

A Deep Dive into Docker Hub's Security Landscape

A story of inheritance?

Emilien Socchi
Jonathan Luu



Thesis submitted for the degree of
Master in Network and System Administration
30 credits

Department of Informatics
Faculty of Mathematics and Natural Sciences

UNIVERSITY OF OSLO

Spring 2019

A Deep Dive into Docker Hub's Security Landscape

A story of inheritance?

Emilien Socchi
Jonathan Luu

© 2019 Emilien Socchi, Jonathan Luu

A Deep Dive into Docker Hub's Security Landscape

<http://www.duo.uio.no/>

Printed: Reprosentralen, University of Oslo

Abstract

Docker containers have become a popular virtualization technology for running multiple isolated application services on a single host using minimal resources. That popularity has led to the creation of an online sharing platform known as Docker Hub, hosting images that Docker containers instantiate. In this thesis, a deep dive into Docker Hub's security landscape is undertaken. First, a Python based software used to conduct experiments and collect metadata, parental and vulnerability information about any type of image available on Docker Hub is developed. Secondly, our tool allows analyzing the most recent image found in each Certified, Verified and Official repository, as well the most recent image found in 500 random Community repositories among the most popular ones. Using our software named Docker imAge analyZER (DAZER), the following discoveries were made: (1) the Certified and Verified repositories introduced by Docker Inc. in December 2018 do not improve the overall Docker Hub's security landscape in a way that is significant; (2) the most influential parent images on Docker Hub are all Official images and although vulnerabilities in the platform are still inherited in a highly manner, they do not tend to be introduced by the top root parents as suggested by previous studies; (3) the average number of unique vulnerabilities found across all types of repositories is expected to grow with a rate of approximately 105 vulnerabilities per year between 2019 and 2025 if Docker Hub's security landscape continues evolving the same way. While set in perspective with results from previous studies, our findings demonstrate the deterioration of Docker Hub's security landscape over the years and the strong need for automated Docker image security updates of a significantly higher quality than what today's procedures are offering.

Acknowledgements

First and foremost, we would like to express our sincere gratitude and appreciation to our supervisors I. Hassan and V. Tasoulas for their support and enthusiasm throughout the entire thesis. Their constant availability and constructive feedback provided valuable guidance, as well as inspirational encouragements during the entire project.

Secondly, we would like to express a special thanks to our closest friends and family who helped us getting through this demanding but exciting master's studies that is the Network and System Administration (NSA) program.

Finally, we wish to express our sincere appreciation to Oslo Metropolitan University (OsloMet) and the University of Oslo (UiO) for giving us the opportunity to take part in the NSA program and thank all of our lecturers for their inspiring work and constant dedication.

Oslo, May 2019

Emilien Socchi & Jonathan Luu

Preface

The basis of this research originally stemmed from the master's topic proposed by V. Tasoulas regarding the investigation of container security in the world of microservices. Our initial background survey revealed that a strong need for examining the security landscape of the biggest container image sharing platform known as Docker Hub was needed, as very little study had been conducted on the subject so far. Both interested in conducting research about the same topic, we decided collaborating in order to enhance our productivity and demonstrate that a binomial cooperation may produce increased valuable results and contributions for the research community.

Our final contributions in this research are multiple and are not strictly limited to the scope of the problem statement. Finally, we have intended to make the reading of this thesis as easy as possible, by writing important keywords and concepts in the background chapter in *italic*. Moreover, important findings are summarized in the result and analysis chapters for better readability and understanding, while all the details are available in their entirety in the appendix.

We hope that you enjoy your reading and find our research of interest.

Contents

| | | |
|----------|-------------------------------------|-----------|
| 1 | Introduction | 1 |
| 1.1 | Motivation | 1 |
| 1.2 | Problem statement | 2 |
| 1.3 | Thesis outline | 3 |
| 2 | Background and literature | 5 |
| 2.1 | Software vulnerabilities | 5 |
| 2.1.1 | What is a software vulnerability? | 5 |
| 2.1.2 | Enumerating vulnerabilities | 6 |
| 2.1.3 | Classifying vulnerabilities | 6 |
| 2.1.4 | Severity levels | 7 |
| 2.2 | Software containers | 8 |
| 2.2.1 | What is a software container? | 8 |
| 2.2.2 | Container vs. Virtual Machine (VM)? | 9 |
| 2.3 | Docker | 9 |
| 2.3.1 | What is Docker? | 10 |
| 2.3.2 | What is a Docker container? | 10 |
| 2.3.3 | How are Docker images distributed? | 11 |
| 2.3.4 | Docker's architecture | 11 |
| 2.4 | The Docker engine | 12 |
| 2.4.1 | What is the Docker engine? | 12 |
| 2.4.2 | Managing images | 13 |
| 2.5 | Docker Hub | 14 |
| 2.5.1 | What is Docker Hub? | 14 |
| 2.5.2 | Repository types | 15 |
| 2.5.3 | Repository naming convention | 17 |
| 2.5.4 | Docker image reusability | 18 |
| 2.5.5 | Docker image dependencies | 18 |
| 2.5.6 | Have you said API? | 20 |
| 2.6 | Docker Hub's security landscape | 20 |
| 2.6.1 | Current knowledge | 20 |
| 2.6.2 | Docker Inc.'s response | 21 |
| 3 | Methodology | 23 |
| 3.1 | Objectives | 23 |
| 3.2 | Design | 25 |
| 3.2.1 | Data set definition | 25 |

| | | |
|----------|---|-----------|
| 3.2.2 | Preliminary requirements | 26 |
| 3.2.3 | Overview | 27 |
| 3.2.4 | Result data format definition | 28 |
| 3.2.5 | Detailed research questions definition | 30 |
| 3.3 | Implementation | 30 |
| 3.3.1 | Tools and technologies | 31 |
| 3.3.2 | Architecture | 32 |
| 3.3.3 | Intended workflow | 33 |
| 3.3.4 | Research queries definition | 34 |
| 3.4 | Measurements and analysis | 35 |
| 3.5 | Expected results | 35 |
| 4 | Result 1: Design | 37 |
| 4.1 | Data set | 37 |
| 4.1.1 | Defined data set | 37 |
| 4.1.2 | Skipped repositories | 38 |
| 4.2 | Preliminary requirements | 39 |
| 4.2.1 | Two parent databases | 39 |
| 4.2.2 | Manual image checkout | 41 |
| 4.3 | Overview | 42 |
| 4.4 | Designed result data format | 43 |
| 4.5 | Detailed research questions | 46 |
| 5 | Result 2: Implementation | 49 |
| 5.1 | Tools and technologies | 49 |
| 5.2 | Retrieving data | 50 |
| 5.2.1 | The Docker Hub API: version 1 | 50 |
| 5.2.2 | The Docker Hub API: version 2 | 52 |
| 5.2.3 | CIRCL's CVE API | 53 |
| 5.2.4 | The MicroBadger API | 54 |
| 5.2.5 | The Red Hat security data API | 54 |
| 5.2.6 | Enterprise Linux Security Advisory | 54 |
| 5.3 | Implemented architecture | 55 |
| 5.4 | Implemented workflow | 56 |
| 5.5 | Getting ready for analysis | 58 |
| 5.5.1 | Importing result data to MongoDB | 59 |
| 5.5.2 | Research queries | 59 |
| 6 | Result 3: Measurements | 63 |
| 6.1 | Describing the results | 63 |
| 6.2 | RQ3: Vulnerability distribution across repository types | 67 |
| 6.2.1 | Quantitative vulnerability distribution | 67 |
| 6.2.2 | Severity distribution | 69 |
| 6.2.3 | Vulnerable image distribution | 71 |
| 6.2.4 | Potential correlations | 73 |
| 6.3 | RQ2: Vulnerabilities and inheritance | 80 |
| 6.4 | RQ1: Certified and Verified vs. Official and Community repositories | 81 |
| 6.5 | Additional research question | 84 |
| 6.6 | Summary | 86 |

| | | |
|----------|--|------------|
| 7 | Analysis | 87 |
| 7.1 | Vulnerability distributions and predictions | 87 |
| 7.1.1 | General interpretation | 87 |
| 7.1.2 | Interpreting box plots | 88 |
| 7.1.3 | Interpreting density plots | 90 |
| 7.1.4 | Analyzing potential quantitative vulnerability correlations between dependent repository types | 94 |
| 7.1.5 | Predicting quantitative software vulnerabilities by 2025 | 96 |
| 7.2 | Parental relationships and vulnerability inheritance | 97 |
| 7.2.1 | Modelling parental and vulnerability relationships in a network | 99 |
| 7.2.2 | Analyzing egocentric networks | 100 |
| 8 | Discussion | 107 |
| 8.1 | Validity of the study | 107 |
| 8.1.1 | Analyzed set of Docker images | 107 |
| 8.1.2 | Applied methodology | 107 |
| 8.1.3 | Software vulnerability identification | 108 |
| 8.1.4 | Unidentifiable parent images | 108 |
| 8.1.5 | Discovered vulnerabilities and exploitability | 109 |
| 8.2 | Encountered challenges | 109 |
| 8.2.1 | Retrieving data from Docker Hub | 109 |
| 8.2.2 | Manual image checkout | 109 |
| 8.2.3 | Overwhelming the Docker engine | 109 |
| 8.2.4 | Image parent retrieval | 110 |
| 8.2.5 | Confusing terminology | 111 |
| 8.3 | Future work | 112 |
| 9 | Conclusion | 113 |
| | References | 115 |
| A | Excluded repositories | 120 |
| A.1 | Paid repositories | 120 |
| A.2 | Manifest not found error | 121 |
| A.3 | No matching manifest or incompatible platform error | 121 |
| A.4 | Pull access denied error | 122 |
| A.5 | Manual checkout of repositories (kept) | 122 |
| A.6 | Summary | 124 |
| B | Scripts | 125 |
| B.1 | Installing the required tools for the VMs | 125 |
| B.2 | Setup of the environment | 126 |
| B.2.1 | Requirements for *.nix | 126 |
| B.2.2 | Requirements for Windows | 126 |
| B.2.3 | Prerequisite | 126 |
| B.2.4 | Getting Started | 127 |
| C | Research queries | 129 |
| C.1 | MongoDB queries | 129 |
| C.2 | Miscellaneous MongoDB queries | 137 |

| | | |
|----------|--|------------|
| D | Result data | 139 |
| D.1 | Top ten most vulnerable repositories across image types | 139 |
| D.2 | Top ten most pulled repositories across image types | 141 |
| D.3 | Top ten last updated repositories across image types | 142 |
| D.4 | All base repositories across image types - sorted by popularity | 144 |
| D.5 | Top ten most vulnerable base repositories across image types | 147 |
| D.6 | Top ten most used parent images across image types | 148 |
| D.6.1 | Top ten most used parent images across all repository types | 150 |
| D.7 | Top ten most vulnerable packages | 151 |
| D.7.1 | Across all repository types | 151 |
| D.7.2 | Across the most popular parents | 151 |
| D.8 | CWE vulnerability categories | 152 |
| D.9 | Predicting an estimation of total vulnerabilities across repository types between 2019 and 2025 | 154 |
| E | Source code | 156 |
| E.1 | dockerhub_api.py | 156 |

List of Figures

| | | |
|------|---|----|
| 2.1 | The container creation process | 10 |
| 2.2 | Docker’s architecture | 12 |
| 2.3 | The Docker engine | 13 |
| 2.4 | The Docker engine’s use of short layer IDs and long image digests | 14 |
| 2.5 | Dependencies of the Official Tomcat image on Docker Hub | 19 |
| 3.1 | The thesis’ methodology | 24 |
| 3.2 | The planned parent database’s design | 27 |
| 3.3 | The planned design’s overview | 28 |
| 3.4 | The planned architecture | 33 |
| 3.5 | The planned experiments’ workflow | 34 |
| 4.1 | The designed Official parent database | 40 |
| 4.2 | The implemented design’s overview | 42 |
| 5.1 | The implemented architecture | 55 |
| 5.2 | The DAZER software’s workflow | 58 |
| 6.1 | Analyzed Official repositories distribution | 65 |
| 6.2 | Analyzed Community repositories distribution | 65 |
| 6.3 | Analyzed Verified repositories distribution | 66 |
| 6.4 | Analyzed Certified repositories distribution | 67 |
| 6.5 | Distribution of unique vulnerabilities per repository type and per year | 69 |
| 6.6 | Distribution of severity levels for unique vulnerabilities across repository types | 70 |
| 6.7 | Distribution of images across repository types with a critical and high severity | 72 |
| 6.8 | Distribution of images across repository types with a medium and low severity | 73 |
| 6.9 | The top 10 most vulnerable and most pulled Official repositories | 74 |
| 6.10 | The top 10 most vulnerable and most pulled Community repositories | 75 |
| 6.11 | The top 10 most vulnerable and most pulled Verified repositories | 76 |
| 6.12 | The top 10 most vulnerable and most pulled Certified repositories | 76 |
| 7.1 | Total number of contained vulnerabilities per image across repository types | 89 |
| 7.2 | Density distribution of the total number of contained vulnerabilities per Official and Community image | 91 |
| 7.3 | Density distribution of the total number of contained vulnerabilities per Verified and Certified image | 92 |
| 7.4 | Density distribution of the total number of contained vulnerabilities per image across repository types | 93 |

| | | |
|------|--|-----|
| 7.5 | Linear relationships of the total number of unique vulnerabilities between in each type of repository | 95 |
| 7.6 | Estimating the total vulnerabilities across repository types by year 2025 | 97 |
| 7.7 | Direct and indirect parental relationships to the Official alpine:3.8 image | 98 |
| 7.8 | Parental relationships and vulnerability inheritance in the network of analyzed Docker images | 99 |
| 7.9 | Parental relationships and vulnerability inheritance related to the Official alpine:3.8 image | 101 |
| 7.10 | Parental relationships and vulnerability inheritance related to the Official debian:9-slim image | 102 |
| 7.11 | Parental relationships and vulnerability inheritance related to the Official java:openjdk-8-jre image | 103 |
| 7.12 | Parental relationships and vulnerability inheritance related to the Official debian:latest image | 104 |
| 7.13 | Parental relationships and vulnerability inheritance related to the Official ubuntu:xenial image | 105 |
| 7.14 | Parental relationships and vulnerability inheritance related to the Official debian:stretch-20180716 image | 106 |
| 8.1 | Docker Hub’s confusing terminology | 111 |

List of Tables

| | | |
|------|--|-----|
| 2.1 | NVD's Common Vulnerability Scoring System (CVSS) [16] | 7 |
| 2.2 | Main differences between VMs and software containers | 9 |
| 2.3 | Docker Hub's repository type distribution as of April 5th 2019 | 16 |
| 2.4 | Docker Hub's paid repository distribution as of April 5th 2019 | 16 |
| 2.5 | Docker Hub's namespaces per repository type | 17 |
| 4.1 | A summary of the experiments performed in this study | 39 |
| 6.1 | A summary of the experiments performed in this study | 64 |
| 6.2 | Quantitative vulnerability distribution across repository types | 68 |
| 6.3 | Distribution of severity levels for unique vulnerabilities across repository types | 71 |
| 6.4 | Comparison of the average number of total vulnerabilities per repository type for the last 10 updated and the complete set of repositories | 77 |
| 6.5 | Correlations between vulnerabilities found in base and non-base images | 79 |
| 6.6 | The top 10 vulnerable packages across repository types with their corresponding CVE number and CWE-ID | 80 |
| 6.7 | Introduced and inherited vulnerabilities across repository types | 81 |
| 6.8 | The ten most popular vulnerability categories across all types of repositories | 83 |
| 6.9 | Correlations between the ten most popular parent images and the ten most vulnerable packages across all types of repositories | 84 |
| 6.10 | Correlations between the ten most popular parent images and the ten most vulnerable packages found across those images | 85 |
| 7.1 | Descriptive statistics of the total number of vulnerabilities found in each repository type | 88 |
| 7.2 | Share of images in each type of repository with less than or 180 contained vulnerabilities | 93 |
| 7.3 | The top 10 most popular parent images in the network of analyzed images with their total number of descendant children | 100 |
| A.1 | A summary of repositories which are not included in this study | 124 |
| D.1 | Detailed numbers of unique vulnerabilities estimated for Official repository | 154 |
| D.2 | Detailed numbers of unique vulnerabilities estimated for Community repository | 154 |
| D.3 | Detailed numbers of unique vulnerabilities estimated for Verified repository | 155 |
| D.4 | Detailed numbers of unique vulnerabilities estimated for Certified repository | 155 |

Abbreviations

| | |
|----------------|--|
| API | Application Programming Interface |
| AWS | Amazon Web Services |
| CD | Continuous Delivery |
| CERT/CC | Computer Emergency Response Team Coordination Center |
| CI | Continuous Integration |
| CIRCL | Computer Incident Response Center Luxembourg |
| CLI | Command Line Interface |
| CNA | CVE Numbering Authority |
| CSV | Comma Separated Values |
| CVE | Common Vulnerabilities and Exposures |
| CVSS | Common Vulnerability Scoring System |
| CWE | Common Weakness Enumeration |
| DAZER | Docker imAge analyZER |
| DE | Docker Engine |
| ELSA | Enterprise Linux Security Advisory |
| FIRST | Forum of Incident Response and Security Teams |
| HTTP | Hypertext Transfer Protocol |
| IT | Information Technology |
| JSON | JavaScript Object Notation |
| MITM | Man-In-The-Middle attack |
| noSQL | not only Structured Query Language |
| NCSD | National Cyber Security Division |
| NCF | National Cyber security Federally funded research and development center |
| NIST | National Institute of Standards and Technology |
| NVD | National Vulnerability Database |

| | |
|-------------|---------------------------------|
| OS | Operating System |
| REST | REpresentational State Transfer |
| RHSA | Red Hat Security Advisory |
| SaaS | Software as a Service |
| SDK | Software Development Kit |
| SHA | Secure Hash Algorithm |
| SSD | Solid State Drive |
| ULN | Unbreakable Linux Network |
| UUID | Universally Unique Identifier |
| VM | Virtual Machine |
| VPN | Virtual Private Network |

Chapter 1

Introduction

Over the past few years, software containers have become a popular virtualization technology for running multiple isolated application services on a single host using minimal resources. As a consequence, containers have been easily integrated into Continuous Integration and Continuous Delivery (CI/CD) workflows, resulting into numerous DevOps tools and frameworks. The latter are increasingly utilized for application development and Information Technology (IT) operations, where speed and agility are two important factors for deployment processes, as Gartner predicts that more than 50% of global organizations will be running containerized applications in production by 2020, compared to today's 20% [1].

1.1 Motivation

Although there exists many container orchestration solutions, Docker has rapidly become the most widely used and recognized container technology over the years. Its online platform known as Docker Hub is the world's largest library for container images backed by a broad audience of users and a strong community [2]. At the time of this writing, there are approximately 2.1 million repositories available on Docker Hub with approximately 80 billion downloads since the platform's introduction [3]. On one hand, the platform hosts Official repositories maintained by Docker's own dedicated team [4]. On the other hand, Verified and Certified repositories are maintained by third-party vendors, while anyone may create a Community repository.

A commissioned study from 2016 conducted by Forrester Consulting on behalf of Red Hat reports that three-quarters of security-minded respondent claimed that their major concern about containers is security [5]. As the number of Docker images is growing, the latter have to continuously be maintained. As an example, a German-based IT investment and development company reported that Docker Hub hosted 17 malicious images that had been stored on platform for an entire year between May 2017 and May 2018 [6]. One of them was used to mine Monero, an open-source cryptocurrency, which rewarded the attackers with non-less than 544.74 Monero, approximating to 90,000 dollars.

Docker images are comprised of a series of layers and may be either base, parent or a child images (note that the terms parent and base image are sometimes used interchangeably). A base image

is an image which has no parent, typically containing basic tools and packages, while a child image depends on a single parent inheriting all of its layers [7].

Once new images are uploaded to Docker Hub, the latter are run through Docker's own security scanner, checking against well-known Common Vulnerability and Exposures (CVE) databases in order to map out images' vulnerabilities [8]. Nonetheless, many images are not updated or rebuilt for weeks, months, or even years.

In late 2018, Docker Inc. announced that both Docker Store and Docker Cloud were becoming a part of Docker Hub [9]. The result of that merging translated into the introduction of Certified and Verified repositories defined as followed:

- Certified repository: "Docker Certified technologies are built with best practices, tested and validated against the Docker Enterprise Edition platform and APIs, pass security requirements, and are collaboratively supported."
- Verified repository: "High-quality Docker content from verified publisher. These products are published and maintained directly by a commercial entity."

As the number of Docker images is increasing, the room for security improvement is also growing. The main goal of this thesis is to take a deep dive into Docker Hub's security landscape.

In [10], Gummaraju et al. studied how vulnerable Docker Hub images may represent a concrete security threat. They found that over 30% of the Official repositories hosted on the online platform contain images highly susceptible to a variety of security attacks such as Shellshock- or Heartbleed-based attacks, while about 40% of the Community repositories are in that case.

A similar research from 2016 created a Docker image vulnerability analysis framework named DIVA, which semi-automatically discovered, downloaded, and analyzed both Official and Community images on Docker Hub. They found that both types of images contained more than 180 vulnerabilities on average when considering all versions. The authors pointed out that many of the top vulnerable packages appeared in the most popular base images such as Ubuntu, Node or Debian, suggesting that the root cause of such a concerning security landscape may be due to a potentially small set of very influential base images [11].

Following those research, Docker Inc. introduced two main security measures in 2016: a dedicated security scanning service [8] and two new types of repositories referred to as Certified and Verified, meeting higher security requirements and best practices [9].

1.2 Problem statement

Based on previous research and the security mechanisms introduced by Docker Inc. in response to those investigations, this thesis addresses the following research questions:

1. Have the security measures introduced by Docker Inc. in response to previous research improved Docker Hub's security landscape and to what extent?
2. Are vulnerabilities still inherited from images' parent(s) and in what proportion?
3. How are discovered vulnerabilities distributed across repository types?

1.3 Thesis outline

The remaining part of this thesis is structured as followed. First, important concepts and technologies will be introduced in chapter 2. Chapter 3 will present the thesis' objectives, while describing the methodology used to solve the posed problem. Chapter 4, 5 and 6 will respectively describe the result of our model's design and implementation, as well as the measurements conducted with it. Chapter 7 will deeply analyze the data obtained and described in chapter 6, using common mathematical concepts and indicators. Chapter 8 will discuss important challenges encountered during the execution of the project and provide a critical analysis of the conducted study, as well as proposals for future work. Finally, a conclusion is presented in chapter 9, followed by a series of developed source code and obtained raw data in the appendix.

Chapter 2

Background and literature

This chapter introduces important concepts and technologies which will be used in later chapters such as software vulnerabilities and containers, Docker's architecture and internals comprising essentially of the Docker engine and Docker Hub, as well as the latter's current security landscape.

2.1 Software vulnerabilities

Software vulnerabilities have been an increasing problem with the growth of the Internet, which has greatly favoured their exploitation by malicious entities such as nation states or private attackers.

2.1.1 What is a software vulnerability?

According to the National Institute of Standards and Technology (NIST), a vulnerability consists of a "weakness in an information system, system security procedures, internal controls, or implementation that could be exploited or triggered by a threat source" [12]. A software vulnerability consists therefore of a software weakness which may be exploited by an attacker in order to perform an unauthorized action on a computer system.

Software vulnerabilities may be reported by anyone usually referred to as a *vulnerability reporter*, which identifies and informs a security incident team capable of contacting and reporting vulnerabilities to software vendors. Although there exists multiple private and public security incident teams, the Computer Emergency Response Team Coordination Center (CERT/CC) is usually the preferred entity for reporting vulnerabilities to vendors, due to its quasi-governmental profile, as well as its historical influence in the security field [13].

Computer security incident teams are responsible for verifying and confirming a reported vulnerability, before taking contact with the appropriate software vendor(s) and agree on a patching time window for the vendor to develop a *security patch*. At the end of that period of time, a patch is released by the software vendor in parallel with public advisories from the security incident team, providing technical information about the vulnerability and a unique number identifying it, as well as references to the released patch.

2.1.2 Enumerating vulnerabilities

Publicly disclosed vulnerabilities are uniquely identified through so called Common Vulnerabilities and Exposures (CVE) numbers, maintained by the National Cyber security Federally funded research and development center (NCF), operated by the Mitre Corporation and funded by the National Cyber Security Division (NCSA) of the United States department of Homeland Security. A CVE entry contains multiple fields such as a description of its vulnerability, its disclosure date or even references to available patches. CVE numbers follow a strict standard consisting of the CVE prefix, the year of their vulnerabilities' disclosure and a variable length series of arbitrary digits with a minimum length of four digits:

CVE-YYYY-XXXX[XXX...]

Through its CVE number, the disclosure year of a software vulnerability may therefore be identified easily. Although every disclosed vulnerability is uniquely identified with a corresponding CVE-ID, vulnerabilities which are reported by private companies such as Red Hat or Oracle may have their own identifier assigned by such companies. Indeed, Red Hat uses its own Red Hat Security Advisory (RHSA) numbers, whereas Oracle uses Enterprise Linux Security Advisory (ELSA) IDs. Note however that such vulnerability identifiers do not replace CVE numbers, as any publicly disclosed vulnerability is uniquely identified through a CVE-ID. Nonetheless, vulnerabilities disclosed by private companies may temporarily lack a CVE number following their disclosure, due to the amount of time necessary to obtain the identifier. In that case, a software vulnerability may only be identified through a RHSA or ELSA number of the following form:

RHSA-YYYY-XXXX[XXX...]

ELSA-YYYY-XXXX[XXX...]

It is important to note that although the format of RHSA and ELSA numbers is similar to the one used for CVE-IDs, the final digits located at the end of those numbers become different when an RHSA/ELSA number is assigned a CVE number, as those digits are completely arbitrary in both cases. Moreover, it should be noticed that other private companies may use personally assigned vulnerability identifiers, but only the ones from Red Hat and Oracle will be relevant for this thesis besides CVEs.

2.1.3 Classifying vulnerabilities

While CVE numbers are a common way of enumerating disclosed software vulnerabilities, Common Weakness Enumeration (CWE) is a software vulnerability categorization system sponsored by the NCF, operated by the Mitre Corporation and funded by the NCSA of the United States department of Homeland Security. CWE provides over 800 software weakness categories at the time of this writing, ranging from simple authorization concerns to pointer dereference weaknesses [14].

Similarly to software vulnerabilities identified through a CVE number, CWE categories are identified with a CWE number following a strict standard consisting of the CWE prefix as well as a series of three or four digits:

CWE-XXX[X]

CWE weakness categories are therefore recognized through their unique CWE numbers, which help classifying disclosed vulnerabilities identified with a CVE-ID. For example, the integer over-

flow vulnerability found in libssh2 before version 1.8.1 and identified with "CVE-2019-3855" is related to the Integer Overflow or Wraparound CWE category identified through CWE-190. Finally, while CVE and CWE numbers are a common way of enumerating and classifying disclosed software vulnerabilities, the latter need also to be assigned severity levels in order to better understand the impact of a vulnerability on a system or infrastructure.

2.1.4 Severity levels

Any publicly disclosed software vulnerability with an assigned CVE number may be assigned a severity level determined based on a Common Vulnerability Scoring System (CVSS) score. CVSS is an industry standard developed by the Forum of Incident Response and Security Teams (FIRST), a nonprofit corporation aiming at improving the way incident response teams react to security incidents [15]. That scoring system is used by many private companies and governmental organizations such as the US government repository of standards-based vulnerability management known as the National Vulnerability Database (NVD).

| Severity | Score range |
|----------|-------------|
| None | 0.0 |
| Low | 0.1-3.9 |
| Medium | 4.0-6.9 |
| High | 7.0-8.9 |
| Critical | 9.0-10.0 |

Table 2.1: NVD's Common Vulnerability Scoring System (CVSS) [16]

CVSS consists of a rating system assessing the severity of disclosed vulnerabilities depending on their ease and direct impact of exploitation. CVSS scores range from 0 to 10 and lead to five different levels of severity as shown in table 2.1 above. Atlassian Corporation, an influential Australian software company using CVSS actively, describes the different severity levels provided by the scoring system as followed [17]:

Severity Level: Critical

- Exploitation of the vulnerability likely results in root-level compromise of servers or infrastructure devices.
- Exploitation is usually straightforward, in the sense that the attacker does not need any special authentication credentials or knowledge about individual victims, and does not need to persuade a target user, for example via social engineering, into performing any special functions.

Severity Level: High

- The vulnerability is difficult to exploit.
- Exploitation could result in elevated privileges.
- Exploitation could result in a significant data loss or downtime.

Severity Level: Medium

- Vulnerabilities that require the attacker to manipulate individual victims via social engineering tactics.
- Denial of service vulnerabilities that are difficult to set up.
- Exploits that require an attacker to reside on the same local network as the victim.
- Vulnerabilities where exploitation provides only very limited access.
- Vulnerabilities that require user privileges for successful exploitation.

Severity Level: Low

Vulnerabilities in the low range typically have very little impact on an organization's business. Exploitation of such vulnerabilities usually requires local or physical system access.

Finally, note that understanding the CVSS scoring system and the different severity levels it provides is very important in order to understand the measurement and analysis chapters.

2.2 Software containers

Software containers have become a popular virtualization technology which goes all the way back to 1979 with the very first software process isolation attempt, through the 7th version of the Unix Operating System (OS) [18].

2.2.1 What is a software container?

A software container consists of a virtualization technology allowing to run multiple isolated application services on a single host using minimal resources. Usually referred to as simply *containers*, the latter are isolated through the use of three key components added to the Linux kernel since 1979 known as chroot, Linux namespaces and control groups (cgroups).

The *chroot* utility introduced in 1979 allows changing the root directory of a running process and all of its children. Although it is considered the very first step towards containerized technology, the *chroot* utility does not strictly provide process isolation, as a chrooted process is still able to access files and directories outside the specified root through the use of relative paths [18]. *Linux namespaces* introduced in the kernel in 2002 however, constitute a major step towards software isolation by allowing processes to be completely isolated from each other on different levels such as networking, disk access, process IDs or even user and group access [19]. *Cgroups* constitute the final component of any software container technology available today. Originally developed by Google and added to the Linux kernel in 2007, cgroups allow limiting the consumed resources by a certain process or group of processes, such as memory, CPU, disk or network usage [20].

A software container consists therefore of an application service isolated from other containers, through the use of cgroups, Linux namespaces and the *chroot* utility. Although there exists multiple software container orchestration solutions, they are all based on the combination of those three technologies provided by the Linux kernel. Moreover, although software containers go all the way back to the early 1980s, they only became popular in 2013 with the rise of Docker, which provides a simple container packaging solution, allowing developers and operators

to deploy their applications easily. Finally, note that Docker is discussed in detailed in the next section under [2.3](#).

2.2.2 Container vs. Virtual Machine (VM)?

Contrary to common misconceptions, software containers do not make VMs obsolete as they are simply used for different purposes.

First, containers aim at virtualizing a single or a few applications including as few dependencies in order to be lightweight and portable, while VMs virtualize a whole OS in view of running multiple applications, making them more heavyweight.

| Container | VM |
|-------------------------------|-------------------------------|
| Lightweight | Heavyweight |
| Native performance | Limited performance |
| Shared host kernel | Own virtualized kernel |
| Software-level virtualization | Hardware-level virtualization |
| Startup time in milliseconds | Startup time in minutes |
| Process-level isolation | Full isolation |

Table 2.2: Main differences between VMs and software containers

Secondly, containers share the underlying kernel of their host machine, providing native bare metal performances at runtime, as they may be started in a matter of seconds. VMs on the other hand virtualize a whole OS, which requires booting a complete kernel at runtime, creating a significant overhead compared to containers.

Thirdly, containers virtualize solely software applications whereas VMs virtualize both software, firmware and hardware such as disks, making them a lot more suitable for advanced operational purposes.

Finally, software containers only provide a process-level isolation, theoretically less secure than the full OS-level isolation provided by VMs.

2.3 Docker

As explained in [2.2.1](#), software containers have become a popular virtualization technology for running multiple isolated application services on a single host using minimal resources. Although there exists many container orchestration solutions, Docker has rapidly become the most widely used and recognized container technology over the years.

2.3.1 What is Docker?

Docker is a container orchestration solution allowing developers and other IT operators to create, deploy and manage standardized virtualization units referred to as "containers", packaging up code for a single application and all its required dependencies [21]. Originally closed source under the name dotCloud, Docker was released as an open source project in March 2013 and is primarily developed by the Docker Inc. company at the time of this writing [22]. Since then, its popularity has increased constantly within the IT industry, with a growth rate of 40% for the year 2017 only [23]. Docker consists therefore of a tool designed to create, ship and run containerized applications based on two central components: the **Docker engine** and the company's own Software as a Service (SaaS) sharing platform known as **Docker Hub** [24]. Note that the term "Docker" is often misused to only refer to the Docker engine or even the company developing the container orchestration solution. Throughout this thesis however, the term "Docker" will be used to strictly refer to the Docker technology, whereas "Docker Inc." will only refer to the company developing the container orchestration solution.

2.3.2 What is a Docker container?

As briefly mentioned in 2.3.1, a Docker container consists of a standardized virtualization unit, packaging up code for a single application and all its required dependencies [21]. The goal of Docker containers is to facilitate the modular development and deployment of software applications, by incorporating only the necessary packages and configuration files required by a containerized service (e.g. an Apache server). As a result, Docker containers tend to be very lightweight and easy to deploy or duplicate.

Docker distributes applications in the form of *images* built upon so called *Dockerfiles* [25]. The latter contain a set of directives specifying what an image should contain upon building, as shown in figure 2.1 below. For example, such directives may consist of specifying a parent image which can be used as a base for extension or specifying certain packages that need to be included in the image to be built.

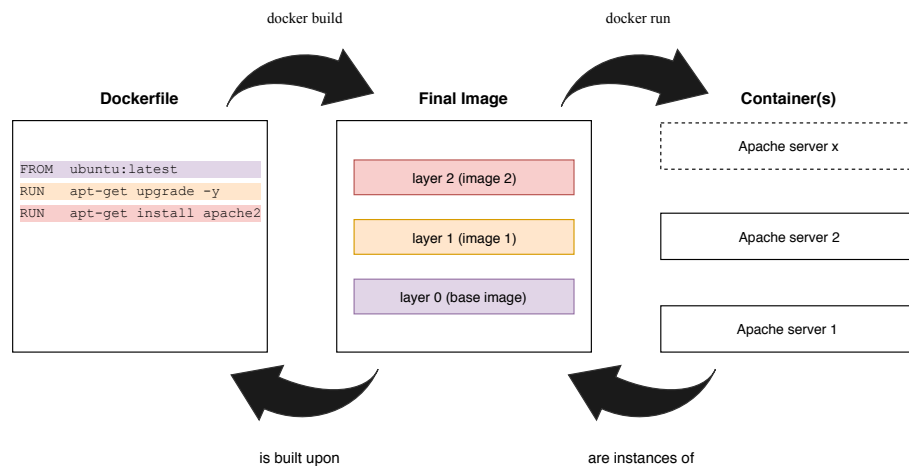


Figure 2.1: The container creation process

Once built into an image through Docker's internal containerization technology referred to as the **Docker engine**, each directive is built into an intermediate image called an *image layer* [26]. Once combined together, those layers form a *final Docker image* with a number of layers matching the directives located in its original Dockerfile. The goal behind image layers is to facilitate and optimize image rebuilds, as the Docker engine is able to reuse intermediate images, requiring only the rebuild of modified or added layers upon changes from a Dockerfile (more details in 2.4.2). A final Docker image consists therefore of an *immutable* read-only template containing instructions for creating a Docker container.

It is only once instantiated that an image results into the deployment of a Docker container, running a particular application service. Note that a single image may be instantiated multiple times, as one of Docker's goals is to make containers easy to deploy and duplicate.

2.3.3 How are Docker images distributed?

As explained in 2.3.2, a Docker container is only an instantiation of an image holding a containerized piece of software. Thus, the central part of the containerized software distribution is executed through the sharing of final Docker images, as they constitute single portable and immutable files easy to distribute. Anyone may build a Docker image from a Dockerfile and redistribute it as pleased. Nonetheless, stateless and highly scalable servers referred to as *Docker image registries* are a common way of storing and redistributing images to the masses [27].

