days since the last update is also lower for images without vulnerabilities, meaning they are more frequently updated.

	Images with vulnerabilities	**Images without vulnerabilities**
Number of images	1982	430
Average number of pulls	48,785,918	90,766,525
Average number of stars	118	94
Average number of days since last update	334	279

Table 5.7: Comparison of attributes of images with and without vulnerabilities.

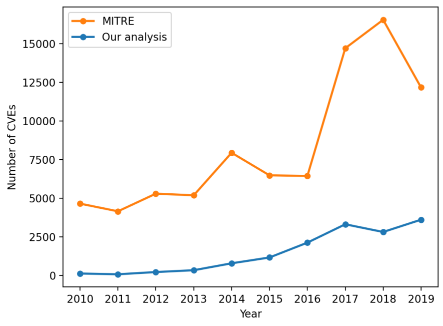
Additionally, we also looked at the number of images that only contain vulnerabilities with negligible and unknown severities. There are 93 such images, and if these are considered as images with no vulnerabilities, the number of images increases to 523 (21.7%).

5.2.9 The trend in CVE vulnerabilities

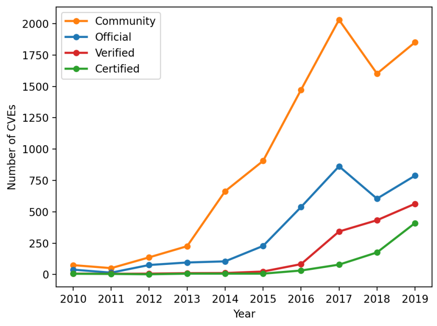
The number of reported CVEs each year is highly varying. The variations are expected to be reflected in the vulnerabilities found in our analysis as well. Thus, we aim to identify the overall trend in CVE vulnerabilities, compared to the unique vulnerabilities found throughout our analysis. Data gathered from the CVE Details database [CVE19a] is used to display the number of newly reported CVE vulnerabilities each year.

In Figure 5.12a the reported CVE vulnerabilities each year is presented together with the unique CVE vulnerabilities found in our analysis each year from 2010 to 2019. The orange line shows how the number of newly discovered CVE vulnerabilities varies by a few thousand vulnerabilities each year. However, there is a significant increase in 2017. This increase is not reflected in our analysis data, which is following a steady increase between 2014 and 2017. This increase can be explained by the introduction of Docker Hub in 2014, making new vulnerabilities more represented in images. As a final observation, the number of newly reported vulnerabilities from MITRE between 2018 and 2019 is decreasing, while our results show an increase.

Figure 5.12b shows the number of unique vulnerabilities found in each image type (i.e., community, official, verified, and certified) in our analysis from 2010 to 2019. This figure gives an insight into how the overall changes are reflected in each image type. Verified and certified images have had an increase in the number of unique CVE vulnerabilities each year from 2015. Community and official images, however, have had a significant decrease in unique vulnerabilities from 2017 to 2018. It is noteworthy to point out that the curves are affected by the time of introduction of the different image types. Official images were introduced in 2014, whereas verified and certified images were introduced in 2018.



(a)



(b)

Figure 5.12: CVE trend from 2010 to 2019. (a) displays all reported CVEs and unique CVEs from our analysis, (b) displays the observed CVEs in each image type from our analysis. Inspiration from [SGE17].

5.2.10 Days since last update

There is a high variation in how often Docker Hub images are updated. Intuitively, this affects the vulnerability landscape of Docker Hub. We have gathered data about when images were last updated, and calculated the number of days since the images were last updated, counting back from February 25th, 2020. The data set consists of last updated data for all analyzed images, except five.

The numbers from our database revealed that 31.4% of images have not been updated in 400 days or longer, and 43.8% have not been updated in 200 days or longer. The percentage of images that have been updated during the last 14 days is 29.8%. This implies that if these numbers are representative for all images on Docker Hub, a third of the images (31.4%) on Docker Hub have not been updated in the last 400 days or longer.

To go into more detail, Table 5.8 presents how often images in each of the image types are updated. Community and certified images are the least updated image categories, where 47.0% of community images and 36.4% of certified images have not been updated for the last 200 days or more. The verified images are the most frequently updated category, where 83.3% of images have been updated during the last 14 days.

Image type	**More than 400 days**	**More than 200 days**	**Less than 14 days**
Community	33.9%	47.0%	27.0%
Official	9.6%	14.7%	51.3%
Certified	18.2%	36.4%	13.6%
Verified	1.7%	5.0%	83.3%

Table 5.8: Percentages of images that are not updated for more than 400, 200, and less than 14 days

Figure 5.13 shows the exact update frequency for images of each type. Each mark in the plot represents the last update time for an image. Stapled vertical lines are inserted on 400, 200, and 14 days from the latest date (February 25th, 2020) to easily see the correspondence between the figure and Table 5.8. One can observe how the last update times of community images are evenly spread, but with a higher density in the last few years. Further, a handful of certified images are highly affecting the percentages from Table 5.8, because the overall number of certified images is small. Official images contain a large portion of images that have been updated recently (January 2020 to March 2020) and some more spread values with images that have not been updated since 2016. The verified images are clearly, from this plot also, the most updated image type, where there is only one image with the last updated time earlier than May 2019.

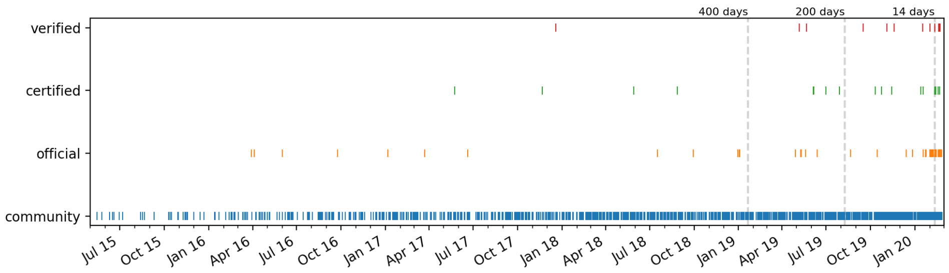


Figure 5.13: Last update dates for images of each image type

5.3 Correlation between image features and vulnerabilities

We investigate whether or not the number of vulnerabilities in an image is affected by a specific image feature. The reviewed image features are the number of times the image has been pulled, the number of stars an image has been given, and the number of days since the image was last updated. The following sections present the results required to answer **RQ2:** *How do image features and the number of vulnerabilities correlate in images?*

To compute the correlation, we used Spearman's r_s correlation coefficient [LOHS05] (described in Section 3.4.8). Spearman's correlation was chosen because our data set contain skewed values and are not normally distributed. When handling entries that contained empty values, we chose the approach of complete case analysis, which means omitting incomplete pairs. The alternative would be to impute missing values, which means to create an estimated value based on the other data values. However, this approach was not chosen because the values in our data set are independent of each other, and it would therefore not make sense to compute an estimated value. The number of entries for each image feature is presented in Table 5.1 (in Section 5.1.2), and it shows that as many as 96-99% of the entries are complete. This means that omitting empty values will not affect our results significantly.

Correlation between pulls and vulnerabilities

There is a common perception that: *images with the most pulls generally have few vulnerabilities, and images with the most vulnerabilities generally have few pulls.* To investigate this particular perception, we created a scatter plot given in Figure 5.14. We calculated the Spearman correlation coefficient between the number of pulls and the number of vulnerabilities for the whole set of investigated images, and the result was $r_s = -0.1115$. This is considered as no particular correlation. To explain this, we refer to the meaning of having a high negative correlation: the markers would gather around a decreasing line (not necessarily linear), indicating that images with more pulls have fewer vulnerabilities. In the case of a high positive correlation, the opposite would apply, i.e., the line would be increasing. There is not enough evidence to conclude that there is any correlation between the number of pulls and the number of vulnerabilities. However, we do observe a trend where images with the most vulnerabilities generally have a low number of pulls and images with the most pulls generally have few vulnerabilities. Also, images with less than 1000 vulnerabilities are roughly spread along the x-axis.

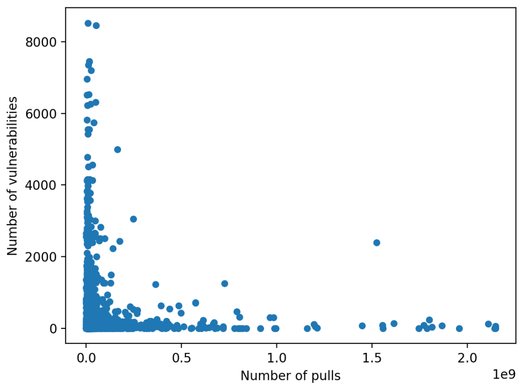


Figure 5.14: Number of pulls and number of vulnerabilities for each image

Correlation between stars and vulnerabilities

The common perception mentioned above can be rephrased and applied for the correlation between stars and vulnerabilities. The following findings are of interest. Spearman's correlation coefficient between the number of stars and the number of vulnerabilities is $r_s = -0.0335$, indicating no particular correlation. Figure 5.15 shows the scatter plot when including the number of stars instead of the number of pulls. The plot is similar to Figure 5.14, but the correlation is even weaker. The majority of the markers are gathered on the lower range of the x-axis, and along the y-axis, indicating that images with a high number of vulnerabilities generally have few stars. Also, images with a high number of stars have few vulnerabilities.

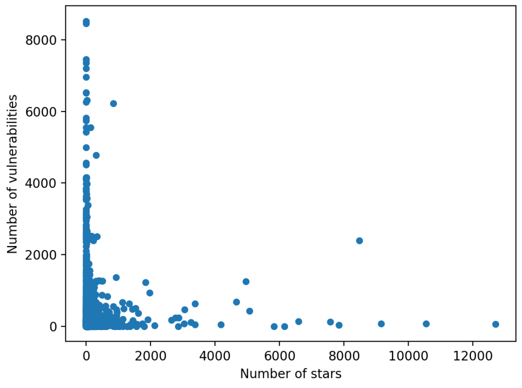


Figure 5.15: Number of stars and number of vulnerabilities for each image

Correlation between the time since last update and vulnerabilities

This correlation is calculated by computing the number of days since the last update counting back from the day we gathered the data (February 25th, 2020). The correlation is $r_s = 0.1075$, which shows a positive correlation as opposed to the other two results. However, it is too weak to conclude that there is any correlation. We present the scatter plot in Figure 5.16. The markers slightly approach an increasing line, indicating a weak tendency that there are more vulnerabilities in images that have not been updated for a long time. Still, the distribution of markers is relatively even along the x-axis with the most markers in the lower part of the y-axis, supporting that there is no correlation.

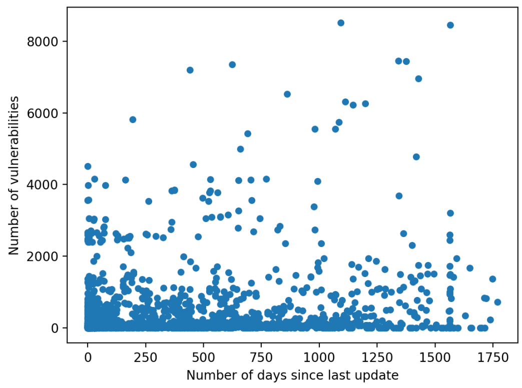


Figure 5.16: Number of days since last update and number of vulnerabilities for each image

5.4 The most severe vulnerabilities

This section will present which vulnerabilities that are the most severe, as stated in **RQ3:** *Which types of vulnerabilities are the most severe?*

5.4.1 The most represented critical vulnerabilities

The most represented severe vulnerabilities are, intuitively, the ones having the highest impact on the vulnerability landscape. Table 5.9 presents the most represented critical rated vulnerabilities in descending order. The results are obtained by counting the number of occurrences for each vulnerability ID in the critical severity level. The critical count column is the number of occurrences for a specific vulnerability. Lastly, the type(s) column presents the vulnerability type. This data is gathered from the CVE Details database [CVE20] by looking at the *Vulnerability Type(s)* attribute for each vulnerability. In cases where no vulnerability type is assigned, the name of the CWE ID is used instead, as each CWE represents a single vulnerability type.

	Vulnerability ID	Critical count	Type(s)
1	CVE-2019-10744	466	Improper Input Validation
2	CVE-2017-1000158	464	Execute Code, Overflow
3	CVE-2019-9948	378	Bypass a restriction or similar
4	CVE-2019-9636	374	Credentials Management Errors
5	CVE-2018-16487	365	Security Features
6	CVE-2018-14718	354	Execute Code
7	CVE-2018-11307	337	Deserialization of Untrusted Data
8	CVE-2018-7489	318	Execute Code, Bypass a restriction or similar
9	CVE-2016-5636	302	Overflow
10	CVE-2017-15095	295	Execute Code

Table 5.9: Top 10 most represented vulnerabilities (based on critical severity level)

5.4.2 Vulnerability characteristics

This section elaborates on the top five most represented vulnerabilities presented in Table 5.9 regarding their characteristics and common features.[4] As a general observation, the execute code is the most common vulnerability type, followed by overflow.

The most represented critical vulnerability is found 466 times throughout our scanning. It has vulnerability ID *CVE-2019-10744*, and a base score of 9.8, which is in the upper range of the critical category (to examine how base scores are determined, see Section 2.3.2). The vulnerability is related to the JavaScript library Lodash, which is commonly used as a utility function provider in relation to functional programming. This particular vulnerability is related to improper input validation and makes the software vulnerable to prototype pollution. It is affecting versions of Lodash lower than 4.17.12 [Nat19a]. In short, this means that an adversary can execute arbitrary code by modifying the properties of the Object.prototype. This is possible as most JavaScript objects inherit the properties of the built-in Object.prototype object. The fifth vulnerability on the list, *CVE-2018-16487*, is also related to Lodash and the prototype pollution vulnerability.

Further, the second, third, and fourth most represented critical vulnerabilities are related to Python vulnerabilities. The second vulnerability with vulnerability ID, *CVE-2017-1000158*, is related to versions of Python up to 2.7.13. The base

[4]Information about all vulnerabilities could be found by visiting *https://nvd.nist.gov/vuln/detail/<vulnerability ID>*

score is rated 9.8, and the vulnerability enables arbitrary code execution to happen through an integer overflow leading to a heap-based buffer overflow [Nat]. Overflow vulnerabilities could be of different types, such as heap overflow, stack overflow, and integer overflow. Heap overflow and stack overflow are related to overflowing a buffer, whereas integer overflow could lead to a buffer overflow. A buffer overflow is related to overwriting a certain allocated buffer, causing adjacent memory locations to be overwritten. Any exploitation of these kinds of vulnerabilities is typically related to the execution of arbitrary code, where the adversary is taking advantage of the buffer overflow vulnerability to run malicious code.

The third presented vulnerability with vulnerability ID *CVE-2019-9948* is affecting the Python module urllib in Python version 2.x up to 2.7.16. It is rated with a base score of 9.1. This vulnerability makes it easier to get around security mechanisms that blacklist the `file:URIs` syntax, which in turn could give an adversary access to local files such as the */etc/passwd* file [Nat19c]. The fourth vulnerability is found 374 times with vulnerability ID *CVE-2019-9636.* It is affecting both the second and the third version of Python (versions 2.7.x up to 2.7.16, and 3.x up to 3.7.2). This vulnerability is also related to the urllib module, more precisely, incorrect handling of unicode encoding. The result is that information could be sent to different hosts than intended if it was parsed correctly [Nat19b]. It has a base score of 9.8.

5.5 Vulnerabilities in packages

To investigate the origin of the most severe vulnerabilities, we seek to determine the top 10 most vulnerable packages, as well as how many images that use these packages. We will also determine the number of vulnerabilities in the most used packages. This is in accordance with **RQ4:** *Which packages contain the most severe vulnerabilities?*

5.5.1 The most vulnerable packages

Table 5.10 presents the packages that contain the most critical vulnerabilities. The critical count column is obtained by counting the total number of occurrences of critical vulnerabilities in each package. The image count column displays the number of images that use each package.

	Package	Critical count	Image count
1	jackson-databind-2.4.0	710	15
2	Python-2.7.5	520	207
3	jackson-databind-2.9.4	354	4
4	lodash-3.10.1	312	76
5	silverpeas-6.0.2	280	1
6	Python-2.7.13	248	141
7	Python-2.7.16	224	117
8	jackson-databind-2.6.7.1	215	13
9	jackson-databind-2.9.6	192	12
10	Python-2.7.12	185	107

Table 5.10: Top 10 most vulnerable packages (based on critical severity level)

Number of images that use the most vulnerable packages

To be able to get a better view of the security impact of these packages, we present Figure 5.17. It displays the number of images that use each of the packages from Table 5.10. There is a clear relation between the most vulnerable packages and the most represented vulnerabilities from Section 5.4, as expected. For example, vulnerabilities found in Python version 2.x packages and the Lodash package are both presented in Section 5.4.

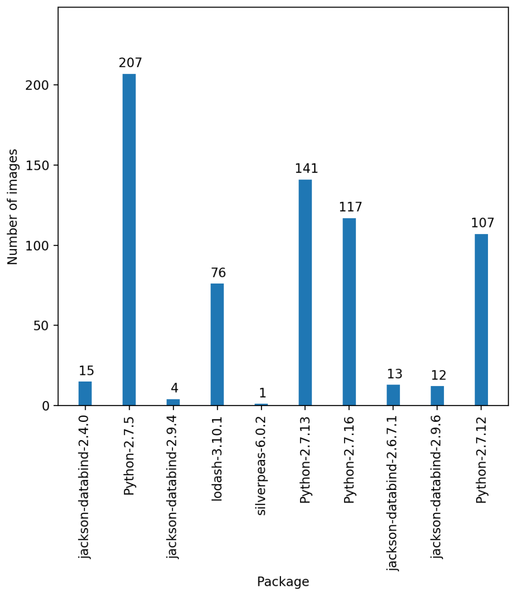


Figure 5.17: Most vulnerable packages and the number of images that use them

From Figure 5.17, it is observable that the Python packages are by far the most used, and therefore they expose the biggest impact regarding the threat landscape. The lodash-3.10.1 package is found in 76 images. This package contains the prototype pollution vulnerability affecting JavaScript code, which also is the most represented vulnerability in Table 5.9. Further, the jackson-databind package is represented with four different versions in Table 5.10 (entries 1, 3, 8, and 9). This package is used to transform JSON objects to Java objects (e.g., Lists, Numbers, Strings and Booleans), and vice versa. In total, these packages are used by 44 images, which is relatively low compared to the usage of the Python packages. The silverpeas-6.0.2 package contains 280 critical vulnerabilities and is only used by a single image: the Silverpeas image on Docker Hub.[5]

5.5.2 Vulnerabilities in popular packages

When considering the packages that have the most critical vulnerabilities (Table 5.10), some of the packages are only used by a few images (like the Silverpeas package). Therefore, Table 5.11 is presented, as it is desirable to see what the vulnerability distribution is like in the most popular packages. The table shows the most used packages and the number of vulnerabilities in them, considering all severity levels. The image count column contains the number of images that use this package.