Such registries simply hosting image repositories may be local and private (typically for enterprise environments) or global and public such as Docker Inc.'s official registry named *Docker Hub*. The latter is the world's largest library of Docker images at the time of this writing and contains both Official repositories with certified images from vendors such as Canonical, Oracle, Red Hat or Microsoft, as well as Community repositories containing images which may be uploaded by any user or organization [2]. Finally, note that the Docker Hub registry is discussed in more details in section 2.5.

2.3.4 Docker's architecture

Docker makes use of a client-server architecture composed of three entities consisting of a Docker client, a Docker host and a Docker image registry [26]. As shown in figure 2.2 below, the *client* consists of a simple interface provided to the user in order to execute Docker commands such as *docker build*, *docker pull* or *docker run* and be able to build, download and deploy Docker containers.

The actual execution of those tasks is however managed on the *Docker host*, which simply consists of a physical machine or VM with a running server known as the "Docker daemon", able to handle tasks requested by the client. The Docker daemon (often abbreviated "dockerd") consists therefore of the core component of Docker's architecture, as it translates user requests such as *docker pull* or *docker run* commands into the concrete download of images or deployment of Docker containers.

Although the Docker host usually holds both the Docker daemon and client, the latter may also be run from a remote machine and access the daemon via a network, as the communication between the two components is established through a REpresentational State Transfer (REST) Application Programming Interface (API), as shown in figure 2.2.

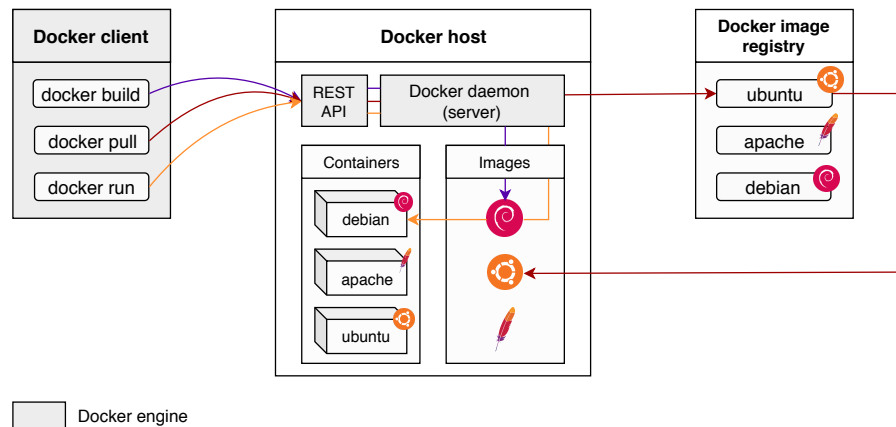


Figure 2.2: Docker's architecture

Besides handling the building of images and their deployment through containers, the Docker daemon is also responsible for interacting with a so called Docker image registry, allowing users to share Docker images. As briefly mentioned in 2.3.3, image registries consist of public or private SaaS platforms, hosting pre-built images uploaded by users, in view of being shared with other peers. Thus, a *docker pull* command initiated by a user through the Docker client in view of downloading a certain image, would therefore be handled by the Docker daemon, which would retrieve the image from its configured registry in order to make it available locally for the user.

Finally, it is important to note that the Docker client, the REST API and the Docker daemon are all parts of the so called **Docker engine**, consisting of one of Docker's two central components, discussed in details in the next section.

2.4 The Docker engine

In combination with Docker Inc.'s own image sharing platform known as Docker Hub, the Docker engine constitutes an essential part of the Docker container orchestration solution.

2.4.1 What is the Docker engine?

The Docker engine is a client-server application composed of three major components, as illustrated in figure 2.3 below [26].

First, the engine's most abstract level consists of the *Docker client*, also known as the Docker Command Line Interface (CLI) utility. The latter consists of a simple Hypertext Transfer Protocol (HTTP) client, implementing an easy way for end users to interact with the rest of the engine using simple commands such as *docker pull* or *docker run*. As shown in figure 2.2 above, the Docker client is therefore responsible for translating commands requested by the user into HTTP requests destined to the engine's server through its API.

Secondly, the Docker Engine's API is a *REST API* which may be accessed by any HTTP client. Indeed, the Docker client only consists of the default interface towards the engine, but any

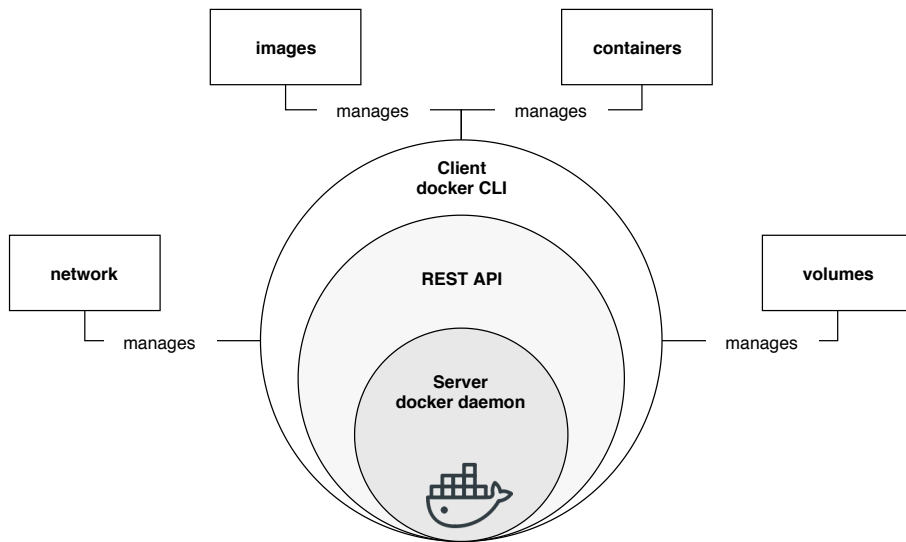


Figure 2.3: The Docker engine

HTTP client or programming language with an HTTP library is able to interact with the API . Moreover, an official Go and Python Software Development Kit (SDK) have been made available by Docker Inc. for an easy programming interaction with the Docker engine's REST API [28].

Finally, the engine's core component consists of the *Docker daemon* which functions as a server interacting with its host OS to build, run and deploy containers using Docker components and services. Consequently, the daemon server listens for HTTP requests coming through its REST API, in order to execute a user requested task such as the download of an image or the deployment of the latter in the form of a container.

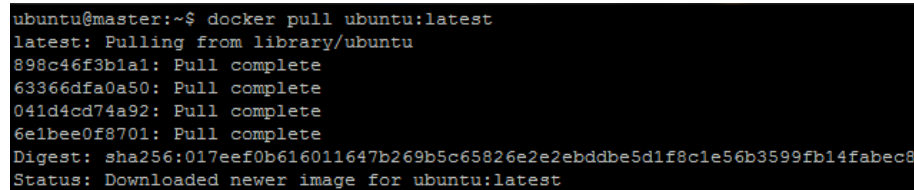
The Docker engine is therefore the core of the Docker technology, as it allows handling everything related to the management of Docker containers and the images they are based on, as well as the volumes and networks they may use. Nonetheless, note that the automated sharing of images involves Docker's other main component known as Docker Hub, which is discussed in details in 2.5. Additionally, it is important to note that the Docker engine literature tend to refer to Docker images and containers as "Docker objects", although only the specific terms will be used throughout this thesis for better clarity [26].

2.4.2 Managing images

As explained in 2.3.2 and 2.3.3, Docker images constitute the central piece of containerized software virtualization in Docker. Although the Docker engine tends to blurry the perception of Docker image management due to its accessible and intuitive CLI utility, it is important to understand how they are operated behind the curtain. Whenever a Dockerfile is built into a final image, all of its intermediate images become referred to as "blobs" or "layers". Each layer contains the files and directories created based on its corresponding directive in the image's Dockerfile and is located under `/var/lib/docker/image/overlay2/` on a Unix-based Docker host (more details about Docker hosts in 2.3.4).

The order of an image's layers as well as the rest of its metadata are contained in a JavaScript Object Notation (JSON) configuration file referred to as the image's manifest file. The latter identifies the layers composing the image using a unique ID number. Prior to Docker version 1.10 introduced in February 2016, images were composed of a single layer with an ID consisting of a randomly generated Universally Unique Identifier (UUID) of a length of 256 bits [29]. In order to retrieve a complete image, manifest files prior to Docker version 1.10 used a parent attribute containing the UUID of the next layer to be retrieved. This way, complete images composed of multiple layers in practise could be created and retrieved from registries.

Since Docker version 1.10 however, images' layers are now identified through a hash of their compressed content using the 256-bit version of the Secure Hash Algorithm (SHA) [30]. That fundamental change has greatly improved security, making layer content directly addressable through a unique SHA-256 digest. Manifest files have now removed the parent attribute or left it completely empty to avoid breaking earlier specifications. Thus, all the layers composing an image are now indexed in a single manifest file using their SHA-256 digests, which identifies their content directly. It is important to note however that this new specification has made the identification of an image's parent a lot more challenging, as all parental references have been eradicated from images' manifests.



```
ubuntu@master:~$ docker pull ubuntu:latest
latest: Pulling from library/ubuntu
898c46f3b1a1: Pull complete
63366dfa0a50: Pull complete
041d4cd74a92: Pull complete
6e1bee0f8701: Pull complete
Digest: sha256:017eef0b616011647b269b5c65826e2e2ebddbe5d1f8c1e56b3599fb14fabec8
Status: Downloaded newer image for ubuntu:latest
```

Figure 2.4: The Docker engine's use of short layer IDs and long image digests

Regarding single Docker images as a whole, the latter are identified using a SHA-256 digest of their manifest file. Contrary to layer IDs, image digests are always use in their entirety by the Docker engine [31]. Indeed, layer IDs are commonly shorten within the Docker engine using only the first 12 characters, as a attempt to improve human interaction with the engine. For example, whenever an image is retrieved from a registry (referred to as "pulling"), the short ID of each layer composing the image is displayed to the user, while the entire image digest is shown as illustrated in figure 2.4. Finally, it is important to note that the use of short layer IDs and the removal of an easy image parent identification method from the Docker engine's API are important details, which will play a major role in the methodology and results chapters of this thesis.

2.5 Docker Hub

In combination with the Docker engine, Docker Hub constitutes an essential part of the Docker container orchestration solution.

2.5.1 What is Docker Hub?

Docker Hub is Docker's default image registry, consisting of the largest public library of Docker images at the time of this writing [2]. As briefly mentioned in 2.3.3, *container image registries*

are private or public stateless and highly scalable servers used to store and distribute images to the masses [27]. Within a registry, images are organized into so called *repositories*, which may be visualized as folders holding images and maintained by different users.

At the time of this writing, Docker Hub hosts over 2.1 million repositories with approximately 80 billion downloads since the platform's introduction in 2013 [3]. Each Docker Hub repository contains a series of images for different versions of the same piece of containerized software. Furthermore, all the images contained in a repository need to include a so called *image tag*, identifying a specific image from another one. For example, a repository named `<example-repository>` for the containerized software `<example-software>` may contain images tagged as followed:

- `<example-repository>/<example-software>:1.0.0`
- `<example-repository>/<example-software>:v0.9.4-server`
- `<example-repository>/<example-software>:2.00.035.00.20190115.1`

It is important to note that although some repositories may contain a large number of tags, it does not mean that their number of images is equally large. Indeed, a single image may have an unlimited number of tags, whereas all the images contained within a same repository must be unique. Thus, duplicate images with the exact same layers are not allowed to coexist within a repository to avoid image impersonation and unnecessary redundancy. Furthermore, it should be noted that the standard required for image tags is very loose, as they may consist of any combination of lowercase and uppercase letters, digits, underscores, periods and dashes, with a maximum of 128 characters [32]. Finally, note that the Docker literature tends to use the terms "images" and "repositories" interchangeably, although the former are objects contained in the latter and are significantly more numerous than the number of repositories available on Docker Hub.

2.5.2 Repository types

As explained in 2.5.1, images on Docker Hub are organized into repositories managed by the platform's users. At the time of this writing, there exists four different types of repositories fulfilling different best practices and security requirements.

First, *Official* repositories consist of a curated set of Docker repositories, aiming at providing base OS and drop-in solutions for popular programming language runtimes, data stores, and other services, while exemplifying Dockerfile best practices and ensure that security updates are applied in a timely manner [4]. Due to their large popularity, Official repositories are maintained by a dedicated team sponsored by Docker Inc., who is responsible for reviewing and publishing all content in the Official images. As of April 5th 2019, Official repositories constitute the most popular type of repositories with the largest number of downloads ranging from 50 000 to over 10 million pulls for the most popular ones [33]. Surprisingly however, the total number of Official repositories is minimal, as it only represents 0.007 % of the global amount of available repositories on Docker Hub, as illustrated in table 2.3.

Secondly, *Community* repositories contain images which may be uploaded by any user or organization [34]. Although their popularity varies from a couple of downloads to over 10 million pulls, Community repositories are by far the most numerous type of repository on Docker Hub, representing more than 99 % of the available repositories on the platform, as shown in table 2.3 below. Contrary to their three other peers, Community repositories do not need to fulfill any

| Repository type | Total | Share (in %) |
|-----------------|-----------|--------------|
| Official | 151 | 0.007 |
| Certified | 44 | 0.002 |
| Verified | 252 | 0.012 |
| Community | 2,143,462 | 99.982 |
| All | 2,143,865 | 100 |

Table 2.3: Docker Hub’s repository type distribution as of April 5th 2019

special requirements, allowing anyone with a valid email address to open a Docker Hub account, start initiating a Community repository and publish custom made images available for all users.

Thirdly, *Verified* repositories were introduced to Docker Hub in December 2018 as a result of the merging of multiple Docker image registry platforms and in an attempt to make Docker Hub more secure [9]. Indeed, Docker Inc. used to offer a separate platform for third party enterprise vendors known as the "Docker store", as well as a hosted registry service dedicated to help users connecting Docker to their existing cloud providers known as the "Docker Cloud". Since the end of last year, the three services have been merged into a single place, making Docker Hub the one and only Docker image registry and cloud service operated by Docker Inc.

| Repository type | Total | Paid | Share (in %) |
|-----------------|-----------|------|--------------|
| Verified | 252 | 26 | 10.317 |
| Certified | 44 | 11 | 25 |
| All | 2,143,865 | 37 | 1.726 |

Table 2.4: Docker Hub’s paid repository distribution as of April 5th 2019

As a result, Docker Hub now offers Verified repositories provided by third-party software vendors such as Oracle, IBM or Microsoft. Similarly to their Official peer, Verified repositories are vetted by Docker Inc. before their introduction on the platform. Their maintenance and the publication of their images is however left entirely to the commercial entities. Moreover, the latter are allowed to provide paid content via a subscription model, making some Verified repositories paid only. As illustrated in table 2.4 however, the number of Verified repositories requiring a payment as of April 5th 2019 is very limited, as it only consists of about 10 % of the total number of Verified repositories and less than 2 % of the global amount on Docker Hub.

Fourthly, *Certified* repositories consist of a very small subset of Verified repositories, meeting additional quality, best practise and security requirements established by Docker Inc. [9]. Making up about 17 % of the Verified type, Certified repositories also contain a minimal amount of paid repositories, with 25 % of them requiring a payment [table 2.4]. Thus, Certified images are supposed to be the most stable and secure images available on the Docker Hub platform today.

Finally, note that the statistics available in table 2.3 and 2.4 are obtained directly from the official Docker Hub’s website as of April 5th 2019, using the repository filters available through the platform’s Web interface.

2.5.3 Repository naming convention

As discussed in 2.5.1, each Docker Hub repository is administrated by a user and contains a series of images with different versions of the same piece of containerized software. Repositories use therefore a naming convention based heavily on the username of their creator, as well as the name of the software contained in the images they hold. The naming convention for all types of repositories is therefore of the following form:

<namespace>/<containerized software>

| Repository type | Namespace | Example |
|--------------------|------------------|------------------------|
| Official | library | library/ubuntu |
| Community | <username> | doct15/mysql |
| Certified/Verified | store/<username> | store/ibmcorp/db2wh_ce |

Table 2.5: Docker Hub's namespaces per repository type

As shown in table 2.5, a repository's *namespace* is entirely dependent on its type. Indeed, Official repositories are contained under the *library* namespace, whereas Community repositories simply use the *username* of their creator. In a similar way, Certified and Verified repositories also make use of their creator's *username*, preceded by the *store* string.

Note that the library namespace is completely optional for Official repositories, which may be identified as either *library/<software-name>* or directly *<software-name>* within Docker. Furthermore, it should be noted that Verified repositories owned by the Microsoft publisher use a completely dedicated naming convention, consisting of the *mcr.microsoft.com* namespace, followed by an optional repository name and a required containerized software name such as:

- *mcr.microsoft.com/mssql-tools*
- *mcr.microsoft.com/cntk/release*
- *mcr.microsoft.com/dotnet/framework/aspnet*

The Microsoft publisher makes therefore heavily use of repositories of repositories, allowing the latter to hold either images or other repositories (not both), which may lead to very long image names such as the *mcr.microsoft.com/dotnet/framework/aspnet*. In that particular example, the image for the containerized *aspnet* software is located under the *framework* repository, which in its turn is located under the *dotnet* repository present under the *mcr.microsoft.com* namespace.

Finally, note that an image is identified within Docker through the use of a specific tag. As briefly mentioned in 2.5.1, the Docker literature tends to use the terms "images" and "repositories" interchangeably due to the misconception that an image related to a certain piece of software may be identified through its repository name only. Nonetheless, an image is identified through and only through the use of a tag, as a repository name such as *library/ubuntu* only identifies a certain containerized piece of software, but not a specific version. Docker images identify however precised version of a containerized piece of software, such as *library/ubuntu:bionic* or *library/ubuntu:xenial* in the case of Ubuntu. Moreover, it is important to note that all repositories make use of a default tag referred to as "latest", which does not necessarily identify the last updated image in the repository, but is used by default when an image is pulled without specifying a tag.

2.5.4 Docker image reusability

An extremely common practise in the Docker world is to base a new image on a so called *parent image* containing basic files and libraries. As illustrated in figure 2.1, a parent image is always specified as the very first line of a Dockerfile in the form of *FROM* *<parent-image-name>* directive, which downloads the parent image from Docker Hub upon building of the new image [35]. Thus, the rest of a Dockerfile's declarations simply consist of modifying the parent image (e.g. adding packages or directories), in order to create into a brand new one once the building process is completed.

It is important to note that the vast majority of Docker images are based on a parent image, whether the latter comes from an Official, Community or Verified repository on Docker Hub [7]. Nonetheless, using a parent image is in no way a requirement, as many of the popular images used as parents such as Ubuntu or Debian are not based on anything. Such images are commonly referred to as *base images* built from a Dockerfile containing no *FROM* directive or starting with a *FROM scratch* declaration in order to signify their total independence.

Finally, the type of image allowed to be used as a parent depends on an image's type. Indeed, Official images are only allowed to be based on images of the same type, while Community images may be based on any type. Similarly to their Official peers, Certified and Verified images are solely allowed to use images of the same type as their parents, as well as Official images [7]. Note that since Certified images are a sub-type of Verified images, they may therefore be based on either a Certified or a Verified image, as well as an Official image.

2.5.5 Docker image dependencies

The possibility of extending a parent image into a brand new one greatly facilitates the creation of new images for Docker users. However, that reusability creates a certain chain of dependencies between images, raising a certain number of security concerns when it comes to vulnerability isolation and inheritance.

For example, the Official Docker image for the Tomcat server (version 9.0-jre8 at the time of this writing) is based on nothing less than three parent images, as indicated in figure 2.5 below. Indeed, that image is directly based on the official openjdk:8-jre image, which in its turn is based on the Official buildpack-deps:stretch-curl image. Finally, the latter is based on the Official debian:stretch image, which is a base image and therefore not based on anything else. Nonetheless, the effective security of the official Tomcat image implies that patches are applied to vulnerable images upstream, as any non-patched vulnerability in one of the parents makes the children vulnerable.

Indeed, note that a child image only has one *direct parent* and may have multiple *indirect parents*. A child image consists therefore of a simple extension of its parent, leading to the inheritance of all the latter's layers, as shown in figure 2.5. Consequently, that dependency chain leads child images to also inherit all the vulnerabilities from their parent(s).

Dependency management is a recurrent security problem not only limited to Docker containers, which has been largely studied in the computer science literature.

In [36], Lauinger et al. analyzed the challenge of maintaining JavaScript library dependencies up to date and found that there is a strong need for better dependency management, as 37% of the analyzed websites in 2018 included at least one dependent library with a known vulnerability.

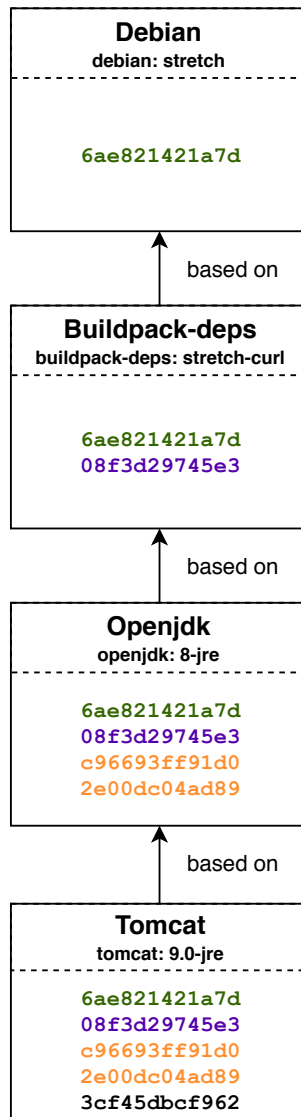


Figure 2.5: Dependencies of the Official Tomcat image on Docker Hub

In [37], Gaikovina Kula et al. examined the impact of library dependencies in GitHub in 2017, covering over 4,600 GitHub software projects and 2,700 library dependencies.

They discovered that many repositories rely heavily on dependencies, but 81.5% of them keep using outdated ones.

Dependency management is therefore an extensive problem within IT, which is essentially related to the image dependency mechanism in the case of Docker containers.

2.5.6 Have you said API?

Surprisingly, the Docker Hub platform does not have any official API at the time of this writing. Although there exists a documentation for the Docker registry HTTP API, the latter only applies to private registries but does not mention anything about Docker Hub's entry points [38]. Nevertheless, specific HTTP requests greatly differing from the original registry API seem to be valid towards the online platform. Thus, it is theoretically possible to make use of Docker Hub's unofficial and undocumented REST API, by filtering out valid requests using an automated trial and error approach.

2.6 Docker Hub's security landscape

Docker Hub's security landscape constitutes the core of this thesis and is therefore an essential part of this chapter.

2.6.1 Current knowledge

Due to the rather new aspect of Docker's popularity and its rapid development, studies about Docker Hub's security landscape are limited but highly concerning.

In [39], Lin et al. demonstrated the poor security of Linux containers, which Docker containers are an extension of. Their analytical study shows that containers are generally not very resistant to internal exploitation, as 56.82% of the used exploits during their experiments could successfully launch attacks from inside a container with a default configuration.

In [10], Gummaraju et al. studied how vulnerable Docker Hub images may represent a concrete security threat. They found that over 30% of the official repositories hosted on the online platform contain images highly susceptible to a variety of security attacks such as Shellshock-, Heartbleed- or Poodle-based attacks, while about 40% of the community repositories are in that case. Moreover, the empirical study revealed that 74% of all the images created in 2015 contained relatively easy to exploit vulnerabilities such as Shellshock or Heartbleed.

In [11], Shu et al. made four main discoveries in their analytical study examining the state of security vulnerabilities in Docker Hub images as of 2016 (date of the executed experiment). First, they found that both Official and Community images contain more than 180 vulnerabilities on average when considering all versions and that more than 80% of both types of images contain at least one highly severe vulnerability. Secondly, the study shows that many images or not updated frequently, as about 50% of both Community and Official images had not been updated in 200 days, while about 30% of them had not been updated in 400 days. Thirdly, Shu et al. discovered that vulnerabilities commonly propagate from parent to child images, as the latter inherit 80 vulnerabilities from their parents on average, while child images typically add about 20 more new vulnerabilities to their extended parents. Finally, the analytical study points out that many of the top vulnerable packages appear in the most popular base images such as Ubuntu, Node or Debian images, suggesting that the root cause of such a severe security landscape may be due to a potentially small set of very influential base images.

In [40], Zerouali et al. analyzed the relationship between outdated containers and their vulnerable/buggy OS packages, by examining 7,380 Official and Community Docker images based on Debian in October 2018. They found that the number of outdated OS packages is highly

correlated to the number of vulnerabilities found in a container. Furthermore, the conducted study shows that no image is devoided of vulnerable or buggy OS packages, confirming therefore the claims of Shu et al. in their analytical study.

Based on the available literature and recent studies, Docker Hub's security landscape seems very concerning at the time of this writing, as many images contain an alarming amount of vulnerabilities with a high propagation rate from parent to child images. It is however important to note that the last comprehensive study conducted around this subject is dated from April 2016, which is almost three years old at the time of this writing. The security landscape of Docker Hub may therefore have changed since the above studies were conducted, as the Docker world is evolving extremely rapidly.

2.6.2 Docker Inc.'s response

In response to Docker Hub's alarming security landscape pointed out by multiple research discussed in 2.6.1, Docker Inc. has introduced two main measures to the platform in an attempt to make Docker Hub more secure [9].

Docker Security Scanning

Docker Security Scanning is a vulnerability scanning service introduced to Docker Hub in May 2016 [41]. Available for both Community and Official repositories at the time of its release, the service provides a detailed security profile of a Docker image, by automatically analyzing and detecting vulnerable software and dependencies at its layer level. Since March 31st 2018 however, the service has been only made available for Official and Certified repositories, leaving Community and Verified repositories uncovered [42].

Moreover, Docker Security Scanning is a type of service which only scans a Docker image on upload, meaning that once the image has been updated it is never analyzed again for vulnerabilities. Indeed, it is true that a Docker image is immutable and therefore cannot be changed. However, contained packages which are not vulnerable at the time of their upload do not mean that they will not contain a discovered vulnerability later on.

Nonetheless, the Docker Security Scanning service constituted Docker Inc.'s first attempt to improve the platform's security, by integrating the service directly into Docker Hub's Web interface for Official repositories, while only sharing results with the appropriate vendors for security reasons, when it comes to Certified repositories .

Certified & Verified repositories

In December 2018, Docker Inc. announced the merging of multiple Docker image registry platforms to Docker Hub, resulting into the introduction of two new types of repositories to the platform, defined as followed by the company [9]:

- Certified repository: "Docker Certified technologies are built with best practices, tested and validated against the Docker Enterprise Edition platform and APIs, pass security requirements, and are collaboratively supported."
- Verified repository: "High-quality Docker content from verified publisher. These products are published and maintained directly by a commercial entity."

Certified and Verified repositories provide therefore high quality content maintained by third-party software vendors such as Oracle, IBM or Microsoft. Furthermore, Certified repositories consist of a very small subset of Verified repositories, meeting additional quality, best practise and security requirements established by Docker Inc. The introduction of Certified and Verified repositories to Docker Hub consists therefore of a second attempt in making the online platform more secure, by providing images of higher quality vetted by Docker Inc. before their introduction on the platform.

Finally, note that Verified and Certified repositories have technically existed since 2016 through Docker Inc's enterprise image registry for third part vendors known as the "Docker Store", which was merged into Docker Hub in December 2018 (more information about repository types in [2.5.2](#).

Chapter 3

Methodology

This chapter describes the methodology adopted in order to answer the thesis' problem statement formulated in the form of three central research questions:

1. **Have the security measures introduced by Docker Inc. in response to previous research improved Docker Hub's security landscape and to what extent?**
2. **Are vulnerabilities still inherited from images' parent(s) and in what proportion?**
3. **How are discovered vulnerabilities distributed across repository types?**

Note that the above problem statement implies many more research questions which are discussed in details in section [3.4](#).

3.1 Objectives

This research aims at answering the problem statement introduced in [1.2](#) using a methodology relying on three main phases. First, the design of a suitable software to gather metadata, vulnerability and parental information from a set of defined images on Docker Hub will be developed. Additionally, detailed research questions implied by the problem statement will be identified. Secondly, the designed software will be implemented using selected tools and technologies, while the research questions identified in phase 1 will be translated into noSQL queries for further use during the last phase of our methodology. Finally, the latter will consist of conducting experiments using the implemented software in phase 2 and provide a detailed analysis of the gathered data using the research queries developed previously, in view of answering the original problem statement.

The thesis' objectives may therefore be illustrated as shown in figure [3.1](#) and are detailed as followed:

1. Design
 - (a) Define the sets of Docker images involved in the research

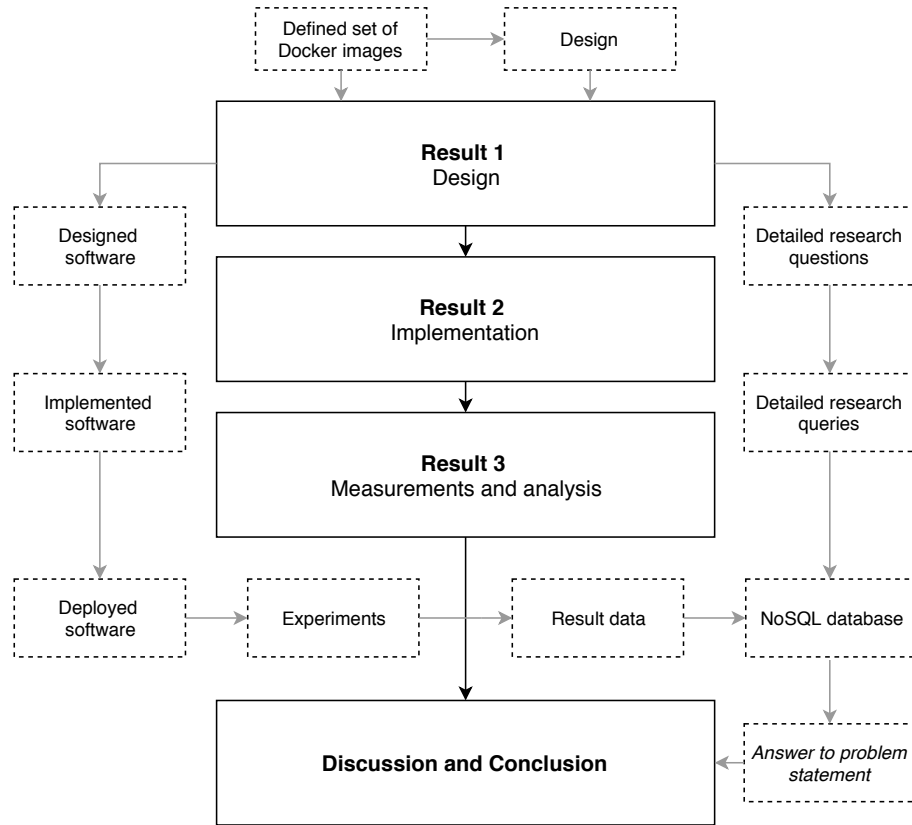


Figure 3.1: The thesis' methodology

- (b) Identify preliminary requirements needed to conduct the research
- (c) Create a software to gather metadata, vulnerability and parental information for each set of Docker images involved in the research
- (d) Design the format of the gathered metadata and vulnerability information for an image
- (e) Identify detailed research questions implied by the problem statement

2. Implementation

- (a) Identify tools and technologies required for creating and running the software designed in the previous phase
- (b) Implement the software designed in phase 1 to gather metadata, vulnerability and parental information for each set of Docker images involved in the research
- (c) Build the environment required to run the implemented tool
- (d) Translate the detailed research questions identified in phase 1 into noSQL research queries

3. Measurements and analysis

- (a) Conduct an experiment for each set of Docker images defined in phase 1
- (b) Import the gathered metadata, vulnerability and parental information of each set into a noSQL database for analysis
- (c) Make use of the noSQL queries implemented in phase 2 in order to answer the problem statement

Note that contrary to similar research studying Docker Hub’s security landscape, this research not only intends to answer the problem statement discussed in 1.2, but it also aims at providing an appropriate solution to make this study reproducible by other researchers, while providing a basic tool to conduct similar future studies.

3.2 Design

The design phase of the methodology aims at defining the study’s scope in terms of concrete images and repositories that should be taken into consideration for this study, as well as defining detailed research questions and design a suitable software to conduct experiments and gather metadata, vulnerability and parental information for the defined data set of Docker images, in view of answering the defined research questions.

3.2.1 Data set definition

As previously mentioned in the background chapter, the Docker Hub platform contains over two million images in total at the time of this writing [3]. A limited set of repositories and images needs therefore to be defined in order to conduct our study.

First, it is important to determine the subset of repositories that should be part of the study. Since one of the main goals of this work is to determine whether the security measures introduced by Docker Inc. in the form of an image security scanner and two new types of repositories with higher security requirements have modified Docker Hub’s security landscape, it has been agreed that all types of repositories should be analyzed, especially the ones introduced as an attempt to increase security on Docker Hub. As discussed in details in 2.5.2, Community repositories make up more than 99.98% of all repositories on Docker Hub, leaving Official, Certified and Verified images with a very minimal amount of repositories. Thus, all the repositories of those three types will be a part of the studied data set, whereas only a fixed length subset of randomly selected Community repositories among the most popular ones will be taken into account (e.g. 500 repositories).

Secondly, the set of images within each analyzed repository needs to be strictly defined, as many repositories contain up to several hundred images. Since this research aims at conducting an analysis study of Docker Hub’s security landscape in view of answering the detailed research questions discussed in 3.2.5, it has been chosen to only analyze the most recent image in each repository, as it represents their potentially least vulnerable image due to its most up-to-date property. Indeed, a repository’s most up-to-date image is the one that should include most of the patches available for the vulnerable packages it contains. Thus, only the most recent image in each of the considered repositories will be part of the study’s data set. Note that a repository’s

most recent image may consist of a brand new image or a previously uploaded image which has been updated.

Finally, the scope of this study will limit the chosen data set to images compiled for the x86-64-bit architecture and dedicated to the Linux platform only. Thus, Docker images compiled for the Windows platform or other processor architectures will not be taken into account, as their number in Docker Hub's ocean is very minimal [33].

3.2.2 Preliminary requirements

The design of the proposed software described in 3.2.3 in view of conducting experiments requires the preliminary population of a dedicated parent database to retrieve an image's parents, as well as the manual checkout of certain types of images to allow their download.

A parent database

As previously discussed in the background chapter, one of the main research questions of this thesis is to address whether vulnerabilities are still largely inherited from images' parent(s), as demonstrated in similar research prior to Docker's Inc.'s introduction of new security measures 2.6.1. Thus, it is essential to be able to determine the parental relationship between images in order to successfully analyze the evolution of vulnerability inheritance between images in Docker Hub. As explained in details in the background chapter, retrieving an image's parent is not as easy as it used to be since the changes introduced in the version 1.10 of the Docker engine 2.4.2.

Indeed, images now consist of a collection of tarball files identified through layer IDs, where each ID corresponds to a SHA256 hash of the layer's content. Whenever an image is based on another image, it inherits all the layers from its parent as it really consists of an extension of the latter. We propose therefore to take advantage of that layer inheritance mechanism in order to retrieve an image's parent. However, this methodology implies that a parent database containing a layer signature uniquely identifying all the images present in all the repositories susceptible to be used as parents is created prior to the conduction of each experiment. As explained in 2.5.4, Official images which are not base ones are indeed extending another Official image, whereas Certified and Verified images may be based on Certified, Verified or Official images. As for Community images, they may simply be based on any type of image.

Having a parent database containing a unique layer signature for all the images comprised in all the Official, Certified and Verified repositories will therefore allow determining whether an image has a parent, by simply looking for the upper layer combination of an input image into the database. This way, the latter should be able to return the name and tag of an image's parent if any, as illustrated in figure 3.2 below. Note that the designed format of the parent database planned on being used consists of a single file using the JavaScript Object Notation (JSON) format and will not consist of a strictly speaking database able to handle queries. However, the JSON format allows finding data in a way that should be efficient enough for the amount of data that the parent database will contain.

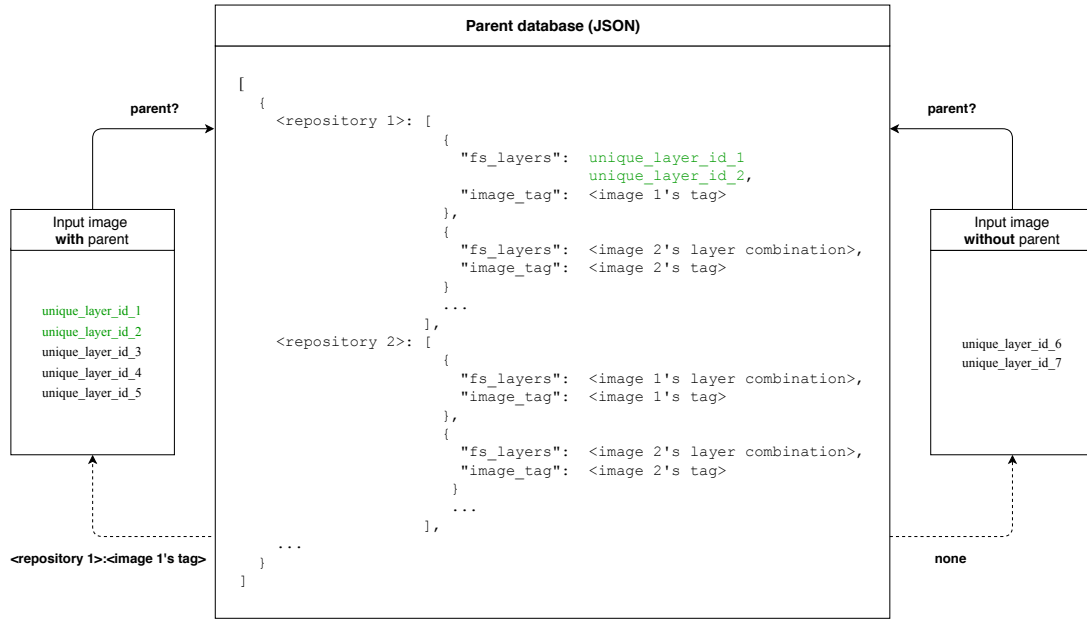


Figure 3.2: The planned parent database's design

Manual image checkout

As already discussed in the background chapter, the two new types of repositories introduced by Docker Inc. in an attempt to increase its public registry's security landscape are provided by vetted third-party software vendors who often offer proprietary and/or paid content. As a consequence, many providers require a valid Docker Hub account as well the manual checkout of their Certified/Verified repositories providing some contact information in order to be able to download the images they hold. The conduction of experiments to gather metadata, vulnerability and parental information for the set of Docker images defined in 3.2.1 using the designed software discussed in the next subsection will therefore require a preliminary manual checkout of those repositories through the platform's website. Note that Certified and Verified repositories which require a payment will not be taken into account in this research, as their number is extremely minimal.

3.2.3 Overview

Once the preliminary requirements and the data set of interest are defined, a suitable software to gather metadata, vulnerability and parental information for the defined data set of Docker images is needed. As shown in figure 3.3 below, the designed software interacts with three main components.

First, a list with the most recent image in all the repositories of a specific type or a subset of them (e.g. Official repositories) will be retrieved from Docker Hub. Secondly, the software should be able to interact with the Docker engine running on the same machine in order to download all the images in the retrieved list of most recent images and gather all of their metadata and parental information. Finally, all the downloaded images will be analyzed for vulnerability identification

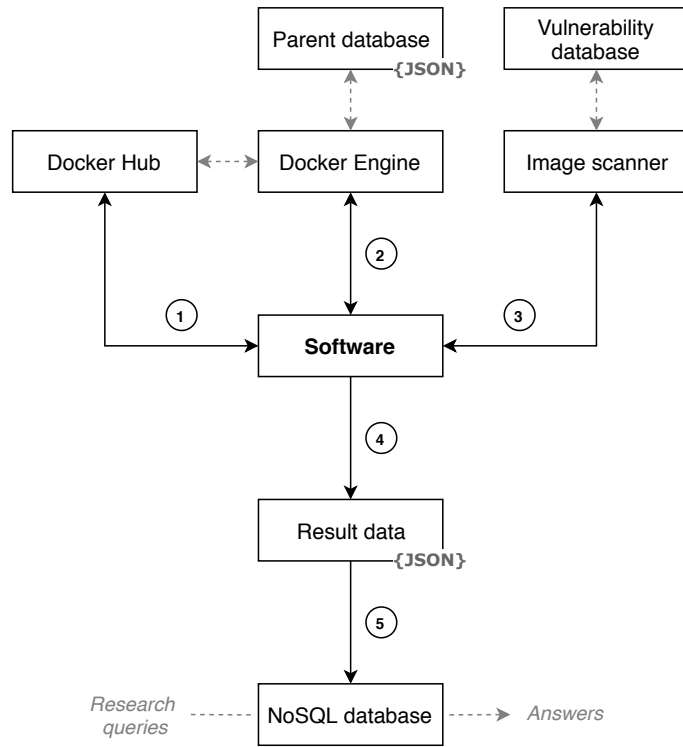


Figure 3.3: The planned design's overview

using an appropriate Docker image scanner, as well as a suitable vulnerability database (more details about the chosen tools and technology in 3.3.1). The result of an experiment using this software will consist of a list of structured objects containing all the gathered metadata, vulnerability and parental information of each of the analyzed images into a JSON formatted file discussed in details in the next subsection. The result data will then be able to be imported into a noSQL database for further analysis using what will be referred to as "research queries" discussed in the 3.3 implementation section of this chapter.

3.2.4 Result data format definition

As previously mentioned at the end of the design subsection in 3.2.3, it is planned to import the result data structured in the JSON format into a noSQL database for further analysis. Due to the large number of metadata, vulnerability and parental information gathered about each image composing the defined data set of Docker images to be studied, it is necessary to select, design and formally define the structured format of those collected data.

The latter will actually be stored in two separate JSON files, each containing a list of structured objects holding the gathered data. Indeed, the first file referred to as the "analysis file" will contain all the collected metadata and parental information of each of the analyzed images, whereas the second file referred to as the "vulnerability file" will contain all the gathered vulnerability information for each of the discovered vulnerabilities during an experiment.

Analysis results' format definition

As illustrated in the below 3.1 listing, each JSON object representing an analyzed image in the analysis file will contain a total of 9 attributes structuring the collected metadata and parental information about the image. Besides basic metadata information such as the image's name, tag or type, the rest of the attributes have been chosen based on the amount of valuable information they potentially provide. For example, the number of times an image has been downloaded (referred to as "pulling") may provide valuable information to see whether there exists a correlation between an image's number of vulnerabilities and number of pulls. The same approach may also be applied to an image's last updated timestamp in order to see whether it is correlated to the image's total number of vulnerabilities. The retrieved parental information is planned on consisting only of a single string referring to the parent's name and tag for quick and easy identification (empty otherwise).

Finally, it is important to note that the only vulnerability information included in the analysis file will consist of the total number of vulnerabilities contained in each image, as well as a list of CVE numbers referring to their detailed vulnerabilities located in the vulnerability file.

```
1 [
2   {
3     "image_id":      <image 1's digest>,
4     "type":          <image 1's type of image>,
5     "name":          <image 1's name>,
6     "tag":           <image 1's tag>,
7     "last_updated":  <image 1's last updated epoch timestamp>,
8     "total_pulled":  <image 1's total pull count >,
9     "vulnerabilities": [<image 1's list of CVEs>],
10    "total_vulnerabilities": <image 1's total num. of vulnerabilities >,
11    "parent":         <image 1's parent if any>
12  },
13  {
14    "image_id":      <image 2's digest>,
15    "type":          <image 2's type of image>,
16    "name":          <image 2's name>,
17    "tag":           <image 2's tag>,
18    "last_updated":  <image 2's last updated epoch timestamp>,
19    "total_pulled":  <image 2's total pull count >,
20    "vulnerabilities": [<image 2's list of CVEs>],
21    "total_vulnerabilities": <image 2's total num. of vulnerabilities >,
22    "parent":         <image 2's parent if any>
23  },
24  ...
25 ]
```

Listing 3.1: The designed analysis JSON file

Vulnerability results' format definition

As illustrated in the below 3.2 listing, all the detailed vulnerabilities collected about the analyzed images of an experiment will be stored in a dedicated JSON formatted file. Although many metadata may be gathered about software vulnerabilities, only three types of information are considered valuable for this study, besides a vulnerability's CVE number. Indeed, the package name and version affected by a specific vulnerability, as well as its severity level will be essential

for our study in order to analyze whether one of those attributes stand out for certain types of Docker images or across them.

```

1 [
2   {
3     "cve_number":      <vulnerability 1's CVE number>,
4     "package_name":   <vulnerability 1's affected package name>,
5     "package_version": <vulnerability 1's affected package version>,
6     "severity":       <vulnerability 1's severity>
7   },
8   {
9     "cve_number":      <vulnerability 2's CVE number>,
10    "package_name":   <vulnerability 2's affected package name>,
11    "package_version": <vulnerability 2's affected package version>,
12    "severity":       <vulnerability 2's severity>
13  },
14  ...
15 ]

```

Listing 3.2: The designed vulnerability JSON file

3.2.5 Detailed research questions definition

As discussed in 3.2.3, the JSON formatted result data of each experiment will be imported into a noSQL database for further analysis. The objective is to analyze all of the gathered data through so called "research queries", which will simply consist of a noSQL translation of textual detailed research questions implied by the thesis' problem statement. As explained in 1.2, the latter consists of three main research questions which imply many more. For example, the question *Are vulnerabilities still inherited from images' parent(s) and in what proportion?* imply that the following detailed questions need to be addressed:

1. What proportion of Docker images depends on a parent?
2. What proportion of Docker images contain inherited vulnerabilities?
3. How many vulnerabilities are inherited by an image in average?

The above list is only a very small sample of the detailed research questions which need to be defined in order to deeply answer the problem statement. The last part of the design phase will therefore consist of defining detailed research questions that may be translated into research queries in the implementation phase, in view of answering the original problem statement in details during the analysis.

3.3 Implementation

The implementation phase of the methodology aims at defining research queries, as well as implementing the software designed in the previous section using appropriate tools and technologies, while creating a dedicated architecture to deploy the software and conduct experiments.

3.3.1 Tools and technologies

The implementation and deployment of the software designed in 3.2.3 to conduct experiments gathering metadata, vulnerability and parental information for the defined data set of Docker images requires a defined set of tools and technologies.

Python 3

The software will be implemented in Python 3 as a result of two factors. First, the software is planned on interacting heavily with the Docker engine in order to download images from Docker Hub and gather their basic metadata information once downloaded (explained in details in 3.2.3). Since Docker offers a complete Software Development Kit (SDK) in Python for its Docker engine, the programming language is therefore a language of choice for the implemented software. Secondly, the latter will also interact densely with the Docker Hub platform in order to retrieve lists with the most recent images in all the repositories of a specific type or a subset of them, as well as collecting extra metadata information about those images. Due to the language's huge library versatility for Web interactions, Python is therefore the absolute language of choice for implementing the software. Note that the version 3 of the language will be used as it will be long-term supported, whereas version 2 will not be maintained after 2020 and one of this work's objectives is to provide a durable basic tool to conduct similar future studies [43].

Docker Hub's API

As discussed in 2.5.6, Docker Hub does not possess any documented API at the time of this writing. Nonetheless, a hidden and undocumented REST API seem to exist, as some very specific HTTP GET requests towards the *hub.docker.com* domain are authorized and return valid JSON objects. Thus, Docker Hub's hidden API is planned on being used by the implemented software to communicate with the platform and retrieve lists of images to be downloaded, as well as obtaining extra metadata information about them. A list of valid HTTP GET requests will therefore be needed to be identified first, in order to implement them into a Python API and be used actively by the rest of the software. Note that the provided Python API, as well as the list of valid HTTP requests towards Docker Hub's API will constitute an extra contribution to the research community besides our result data an analysis.

Clair scanner

As explained in details in 3.2.3, Docker images which are downloaded in order to be studied will be scanned for vulnerability information gathering using an appropriate image scanner. Although there exists several paid, free, closed and open source alternatives, the open source Clair software developed by CoreOS, a Red Hat company, is the technology of choice for this task. Indeed, Clair is considered the state of the art of container image vulnerability scanners and has been used by previous similar research such as the one conducted by Shu et al. in 2016 [11] (discussed in details in 2.6.1 of the background chapter). Since one of this study's objectives is to see whether the security measures introduced by Docker Inc. in response to previous research have improved Docker Hub's security landscape, the vulnerability information collected from analyzed images should be as comparable as possible to previous studies. Thus, Clair is the technology of choice selected to conduct static analysis of the set of Docker images that will be studied.

Nonetheless, the Clair software comes in multiple implementations which offer different degrees of complexity with distinct architectural differences [44]. Originally based on a client-server

paradigm which may potentially make Clair’s setup heavy and fastidious, other standalone and containerized implementations offer out of the box Clair functionalities, which are more suitable for our purpose. For example, Clair scanner consists of an implementation of the Clair technology into a standalone vulnerability scanner for the analysis of local Docker images [45]. The tool is actually based on the original but poorly maintained `analyze-local-images` utility created by CoreOS and is actively supported at the time of this writing. Whenever a Docker image is inspected by Clair scanner, the latter investigates each and every layer making up the image using the Clair technology, in order to identify contained packages and verify whether their version is contained in a vulnerability database. Clair scanner offers a containerized version of the software, as well as a containerized version of the vulnerability database, which is updated daily and indexes vulnerabilities from multiple sources such as Debian’s security bug tracker, the National Institute of Standards and Technology (NIST)’s National Vulnerability Database (NVD) or the Alpine Security Database [46]. Clair scanner will therefore be the tool of choice selected to interact with the implemented software and analyze the set of downloaded Docker images to be studied for vulnerability information.

MongoDB

As discussed in 3.2.3 of the design section above, the result data consisting of a list of structured objects containing all the gathered metadata, vulnerability and parental information of each of the analyzed images into a JSON formatted file is planned on being imported into a noSQL database. Due to its quick setup, multi-platform support and easy declarative query language using JSON syntax, MongoDB will be the preferred noSQL database for holding the result data in view of further analysis from any machine.

3.3.2 Architecture

Once the software to gather metadata, vulnerability and parental information for each set of Docker images involved in the research is implemented using the tools and technologies discussed in 3.3.1, it may finally be deployed onto a suitable architecture for conducting experiments. Although the latter may be conducted on a series of *nix based platforms, it is important to note that some images which will be analyzed may consist of several gigabytes. Considering a platform with a large disk space capacity and a high speed network, while providing fast recovery in case of problems is therefore essential. The OpenStack platform hosted at Oslo Metropolitan University, Norway (OsloMet) fulfills such requirements and will therefore be the platform of choice to deploy the implemented software.

As illustrated in figure 3.4 below, the software should be deployed onto a single VM running in OsloMet’s own OpenStack environment with the Docker engine installed and an access to the outside world in order to communicate with the Docker Hub platform. Furthermore, a containerized version of Clair scanner and a containerized Clair database will be deployed alongside the software and the Docker engine running inside the VM, to provide the software with Docker image vulnerability scanning capabilities. Note that the Clair services will not be communicating with the outside world, as they should only interact with each other and the deployed software to retrieve an image’s vulnerability information. Finally, a preliminary parent database generated by the deployed software through the Docker engine during its initialization phase will also be located aside the other components in order to successfully retrieved images’ parents when applicable.

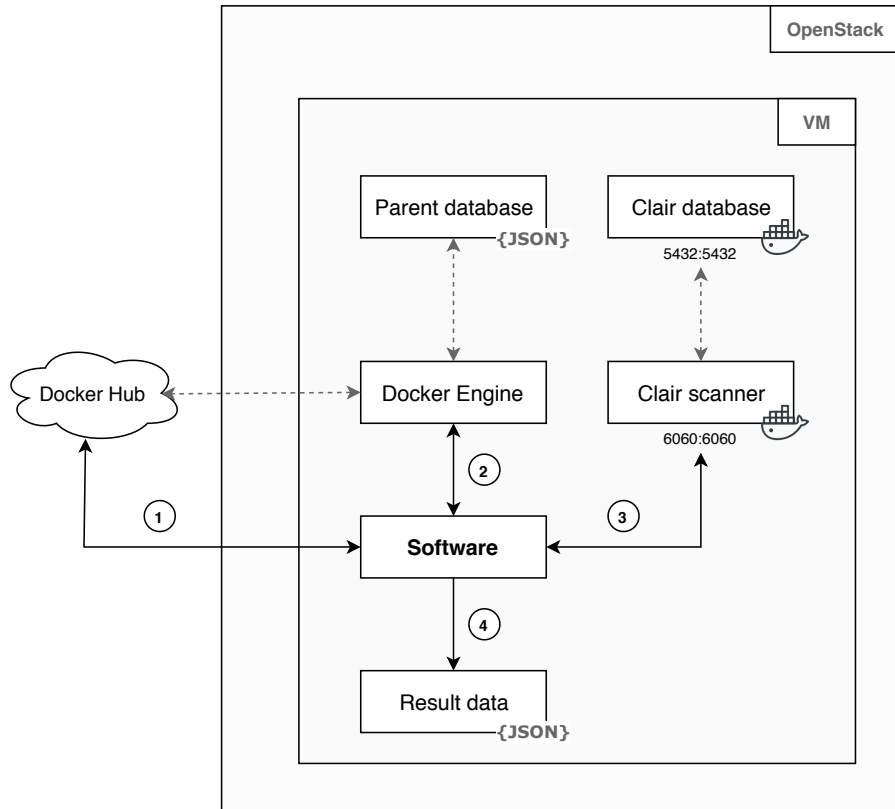


Figure 3.4: The planned architecture

It is important to note that the architecture planned on being used to deploy the software and conduct experiments will not include the Mongo database discussed in 3.3.1, as its only purpose is to allow collecting data and not provide a solution for data analysis. Indeed, the latter will be executed locally on one of our machines using an instance of MongoDB and a copy of the analysis and vulnerability files gathered during experiments, as discussed in 3.4.

3.3.3 Intended workflow

Once the implemented software is deployed onto an OpenStack VM loaded with the parent database, the Docker engine and the Clair scanner services, experiments to gather metadata, vulnerability and parental information for each set of Docker images involved in the research may be conducted. The intended workflow planned on being used for conducting experiments with each type of repository consists of six central steps once the deployed software is run, as illustrated in figure 3.5 below.

First, a list of names and tags for the most recent images in each repository of the required type is retrieved from Docker Hub through its identified API. Secondly, all the images retrieved in the list are downloaded locally on the OpenStack VM running the software for further analysis. Thirdly, basic metadata extracted directly from the local images such as their ids, names and

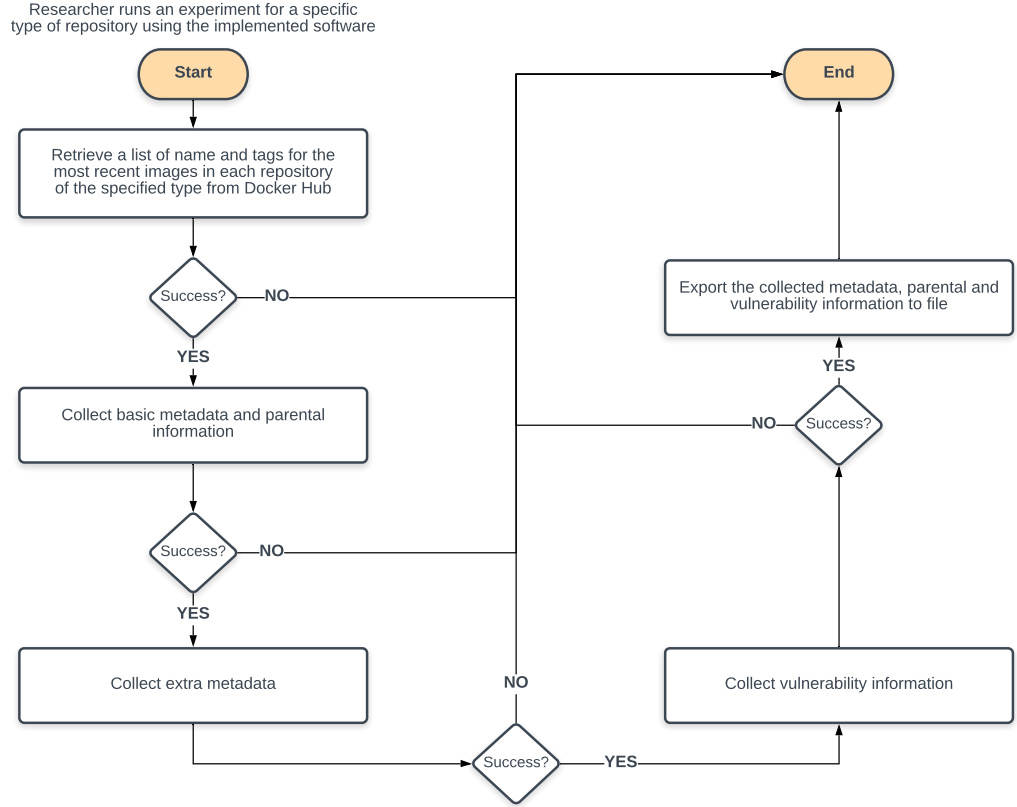


Figure 3.5: The planned experiments' workflow

tags are collected, as well as parental information retrieved via the preliminary parent database generated by the software prior to the conduction of each experiment (more details in 3.2.2). Fourthly, extra metadata such as images' last updated timestamp or total number of pulls is obtained from Docker Hub's API for each of the downloaded images. Fifthly, each local image is scanned for vulnerabilities using Clair scanner and its vulnerability database. Finally, all the collected basic and extra metadata information, as well as the gathered parental and vulnerability information is structured into JSON objects as explained in 3.2.4 and exported to a vulnerability and analysis JSON file, which may be imported into a Mongo database later on for further analysis through the use of implemented research queries.

3.3.4 Research queries definition

As discussed in 3.2.3, the JSON formatted result data of each experiment will be imported into a noSQL database for further analysis. The objective is to analyze all of the gathered data through so called "research queries", which consist of a noSQL translation of the textual detailed research questions identified in the design phase of our methodology in 3.2.5. For example, the detailed

question *What proportion of Docker images depends on a parent?* may be translated into a noSQL research query as followed:

```
db.getCollection(<name>).find({"parents": {"$exists" : true, "$ne" : ""}},
{ _id: 0, name: 1, parents: 1 }).count();
```

The last part of the implementation phase will therefore consist of translating all the detailed research questions identified in the design phase of the methodology into research queries, in view of using them to address the original problem statement in depth during the analysis.

3.4 Measurements and analysis

As explained in 3.3.2, measurements will be made through the conduction of experiments for each set of defined Docker images intended to be studied. Indeed, those measurements will be executed from an OpenStack Cloud architecture and will consist of a collection of metadata, vulnerability and parental information of each of the analyzed images into a list of structured JSON objects contained in two distinct result files (explained in details in 3.2.4). The obtained JSON files are thereafter intended to be imported into a noSQL database such as MongoDB for further analysis through the use of research queries discussed in details in 3.3.4.

As briefly mentioned in 3.3.2, the analysis is not intended to be conducted on the OpenStack platform, but rather on a local machine loaded with an instance of MongoDB. This way, the analysis of the collected data may be conducted anywhere, without having to rely on a third party platform such as OpenStack or any specific OS thanks to MongoDB's and the JSON format's cross-platform compatibility. Moreover, the analysis will intend to answer the identified detailed research questions mentioned in 3.2.5 of our methodology, by using their translated version in noSQL referred to as research queries discussed in 3.3.4.

The final phase of this thesis' methodology aims therefore at conducting measurements through data collection and analyze the obtained results using detailed research queries, in view of answering the original problem statement introduced in 1.2.

3.5 Expected results

It is expected that each set of Docker images to be studied and defined in 3.2.1 will consist of a JSON formatted analysis and vulnerability file containing a collection of metadata, vulnerability and parental information for each of the analyzed images in the set. Although some images will be skipped due to their incompatible platform as discussed in 3.2.1, it is expected that a vast majority of those images will be successfully analyzed. Furthermore, it is anticipated that the result JSON files will be importable into a noSQL database such as MongoDB for further analysis and that such results will be analyzable through the help of our implemented research queries to answer the original problem statement.

Note that there exists some uncertainties around the potentially excessive downloading of images from the Docker Hub platform, which may lead to the blacklisting of our OpenStack VM's public IP address. In that case the use of a proxy such as a Virtual Private Network (VPN)

is intended to be used in order to switch between multiple IP addresses. Additionally, it is unclear whether certain images with very specific properties may require some hard coding of their metadata information for a successful retrieval and complete measurement through the implemented software, as some repositories may not follow recommended standards and practises.

Finally, Certified repositories are expected to provide the best security level of all types, followed by Official and Verified repositories anticipated to be similarly secure and concluded by Community repositories which are expected to offer the worst level of security of all four types of repositories. Indeed, Certified repositories are supposed to provide better security standards as explained in [2.5.2](#), they are therefore expected to be the most secure repositories of all types, with the lowest number of vulnerabilities in average, as well as the least number of vulnerabilities with a high severity. Moreover, Official and Verified repositories vetted by Docker's team as well as trusted third party vendors are expected to provide a moderate level of security with a limited number of vulnerabilities and severity level in average due to their approved profile provide by Docker Inc. Community repositories however may be uploaded by anyone and are therefore anticipated to contain a large number of vulnerabilities in average with a high level of severity. It is also important to note that most of the inherited vulnerabilities are expected to be acquired from Official images, as they seem to be the most popular ones on Docker Hub at the time of this writing.

Chapter 4

Result 1: Design

This chapter describes the result design of the methodology outlined in chapter 3, defining the study's scope in terms of concrete images and repositories that will be analyzed, as well as the designed software that will be used to conduct experiments and gather metadata, vulnerability and parental information for the defined data set of Docker images, in view of answering the defined research questions identified during this phase of the methodology.

4.1 Data set

As previously mentioned in 2.5.1, Docker Hub hosts over two million repositories at the time of this writing, making a comprehensive study of the whole platform out of scope for this thesis.

4.1.1 Defined data set

Since one of the main goals of this work is to determine whether the security measures introduced by Docker Inc. in the form of an image security scanner and two new types of repositories with higher security requirements have modified Docker Hub's security landscape, all types of repositories were analyzed.

As discussed in detailed in 2.5.2, Community repositories make up more than 99.98% of all repositories on Docker Hub, leaving Official, Certified and Verified images with a very minimal amount of repositories (a few dozen to a couple of hundred). Thus, all three types of repositories were part of the studied data set in their entirety, whereas only 500 out of 1500 randomly selected Community repositories among the most popular ones were included in this research due to time constraint.

Moreover, many repositories carry hundreds or even thousands of images. For that reason, it was chosen to solely analyze the most recent image in each repository, as such images should consist of the least vulnerable image in their repository due to their most up-to-date property.

Finally, it should be noted that this study limited the chosen data set to images compiled for the x86-64-bit architecture and dedicated to the Linux platform only. Thus, Docker images compiled for the Windows platform or other processor architectures were not taken into account, as their

number in Docker Hub's ocean is very minimal [33]. Nonetheless, skipped repositories did not limit themselves to the ones incompatible with the Linux platform or x86-64-bit architecture, as many repositories contained unpredictable or abnormal properties leading to their leaping as explained in the next subsection.

4.1.2 Skipped repositories

Due to unpredictable or abnormal properties, some repositories defined in our data set in 4.1.1 and available on Docker Hub were not included in this study. There was mainly six different reasons for a repository to be skipped from our analysis.

First, certain repositories such as the "portworx/px-dev" Certified repository were visible through Docker Hub's Web interface but were not downloadable through the docker pull command provided by the repository owner, throwing the below error. The same error message applied to 20 other repositories, which were therefore excluded from the analyzed data set of images. Note that the complete list of skipped repositories due to that error is provided in Appendix A.2.

```
Error response from daemon: manifest for store/portworx/px-dev:latest not found
```

Secondly, repositories containing images only compiled for other processor architectures than the x86-64-bit architecture or the Linux platform were skipped throwing one of the below errors. Note that most of the incompatible repositories were maintained by the Microsoft publisher at the time of this writing and constituted a total of 10 repositories listed in details in Appendix A.3.

```
no matching manifest for unknown in the manifest list entries
```

```
image operating system "windows" cannot be used on this platform
```

Thirdly, the Official repository known as "scratch" was a special repository which did not contain any image that may be pulled, run or tagged, as it simply consisted of a reserved repository which may be optionally used in a Dockerfile when creating a base image, as explained in 2.5.4. Thus, the Official *scratch* repository was skipped from the analysis.

Fourthly, images throwing a "Manifest not found" error due to miscellaneous reasons such as internal server errors or an over solicited Docker daemon incapable of handling a certain pulling request were automatically handled by our software and skipped from analysis.

Fifthly, certain Microsoft repositories containing only other repositories and no images as explained in details in 2.5.3 were also skipped. Indeed, repositories such as "microsoft-dotnet-framework" were handled as any other repository on Docker Hub, but they did not contain any image, leading to the below error to be thrown when attempted to be pulled. Note that a total of 9 repositories were in that case at the time of this writing, all maintained by the Microsoft publisher and listed out in details in Appendix A.4.

```
Error response from daemon: pull access denied for microsoft-dotnet-framework,  
repository does not exist or may require 'docker login'
```

Finally, the remaining images which were not included in this study were unsupported images by Clair scanner , duplicate repositories or images without a valid tag.

| | Initial number of repositories | Total analyzed repositories | Skipped repositories | Analysis rate |
|-----------|-----------------------------------|--------------------------------|-------------------------|------------------|
| Official | 151 | 128 | 23 | 84,77% |
| Community | 500 | 500 | 0 | 100,00% |
| Verified | 208 | 98 | 110 | 47,12% |
| Certified | 44 | 31 | 13 | 70,45% |
| Total | 903 | 757 | 146 | 83,83% |

Table 4.1: A summary of the experiments performed in this study

Table 4.1 shows a summary of the repositories which were not included in this research. To sum up, about 20% of the initial number of repositories were skipped. Furthermore, it should be noticed that 100% of the planned number of Community repositories were actually analyzed, as any skipped repository due to a lack of Clair scanner support was replaced by another one among the most popular Community repositories. Finally, note that the initial number of Verified repositories (208) did not include Certified repositories, as the latter (44) were analyzed on their own.

4.2 Preliminary requirements

Preliminary requirements were needed to be taken into consideration before developing the software. The planned parent database discussed in 3.2.2 became separated into two different databases, while the detailed manual checkout of certain Certified and Verified repositories mentioned in the planned design were executed as precised below.

4.2.1 Two parent databases

The single parent database proposed in the planned design in 3.2.2 became separated into two distinct databases: one for Official images and one for Verified images (including Certified). As explained in 3.2.2, our methodology implies that a parent database containing a layer signature uniquely identifying all the images present in all the repositories susceptible to be used as parents is created prior to the conduction of each experiment. As a reminder from 2.5.2, Official images which are not base ones are extending another Official image, whereas Certified and Verified images may be based on Certified, Verified or Official images. As for Community images, they may simply be based on any type of image.

Having two distinct parent databases containing a unique layer signature for all the images comprised in all the Official, Certified and Verified repositories allowed therefore determining whether an image had a parent, by simply looking at the upper layer combination of an input image into the databases. This way, the latter were able to return the name and tag of an image’s parent if any, as illustrated in figure 4.1 below with the Official parent database. In

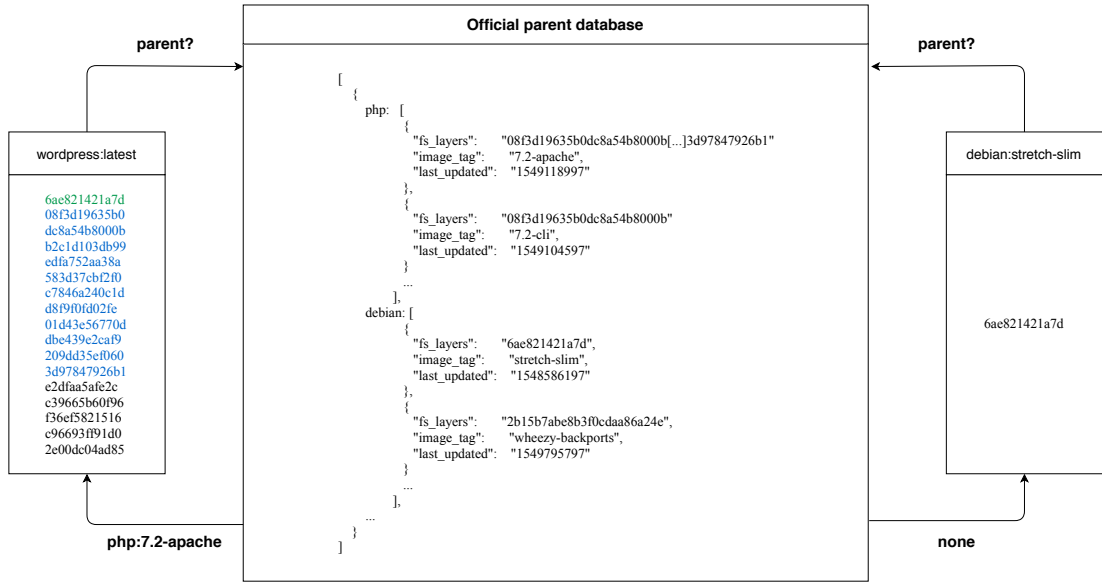


Figure 4.1: The designed Official parent database

that example, the input image "wordpress:latest" successfully retrieves its parent, the "php:7.2-apache" image, from the database. The Wordpress image contains a total of 17 layers and the database contains all the layer combinations of all the Official images. As shown in the above figure, all the repositories' images were stored in a list with their tags and layer combinations inside a nested JSON object.

Thus, a parent lookup first traversed through the upper layer combination of an image, in order to find whether that combination was found in the database. In this example, the algorithm finds the image "debian:stretch-slim" as the input image's first and furthest parent (marked in green). Thereafter, the following upper layer combinations would be processed until another match was found, leading to the discovery of a second closer parent ("php:7.2-apache" marked in blue). Finally, the designed algorithm for parent retrieval processes the remaining layer combinations until the end, leading to no other match in this particular example. It is important to note that the direct parent of an input image (i.e. the one specified in the image's Dockerfile) is last one retrieved from a parent database ("php:7.2-apache" in this case), whereas the first ones correspond the input image's grandparents ("debian:stretch-slim" in this case).

The advantage of having two separate databases was that it provided a more efficient way of indexing images' layers and identify parent images, by only executing parent lookups towards the appropriate database. For example an Official image may only be based on another Official image. Thus, trying to retrieve its parent from a single database containing thousands of layer signatures for other types of images would add an unnecessary overhead. Using a separate database for Official signatures and another one for Verified signatures allowed therefore improving parental lookups greatly.

The way those parent databases populated was however very different due to the way distinct repository types were handled by the Docker engine.

The Official parent database

The Official parent database was populated by making use of the Docker Engine's pull command and images' short layer IDs described in 2.4.2, avoiding therefore downloading Official images completely. Indeed, the Docker SDK in Python allowed interacting with the low-level API of the Docker daemon, streaming pulling outputs in real time and therefore displaying all the layers composing an image with their short IDs, before actually pulling those layers. By parsing an image's short layer IDs from its live pulling output, all the images in all the Official repositories might be retrieved in an acceptable amount of time, without the need of actually downloading them. Moreover, many repositories contain several thousand tags, which does not necessarily mean that their number of images is equally large, as a single image may be tagged an indefinite number of times. Thus, it is important to note that the Official parent database contained an image's signature only once, using the image's first retrieved tag.

The Verified parent database

Contrary to its peer, the Verified parent database was not able to populate by making use of the Docker Engine's pull command and images' short layer IDs. Indeed, this technique implied that a list of all tags present in a repository was retrieved prior to the download of the repository's images, so that each specific image might be started to be pulled individually and stopped once their short layer IDs were retrieved to avoid downloading them entirely. As explained in details in 5.2.2, a repository's list of tags was retrieved through the use of Docker Hub's non-documented API version 2, which did not support Verified repositories at the time of this writing. Furthermore, although version 1 of Docker Hub's API supported Verified repositories as discussed in 5.2.1, it did not provide a way of retrieving a repository's list of tags.