[5]The Silverpeas image: *https://hub.docker.com/_/silverpeas*

	Package	**Critical**	**High**	**Medium**	**Low**	**Negligible**	**Unknown**	**Image count**
1	tar-1.29b-1.1	0	0	0	0	482	0	241
2	coreutils-8.26-3	0	0	0	0	240	0	240
3	libpcre3-2:8.39-3	0	0	0	0	956	0	239
4	login-1:4.4-4.1	0	0	0	0	714	0	238
5	passwd-1:4.4-4.1	0	0	0	0	708	0	236
6	sensible-utils-0.0.9	0	0	103	0	0	111	214
7	libgcrypt20-1.7.6-2+deb9u3	0	0	0	0	211	0	211
8	libgssapi-krb5-2-1.15-1+deb9u1	0	0	0	0	621	0	207
9	libk5crypto3-1.15-1+deb9u1	0	0	0	0	621	0	207
10	libkrb5-3-1.15-1+deb9u1	0	0	0	0	621	0	207

Table 5.11: Vulnerabilities in the top 10 most used packages

As observable from Table 5.11, the most used packages are not containing any critical, high, medium, or low vulnerabilities (except for one entry). However, they do contain a vast number of negligible vulnerabilities, which is of less significance from a security point of view, as mentioned in previous sections.

Chapter 6

Discussion

In this chapter, our results will first be put in relation to the respective research question they are related to by being thoroughly interpreted and discussed. Further, the limitations and validity of this study will be evaluated. Our obtained results and results from previous research will be compared in order to say something about the trend. Finally, we propose our recommendations for future work.

6.1 Interpretation of results in relation to research questions

The obtained results from Chapter 5 will be put in a broader perspective in this section, and used to answer each research question, as presented in Section 1.2.1, sufficiently. This will be achieved by interpreting, discussing, and explaining the results in relation to the research questions.

6.1.1 RQ1: How can vulnerabilities found in Docker images be systemized in order to investigate the current vulnerability landscape of Docker Hub?

Several measures and systematizations are presented in Section 5.2 as results to answer **RQ1**. In order to be specific enough, a single measure is not sufficient to say something about the vulnerability landscape as a whole. However, we do consider some of the results as of more importance when addressing the vulnerability landscape. As pointed out, the vulnerability landscape of Docker Hub is complex, but we consider the central tendency of our data set as one of the most important characteristics. The values in our data set are skewed and contain some extreme values that influence the average, which could be seen from Table 5.4, where the standard deviation for all image types is high. This is an indication that the median value of our data set gives the best view on the actual characteristics of the data. As Section 5.2.2 reveals, the median value, when omitting the negligible and unknown vulnerabilities, is 26 vulnerabilities per image. These vulnerabilities are spread across all severity levels, where most of the vulnerabilities were found in the medium severity category,[1] both when considering unique vulnerabilities and the total number of vulnerabilities, as presented in Section 5.2.1. The ratio indicates that a relatively small number of unique vulnerabilities are found in a vast number of images, for example, there are 206 unique critical vulnerabilities found 10,378 times (Table 5.3). As additional information about the vulnerability landscape, we would like to emphasize that 430 images (17.8%) in the data set do not contain any vulnerabilities, and as many as 523 images if we are considering negligible and unknown vulnerabilities as no vulnerability.

Vulnerabilities in image types

Images on Docker Hub are categorized in one of the following categories: official, verified, certified, or community. Thus, it is of importance to address how vulnerable these image types are. Especially since official, verified, and certified images are images that are reviewed by Docker, meaning they are said to fulfill requirements concerning quality, best practices, and *security*. As presented in Table 5.4 in Section 5.2.2, there are differences when considering the average and when considering the median value in the different image types. Surprisingly, certified images are the most vulnerable when considering the median value. However, when considering the average, community images are the most exposed. Hence, community images have more spread values, whereas certified images have values centered around the average. This is also reflected in Section 5.2.5, where 8 out of the top 10 most vulnerable images are community images.[2] Official images come out as the most secure image type.

These results should be seen in conjunction with the results presented in Section 5.2.3, where the vulnerabilities in each severity category in the different image types are investigated. The median value of the number of critical vulnerabilities in images is almost identical for all four image types, so the ranking of the image types is primarily based on the other vulnerability severities. The certified images come out as the most vulnerable image type, as these results are based on median value, with the most high and low rated vulnerabilities overall. Also, when considering the results from Section 5.2.6, certified images are the most vulnerable, where as many as 82% of certified images contain at least either one high or critical vulnerability. Interestingly, this number is 57% for verified images, making it the second most secure image type. Moreover, verified images also hold a significantly lower number of vulnerabilities when looking at the results from Section 5.2.3 again.

[1]When not considering the negligible and unknown vulnerabilities.
[2]Based on the total number of critical vulnerabilities in images.

Based on the fact that certified images are a subgroup of verified images that are claimed to fulfill even more requirements, these results are highly unexpected. It could be discussed whether some of the introduced image types on Docker Hub provide a false sense of security. Some of the vulnerable images could, however, be explained by the fact that the vendor deliberately keeps the software from being updated to ensure backward compatibility. There is no reason why this would affect certified images more however, other than the fact that the total number of certified images is low, and therefore, changes in the vulnerabilities will have a big impact on the results.

Contextualization with the CVE trend

In Section 5.2.9, the overall evolution in CVE vulnerabilities is shown. From Figure 5.12b, one can observe an increase of unique vulnerabilities found in verified and certified image types in 2018, whereas it is a significant decrease for community images and official images. Generally, the number of newly introduced vulnerabilities on Docker Hub is rapidly increasing between 2012 and 2020. Certified and verified images were introduced on Docker Hub in 2018, so the fact that these types became more publicly available could be the reason for the increase. Moreover, one can observe how the community and official image types follow somewhat the same curve as the overall evolution of the reported CVEs each year. The fact that the certified image type seems to follow a steep curve the last few years matches well with our obtained results previously explained. From our analysis, official images generally come out as the least vulnerable image type and certified come out as the most vulnerable image type. However, Figure 5.12b shows that official images have a higher number of unique vulnerabilities than certified. The explanation for this could be that there are more official images and that the vulnerability ratio in certified images is higher.

It needs to be pointed out that the data presented in Section 5.2.9 (and generally in this thesis), is not concerning zero-days vulnerabilities, which are vulnerabilities that are still unknown or unaddressed. It is clear that the discoveries in these kinds of analyses address only a small part of the actual vulnerability landscape, as there are a large number of vulnerabilities that are not yet known to the public. Moreover, it could be discussed what impact these vulnerabilities actually have, as Gartner predicts that 99% of vulnerabilities exploited by the end of 2020 will continue to be publicly known vulnerabilities [Moo17].

The frequency of software updating

Another important measure regarding the vulnerability landscape of Docker Hub is how often images are updated. There is no doubt that the way to make Docker Hub more secure and reduce the number of vulnerable images is to make sure that software is patched, and always up to date. Section 5.2.10 reveals that 31% of images have not been updated in the last 400 days or longer, which is problematic in terms of security. We expect to see this trend (potentially even stronger) if the rest of the images on Docker Hub were to be considered. This is due to the fact that the rest of the repositories on Docker Hub that we have not analyzed are mostly community repositories, which we have observed to be one of the least updated types, overall. Also, this thesis has focused on the latest version of images, which is expected to be the most updated version of an image. The reason that a significant portion of images is not maintained could be that many images are created for serving a specific purpose, and never maintained afterwards. This applies especially to community images. Additionally, the backward compliance, as previously mentioned, could be another reason that images are not frequently updated. Verified and official images are the most frequently updated, and community and certified images are the least frequently updated image types. Thus, these results correlate well with our previous results regarding the most vulnerable image types.

The vulnerability landscape in Microsoft images

This thesis has also investigated the vulnerabilities found in Microsoft images, as presented in Section 5.2.7. The execution and overflow vulnerabilities of the Windows operating system are known for more than two decades [SGGK07]. Those types of vulnerabilities continued to be present even in the latest versions of Microsoft Windows [Tun18]. However, the results revealed that Microsoft images contain a lower number of high and critical vulnerabilities on average and that Microsoft images have a lower vulnerability ratio. It is challenging to identify the reason for these results. This is because even though the vulnerabilities that exist in, for example, a Windows OS and in a Linux OS will be different, most of the vulnerabilities are introduced by third party software independently from the vulnerabilities introduced by the OS. However, this could be too generalizing to state, as the third party software for different OSs will not be identical, and thus, different vulnerabilities could exist. Another reason could be that most of our analyzed Microsoft images are verified images, which is also one of the most updated image types, as already mentioned.

Propagation of vulnerabilities

To understand why there are so many vulnerable images found on Docker Hub, we will discuss some of the problems with the way the Docker ecosystem works concerning the vulnerability landscape. Firstly, it is a common practice to create an image based on another image by using the `FROM` syntax in the Dockerfile. This saves the developer much time not having to rewrite already existing code. It is, for example, usual to build an image from an existing base image that provides some functionality. However, not everyone is aware that the images they use as base images potentially could be extremely vulnerable. Thus, all vulnerabilities in the parent image will be introduced in the child image. As a result, a cascading effect where vulnerabilities propagate from parent to child images will arise. Secondly, the Docker principle known as copy-on-write [And15] could be seen as problematic from a security point of view. This principle is based on new layers being added to the image when changes are conducted, and the older layers are unchanged. This means that the older versions of an image with vulnerabilities that are patched in a newer version will still exist because the content of an image is immutable after building [Kar20]. Hence, it is desirable to occasionally create a whole new image instead of updating an old one.

Concluding remarks

This section has summarized and reflected upon the most important findings in order to answer **RQ1**. This is the main research question of this thesis, and thus, the most comprehensive. In terms of how vulnerabilities found in Docker images can be systemized to investigate the current vulnerability landscape, this section has argued for the following findings as of most importance:

– When considering the median value, there are 26 vulnerabilities in each image, where we have observed a relatively small number of unique vulnerabilities found at a high ratio.

– There are 430 images that do not contain any vulnerabilities.

– The certified images are the most vulnerable image type when considering the number of vulnerabilities based on the median, and when considering the percentage of images with at least one high or critical vulnerability.

– Community images are the second most vulnerable image type based on most of the results.[3]

– Verified images are the second most secure image type.

– Official images are the most secure image type, with the least number of vulnerabilities, both based on median and average.

– Approximately 30% of images have not been updated for the last 400 days.

– Verified and official images are the most updated, and community and certified images are the least updated.

– Microsoft images are less vulnerable than other images.

[3]Community images are the most vulnerable based on the average number of vulnerabilities, but this measure is considered of less importance as previously discussed.

6.1.2 RQ2: How do image features and the number of vulnerabilities correlate in images?

To answer **RQ2**, we calculated the correlation between the number of vulnerabilities in each image and image features. When investigating the correlation, we opted for Spearman's correlation method. It could be discussed whether or not this method is the optimal one for our data set. We considered using Pearson's correlation, but it works best on normally distributed data. This was not the case for us because of the skewness of the data and extreme values. Pearson correlation only finds the linear relationship of values, and the Spearman method can find nonlinear relationships. Also, the values do not need to be normally distributed. It detects if sample data is monotonically increasing or decreasing, and was the better option for our data set.

The results presented in Section 5.3 show that the correlation coefficients were -0.1115 for the number of pulls, -0.0335 for the number of stars, and 0.1075 for the number of days since the last update. None of these results are strong enough to conclude that there is a correlation between the features. We also inspected the relationship between variables by looking at the scatter plots presented in Figures 5.14, 5.15 and 5.16. There is also considering this measure, no correlation between any of the image attributes, neither linear or nonlinear. As observable from the plots for pulls and stars, the images with the most vulnerabilities have few pulls and stars. This is a positive trait from a security perspective and an indication that security awareness among the community of Docker users is relatively high. We repeat the common perception, as stated in Section 5.3, *images with the most pulls/stars generally have few vulnerabilities, and images with the most vulnerabilities generally have few pulls/stars.* Even though there is no correlation between the number of vulnerabilities in images and either of the image features, the common perception seems to hold. On the other hand, when considering the scatter plot for the time since the last update and the number of vulnerabilities (Figure 5.16), there is no such evidence to be found. In this case, one could argue that the randomness of the markers in this scatter plot is exposing a threat. This is because users might intuitively think they are more secured by choosing a recently updated image. As this is not the case, these images may create a false sense of security. As such, the results from Section 5.2.10, which revealed that 31.4% of images had not been updated for 400 days or longer, represent a smaller security threat than previously anticipated.

It is difficult to determine if our analyzed data set fits the target population of Docker Hub, especially since we have analyzed the most recognized images. Thus, we want to point out that an analysis of a significantly larger number of images could have revealed a correlation between the variables.

Concluding remarks

As some concluding remarks to answer **RQ2**, we emphasize the following findings of the most importance.

- There is no correlation between the number of vulnerabilities and the evaluated image features (i.e., the number of pulls, the number of stars, and the last update time).
- The images with many vulnerabilities generally have few pulls and stars.
- The images with many pulls and stars generally have few vulnerabilities.
- There is no significant relationship found between the number of vulnerabilities and the time since the last update.

6.1.3 RQ3: Which types of vulnerabilities are the most severe?

It is highly relevant to address the most severe vulnerabilities when analyzing the vulnerability landscape of Docker Hub, which relates to **RQ3**. As the data presented in Sections 5.4.1 and 5.4.2 reveal, the most severe vulnerability is represented 466 times with critical severity level and is found in the Lodash library. Actually, two out of the top five most represented vulnerabilities are found in the Lodash library. Different conditions could explain this. Firstly, the Lodash library is an extremely popular JavaScript library used by 4.35 million projects on GitHub [Tal19]. Secondly, these two vulnerabilities are relatively newly discovered, as both were published for the public in 2019. This can be seen in relation to the results presented in Figure 5.13, where a large number of images have not been updated since before 2019. These observations explain why so many images contain the Lodash vulnerabilities.

The remaining three vulnerabilities out of the five most represented ones are Python vulnerabilities. The second most represented vulnerability is found 464 times (Section 5.4.1), meaning that there are only a couple occurrences that separate the top two most represented vulnerabilities. The second and third most represented vulnerabilities are affecting the second version of Python, which is not maintained after April 2020 [Pyt19]. As a consequence, many vulnerabilities in older versions of Python will not be fixed. Reasons such as policies, lack of resources, backward compatibility, and company restrictions may stop developers from upgrading to Python 3. The fact that Python is a widely used programming language is making the attack surface huge.

The number of occurrences of critical rated vulnerabilities was used as a measure. As an alternative measure, one could also include the high rated vulnerabilities. However, when ordering the data based on the sum of critical and high vulnerabilities, the high rated vulnerabilities profoundly dominated the result so that the top ten vulnerabilities contained few critical values. Because the critical vulnerabilities expose the highest impact if exploited, there is no doubt that they are the most severe vulnerabilities. Indeed, this made is desirable to focus on the critical rated vulnerabilities. As presented in Table 5.9, *Execute Code* is the most represented vulnerability type with four occurrences, followed by *Overflow*, with two occurrences. Data presented in the CVE Details database [CVE19b], reveals that execute code is altogether the most common vulnerability type, where overflow is the third most common vulnerability type. Based on these observations, and the results from our analysis, we expect to see these types of vulnerabilities to be the most represented in the rest of Docker Hub as well. These results are important in addressing what kind of vulnerabilities that are considered the biggest threat, with the objective to raise awareness and make both enterprises and other users aware of flaws in the software they are using.

Concluding remarks

To properly answer **RQ3**, based on the results presented in Section 5.4 and the aspects discussed here, there is no doubt that the most severe vulnerabilities are the ones found in popular and much used software. The following findings are important.

- The most severe and represented vulnerabilities are found in the Lodash library and in Python packages, and thus coming from two of the most popular scripting languages, JavaScript and Python.
- Vulnerabilities related to execution of code and overflow are the most frequently found critical vulnerabilities.

6.1.4 RQ4: Which packages contain the most severe vulnerabilities?

When inspecting the results from Section 5.5 that answers **RQ4**, there is no surprise that we found the most severe vulnerabilities, discussed in the previous section, to be highly related to the most vulnerable packages. Moreover, as vulnerable packages introduce severe vulnerabilities, this perspective is accordingly also of importance to address.

As presented in Section 5.5, we found the most critical vulnerabilities in jackson-databind packages, Python packages, in the Silverpeas package, and in the Lodash package. Not surprisingly, both Python vulnerabilities and Lodash vulnerabilities were found as the most severe vulnerabilities in the previous section. However, the most vulnerable package is the jackson-databind-2.4.0 package, with overwhelming

710 critical vulnerabilities, followed by Python-2.7.5 with 520 critical vulnerabilities. The Python 2.x packages are by far the most used ones, as seen in Figure 5.17. It is expected that when a widely used programming language is ended, it will take much time before all code is migrated to the newest version (i.e., Python version 3). The ending of Python 2.x is, thus, a contributing factor to the high usage of vulnerable Python packages.

Section 5.5.2 presents how vulnerable the most used packages are. These results reveal that the most popular packages do not contain a large number of severe vulnerabilities, which is positive when considering the security of Docker Hub. As a final observation, many of the vulnerable packages might not be in use anymore, but are still found in images. Hence, we want to stress that packages that are not explicitly removed from an image, even though it is not in use anymore, is making the image vulnerable.

Concluding remarks

The comments made in this section, as well as the results from Section 5.5, will be used as a foundation to answer **RQ4**. We have made the following observations.

- The Python 2.x packages and jackson-databind packages were found to be the most vulnerable packages.
- Out of the most vulnerable packages, the Python 2.x packages are the most used.
- Widely used packages were generally discovered to be quite secure, containing few severe vulnerabilities.

6.2 Limitations and validity of this study

The correctness of this study and factors limiting the results of this study need to be evaluated. Hence, this section will present the factors that are considered the most important in relation to limitations.

6.2.1 The impact of false positives and false negatives

There are issues related to how container scanners work and how the scanner's output might give false positive and false negative results. It is of high importance to see our results in light of these issues. A false positive result is when a scanner incorrectly reports a vulnerability, while a false negative result is when a present vulnerability is not reported.

Anchore Engine works by collecting vulnerability data from sources like NVD, RedHat, and Debian. The gathered vulnerability data is checked up against the results from scanning each image layer. Concerning false positives, the problems occur when the scanner does not have enough information to determine if the vulnerability is present or not, this could be due to authentication issues making Anchore Engine unable to get the correct software version. Moreover, false positives could also occur when a vendor has fixed a vulnerability and also included other updates in the new version. A user could choose to pick only the fix out of these updates and rename this version of the packet to something else. This packet would therefore be found vulnerable by the scanner because the packet is not of a certain version, even though the vulnerability is fixed. Furthermore, a vulnerability could exist in software, but only be exploitable in combination with some other functionality or component. Thus, an existing vulnerability in software does not mean that it is a vulnerability in this specific software, making the reporting of these kinds of vulnerabilities false positives [Ric17]. These presented scenarios are just *some* examples of why container scanners report false positive vulnerabilities in order to concretize for the reader.