Thus, the Verified parent database was populated by downloading all the images contained in every single Verified repository using the *docker pull -all-tags* equivalent in the Docker SDK in Python and retrieve a unique layer signature for each downloaded image. Moreover, note that similarly to its peer, the Verified parent database contained an image's signature only once, using the image's first retrieved tag, as a single image located in a repository may have multiple tags, but only consists of a single layer combination.

Finally, the handling of the two parent databases consisted of a standalone Python module separated from the main module for gathering metadata and vulnerability information, which made it possible to update single databases in order to overcome the problem of having a too long lapse of time for image signatures not being updated while the databases were repopulated from scratch, which took several days. Compared to the planned design discussed in 3.2.2, that particular improvement led to a small modification in the implemented design of the database, with two database files and a supplementary attribute in each image object discussed in details in 4.4.

4.2.2 Manual image checkout

As explained in details in 2.5.2, the two new types of repositories introduced by Docker Inc. in an attempt to increase its public registry's security landscape are provided by vetted third-party software vendors who often offer proprietary and/or paid content. As a consequence, many providers require a valid Docker Hub account as well the manual checkout of their Certified/Verified repositories providing some contact information in order to be able to download the images they hold. Out of the 44 Certified repositories hosted on Docker Hub at the time of this writing, only

11 required a manual checkout through the platform’s Web interface, whereas 26 out of the 208 Verified repositories were in that case. Finally, note that a complete list of the manually checked out repositories is available in Appendix A.5.

4.3 Overview

The software was designed to conduct multiple experiments, gather metadata, vulnerabilities, and parental information for the data set of Docker images defined in 4.1. Each experiment applied for each type of repository on Docker Hub: Official, Certified, Verified, and Community repositories. However, a limited set of Community images was required, as analyzing over million images is out of this project’s scope. Therefore, the designed software under the name DAZER standing for Docker imAge analyZER and consisting of a Python bundle provided an extra argument besides the repository type to be analyzed, specifying the number of Community repositories to analyze (discussed in details in 5.4).

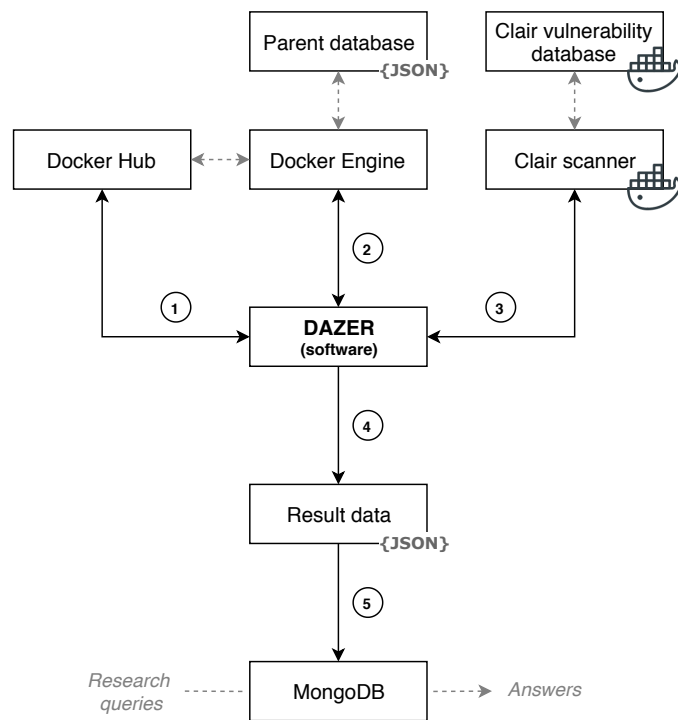


Figure 4.2: The implemented design’s overview

Figure 4.2 above shows the designed software used to conduct experiments to gather metadata, vulnerability and parental information for the defined data set of Docker images. As illustrated, DAZER interacted with three main components, similar to the planned design proposed in 3.2.3 and consisting of the following five steps:

1. A list of the most recent images in all repositories of a specific type (e.g. Certified repositories) or a subset of them is retrieved from Docker Hub.

2. The software interacts with the Docker engine running on the same host and downloads the most recent image in all the retrieved repositories, while gathering their metadata and parental information.
3. All the downloaded images are analyzed for vulnerability information using a containerized version of Clair scanner linked to an up-to-date containerized vulnerability database.
4. The retrieved information is wrapped up to a list of structured objects containing the gathered metadata, parental information (if any), and all the vulnerabilities of each image. The latter will be converted to two JSON files: one for the images' metadata and parental information and one for the images' vulnerability metadata.
5. The result data will be imported into a MongoDB for further analysis using research queries described in details in [4.5](#).

Finally, note that the actual design of the software was very similar to the one that was originally planned in the methodology chapter presented in [3.2.3](#).

4.4 Designed result data format

As previously mentioned in [4.3](#), the result data structured in the JSON format was imported into a noSQL Mongo database for further analysis in the next phase of our methodology. Due to the large number of metadata, vulnerability and parental information gathered about each image composing the data set of Docker images defined in [4.1](#), the structured format of those collected data was rigorously designed as followed.

Designed parent database format

Both the Official and Verified parent databases used the JSON format in order to structure their data. Compared to the proposed design in [3.2.4](#) as part of the methodology chapter, the *last_updated* attribute was added to every single image object contained in the databases. That attribute was utilized to verify whether an image had been updated since the last database update and might therefore need to be excluded from analysis, as such an image might potentially be based on a new parent which would not be contained in any of the parent databases.

The parent database format consisted therefore of a series of repositories structured in a list of JSON objects, containing each a list of unique images with a corresponding tag, layer combination and last updated timestamp, as illustrated with a part of the Official database below:

```

1  [
2    {
3      "mongo": [
4        {
5          "fs_layers": "f3d19635b0dc ",
6          "image_tag": "v1.2 ",
7          "last_updated": "34534345753567"
8        },
9        {
10         "fs_layers": "re349635b0edde316d5cf0dc ",
11         "image_tag": "3.1.0 ",
12         "last_updated": "1294348944848"
13       }
14     ]
15   }

```

```

14         ], ...
15     ],
16     "ubuntu": [
17         {
18             "fs_layers": "dc8a54b8000b",
19             "image_tag": "1",
20             "last_updated": "0348593458558"
21         },
22         {
23             "fs_layers": "ab8fbc38cdf2b60ff3e76baa",
24             "image_tag": "latest",
25             "last_updated": "34534345753567"
26         }
27     ], ...
28 ], ...
29 ], ...
30 ], ...
31 }
32 ]

```

Listing 4.1: Sample of the designed Official parent database (JSON file)

Designed analysis and vulnerability results format

Due to the large number of metadata and vulnerability information, the design of the data formats had to be properly defined and structured. After the execution of an experiment, two JSON files were generated by the DAZER software: **analysis-YYYY-MM-DD-HH-MM.json** and **vulnerabilities-YYYY-MM-DD-HH-MM.json**, where the timestamp corresponded to the time when the experiment was initiated. The resulting JSON files were respectively defined as followed:

- A file containing metadata and parental information for all the analyzed images
- A file containing all the vulnerability information discovered in the analyzed images

The example below illustrates a sample of an **analysis-YYYY-MM-DD-HH-MM.json** file with two analyzed images:

```

1  [
2    {
3      "image_id": "94e814e2efa8845d95b2112d54497fbad173e45121ce9255b93401392f538499",
4      "type": "official",
5      "name": "ubuntu",
6      "tag": "latest",
7      "last_updated": 1552350017,
8      "total_pulled": 1007214825,
9      "vulnerabilities": ["CVE-2018-10846", "CVE-2018-16869", ... ],
10     "total_vulnerabilities": 31,
11     "parent": ""
12   },
13   {
14     "image_id": "facedfff4424a99d694b03ca36060eb0dd1e35dbe26e0d1bae2986fbbef7092e",
15     "type": "community",
16     "name": "drone/drone",
17     "tag": "latest",
18     "last_updated": 1553032060,

```



```

19     "total_pulled":          64727991,
20     "vulnerabilities":      [],
21     "total_vulnerabilities": 0,
22     "parent":               "alpine:latest"
23 },
24 ....
25 ]

```

Listing 4.2: Sample of the designed analysis JSON file

As mentioned in 3.2.3, each Docker image is composed of a series of layers identified through layer IDs, corresponding to a SHA256 hash of the layers' content. Therefore, it was not possible to use an image's name as a valid identifier, as many images contain the same name. Alongside the essential metadata information for each analyzed images, some of the attributes contained in the analysis file were chosen based on the potential valuable information they may provide to this study. For instance, it might be interesting to see whether there exists a correlation between attributes such as images' last updated timestamps (*last_updated*) or their total amount of downloads (*total_pulled*). Moreover, this thesis attends to address a research question related to vulnerability inheritance, making the identification of an image's parent essential. The latter corresponded therefore to a string within the analysis JSON file, which was either empty, meaning that the image was a base image, or containing the name and tag identifying its direct parent.

Combined with the field *vulnerabilities*, which represented a list containing all the image's vulnerabilities holding unique CVE numbers, it might be compelling to analyze whether there exists a correlation between the contained vulnerabilities in child and parent images. Each CVE number was therefore stored as a dedicated JSON object within the vulnerabilities JSON file illustrated below with two vulnerabilities:

```

1 [
2   {
3     "cve_number":          "CVE-2017-14532",
4     "cwe_number":          "CWE-476",
5     "package_name":        "imagemagick",
6     "package_version":     "8:6.9.7.4+dfsg-11+deb9u6",
7     "severity":             "High"
8   },
9   {
10    "cve_number":           "CVE-2018-1000500",
11    "cwe_number":           "CWE-295",
12    "package_name":         "busybox",
13    "package_version":      "1:1.22.0-19",
14    "severity":              "Negligible"
15  },
16  ...
17 ]

```

Listing 4.3: Sample of the designed vulnerability JSON file

The vulnerability file included the metadata for all the images' vulnerabilities which had been collected throughout a conducted experiment in the form of their CVE number, package name and version, as well as severity level. Those information were extremely important for this study in order to analyze whether one of these fields stood out for certain types of Docker repositories or across them. However, compared with the previously proposed design in 3.2.4, the *cwe_number* attribute which categorizes software weaknesses and vulnerabilities was introduced (more details about CWE in 2.1.3). For instance, CWE-295 which relates to CVE-2018-1000500, corresponds

to an "Improper Certificate Validation" vulnerability which may be exploited by an attacker in order to impersonate a trusted entity via a Man-In-The-Middle (MITM) attack [47]. This newly introduced attribute might therefore help answering what kind of vulnerabilities Docker images were most susceptible to during the measurement and analysis chapters.

Finally, note that only the analysis and vulnerabilities JSON files were imported to the Mongo database in the next phase of our methodology, as they contained the entirety of the gathered metadata, parental and vulnerability information of a set of analyzed images during an experiment.

4.5 Detailed research questions

As discussed in 4.4, the JSON formatted result data of each experiment was imported into a noSQL Mongo database for further analysis in the next phase of our methodology described in chapter 5. The objective was to analyze all of the gathered data through so called research queries, which consisted of a noSQL translation of textual detailed research questions implied by the thesis' problem statement. As explained in 1.2, the latter consisted of three main research questions which therefore implied the following detailed inquiries:

RQ1: Have the security measures introduced by Docker Inc. in response to previous research improved Docker Hub's security landscape and to what extent?

1. How many vulnerabilities do Certified or Verified images contain compared to Official and Community images?
2. How many vulnerabilities do Certified or Verified images inherit compared to Official and Community images?
3. How many vulnerabilities do Certified or Verified images introduce compared to Official and Community images?
4. How many vulnerabilities with a high severity do Certified or Verified images contain compared to Official and Community images?
5. What are the top 10 vulnerability types contained in Certified or Verified images compared to Official and Community images?
6. Do Certified images provide better security than the rest of the Verified images?

RQ2: Are vulnerabilities still inherited from images' parent(s) and in what proportion?

1. What proportion of images depends on a parent?
2. What proportion of images contains inherited vulnerabilities?
3. What proportion of images introduce vulnerabilities?
4. How many vulnerabilities do images inherit in average?
5. How many vulnerabilities do images introduce in average?

RQ3: How are discovered vulnerabilities distributed across repository types?

1. What proportion of images contains no vulnerabilities?
2. What proportion of images contains at least one vulnerability?
3. How many vulnerabilities do images contain in total?
4. How many unique vulnerabilities do images contain in total?
5. How are unique vulnerabilities distributed per severity among images?
6. How are unique vulnerabilities distributed per year among images?
7. Is there a correlation between the most popular images (most pulled) and the most vulnerable ones?
8. Is there a correlation between the last updated images and the most vulnerable ones?
9. Is there a correlation between base images and the most vulnerable ones OR base images and the most popular ones OR base images and the last updated ones?
10. What are the top 10 vulnerabilities across all types of repositories?
11. Are vulnerabilities found in base images correlated with non-base images in some way?

Chapter 5

Result 2: Implementation

This chapter describes how the designed software discussed in 4.3 is implemented using selected tools and technologies, while translating the detailed research questions identified in section 4.5 into research queries. Note that the latter will be utilized actively in the next chapter, where a detailed presentation of the gathered metadata, parental and vulnerability information for the defined set of Docker images will be given. Note also that the code developed for implementing the software is available in its entirety at <https://github.com/dockalyzer/dazer>.

The implementation phase of our methodology is discussed in details in 3.1 and consists of the following steps:

1. Identify tools and technologies required for creating and running the software designed in the previous phase
2. Implement the software designed in phase 1 to gather metadata, vulnerability and parental information for each set of Docker images involved in the research
3. Build the environment required to run the implemented software
4. Translate the detailed research questions identified in phase 1 into NoSQL research queries

5.1 Tools and technologies

The DAZER software has been implemented using a certain number of tools and technologies discussed in details in chapter 3 under 3.3.1 and are briefly summarized here.

First, multiple APIs were used in order to gather Docker image metadata and vulnerability information: Docker Hub API v1, Docker Hub API v2, Microbadger's API and the Computer Incident Response Center Luxembourg's (CIRCL) CVE API. The Docker Hub APIs discussed in details in 5.2 are not made public by Docker Inc. and many of the API calls were found using a simple trial and error approach. As for CIRCL's CVE API, the latter was implemented in the DAZER software to collect metadata such as affected package names and versions for the vulnerabilities discovered in the analyzed Docker images. Note that all the APIs used in the software are discussed in details in the next section.

Secondly, the Docker engine (version 18.09.3 at the time of this writing) was required to download Docker Hub images. Additionally, Clair's vulnerability scanner was utilized to analyze the downloaded images and retrieve their vulnerability information. Clair scanner, which was run inside a Docker container, was further linked to another Docker container running a PostgreSQL database containing all the disclosed software vulnerabilities made public so far. Note that a detailed description of Clair scanner and its containerized vulnerability database is provided in [3.3.1](#) and [4.4](#).

Thirdly, Python 3 and the Docker SDK for Python was used to interact programmatically with the Docker engine directly from within the implemented DAZER software. Along with the Docker SDK for Python, Python 3's requests library was utilized to send requests towards the REST APIs mentioned above and obtain JSON formatted data that could be parsed further into structured JSON objects for further analysis, as discussed in [4.4](#).

Finally, the gathered metadata, parental and vulnerability information was imported to a Mongo database for further analysis. MongoDB consists of a NoSQL database providing high-performance data processing, which allowed us to make use of the developed research queries detailed in [5.5.2](#), in order to answer the detailed research questions implied by the problem statement and identified in [4.5](#).

5.2 Retrieving data

The authors of the research paper "A Study of Security Vulnerabilities on Docker Hub" generated manually 5,000,000 random strings, which resulted in a list of 440,524 unique images on Docker Hub [11]. In our study, several APIs were used to automatically obtain the metadata, parental and vulnerability information of all the analyzed images. Given the fact that version 2 of Docker Hub's API was the most recent one at the time of this writing, using two different APIs for retrieving the metadata of Docker Hub images was needed, as the new types of images, Verified and Certified images, were only supported in version 1 of the platform's API.

Contrary to the planned implementation discussed in [3.3.1](#), two additional APIs were used for retrieving metadata besides Docker Hub's API. The first one was the MicroBadger API, which yields metadata for Official repositories on Docker Hub. The latter was utilized to discover all the Official repositories' tags, in use for updating the Official parent database. The second one was CIRCL's CVE API which provides metadata for CVE vulnerabilities.

5.2.1 The Docker Hub API: version 1

The Docker Hub API is a simple REST API for the Docker Hub platform, using basic HTTP requests and returning JSON formatted data. As previously mentioned in [5.1](#), the Docker Hub API is not made public by Docker Inc, which is therefore completely unofficial and undocumented. Thus, many of the API calls were found using a simple trial and error approach. One interesting finding was that only Official and Community repositories were supported by the version 2 of the API, while both Official, Verified, and Certified repositories were supported by version 1. As discussed in [2.5.3](#), Official repositories use "library" as their namespace and Community repositories use their username, while Certified and Verified repositories are located under the "store/<username>" namespace. Additionally, Microsoft repositories use "mcr.microsoft.com" as their namespace. Unfortunately, the version 1 of Docker Hub's API does not support tag

retrieval for a given repository. Thus, some workarounds were necessary in order to retrieve the tags comprised in Verified and Certified repositories, which is covered in details in 4.2.

A list of all the discovered API calls valid for the Docker Hub API v1 is available below. Note that the endpoint for the version 1 of the API was located at <https://hub.docker.com/api/content>.

Fetch a specific repository identified by its name

The repositories retrieved using version 1 of Docker Hub's image API contain a lot more details than the ones retrieved using version 2. Therefore, this call was utilized to determine whether an image belonged to an Official repository (under the "library" namespace) or a Certified/Verified repository (under the "store" namespace).

```
GET /v1/products/images/<repository_name>
```

Example with the Official "Ubuntu" repository:

<https://hub.docker.com/api/content/v1/products/images/ubuntu>

Example with the Certified "Filebeat" repository:

<https://hub.docker.com/api/content/v1/products/images/filebeat>

Example with the Verified "Filecloud" repository:

<https://hub.docker.com/api/content/v1/products/images/filecloud>

Fetch all Certified, Verified, and Official repositories

For better rigor and convenience purposes, the conducted experiments were divided based on the four repository types that were analyzed: Official, Certified, Verified, and Community repositories. The following API call was therefore used as a base to retrieve all the Certified repositories on Docker Hub, by iterating through the pages specified as the last argument of the call (*page=x*). Note that a single page returned 25 image objects by default, although that might be increased using the *page_size=xx* argument.

```
GET /v1/products/search?q=&type=image&certification_status=certified&page=1
```

In order to determine if a repository was a Verified repository, the first mentioned call, *GET /v1/products/images/<repository_name>* returned the *certification_status* field which was either equal to *not_certified* or *certified*. Hence, if the namespace was "store" and the *certification_status* was equal to *not_certified*, the image belonged to a Verified repository. The API call below may be used as a base to retrieve all Certified, Verified, and Official repositories by iterating through the pages specified as the last argument of the call (*page=x*). Therefore, combining the mentioned call above with the API call below, it was possible to filter and fetch all the purely Verified repositories (i.e. without Certified ones). Note that a single page returned 25 image objects by default, although that might be increased using the *page_size=xx* argument.

```
GET /v1/products/search?q=&type=image&image_filter=store&page=1
```

5.2.2 The Docker Hub API: version 2

As previously mentioned, the version 2 of the Docker Hub API only supported Official and Community repositories. Docker Hub's API v2 might be used to retrieve various types of metadata about specific images and repositories, such as the list of available tags within a repository.

A list of all the discovered API calls valid for the Docker Hub API v2 is available below. Note that the endpoint for the version 2 of the API was located at <https://hub.docker.com>.

Fetch a specific repository identified by its name

The repositories retrieved using version 2 of Docker Hub's image API contained less details than the ones retrieved using version 1. Similar to the API version 1, the API call below retrieved all the metadata of the specified repository.

```
GET /v2/repositories/<repository_name>
```

Example with the Official "Ubuntu" repository:

<https://hub.docker.com/v2/repositories/library/ubuntu>

Example with the Community "Amazon-ecs-agent" repository:

<https://hub.docker.com/v2/repositories/amazon/amazon-ecs-agent>

Fetch all the tags available under a specified repository identified by its name

Unlike version 1 of Docker Hub's API which was missing that ability, the following call was able to retrieve all the available tags present in the specified Official or Community repository identified by its name.

```
GET /v2/repositories/<repository_name>/tags
```

Example with the Official "Ubuntu" repository:

<https://hub.docker.com/v2/repositories/library/ubuntu/tags>

Example with the Community "Amazon-ecs-agent" repository:

<https://hub.docker.com/v2/repositories/amazon/amazon-ecs-agent/tags>

Fetch a specific tag located in the specified repository identified by its name

This particular API call illustrated below retrieved all the metadata of a specific tag located in a certain Official or Community repository identified by its name.

```
GET /v2/repositories/<repository_name>/tags/<tag_name>
```

Example with the Official repository "Node" and tag name "8.15-jessie":

<https://hub.docker.com/v2/repositories/library/node/tags/8.15-jessie>

Example with the Community repository "Amazon-ecs-agent" and tag name "latest":

<https://hub.docker.com/v2/repositories/amazon/amazon-ecs-agent/tags/latest>

Fetch all Official repositories on Docker Hub

Similar but more specific than Docker Hub's API version 1, the following call was used as a base to retrieve all the Official repositories available on Docker Hub exclusively, by iterating through the pages specified as the last argument of the call (*page=x*). Note that a single page returned 10 image objects by default, although that might be increased using the *page_size=xx* argument.

```
GET /v2/search/repositories/?query=library&is_official=true&page=1
```

Fetch all Community repositories on Docker Hub

At the time of this writing, there are over 2 million Community repositories available on Docker Hub. The following API call was therefore used as a base to retrieve 500 Community repositories among the most popular ones available on Docker Hub, by iterating through the pages specified as the last argument of the call (*page=x*). Note that a single page returned 10 image objects by default, although that might be increased using the *page_size=xx* argument.

```
GET /v2/search/repositories/?query=%2B&ordering=-pull_count  
&is_official=false&page=1
```

5.2.3 CIRCL's CVE API

The Computer Incident Response Center Luxembourg (CIRCL) is a government-driven initiative designed to gather, review, report and respond to computer security threats and incidents [48]. That organization provides a public REST API offering developers the possibility to fetch complex metadata about any disclosed vulnerability with an assigned CVE number in JSON. The goal was to utilize that API in view of extracting valuable metadata for all of the vulnerabilities discovered in the analyzed images through Clair scanner during our experiments. Note that the API's endpoint was located at <https://cve.circl.lu> and that only two calls towards CIRCL's CVE API were implemented in the DAZER software.

First, the following call was used to retrieve all the available information about a certain vulnerability identified by its CVE number.

```
GET /api/cve/<cve_id>
```

Example with the integer overflow vulnerability found in libssh2 before version 1.8.1 and identified with "CVE-2019-3855":

<https://cve.circl.lu/api/cve/CVE-2019-3855>

Secondly, CVE numbers are related to CWE numbers, classifying software vulnerabilities into specific categories as explained in 2.1.3. For example, the "CVE-2019-3855" number mentioned in the previous example is related to a CWE number with an id of 190. By utilizing the API call below, the meta information of CWE-190 might be retrieved.

```
GET /api/capec/<cwe_id>
```

Example with the CWE id 190, corresponding to the "Reverse Engineer an Executable to Expose Assumed Hidden Functionality or Content" category:
<https://cve.circl.lu//api/capec/190>

5.2.4 The MicroBadger API

MicroBadger is an online service providing a simple but limited way of looking at the content contained in an Official or Community image hosted on Docker Hub. Microbadger provides a simple REST API for inspecting Docker repositories and their metadata. The API follows the same scheme as the Docker Hub API, where "library" is the used namespace for Official repositories, while usernames are used for Community repositories. The Microbadger API was therefore integrated easily in the DAZER software, in order to retrieve extra valuable information which might be missing from the version 2 of Docker Hub's API. Note that Microbadger's API only contained one single call which is illustrated below and originated in the endpoint located at <https://api.microbadger.com>.

```
GET /v1/images/<repository_name>
```

Example with the Official "Ubuntu" repository:
<https://api.microbadger.com/v1/images/library/ubuntu>

Example with the Community "Amazon-ecs-agent" repository:
<https://api.microbadger.com/v1/images/amazon/amazon-ecs-agent>

5.2.5 The Red Hat security data API

Some of the repositories on Docker Hub are based on a CentOS or Red Hat distribution, which disclose vulnerabilities using their own RHSA numbers, as described in 2.1.2. Thus, the Red Hat security data REST API was utilized to translate vulnerabilities identified with a RHSA ID into CVE and CWE numbers, so that such vulnerabilities could be a part of the final measurements and analysis. Note that the endpoint for the Red Hat security data API was located at <https://access.redhat.com/labs/securitydataapi/cve.json?>.

```
GET /advisory=<rhsa_id>
```

Example with a Linux kernel vulnerability identified with "RHSA-2016:1847":
<https://access.redhat.com/labs/securitydataapi/cve.json?advisory=RHSA-2016:1847>

5.2.6 Enterprise Linux Security Advisory

As explained in 2.1.3, Enterprise Linux Security Advisory (ELSA) is a reference-method for vulnerabilities and exposures provided by Oracle Inc. The metadata for retrieving ELSA vulnerabilities may be fetched using an API called Unbreakable Linux Network (ULN) API. However, due to time constraints, vulnerability information from Oracle Linux Docker images were found manually by doing Google searches of the ELSA ID. This was because the setup of their script

was to some extent tedious and required the user to register an Oracle account before utilizing the API. Additionally, there were only a few of the analyzed images (less than five images), which were based on an Oracle Linux distribution, making this manual process relatively manageable.

5.3 Implemented architecture

The OpenStack platform hosted at OsloMet, was utilized to deploy the implemented model consisting of a Python software under the name Docker imAge analyZER (DAZER) discussed in details in 5.4. Up to five Ubuntu 16.04 VMs with 8 VCPUs, 16GB of RAM and 160GB of storage were used to run each type of experiment corresponding the different repository types being part of the study: Official, Verified, Certified, and Community repositories.

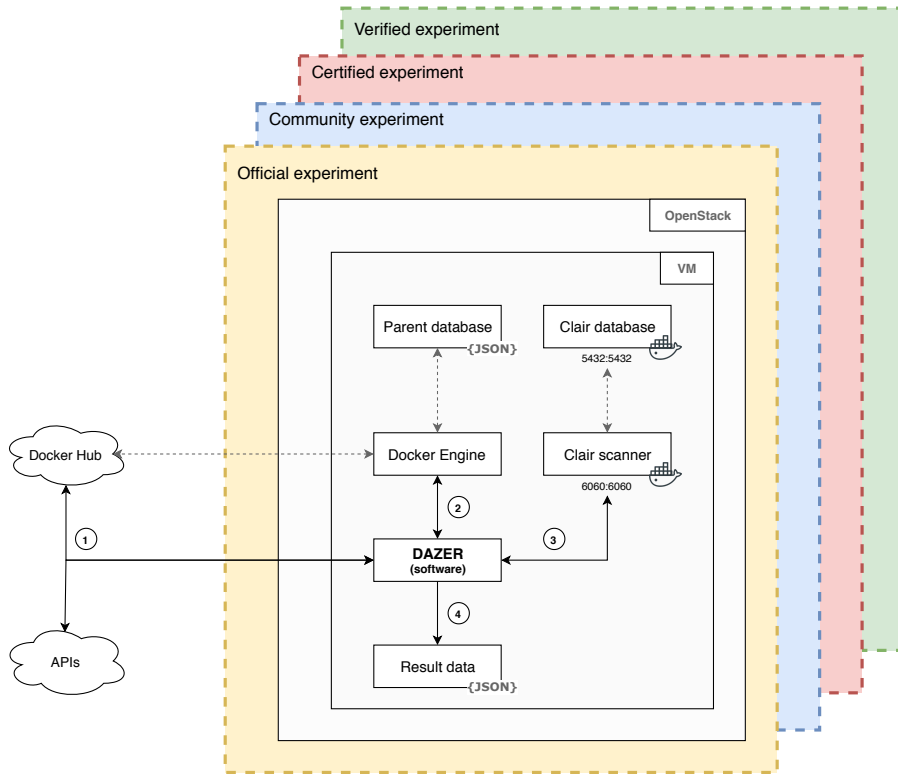


Figure 5.1: The implemented architecture

Figure 5.1 above shows the implemented architecture used to deploy DAZER on a single VM. The implemented software was actually deployed using several replicas of the same VM running in OsloMet’s own OpenStack environment for optimization purposes. Each VM was comprised of the Docker engine and an access to the outside world in order to communicate with the Docker Hub platform and other APIs discussed in 5.2.

Moreover, a containerized version of Clair scanner and a containerized Clair database were deployed alongside the measuring software and the Docker engine running inside each VM, in order to provide DAZER with Docker image vulnerability scanning capabilities. Note that the

Clair services only communicated with each other and the deployed DAZER software in order to retrieve an image's vulnerability information and never communicated with the outside world.

Finally, a preliminary parent database generated by DAZER through the Docker engine during its initialization phase was located aside the other components in order to successfully retrieved images' parents when applicable.

It should be noted that the implemented architecture was similar to the one that had been planned in 3.3.2, although multiple replicas of the same VM were actually used to speed up the conducted experiments, whereas only one single instance had been planned on being used. It is also important to note that a script available in Appendix B.1 was used in order to automatically setup VMs with DAZER's requirements such as the installation of Clair scanner and database.

5.4 Implemented workflow

There are four types of repositories on Docker Hub: Official, Certified, Verified, and Community repositories. When starting the implemented DAZER software consisting of a Python bundle available at <https://github.com/dockalyzer/dazer>, the user is prompted to choose which of the desired type of repository the program should run. The following command below will start the software.

```
python3 main.py <repository_type> [x_images]
```

Note that the most recent image in all the repositories of the specified type are analyzed by default. Moreover, a required number of images to be analyzed needs to be specified when conducting an experiment for Community images (*x_images*), as analyzing over two million repositories is out of this thesis' scope.

When running the program for the first time, DAZER prepares and ensures that all the preliminary requirements are in place as followed before pursuing further:

- Python3 is installed (Python 3.6.x is recommended)
- The Docker engine is installed (Version 18.09.x is recommended)
- A containerized instance of Clair scanner is running
- A containerized instance of the Clair vulnerability database is running
- Valid Docker Hub credentials are provided in the appropriate file
- Python 3 packages required by DAZER are installed

The above requirements are also described on the publicly accessible Github repository dedicated to the DAZER software, available at <https://github.com/dockalyzer/dazer>. Note also that DAZER will notify its user if one of those requirements is not satisfied.

Once the DAZER program is started and all of its requirements are fulfilled, the user will see the output below displayed in the terminal. If the program stops running, the script will notice and handle the exception(s) automatically.

```
DAZERing ...
```

```
Note: see the log file for more information.
```

DAZER does not show the underlying events and actions undertaken by the software to the user, but rather logs its complete workflow to a dedicated file for debugging or other analysis aspirations. The very first lines of the log file are demonstrated below and shorten where needed for demonstration purposes.

```
2019-04-07 18:12:44 Updating Official parent database ...
...
2019-04-07 18:22:32 Checking perl ...
2019-04-07 18:22:37 Repository up to date
2019-04-07 18:22:40 Checking geonetwork ...
2019-04-07 18:22:44 Images have emerged or have been updated in the repository:
geonetwork
2019-04-07 18:22:50 Updating image's timestamp: geonetwork:3.6.0-postgres
2019-04-07 18:23:05 Updating image's timestamp: geonetwork:latest
...
2019-04-07 18:40:40 Images have emerged or have been updated in the repository:
wordpress
2019-04-07 18:40:58 Updating image: wordpress:cli-php7.3
2019-04-07 18:41:17 Updating image: wordpress:cli-php7.2
```

As illustrated by the shortened extract of the log file above, the retrieval of images' metadata starts after the parent database has been fully updated, which is where DAZER's workflow actually starts, as illustrated in figure 5.2 below. Many repositories available on Docker Hub have not been included in this study due to various reasons explained in details in 4.1.2. Compared to the planned workflow illustrated in figure 3.5 under chapter 3, the implemented workflow has now two threads running simultaneously: a downloading thread and a main thread. The downloading thread, which runs in the background, is responsible for downloading multiple images serially and continuously. However, to avoid reaching a full disk space usage, the downloading thread may suspend itself temporarily when the number of downloaded images makes up more than 60% of the available disk space on the machine running the software. Furthermore, note that the downloading thread stops once all the images that should be analyzed have been downloaded.

Simultaneously, the main thread periodically checks for new downloaded images, automatically collecting all of their metadata, parental, and vulnerability information. Once a new batch of downloaded images have been analyzed, all of their gathered information are further exported to the analysis and vulnerability JSON files discussed in details in 4.4. Finally, the main thread is responsible for deleting the images it analyzes during a scanning iteration, in order to make space for new ones to be retrieved by the downloading thread running in the background. The main thread ends when it detects that no new images need to be analyzed and that the downloading thread has ended its execution due to the complete download of all the images which are part of the current experiment.

The use of two simultaneous threads instead of a single one handling both image downloads and analysis allows optimizing the experimental process by reducing the time spent for analyzing new images and exporting their metadata, parental and vulnerability information to the JSON files, while reducing waiting times between downloads to the bare minimum.

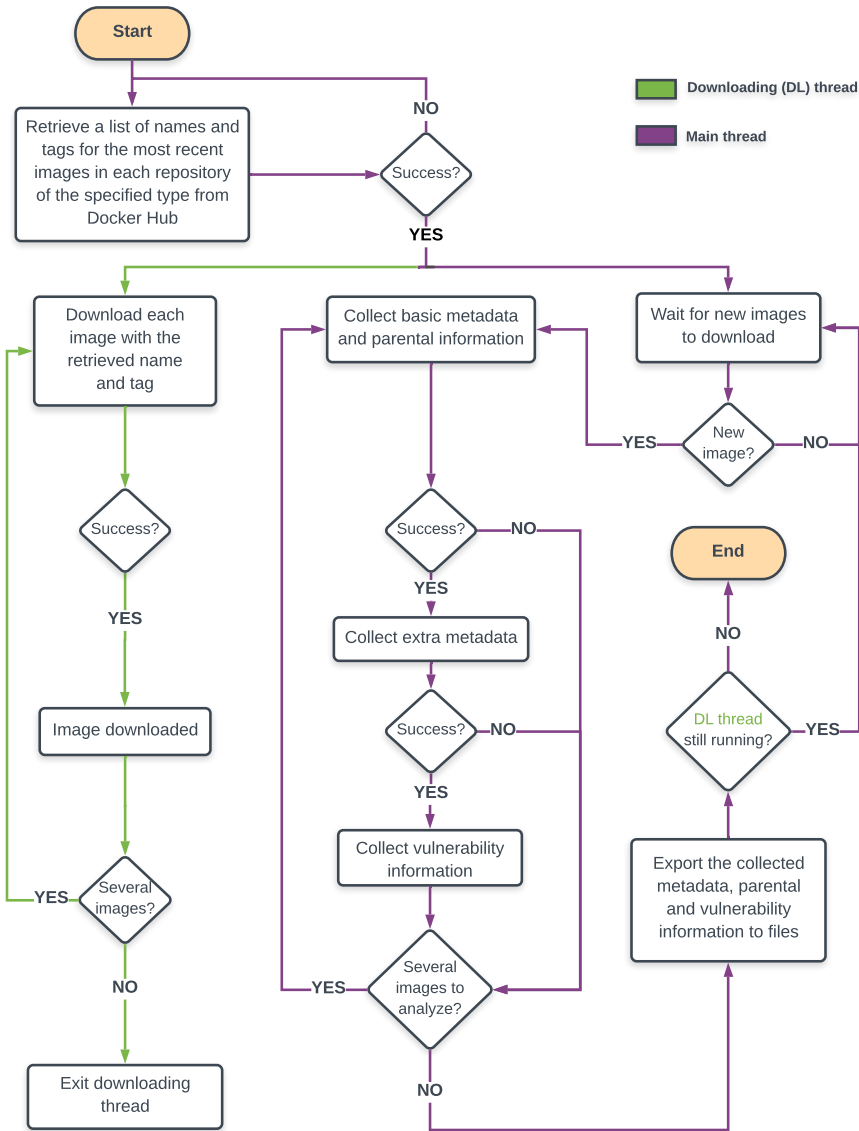


Figure 5.2: The DAZER software's workflow

5.5 Getting ready for analysis

Once all the metadata, parental and vulnerability information about all the different types of Docker images being part of the experiments have been collected, the result data may be imported to a Mongo database for further analysis via research queries defined in this section.