One could argue that it is more critical not to report actual vulnerabilities than to report vulnerabilities that are not actually exposing the system. Therefore, false negatives could be considered even more important to avoid. There are different reasons why a container scanner does not detect all vulnerabilities, for instance, zero-day vulnerabilities, rapid updates from a vendor, authentication issues, and network glitches [Tri17].

When tuning container scanners, it is essential to tune the scanners such that the balance between false positives and false negatives is appropriate. As an explanation, when accepting a lower number of false positives, the number of false negatives is likely to go up. The fact that Anchore Engine (and other container scanners) is a static analyzing tool[4] contributes to a higher amount of false positives and false negatives, because the runtime behaviour is something different. There is no doubt that both false positives and false negatives are affecting our obtained results in this project. Based on the previously discussed aspects, the results could be affected in two ways. Either the number of reported vulnerabilities is higher than the actual number, or there are vulnerabilities that are not reported. Nevertheless, it is expected that the impact they have is limited due to the size of our data set.

6.2.2 The CVSS score

As all results presented in this thesis are based on the CVSS score system as ranking, the correctness and implication of doing so are discussed. The main problem related to using the CVSS score as a substitute for *risk* is that the vulnerability is taken out

[4]Code is analyzed before it is run.

of its context and given a rating, when in fact, the context is rather important for determining the risk [SQ17]. The CVSS score is based on how easy something is to exploit calculated up against the impact of exploiting. However, as the attacker's focus is to achieve their malicious goals, rather than exploiting what is easy, the CVSS score should not be used as the only measure for determining what vulnerabilities are the most severe.

We would like to point out that in analyses like the one performed in this thesis, where the interest is to determine the vulnerability landscape as a whole, CVSS is the most informative measure of risk. This is based on its structured way of categorizing vulnerabilities in addition to its broad usage. Hence, we consider the CVSS score as the most appropriate measure for categorizing vulnerabilities, but we want the reader to be aware of the implications related to using the CVSS score as the only measure of risk.

6.2.3 Restraints of the data set

Some characteristics of our obtained data set could affect the validity of the results. To start with, we only analyzed a portion of all images on Docker Hub. It could be problematic to generalize our results to apply for all images. Nevertheless, as our analysis included the latest version of the most recognized images,[5] it is expected that the actual vulnerability landscape of Docker Hub is even worse than what is presented in this thesis. Another aspect to consider is that we chose to go for the uppermost version of the image when the latest tag was missing. Especially cases where another version than the uppermost is more frequently updated would impact the results. Further, the fact that an overall low number of verified images were analyzed, and that most of them were Microsoft images, is of importance to mention. One could argue that our results give a better view of how vulnerable Microsoft images are, rather than how vulnerable verified images are.

In some of the presented results, we disregard the negligible and unknown categories to focus on the vulnerabilities with higher severity levels and to comply with CVSS v3.1. It could be discussed if this has any impact on the obtained results. However, as it is desirable to focus on the most severe vulnerabilities, the impact is likely to be low.

6.2.4 Inconsistency in Docker Hub

Throughout the work on this thesis, we have found the inconsistency of Docker Hub as highly problematic and as something that has limited our obtained results. Firstly,

[5]All official, a portion of verified and certified images, and more than 2000 of the most popular community images.

Docker Inc. does not provide a complete and well documented endpoint with data about all Docker Hub images. Hence, we needed to access several endpoints to gather the desired information. This approach made it challenging to gather image data at the exact same time, and in general, it is not ideal to gather the same data from different sources. Secondly, not all image types contain all desired data (e.g., verified and certified images do not contain star ratings, and many are lacking data about the number of pulls), which means that some information is lacking from our data set. Thirdly, on Docker Hub, it is only possible to navigate through the first 100 pages of images. This restriction makes it challenging to discover the rest of the images on Docker Hub, as they have to be found through search words or using other techniques. We tried contacting Docker Inc. about this, as well as asking in Docker community forums; however, we received no response. Considering these remarks, the high inconsistency in Docker Hub *have* affected the obtained image data, and thus, affected the rest of the performed analysis.

6.3 Comparison between our results and previous studies

To be able to say something about trends in the vulnerability landscape of Docker Hub, this section provides a comparison between our results and results obtained in previous research. It is highly relevant to see in what direction Docker Hub is heading in terms of its vulnerability landscape, and whether it is becoming more secure. The previous research in this field that will be used as comparison is by BanyanOps [GDT15] in 2015, Shu et al. [SGE17] in 2017 and Socchi and Luu [SL19] in 2019, which all are explicitly described in Section 2.4. We want to emphasize that their research, as well as our research, is difficult to compare directly, based on the fact that there are many differences in the data set and applied methodology. We still see it as interesting to do this kind of comparison in the discussion to discover the trends, as briefly mentioned in the introduction of this thesis.

One essential difference to be aware of is that BanyanOps and Shu et al. use the older CVSS v2.0 to categorize vulnerabilities in severity levels. However, Socchi and Luu, and the research performed in this thesis, is based on the newer CVSS v3.1 to categorize vulnerabilities. The most apparent difference is the two new severity categories in CVSS v3.1, named critical and none. In CVSS v2.0, the high severity level has base score range from 7.0 to 10.0, while in CVSS v3.1, the high severity level ranges from 7.0 to 8.9 and the critical ranges from 9.0 to 10.0. As such, our high and critical vulnerabilities combined are equivalent to their (BanyanOps and Shu et al.) high rated vulnerabilities. Additionally, our research includes negligible and unknown as severity categories instead of the none category from CVSS v3.1 because of how our applied scanning tool, Anchore Engine, classifies vulnerabilities.

The trend in the percentage of images with at least one high rated vulnerability

All results regarding the percentage of images that contain at least one high rated vulnerability are presented in Table 6.1.[6] In 2015, BanyanOps found that 36% of official images and 40% of community images contained high rated vulnerabilities. Only two years later, in 2017, Shu et al. could report that more than 80% of both official and community images contained high rated vulnerabilities, which is a massive increase. Since their research, Docker Inc. has introduced two new image categories: verified and certified. Our results, as presented in Section 5.2.6, show that since 2017 (the analysis of Shu et al.), the number of images with at least one high rated vulnerability has decreased in both official and community images (see Table 6.1). Generally speaking, there has been an increase in the number of vulnerabilities since 2015 (BanyanOps' analysis). Even when considering the improvements in vulnerabilities found in official and community images today, compared to the results from 2017 (study by Shu et al.), there still is a massive number of vulnerable images on Docker Hub. It should be stressed that these are vulnerabilities that are potentially easy to exploit, and that could have a high impact if exploited.

	2015	**2017**	**2020**
Official	36%	>80%	46%
Community	40%	>80%	68%
Verified	-	-	57%
Certified	-	-	82%

Table 6.1: Percentages of images with at least one high rated vulnerability. Based on data from [GDT15] [SGE17] and the results presented this thesis.

The days since last update

Further, Shu et al. found that 50% of all images had not been updated for the last 200 days, and 30% of images had not been updated in 400 days. Our results from three years later, presented in Section 5.2.10, reveal that 44% of images have not been updated in 200 days or more, and 31% of images have not been updated for the last 400 days. Hence, there is no indication of any significant improvement in how often images are updated. Shu et al. also investigated the more frequently updated images. They found that 20% of all official images and 10% of community images had been updated during the last 14 days. From our analysis, 51% of official images and 27% of community images had been updated in the last 14 days. However, it is important to mention that Shu et al. analyzed all versions of images, whereas we

[6] For clarity, *high* in the CVSS v2.0 format. For comparison, our analysis' vulnerability severities are converted from CVSS v3.1 to v2.0 by including both high and critical vulnerabilities.

have focused on the latest version of images. The latest version of images is usually the most updated version, so these results are not entirely comparable. Additionally, Shu et al. presented results for official images with the latest tag, and then the percentage for images that had been updated for the last 14 days increased to 86%.[7] Thus, the trend seems to be that official images are not as frequently updated as before.

The tendency in the average number of vulnerabilities in different image types

We examine the trend in the average number of vulnerabilities per image found in the latest version of the different image types, and the results are presented in Table 6.2. The data is based on results from Shu et al. in 2017 [SGE17] and the data from 2019 is gathered from Table 7.1 on page 88 in [SL19]. Lastly, the data from 2020 is based on the results presented in Section 5.2.2 of this thesis.[8]

To say something about the trend, the total average number of vulnerabilities in the latest version of official images did heavily increase from 2017 to 2019 (from an average of 70 vulnerabilities to an average of over 170 vulnerabilities). However, based on our analysis, the average number of vulnerabilities in official images is back at 70, which means that it has dropped with 100 vulnerabilities in just one year. Since there only are ~160 official images on Docker Hub in total, small changes in the vulnerability landscape will have a high impact on the result. Additionally, the fact that we have analyzed more official images could also influence the result. When considering community images, the number of vulnerabilities decreased from 2017 to 2019, where it went from more than 180 to more than 150 vulnerabilities in each image on average. As found in our analysis, the average is still at 150 vulnerabilities. Moreover, verified images contain fewer vulnerabilities now than last year, with a difference of ~60 vulnerabilities. Since Socchi and Luu did not analyze Microsoft images, and most of our verified images are Microsoft images, these results are problematic to compare directly. However, our results do reveal that Microsoft images tend to contain fewer vulnerabilities than other verified images. Lastly, certified images have become more vulnerable the last year based on this measure, with an increase from more than 30 vulnerabilities to more than 90 vulnerabilities in each image on average.

To summarize, the different image categories are heading in different directions from a security point of view. Official images seem to be just as vulnerable as they were three years ago. While community and verified images have had a decrease in

[7]This percentage is 51% from our analysis.
[8]The total number of vulnerabilities is disregarding the unknown and negligible categories.

the average number of vulnerabilities, certified images are becoming more vulnerable this last year.

	2017	**2019**	**2020**
Official	>70	>170	>70
Community	>180	>150	>150
Verified	-	>150	>90
Certified	-	>30	>90

Table 6.2: The average number of vulnerabilities found in each image type (the latest image version). Based on data from [SGE17] [SL19] and the results presented in this thesis.

The median value

As some final comments, if median is considered as measure, our median value (presented in Table 5.4 in Section 5.2.2) in all image types are significantly lower than than the median from Shu et al. (presented in Table 3 in [SGE17]). Generally, our obtained median values are also significantly lower than Socchi and Luu's results, as presented in Table 7.1 in [SL19]. The big differences between the average and the median in our results are explained by extreme values that influence the average, and because our values are more spread, as also seen from the standard deviation. For example, the highest number of vulnerabilities in a single community image in our results is 6,509, compared to 1,779 in the results from Shu et al., and 1,792 from Socchi and Luu. Also, many entries with a low number of vulnerabilities contribute to lowering the median remarkably in our results. Hence, with regards to the median value, the trend seems to be that most images contain a lower number of vulnerabilities than before.

6.4 Future work

This thesis has presented a thorough analysis of the current vulnerability landscape of Docker Hub images. For future work, we present our recommendations for improvements and other research areas related to this field. First of all, we highly recommend to analyze all 3.5 million images on Docker Hub to get a more realistic view of the vulnerability landscape of Docker Hub in its entirety. One aspect that needs to be addressed in regards to this is that the accessibility of a large number of images on Docker Hub is not sufficient. We want to use this opportunity to encourage two improvements: a complete and well-documented endpoint for image data gathering, and an improvement on the Docker Hub web page to make it possible to access all images through navigation.

Secondly, we suggest that future analysis should run over a more extended period. The previous studies conducted in this field, as well as this thesis, have only analyzed vulnerabilities in Docker Hub images captured from one single data gathering. Thus, changes in the data set over time are still not investigated. As Docker Hub is in constant change, we suggest that future work will focus on this aspect. This type of analysis could reveal more in-depth details about the characteristics and evolution of the vulnerability landscape.

Lastly, we suggest future work to be targeting the false positives and false negatives in container scanners. Previous work has been done in the field of using machine learning to reduce the number of false positive software vulnerabilities, as proposed in [GPSG18]. We see it as beneficial to also integrate machine learning into container scanners, and propose this research field as of importance in future work. The use of machine learning would mainly contribute in reducing the amount of false negatives and false positives, to get an improved analysis result.

Chapter 7 Conclusion

The scope of this thesis was to answer the following research questions:

RQ1: *How can vulnerabilities found in Docker images be systemized in order to investigate the current vulnerability landscape of Docker Hub?*

RQ2: *How do image features and the number of vulnerabilities correlate in images?*

RQ3: *Which types of vulnerabilities are the most severe?*

RQ4: *Which packages contain the most severe vulnerabilities?*

To answer our first research question, several findings are of importance. As discussed in Chapter 6, the median value of vulnerabilities in each image is 26, where a relatively small number of unique vulnerabilities are found at a high frequency. When considering the newly introduced CVEs each year on Docker Hub, it is evident that the trend from the last few years is that the number of new vulnerabilities is rapidly increasing. If these discoveries are persistent, it indicates that the total number of vulnerabilities in Docker images is increasing even faster. Our results further show that certified is the most vulnerable image type, followed by community and then verified. Official images are the most secure. Furthermore, we have discovered that approximately 30% of images have not been updated for the last 400 days and that Microsoft images are found to be less vulnerable compared to all other images. Lastly, we observed that 430 analyzed images do not contain any vulnerabilities. We conclude that these findings are the most important to describe the current vulnerability landscape of Docker Hub.

The focus of the second research question is to investigate the correlation between image features and the number of vulnerabilities. We conclude that there is, in fact, no significant correlation between the vulnerability count and any of the image features (i.e., the number of pulls, stars and days since last update). From the discussion of the third research question, it is clear that vulnerabilities found in the Lodash library and Python packages, originating from two of the most popular scripting languages: JavaScript and Python, are the most severe. As such, vulnerabilities found in frequently used software are the most severe, and more specifically, vulnerabilities related to execution of code and overflow are the most frequently represented. The following discoveries are of importance to answer the final research question of this thesis. The Python 2.x packages and jackson-databind packages were found to hold the highest number of severe vulnerabilities. Also, we discovered that the widely used packages generally contain few severe vulnerabilities.

The discoveries and work done in this thesis are composed of several contributions. The main contribution is the new insights into the vulnerability landscape of Docker Hub. These findings aim to raise awareness in the community and have a preventive impact on vulnerability management. The developed software is an additional contribution. Our implemented software is thoroughly documented and explained in this thesis, and also provided in a public repository on GitHub.[1] These aspects ensures that this research is fully reproducible. The software is adaptable and can be used for future research. Lastly, as a final contribution, a conference paper summarizing the work done in this thesis has been submitted and accepted to the SAM2020 conference (Appendix H).

[1]Repository on GitHub: *https://github.com/katrinewi/Docker-image-analyzing-tools*

References

[And15] Charles Anderson. *Docker.* IEEE Software, 2015. Accessed: 3. Feb 2020.

[Arc20] ArchWiki, https://wiki.archlinux.org/index.php/Linux_Containers. *Linux Containers*, 2020. Accessed: 17. Apr 2020.

[BK10] Diane Barrett and Gregory Kipper. *Virtualization Technique.* ScienceDirect, https://www.sciencedirect.com/topics/computer-science/virtualization-technique, 2010. Accessed: 17. Apr 2020.

[CMDP16] Theo Combe, Antony Martin, and Roberto Di Pietro. *To Docker or Not to Docker: A Security Perspective.* IEEE Cloud Computing, 2016. Accessed: 24. Jan 2020.

[CSR17] Jeeva S. Chelladhurai, Vinod Singh, and Pethuru Raj. *Learning Docker*, pages 96, 111. Packt Publishing, 2017.

[CVE19a] CVE Details, https://www.cvedetails.com/browse-by-date.php. *Browse Vulnerabilities By Date*, 2019. Accessed: 21. Apr 2020.

[CVE19b] CVE Details, https://www.cvedetails.com/vulnerabilities-by-types.php. *Browse Vulnerabilities By Type*, 2019. Accessed: 5. May 2020.

[CVE20] CVE Details, https://www.cvedetails.com/. *CVE Details*, 2020. Accessed: 20. Apr 2020.

[CZC09] Zhongqiang Chen, Yuan Zhang, and Zhongrong Chen. *A Categorization Framework for Common Computer Vulnerabilities and Exposures.* Oxford University Press on behalf of The British Computer Society, 2009. Accessed: 28. Jan 2020.

[Dat18] Datadog, https://www.datadoghq.com/docker-adoption/. *8 surprising facts about Docker adaption*, 2018. Accessed: 17. Apr 2020.

[Doca] Docker Inc., https://docs.docker.com/registry/introduction/. *About Registry.* Accessed: 27. May 2020.

[Docb] Docker Inc., https://docs.docker.com/get-started/overview/. *Docker overview.* Accessed: 17. Apr 2020.

[Docc] Docker Inc., https://docs.docker.com/engine/security/security/. *Docker security.* Accessed: 15. Apr 2020.

[Docd] Docker Inc., https://docs.docker.com/registry/spec/api/. *HTTP API V2.* Accessed: 18. Feb 2020.

[Doce] Docker Inc., https://www.docker.com/resources/what-container. *What is a Container?* Accessed: 31. Jan 2020.

[FB08] William Fox and Mohamed Saheed Bayat. *A Guide to Managing Research*, page 8. Juta and Company Ltd, 2008.

[FIR19] FIRST, https://www.first.org/cvss/specification-document. *Common Vulnerability Scoring System version 3.1: Specification Document*, 2019. Accessed: 3. Feb 2020.

[FO19] David Fiser and Alfredo Oliveira. *Why Running a Privileged Container in Docker Is a Bad Idea.* Trend Micro Inc., https://blog.trendmicro.com/trendlabs-security-intelligence/why-running-a-privileged-container-in-docker-is-a-bad-idea/, 2019. Accessed: 15. Apr 2020.

[GDT15] Jayanth Gummaraju, Tarun Desikan, and Yoshio Turner. Over 30% of Official Images in Docker Hub Contain High Priority Security Vulnerabilities. Technical report, BanyanOps, 2015. Accessed: 5. Mar 2020.

[Gol14] Ben Golub. *Announcing Docker Hub and Official Repositories.* Docker Inc., https://www.docker.com/blog/announcing-docker-hub-and-official-repositories/, 2014. Accessed: 15. May 2020.

[GPSG18] S. Gowda, D. Prajapati, R. Singh, and S. S. Gadre. *False Positive Analysis of Software Vulnerabilities Using Machine Learning.* IEEE International Conference on Cloud Computing in Emerging Markets (CCEM), 2018. Accessed: 11. May 2020.

[Hew19] Jeffrey Hewitt. *3 Critical Mistakes That IO Leaders Must Avoid With Containers.* Gartner, The address of the publisher, 2019. Accessed: 15. Apr 2020.

[Hil19] Zach Hill. *Feeds Overview.* Anchore, https://docs.anchore.com/current/docs/engine/usage/cli_usage/feeds/, 2019. Accessed: 17. Mar 2020.

[HW19] Malene Helsem and Katrine Wist. An extensive analysis of the current threat landscape in Docker Hub images. Project report in TTM4502, Department of Information Security and Communication Technology, NTNU – Norwegian University of Science and Technology, Dec. 2019.