5.5.1 Importing result data to MongoDB

Before importing the result to the Mongo database, the latter needs to be configured to allow connections from the local host, as the same machine was used to hold and access the database. Once MongoDB is up and running, the following commands were used to import the result data in the form of an analysis and vulnerabilities JSON file described in details in [4.4](#).

```
mongoimport --host <local_ip>:27017 --db <db_name> --collection  
<analysis_collection_name> --file analysis_yyyy-mm-dd_hh-mm-ss.json --jsonArray  
  
mongoimport --host <local_ip>:27017 --db <db_name> --collection  
<vulnerabilities_collection_name> --file  
vulnerabilities_yyyy-mm-dd_hh-mm-ss.json --jsonArray
```

As demonstrated above, the JSON files are separated into two different noSQL collections, which correspond to tables in a usual SQL database. Indeed, collections hold so called documents, corresponding to a JSON object such as an analyzed Docker image or a vulnerability in this case. Using two separate documents allowed therefore separating the collected data logically, while still gathering everything in one single database for further analysis using research queries.

Finally, note that each set of result data from each type of experiment (Certified, Verified, Official, Community) needed to be imported on its own using the methodology described above.

5.5.2 Research queries

As discussed in [1.2](#), this thesis' problem statement consists of three main research questions implying many detailed research questions debated in [4.5](#). A part of the implementation phase of our methodology consists therefore of translating those detailed questions into so called research queries.

Although most of the research queries consist of a single line of code, some of the detailed research questions cannot be directly translated into a single query. Thus, JavaScript code along with Mongo queries are utilized to fully develop some of the detailed research questions. Note that such research queries are often too long to be included here and are therefore fully available in [Appendix C](#). As an example however, some of the shorter research queries matching their textual detailed and main research questions discussed in details in [4.5](#) and [1.2](#) are available as followed.

RQ1: Have the security measures introduced by Docker Inc. in response to previous research improved Docker Hub's security landscape and to what extent?

The first main research question consists of several detailed research questions defined in section [4.5](#), which only apply to Certified and Verified images. Those detailed research questions cannot be directly translated into research queries, as they essentially consist of comparing and identifying correlations between the results of RQ2 and RQ3 for Certified and Verified images with other types of images.

RQ2: Are vulnerabilities still inherited from images' parent(s) and in what proportion?

RQ2.1: What proportion of images depends on a parent?

```
db.data.find(
  { $and: [ { "type": { "$eq": image_type } }, { "parent": { "$ne" : "" } } ] },
  { _id: 0, name: 1, parent: 1 }).count();
```

RQ3: How are discovered vulnerabilities distributed across repository types?

RQ3.1: What proportion of images contains no vulnerabilities?

```
db.data.find(
  { $and: [ { "type": { "$eq": image_type } }, { "total_vulnerabilities":
    { "$eq" : 0 } } ] },
  { _id: 0, name: 1, vulnerabilities: 1 }).count();
```

RQ3.2: What proportion of images contains at least one vulnerability?

```
db.data.find(
  { $and: [ { "type": { "$eq": image_type } }, { "total_vulnerabilities":
    { "$ne" : 0 } } ] },
  { _id: 0, name: 1, vulnerabilities: 1 } ).count();
```

The following two research queries involve some manual operations. Unlike the find() function in MongoDB, the aggregation() function does not support retrieving the value of a property. As an example, the average number of vulnerabilities contained in an image was therefore calculated using the total number of discovered vulnerabilities obtained with the research query above, divided by the total number of analyzed images.

RQ3.3: How many vulnerabilities do images contain in average?

```
db.data.aggregate(
  { $group: { _id: "$type", sumTotalVulnerabilities: { $sum:
    "$total_vulnerabilities" } } },
  { $project: { "_id": 0, "sumTotalVulnerabilities": 1 } }
); // Average to be calculated manually
```

RQ3.4: How many unique vulnerabilities do images contain in total?

```
db.vuln.aggregate([
  { $group : { _id: { cve_number : { $gt: ["$cve_number", null] } },
    count : { $sum : 1 } } }
]); // Average to be calculated manually
```


RQ3.5: How are unique vulnerabilities distributed per severity among images?

```
db.vuln.find({"severity": {"$eq" : "Critical"}} , { _id: 0, severity: 1 }).count();
db.vuln.find({"severity": {"$eq" : "High"}} , { _id: 0, severity: 1 }).count();
db.vuln.find({"severity": {"$eq" : "Medium"}} , { _id: 0, severity: 1 }).count();
db.vuln.find({"severity": {"$eq" : "Low"}} , { _id: 0, severity: 1 }).count();
db.vuln.find({"severity": {"$eq" : "Negligible"}} , { _id: 0, severity: 1 }).count();
db.vuln.find({"severity": {"$eq" : "Unknown"}} , { _id: 0, severity: 1 }).count();
```

RQ3.7: Is there a correlation between the most popular images (most pulled) and the most vulnerable ones?

```
db.data.find(
  { $and: [ {"type": {"$eq": image_type } }, {"name": {"$ne" : ""} } ] },
  { _id: 0, name: 1, total_pulled: 1, total_vulnerabilities: 1 }).
  sort({total_pulled:-1}).limit(10);

db.data.find(
  { $and: [ {"type": {"$eq": image_type }}, {"name": {"$ne" : ""} } ] },
  { _id: 0, name: 1, total_vulnerabilities: 1 }).
  sort({total_vulnerabilities:-1}).limit(10);
```

RQ3.10: What are the top 10 vulnerabilities (across all types of repositories)?

```
db.data.aggregate([
  { $match: { type: image_type } },
  { $unwind: "$vulnerabilities" },
  {$group: { _id: { id: "$vulnerabilities"}, count: {$sum : 1} } },
  { $sort: { count: -1 } }, { $limit: 10 } ]]);
```

Finally, note that the above queries and the rest of them available in [Appendix C](#) were executed against the database by storing them into a Javascript file run via the Mongo shell, as it was a convenient way of mining the result data and answer the detailed research questions.

Chapter 6

Result 3: Measurements

This chapter exhibits the data from all the conducted experiments using the implemented software in phase 2 of our methodology known as the Docker imAge analyZER (DAZER) software discussed in details in chapter 5. The measurement phase of the methodology intends to make use of the noSQL research queries implemented in phase 2 and detailed in 5.5.2, in order to answer the detailed research questions developed in 4.5 and ultimately address the original problem statement defined in 1.2. Note that the first main research question (abbreviated RQ1) making up the problem statement consists of several detailed research questions which only apply to Certified and Verified repositories. Furthermore, it should be noticed that only the main research questions composing the original problem statement will be reminded here, as the detailed research questions they imply were mostly utilized to produce the tables and charts discussed throughout this chapter. Note however that the complete list of detailed research question developed in chapter 4 is available in 4.5.

As a reminder, the last phase of this thesis' methodology (phase 3) described in 3.1 consists of the following steps:

1. Conduct an experiment for each set of Docker images defined in phase 1 (design phase)
2. Import the gathered metadata, vulnerability and parental information of each set into a NoSQL database for analysis
3. Make use of the NoSQL queries implemented in phase 2 in order to answer the problem statement

6.1 Describing the results

All the experiments were conducted between March 31st and April 14th 2019 included. Before the conduction of each experiment, the relevant Official and/or Verified parent databases were fully updated in order to guarantee that they contained the layer signatures of all the Official and/or Verified images available on Docker Hub at that particular time.

As previously mentioned in 5.3, up to four individual VMs running in OsloMet's OpenStack cloud were used to run all the experiments gathering metadata, parental and vulnerability information

for all types of repositories. The Community experiment took approximately 141 hours to finish, while the Official measurement took roughly 16 hours, followed by the Verified experiment which took 14,5 hours and the Certified measurement which lasted for almost an hour. Thus, the time spent on each experiment greatly varied from one experiment to another due to the number of images involved in each experiment, as well the size of those images, which could reach up to almost 30 GB for certain Verified images. Moreover, it is important to note that the network speed as well as the actual resources available for the physical server(s) hosting the VMs at the time of the conduction of our experiments might have played a role in the total amount of time used to complete each measurement.

The following four pie charts as well as the summary table located below show the distribution of base, non-base and skipped images across the four types of analyzed repositories. Note that since only one image (i.e. the most recent one) was analyzed for each repository being part of the conducted experiments, the terms "image" and "repository" may be used interchangeably in this chapter.

| | Initial number of repositories | Total analyzed repositories | Base repositories | Non-base repositories | Skipped repositories | Analysis rate |
|-----------|-----------------------------------|--------------------------------|----------------------|--------------------------|-------------------------|------------------|
| Official | 151 | 128 | 11 | 117 | 23 | 84,77% |
| Community | 500 | 500 | 266 | 234 | 0 | 100,00% |
| Verified | 208 | 98 | 33 | 65 | 110 | 47,12% |
| Certified | 44 | 31 | 19 | 12 | 13 | 70,45% |
| Total | 903 | 757 | 329 | 428 | 146 | 83,83% |

Table 6.1: A summary of the experiments performed in this study

Analyzed Official repositories

During the conduction of our Official experiment, there were 11 Official base repositories on Docker Hub, constituting over 7% of the total amount of that type of repositories on the platform, as illustrated in figure 6.1 below. Furthermore, 117 non-base repositories were identified, implying that almost 80% of all the Official repositories contained images which were based on another Official image. Finally, note that the number of skipped repositories is minimal, as it only concerns 23 repositories out the 151, making up only 15% of the total amount.

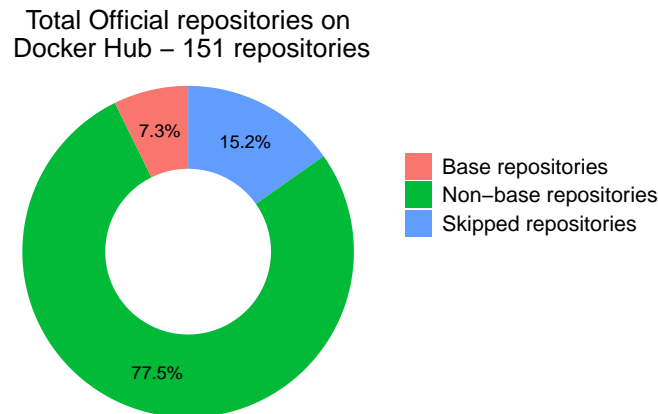


Figure 6.1: Analyzed Official repositories distribution

Analyzed Community repositories

Contrary to its three other peers, the Community experiment revealed that over 50% of its analyzed repositories were base repositories, as illustrated in figure 6.2. Indeed, 266 out of the 500 analyzed repositories were base ones, whereas only 234 contained images based on another Official, Certified or Verified image, making up less than 30% of the total amount of analyzed Community repositories. Note that those numbers are surprisingly high due to a significant error caused by many Community images having a non-identifiable parent, as discussed in details in the discussion chapter under 8.2. Finally, it should be noticed that contrary to the three other types of experiments, the Community measurement consisted of analyzing a fixed number of 500 repositories among the most popular ones, justifying that no repositories were skipped.

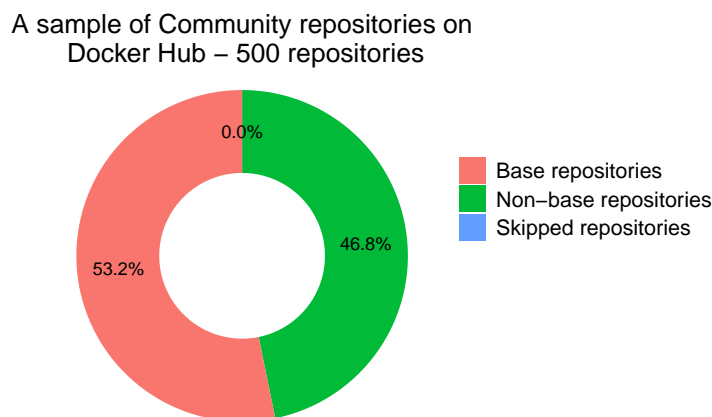


Figure 6.2: Analyzed Community repositories distribution

Analyzed Verified repositories

During the conduction of the Verified experiment, over three times as many repositories were skipped compared to the Official measurement, making up almost 53% of the total number of Verified repositories available on Docker Hub, as shown in figure 6.3. As explained in chapter 4 under 4.1.2, the reason for such a high number is due to the significant presence of the Microsoft publisher among Verified repositories, which provides incompatible images for the Linux platform or x86-64 architecture in many of its repositories, as well as offering non-downloadable images with a missing or corrupted manifest file. Nonetheless, 33 out of the 208 available repositories were base ones making up almost 16% of the total amount of Verified repositories, whereas 31% were non-based ones with a total analysis rate right below 50%, as illustrated in table 6.1 above. Finally, note that more than a third of all the Verified repositories available on Docker Hub contained images based on another Official or Verified image.

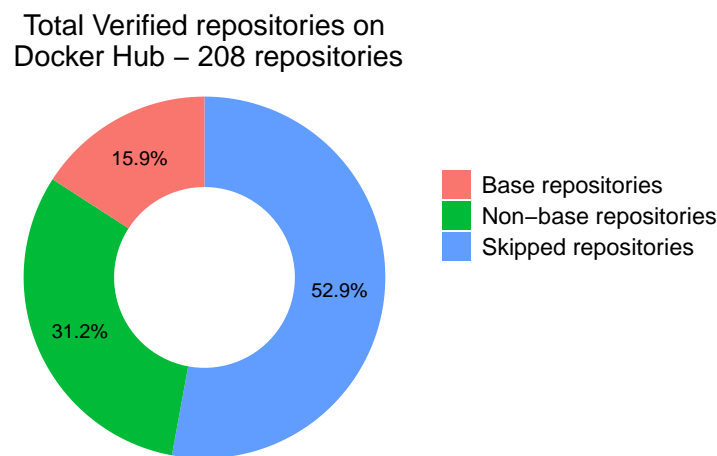


Figure 6.3: Analyzed Verified repositories distribution

Analyzed Certified repositories

Since Certified repositories are a special type of Verified repositories, the number of skipped repositories during its measurement was consequently significant with 13 out of the 44 available repositories, making up more than 29% of the total number of repositories on Docker Hub, as illustrated in figure 6.4 below. Nevertheless, 19 out of the 44 repositories were base ones, making up more than 43% of all the Certified repositories on Docker Hub, whereas more than 27% were non-base ones with 12 repositories out of 44. Finally, note that similarly to Verified repositories, almost a third of all the Certified repositories contained images based on another Official or Verified image.

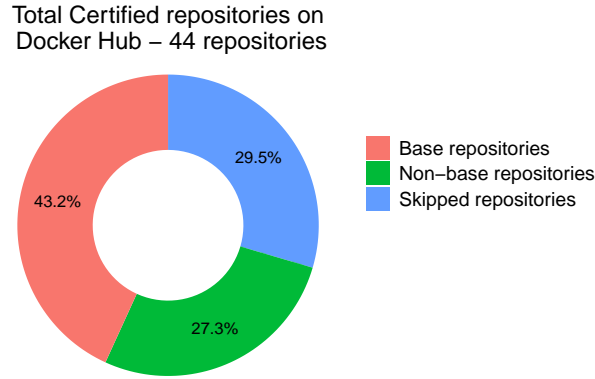


Figure 6.4: Analyzed Certified repositories distribution

To sum up, this section covered some general statistical information for all the repository types analyzed from Docker Hub. The following three next sections will address the original main research questions making up the problem statement introduced in 1.2, by answering their implied detailed research questions using the developed MongoDB queries discussed in 5.5.2. Note that the three main research questions (abbreviated RQ) will be answered in reverse order starting with RQ3, as the latter and RQ2 contain detailed research inquiries which will be later utilized to fully answer RQ1 in 6.4.

6.2 RQ3: Vulnerability distribution across repository types

Main research question 3: *How are discovered vulnerabilities distributed across repository types?*

6.2.1 Quantitative vulnerability distribution

First, the distribution of vulnerability-free images across repository types is very surprising, as illustrated in table 6.2 below. Indeed, almost 23% of all the analyzed Official images contain no vulnerabilities, whereas almost 19% and 16% are respectively in that case for Community and Verified repositories. Certified repositories are therefore performing worse when it comes to providing vulnerability free content, as only 10% of them do not contain any vulnerability.

Secondly, there is a significant difference in the total number of vulnerabilities found for each type of repository, as Certified images contain only 921 vulnerabilities in total, compared to 15 342 and 22 683 for Verified and Official repositories respectively. As discussed in our expectations in 3.5, Certified repositories are therefore the most secure when it comes to the total number of vulnerabilities found in their images, while Community repositories are performing worse with more than 76 673 vulnerabilities in total. Indeed, although less Certified images are completely free of vulnerabilities compared to the three other types of repositories, they contain far less vulnerabilities than their other peers in total. Moreover, it should be noticed that despite their maintenance by Docker Inc.'s dedicated team and trusted vendors, Official and Verified

repositories contain a large number of vulnerabilities in total, which constitutes therefore an interesting but alarming discovery.

| Quantitative vulnerability distribution across repository types | | | | | |
|---|--------------------------------|---------------------------------|-------------------------------|--------------------------------|---------------------------|
| | Official (128 repositories) | Community (500 repositories) | Verified (98 repositories) | Certified (31 repositories) | All (757 repositories) |
| No vulnerabilities | 29 (22,66%) | 94 (18,80%) | 16 (16,33%) | 3 (9,68%) | 142 (18,76%) |
| At least one vulnerability | 99 (77,34%) | 406 (81,20%) | 82 (83,67%) | 28 (90,32%) | 615 (81,24%) |
| Total number of vulnerabilities | 22 683 | 76 673 | 15 342 | 921 | 115 619 |
| Unique number of vulnerabilities | 2 304 | 5 095 | 2 586 | 353 | 5449 |

Table 6.2: Quantitative vulnerability distribution across repository types

Thirdly, the study of unique vulnerabilities for each of the analyzed repository type may help understand whether certain types of repositories contain many redundant vulnerabilities. Certified images have the highest percentage of unique vulnerabilities, making up 38% of their total amount of vulnerabilities, whereas Verified, Official and Community images have respectively 17%, 10% and almost 7% of their total amount of vulnerabilities being unique. Apart from Certified repositories where more than one vulnerability out of three is unique, such results suggest that the three other types of repositories offer images using many identical packages, leading to the vast majority of their vulnerabilities being duplicates.

Fourthly, the yearly distribution of unique vulnerabilities per repository type illustrated in figure 6.5 below, shows that the total number of reported vulnerabilities for all types of repositories started to grow in 2013, reaching an all-time high in 2017. It should be noticed that although the Docker technology did not exist prior to 2013, some Docker images contain software and libraries with disclosed vulnerabilities prior to that date. Indeed, vulnerabilities follow a certain format containing the exact year of their disclosure, as explained in details in 5.2.3 and 5.2.5. For example, vulnerabilities disclosed through the CVE Numbering Authorities (CNAs) or the Red Hat company use the following format respectively: `CVE-yyyy-xxxxx`, `RHSA-yyyy:xxxx`. The `y` portion indicates the year of a vulnerability's public disclosure, while the `x` portion corresponds to a unique identifier for the vulnerability. The yearly distribution of unique vulnerabilities across repositories types was therefore created by taking advantage of the format used by disclosed vulnerabilities. Note that although the total number of unique vulnerabilities across all the repository types was lower in 2018 compared to 2017, it was still higher than in 2016. Furthermore, certain vulnerabilities appearing in late 2018 may still be missing a CVE number, as they may have not been disclosed yet in order to give their vendor(s) enough time to release a patch before making them public. Some vulnerabilities from 2018 found in the set of analyzed Docker images may therefore be missing from our analysis. Note also that the number of vulnerabilities disclosed in 2019 is very limited due to the early conduction of the experiments during that same year (between March 31st and April 14th 2019).

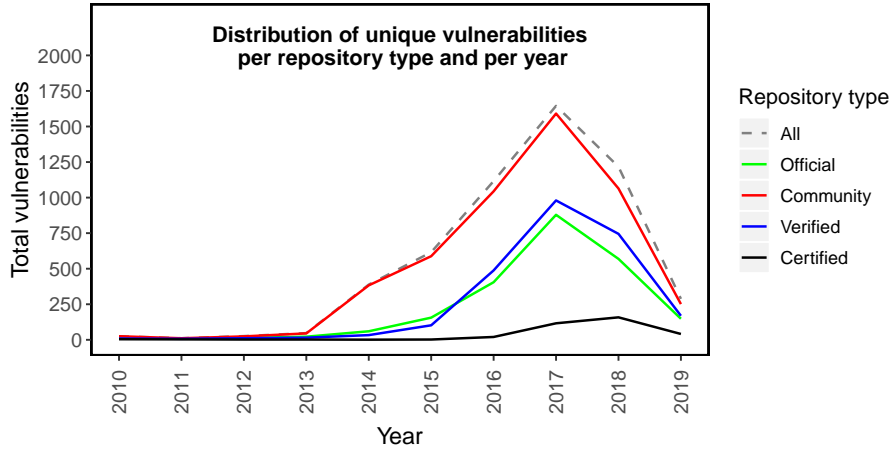


Figure 6.5: Distribution of unique vulnerabilities per repository type and per year

Finally, the all-time high number of uniquely reported vulnerabilities in 2017 is very characteristic of the vulnerability distribution across the different types of repositories. Indeed, similarly to their total number of unique vulnerabilities across all years illustrated in table 6.2 and discussed above, Community images contained the highest number of unique vulnerabilities in 2017 with over 1500 vulnerabilities, followed by Verified, Official, and Certified images which held approximately 1000, 900, and 200 unique vulnerabilities respectively. Note however that a high number of vulnerabilities is not enough to draw conclusions about a certain type of repository, as vulnerabilities have different severity levels based on their level of exploitation difficulties, as well as the consequences following an exploitation.

6.2.2 Severity distribution

As briefly mentioned at the end of the previous subsection, a purely quantitative indicator such as the total number of unique vulnerabilities found in a given repository type is not enough to draw any conclusion about its security. Indeed, a more qualitative indicator such as vulnerabilities' severity levels may help confirm or relativize the findings discussed in 6.2.1.

Figure 6.6 below illustrates the severity distribution of all the uniquely discovered vulnerabilities across all repository types, using six different severity levels derived from NVD's CVSS scores discussed in 2.1.4 and identified as "Negligible", "Low", "Medium", "High", "Critical", and "Unknown". Note that such levels were provided by Clair scanner and that a "Negligible" severity corresponds to the "None" level in NVD's categorization, while "Unknown" severities simply consist of vulnerabilities which have not been assigned a severity level yet.

As illustrated below, the large majority of unique vulnerabilities across all types of repositories has a "Medium" level of severity. Indeed, as many as 3 156 unique vulnerabilities, representing 60% of the total amount of unique vulnerabilities found in the analyzed set of Docker images have a "Medium" level of severity. As a reminder from 2.1.4, vulnerabilities with a "Medium" level of severity provide limited access to an attacker once exploited and usually involve social engineering

techniques or being on the same local network as the victim. The fact that most vulnerabilities across all types of repositories have a "Medium" level of severity is relatively concerning, although not all of them may be exploitable as argued in 8.1 as part of the discussion chapter.

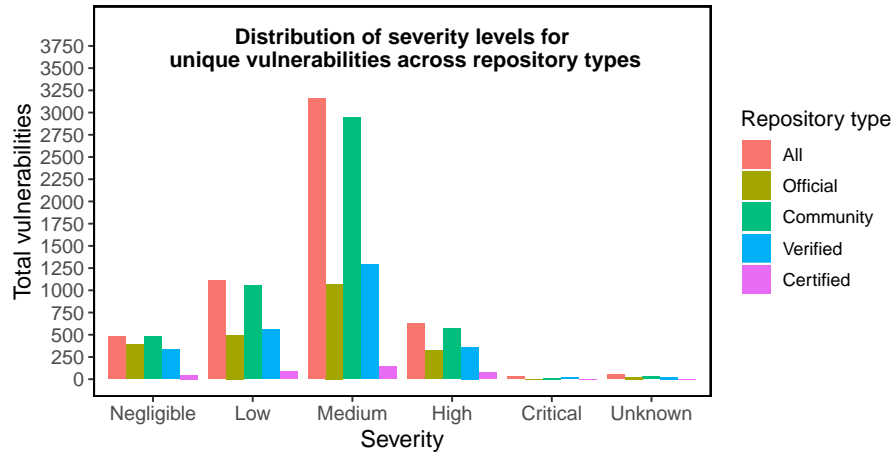


Figure 6.6: Distribution of severity levels for unique vulnerabilities across repository types

Based on table 6.3 located below and providing a statistical representation of the same severity distribution as the one illustrated in figure 6.6, it may be observed that the analyzed Community images contain almost all the unique vulnerabilities identified during all the experiments, while Official and Verified images contain about 50% of them. As anticipated in our expectations and discussed in 3.5, Certified repositories offer images with the most secure content of all three, as they only contain 8% of all the unique vulnerabilities discovered across all types of repositories.

Moreover, the share of unique vulnerabilities found in the analyzed Certified images for all levels of severity illustrated in figure 6.6 above is significantly lower than the ones discovered for the three other types of images, which confirms our findings from 6.2.1 showing that Certified images contain less unique vulnerabilities in average than the three other types of repositories. It is important to note that all the repository types mostly hold vulnerabilities with a "Medium" severity, consisting of almost 50% of all the unique vulnerabilities identified in Verified repositories, while the proportions for Official, Community, and Certified repositories are 46,48%, 57,76%, and 36,91% respectively (see table 6.3 below).

As for the "Critical" and "Unknown" severity levels combined, the proportions of all the repository types with such vulnerabilities are below 2% of all the analyzed images, which is somewhat reassuring as the number of unique vulnerabilities with a critical severity across all types of repositories is therefore very limited. Additionally, table 6.3 below illustrates that all the analyzed images have more or less 20% of all their unique vulnerabilities with a "Low" severity, suggesting that any image contains 1 vulnerability out of 5 carrying a very minimal risk requiring local or physical access for successful exploitation (more details about vulnerabilities of "Low" severities in 2.1.4).

| | Official | Community | Verified | Certified | Total unique |
|-------------------|----------------|----------------|---------------|---------------|----------------|
| Negligible | 16,88% (389) | 9,38% (478) | 13,03% (337) | 13,31% (47) | 9,05% (493) |
| Low | 21,62% (498) | 20,69% (1054) | 21,50% (556) | 23,80% (84) | 20,59% (1122) |
| Medium | 46,48% (1071) | 57,76% (2943) | 49,88% (1290) | 39,66% (140) | 57,86% (3153) |
| High | 13,93% (321) | 11,25% (573) | 14,00% (362) | 21,81% (77) | 11,30% (616) |
| Critical | 0,00% (0) | 0,27% (14) | 0,70% (18) | 0,85% (3) | 0,50% (27) |
| Unknown | 1,09% (25) | 0,65% (33) | 0,89% (23) | 0,57% (2) | 0,70% (38) |
| Total | 100,00% (2304) | 100,00% (5095) | 100% (2586) | 100,00% (353) | 100,00% (5449) |

Table 6.3: Distribution of severity levels for unique vulnerabilities across repository types

Finally, note that over 20% of all the unique vulnerabilities found in Certified images have a "High" severity level, whereas that proportion for the three other types of repositories varies between 11 and 14%. Thus, although Certified images contain less unique vulnerabilities than their peers in total as explained above and in 6.2.1, their vulnerabilities tend to be of higher level. The qualitative indicator that is vulnerabilities' severity levels allows therefore refuting the original result direction suggesting that Certified images were the most secure due to the lower number of vulnerabilities they hold compared to the other types of repositories. Nonetheless, it is important to note that the distribution of severity levels for unique vulnerabilities across all types of repositories does not indicate the actual proportion of images containing at least one vulnerability with a particular severity level.

6.2.3 Vulnerable image distribution

As briefly mentioned at the end of the previous subsection, the distribution of severity levels for unique vulnerabilities across Community, Official, Verified and Certified repositories does not provide any indication on the actual proportion of images containing at least one vulnerability with a particular severity level. Indeed, while only 3 unique vulnerabilities have a critical severity level for Certified images as shown in table 6.3 above, their total number including duplicates may be a lot larger and spread out over many Certified images, or contained within a single one. Figure 6.7 and 6.8 below illustrates the quantitative distribution of images across repository types containing at least one critical, high, medium or low vulnerability.

First, it may be observed on the left-hand side of figure 6.7 that although the number of Community images containing at least one critical severity is more important than the number of images in the same case in other types of repositories, their proportion to their total amount of images is actually the lowest after Official repositories. Indeed, under 3% of the Community images hold at least one critical severity, whereas 4% and 16% are in that case for Verified and Certified images respectively. As a reminder from 2.1.4, critical vulnerabilities are usually easy and straight-forward to exploit, while mostly leading to root-level privileges. Thus, a large proportion of Certified images contain at least one critical severity, which confirms our findings discussed in 6.2.2 showing that although Certified images contain less unique vulnerabilities than their peers in total, their vulnerabilities tend to be of higher level, which is very alarming. Moreover, it was pointed out in the previous subsection that the number of unique vulnerabilities with a critical severity across all types of repositories is very limited, suggesting that many Certified images contain the same critical vulnerabilities. Note also that a non-negligible proportion of

Verified images contain at least one critical vulnerability compared to Community images, which actually perform best among the repository types holding images with critical vulnerabilities. As for the Official images, it should be noticed that none of them contain a critical vulnerability, which is somewhat reassuring as they are by far the most popular type of images on the Docker Hub platform, as discussed further in 6.2.4.

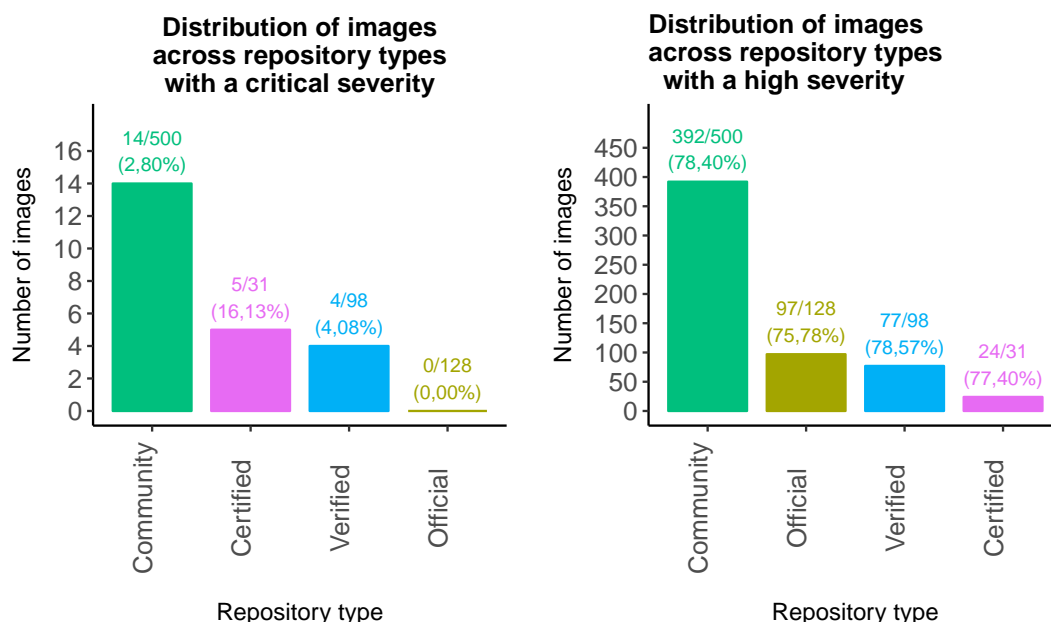


Figure 6.7: Distribution of images across repository types with a critical and high severity

Secondly, the share of images containing at least one vulnerability with a high severity is similar in each type of repository. Indeed, 77% of the Certified images hold at least one vulnerability with a high severity, whereas 76% , 79% and 78% are in that case for Official, Verified and Community images respectively. Thus, although more than 20% of all the unique vulnerabilities found in Certified images have a high severity level whereas that proportion for the three other types of repositories varies between 11 and 14% as explained in 6.2.2, the share of images containing at least one vulnerability with a high severity is similar in each type of repository. Such results suggest therefore that Official, Verified and Community repositories contain a significantly larger amount of duplicate vulnerabilities with a high severity level across their images compared to Certified repositories. As a reminder from 2.1.4, high severities may have severe consequences such as significant data losses or downtime, although they may require advanced techniques in order to be exploited. Thus, a such important proportion of images containing at least one vulnerability with a high severity level is very alarming, as more than 3 images out of 4 are in that case for any type of repository.

Thirdly, it may be observed on the left-hand side of figure 6.8 below that the proportion of images containing at least one medium vulnerability is significantly higher for Certified and Verified images than the two older types of repositories. On one hand, as much as 87% of the analyzed Certified images contain at least one vulnerability with a medium severity, while 84% of the Verified images are in that case. On the other hand, 75% of the Community repositories hold at least one medium vulnerability, whereas 73% of the analyzed Official images are in that

case. Although all types of repositories contain a large number of images holding at least one medium severity, Certified and Verified repositories tend to contain several images in that case. As a reminder from 2.1.4, vulnerabilities with a medium severity provide limited access to an attacker once exploited and usually involve social engineering techniques or being on the same local network as the victim. The fact that such a vast majority of images contain at least one vulnerability with a medium severity is therefore relatively concerning.

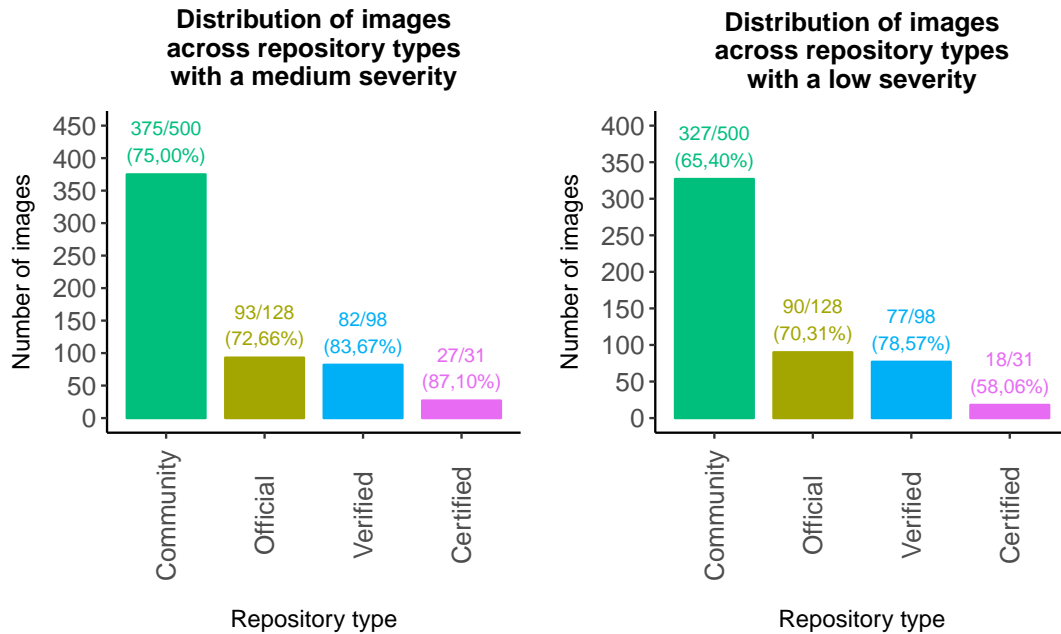


Figure 6.8: Distribution of images across repository types with a medium and low severity

Finally, the share of images containing at least one vulnerability with a low severity is somewhat disparate from one repository type to another. Indeed, Certified repositories perform best with 58% of their images containing at least one low vulnerability, whereas 65% , 70% and 79% of the Community, Official and Verified images are in that case. As a reminder from 2.1.4, vulnerabilities in the low range typically have very little impact on an organization’s business and usually require local or physical access to a vulnerable system. Although it is a good sign that Certified images have the lowest proportion of their total number of images containing at least one low vulnerability compared to the three other types of repositories, it is definitely not enough to claim that they offer a better level of security than their peers. Moreover, it should be noticed that Verified repositories contain the highest share of images with a low vulnerability, although that level of severity does not play a major role in determining the level of security provided by a certain type of image, contrary to the critical, high and medium severity levels discussed above.