[Kar20] Dipto Karmakar. *How to find and fix Docker container vulnerabilities in 2020.* FreeCodeCamp, https://www.freecodecamp.org/news/how-to-find-and-fix-docker-container-vulnerabilities-in-2020/, 2020. Accessed: 7. May 2020.

[LOHS05] Ann Lehman, Norm O'Rourke, Larry Hatcher, and Edward Stepanski. *JMP for Basic Univariate and Multivariate Statistics: A Step-by-Step Guide.* SAS Institute, 2005.

[McC20] Shona McCombes. *Correlational research.* Scribbr, https://www.scribbr.com/methodology/correlational-research/, 2020. Accessed: 25. May 2020.

[Mic17] Microsoft Corporation, https://docs.microsoft.com/en-us/cpp/cpp/namespaces-cpp?view=vs-2019. *Namespaces (C++)*, 2017. Accessed: 27. May 2020.

[MIT19] The MITRE Corporation, https://cve.mitre.org/about/faqs.html#what_is_cve_id. *Frequently Asked Questions*, 2019. Accessed: 3. Feb 2020.

[Moo17] Susan Moore. *Focus on the Biggest Security Threats, Not the Most Publicized.* Gartner, https://www.gartner.com/smarterwithgartner/focus-on-the-biggest-security-threats-not-the-most-publicized/, 2017. Accessed: 29. Apr 2020.

[Mor18] Jeff Morgan. *Introducing the New Docker Hub.* Docker Inc., https://www.docker.com/blog/the-new-docker-hub/, 2018. Accessed: 15. May 2020.

[Nat] National Institute of Standards and Technology (NIST), https://nvd.nist.gov/vuln/detail/CVE-2017-1000158, year = 2019, note = Accessed: 27. Mar 2020. *CVE-2017-1000158 Detail.*

[Nat19a] National Institute of Standards and Technology (NIST), https://nvd.nist.gov/vuln/detail/CVE-2019-10744. *CVE-2019-10744 Detail*, 2019. Accessed: 27. Mar 2020.

[Nat19b] National Institute of Standards and Technology (NIST), https://nvd.nist.gov/vuln/detail/CVE-2019-9636. *CVE-2019-9636 Detail*, 2019. Accessed: 27. Mar 2020.

[Nat19c] National Institute of Standards and Technology (NIST), https://nvd.nist.gov/vuln/detail/CVE-2019-9948. *CVE-2019-9948 Detail*, 2019. Accessed: 27. Mar 2020.

[Nat20a] National Institute of Standards and Technology (NIST), https://csrc.nist.gov/glossary/term/vulnerability. *Glossary - vulnerability*, 2020. Accessed: 28. May 2020.

[Nat20b] National Institute of Standards and Technology (NIST), https://nvd.nist.gov/vuln-metrics/cvss. *NVD - Vulnerability Metrics*, 2020. Accessed: 27. May 2020.

[NKK17] Sarang Na, Taeeun Kim, and Hwankuk Kim. *A Study on the Classification of Common Vulnerabilities and Exposures using Naïve Bayes.* Springer International Publishing AG, 2017. Accessed: 28. Jan 2020.

[Off19] Official Anchore Engine Github Repository, https://github.com/anchore/anchore-engine. *Anchore Engine*, 2019. Accessed: 12. Feb 2020.

[Ora20a] Oracle Corporation, https://dev.mysql.com/doc/refman/8.0/en/what-is-mysql.html. *MySQL 8.0 Reference Manual - 1.3.1 What is MySQL?*, 2020. Accessed: 21. Feb 2020.

[Ora20b] Oracle Corporation, https://dev.mysql.com/doc/refman/8.0/en/workbench.html. *MySQL 8.0 Reference Manual - Chapter 31 MySQL Workbench*, 2020. Accessed: 21. Feb 2020.

[Pyt19] Python.org, https://www.python.org/psf/press-release/pr20191220/. *Python Software Foundation: Press Release 20-Dec-2019*, 2019. Accessed: 27. Mar 2020.

[Ric17] Liz Rice. *Three Overlooked Lessons about Container Security.* The New Stack, https://thenewstack.io/three-overlooked-lessons-container-security/, 2017. Accessed: 7. May 2020.

[San19] Nandhini Santhanam. *Registry v1 API Deprecation.* Docker Inc., https://www.docker.com/blog/registry-v1-api-deprecation/, 2019. Accessed: 21. Feb 2020.

[SBW17] Daniel Schatz, Rabih Bashroush, and Julie Wall. *Towards a More Representative Definition of Cyber Security.* The Association of Digital Forensics, Security and Law (ADFSL), 2017.

[SGE17] Rui Shu, Xiaohui Gu, and Willian Enck. *A Study of Security Vulnerabilities on Docker Hub.* CODASPY '17: Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy, 2017. Accessed: 5. Mar 2019.

[SGGK07] Nenad Stojanovski, Marjan Gusev, Danilo Gligoroski, and Svein J Knapskog. Bypassing data execution prevention on Microsoft Windows XP SP2. In *The Second International Conference on Availability, Reliability and Security (ARES'07)*, pages 1222–1226. IEEE, 2007.

[SL19] Emilien Socchi and Jonathan Luu. A Deep Dive into Docker Hub's Security Landscape - A story of inheritance? Master's thesis, University of Oslo (UiO), 2019. Accessed: 5. Mar 2019.

[SQ17] Brook Schoenfield and Damian Quiroga. *Don't Substitute CVSS for Risk: Scoring System Inflates Importance of CVE-2017-3735.* McAfee Labs, https://www.mcafee.com/blogs/other-blogs/mcafee-labs/dont-substitute-cvss-for-risk-scoring-system-inflates-importance-of-cve-2017-3735/, 2017. Accessed: 11. May 2020.

[Suk96] Suphat Sukamolson. *Fundamentals of quantitatice research.* PhD thesis, Chulalongkort University, Language Institute, 1996. Accessed: 10. Feb 2020.

[Tal19] Liran Tal. *Snyk research team discovers severe prototype pollution security vulnerabilities affecting all versions of lodash.* Snyk, 2019. Accessed: 4. May 2020.

[Tri17] Tripwire, https://www.tripwire.com/state-of-security/vulnerability-management/myth-false-positives-vulnerability-assessments/. *The Myth of "False Positives" in Vulnerability Assessments*, 2017. Accessed: 8. May 2020.

[Tun18] Liam Tung. *Windows security: Microsoft issues fix for critical Docker tool flaw, so patch now.* ZDNet, https://www.zdnet.com/article/windows-security-microsoft-issues-fix-for-critical-docker-tool-flaw-so-patch-now/, 2018. Accessed: 4. june 2020.

[ZCO11] Zu Zhang, Doina Caragea, and Xinming Ou. *An Empirical Study on Using the National Vulnerability Database to Predict Software Vulnerabilities.* Springer, Berlin, Heidelberg, 2011.

[ZTA$^+$19] Nannan Zhao, Vasily Tarasov, Hadeel Albahar, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Amit S. Warke, Mohamed Mohamed, and Ali R. Butt. *Large-Scale Analysis of the Docker Hub Dataset.* 2019 IEEE International Conference on Cluster Computing (CLUSTER), 2019. Accessed: 27. Jan 2020.

Appendix A
Scripts prerequisites

In order to use our scripts, a few requirements need to be met. This appendix will explain how to get started. Please note that these steps are based on how to install dependencies on a Unix system.

A.1 Web scraper

The code for the web scraper is presented in Appendix B and the script is written in Python 3. The next section presents the needed software for the web scraper to work.

Prerequisites

– **Selenium:** An explanation of how to to install Selenium is given here: *https://pypi.org/project/selenium/*

– **Geckodriver:** The Geckodriver executable can be downloaded from the following page: *https://github.com/mozilla/geckodriver/releases*. In our scraper script, we specify the driver on line 34. The path must be changed to the location of the Gecodriver executable on the user's computer. Alternatively, it can be added to the PATH by placing it in the /usr/bin or /usr/local/bin folder, and remove everything inside the brackets on line 34 in the script. It is important to use versions of Selenium, Geckodriver, and Firefox that are compatible.

Behavior

The scraper creates two files: image-names.txt and image-info.csv, and writes the gathered data to them. If these files already exist, the content inside will be overwritten.

A.2 API scripts

The scripts for accessing the Docker Registry HTTP API V1 and V2 is found in Appendix C and in Appendix D. The scripts are written in Python 3. The API script for the first version of the API supports verified and certified images (Appendix C), where the API script for the second version of the API supports official and community images (Appendix D).

Prerequisites

urllib.request, urllib.error, and json are built in packages in Python 3, so there are no required installations needed to run the API scripts.

Behavior

The API scripts create two files each: results_apiv1.csv and failed_apiv1.txt, and results_apiv2.csv and failed_apiv2.txt. The gathered data is written to these files. If they already exist, the content inside will be overwritten. Additionally, the scripts take the image_names.txt file as input, which constitutes of image names separated by line shift. If this file is not to be found, the scripts will not run.

A.3 Automate analysis script

The bash script for automating the analysis with the Anchore Engine scanner is found in Appendix E.

Prerequisites

- **Anchore Engine:** For installation instructions of Anchore Engine, visit *https://github.com/anchore/anchore-engine*. To install Anchore Engine, Docker and Docker Compose is required.

- **Anchore Engine CLI:** The following page shows the installation guide for the command line interface for Anchore Engine: *https://github.com/anchore/anchore-cli*

- **Docker:** For installation instructions of Docker, visit *https://docs.docker.com/get-docker/*

- **Docker Compose:** For installation instructions of Docker Compose, visit *https://docs.docker.com/compose/install/*

Behavior

In order to run the script, Docker Compose is required to run. Docker Compose is started with the following command: `docker-compose up -d`. The script will first try to rename the vuln.csv and the failed.txt files. Because of the possible long run time of the script, it is important that this content is not overwritten by mistake. Thus, an error will be outputted if these files are not found, but the script will continue to run as expected. The script takes the image_names.txt file as input, which constitutes of image names separated by line shift. If this file is not to be found, the script will not run. As a final notice, Docker is by default running as root, and thus, needs to be run using the SUDO command. We highly recommend running Docker as a non-root user.

Appendix B

Script for web scraper

```python
################################################################################
#
# Web scraper for gathering images with their image type from the Docker Hub website
# Copyright (C) 2020 Katrine Wist and Malene Helsem
#
# This program is free software: you can redistribute it and/or
# modify it under the terms of the GNU General Public License as
# published by the Free Software Foundation, either version 3 of the
# License, or (at your option) any later version.
#
# This program is distributed in the hope that it will be useful, but
# WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
# General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program. If not, see http://www.gnu.org/licenses/.
#
################################################################################

from selenium import webdriver
import time

urlpage = 'https://hub.docker.com/search/?q=&type=image&page={}'
page_count = 0
print("Scraping started ...")
image_names = open("./image-names.txt","w")
image_info = open("./image-info.csv", "w")

while page_count < 100:
    page_count +=1
    url = urlpage.format(page_count)
    #Run the firefox webdriver, this is where you need to specify the path to
    ↪ Geckodriver
    driver = webdriver.Firefox(executable_path = '/path-to-geckodriver-here')
    #Get web page
    driver.get(url)
    #Sleep for 10s, to let the page fully load
    time.sleep(10)
    #Find elements by xpath
    results = driver.find_elements_by_xpath("//*[@class='imageSearchResult
    ↪  styles__searchResult___EBKah styles__clickable___2bfia']")

    print('Scraping page: ', page_count)
    for info in results:
        image_type = ""
        #Set the image type
        if('OFFICIAL' in info.text):
            image_type = "official"
        elif('Certified' in info.text):
            image_type = "certified"
        elif('VERIFIED' in info.text):
            image_type = "verified"
        else:
            image_type = "community"

        #Gather the image name from URL, different procedure for each image type
        image_link = info.get_attribute('href')
        if(image_type == "official"):
            tmp = image_link.split("/")[-1] + "," + image_type + "\n"
            image_names.write(image_link.split("/")[-1]+"\n")
            image_info.write(tmp)
        elif(image_type == "community"):
            tmp = image_link.split("/")[-2] + "/" + image_link.split("/")[-1] + "," +
            ↪  image_type + "\n"
            image_names.write(image_link.split("/")[-2] + "/" +
            ↪  image_link.split("/")[-1] + "\n")
            image_info.write(tmp)
        else:
            tmp = image_link.split("/")[-1] + "," + image_type + "\n"
            image_names.write(image_link.split("/")[-1]+"\n")
            image_info.write(tmp)

    time.sleep(5)
    driver.close()

#Close driver
driver.quit()
#Close files
image_names.close()
image_info.close()
```

Appendix C

Script for accessing the Docker Registry HTTP API V1

```
####################################################################################
#
# Gather information (pull count and last updated time) about verified and certified
↪  Docker images
# Copyright (C) 2020 Katrine Wist and Malene Helsem
#
# This program is free software: you can redistribute it and/or
# modify it under the terms of the GNU General Public License as
# published by the Free Software Foundation, either version 3 of the
# License, or (at your option) any later version.
#
# This program is distributed in the hope that it will be useful, but
# WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
# General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program. If not, see http://www.gnu.org/licenses/.
#

####################################################################################

import urllib.request, json
from urllib.error import HTTPError

names_file = open("./image_names.txt")
image_names = [line.rstrip('\n') for line in names_file]
names_file.close()
print("Finished reading image names")

results = open("./results_apiv1.csv","w")
failed = open("./failed_apiv1.txt","w")
results.write("image_name,pull_count,star_count,last_updated" + "\n")

print("Starting data gathering ...")
for i in image_names:
    print("Gathering data about: " + i)
    url_page = 'https://hub.docker.com/api/content/v1/products/images/{}'
    try:
        with urllib.request.urlopen(url_page.format(i)) as url:
            data = json.loads(url.read().decode())
            popularity =  str(data['popularity'])
            if(str(data['popularity']) == str(0)):
                popularity = ""
            results.write(i + "," + popularity + "," + "," + str(data['updated_at'])
            ↪  + "\n")
    except HTTPError as e:
        failed.write(i + "\n")
        print(e.reason)
        continue

results.close()
failed.close()
```

Appendix D

Script for accessing the Docker Registry HTTP API V2

```
################################################################################
#
# Gather information (pull count, star count, last updated time) about official and
↪  community Docker images
# Copyright (C) 2020 Katrine Wist and Malene Helsem
#
# This program is free software: you can redistribute it and/or
# modify it under the terms of the GNU General Public License as
# published by the Free Software Foundation, either version 3 of the
# License, or (at your option) any later version.
#
# This program is distributed in the hope that it will be useful, but
# WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
# General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program. If not, see http://www.gnu.org/licenses/.
#
################################################################################

import urllib.request, json
from urllib.error import HTTPError

names_file = open("./image_names.txt")
image_names = [line.rstrip('\n') for line in names_file]
names_file.close()
print("Finished reading image names ...")

results = open("./results_apiv2.csv","w")
failed = open("./failed_apiv2.txt","w")
results.write("image_name,pull_count,star_count,last_updated" + "\n")

print("Starting data gathering ...")
for i in image_names:
    print("Gathering data about: " + i)
    if ("/" in i):
        url_page = 'https://hub.docker.com/v2/repositories/{}'
    else:
        url_page = 'https://hub.docker.com/v2/repositories/library/{}'
    try:
        with urllib.request.urlopen(url_page.format(i)) as url:
            data = json.loads(url.read().decode())
            results.write(i + "," + str(data['pull_count']) + "," +
            ↪  str(data['star_count'])+ "," + str(data['last_updated']) + "\n")
    except HTTPError as e:
        failed.write(i + "\n")
        print(e.reason)
        continue

results.close()
failed.close()
```

Appendix E
Script for automate analysis

```bash
################################################################################
#
# Automate the process of analyzing Docker images with the latest tag using the
↪  Anchore Engine vulnerability scanner
# Copyright (C) 2020 Katrine Wist and Malene Helsem
#
# This program is free software: you can redistribute it and/or
# modify it under the terms of the GNU General Public License as
# published by the Free Software Foundation, either version 3 of the
# License, or (at your option) any later version.
#
# This program is distributed in the hope that it will be useful, but
# WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
# General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program. If not, see http://www.gnu.org/licenses/.
#
################################################################################

#!/bin/bash

mv vuln.csv vuln_"$(date +%Y-%m-%d_%H-%M-%S)".csv
touch "vuln.csv"
mv failed.txt failed_"$(date +%Y-%m-%d_%H-%M-%S)".txt
touch "failed.txt"

for image in $(cat ./image-names.txt)
do
    echo "Adding $image..."
    success=true
    SECONDS=0
    while true
    do
        output=$(anchore-cli --url "http://localhost:8228/v1" --u "admin" --p
        ↪  "foobar" image add "$image:latest")
    echo "$output" | grep "error_code=REGISTRY_PERMISSION_DENIED'}" -q && echo
    ↪  "Permission denied to scan image." && echo "$image,permission_denied" >>
    ↪  failed.txt && success=false && break
    echo "$output" | grep "error_code=REGISTRY_IMAGE_NOT_FOUND'}" -q && echo
    ↪  "Image not found."&& echo "$image,not_found" >> failed.txt &&
    ↪  success=false && break
    echo "$output" | grep "HTTP Code: 400" -q && echo "Unknown error occured." &&
    ↪  echo "$image,unknown_error" >> failed.txt && success=false && break
    if [ "$SECONDS" -gt "3600" ]
    then
      echo "Timeout"&& echo "$image,timeout" >> failed.txt && success=false &&
      ↪  break
   fi
   echo "$output" | grep "Analysis Status: analyzed" -q && break || (echo "Not
   ↪  finished, sleeping..." && sleep 20)
  done
  if $success
  then
    echo "Finished, writing result to file..."
    if [[ $(anchore-cli --url "http://localhost:8228/v1" --u "admin" --p "foobar"
    ↪  image vuln "$image:latest" all) ]]
    then
      anchore-cli --url "http://localhost:8228/v1" --u "admin" --p "foobar"
      ↪  image vuln "$image:latest" all | sed -r 's/,/;/g' |sed -r 's/
      ↪  +/,/g'| sed 's/,$//' | egrep -v '^Vulnerability ID' | awk '{ print
      ↪  "'"$image"',"$1 }' | awk -F, -v OFS=, '{ if(NF==6) { k=$NF; $6="";
      ↪  $7=k; print } else { for ( i=NF; i<=7; i++ ) { $i=$i"" } print }}' >>
      ↪  vuln.csv
   else
      echo "No found vulnerabilities"
   fi

  fi
done
```