6.2.4 Potential correlations

While the previous subsections presented general vulnerability results issued from the six first detailed research questions composing RQ3 and discussed in 4.5, this subsection will address

the remaining ones in view of addressing a part of the original problem statement introduced in 1.2 and provide a better understanding of Docker Hub's the security landscape. Note that the indicators taken into account in this subsection will consist of the number of times an image has been downloaded (referred to as "total pulled"), the last time an image was updated on Docker Hub (referred to as "last updated"), as well as the images containing the highest number of vulnerabilities (referred to as "most vulnerable").

Is there a correlation between the most popular images (most pulled) and the most vulnerable ones?

IMPORTANT - Figure 6.10, 6.11 and 6.12 below contain an x-axis labelled with a shortened version of their repository names due to their length being too important. Note however that the detailed statistics with the exact number of downloads and vulnerabilities for each of the ten most pulled and most vulnerable repositories discussed below are available in Appendix D.2 and D.1 respectively with full repository names.

Many *Official* repositories possess a significant popularity among Docker Hub users. Indeed, the right hand side of figure 6.9 illustrates the ten most popular *Official* repositories available on Docker Hub during the conduction of our experiments in terms of number of downloads, while the figure's left hand side highlights the ten most vulnerable *Official* repositories in terms of number of contained vulnerabilities. Note that the number of vulnerabilities hold in each of the most popular repositories is also present next to the latter's names. On one hand, it may be observed that the "rails" and "django" repositories contain a significantly higher number of unique vulnerabilities with a number as high as 1500, whereas the remaining most vulnerable *Official* repositories hold approximately 600 unique vulnerabilities.

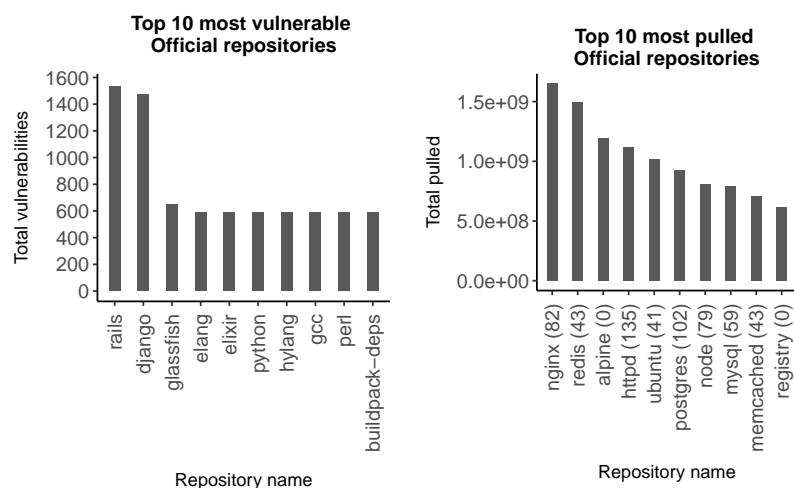


Figure 6.9: The top 10 most vulnerable and most pulled *Official* repositories

On the other hand, the most pulled *Official* repository consists of the "nginx" repository with 1.65 billion downloads, followed by "redis" and "alpine" with 1.49 and 1.19 billion pulls respectively.

Although there are no apparent correlations between the most popular and the most vulnerable Official images, the "httpd" and "postgres" repositories stand out with their significantly higher number of contained vulnerabilities (135 and 102 respectively) compared to the rest of the ten most downloaded Official images. Finally, note that the detailed statistics with the exact number of downloads and vulnerabilities for each of the top ten most pulled and most vulnerable repositories are available in Appendix D.2 and D.1 respectively.

Contrary to their Official peers, all the ten most vulnerable *Community* repositories contain a high number of unique vulnerabilities averaging around 1500, as shown in figure 6.10 below. Indeed, the bottom two most vulnerable Community repositories identified as "herightplace/bedboard2-sidekiq" (A9) and "springcloud/spring-pipeline-m2" (A10) are representative of that characteristics, as they contain the least number of vulnerabilities among the 10 most vulnerable Community repositories and still hold 1277 and 1218 unique vulnerabilities respectively. Surprisingly, the three most popular Community repository identified as "jtarchie/pr" (B1), "pivotalcf/pivnet-resources" (B2) and "cfcommunity/slack-notification-resource" (B3) have a higher number of unique downloads than the most popular Official repository. Indeed, the three repositories possess a total number of downloads of 2.06, 1.95 and 1.74 billion respectively, while Nginx only has 1.6 billion downloads. Furthermore, the number of downloads for the rest of the most popular Community repositories is significantly lower than the top 3, whereas the same indicator decreases almost linearly for the 10 most popular Official repositories illustrated in figure 6.9 above. Finally, note that similarly to Official repositories, there is no apparent correlation between the ten most popular and most vulnerable Community images.

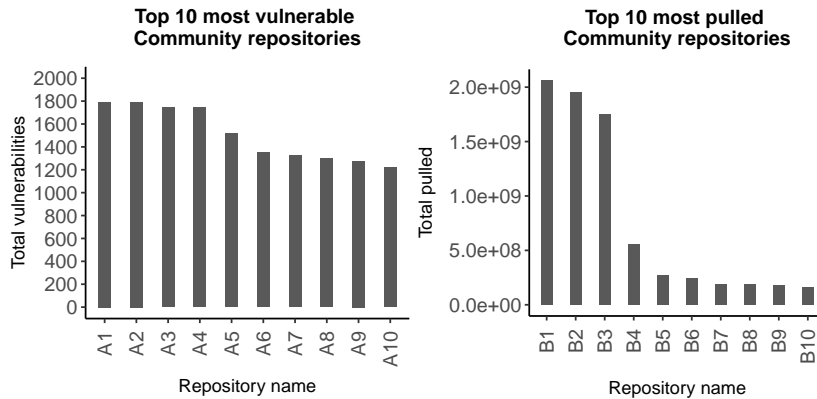


Figure 6.10: The top 10 most vulnerable and most pulled Community repositories

Similarly to Official repositories, the number of vulnerabilities contained in the two most vulnerable *Verified* repositories is rather high, but decreases rapidly for the remaining repositories as illustrated in figure 6.11 below. Indeed, the top 2 most vulnerable Verified repositories in the name of "mcr.microsoft.com/azuredocs/azure-vote-front" (D1) and "tore/klokantech/openmaptiles-server-dev" (D2) contain 1531 and 1171 unique vulnerabilities respectively, whereas the remaining repositories average around 630 uniquely contained vulnerabilities. That indication is therefore comforting our expectations discussed in 3.5, where Official and Verified repositories were expected to have a higher number of vulnerabilities.

ted to offer a similar degree of security. On the other hand, it should be noticed that there exists a huge gap between the number of downloads observed for each of the ten most popular Verified images. Indeed, the most popular image known as "mcr.microsoft.com/dotnet/core/runtime-deps" (E1) possesses more than 348 million downloads, whereas the least popular one in the top 10 identified as "mcr.microsoft.com/azuredocs/aci-tutorial-sidecar" (E10) only has approximately 1 million downloads, which is more than 300 times less pulls. Nonetheless, the three most popular repositories (E1, E2, E3) which possess a significantly higher number of downloads than the rest constitute the most vulnerable Verified repositories of the ten most popular ones with total numbers of unique vulnerabilities ranging from 54 to 104, as detailed in Appendix D.2. That indicator is very concerning, as it shows that the most popular Verified repositories are the most vulnerable among that top 10. Finally, note that similarly to the other types of repository, there is no apparent direct correlation between the ten most popular and most vulnerable Verified images.

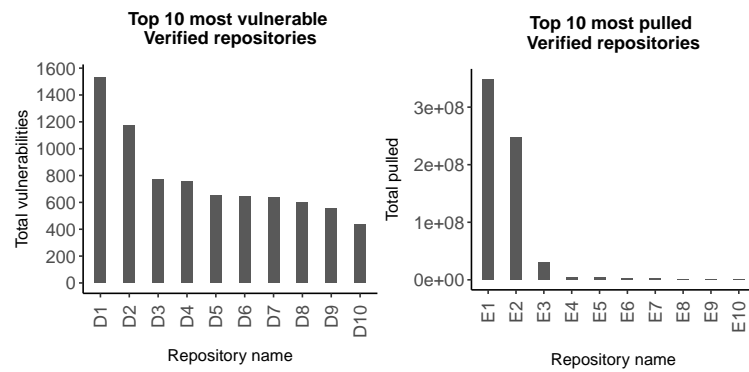


Figure 6.11: The top 10 most vulnerable and most pulled Verified repositories

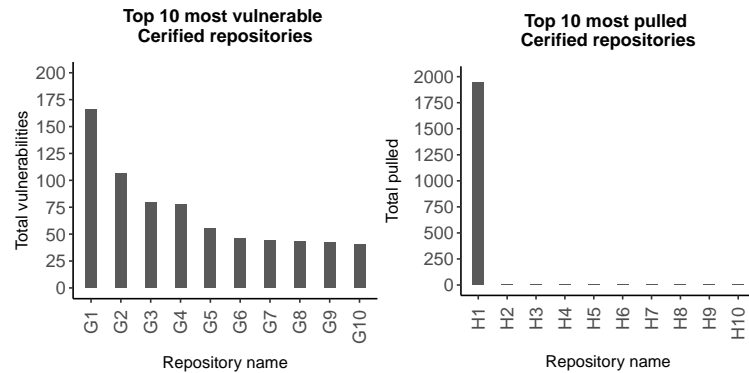


Figure 6.12: The top 10 most vulnerable and most pulled Certified repositories

Similarly to Official and Verified repositories, the most vulnerable *Certified* repository contains a significantly higher number of unique vulnerabilities than the remaining repositories, as illustrated in figure 6.12 above. Indeed, the most vulnerable repository identified as "store/opsan-i/skopos" contains 166 unique vulnerabilities, whereas the remaining Certified repositories contain less than 60 vulnerabilities in average. Nonetheless, the number of unique vulnerabilities contained in the most vulnerable Certified repository is significantly lower than the one contained in the most vulnerable repository of the three other types with only 166 unique vulnerabilities, against 1531 for Verified, 1530 for Official and 1792 for the most vulnerable Community repository. The most vulnerable Certified repositories offer therefore a higher degree of security compared to its three other peers, with a significantly lower number of uniquely contained vulnerabilities. Finally, note that almost all Certified repositories do not provide statistical information about their number of downloads through Docker Hub's unofficial API discussed in 5.2.1 and 5.2.2, which makes the identification of eventual correlations between the ten most popular and most vulnerable Certified images impossible.

Is there a correlation between the last updated images and the most vulnerable ones?

A complete list of the ten last updated images for each type of repository is provided in its entirety with timestamps in Appendix D.3, while the most vulnerable repositories for each type of image are illustrated on the left hand side of figure 6.9, 6.10, 6.11 and 6.12 above. Although there are no apparent correlations between the last updated and the most vulnerable images across all types of repositories, table 6.4 below shows that the last ten repositories to be updated for each type of repository contain an average number of total vulnerabilities lower than the average for the entire set. Thus, Docker images which have been recently updated are more likely to contain less vulnerabilities than older ones.

| Average number of vulnerabilities for | Official repositories | Community repositories | Verified repositories | Certified repositories |
|---------------------------------------|-----------------------|------------------------|-----------------------|------------------------|
| Top 10 last updated repositories | 91 | 139 | 129 | 14 |
| Complete set of repositories | 177 | 153 | 157 | 30 |

Table 6.4: Comparison of the average number of total vulnerabilities per repository type for the last 10 updated and the complete set of repositories

Nonetheless, a small correlation regarding Certified repositories with two of the last ten updated repositories in the name of "store/ibmcorp/websphere-liberty" and "store/sematext/agent" are found in the top 10 most vulnerable Certified repositories. Indeed, the former repository contains 44 vulnerabilities although it was updated recently (March 27th) at the time of the experiment's conduction, whereas the latter holds 46 vulnerabilities despite its last update on March 20th. Additionally, a similar correlation exists for a single Verified image identified as "mcr.microsoft.com/oryx/build", which holds 643 vulnerabilities in total despite being recently updated. Although insufficient to draw solid conclusions, those indications suggest that some images may be updated without actually patching all of their vulnerabilities.

Finally, the rest of the collected information available in Appendix D.3 reveals that Official images are the most frequently updated of all three types, as they are generally updated every 12 minutes on average, whereas Verified and Certified images are the least frequently updated with an updating frequency averaging around 36 hours.

Is there a correlation between base images and the most vulnerable, most popular or last updated images?

A complete list of all the base images for each type of repository is provided in its entirety in Appendix D.4. As previously mentioned, a list with the ten last updated images for each type of repository is also provided in its entirety with timestamps in Appendix D.3, while the most popular and most vulnerable repositories for each type of image are illustrated in figure 6.9, 6.10, 6.11 and 6.12 above.

As mentioned in 6.1, over 50% of the analyzed Community repositories are base images due to a significant error caused by many Community images having a non-identifiable parent, as discussed in details in the discussion chapter under 8.2. This detailed research question is therefore irrelevant for Community repositories, although by only considering the top ten most used base Community repositories, it may be observed that 7 out of 10 base repositories correlate with the most popular repositories in the set.

Regarding the Official set, Glassfish is both a base repository and the third most vulnerable Official repository on Docker Hub. The correlation between the number of base and the most vulnerable Official repositories is therefore limited, which is somewhat reassuring as Official base repositories are often used as parents. Additionally, the Ubuntu and Alpine repositories figuring among the ten most popular Official repositories are also base repositories. The limited number of correlations between the Official base images and the most popular images on the Docker Hub platform suggests therefore that base images may not be as popular and influential as expected, although no conclusion may be drawn without further investigation. Note also that there is no apparent correlation between the base and the last updated Official repositories.

Half of the Verified base repositories correlate with the ten most popular repositories of that kind, whereas only 1/5 of the top ten last updated Verified repositories are base ones. Contrary to their Official peers, Verified base repositories tend therefore to be updated more often, while their popularity is also significantly higher. Moreover, it should be noticed that there is no apparent correlation between the base and the most vulnerable Verified repositories.

Finally, the entirety of the top ten most vulnerable Certified repositories are base ones, while half of the most popular Certified repositories are in that case. Those findings are very alarming as they suggest that many Certified base repositories with a high popularity are also among the most vulnerable repositories, potentially spreading vulnerabilities to many child images using them as parents. Furthermore, there is no apparent correlation between the base and the last updated Certified repositories, which also suggests that Certified base repositories are not updated that often despite their popularity and high vulnerability.

Is there a correlation between vulnerabilities found in base and non-base images?

| Rank | Official | | Community | | Verified | | Certified | |
|------|-----------------------|-----------------------|-----------------------|-----------------------|----------------|----------------|----------------|----------------|
| | Base | Non-base | Base | Non-base | Base | Non-base | Base | Non-base |
| 1 | CVE-2018-6829 | CVE-2017-7245 | CVE-2016-9842 | CVE-2016-10228 | RHSA-2018:3140 | CVE-2019-7309 | RHSA-2018:3157 | RHSA-2019:0483 |
| 2 | CVE-2016-2781 | CVE-2017-7246 | CVE-2016-9840 | CVE-2017-7246 | RHSA-2018:3158 | CVE-2016-10228 | ELSA-2018-0849 | RHSA-2019-0679 |
| 3 | CVE-2016-10228 | CVE-2013-4235 | CVE-2016-9841 | CVE-2018-7169 | RHSA-2018:3059 | CVE-2016-2779 | RHSA-2019:0201 | CVE-2019-7309 |
| 4 | CVE-2013-4235 | CVE-2016-2781 | CVE-2016-9843 | CVE-2013-4235 | RHSA-2018:3157 | CVE-2018-7169 | RHSA-2018:3092 | CVE-2018-20482 |
| 5 | CVE-2017-11671 | CVE-2018-7169 | RHSA-2018:3140 | CVE-2019-7309 | RHSA-2019:0201 | CVE-2017-7246 | RHSA-2019:0049 | CVE-2019-1543 |
| 6 | CVE-2018-7169 | CVE-2016-10228 | CVE-2016-10228 | CVE-2016-2781 | RHSA-2019:0049 | CVE-2016-2781 | ELSA-2018-2768 | CVE-2017-7245 |
| 7 | CVE-2017-7526 | CVE-2019-7309 | CVE-2019-7309 | CVE-2017-7245 | RHSA-2019:0368 | CVE-2013-4235 | RHSA-2019:0368 | CVE-2016-10228 |
| 8 | CVE-2017-7246 | CVE-2016-2779 | CVE-2017-7245 | CVE-2019-3842 | RHSA-2018:3092 | CVE-2017-7245 | RHSA-2018:3041 | CVE-2018-7169 |
| 9 | CVE-2017-2616 | CVE-2015-8985 | CVE-2018-20482 | CVE-2019-9947 | RHSA-2018:3041 | CVE-2017-12132 | RHSA-2018:3140 | CVE-2015-8985 |
| 10 | CVE-2017-7245 | CVE-2017-12424 | CVE-2017-7246 | CVE-2019-9740 | RHSA-2018:2768 | CVE-2018-20482 | ELSA-2018-0805 | CVE-2013-4235 |

Table 6.5: Correlations between vulnerabilities found in base and non-base images

As shown in table 6.5, the correlations between vulnerabilities found in base and non-base images across repository types are marked with specific colors. As illustrated, there are no correlations between the vulnerabilities found in base and non-base Verified repositories or the ones found in base and non-base Certified repositories. However, 6 vulnerabilities among the ten most popular ones exist in both base and non-base Official repositories, while 4/10 are found in both types of Community repositories. Furthermore, it should be noticed that ten most popular vulnerabilities for Verified and Certified base repositories only contain vulnerabilities from Red Hat and Enterprise Linux distributions, whereas non-base repositories hold mostly CVE vulnerabilities.

Is there a correlation between the most vulnerable packages across repository types?

Table 6.6 below classifies the ten most vulnerable packages for each type of repository with their corresponding CVE and CWE numbers. As discussed in detailed in 2.1.2 and 2.1.3, publicly disclosed vulnerabilities are identified with a CVE, RHSA or ELSA number which may be classified in a CWE category corresponding to a certain type of vulnerability (e.g. privilege escalation, buffer overflows, etc.).

As illustrated below, all the reported vulnerabilities across the four repository types have at least one package containing a CWE-ID of 20. Indeed, the concerned packages are "pcre", "pcre3", "libcrypt20", "sssd-proxy", "openssl-lib", "systemd", and "python-lib". By considering the three most frequent CWE-IDs in the entire analyzed set, the following vulnerability categories may be identified as the most popular across all types of repositories:

- **CWE-20: Input Validation**

The product does not validate or incorrectly validates input that can affect the control flow or data flow of a program.

- **CWE-835: Loop with Unreachable Exit Condition ('Infinite Loop')**

The program contains an iteration or loop with an exit condition that cannot be reached, i.e., an infinite loop.

- **CWE-271: Privilege Dropping / Lowering Errors**

The software does not drop privileges before passing control of a resource to an actor that does not have those privileges.

| Rank | Top 10 vulnerable packages with their corresponding CVE number and CWE-ID | | | |
|------|---|---------------------------------------|---|--|
| | Official | Community | Verified | Certified |
| 1 | pcrc: CVE-2017-7246 (CWE-20) | glibc: CVE-2016-10228 (CWE-835) | glibc: RHSA-2018:3140 (CWE-121) | nss-pem: RHSA-2018:3157 (CWE-125) |
| 2 | shadow: CVE-2013-4235 (CWE-367) | glibc: CVE-2019-7309 (CWE-393) | sssd-proxy: RHSA-2018:3158 (CWE-200) | openssl-lib: RHSA-2019:0483 (CWE-200) |
| 3 | coreutils: CVE-2016-2781 (CWE-270) | pcrc3: CVE-2017-7245 (CWE-20) | libxcb: RHSA-2018:3059 (CWE-122) | libgcc: ELSA-2018-0849 (CWE-338) |
| 4 | glibc: CVE-2016-10228 (CWE-835) | pcrc3: CVE-2017-7246 (CWE-20) | glibc: CVE-2016-10228 (CWE-835) | glibc: RHSA-2018:3092 (CWE-470) |
| 5 | shadow: CVE-2018-7169 (CWE-271) | shadow: CVE-2013-4235 (CWE-367) | glibc: CVE-2019-7309 (CWE-835) | systemd: RHSA-2019:0049 (CWE-122) |
| 6 | pcrc3: CVE-2017-7245 (CWE-20) | coreutils: CVE-2016-2781 (CWE-270) | util-linux: CVE-2016-2779 (CWE-270) | systemd-lib: RHSA-2019:0201 (CWE-400) |
| 7 | glibc: CVE-2019-7309 (CWE-393) | shadow: CVE-2018-7169 (CWE-271) | shadow: CVE-2013-4235 (CWE-367) | systemd: RHSA-2019:0368 (CWE-20) |
| 8 | libcrypt20: CVE-2018-6829 (CWE-200) | tar: CVE-2018-20482 (CWE-835) | shadow: CVE-2018-7169 (CWE-271) | nss: ELSA-2018-2768 (CWE-254) |
| 9 | util-linux: CVE-2016-2779 (CWE-270) | python3.5: CVE-2019-9947 (CWE-113) | pcrc: CVE-2017-7246 (CWE-20) | libssh2: RHSA-2019:0679 (CWE-787) |
| 10 | glibc: CVE-2015-8985 (CWE-19) | python3.5: CVE-2019-9740 (CWE-113) | pcrc3: CVE-2017-7245 (CWE-20) | python-lib: RHSA-2018:3041 (CWE-20) |

Table 6.6: The top 10 vulnerable packages across repository types with their corresponding CVE number and CWE-ID

The identified categories above raise some concerns, as they are related to essential system packages (pcrc3, glibc and shadow) present in any Linux-based distribution. Thus, the fact that such packages are part of the ten most vulnerable packages found across all types of repositories is very concerning. Moreover, it should be noticed that other important and core packages such as systemd, coreutils or util-linux are found among the ten most vulnerable packages, although not for every type of repository.

Finally, note that a complete list of all the CWE-IDs found in the ten most vulnerable packages across all repositories types is available with a corresponding description retrieved from NVD's website in Appendix D.8.

6.3 RQ2: Vulnerabilities and inheritance

Main research question 2: *Are vulnerabilities still inherited from images' parent(s) and in what proportion?*

It may be observed in table 6.7 below that almost 80% of all the analyzed Official repositories depends on a parent Official image, while over 40% of all the analyzed Community repositories are in that case. Note that the result for Community repositories should be relativized as discussed in 8.1.4 due to a significant error caused by many Community images having a non-identifiable

parent. As for Verified and Certified repositories, approximately 1/3 of them are based on a parent Verified or Official image.

| Vulnerabilities and inheritance - total images in our analyzed set (in percent) | | | | | |
|---|--------------------------|---------------------------|--------------------------|--------------------------|---------------------|
| Derived from detailed RQ2 | Official (174 images) | Community (623 images) | Verified (124 images) | Certified (38 images) | All (920 images) |
| Images depending on a parent | 134 (77,01%) | 261 (41,89%) | 36 (29,03%) | 12 (31,58%) | 440 (47,83%) |
| Images with inherited vulnerabilities | 99 (56,90%) | 184 (29,53%) | 32 (25,80%) | 10 (26,32%) | 327 (35,54%) |
| Images with introduced vulnerabilities | 85 (48,85%) | 171 (27,45%) | 28 (22,58%) | 4 (10,53%) | 296 (32,17%) |
| Average number of inherited vulnerabilities | 117 | 113 | 110 | 18 | 118 |
| Average number of introduced vulnerabilities | 158 | 177 | 184 | 10 | 175 |

Table 6.7: Introduced and inherited vulnerabilities across repository types

Moreover, Official images inherit more vulnerabilities in average compared to other repository types, as over 50% of the totally analyzed Official repositories inherit vulnerabilities from their parent, while over 1/4 of the analyzed Community, Verified, and Certified repositories are in that case. Note that this indication is coherent with the proportion of images depending on a parent in each type of repository described in the previous paragraph. Indeed, images relying heavily on the extension of a parent repository are most likely to contain inherited vulnerabilities, as demonstrated with Official repositories which have both the highest proportion of images being based on a parent, as well as the highest proportion of inherited vulnerabilities.

Nonetheless, the average number of introduced vulnerabilities is higher than the average number of inherited vulnerabilities for all types of repositories and Verified images introduce more vulnerabilities in average compared to the rest of the other repository types, without being significantly higher. Furthermore, approximately 25% of all the analyzed Community and Verified repositories introduce new vulnerabilities, which is similar to the proportion of repositories inheriting vulnerabilities for those types. Indeed, it should be noticed that more than 50% of all the analyzed Community and Verified repositories have either no parents or no vulnerabilities at all. Note also that some images may contain both introduced and inherited vulnerabilities. Similarly, the proportion of inherited and introduced vulnerabilities for Official images is of approximately 50%. Finally, only less than 11% of the analyzed Certified repositories introduce new vulnerabilities, suggesting that the vulnerabilities contain in vulnerable Certified images are likely to be inherited.

6.4 RQ1: Certified and Verified vs. Official and Community repositories

Main research question 1: *Have the security measures introduced by Docker Inc. in response to previous research improved Docker Hub's security landscape and to what extent?*

Finally, one of the main goals of this study is to determine whether the measures introduced by Docker Inc. in the form of two new types of repositories (Certified and Verified) in late 2018 have improved Docker Hub's security landscape. As mentioned in the introduction of this chapter, the detailed research questions implied by RQ1 reminded above will be answered in this section.

As briefly discussed in [6.2.1](#) and illustrated in [table 6.2](#), Certified and Verified images contain 353 and 2 586 unique vulnerabilities respectively, while Official and Community images hold 2 304 and 5 095 in total. Based solely on the unique number of vulnerabilities contained in each type of repository, Certified images are by far the most secure, followed by Official and Verified images containing almost fourteen times the unique number of vulnerabilities contained in Certified images. As for Community repositories, they offer the worse level of security as they contain the highest number of unique vulnerabilities for all types of repositories, which is about twice as high as the unique number of vulnerabilities contained in Official and Verified repositories.

When considering the average number of inherited and introduced vulnerabilities illustrated in [table 6.7](#), Official and Community images inherit more vulnerabilities on average compared to Verified and Certified repositories. Furthermore, the average number of inherited vulnerabilities is significantly lower for Certified images than the three other types. As for introduced vulnerabilities, Verified repositories actually introduce more vulnerabilities in average than their peers, although that number (184) is only slightly higher than the average number of introduced vulnerabilities in Community (177) and Official (152) repositories. In the same way as with the average number of inherited vulnerabilities, Certified images introduce a lot less vulnerabilities than their peers in average (only 10).

The distribution of severity levels for unique vulnerabilities across the four repository types shown in [table 6.3](#) highlights that Official images contain 321 vulnerabilities with a high severity, indicating an average of three highly severe unique vulnerabilities per image. On the other hand, each Community image holds only a single unique vulnerability with a high severity in average, while Verified and Certified images hold about four and three respectively.

When considering the distribution of images containing at least one vulnerability of a particular severity level illustrated in [figure 6.7](#) and [6.8](#), Certified repositories hold by far the highest proportion of images with at least one critical severity, while all types of repository contain a large number of images with a vulnerability of high or medium severity. Certified images are therefore the most insecure of all four types when severity levels are set in perspective with the number of affected images in each type of repository, while Verified images perform as poorly as Official and Community images.

Since many of the available images on Docker Hub are based on parent images, it is very likely that they also share identical vulnerabilities. As a simple reminder from [2.1.2](#) in the background chapter, publicly disclosed vulnerabilities with an assigned CVE number are enumerated into vulnerability categories identified through CWE-IDs. [Table 6.8](#) below displays the ten most popular vulnerabilities found across all four types of repositories. The CWE-20 category corresponding to an Input Validation vulnerability, exists across all the repository types. Moreover, 7/10 of the most vulnerable types of attacks in Official images remain among Verified images. By comparing the vulnerability types in the following combined sets: Official and Community against the Verified and Certified set, 2/3 of all the vulnerability categories in the combined Official and Community set is shared with the combined Verified and Certified set. The large majority of vulnerabilities contained in Certified and Verified repositories shares therefore a vulnerability category with the two other types of repositories. Note that a complete list of CWE-IDs with their corresponding vulnerability category and description is available in [Appendix D.8](#).

| Top 10 vulnerability categories across repository types (CWE-ID) | | | | |
|--|--|---|---|---|
| Rank | Official | Community | Verified | Certified |
| 1 | Input Validation (CWE-20) | Infinite Loop (CWE-835) | Stack-based Buffer Overflow (CWE-121) | Out-of-bounds Read (CWE-125) |
| 2 | Race Condition (CWE-367) | Return of Wrong Status Code (CWE-393) | Information Leak / Disclosure (CWE-200) | Information Leak / Disclosure (CWE-200) |
| 3 | Privilege Context Switching Error (CWE-270) | Input Validation (CWE-20) | Heap-based Buffer Overflow (CWE-122) | PRNG (CWE-338) |
| 4 | Infinite Loop (CWE-835) | Race Condition (CWE-367) | Infinite Loop (CWE-835) | Unsafe Reflection (CWE-470) |
| 5 | Privilege Dropping / Lowering Errors (CWE-271) | Privilege Context Switching Error (CWE-270) | Privilege Context Switching Error (CWE-270) | Heap-based Buffer Overflow (CWE-122) |
| 6 | Return of Wrong Status Code (CWE-393) | Privilege Dropping / Lowering Errors (CWE-271) | Race Condition (CWE-367) | Resource Exhaustion (CWE-400) |
| 7 | Information Leak / Disclosure (CWE-200) | HTTP Response Splitting (CWE-113) | Privilege Dropping / Lowering Errors (CWE-271) | Input Validation (CWE-20) |
| 8 | Data Handling (CWE-19) | CRLF Injection (CWE-93) | Input Validation (CWE-20) | Out-of-bounds Write (CWE-787) |
| 9 | Integer Overflow or Wraparound (CWE-190) | Permissions, Privileges, and Access Control (CWE-264) | Permissions, Privileges, and Access Control (CWE-264) | Resource Management Errors (CWE-399) |
| 10 | Buffer Errors (CWE-119) | Resource Management Errors (CWE-399) | Out-of-bounds Read (CWE-125) | Buffer Errors (CWE-119) |

Table 6.8: The ten most popular vulnerability categories across all types of repositories

To conclude, the level of security provided by Certified repositories is bar far superior to the three other types in every aspect when severity levels are not taken into account. However, when the latter are a part of the equation, Certified repositories actually perform worse, as they tend to hold vulnerabilities of higher severity, although their total number of contained vulnerabilities is significantly lower than the other types of repositories. As for Verified repositories, the latter offer a very similar degree of security than Official repositories, which does not vary with/without the consideration of severity levels. Finally, note that Community repositories perform worse than Official and Verified repositories in all scenarios, but provide a higher degree of security than Certified images when severity levels are taken into account.

6.5 Additional research question

Additional research question: *Are vulnerabilities propagating from a small set of highly influential base images?*

In addition to the main research questions defined in 1.2, the investigation of an additional research question based on an observation by Shu et al. in [11] suggesting that "*it is highly likely that the root cause of pervasive vulnerabilities on Docker Hub is the result of propagation from a relatively small set of highly influential base images*" was addressed. Indeed, their suggestion was first formally stated into a dedicated additional research question, as stated above. Secondly, the detailed research questions implied by that new inquiry were identified and answered as followed:

Are the ten most popular parent images correlated with the the ten most vulnerable packages across all types of images in some way?

Table 6.9 below sets in perspective the ten most popular parent images across all types of repositories with the ten most vulnerable packages across all types of images ("firefox" being the most vulnerable package and "openssl" being the least vulnerable of all ten). As illustrated, only one parent image out of the ten most popular ones across all types of repositories contain one of the most vulnerable packages found across all types of repositories. Thus, there are almost no correlations between the ten most popular parent images and the the ten most vulnerable packages found across all types of repositories, advising that Shu et al.'s suggestion may be wrong. Note however that it has been three years since their research was conducted. Their assumption might have therefore been correct at the time of their study (April 2016), as the Docker Hub ecosystem has rapidly changed during that period of time, as explained in 2.5. However, if only the most vulnerable packages across the ten most popular parent images are considered, a certain number of correlations may be observed, as addressed by the second detailed research question implied by Shu et al.'s suggestion.

| Rank | Parent image | Number of descendant images | Top 10 most vulnerable packages across all repository types | | | | | | | | | |
|------|----------------------------|-----------------------------|---|-------|-------------|----------|------|---------|-----------|------|-----------|---------|
| | | | firefox | linux | imagemagick | binutils | php5 | tcpdump | mysql-5.5 | tiff | openjdk-7 | openssl |
| 1 | centos:7 | 26 | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| 2 | debian:9-slim | 26 | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| 3 | alpine:3.8 | 25 | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| 4 | alpine:latest | 23 | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| 5 | ubuntu:xenial | 18 | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| 6 | ubuntu:latest | 14 | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| 7 | java:openjdk-8-jre | 11 | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ |
| 8 | debian:latest | 10 | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| 9 | ubuntu: bionic-20190204 | 8 | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| 10 | ubuntu: bionic-20181204 | 7 | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |

Table 6.9: Correlations between the ten most popular parent images and the ten most vulnerable packages across all types of repositories

Are the ten most popular parent images correlated with the ten most vulnerable packages found across those images in some way?

Table 6.10 below shows the ten most popular parent images across all types of repositories with the ten most vulnerable packages found across those images ("glibc" being the most vulnerable package and "sqlite3" being the least vulnerable of all ten). As illustrated, the number of parent images containing one of the most vulnerable packages found among those images is fairly high. Indeed, 5 of the unique vulnerabilities contained in the ubuntu images as listed in Appendix D.6.1 are among the ten most popular vulnerabilities found in the most popular parent images used on Docker Hub (apart from "ubuntu:xenial" holding 4). Moreover, the 4 vulnerabilities contained in the "debian:latest" image are all found among the ten most popular vulnerabilities for the top ten downloaded images. The "java:openjdk-8-jre" image containing 271 unique vulnerabilities also holds almost all the ten most popular vulnerabilities found in the most popular parent images, apart from the one related to the "curl" package.

Nonetheless, "centos:7" and "debian:9-slim" are the two most popular parent images used across Docker Hub at the time of this writing and contain respectively 4 and 47 unique vulnerabilities. However, none of the unique vulnerabilities contained in the "centos:7" image are among the ten most popular vulnerabilities found in the most popular parent images used on Docker Hub. In a similar way, only 4 out of the 47 unique vulnerabilities contained in the "debian:9-slim" image are found among the ten most popular vulnerabilities identified in the most downloaded parent images. Additionally, the "alpine:3.8" and "alpine:latest" images do not contain such vulnerabilities at all.

| Rank | Parent image | Number of descendant images | Top 10 most vulnerable packages across the ten most popular parent images | | | | | | | | | |
|------|----------------------------|-----------------------------|---|---------|---------|----------|------|-------|---------|------|-----|---------|
| | | | glibc | ncurses | systemd | gnutls28 | curl | pcrc3 | openssl | krb5 | nss | sqlite3 |
| 1 | centos:7 | 26 | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| 2 | debian:9-slim | 26 | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ |
| 3 | alpine:3.8 | 25 | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| 4 | alpine:latest | 23 | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| 5 | ubuntu:xenial | 18 | ✓ | ✓ | ✗ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ |
| 6 | ubuntu:latest | 14 | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ |
| 7 | java:openjdk-8-jre | 11 | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ |
| 8 | debian:latest | 10 | ✓ | ✓ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ |
| 9 | ubuntu: bionic-20190204 | 8 | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ |
| 10 | ubuntu: bionic-20181204 | 7 | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ | ✗ |

Table 6.10: Correlations between the ten most popular parent images and the ten most vulnerable packages found across those images

Based on the above results, it seems therefore that Shu et al.'s suggestion about vulnerabilities potentially propagating from a small set of highly influential base images to the whole Docker Hub platform is refuted.

6.6 Summary

It is clear that the identified detailed research questions implied by the original problem statement introduced in 1.2 provided a straightforward overview of Docker Hub’s current security landscape. To conclude this chapter, a summary of our findings will therefore be provided here with references to the appropriate sections.

First, it was found that although less Certified images are completely free of vulnerabilities compared to the three other types of repositories, they contain far less vulnerabilities than their other peers even for the most vulnerable Certified repositories (6.2.1, 6.2.4). Nonetheless, Certified images tend to contain vulnerabilities of higher severity, especially when it comes to critical vulnerabilities (6.2.2, 6.2.3). Similarly, Verified and Official repositories contain a significantly lower number of unique vulnerabilities than Community images, with a higher chance of being of a lower severity level (6.2.1, 6.2.2). Overall, all types of repository contain a large number of images with a vulnerability of high or medium severity.

Secondly, the proportion of images containing at least one inherited vulnerability across all types of repositories is similar to the proportion of images containing at least one introduced vulnerability (6.3). Nonetheless, the average number of introduced vulnerabilities is higher than the average number of inherited vulnerabilities for all types of repositories.

Thirdly, the number of correlations between the most vulnerable, most popular and the most recently updated images across the four repository types is limited (6.2.4). Indeed, Docker images which have been recently updated are more likely to contain less vulnerabilities than older ones, although it was found that some images may be updated without actually patching all of their vulnerabilities, as some recently updated images were found in the top 10 most vulnerable images.

Fourthly, the number of critically vulnerable images across all four types of repositories is very limited when only unique vulnerabilities are considered (6.2.2). However, many essential system packages such as `pcre3`, `glibc`, `shadow` or `systemd` are part of the ten most vulnerable packages found across all types of repositories, which raises a certain number of concerns as such packages are found in the vast majority of Docker images (6.2.4). Note that the most popular vulnerability categories found across all repository types are input validation, followed by infinite loop and privilege dropping.

Finally, it was concluded based on the result data discussed in 6.5 that Shu et al.’s suggestion about vulnerabilities potentially propagating from a small set of highly influential base images to the whole Docker Hub platform is refuted.

In general, the level of security provided by each type of repository is more or less as anticipated in our expectations detailed in 3.5. Indeed, Certified repositories offer by far the best level of security when severity levels are not taken into account, followed by Verified and Official, while Community repositories usually contain a significantly higher number of unique vulnerabilities. When severity levels are taken into account however, Certified repositories actually perform worse, as they tend to hold vulnerabilities of higher severity, although their total number of contained vulnerabilities is significantly lower than the other types of repositories. Official and Verified repositories still perform similarly when including severity levels, while Community images continue offering a lower level of security.

Chapter 7

Analysis

Chapter 7 intends to deeply analyze the data obtained and described in the previous chapter by using common mathematical concepts and indicators. This chapter is divided into two parts. First, we utilize general statistics and modeling in order to determine the distribution of the collected data, identify correlations between the total number of vulnerabilities found in each type of repository and estimate their future number for the next seven years (until 2025 included). Secondly, a network model illustrating parental relationships between images with their inherited vulnerabilities will be created in order to identify patterns, influential repositories and correlations related to dependencies and inheritance. Note that since only one image (i.e. the most recent one) was analyzed for each repository being part of the conducted experiments, the terms "image" and "repository" may be used interchangeably in this chapter.

7.1 Vulnerability distributions and predictions

General statistics may help determining the distribution of the collected data, identify correlations between the total number of contained vulnerabilities found in each image composing the four types of repository and estimate their future number for the next seven years (until 2025 included) through modelling and important statistical indicators.

7.1.1 General interpretation

As presented in table 7.1 below, 752 images were analyzed in the total during the conduction of our experiments. If parent images are considered as well, the total number of analyzed images is of 920. Note that table 7.1 focuses on illustrating descriptive statistics for the total number of vulnerabilities found in each image composing the different repository types, leading to multiple observations.

First, Certified images hold only 30 vulnerabilities in average, compared to approximately 155 for both Verified and Community images. Official repositories perform worse with 177 vulnerabilities in average. Thus, Certified images offer the best level of security when solely the total number of contained vulnerabilities is considered, while the three other types provide a similar level of security, although Official repositories contain a slightly higher number of vulnerabilities in

average. Furthermore, the standard deviation for the total number of contained vulnerabilities within Certified repositories is significantly lower than the one applying for the three other types of images. Thus, the number of vulnerabilities hold within Official, Community and Verified repositories varies substantially more than the ones contained in Certified repositories, although that number is slightly higher for Community images.

Secondly, when only the most vulnerable image for each type of repository is considered, the Certified type performs significantly better than its peers with almost ten times less contained vulnerabilities, while the three other types provide a similar level of security, although the most vulnerable Community repository holds a slightly higher number of contained vulnerabilities. Moreover, it should be noticed that all four types of repositories contain at least one image without any vulnerability.

| Vulnerability statistics across repository types | | | | |
|--|----------|-----------|----------|-----------|
| | Official | Community | Verified | Certified |
| Total analyzed images | 128 | 500 | 98 | 31 |
| Mean | 177 | 153 | 157 | 30 |
| Standard deviation | 244 | 286 | 248 | 37 |
| Median | 93 | 39 | 90 | 22 |
| Minimum | 0 | 0 | 0 | 0 |
| Maximum | 1530 | 1792 | 1531 | 166 |
| 25th percentile (Q1) | 1 | 3 | 18 | 2 |
| 75th percentile (Q3) | 258 | 157 | 144 | 43 |

Table 7.1: Descriptive statistics of the total number of vulnerabilities found in each repository type

Finally, the median is significantly lower than the mean for Official, Verified and Community repositories, suggesting that many of those images contain a high number of vulnerabilities compared to Certified repositories where the difference is substantially lower.

7.1.2 Interpreting box plots

Plotting the distribution of the total number of vulnerabilities found in each image of the four types of repository with the help of a box plot displayed in figure 7.1 below allows determining whether there exists outliers, while summarizing the so called "five-number summary" illustrated in table 7.1 above consisting of the sample's minimum value, lower quartile (Q1), median value, upper quartile (Q3) and maximum value. Every box plot has two parts: a box and two whiskers. The box's length indicates the global variation of the data it represents by starting at the first quartile of the sample and ending at its upper quartile (noted Q1 and Q3 in table 7.1). The box's thick black horizontal line, also known as its median value, indicates where the sample is centered. As for the whiskers, the latter are simply two vertical lines outside the box, which extend to the highest and lowest observations.

Based on figure 7.1, it may be observed that the box plot for Official images is significantly longer than the one for the three other types of repositories. Thus, the overall total number of vulnerabilities found in each Certified (may be hard to distinguish since the maximum value for Certified images is only 166), Verified and Community images is more compact, while the

total number of vulnerabilities found in each Official repository is more spread out. Moreover, the data is almost centred on the same level for Official and Verified images, while the median is substantially lower for Community and Certified repositories, suggesting that the number of contained vulnerabilities within the latter varies more than for Verified and Official images. Since the maximum number of vulnerabilities varies greatly from one repository type to another, the size of the box plots differs subsequently.

A box plot is divided into four quartiles. The body of the box goes from the first quartile (also referred to as Q1, the 25th percentile or the lower quartile) and the third quartile (also known as Q3, the 75th percentile or the upper quartile). As shown in table 7.1 and illustrated in figure 7.1, we may confirm that 75% of the data in each type of repository is less than its corresponding Q3 value, while 25% is less than its corresponding Q1 value. Thus, 75% of the analyzed Official images contain less than 258 vulnerabilities in total, while 75% of the Community, Verified and Certified images contain less than 157, 144 and 43 vulnerabilities respectively. Similarly, 25% of the analyzed Official images contain 0 or 1 vulnerability in total, while 25% of the Community, Verified and Certified images contain less than or equal to 3, 18 and 2 vulnerabilities respectively.

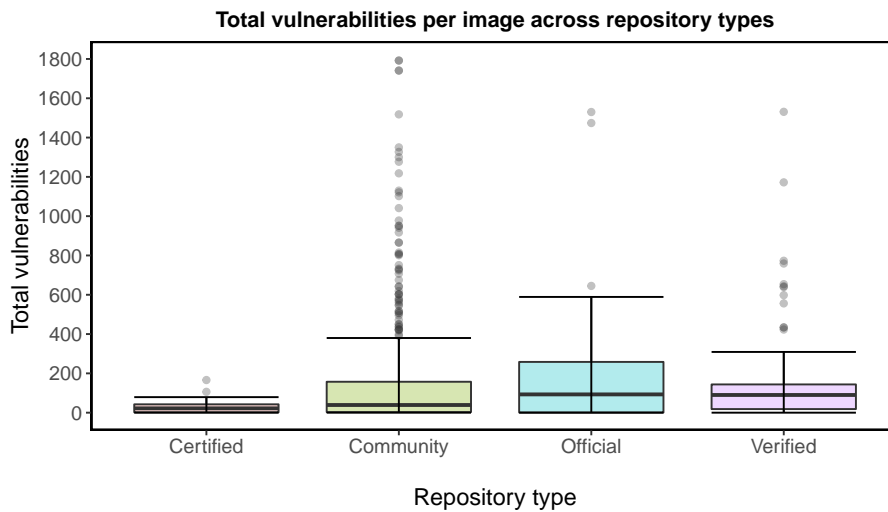


Figure 7.1: Total number of contained vulnerabilities per image across repository types

The analysis of the first and third quartile reveals therefore that Official images tend to contain more vulnerabilities than their peers in general, while the 25% least vulnerable Verified images contain a significantly higher number of vulnerabilities than the 25% least vulnerable images of other types. Furthermore, Certified images are confirmed to be a lot more secure than their peers, as their 75% least vulnerable images contain more than three times less vulnerabilities than Verified and Official images, as well as six times less vulnerabilities than the 75% least vulnerable Community images. Note that the difference between Q1 and Q3 is called the inter-quartile range (IRQ).

The whiskers, which are the vertical lines above and below the box plots' bodies, represent the range for the bottom and top 25% of the data values (1.5 times the IRQ) where the extreme values (referred to as outliers) are excluded. As observed in figure 7.1, the lower whisker for all the box plots are hard to identify, as they lie very close to the minimum values. Indeed,

every type of repository contains multiple images without any vulnerabilities. On the other hand, all of the box plots have visible upper whisker. For the example, the upper 25% of the Official box plot contains 645 vulnerabilities in total, while the upper 25% of the Community, Verified, and Certified repositories hold 380, 309, and 79 respectively. Thus, the 25% most vulnerable Official repositories contain a significantly higher number of vulnerabilities than the other types of repositories, while the 25% most vulnerable Certified images contain almost four times less vulnerabilities than Verified and Community images, as well as eight times less than Official images. When considering the 25% most vulnerable images, Certified repositories provide therefore a significantly better security level than their peers, followed by Verified Community and Official repositories.

Finally, it should be noticed that the Community box plot has a lot of extreme values compared to the other types of repository, partly due to its significantly larger sample size.

7.1.3 Interpreting density plots

Contrary to box plots summarizing many statistical indicators while limiting the level of details provided, density plots offer a more precise way of representing the distribution of a sample, while smoothing out the noise created by outliers displayed in a usual histogram. Indeed, plotting the density distribution of the total number of vulnerabilities found in each image of the four types of repository allows determining how tight the data is grouped (i.e. the density according to which the vulnerability sample is distributed), as well as whether it is symmetrical, left or right skewed. Note that the density scale of such a plot ranges from 0 to 1, where 0 indicates that no image contain the number of vulnerabilities displayed in a particular point in the x-axis, while a density of 1 represents the entire sample. Figure 7.2 and 7.3 below show the density distribution of the total number of vulnerabilities found in each image of the four types of repository, where the peak of each curve displays where most of the vulnerabilities are concentrated over the interval.

First, the distribution of the total number of vulnerabilities found in each Official image illustrated on the left hand side of figure 7.2 reveals that most of the Official images contain a number of vulnerabilities located on the lower part of the vulnerability interval, as the distribution is clearly right-skewed. Furthermore, smoothing out outliers allows identifying that the vast majority of Official images contain a total number of vulnerabilities in the range of 0 to 450 vulnerabilities, which is significantly higher than the average number (mean) of vulnerabilities (177) found in Official repositories as demonstrated in table 7.1. Since the distribution of the total number of vulnerabilities found in each Official image is right-skewed, the latter decreases rapidly after 125 vulnerabilities, confirming that most Official images contain a total number of vulnerabilities kept under that threshold. Moreover, it should be noticed that there are no Official images containing a total number of vulnerabilities between 450 and 500 vulnerabilities, as illustrated by the gap in the x axis of the Official plot situated on the left hand side of figure 7.2. Nonetheless, a resurgence of a significantly dense number of Official images may be observed after that gap, illustrating that a non-negligible number of Official images hold between 500 and 650 vulnerabilities in total.

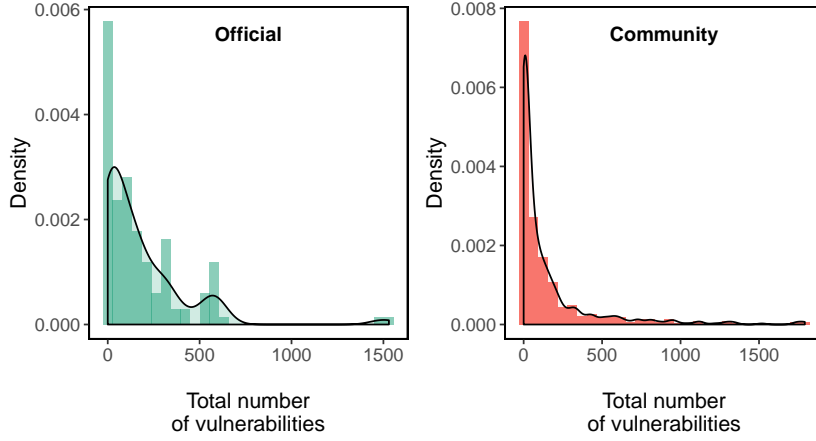


Figure 7.2: Density distribution of the total number of contained vulnerabilities per Official and Community image

Secondly, the distribution of the total number of vulnerabilities found in each Community image illustrated on the right hand side of figure 7.2 shows that similarly to their Official peers, most of the Community images contain a number of vulnerabilities located on the lower part of the vulnerability interval, as the distribution is clearly right-skewed. Furthermore, smoothing out outliers allows identifying that the vast majority of Community images contain a total number of vulnerabilities in the range of 0 to 250 vulnerabilities, which is significantly higher than the average number (mean) of vulnerabilities (153) found in Community repositories as shown in table 7.1. Since the distribution of the total number of vulnerabilities found in each Community image is right-skewed, the latter decreases rapidly after 250 vulnerabilities, confirming that most Community images contain a total number of vulnerabilities kept under that threshold. Note however that there are still a certain number of Community images with various sums of totally contained vulnerabilities, as the density illustrated on the right hand side of figure 7.2 is almost never equal to 0, contrary to the three other types of repositories displayed in the same figure as well as figure 7.3.

Thirdly, the distribution of the total number of vulnerabilities found in each Verified image illustrated on the left hand side of figure 7.3 reveals that similarly to their Official and Community peers, most of the Verified images contain a number of vulnerabilities located on the lower part of the vulnerability interval, as the distribution is clearly right-skewed. Furthermore, smoothing out outliers allows identifying that the vast majority of Verified images contain a total number of vulnerabilities in the range of 0 to 250 vulnerabilities, which is significantly higher than the average number (mean) of vulnerabilities (157) found in Verified repositories as demonstrated in table 7.1. Since the distribution of the total number of vulnerabilities found in each Verified image is also right-skewed, the latter decreases rapidly after 250 vulnerabilities, confirming that most Verified images contain a total number of vulnerabilities kept under that threshold. Moreover, it is important to note that similarly to Official images, certain gaps in the x axis of the Verified plot situated on the left hand side of figure 7.3 illustrate that there exist no Verified image with a total number of vulnerabilities located within certain ranges (e.g. between 450 and 550 or 850 and 1150). Nonetheless, small spikes may be observed long after the 250 threshold, showing that there exists a resurgence of a non-negligible number of Verified images containing a much higher

number of vulnerabilities than 250.

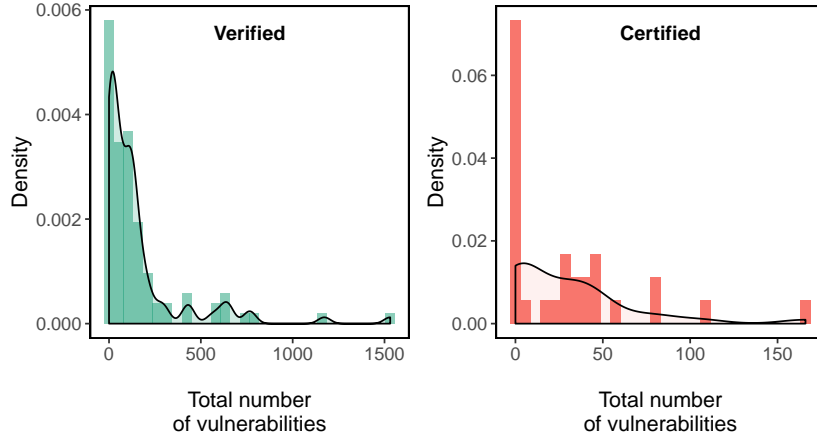


Figure 7.3: Density distribution of the total number of contained vulnerabilities per Verified and Certified image

Fourthly, the distribution of the total number of vulnerabilities found in each Certified image illustrated on the right hand side of figure 7.3 shows that similarly to their three other peers, most of the Certified images contain a number of vulnerabilities located on the lower part of the vulnerability interval, as the distribution is clearly right-skewed. Furthermore, smoothing out outliers allows identifying that the vast majority of Certified images contain a total number of vulnerabilities in the range of 0 to 60 vulnerabilities, which is significantly higher than average number (mean) of vulnerabilities (30) found in Certified repositories as demonstrated in table 7.1. Since the distribution of the total number of vulnerabilities found in each Certified image is also right-skewed, the latter decreases rapidly after 60 vulnerabilities, confirming that most Certified images contain a total number of vulnerabilities kept under that threshold. Moreover, it should be noticed that similarly to Official and Verified images, large gaps in the x axis of the Certified plot situated on the right hand side of figure 7.3 illustrate that there exist no Certified image with a total number of vulnerabilities located within certain ranges (e.g. between 60 and 80 or 85 and 105). Although small spikes may be observed long after the 60 threshold, those values do not influence the global shape of the distribution, clearly showing that the major part of the Certified images are concentrated between 0 to 60 contained vulnerabilities.

Finally, by displaying the density distribution of the total number of vulnerabilities found in each image across all four types of repository as illustrated in figure 7.4 below, it may be observed that the distribution curves for the Official, Community and Certified repositories meet around 75 contained vulnerabilities. Thus, the majority of images across those three types of repositories contain less than 75 vulnerabilities in total, where Certified images are the most dense of all three types within that range, followed by Community and Official images respectively. Furthermore, it should be noticed that the density of Certified images at the very beginning of the vulnerability range is significantly higher than the three other types of repositories, confirming that the number of contained vulnerabilities in the large majority of the Certified images is very low compared to their peers.

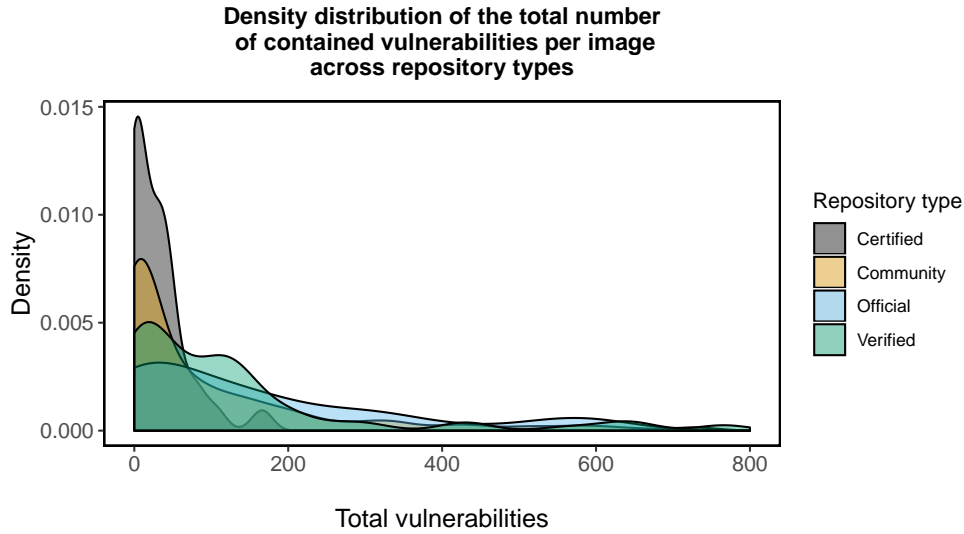


Figure 7.4: Density distribution of the total number of contained vulnerabilities per image across repository types

| Share of images with less than or 180 contained vulnerabilities | | | | |
|---|----------|----------|-----------|---------|
| Community | Official | Verified | Certified | Total |
| 388/500 | 85/128 | 76/98 | 31/31 | 580/757 |
| 77.60 % | 66.41% | 77.56% | 100% | 76.6% |

Table 7.2: Share of images in each type of repository with less than or 180 contained vulnerabilities

As for Verified images, it may be observed that the total number of vulnerabilities they hold is significantly higher than their peers until a certain threshold of about 180 contained vulnerabilities is reached. Indeed, while the majority of images across Official, Community and Certified repositories contain less than 75 vulnerabilities, the majority of Verified images hold less than 180 vulnerabilities. Thus, the number of Verified images with a high amount of contained vulnerabilities before the 180 threshold is a lot more dense than the three other types of images, showing that Verified images with less than 180 vulnerabilities are more likely to hold a more important number of total vulnerabilities than images in the same case from one of the three other types of repositories.

Note also that the number of Official images with a higher number of contained vulnerabilities than the 180 threshold decreases less rapidly than the three other types of repositories, showing that Official images are more likely to contain a higher number of total vulnerabilities than their peers. Overall, the majority of images across all types of repositories contain less than 180 vulnerabilities in total, as summarized in table 7.2 above.

7.1.4 Analyzing potential quantitative vulnerability correlations between dependent repository types

Section 6.3 under chapter 6 set in perspective the number of vulnerabilities found in each type of repository with their inheritance and introduced aspects. Although the average number of introduced vulnerabilities was superior to the number of inherited vulnerabilities for most types of repository, images of all types except Certified ones tended to contain a large number of inherited vulnerabilities. Furthermore, Certified repositories contained a significantly higher number of inherited vulnerabilities in average than the number of new vulnerabilities they introduced. Analyzing the correlation coefficients of the total number of unique vulnerabilities between in each type of repository using a scientific approach may therefore help revealing that the quantitative evolution of vulnerabilities found in dependent repository types is linearly correlated in some way. As a reminder from 2.5.4, Official images may only be based on images of the same type, while Community images may be based on any type. Similarly to their Official peers, Certified and Verified images are solely allowed to use images of the same type as their parents, as well as Official images.

In order to scientifically identify potential linear correlations between the total number of unique vulnerabilities contained in each type of repository, the Pearson product-moment correlation coefficient may be used, as illustrated in figure 7.5 below. Indeed, the line of best fit drawn by a Pearson correlation, as well as its correlation coefficient allows determining whether the total number of unique vulnerabilities contained in two different types of repositories are linearly correlated. It should be noticed that only the linear associations of the total number of unique vulnerabilities contained in different image types with a parent-child relationship are of interest (i.e Community-Official, Certified-Verified, Certified-Official and Verified-Official). Note also that the value of the correlation coefficient determines the linear strength of the relationship between the total number of unique vulnerabilities contained in two different repository types and may be interpreted as followed:

- 0: indicates no linear relationship
- 1: indicates a perfect positive linear relationship. If one variable increases its value, then the other variable also increases its value and follows the exact linear line
- -1: indicates a perfect negative linear relationship. If one variable decreases its value, then the other variable decreases its value and follows the exact linear line.
- Between 0 and 0.3 or 0 and -0.3: indicates a weak positive/negative linear relationship
- Between 0.3 and 0.7 or -0.3 and -0.7: indicates a moderate positive/negative linear relationship
- Between 0.7 and 1.0 or -0.7 and -1.0: indicates a strong positive/negative linear relationship

As illustrated in figure 7.5 below, the total number of unique vulnerabilities contained in each type of repository correlates with the other types in some way, as none of them have correlation coefficient equal to 0. For example, it may be observed in the plot associating the unique vulnerabilities contained in Community repositories with the unique vulnerabilities contained in Official repositories (column 1, row 2) that they barely correlate with each other in a way that is negatively linear, with a corresponding correlation coefficient of -0.0479 (column 2, row 1). Thus, whenever the total number of unique vulnerabilities contained in Official repositories increases, the number of unique vulnerabilities contained in Community repositories decreases slightly.

That analysis confirms our observations of the average number of introduced vulnerabilities in Community images being more important than the average number of inherited vulnerabilities discussed in 6.3, indicating that Community images do not inherit most of their vulnerabilities.

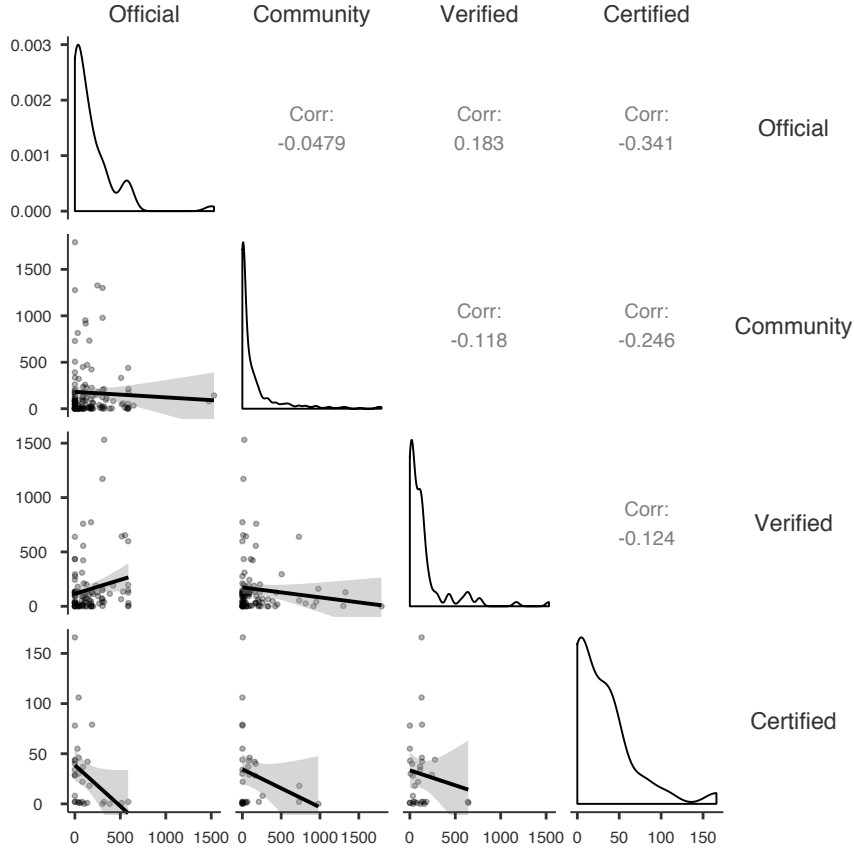


Figure 7.5: Linear relationships of the total number of unique vulnerabilities between in each type of repository

Moreover, Certified images which may be based on either Verified or Official repositories have a -0.124 and -0.341 correlation coefficient with the latter repository types respectively. Thus, Certified images are only slightly negatively correlated with Verified images, whereas their correlation coefficient with Official images indicates a moderate negative linear relationship. Whenever the total number of unique vulnerabilities contained in Verified images increases, the number of unique vulnerabilities contained in Certified repositories only decreases slightly. Similarly but more accentuated, an increase of the total number of unique vulnerabilities contained in Official images leads to a moderate decrease of the number of unique vulnerabilities contained in Certified images. That analysis contradicts our observations from 6.3 indicating that Certified images tend to inherit more vulnerabilities in average than the number of vulnerabilities they introduce. Note however that the reliability of the linear model requires a sufficient number of sample, which is somewhat limited for Certified images due to their low number of repositories

available on Docker Hub.

Finally, when comparing the total number of unique vulnerabilities contained in Verified repositories with the number of unique vulnerabilities contained in Official repositories which they may be based on, a weak positive linear relationship may be identified. Indeed, their correlation coefficient of 0.183 indicates that whenever the number of unique vulnerabilities contained in Official repositories increases, the total number of unique vulnerabilities contained in Verified repositories increases similarly in a weak way. That analysis contradicts our observations from 6.3 indicating that Verified images contain less inherited vulnerabilities in average than the number of vulnerabilities they introduce. Nonetheless, it should be noticed that such a positive linear relationship may also indicate that Verified images introduce many of the same vulnerabilities when an quantitative increase occurs, due to many similar software packages being containerized in both types of images without being inherited.

7.1.5 Predicting quantitative software vulnerabilities by 2025

As demonstrated in figure 6.5 from chapter 6.2, the total number of unique vulnerabilities reported for Official, Community and Verified repositories reached an all-time high in 2017 and dropped by 25% to 35% the year after. On the contrary, the number of unique vulnerabilities contained in Certified images increased by about 30% from 2017 to 2018. Moreover, the global distribution of known unique vulnerabilities from 2010 to 2018 across all types of repositories is more or less linear, which allows estimating the future number of unique vulnerabilities using a best-fit linear trend line. Indeed, although such estimations are not perfect they provide a certain indication of how Docker Hub's security landscape will look like by 2025 if the number of contained unique vulnerabilities across all types of repositories continues evolving in the same way as they have so far.

The formula for estimating the unknown y -value corresponding to the predicted total number of unique vulnerabilities found yearly across all types of repositories in figure 7.6 below, utilizes the equation for a so called best-fit straight line which linear trend lines are based on and is defined as followed:

$$y = ax + b$$

Note that the x -value is the sample mean of the known total number of vulnerabilities per year and the y -value corresponds to the sample mean of the known years of disclosures for the past vulnerabilities, while a and b are defined as followed:

$$a = \bar{y} - b\bar{x} \qquad b = \frac{\sum (x - \bar{x})(y - \bar{y})}{\sum (x - \bar{x})^2} \qquad (7.1)$$

Using the above formula, the future total number of unique vulnerabilities contained in each type of repository from 2019 to 2025 may be estimated as shown in figure 7.6 below. Each line provides the mean value for the corresponding repository type, while the transparent areas surrounding each curve are the limits defining the confidence intervals for the mean values. Note that a 95% confidence interval was chosen, meaning that there is a 95% certainty that the predicted numbers of unique vulnerabilities for each type of repository between 2019 and 2025 will lie within those

transparent areas. Note also that the detailed numbers of unique vulnerabilities estimated for each type of repository between 2019 and 2025 are available in Appendix D.9.

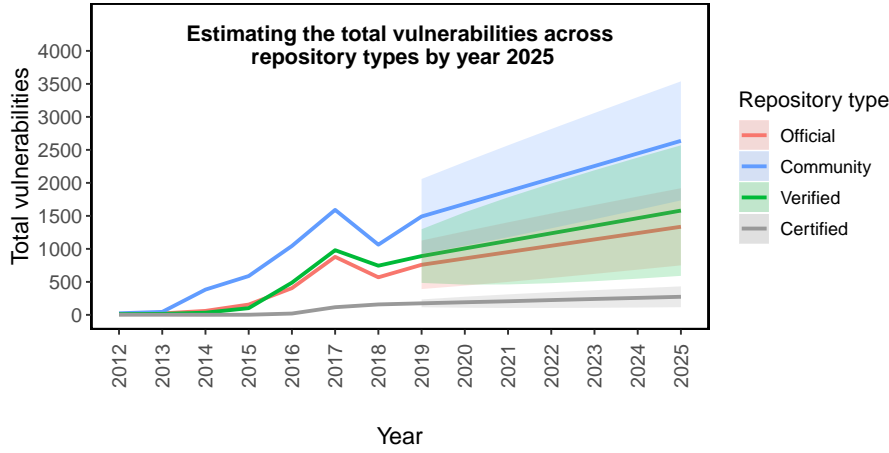


Figure 7.6: Estimating the total vulnerabilities across repository types by year 2025

The computed linear trend line illustrates therefore that the average number of unique vulnerabilities contained in Community images will increase with a rate of approximately 191 vulnerabilities per year between 2019 and 2025. Similarly, the average number of unique vulnerabilities contained in Official, Verified and Certified repositories is expected to grow with approximately 96, 115 and 17 vulnerabilities per year respectively. The order of the four types of repositories based on their level of security will therefore remain unchanged, with Certified repositories providing the best security of all four types, followed by Official and Verified images, while Community repositories will still perform worst. Note however that the global trend is an increase of the number of unique vulnerabilities found across all types of repositories, leading therefore to a deterioration of Docker Hub’s global security landscape when only the total number of unique vulnerabilities is taken into account.

Moreover, it should be noticed that the number of vulnerabilities disclosed in 2019 is very limited due to the early conduction of our experiments during that same year, as explained in chapter 6 under 6.2. Nonetheless, the global growth rate of unique vulnerabilities contained in each type of repository is expected to slow down this year, while it will slowly increase from 2020.

Finally, the linear trend line displayed in figure 7.6 above indicates that Community, Verified, Official and Certified repositories will reach again their all-time high number of uniquely contained vulnerabilities from 2017 in 2028, 2031, 2032 and 2020 respectively.

7.2 Parental relationships and vulnerability inheritance

As explained in 2.5.5, a Docker image is often based on a parent image, which simply consists of extending the latter’s list of installed software packages. In its turn, a parent image may also be

based on another image, therefore creating a certain dependency chain between Docker images, which may inherit vulnerabilities from one or multiple direct/indirect parent(s).