Appendix F
SQL imports

```sql
USE master_db;
-----------------------------------------------------------------------------------------
#process for creating image_info table

CREATE TABLE image_info_scraper (
    image_id INT AUTO_INCREMENT PRIMARY KEY,
    image VARCHAR(255) NOT NULL,
    i_type VARCHAR(255));
#content is image_info_scraper.csv

CREATE TABLE image_info_api (
    image VARCHAR(255) NOT NULL,
    pulls VARCHAR(255),
    stars VARCHAR(255),
    last_updated VARCHAR(255));
#content is image_info_api.csv

CREATE TABLE image_info (
    image_id INT NOT NULL PRIMARY KEY,
    image VARCHAR(255) NOT NULL,
    i_type VARCHAR(255),
    pulls VARCHAR(255),
    stars VARCHAR(255),
    last_updated VARCHAR(255));
#content is result of join on tables image_info_scraper and image_info_api

#join two tables and insert into table
INSERT INTO image_info
SELECT
    A.image_id,
    A.image,
    A.i_type,
    COALESCE(B.pulls, ''),
    COALESCE(B.stars, ''),
    COALESCE(B.last_updated, '')
FROM image_info_scraper A
LEFT JOIN image_info_api B USING (image);
```

```
#COALESCE is used to change empty values from NULL to ''

-------------------------------------------------------------------------------------
#process for creating vuln table

CREATE TABLE vuln (
    vuln_id INT AUTO_INCREMENT PRIMARY KEY,
    image_id INT,
    image VARCHAR(255) NOT NULL,
    vuln_name VARCHAR(255),
    package VARCHAR(255),
    severity VARCHAR(255),
    fix VARCHAR(255),
    cve_refs VARCHAR(800),
    vuln_url VARCHAR(255),
    FOREIGN KEY (image_id) REFERENCES image_info(image_id));

#update the image_id column with correct values
UPDATE vuln A LEFT JOIN image_info B USING(image) SET A.image_id=B.image_id;

-------------------------------------------------------------------------------------
#create failed table

CREATE TABLE failed (
    failed_id INT AUTO_INCREMENT PRIMARY KEY,
    image_id INT,
    image VARCHAR(255) NOT NULL,
    failure_reason VARCHAR(255),
    FOREIGN KEY (image_id) REFERENCES image_info(image_id));
#content is failed.csv

#update the image_id column with correct values
UPDATE failed A LEFT JOIN image_info B USING(image) SET A.image_id=B.image_id;
```

Appendix G
SQL for data analysis

```sql
USE master_db;
------------------------------------------------------------------------------------------
#image information and failed images

#number of images, pulls, stars and last updated of each image type
SELECT
    i_type,
    COUNT(DISTINCT image_id) AS number_of_images,
    COUNT(DISTINCT IF(NOT image_id='',image_id,Null)) AS image_id,
    COUNT(DISTINCT IF(NOT A.image='',image_id,Null)) AS image,
    COUNT(DISTINCT IF(NOT i_type='',image_id,Null)) AS i_type,
    COUNT(DISTINCT IF(NOT pulls='',image_id,Null)) AS pulls,
    COUNT(DISTINCT IF(NOT stars='',image_id,Null)) AS stars,
    COUNT(DISTINCT IF(NOT last_updated='',image_id,Null)) AS last_updated
FROM (SELECT * FROM image_info WHERE image_id NOT IN (SELECT image_id FROM failed)) AS
↪  A
LEFT JOIN vuln B USING (image_id)
GROUP BY i_type;
# '' indicates empty field

#number of analyzed vs failed images of the each image type
SELECT
    i_type,
    COUNT(DISTINCT IF(NOT A.image_id IN (SELECT image_id FROM failed), A.image_id,
    ↪  Null)) AS analyzed_images,
    COUNT(DISTINCT B.image_id) AS failed_images
FROM image_info A LEFT JOIN failed B USING (image_id)
GROUP BY i_type;

#number of images of each failure reason
SELECT
    failure_reason,
    COUNT(*)
FROM failed
GROUP BY failure_reason;

------------------------------------------------------------------------------------------
```

```sql
#vulnerability information

# number of values of each column in vuln
SELECT
    (SELECT COUNT(IF(NOT vuln_id='', vuln_id, Null))) AS vuln_id,
    (SELECT COUNT(IF(NOT image_id='', image_id, Null))) AS image_id,
    (SELECT COUNT(IF(NOT image='', image, Null))) AS image,
    (SELECT COUNT(IF(NOT vuln_name='', vuln_name, Null))) AS vuln_name,
    (SELECT COUNT(IF(NOT package='', package, Null))) AS package,
    (SELECT COUNT(IF(NOT severity='', severity, Null))) AS severity,
    (SELECT COUNT(IF(NOT fix='', fix, Null))) AS fix,
    (SELECT COUNT(IF(NOT cve_refs='', cve_refs, Null))) AS cve_refs,
    (SELECT COUNT(IF(NOT vuln_url='', vuln_url, Null))) AS vuln_url
FROM vuln;

#distinct number of values of each column in vuln
SELECT
    COUNT(DISTINCT vuln_id) AS vuln_id,
    COUNT(DISTINCT image_id) AS image_id,
    COUNT(DISTINCT image) AS image,
    COUNT(DISTINCT vuln_name) AS vuln_name,
    COUNT(DISTINCT package) AS package,
    COUNT(DISTINCT severity) AS severity,
    COUNT(DISTINCT fix) AS fix,
    COUNT(DISTINCT cve_refs) AS cve_refs,
    COUNT(DISTINCT vuln_url) AS vuln_url
FROM vuln;

#number of empty values of each column in vuln (null values are empty string '')
SELECT
    (SELECT COUNT(IF(vuln_id='', vuln_id, Null))) AS vuln_id,
    (SELECT COUNT(IF(image_id='', image_id, Null))) AS image_id,
    (SELECT COUNT(IF(image='', image, Null))) AS image,
    (SELECT COUNT(IF(vuln_name='', vuln_name, Null))) AS vuln_name,
    (SELECT COUNT(IF(package='', package, Null))) AS package,
    (SELECT COUNT(IF(severity='', severity, Null))) AS severity,
    (SELECT COUNT(IF(fix='', fix, Null))) AS fix,
    (SELECT COUNT(IF(cve_refs='', cve_refs, Null))) AS cve_refs,
    (SELECT COUNT(IF(vuln_url='', vuln_url, Null))) AS vuln_url
FROM vuln;

----------------------------------------------------------------------------------
# distribution of vulnerabilities in each severity category

SELECT
    severity,
    COUNT(*) AS num_of_vulns,
    COUNT(DISTINCT vuln_name) AS num_of_unique_vulns,
    COUNT(*) / COUNT(DISTINCT vuln_name) AS ratio
FROM vuln
GROUP BY severity;

```

```
--------------------------------------------------------------------------------------
#central tendency

#number of vulnerabilities for each image, used for plotting histogram
SELECT
    A.image_id,
    COUNT(vuln_name)  AS vuln_count
FROM (SELECT * FROM image_info WHERE image_id NOT IN (SELECT image_id FROM failed)) AS
↪  A
LEFT JOIN vuln B USING (image_id)
GROUP BY A.image_id;

#average number of vulns per image
SELECT AVG(vuln_count)
FROM
    (SELECT
        image_id,
        COUNT(vuln_name) AS vuln_count
    FROM (SELECT * FROM image_info WHERE image_id NOT IN (SELECT image_id FROM
    ↪  failed)) AS A
    LEFT JOIN vuln B USING (image_id)
    GROUP BY image_id) AS C;

#median number of vulns per image
SET @rowindex := -1;
SELECT AVG(D.vuln_count) AS median
FROM
    (SELECT @rowindex:=@rowindex + 1 AS rowindex, C.vuln_count
    FROM
        (SELECT
            A.image_id,
            COUNT(vuln_name) AS vuln_count
        FROM (SELECT * FROM image_info WHERE image_id NOT IN (SELECT image_id FROM
        ↪  failed)) AS A
        LEFT JOIN vuln B USING (image_id)
        GROUP BY A.image_id
        ORDER BY vuln_count DESC) AS C) AS D
WHERE D.rowindex IN (FLOOR(@rowindex / 2) , CEIL(@rowindex / 2));


#average and median when disregarding unknown and negligible vulns

#average number of vulns per image, disregarding unknown and negligible vulns
SELECT AVG(C.vuln_count)
FROM
    (SELECT
        A.image_id,
        COUNT(IF(NOT severity='negligible' AND NOT severity='unknown', 1, Null))  AS
        ↪  vuln_count
    FROM (SELECT * FROM image_info WHERE image_id NOT IN (SELECT image_id FROM
    ↪  failed)) AS A
    LEFT JOIN vuln B USING (image_id)
    GROUP BY A.image_id) AS C;

#median number of vulns per image, disregarding unknown and negligible vulns
SET @rowindex := -1;
SELECT AVG(D.vuln_count) AS median
FROM
    (SELECT @rowindex:=@rowindex + 1 AS rowindex, C.vuln_count
    FROM
        (SELECT
            A.image_id,
            COUNT(IF(NOT severity='negligible' AND NOT severity='unknown', 1, Null))
            ↪  AS vuln_count
        FROM (SELECT * FROM image_info WHERE image_id NOT IN (SELECT image_id FROM
        ↪  failed)) AS A
        LEFT JOIN vuln B USING (image_id)
        GROUP BY A.image_id
        ORDER BY vuln_count DESC) AS C) AS D
WHERE D.rowindex IN (FLOOR(@rowindex / 2) , CEIL(@rowindex / 2));
#insert WHERE i_type = verified, certified, official or community before GROUP BY
↪  clause to find median for each image type


#queries for table about statistical values:

#number of images in each type and number of vulnerabilities, not counting negligible
↪  and unknown
SELECT
    i_type,
    COUNT(DISTINCT image_id) AS num_of_images,
    COUNT(IF(NOT severity='negligible' AND NOT severity='unknown', vuln_id, Null)) AS
    ↪  num_of_vulns
FROM (SELECT * FROM image_info WHERE image_id NOT IN (SELECT image_id FROM failed))AS
↪  A
LEFT JOIN vuln B USING (image_id)
GROUP BY i_type;
#average, max, min and standard deviation of number of vulnerabilities in each image
SELECT
    i_type,
    AVG(vuln_count) AS average,
    MAX(vuln_count) AS max_,
    MIN(vuln_count) AS min_,
    STDDEV_SAMP(vuln_count) AS stddev_
FROM
    (SELECT
        A.image_id,
        MIN(i_type) AS i_type,
        COUNT(IF(NOT vuln_name='' AND (NOT severity='negligible' AND NOT
        ↪  severity='unknown'), 1, Null))  AS vuln_count
    FROM (SELECT * FROM image_info WHERE image_id NOT IN (SELECT image_id FROM
    ↪  failed)) AS A
    LEFT JOIN vuln B USING (image_id)
    GROUP BY A.image_id) AS C
```

```sql
GROUP BY i_type;

-------------------------------------------------------------------------------------
# median number of vulnerabilities for each image type

SET @rowindex := -1;
SELECT AVG(D.vuln_count) AS median
FROM
    (SELECT @rowindex:=@rowindex + 1 AS rowindex, C.vuln_count
    FROM
    (SELECT
        A.image_id,
        COUNT(vuln_name)  AS vuln_count
    FROM (SELECT * FROM image_info WHERE image_id NOT IN (SELECT image_id FROM
    ↪  failed)) AS A
    LEFT JOIN vuln B USING (image_id)
    WHERE severity = 'critical' AND i_type = 'verified'
    GROUP BY A.image_id
    ORDER BY vuln_count DESC) AS C) AS D
WHERE D.rowindex IN (FLOOR(@rowindex / 2) , CEIL(@rowindex / 2));
#to get the median for all severities and all image types:
  #change between severity = critical, high, medium and low     AND
  #change between i_type = verified, certified, official and community

-------------------------------------------------------------------------------------
#density distribution plots

#creating output for tables with number of vulns per image in each image type - used
↪  in density distribution
SELECT
    image_id,
    COUNT(IF(NOT severity='negligible' AND NOT severity='unknown', 1, Null))  AS
    ↪  vuln_count
FROM (SELECT * FROM image_info WHERE image_id NOT IN (SELECT image_id FROM failed)) AS
↪  A
LEFT JOIN vuln B USING (image_id)
WHERE i_type='verified'
GROUP BY image_id;
#to get tables with the data for each image type:
  #change between i_type = verified, certified, official and community

-------------------------------------------------------------------------------------
#images that contain the most critical and high vulns

SELECT
    A.image,
    SUM(severity='Critical') AS critical_count,
    SUM(severity='High') AS high_count,
    SUM(severity='Medium') AS medium_count,
    SUM(severity='Low') AS low_count,
    AVG(pulls) AS no_of_pulls
FROM vuln A LEFT JOIN image_info B USING(image_id)
```

```
GROUP BY A.image
ORDER BY critical_count DESC;

--------------------------------------------------------------------------------
#percentage of images with critical and high vulnerabilities

SELECT
    i_type,
    (COUNT(DISTINCT IF(severity='critical',image_id, Null)) / COUNT(DISTINCT
    ↪  image_id))*100 AS percentage_critical,
    (COUNT(DISTINCT IF(severity='high',image_id, Null)) / COUNT(DISTINCT
    ↪  image_id))*100 AS percentage_high,
    (COUNT(DISTINCT IF(severity='critical' OR severity='high',image_id, Null)) /
    ↪  COUNT(DISTINCT image_id))*100 AS percentage_both
FROM (SELECT * FROM image_info WHERE image_id NOT IN (SELECT image_id FROM failed)) A
LEFT JOIN vuln B USING (image_id)
GROUP BY i_type;

--------------------------------------------------------------------------------
#vulnerabilities in microsoft images

#number of analyzed and failed microsoft images
SELECT
    (SELECT COUNT(DISTINCT IF(INSTR(A.image, 'microsoft'), image_id, Null))
    FROM (SELECT * FROM image_info WHERE image_id NOT IN (SELECT image_id FROM
    ↪  failed)) A
    LEFT JOIN vuln B USING (image_id)) AS analyzed_images,
    (SELECT COUNT(DISTINCT IF(INSTR(image, 'microsoft'), image_id, Null))
    FROM failed) AS failed_images;

#number of vulns and unique  vulns in microsoft images for each severity
SELECT
    severity,
    COUNT(*) AS vulns,
    COUNT(DISTINCT vuln_name) AS distinct_vulns,
    COUNT(*) / COUNT(DISTINCT vuln_name) AS ratio
FROM vuln
WHERE INSTR(image, 'microsoft')
GROUP BY severity;


#comparing microsoft images and all other images:

#number of vulns in microsoft and in all other images, grouped BY severity
SELECT
    severity,
    COUNT(IF(INSTR(image, 'microsoft'), vuln_id, Null))AS vulns_microsoft,
    COUNT(IF(NOT INSTR(image, 'microsoft'), vuln_id, Null))AS vulns_no_microsoft
FROM vuln
GROUP BY severity;
#number of images in microsoft and in all other images
SELECT
```

```
    COUNT(DISTINCT IF(INSTR(A.image, 'microsoft'), image_id, Null)) AS
    ↪  num_of_microsoft_images,
    COUNT(DISTINCT IF(NOT INSTR(A.image, 'microsoft'), image_id, Null)) AS
    ↪  num_of_non_microsoft_images
FROM (SELECT * FROM image_info WHERE image_id NOT IN (SELECT image_id FROM failed)) A
LEFT JOIN vuln B USING (image_id);

-------------------------------------------------------------------------------------
#images with no vulnerabilities

#number of image with no vulns for each image type
SELECT i_type, COUNT(*)
FROM image_info
WHERE image_id NOT IN (SELECT image_id FROM vuln) AND image_id NOT IN (SELECT image_id
↪  FROM failed)
GROUP BY i_type;

#stats about images without vulnerabilities
SELECT
    #number of images
    (SELECT COUNT(*)
    FROM image_info
    WHERE image_id NOT IN (SELECT image_id FROM vuln) AND image_id NOT IN (SELECT
    ↪  image_id FROM failed)) AS num_of_images,
    #average pulls
    (SELECT AVG(pulls)
    FROM image_info
    WHERE image_id NOT IN (SELECT image_id FROM vuln) AND image_id NOT IN (SELECT
    ↪  image_id FROM failed) AND NOT pulls='') AS avg_of_pulls,
    #average stars
    (SELECT AVG(stars)
    FROM image_info
    WHERE image_id NOT IN (SELECT image_id FROM vuln) AND image_id NOT IN (SELECT
    ↪  image_id FROM failed) AND NOT stars='') AS avg_of_stars,
    #average days since last update
    (SELECT AVG(days_since)
    FROM(
        SELECT AVG(IF(NOT last_updated='',DATEDIFF('2020-02-25', last_updated),Null))
        ↪  AS days_since
        FROM image_info
        WHERE image_id NOT IN (SELECT image_id FROM vuln) AND image_id NOT IN (SELECT
        ↪  image_id FROM failed) AND NOT last_updated=''
        GROUP BY image
        ORDER BY days_since DESC) AS A) AS avg_of_days_since;

#stats about images with vulnerabilities
SELECT
    #number of images
    (SELECT COUNT(*)
    FROM
        (SELECT DISTINCT image_id, pulls
        FROM vuln LEFT JOIN image_info USING(image_id)) AS A) AS num_of_images,
    #average pulls
    (SELECT AVG(pulls)
    FROM
        (SELECT DISTINCT image_id, pulls
        FROM vuln LEFT JOIN image_info USING(image_id) WHERE NOT pulls='') AS A) AS
        ↪  avg_of_pulls,
    #average stars
    (SELECT AVG(stars)
    FROM
      (SELECT DISTINCT image_id, stars
      FROM vuln LEFT JOIN image_info USING(image_id) WHERE NOT stars='') AS A) AS
      ↪  avg_of_stars,
    #average days since last update
    (SELECT AVG(days_since)
    FROM
      (SELECT
            DISTINCT image_id,
            AVG(IF(NOT last_updated='',DATEDIFF('2020-02-25', last_updated),Null)) AS
            ↪  days_since
      FROM vuln LEFT JOIN image_info USING(image_id) WHERE NOT last_updated=''
      GROUP BY image_id
      ORDER BY days_since DESC) AS A) AS avg_of_days_since;

#number of images that only contain vulnerabilities that are negiligible and unknown
SELECT COUNT(DISTINCT image_id)
FROM
(SELECT DISTINCT image_id, image, severity
FROM vuln
WHERE image_id NOT IN
  (SELECT image_id
  FROM vuln
  WHERE severity IN ('critical', 'high', 'medium', 'low'))) AS A;

-----------------------------------------------------------------------------------
#cve trend

#number of unique vulns grouped by year
SELECT
    DISTINCT IF(INSTR(vuln_name,'cve'),SUBSTR(vuln_name, 5, 4),
    ↪  SUBSTR(vuln_name,6,4)) AS year,
    COUNT(*) AS total_count,
    COUNT(IF(i_type='verified', vuln_name, Null)) AS verified_count,
    COUNT(IF(i_type='certified', vuln_name, Null)) AS certified_count,
    COUNT(IF(i_type='official', vuln_name, Null)) AS official_count,
    COUNT(IF(i_type='community', vuln_name, Null)) AS community_count
FROM (SELECT DISTINCT vuln_name, i_type FROM vuln A LEFT JOIN image_info B
↪  USING(image_id)) AS A
GROUP BY year
ORDER BY year;

-----------------------------------------------------------------------------------
#days since last update
```

```sql

#number of images that have not been updated for 400, 200 or less than 14 days, and
↪  percentage
SELECT
    i_type,
    COUNT(IF(A.days_since>400, image, Null)) AS more_than_400,
    COUNT(IF(A.days_since>200, image, Null)) AS more_than_200,
    COUNT(IF(A.days_since<14, image, Null)) AS less_than_14,
    COUNT(IF(NOT days_since IS Null, image, Null)) AS total,
    (COUNT(IF(A.days_since>400, image, Null))/COUNT(IF(NOT days_since IS Null, image,
    ↪  Null)))*100 AS more_than_400_percent,
    (COUNT(IF(A.days_since>200, image, Null))/COUNT(IF(NOT days_since IS Null, image,
    ↪  Null)))*100 AS more_than_200_percent,
    (COUNT(IF(A.days_since<14, image, Null))/COUNT(IF(NOT days_since IS Null, image,
    ↪  Null)))*100 AS less_than_14_percent
FROM(
    SELECT
        A.image,
        MIN(i_type) AS i_type,
        AVG(IF(NOT last_updated='',DATEDIFF('2020-02-25', last_updated),Null)) AS
        ↪  days_since
    FROM (SELECT * FROM image_info WHERE image_id NOT IN (SELECT image_id FROM
    ↪  failed)) AS A
    LEFT JOIN vuln B USING (image_id)
    GROUP BY A.image
    ORDER BY days_since DESC) AS A
    GROUP BY i_type;

#image type and last update for each image
SELECT
    DISTINCT image_id,
    i_type, SUBSTR(last_updated,1,10) AS last_update
FROM (SELECT * FROM image_info WHERE image_id NOT IN (SELECT image_id FROM failed)) AS
↪  A
LEFT JOIN vuln B USING (image_id)
ORDER BY last_update;
#the substring function extracts only the date on the format "yyyy-mm-dd"

-------------------------------------------------------------------------------------
#correlation

##Spearman for vulns and pulls
SET @n_x := 0;
SET @n_y := 0;
WITH
cte1 AS
(SELECT
    @n_x := @n_x +1 AS n_x,
    image_id,
    RANK() OVER(ORDER BY vuln_count) AS vuln_count_rank,
    COUNT(vuln_count) OVER (PARTITION BY vuln_count) AS vuln_count_count
FROM
```

```
    (SELECT
        AVG(A.image_id) AS image_id,
        COUNT(vuln_name)  AS vuln_count,
        AVG(IF(NOT pulls='',pulls,Null)) AS pulls
    FROM (SELECT * FROM image_info WHERE image_id NOT IN (SELECT image_id FROM
    ↪  failed)) AS A
    LEFT JOIN vuln B USING (image_id)
    GROUP BY A.image_id) AS C WHERE NOT C.pulls IS Null),
cte2 AS
(SELECT
    @n_y := @n_y +1 AS n_y,
    image_id,
    RANK() OVER(ORDER BY pulls) AS pulls_rank,
    COUNT(pulls) OVER (PARTITION BY pulls) AS pulls_count
FROM
    (SELECT
        AVG(A.image_id) AS image_id,
        COUNT(vuln_name)  AS vuln_count,
        AVG(IF(NOT pulls='',pulls,Null)) AS pulls
    FROM (SELECT * FROM image_info WHERE image_id NOT IN (SELECT image_id FROM
    ↪  failed)) AS A
    LEFT JOIN vuln B USING (image_id)
    GROUP BY A.image_id) AS C WHERE NOT C.pulls IS Null),
cte3 AS
(SELECT
    SUM(n_x) OVER (PARTITION BY vuln_count_rank) / AVG(vuln_count_count) OVER
    ↪  (PARTITION BY vuln_count_rank) AS ranked_vuln_count,
    SUM(n_y) OVER (PARTITION BY pulls_rank) / AVG(pulls_count) OVER (PARTITION BY
    ↪  pulls_rank) AS ranked_pulls
FROM cte1 LEFT JOIN cte2 USING(image_id)),
cte4 AS
(SELECT
    @ax := AVG(ranked_vuln_count) AS avg_x,
    @ay := AVG(ranked_pulls) AS avg_y,
    @div := (STDDEV_SAMP(ranked_vuln_count) * STDDEV_SAMP(ranked_pulls)) AS stddev
FROM cte3)
SELECT
    (SELECT Null FROM cte4) AS null_,
    (SELECT SUM( ( ranked_vuln_count - @ax ) * (ranked_pulls - @ay) ) / ((COUNT(*)
    ↪  -1) * @div)
    FROM cte3) AS correlation;

##Spearman for vulns and stars
SET @n_x := 0;
SET @n_y := 0;
WITH
cte1 AS
(SELECT
    @n_x := @n_x +1 AS n_x,
    image_id,
    RANK() OVER(ORDER BY vuln_count) AS vuln_count_rank,
    COUNT(vuln_count) OVER (PARTITION BY vuln_count) AS vuln_count_count
```

```
FROM
(SELECT
    AVG(A.image_id) AS image_id,
    COUNT(vuln_name)  AS vuln_count,
    AVG(IF(NOT stars='',stars,Null)) AS stars
FROM (SELECT * FROM image_info WHERE image_id NOT IN (SELECT image_id FROM failed)) AS
↪  A
LEFT JOIN vuln B USING (image_id)
GROUP BY A.image_id) AS C WHERE NOT C.stars IS Null),
cte2 AS
(SELECT
    @n_y := @n_y +1 AS n_y,
    image_id,
    RANK() OVER(ORDER BY stars) AS stars_rank,
    COUNT(stars) OVER (PARTITION BY stars) AS stars_count
FROM
(SELECT
    AVG(A.image_id) AS image_id,
    COUNT(vuln_name)  AS vuln_count,
    AVG(IF(NOT stars='',stars,Null)) AS stars
FROM (SELECT * FROM image_info WHERE image_id NOT IN (SELECT image_id FROM failed)) AS
↪  A
LEFT JOIN vuln B USING (image_id)
GROUP BY A.image_id) AS C WHERE NOT C.stars IS Null),
cte3 AS
(SELECT
    SUM(n_x) OVER (PARTITION BY vuln_count_rank) / AVG(vuln_count_count) OVER
    ↪  (PARTITION BY vuln_count_rank) AS ranked_vuln_count,
    SUM(n_y) OVER (PARTITION BY stars_rank) / AVG(stars_count) OVER (PARTITION BY
    ↪  stars_rank) AS ranked_stars
FROM cte1 LEFT JOIN cte2 USING(image_id)),
cte4 AS
(SELECT
    @ax := AVG(ranked_vuln_count) AS avg_x,
    @ay := AVG(ranked_stars) AS avg_y,
    @div := (STDDEV_SAMP(ranked_vuln_count) * STDDEV_SAMP(ranked_stars)) AS stddev
FROM cte3)
SELECT
    (SELECT Null FROM cte4) AS null_,
    (SELECT SUM( ( ranked_vuln_count - @ax ) * (ranked_stars - @ay) ) / ((COUNT(*)
    ↪  -1) * @div)
    FROM cte3) AS correlation;

##Spearman for vulns and days since last update
SET @n_x := 0;
SET @n_y := 0;
WITH
cte1 AS
(SELECT
    @n_x := @n_x +1 AS n_x,
    image_id,
    RANK() OVER(ORDER BY vuln_count) AS vuln_count_rank,
```

```sql
    COUNT(vuln_count) OVER (PARTITION BY vuln_count) AS vuln_count_count
FROM
(SELECT
    AVG(A.image_id) AS image_id,
    COUNT(vuln_name)  AS vuln_count,
    AVG(IF(NOT last_updated='',DATEDIFF('2020-02-25', last_updated),Null)) AS
    ↪  days_since
FROM (SELECT * FROM image_info WHERE image_id NOT IN (SELECT image_id FROM failed)) AS
↪  A
LEFT JOIN vuln B USING (image_id)
GROUP BY A.image_id) AS C WHERE NOT C.days_since IS Null),
cte2 AS
(SELECT
    @n_y := @n_y +1 AS n_y,
    image_id,
    RANK() OVER(ORDER BY days_since) AS days_since_rank,
    COUNT(days_since) OVER (PARTITION BY days_since) AS days_since_count
FROM
(SELECT
    AVG(A.image_id) AS image_id,
    COUNT(vuln_name)  AS vuln_count,
    AVG(IF(NOT last_updated='',DATEDIFF('2020-02-25', last_updated),Null)) AS
    ↪  days_since
FROM (SELECT * FROM image_info WHERE image_id NOT IN (SELECT image_id FROM failed)) AS
↪  A
LEFT JOIN vuln B USING (image_id)
GROUP BY A.image_id) AS C WHERE NOT C.days_since IS Null),
cte3 AS
(SELECT
    SUM(n_x) OVER (PARTITION BY vuln_count_rank) / AVG(vuln_count_count) OVER
    ↪  (PARTITION BY vuln_count_rank) AS ranked_vuln_count,
    SUM(n_y) OVER (PARTITION BY days_since_rank) / AVG(days_since_count) OVER
    ↪  (PARTITION BY days_since_rank) AS ranked_days_since
FROM cte1 LEFT JOIN cte2 USING(image_id)),
cte4 AS
(SELECT
    @ax := AVG(ranked_vuln_count) AS avg_x,
    @ay := AVG(ranked_days_since) AS avg_y,
    @div := (STDDEV_SAMP(ranked_vuln_count) * STDDEV_SAMP(ranked_days_since)) AS
    ↪  stddev
FROM cte3)
SELECT
    (SELECT Null FROM cte4) AS null_,
    (SELECT SUM( ( ranked_vuln_count - @ax ) * (ranked_days_since - @ay) ) /
    ↪  ((COUNT(*) -1) * @div)
    FROM cte3) AS correlation;

-------------------------------------------------------------------------------------
#the most represented critical vulns
SELECT
    vuln_name,
    SUM(severity='Critical') AS critical_count
```

```
FROM vuln
GROUP BY vuln_name
ORDER BY critical_count DESC;

-----------------------------------------------------------------------------------------
#vulnerabilities in packages

#the most vulnerable packages
SELECT
    package,
    SUM(severity='Critical') AS critical_count,
    (SELECT COUNT(DISTINCT image_id)) AS number_of_images
FROM vuln
GROUP BY package
ORDER BY critical_count DESC;

#vulnerabilities in popular packages
SELECT
    package,
    SUM(severity='Critical') AS critical_count,
    SUM(severity='High') AS high_count,
    SUM(severity='Medium') AS medium_count,
    SUM(severity='Low') AS low_count,
    SUM(severity='Negligible') AS negligible_count,
    SUM(severity='Unknown') AS unknown_count,
    (SELECT COUNT(DISTINCT image)) AS number_of_images
FROM vuln
GROUP BY package
ORDER BY number_of_images DESC;

-----------------------------------------------------------------------------------------
#miscellaneous queries

#used to create a file with image, number of vulns, pulls, stars and days since last
↪ update
SELECT
    A.image,
    COUNT(vuln_name)  AS vuln_count,
    AVG(IF(NOT pulls='',pulls,Null)) AS pulls,
    AVG(IF(NOT stars='',stars,Null)) AS stars,
    AVG(IF(NOT last_updated='',DATEDIFF('2020-02-25', last_updated),Null)) AS
    ↪ days_since
FROM (SELECT * FROM image_info WHERE image_id NOT IN (SELECT image_id FROM failed)) AS
↪ A
LEFT JOIN vuln B USING (image_id)
GROUP BY A.image;
```

Appendix H
Conference paper draft

Vulnerability Analysis of 2500 Docker Hub Images

Katrine Wist
Dep. of Inf. Sec. and Comm. Techn.
Norwegian University of Science
and Technology (NTNU), Norway
katrinew0702@gmail.com

Malene Helsem
Dep. of Inf. Sec. and Comm. Techn.
Norwegian University of Science
and Technology (NTNU), Norway
malenehlsm@gmail.com

Danilo Gligoroski
Dep. of Inf. Sec. and Comm. Techn.
Norwegian University of Science
and Technology (NTNU), Norway
danilog@ntnu.no

***Abstract*—The use of container technology has skyrocketed during the last few years, with Docker as the leading container platform. Docker's online repository for publicly available container images, called Docker Hub, hosts over 3.5 million images at the time of writing, making it the world's largest community of container images. We perform an extensive vulnerability analysis of 2500 Docker images. It is of particular interest to perform this type of analysis because the vulnerability landscape is a rapidly changing category, the vulnerability scanners are constantly developed and updated, new vulnerabilities are discovered, and the volume of images on Docker Hub is increasing every day. Our main findings reveal that (1) the number of newly introduced vulnerabilities on Docker Hub is rapidly increasing; (2) certified images are the most vulnerable; (3) official images are the least vulnerable; (4) there is no correlation between the number of vulnerabilities and image features (i.e., number of pulls, number of stars, and days since the last update); (5) the most severe vulnerabilities originate from two of the most popular scripting languages, JavaScript and Python; and (6) Python 2.x packages and jackson-databind packages contain the highest number of severe vulnerabilities. We perceive our study as the most extensive vulnerability analysis published in the open literature in the last couple of years.**

***Index Terms*—Container technology, Docker, Virtual Machines, Vulnerabilities**

1. Introduction

Container technology has been known for a long time in Linux systems through Linux Containers (LXC), but it was not commonly used until a decade ago. The introduction of Docker in [1] made the popularity of containerization rise exponentially. Container technology has revolutionized how software is developed and is seen as a paradigm shift. More concretely, containerization is considered as a beneficial technique for Continuous Integration/Continuous Delivery (CI/CD) pipelines; it is providing an effective way of organizing microservices; it is making it easy to move an application between different environments; and in general, it is simplifying the whole system development life cycle.

Software containers got its name from the shipping industry since the concepts are fundamentally the same. A software container is code wrapped up with all its dependencies so that the code can run reliably and seamlessly in any computer environment isolated from other processes. Hence, containers are convenient, lightweight, and fast technology to achieve isolation, portability, and scalability.

Container technology is replacing virtual machines continuously, and the trend is that more companies are choosing to containerize their applications. Gartner predicts that more than 70% of global companies will have more than two containerized applications in production by 2023. This is an increase from less than 20% in 2019.[1] Docker provides a popular registry service for the sharing of Docker images, called Docker Hub.[2] It currently hosts over 3.5 million container images, and the number keeps growing. Images could be uploaded and maintained by anyone, which creates an innovative environment for anyone to contribute and participate. However, on the downside, this makes it hard for Docker to ensure that packages and applications are up to date to avoid outdated and vulnerable software.

When looking at the security of Docker, two aspects need to be considered: the security of the Docker software at the host, and the security of the Docker containers. Docker Inc. claims that "*Docker containers are, by default, quite secure; especially if you run your processes as non-privileged users inside the container.*" [2]. However, it is a simple fact that Docker (the Docker daemon and container processes) runs with root privileges by default, which exposes a huge attack surface [3]. A single vulnerable container is enough for an adversary to achieve privilege escalation. Hence, the security of the whole Docker ecosystem is highly related to the vulnerability landscape in Docker images.

Related work. One of the first to explore the vulnerability landscape of Docker Hub was BanyanOps [4]. In 2015, they published a technical report revealing that 36% of official images on Docker Hub contained high priority vulnerabilities [4]. Further, they discovered that this number increases to 40% when community images (or general images as they call it in the report) are analyzed. BanyanOps built their own vulnerability scanner based on Common Vulnerabilities and Exposures (CVE)-scores, and analyzed all official images (≈75 repositories with ≈960 unique images) and some randomly chosen community images. However, at that time, Docker Hub consisted of just ≈95,000 images.

1. Gartner: 3 Critical Mistakes That I&O Leaders Must Avoid With Containers
2. Docker Hub webpage: *https://hub.docker.com/*

In 2017, Shu et al. conducted a new vulnerability analysis of Docker Hub images [5]. With the aim of revealing the Docker Hub vulnerability landscape, they created their own analysis framework called DIVA (Docker image vulnerability analysis). The DIVA framework discovers, downloads, and analyses official and community images. It is based on the Clair scanner and uses random search strings to discover images on Docker Hub. The analysis revealed that, on average, an image (official and community) contains more than 180 vulnerabilities. They also found that many images had not been updated for hundreds of days, which is problematic from a security point of view. Further, it was observed that vulnerabilities propagate from parent to child images.

To our knowledge, the most recent vulnerability analysis of Docker Hub images was performed during spring 2019 by Socchi and Luu [6]. They investigated whether the security measures introduced by Docker Inc. (more precisely, the introduction of verified and certified image types) improved the security of Docker Hub. In addition, they inspected the distribution of vulnerabilities across repository types and whether vulnerabilities still are inherited from parent to child image. They implemented their own analyzing software using the Clair scanner, and used the results from Shu et al. [5] from 2017 as a comparison. The data set they successfully analyzed consisted of 757 images in total. Of these, 128 were official, 500 were community, 98 were verified, and 31 were certified. They only analyzed the most recent images in each repository and skipped all Microsoft repositories. Their conclusion was that the security measures introduced by Docker Inc. do not improve the overall Docker Hub security. They stated that the number of inherited vulnerabilities had dropped since the analysis of Shu et al. However, they also found that the average number of new vulnerabilities in child images had increased significantly. Further, they found that the majority of official, community, and certified repositories contain up to 75 vulnerabilities and that the majority of verified images contain up to 180 vulnerabilities.

Our contribution. This is an extended summary of our longer and much more detailed work [7]. We scrutinized the vulnerability landscape in Docker Hub images at the beginning of 2020 within the following framework:

• Images on Docker Hub belong in one of the following four types: "official", "verified", "certified", or "community";

• We used a quantitative mapping of the Common Vulnerability Scoring System (CVSS) [8] (which is a numerical score indicating the severity of the vulnerability in a scale from 0.0 to 10.0) into five qualitative severity rating levels: "critical", "high", "medium", "low", or "none" plus one additional level "unknown".

For performing the analysis of a significant number of images, we used an open-source vulnerability scanner tool and developed our own scripts and tools. All our developed scripts and tools are available from [7].

Our findings can be summarized as follows: **1.** The median value (when omitting the negligible and unknown vulnerabilities) is 26 vulnerabilities per image. **2.** Most of the vulnerabilities were found in the medium severity category. **3.** Around 17.8% (430 images) do not contain any vulnerabilities, and if we are considering negligible and unknown vulnerabilities as no vulnerability, the number increase to as many as 21.6% (523 images). **4.** As intuitively expected, when considering the average, community images are the most exposed. We found that 8 out of the top 10 most vulnerable images are community images. **5.** However, to our surprise, the certified images are the most vulnerable when considering the median value. They had the most high rated vulnerabilities as well as the most vulnerabilities rated as low. As many as 82% of certified images contain at least either one high or critical vulnerability. **6.** Official images come out as the most secure image type. Around 45.9% of them contain at least one critical or high rated vulnerability. **7.** The median value of the number of critical vulnerabilities in images is almost identical for all four image types. **8.** Verified and official images are the most updated, and community and certified images are the least updated. Approximately 30% of images have not been updated for the last 400 days. **9.** There is no correlation between the number of vulnerabilities and the evaluated image features (i.e., the number of pulls, the number of stars, and the last update time). However, the images with many vulnerabilities generally have few pulls and stars. **10.** Vulnerabilities in the Lodash library and vulnerabilities in Python packages are the most frequent and most severe. The top five most severe vulnerabilities are coming from two of the most popular scripting languages, JavaScript and Python. **11.** Vulnerabilities related to execution of code and overflow are the most frequently found critical vulnerabilities. **12.** The most vulnerable package is the `jackson-databind-2.4.0` package, with overwhelming 710 critical vulnerabilities, followed by `Python-2.7.5` with 520 critical vulnerabilities.

Image type	2015 [4]		2017 [5]		2019 [6]		**2020**	
	vuln	avg	vuln	avg	vuln	avg	**vuln**	**avg**
Official	36%	-	80%	75	-	170	**46%**	**70**
Community	40%	-	80%	180	-	150	**68%**	**150**
Verified	-	-	-	-	-	150	**57%**	**90**
Certified	-	-	-	-	-	30	**82%**	**90**

TABLE 1: A summary comparison table of results reported in 2015 [4], in 2017 [5] in 2019 [6] and in our work (2020). The sub-columns "vuln" contain the percentage of images with at least one high rated vulnerability and the "avg" sub-columns contain the average number of vulnerabilities found in each image type.

Last but not least, when put in comparison with the three previous similar studies [4]–[6], our results are summarized in Table 1. Note that some of the cells are empty due to differences in methodologies and types of images when the studies were performed.

2. Preliminaries

Virtualization is the technique of creating a virtual abstraction of some resources to make multiple instances run isolated from each other on the same hardware [9]. There are different approaches to achieve virtualization. One approach is using Virtual Machines (VMs). A VM is a virtualization of the hardware at the host. Hence, each VM has its own kernel, and in order to manage the different VMs, a software called hypervisor is required. The

Repository type	Quantity
Official	160
Verified	250
Certified	51
Community	3,064,454
Total	3,064,915

TABLE 2: Repository type distribution on Docker Hub (February 3rd, 2020

hypervisor emulates the Central Processing Unit (CPU), storage, and Random-Access Memory (RAM), among others, for each virtual machine. This allows multiple virtual machines to run as separate machines on a single physical machine.

In contrast to VMs, containers virtualize the Operating System (OS) level. Every container running on the same machine shares the same underlying kernel, where only bins, libraries, and other run time components are executed exclusively for a single container. In short, a container is a standardized unit of software that contains all code and dependencies [10]. Thus, containers require less memory and achieve a higher level of portability than VMs. Container technology has simplified the software development process as the code is portable, and hence what is run in the development department will be the same as what is run in the production department [11].

On the Docker Hub, image repositories are divided into different categories. Repositories are either private or public and could further be either *official*, *community* or a *verified* repository. In addition, repositories could be certified, which is a subsection of the verified category. The official repositories are maintained and vetted by Docker. Docker vets the verified ones that are developed by third-party developers. Besides being verified, certified images are also fulfilling some other requirements related to quality, support, and best practices [12]. Community images could be uploaded and maintained by anyone. The distribution of the image repository types on Docker Hub can be seen in Table 2. The community repository category is by far the most dominant one and makes up to ≈99% of all Docker Hub repositories.

2.1. Vulnerability databases and categorization method

The severity of vulnerabilities depends on a variety of different variables, and it is highly complex to compare them due to the diversity of different technologies and solutions. Already in 1997, the National Vulnerability Database (NVD) started working on a database that would contain publicly known software vulnerabilities to provide a means of understanding future trends and current patterns [13]. The database can be useful in the field of security management when deciding what software is safe to use and for predicting whether or not software contains vulnerabilities that have not yet been discovered.

Common Vulnerabilities and Exposures (CVE). National Vulnerability Database (NVD) contains Common Vulnerabilities and Exposures (CVE) entries and provides details about each vulnerability like vulnerability overview, Common Vulnerability Scoring System (CVSS), references, Common Platform Enumeration (CPE) and Common Weakness Enumeration (CWE) [14].

CVE is widely used as a method for referencing security vulnerabilities that are publicly known in released software packages. At the time of writing, there were 130,094 entries in the CVE list.[3] The CVE list was created by MITRE Corporation[4] in 1999, whose role is to manage and maintain the list. They work as a neutral and unbiased part in order to serve in the interest of the public. Examples of vulnerabilities found in CVE are frequent errors, faults, flaws, and loopholes that can be exploited by a malicious user in order to get unauthorized access to a system or server. The loopholes can also be used as propagation channels for viruses and worms that contain malicious software [15]. Over the years, CVE has become a recognized building block for various vulnerability analysis and security information exchange systems, much because it is continuously maintained and updated, and because the information is stored with accurate enumeration and orderly naming.

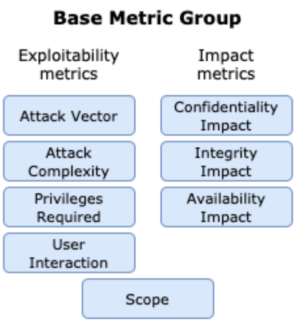


Figure 1: Common Vulnerability Scoring System structure [8]

Common Vulnerability Scoring System (CVSS). The Common Vulnerability Scoring System (CVSS) score is a numerical score indicating the severity of the vulnerability on a scale from zero to 10, based on a variety of metrics. The metrics are divided into three metric groups: Base Metric Group, Temporal Metric Group, and Environmental Metric group. A *Base Score* is calculated by the metrics in the Base Metric Group, and is independent of the user environment and does not change over time. The Temporal Metrics take in the base score and adjusts it according to factors that do change over time, such as the availability of exploit code [8]. Environmental Metrics adjust the score yet again, based on the type of computing environment. This allows organizations to adjust the score related to their IT assets, taking into account existing mitigations and security measures that are already in place in the organization.

In our analysis, it would not make sense to take into account the Temporal or Environmental Metrics as we wanted to discuss the vulnerability landscape independently of the exact time and environment. Therefore, only

3. The number of entries in the CVE list was retrieved 28. Jan 2020 from the official website: *https://cve.mitre.org*

4. MITRE Corporation is a non-profit US organization with the vision to resolve problems for a safer world: *https://www.mitre.org*

Rating	CVSS Score
None	0.0
Low	0.1 - 3.9
Medium	4.0 - 6.9
High	7.0 - 8.9
Critical	9.0 - 10.0

TABLE 3: CVSS Severity scores

the Base Metric group will be described in more detail. It is composed of two sets of metrics: the Exploitability metrics and the Impact metrics, as can be seen in Figure 1 [8]. The first set takes into account *how* the vulnerable component can be exploited and includes attack vector and complexity, what privileges are required to perform the attack, and whether or no user interaction is required. The latter set reflects on the *consequence* of a successful exploit and what impact it has on the confidentiality, integrity, and availability of the system. The last metric is *scope*, which considers if the vulnerability can propagate outside the current security scope.

When the Base Score of a vulnerability is calculated, the eight different metrics from Figure 1 are being considered. Each metric is assigned one out of two to four different values, which is used to generate a vector string. The vector string is then used to calculate the Common Vulnerability Scoring System (CVSS) score, which is a numerical value between 0 and 10. In many cases, it is more beneficial to have a textual value than a numerical value. The CVSS score can be mapped to qualitative ratings where the severity is categorized as either critical, high, medium, low, or none, as can be seen in Table 3 [8].

3. Docker Hub vulnerability landscape

3.1. The distribution of vulnerabilities in each severity category

To determine what the current vulnerability landscape is like in Docker Hub, the number of vulnerabilities found in each severity category is presented in figure 2. As it is interesting to see how many vulnerabilities that are found in total (figure 2a) and how many unique vulnerabilities (figure 2b) there are, both these results are presented in this section.

In figure 2a, the results are based on vulnerability scanning of the complete data set, meaning that this result is based on all found vulnerabilities. The same vulnerability could potentially have multiple entries in the result. This is because a particular vulnerability could be found in multiple images and a single image could contain the same vulnerability in multiple packages. In figure 2b, only unique vulnerabilities are shown. However, some vulnerabilities are present in several severity categories, depending on which image it is found in. In cases like this, all versions of the vulnerability is included, which makes up a total of 14,031 vulnerabilities.

In figure 2a, the negligible and unknown categories clearly stands out, with a total of 315,102 and 240,132 vulnerabilities, respectively. When considering unique vulnerabilities (figure 2b), the medium category is the most dominant one with 5,554 unique vulnerabilities. When examining the relation between figure 2a and 2b, one can observe the ratio of vulnerabilities between severity categories. It becomes clear that the negligible category contains a few number of unique vulnerabilities represented in many Docker images. Whereas the medium category has many unique vulnerabilities represented at a lower ratio. The vulnerability ratio will be explained in detail in the next paragraph.

Severity	Number of vulnerabilities (A)	Number of unique vulnerabilities (B)	Ratio (A/B)
Critical	10,378	206	50
High	44,058	1,313	34
Medium	171,832	5,554	31
Low	137,290	2,326	59
Negligible	315,102	959	329
Unknown	240,132	3,674	65
Total	918,792	14,031	66

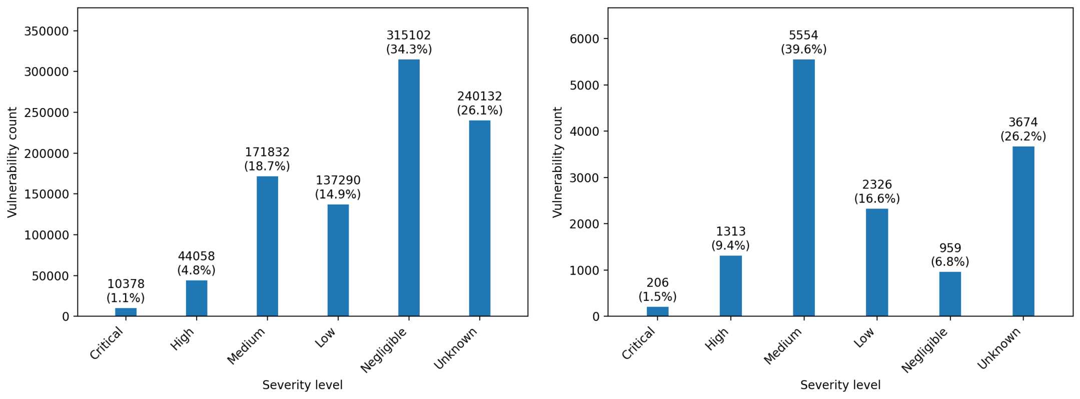
TABLE 4: Vulnerability frequency in severity levels

Table 4 shows the total number of vulnerabilities, the number of unique vulnerabilities, and the ratio, measured as the total number of vulnerabilities divided by the number of unique vulnerabilities. So, for each unique vulnerability, there are a certain number of occurrences of the specific vulnerability in the data set. For example, for each unique vulnerability in the critical category, there are 50 occurrences of this vulnerability in the data set on average. For each unique negligible vulnerability, there are as many as 329 occurrences on average. This is significantly larger than the other values. Despite medium having the highest number of unique vulnerabilities, it has the lowest ratio.

3.2. Central tendency of the vulnerability distribution

We have looked at the average and median values of the number of vulnerabilities in images when disregarding the vulnerabilities that are categorized as negligible and unknown. Looking at Table 4 from the previous section, one can see that negligible and unknown vulnerabilities together make up 555,234 out of the 918,792 vulnerabilities (around 60%). As vulnerabilities in these two categories are considered to contribute with little threat when investigating the current vulnerability landscape, it gives a more accurate result to exclude these. Therefore, we calculated the average and median number of vulnerabilities in images when disregarding negligible and unknown vulnerabilities (counting them as zero). The result was 151 for the average and 26 for the median.

To investigate the data when disregarding the negligible and unknown vulnerabilities further, we created Table 5 that shows statistical values of number of vulnerabilities for each image type. The results show that community images have the highest average and maximum values (158, and 6,509, respectively). The maximum value for community images is significantly larger than the average and the median, which is the case for the other three image types as well. The image type that is considered as the least vulnerable is official. It has the lowest average of 73 and the lowest median value of 9. Further, the maximum value for official images is the second lowest. The lowest maximum value belongs to certified, and is only 428. Although certified has the lowest maximum



(a) Distribution of all 918,792 vulnerabilities (b) Distribution of 14,032 unique vulnerabilities

Figure 2: Vulnerability distribution in severity levels

Image type	Number of analyzed images	Number of vulnera-bilities	Average	Median	Max
Verified	60	6,073	101.2	13	1,128
Certified	22	1,987	90.3	37	428
Official	157	11,489	73.2	9	1,615
Community	2,173	344,009	158.3	28	6,509

TABLE 5: Statistical values for vulnerabilities per image type, disregarding negligible and unknown vulnerabilities.

value, it has the highest median value. This indicates that a larger portion of the images have many vulnerabilities. As a final note, all four image types contain at least one image with zero vulnerabilities.

3.3. Vulnerabilities in each image type

Since the median describes the central tendency better than the average when the data is skewed here we will work with the median values (given in Figure 3). Note that only critical, high, medium and low vulnerabilities are included in the figure. The negligible and unknown vulnerabilities are not included here because they do not usually pose as significant threats, and therefore do not contribute with additional information when investigating the current vulnerability landscape.

The results show that the median of critical vulnerabilities is almost the same for all four image types (4.0 and 3.0). The other severity categories are more varied across the image types. The high severity category is the most represented in certified images, while the medium category is the most represented in the community images. For verified, official and community images, the medium severity has the highest median, while the certified images has the most low vulnerabilities. Overall, it is the certified images that are the most vulnerable.

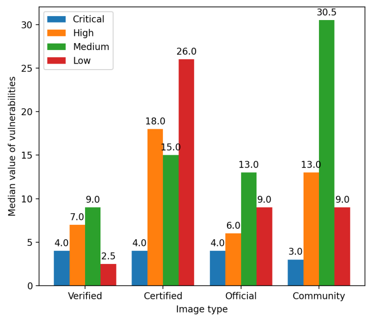


Figure 3: Median values of vulnerabilities for each severity category and image type

3.4. Images that contain the most critical vulnerabilities

Out of all 2,412 successfully analyzed images, this section will present the most vulnerable ones. Table 6 displays the most vulnerable images based on the number of critical vulnerabilities in each image. In cases where the critical count is the same, the image with the highest number of high rated vulnerabilities is considered as the most vulnerable one. The *Number of pulls* column denotes the total number of pulls (downloads) for each image.

Out of the top 10 most vulnerable images, there are 8 community images, 1 official image (silverpeas) and 1 verified image (microsoft-mmlspark-release). There are big variations in the number of vulnerabilities in all presented severity levels. The most vulnerable image, *pivotaldata/gpdb-pxf-dev*, has ~250 more critical vulnerabilities than the second most vulnerable image. However, the second most vulnerable image, *cloudera/quickstart*, contains as many as 2,155 high rated vulnerabilities, which is ~1500 more vulnerabilities than the one rated as the most vulnerable image. It was chosen to focus on the critical vulnerabilities in the ranking of the most vulnera-

	Image	Critical	High	Medium	Low	Number of pulls
1	pivotaldata/gpdb-pxf-dev	822	698	576	132	139,246,839
2	cloudera/quickstart	571	2,155	1,897	158	6,892,856
3	silverpeas	341	264	397	226	828,743
4	microsoft-mmlspark-release	184	428	264	252	1,509,541
5	anchorfree/hadoop-slave	168	636	797	107	5,375,424
6	saturnism/spring-boot-helloworld-ui	133	217	112	2	12,686,987
7	pantsel/konga	133	39	169	0	12,431,685
8	renaultdigital/runner-bigdata-int	127	335	691	103	4,787,745
9	springcloud/spring-pipeline-m2	125	293	2,027	1,357	8,359,973
10	raphacps/simpsons-maven-repo	122	271	399	2	36,136,733

TABLE 6: The most vulnerable images sorted by critical count

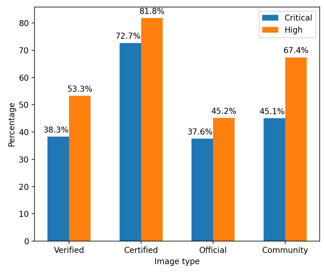


Figure 4: The percentage of images that contain at least one high or critical rated vulnerability.

ble images. This is because it is the highest possible ranking and hence the most severe vulnerabilities will be found in this category. The other severity categories are included in the table as extra information and to give a clear view on the distribution of vulnerabilities. From the number of pulls column one can observe that the most vulnerable image is also the most downloaded one out of the top 10, with as many as 139,246,839 pulls. This is approximately 100 million more pulls compared to the second most pulled image on this list (the *raphacps/simpsons-maven-repo* image). There is no immediate correlation that could be observed between the number of pulls and the number of vulnerabilities in these images.

3.5. Percentage of images with critical and high vulnerabilities

It is enough with a single vulnerability for a system to be compromised. Thus, we determine what percentage of images that contain at least one high or critical rated vulnerability for each image type, as shown in Figure 4.

Our results (Figure 4) reveal that the certified image type, which is a subsection of the verified image type, is the most vulnerable by the means of this measure. 81.8% of all certified images contain at least one vulnerability with high severity level and 72.7% of them contain at least one critical vulnerability. Community images come out as the second most vulnerable image type. 67.4% have high vulnerabilities and 45.1% have critical vulnerabilities. The third most vulnerable image type is verified, followed by official.

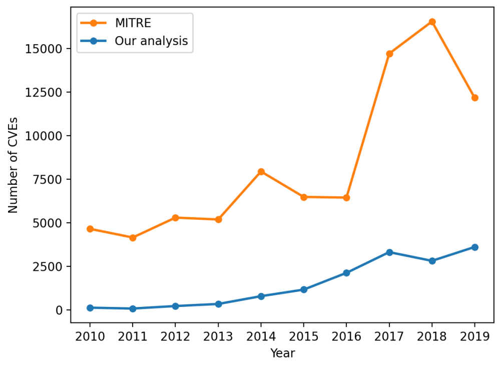
When combining these results, to investigate what amount of the image types that contain *either* at least one critical or high rated vulnerability, the results are as follows: 81.8% for certified images, 68.4% for community images, 56.7% for verified images and 45.9% for official images. This makes the official images the least vulnerable image type. However, it should be emphasized that still almost half of the official images contain critical or high rated vulnerabilities as presented in this section.

3.6. The trend in CVE vulnerabilities

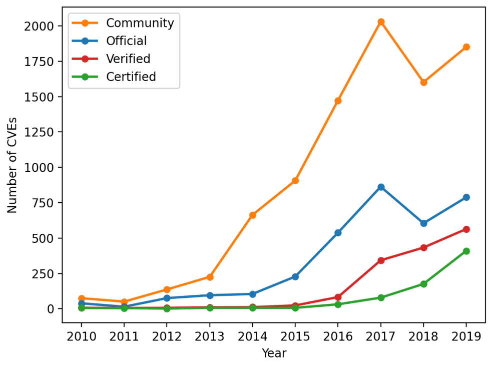
This section will focus on the trend of all reported Common Vulnerabilities and Exposures (CVE) vulnerabilities each year compared to the number of unique CVE vulnerabilities found throughout our analysis. Data gathered from the CVE Details database [16] is used to display the number of new reported Common Vulnerabilities and Exposures (CVE) vulnerabilities each year.

In Figure 5a the reported Common Vulnerabilities and Exposures (CVE) vulnerabilities each year is presented together with the unique CVE vulnerabilities found in our analysis from 2010 to 2019. The orange line shows how the number of new discovered CVE vulnerabilities varies by a few thousand vulnerabilities each year. However, there is a significant increase in 2017. This increase is not reflected in the data from our analysis, which is following a steady increase in the years from 2014 to 2017. This increase can be explained by the introduction of Docker Hub in 2014, making new vulnerabilities more represented in images. As a final observation, the number of new reported vulnerabilities from MITRE between 2018 and 2019 is decreasing, while there is an increase in our results.

Figure 5b shows the number of unique vulnerabilities found in each image type (i.e. community, official, verified and certified) in our analysis from 2010 to 2019. This figure gives an insight in how the overall changes are reflected in each image type. Verified and certified images have had an increase in the number of unique Common Vulnerabilities and Exposures (CVE) vulnerabilities each year from 2015. Community and official images, however, have had a significant decrease of unique vulnerabilities from 2017 to 2018. It is noteworthy to point out that the curves are affected by the time of introduction of the different image types. Official images were introduced in 2014, whereas verified and certified images were introduced in 2018.



(a)



(b)

Figure 5: CVE trend from 2010 to 2019, (a) displays all reported CVEs and all found, unique CVEs in our analysis, (b) displays the CVEs in the different image types from our analysis.

Image type	More than 400 days	More than 200 days	Less than 14 days
Community	33.9%	47.0%	27.0%
Official	9.6%	14.7%	51.3%
Certified	18.2%	36.4%	13.6%
Verified	1.7%	5.0%	83.3%

TABLE 7: The time since last update for all image types presented in percentage

3.7. Days since last update

There is a high variation in how often Docker Hub images are updated. Intuitively, this affects the vulnerability landscape of Docker Hub. Hence, we have gathered data about when images were last updated, and calculated the number of days since the images were last updated, counting back from February 25th, 2020. The data set consists of last updated data for all analyzed images, except five.

A brief analysis of the numbers from our database revealed that 31.4% of images have not been updated in 400 days or longer and 43.8% have not been updated in 200 days or longer. The percentage of images that have been updated during the last 14 days are 29.8%. This implies that if these numbers are representative for all images on Docker Hub, a third of the images (31.4%) on Docker Hub have not been updated in the last 400 days or longer.

To go into more detail, Table 7 presents how often images in each of the image types are updated. Community and certified images are the least updated image categories, where 47.0% of community images and 36.4% of certified images have not been updated for the last 200 days or more. The verified images are the most frequently updated category, where 83.3% of images have been updated during the last 14 days.

A handful of certified images are highly affecting the percentages from Table 7, because the overall number of certified images is small. Official images contain a high portion of images that have been updated recently (January 2020 to March 2020), and some more spread values with images that have not been updated since 2016. The verified images are the most updated image type, where there is only one image with the last updated time earlier than May 2019.

4. Correlation between image features and vulnerabilities

We investigate whether or not the number of vulnerabilities in an image is affected by a specific image feature, such as the number of times the image has been pulled, the number of stars an image has been given, or the number of days since the image was last updated. In order to find out whether there is a correlation, we used Spearman's r_s correlation coefficient [17]. Spearman's correlation was chosen because our data set contain skewed values and are not normally distributed. When handling entries that contained empty values, we opted for the approach of complete case analysis, which means omitting incomplete pairs. The alternative would be imputation of missing values, which means to create an estimated value based on the other data values. However, this approach was not chosen because the values of our data set are independent of each other.

Correlation between pulls and vulnerabilities. To check the folklore wisdom about the following correlation: *images with the most pulls generally have few vulnerabilities, and images with the most vulnerabilities generally have few pulls*, we created a scatter plot given in Figure 6. However, after calculating the Spearman correlation coefficient between the number of pulls and number of vulnerabilities for the whole set of investigated images we got $r_s = -0.1115$. This is considered as no particular correlation. To explain this, we refer to the meaning of having a high negative correlation: the markers would gather around a decreasing line (not necessarily linear), indicating that images with more pulls have less number of vulnerabilities. In the case of high positive correlation, the opposite would apply i.e. the line would be increasing.

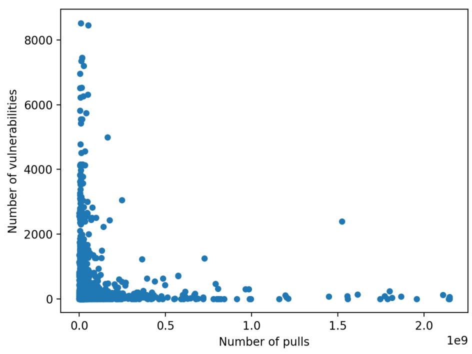


Figure 6: Number of pulls and number of vulnerabilities for each image

Correlation between stars and vulnerabilities. The correlation coefficient between the number of stars and number of vulnerabilities is $r_s = -0.0335$. Figure 7 shows the scatter plot when including number of stars instead of number of pulls. The plot is similar to Figure 6, but the correlation is even weaker.

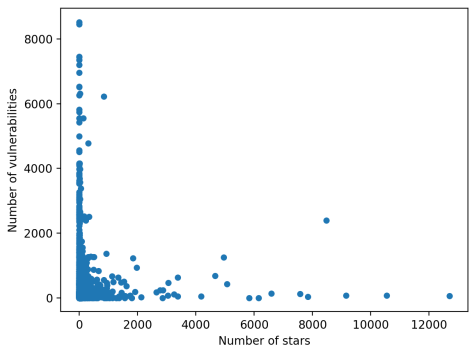


Figure 7: Number of stars and number of vulnerabilities for each image

Correlation between time since last update and vulnerabilities. This correlation is calculated by computing the number of days since the last update counting from the day we gathered the data (which was February 25, 2020). The correlation was $r_s = 0.1075$, which shows a positive correlation as opposed to the other two. Figure 8 shows the scatter plot, and although the markers are approaching an increasing line a tiny bit, this is minimal. The value of 0.1075 is still not enough to state that there is a strong correlation between the number of vulnerabilities and time since the last update. The markers slightly approach an increasing line, indicating a weak tendency that there are more vulnerabilities in images that have not been updated for a long time. Still, the distribution of markers is relatively even along the x-axis with the most markers in the lower part of the y-axis, supporting that there is no correlation.

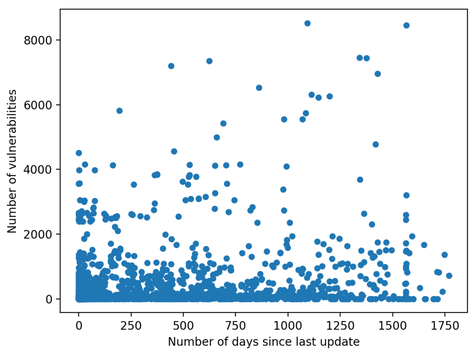


Figure 8: Number of days since last update and number of vulnerabilities for each image

5. The most severe vulnerabilities

5.1. The most represented critical vulnerabilities

The most represented severe vulnerabilities are, intuitively, the ones having the highest impact on the vulnerability landscape. Table 8 presents the most represented critical rated vulnerabilities in descending order. The results are obtained by counting the number of occurrences for each vulnerability ID in the critical severity level. The critical count column is the number of occurrences for a specific vulnerability. Lastly, the type(s) column presents the vulnerability type of each of the vulnerabilities. This data is gathered from the CVE Details database [18].

	Vulnerability ID	Critical count	Type(s)
1	CVE-2019-10744	466	Improper Input Validation
2	CVE-2017-1000158	464	Execute Code, Overflow
3	CVE-2019-9948	378	Bypass a restriction or similar
4	CVE-2019-9636	374	Credentials Management Errors
5	CVE-2018-16487	365	Security Features
6	CVE-2018-14718	354	Execute Code
7	CVE-2018-11307	337	Deserialization of Untrusted Data
8	CVE-2018-7489	318	Execute Code, Bypass a restriction or similar
9	CVE-2016-5636	302	Overflow
10	CVE-2017-15095	295	Execute Code

TABLE 8: The most represented vulnerabilities (based on critical severity level).

5.2. Vulnerability characteristics

We elaborate the top five most represented vulnerabilities presented in Table 8 regarding their characteristics and common features[5]. The top five severe vulnerabilities are coming from two most popular script languages: JavaScript and Python. As a general observation, the execute code is the most common vulnerability type, followed by overflow.

The most represented critical vulnerability is found 466 times throughout our scanning. It has vulnerability ID *CVE-2019-10744*, and a base score of 9.8, which is in the upper range of the critical category (to examine how base scores are determined, see Section 2.1). The vulnerability is related to the JavaScript library lodash, which is commonly used as a utility function provider in relation to functional programming. This particular vulnerability is related to improper input validation and makes the software vulnerable to prototype pollution. It is affecting versions of lodash lower than 4.17.12 [19]. In short, this means that it is possible for an adversary to execute arbitrary code by modifying the properties of the Object.prototype. This is possible as most JavaScript objects inherit the properties of the built in Object.prototype object. The fifth vulnerability on the list, *CVE-2018-16487*, is also related to lodash and the prototype pollution vulnerability.

Further, the second, third and fourth most represented critical vulnerabilities are related to Python vulnerabilities. The second vulnerability with vulnerability ID, *CVE-2017-1000158*, is related to versions of Python up to 2.7.13. The base score is rated 9.8, and the vulnerability enables arbitrary code execution to happen through an integer overflow leading to a heap-based buffer overflow [20]. Overflow vulnerabilities could be of different types, for instance heap overflow, stack overflow and integer overflow. Heap overflow and stack overflow are related to overflowing a buffer, whereas integer overflow could lead to a buffer overflow. A buffer overflow is related to overwriting a certain allocated buffer, causing adjacent memory locations to be overwritten. Any exploit of these kinds of vulnerabilities are typically related to the execution of arbitrary code, where the adversary is taking advantage of the buffer overflow vulnerability to run malicious code.

The third presented vulnerability with vulnerability ID *CVE-2019-9948* is affecting the Python module urllib in Python version 2.x up to 2.7.16. It is rated with 9.1 as base score. This vulnerability makes is easier to get around security mechanisms that blacklist the `file:URIs` syntax, which in turn could give an adversary access to local files such as the */etc/passwd* file [21]. The fourth vulnerability is found 374 times and has vulnerability ID *CVE-2019-9636*. It is affecting both the second and the third version of Python (versions 2.7.x up to 2.7.16, and 3.x up to 3.7.2). This vulnerability is also related to the urllib module, more precisely, incorrect handling of unicode encoding. The result is that information could be sent to different hosts than intended if it was parsed correctly [22]. It has a base score of 9.8.

5. Information about all vulnerabilities could be found by visiting https://nvd.nist.gov/vuln/detail/

	Package	Critical count	Image count
1	jackson-databind-2.4.0	710	15
2	Python-2.7.5	520	207
3	jackson-databind-2.9.4	354	4
4	lodash-3.10.1	312	76
5	silverpeas-6.0.2	280	1
6	Python-2.7.13	248	141
7	Python-2.7.16	224	117
8	jackson-databind-2.6.7.1	215	13
9	jackson-databind-2.9.6	192	12
10	Python-2.7.12	185	107

TABLE 9: The most vulnerable packages (based on critical severity level).

6. Vulnerabilities in packages

6.1. The most vulnerable packages

Table 9 presents the packages that contain the most critical vulnerabilities. The critical count column is obtained by counting the total number of occurrences of critical vulnerabilities in each package, while the image count column is the number of images that uses each package.

There is a clear relation between the most vulnerable packages and the most represented vulnerabilities (Section 5), as expected. For example, vulnerabilities found in Python version 2.x packages and in the Lodash package are both presented in Section 5.

From Table 9, one can observe that the Python packages are by far the most used packages, and therefore they expose the biggest impact regarding the threat landscape. The lodash-3.10.1 package is found in 76 images. This package contains the prototype pollution vulnerability affecting JavaScript code, which also is the most represented vulnerability in Table 8. Further, the jackson-databind package is represented with four different versions in Table 9 (entry 1, 3, 8 and 9). This package is used to transform JSON objects to Java objects (Lists, Numbers, Strings, Booleans, etc.), and vice versa. In total, these packages are used by 44 images: a relatively low amount compared to the usage of the Python packages. Finally, the silverpeas-6.0.2 package contains 280 critical vulnerabilities and is only used by a single image: the silverpeas image on Docker Hub.[6]

6.2. Vulnerabilities in popular packages

When considering the packages that have the most critical vulnerabilities (Table 9), some of the packages are only used by a few images (like the silverpeas package). Therefore, Table 10 is presented, as it is desirable to see what the vulnerability distribution is like in the most popular packages. The table shows the most used packages and the number of vulnerabilities that are present in them, considering all security levels. The image count column contain the number of images that use this package.

As observable from Table 10, the most used packages are not containing any critical, high, medium or low vulnerabilities (except for one entry). However, they are containing a vast number of negligible vulnerabilities, which is of less significance from a security point of view, as mentioned in previous sections.

6. *https://hub.docker.com/_/silverpeas*

	Package	Critical	High	Medium	Low	Negligible	Unknown	Image count
1	tar-1.29b-1.1	0	0	0	0	482	0	241
2	coreutils-8.26-3	0	0	0	0	240	0	240
3	libpcre3-2:8.39-3	0	0	0	0	956	0	239
4	login-1:4.4-4.1	0	0	0	0	714	0	238
5	passwd-1:4.4-4.1	0	0	0	0	708	0	236
6	sensible-utils-0.0.9	0	0	103	0	0	111	214
7	libgcrypt20-1.7.6-2+deb9u3	0	0	0	0	211	0	211
8	libgssapi-krb5-2-1.15-1+deb9u1	0	0	0	0	621	0	207
9	libk5crypto3-1.15-1+deb9u1	0	0	0	0	621	0	207
10	libkrb5-3-1.15-1+deb9u1	0	0	0	0	621	0	207

TABLE 10: Vulnerabilities in the most used packages.

7. Conclusions

This paper summarizes the findings that we reported in a longer and much more detailed work [7]. We studied the vulnerability landscape in Docker Hub images by analyzing 2500 Docker images of the four image repository categories: official, verified, certified images, and community. We found that as many as 82% of certified images contain at least one high or critical vulnerability, and that they are the most vulnerable when considering the median value. Official images came out as the most secure image type with 45.9% of them containing at least one critical or high rated vulnerability. Only 17.8% of the images did not contain any vulnerabilities, and we found that the community images are the most exposed as 8 out of the top 10 most vulnerable images are community images.

Concerning the technical specifics about the vulnerabilities, we found that the top five most severe vulnerabilities are coming from two of the most popular scripting languages, JavaScript and Python. Vulnerabilities in the Lodash library and vulnerabilities in Python packages are the most frequent and most severe. Furthermore, the vulnerabilities related to execution of code and overflow are the most frequently found critical vulnerabilities.

Our scripts and tools are available from [7].

References

[1] A. Avram, "Docker: Automated and consistent software deployments," *InfoQ. Retrieved*, pp. 08–09, 2013.

[2] "Docker security," https://docs.docker.com/engine/security/security/, access date: 15. Apr 2020.

[3] T. Micro, "Why running a privileged container in docker is a bad idea," https://blog.trendmicro.com/trendlabs-security-intelligence/why-running-a-privileged-container-in-docker-is-a-bad-idea/, 2019, access date: 15. Apr 2020.

[4] J. Gummaraju, T. Desikan, and Y. Turner, "Over 30% of official images in docker hub contain high priority security vulnerabilities," May 2015, access date: 5. Mar 2020.

[5] R. Shu, X. Gu, and W. Enck, "A study of security vulnerabilities on docker hub," 2017, access date: 5. Mar 2019.

[6] E. Socchi and J. Luu, "A deep dive into docker hub's security landscape - a story of inheritance?" https://www.duo.uio.no/bitstream/handle/10852/69632/A-Deep-Dive-into-Docker-Hubs-Security-Landscape.pdf?sequence=1&isAllowed=y, 2019, access date: 5. Mar 2019.

[7] K. Wist and M. Helsem, "An Extensive Analysis of the Current Vulnerability Landscape in Docker Hub Images," Master's thesis, Norwegian University of Science and Technology (NTNU), 2020.

[8] FIRST, "Common vulnerability scoring system version 3.1: Specification document," https://www.first.org/cvss/specification-document, 2019, access date: 3. Feb 2020.

[9] D. Barrett and G. Kipper, "Virtualization technique," https://www.sciencedirect.com/topics/computer-science/virtualization-technique, 2010, access date: 17. Apr 2020.

[10] D. Inc., "What is a container?" https://www.docker.com/resources/what-container, access date: 31. Jan 2020.

[11] C. Anderson, "Docker," https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7093032, 2015, access date: 3. Feb 2020.

[12] J. Morgan, "Introducing the new docker hub," https://www.docker.com/blog/the-new-docker-hub/, 2018, access date: 3. Feb 2020.

[13] Z. Zhang, D. Caragea, and X. Ou, "An empirical study on using the national vulnerability database to predict software vulnerabilities," https://link.springer.com/chapter/10.1007/978-3-642-23088-2_15, 2011, access date: 27. Jan 2020.

[14] S. Na, T. Kim, and H. Kim, "A study on the classification of common vulnerabilities and exposures using Naïve Bayes," https://link.springer.com/chapter/10.1007/978-3-319-49106-6_65, 2017, access date: 28. Jan 2020.

[15] Z. Chen, Y. Zhang, and Z. Chen, "A categorization framework for common computer vulnerabilities and exposures," https://academic.oup.com/comjnl/article/53/5/551/415583, 2009, access date: 28. Jan 2020.

[16] "Browse vulnerabilities by date," https://www.cvedetails.com/browse-by-date.php, 2019, access date: 21. Apr 2020.

[17] A. Lehman, *JMP for basic univariate and multivariate statistics: a step-by-step guide*. SAS Institute, 2005.

[18] "Cve details," https://www.cvedetails.com/, 2020, access date: 20. Apr 2020.

[19] "Cve-2019-10744 detail," https://nvd.nist.gov/vuln/detail/CVE-2019-10744, 2019, access date: 27. Mar 2020.

[20] "Cve-2017-1000158 detail," https://nvd.nist.gov/vuln/detail/CVE-2017-1000158, 2019, access date: 27. Mar 2020.

[21] "Cve-2019-9948 detail," https://nvd.nist.gov/vuln/detail/CVE-2019-9948, 2019, access date: 27. Mar 2020.

[22] "Cve-2019-9636 detail," https://nvd.nist.gov/vuln/detail/CVE-2019-9636, 2019, access date: 27. Mar 2020.

Katrine Wist and Malene Helsem

An Extensive Analysis of the Current Vulnerability Landscape in Docker Hub Images

NTNU

Kunnskap for en bedre verden