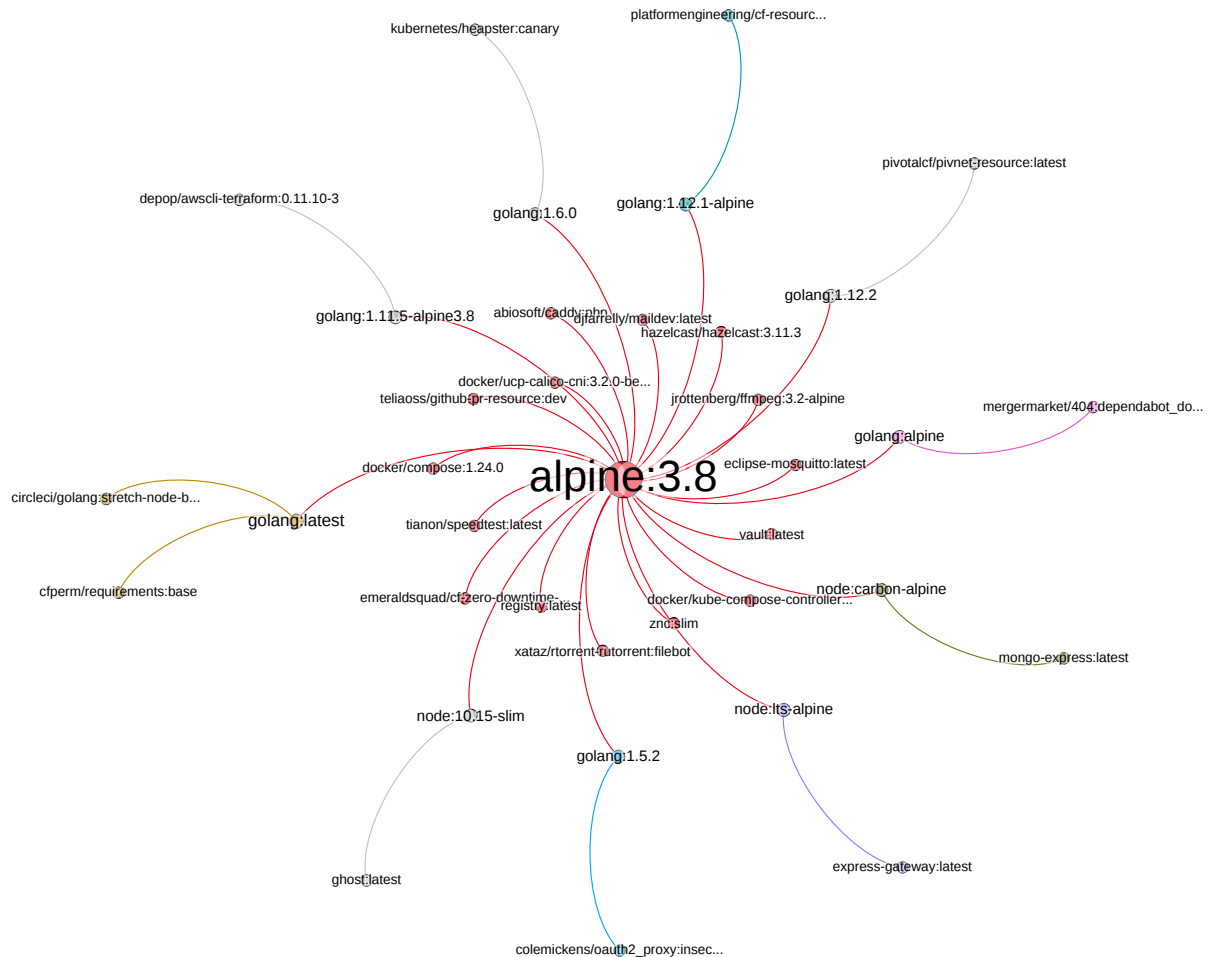


Figure 7.7: Direct and indirect parental relationships to the Official alpine:3.8 image

It may be observed that the latter possesses many direct and indirect child images, creating therefore a dependency network around the "alpine:3.8" parent. In order to better visualize parental relationships on a global scale, a network modelling such relationships as well as the vulnerability inheritance between images may be created, where all the images in the gathered data set are represented as nodes, while the parental relationship between two nodes is represented by an edge in the network. Note that such a model will allow identifying correlations, patterns and clusters' behavior.

7.2.1 Modelling parental and vulnerability relationships in a network

In order to model the parental relationships and vulnerability inheritance between all the images in the gathered data set, the open-source and multi-platform program for visualizing large and complex networks known as Gephi was chosen [49]. Indeed, the latter is able to take any Comma Separated Values (CSV) file as input and use it to model either nodes or edges in a network. Thus, two CSV files in the name of "nodes.csv" and "edges.csv" were created with the following content in order to build the network:

- nodes.csv: list of images identified with their image ID, name and tag
- edges.csv: list of edges identified with the image ID of a child image, the image ID of a direct parent image and the total number of vulnerabilities inherited from the parent image

As shown in table 6.7 from chapter 6.3, 920 unique images were analyzed in total across all four types of repository. However, only images involved in a parental relationship are of interest when modelling the parental relationships and vulnerability inheritance between all the images in the gathered data set. Thus, base images (i.e. without parent) which do not have any child are not considered, therefore lowering the total number of images represented as nodes in the network illustrated in figure 7.8 below to 565. Note that the reason for having so many base images without children (355) is due to an error caused by many Community images having a non-identifiable parent, as discussed in details in 8.1.4. Moreover, 440 analyzed images depend on a parent, making therefore the total number of edges in the network equal to that amount. Also, the weight of each edge depends on the total number of vulnerabilities contained in the child image situated at one side of the edge, which are inherited from the parent image located on the other side. It should also be noticed that the dependency network only contains directed graphs, as parent images may simply not be based on their children.

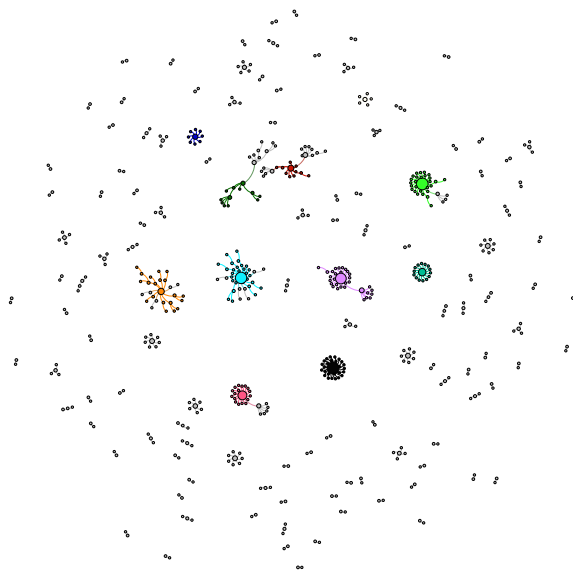


Figure 7.8: Parental relationships and vulnerability inheritance in the network of analyzed Docker images

As illustrated in figure 7.8 above and summarized in table 7.3 below, the most influential images (nodes) are the root parents which multiple child images are directly or indirectly based on. Note that such images form clusters of a significantly more important size in figure 7.8 with an associated color specified in table 7.3.

| Top ten largest clusters in the network | | | | |
|---|-------------------------|--------------------------|-----------------------------|------------|
| Rank | Image name | Directly connected nodes | Number of descendant images | Color |
| 1 | alpine:3.8 | 25 | 36 | Cyan blue |
| 2 | debian:9-slim | 25 | 31 | Green |
| 3 | alpine:latest | 23 | 31 | Purple |
| 4 | java:openjdk-8-jre | 11 | 26 | Orange |
| 5 | centos:7 | 26 | 26 | Black |
| 6 | debian:latest | 10 | 24 | Red |
| 7 | ubuntu:xenial | 18 | 23 | Pink |
| 8 | debian:stretch-20180716 | 5 | 20 | Dark green |
| 9 | ubuntu:latest | 14 | 14 | Turquoise |
| 10 | ubuntu:bionic-20190204 | 10 | 10 | Blue |

Table 7.3: The top 10 most popular parent images in the network of analyzed images with their total number of descendant children

Finally, it should be noticed that the visualization of the parental relationships and vulnerability inheritance between all the images in the gathered data set allows identifying the most influential parent images in the network in the form of clusters, which will be analyzed in details in the next subsection.

7.2.2 Analyzing egocentric networks

As listed in table 7.3 above, the ten most important egocentric networks corresponding to the ten most influential parent images in Docker Hub at the time of this writing are all issued from Official repositories. Indeed, two images issued from the Alpine repository are among the root parents on the platform, while 3 images issued from each the Official Ubuntu and Debian repositories are in that case. Note that figure 7.8 above shows many clusters of different shapes and sizes and only the ones offering interesting observations will be discussed in details here.

Top 1 parent: alpine:3.8

As illustrated in figure 7.9 below, the most influential parent image in the name of "alpine:3.8" possesses a lot of connections to other child images, while the latter usually have a maximum degree of two, suggesting that images based on "alpine:3.8" tend to only have a single child. Furthermore, it may be observed that there is a significant difference between the number of vulnerabilities images inherit from their direct parent, as illustrated by the edges' varying thicknesses in figure 7.9. Indeed, certain edges such as the ones between the "golang:1.6.0" and the "kubernetes/heapster" image or the "golang:1.5.2" and the "colemickens/oauth2_proxy" nodes

are heavy weighted, while others have a medium weight such as the edges pointing towards the "golang:1.12.2", the "node:10.15-slim" or the "golang:latest" image. However, it should be noticed that the entirety of the edges surrounding "alpine:3.8" contain an extremely low weight, showing that child images using "alpine:3.8" as their direct parent do not inherit many vulnerabilities. Thus, most of the vulnerabilities involved in images figuring in "alpine:3.8"'s dependency chain are not introduced by the root, but rather by their closest parent.

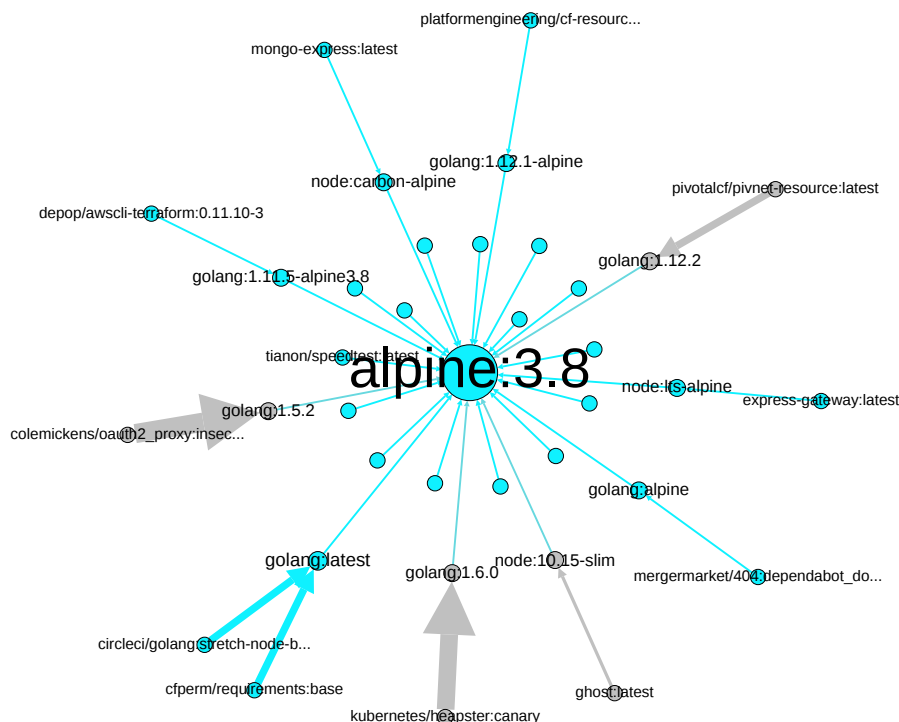


Figure 7.9: Parental relationships and vulnerability inheritance related to the Official alpine:3.8 image

Top 2 parent: debian:9-slim

Since an Alpine image has already been analyzed and the top 2 "debian:9-slim" as well as the top 3 "alpine:latest" image have the exact same number of descendant images (see table 7.3 above), the former was preferred to be analyzed here to provide a more complete and diverse analysis of the parental relationships and vulnerability inheritance between all the images in the gathered data set. The Official "debian:9-slim" image is very popular with more than 200 million downloads and contain 47 vulnerabilities in total. As illustrated in figure 7.10 below, the "debian:9-slim" parent possesses a lot of connections to other children, while the latter usually have a maximum degree of two, suggesting that images based on "debian:9-slim" tend to have a single child if any. Note that all the direct connections to the "debian:9-slim" image have a low weight, showing that children based directly on "debian:9-slim" do not inherit many vulnerabilities from their direct parent. However, it should be noticed that the edges pointing towards the "php:7-2-fpm" image and sourcing from the "nextcloud:16-beta-fpm", "yourls:fpm", and "backdrop:fpm" nodes

are heavily weighted, revealing that images based on "php:7-2-fpm" inherit a significant number of vulnerabilities from that parent. Similarly, the edges pointing towards the "ruby:2.3-slim" and "mysql:latest" nodes have a medium weight, showing that a moderate number of vulnerabilities are inherited from those images. Thus, most of the vulnerabilities involved in images figuring in "debian:9-slim"'s dependency chain are not introduced by the root, but rather by their closest parent.

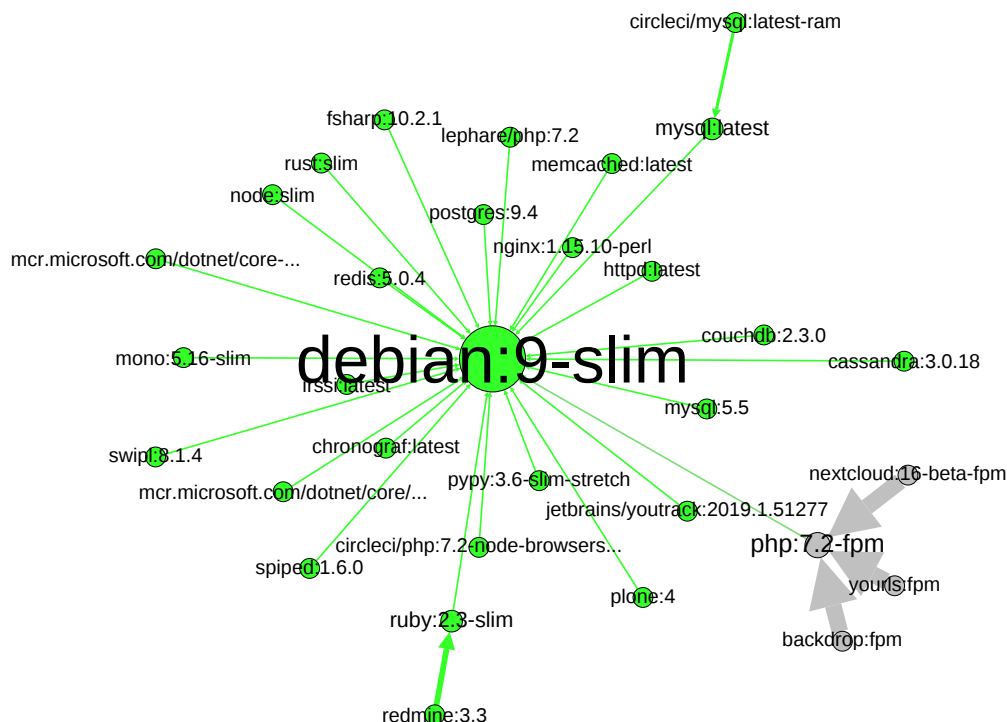


Figure 7.10: Parental relationships and vulnerability inheritance related to the Official debian:9-slim image

Top 4 parent: java:openjdk-8-jre

OpenJDK is an open-source implementation of the Java Platform Standard Edition and its Official "java:openjdk-8-jre" image contains 271 vulnerabilities. As illustrated in figure 7.11 below, the "java:openjdk-8-jre" root parent possesses a rather low number of connections to other child images, while the latter usually have a degree of two or three, suggesting that images based on "java:openjdk-8-jre" tend to have either one or two children. Note that almost all the direct connections to the "java:openjdk-8-jre" parent have a low weight, showing that child images based directly on "java:openjdk-8-jre" do not inherit many vulnerabilities from their direct parent in general. However, it may be observed that the "openjdk:8-jre-slim" image has a heavily weighted connection to the root "java:openjdk-8-jre" parent, as it inherits 90% of its total number of vulnerabilities (84 out of 94) from it. Moreover, two of "java:openjdk-8-jre"'s indirect children in the name of "jenkins/jenkins:lts-jdk11" and "dotc15/k8s-mysql:latest" inherit a huge number of vulnerabilities from their closest parent due to their heavily weighted connection to it. Thus, the

majority of the vulnerabilities involved in images figuring in "java:openjdk-8-jre"'s dependency chain are not introduced by the root parent, but rather by their closest parent.

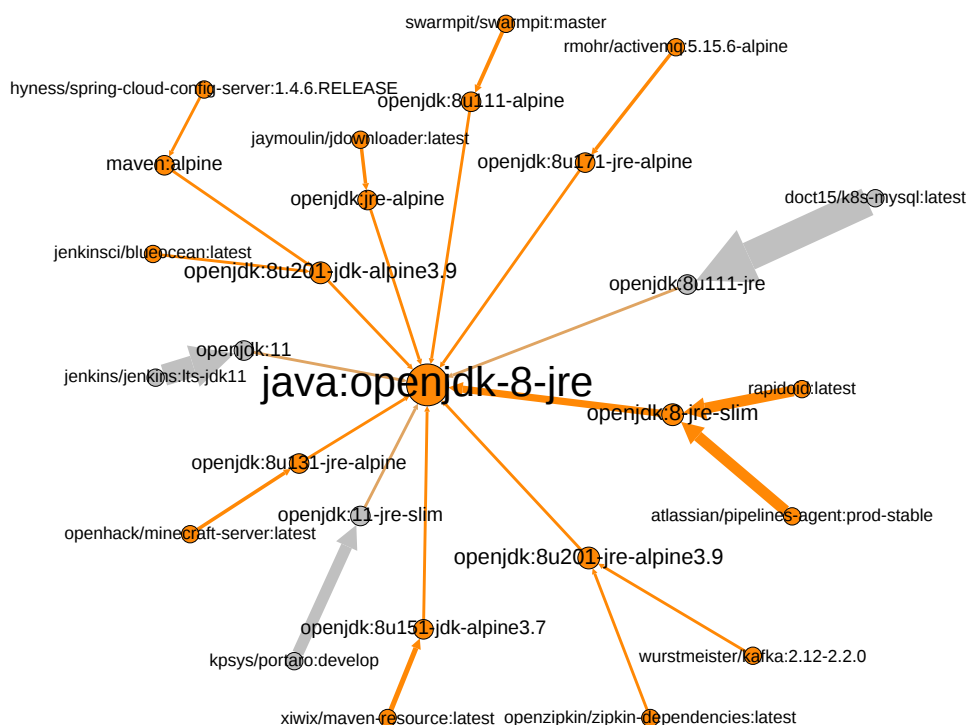


Figure 7.11: Parental relationships and vulnerability inheritance related to the Official java:openjdk-8-jre image

Top 6 parent: debian:latest

As illustrated in figure 7.12 below, the egocentric network of the Official "debian:latest" image has only 10 direct children, but a total number of 24 descendants, making this image the top 6 most used parent on Docker Hub at the time of this writing. It may be observed that the majority of "debian:latest"'s direct children usually have a degree of one or two, suggesting that images based on "debian:latest" usually have a single child if any. Nonetheless, it is important to note that the "debian:latest" root has two direct children in the name of "buildpack-deps:scm" and "buildpack-deps:stretch" forming their own clusters with a high number of local connections to other children. Surprisingly, none of those two images inherit vulnerabilities from their direct "debian:latest" parent, as their connection to the root of the network has a very low weight.

On one hand, all the connections pointing towards the "buildpack-deps:stretch" image have a low weight, indicating that its children do not inherit vulnerabilities from that image. Additionally, it should be noticed that only one of the "buildpack-deps:stretch" image's children have a child of its own. Indeed, the "erlang:19" image containing 589 vulnerabilities passes on non less than 99% of its vulnerabilities (583) to the "elixir:1.4.5" image. On the other hand, the connections pointing towards the "buildpack-deps:scm" image have a heavier weight than the

other cluster, indicating that child images based on the "buildpack-deps:scm" image inherit a significant number of vulnerabilities, although the latter do not come from the "debian:latest" root.

Finally, it should be noticed that the Community "microsoft/dotnet-nightly:1.0-sdk" image directly based on "buildpack-deps:oldstable-scm" and indirectly relying on the "debian:latest" image inherits almost 98% of its total number of vulnerabilities from its direct parent, while the "buildpack-deps:oldstable-scm" image inherits only 30% of its vulnerabilities from "debian:latest". Similarly to the other discussed clusters so far, the majority of the vulnerabilities involved in images figuring in "debian:latest"'s dependency chain are not introduced by the root parent, but rather by their closest parent.

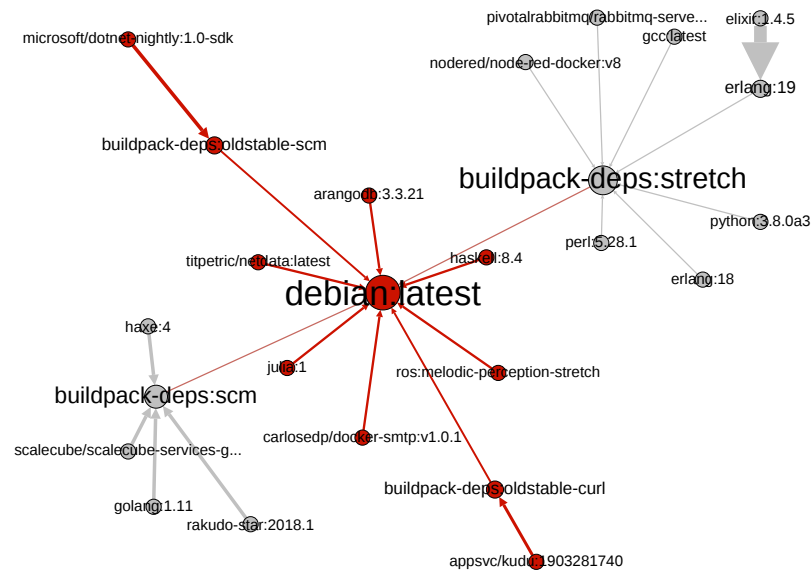


Figure 7.12: Parental relationships and vulnerability inheritance related to the Official debian:latest image

Top 7 parent: ubuntu:xenial

As illustrated in figure 7.13, most of the images based on "ubuntu:xenial" do not have any children, but inherit almost all of their vulnerabilities due to their heavily weighted connections to the root parent. Nonetheless, a single exception may be observed in the name of the "ibmjava:latest" image, which does not inherit any vulnerability from its direct "ubuntu:xenial" parent and forms a distinct cluster with heavily weighted connections to its own children. Indeed, non less than 98% of the vulnerabilities found the "ibmjava:latest" image (42 out of 43) are passed on to each and every of its children. Overall, the majority of the vulnerabilities involved in images figuring in "ubuntu:xenial"'s dependency chain are directly introduced by the root parent and inherited further by the latter's children, although the vulnerability spread is limited as most of the images based on "ubuntu:xenial" do not have children of their own.

vulnerabilities involved in images figuring in "debian:stretch-20180716"'s dependency chain are not introduced by the root, but rather by their closest parent.

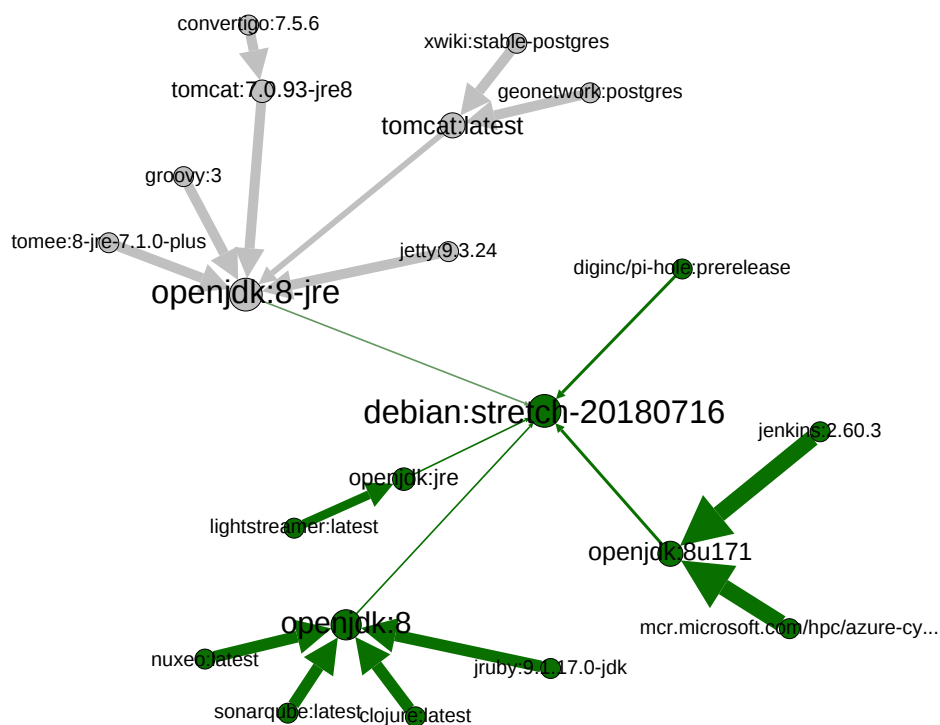


Figure 7.14: Parental relationships and vulnerability inheritance related to the Official debian:stretch-20180716 image

In conclusion, it should be noticed that vulnerabilities in the Docker Hub platform are still inherited between images in a highly manner, although Shu et al.'s suggestion discussed in 6.5 about vulnerabilities potentially propagating from a small set of highly influential images to the whole platform is not confirmed by the above analysis.

Chapter 8

Discussion

This chapter provides a critical analysis of the presented study by discussing its validity, as well as the important challenges that were encountered throughout the study, which may be beneficial for other researchers working with the same topic to be aware of. A discussion of suggested future work will also be addressed here.

8.1 Validity of the study

An objective critique of the conducted study is necessary in order to analyze and point out any factors impacting its validity.

8.1.1 Analyzed set of Docker images

The presented study only analyzes the most recent image in each considered repository, constituting a set of Docker images which may theoretically be unrepresentative of the most secure image in a given repository. Although such images should consist of a repository's most secure image due to their most up-to-date property, they may indeed consist of an updated version of their containerized software, which does not take into consideration the application of security patches for other dependencies. Furthermore, a vulnerable image may technically be updated by only adding some dummy files and directory while re-tagging it with a new version number, which may provide a false sense of the image having actually been updated for vulnerability patching. Nonetheless, this study assumes that maintainers aim at providing secure software as much as possible, therefore assuming that dummy updates leading to the most recent image of a repository not being the most secure is minimal and does not impact the validity of the study.

8.1.2 Applied methodology

The applied methodology introduced in chapter 3 has shown itself to be successful. Indeed, the formulation of a problem statement in form of three main research questions allowed decomposing the problem into detailed research questions addressable separately. Furthermore, the

development of an appropriate software to conduct experiments and gather metadata, parental and vulnerability information for the set of defined Docker images into a single noSQL database allowed addressing every detailed research question through their conversion into noSQL research queries. The latter were used to process a huge and disparate amount of data in a systematic and accurately reproducible way across multiple rounds of experiments. The methodology used throughout this thesis allowed therefore meeting our objectives discussed in 3.1, by grasping the complete spectre implied by the original problem statement, while making this study entirely reproducible and providing the research community with a base tool (DAZER) for conducting similar research about the Docker Hub platform.

8.1.3 Software vulnerability identification

The identification of vulnerabilities present in the analyzed Docker images being part of this study depends heavily on the use and detection quality of Clair scanner discussed in details in 3.3.1. Although other alternatives were considered, Clair scanner consists of the state of the art of container image vulnerability scanners at the time of this writing and has been used in similar research investigating Docker Hub’s security landscape, such as the one conducted by Shu et al. in 2016 [11]. Due to time constraints, the detailed analysis of Clair scanner’s inner workings was limited, but some important limitations should be noted. As any vulnerability scanner, Core OS’s software reports some false positives and negatives. Indeed, some packages such as the ones related to the Linux kernel may be detected as vulnerable by Clair scanner, although such packages are actually not contained in the image. As explained in 2.2.1, Docker containers run on top of a shared Linux kernel, eradicating therefore the need for a Docker image to contain any packages related to kernel. Nonetheless, many images include dummy Linux kernel packages in order to satisfy other dependencies such as the ones required by package management systems and therefore avoid unmet requirements. Since similar studies have not taken this parameter into consideration and one of this thesis’ goals was to compare its results with other studies, such vulnerabilities have not been ignored from our results. However, it should be noted that the total number of vulnerabilities reported in this study should be relativized.

8.1.4 Unidentifiable parent images

Many Community repositories had unidentifiable parents leading to incorrect results related to image and vulnerability inheritance. Indeed, judging from their available Dockerfiles on the Docker Hub platform, some Community images which did not have any parent during our experiments were supposed to be based on another image. The problem was that such child images were usually based on very old parents which had been been updated by their maintainer after building the child image. Thus, the layer signature inherited by such child images was not the same as the one available for their updated parent on Docker Hub. Furthermore, since many Community images are not rebuilt that often, many repositories of that type contained child images with a non-identifiable parent using the implemented approach described in 4.2.1. The results related to image and vulnerability inheritance discussed in 6.3 should therefore be relativized when it comes to Community repositories.

8.1.5 Discovered vulnerabilities and exploitability

The number of vulnerabilities discovered in the different types of images presented in this study as well as their severity should be relativized, as many of the reported vulnerabilities may not be exploitable in a container environment. As explained in 2.2.2, Docker containers are different than traditional architectures such as VM or physical machines. Thus, an exploitable vulnerability in a traditional environment may not be exploitable when containerized due to isolation restrictions implied by containers. Furthermore, false positives reporting vulnerabilities in dummy packages such as the ones related to the Linux kernel are obviously not exploitable in a container environment. The severity of many of the reported vulnerabilities in the presented study should therefore be relativized while set in perspective with an actual exploitation context in a containerized environment.

8.2 Encountered challenges

Throughout the realization of this project, a certain number of non-negligible challenges were encountered. Note that future work about Docker Hub or any research related to the online platform should be aware of the following problems.

8.2.1 Retrieving data from Docker Hub

As explained in 5.2, Docker Hub provides a public but completely undocumented REST API for retrieving metadata about the repositories and images available on the public registry. The identification of valid HTTP requests through a simple trial and error approach was very time consuming, but resulted in a completely automated way of gathering metadata about any type of Docker image directly from the Docker Hub platform, without requiring any download. Although our study required image downloads for local vulnerability analysis via Clair scanner, the complete list of valid requests detailed in 5.2, as well as its developed API in Python available in Appendix E.1 will provide future researchers with a solid base for a better automation of their studies and analysis.

8.2.2 Manual image checkout

As expected in 3.2.2 and executed in 4.2, a valid Docker Hub account with manually checked out Certified and Verified repositories prior to the conduction of automated experiments was necessary. Note that the complete list of non-paid repositories requiring manual checkout at the time of this writing is available in Appendix A.5 for easy reference in future automated studies.

8.2.3 Overwhelming the Docker engine

In very specific situations, interacting with the Docker engine through its SDK in Python may lead to an unexpected timeout error. For example, the latter may occur when the Docker engine is requested to delete a very large image (e.g. "store/saplabs/hanaexpresssa:2.00.035.00.20190115.1" of 25.7 GB), when it is already busy executing other actions such as downloading images. Although the actual time for handling such a situation greatly depends on the specification of the

machine running the Docker engine, the latter is set to use a default timeout of 60 seconds for handling new requests. The deletion of a very large image exceeding 10 GB is most likely bound to use more than one minute to complete, which may therefore lead to a timeout error exception being thrown by the Docker SDK in Python.

Increasing the timeout to 5 and 10 minutes was unfortunately not enough to give our VM with 8 VCPUs, 16 GB of RAM and 160 GB of disk space time to handle such demanding deletion requests. It was a final timeout of 30 minutes which actually succeeded in order to give largely demanding deletion requests time to complete. Note that using a physical machine with a Solid State Drive (SSD) may greatly improve the time required to handle deletion requests, as it seemed that the bottle neck was largely due to the virtualization aspect of disk management through OsloMet's OpenStack platform in our case.

8.2.4 Image parent retrieval

Retrieving an image's parent constituted without a doubt the most challenging part of our work. Indeed, previous research examining the parental relationship of Docker images such as [11] were executed using older versions of the Docker engine, where images contained a direct reference to their parent, as explained in 2.4.2. Since Docker version 1.10 released in February 2016 however, images do not contain such a reference anymore. Thus, a different approach than the one used in [11] was applied in this study, as discussed in details in 4.2.1. Before ending up with two updatable pseudo databases in the form of two JSON files containing the layer combination of each image contained in each of the Official and Verified repositories, other attempts were experimented.

First, an approach based on the analysis of image vulnerabilities was considered. Indeed, vulnerabilities identified with a CVE number are always related to a specific software and version. Thus, it was assumed that a child image based on a vulnerable parent containerizing a certain piece of software would contain vulnerabilities directly related to the parent software. Unfortunately, vulnerability scanners for Docker images such as Clair scanner do not necessarily pick up all vulnerabilities, as discussed in 8.1. Moreover, such an approach would have made it difficult to separate base images from child images based on a non-vulnerable parent, although the goal of that technique was only to identify inherited vulnerabilities from the ones introduced by a child. The final approach used to retrieve an image's parent turned out therefore to be more robust, as it allows identifying not only an image's direct parent, but also its grand-parents specifically.

Secondly, a single non-updatable database was considered in the form of a JSON file containing the layer combination of each image contained in each of the Official and Verified repositories, which any type of image may be based on. Although that second attempt was successful in terms pure parent identification capabilities, many optimizations were added to that approach. The single database was separated into two JSON files, one containing the layer combination of all the Official images and one containing the layer combination of all the Verified images. As explained in 4.2.1, the advantage of having two separate databases was that it provided a more efficient way of indexing images' layers and identify parent images, by only executing parent lookups towards the appropriate database instead of both. Furthermore, the original single parent database was not updatable, requiring its re-population from scratch before the conduction of each new experiment, so that parent image signature would be up to date. Complete re-populations took however more than four days in total, giving the first populated images time to be updated on Docker Hub and change their layer signature during that period of time. In order to avoid that problem and minimize errors related to parent identifications, the two final parent databases could

be updated separately based on the type of experiment being run (e.g. experiments analyzing Official images only updated the Official parent database, whereas a Verified experiment needed to update both), reducing the populating time from several days to a few minutes for the Official parent database and about one hour for the Verified database.

Finally, note that the amount of time used to update a parent database greatly depended on the date of its last update, as a recently updated database was most likely to be up to date faster than a database left without update for several weeks. Note also that the final design of the parent databases is available in chapter 4 under 4.2.1.

8.2.5 Confusing terminology

As explained in chapter 2 under 2.5.3, Docker Inc. tends to use the terms "images" and "repositories" interchangeably due to the misconception that an image related to a certain piece of software may be identified through its repository name only. That misconception is often reflected on the Docker Hub platform as marked in red in figure 8.1. Nonetheless, an image is identified through and only through the use of a tag, as a repository name such as Couchbase only identifies a certain containerized piece of software, but not a specific version. Docker images identify however precised version of a containerized piece of software, such as "couchbase:6.0.0" or "couchbase:enterprise-6.0.0" in the case of Couchbase. Docker Hub provides therefore a lot more images in total than the claimed 2.1 million, as that number actually consist of the number of registered repositories on the platform at the time of this writing and not the total number of available images through those repositories.

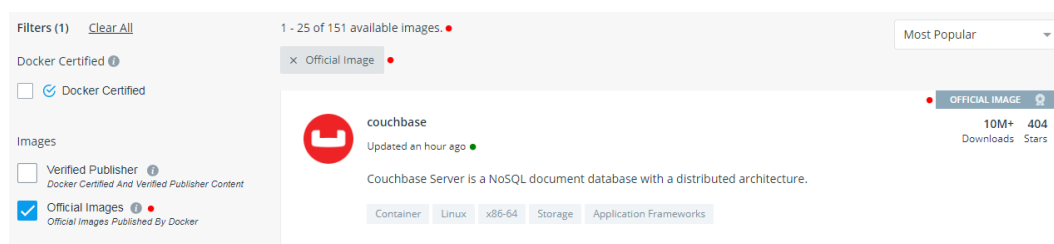


Figure 8.1: Docker Hub's confusing terminology

Moreover, individual repositories on Docker Hub contain a confusing timestamp on their result page when searching or filtering repositories, as marked in green in figure 8.1. Indeed, each repository contain an "Updated x minutes ago" string under their name, which is misleading as it is easy to think that it refers to the last time the repository was updated by its maintainer. However, that string corresponds to the last time Docker Hub's backend engine scanned the repository and reported it as functional. Indeed, the real timestamp showing when a repository's image was last updated is located under the "tags" tab of a repository.

Finally, Docker Hub uses categories in order to classify its repositories based on the type of software they hold. Such categories may be used to filter out repositories on the platform's Web interface or via its undocumented API and may therefore find themselves very convenient. Nonetheless, one category referred to as "base images" is very misleading, as it is easy to think that it contains all the base repositories on Docker Hub (i.e. repositories containing images which do not have a parent image). However, that category does not correspond to what might

be expected, as it contains repositories which are not base ones (e.g. "swift" or "buildpack-deps") and misses truly base repositories such as "ubuntu-debootstrap" or "opensuse".

8.3 Future work

Consequently to the results presented in this study, other research questions and suggestions have emerged for future work.

First, it seems that there is a strong need for more reliable vulnerability scanners for Docker images, as the state of the art solutions available at the time of this writing seem to provide too many false positives and negatives, while not taking into account the contextualization of discovered vulnerabilities and their exploitability in a containerized environment. Further research in that direction is therefore strongly recommended, as it would greatly improve the reliability of other studies based on vulnerability scanners for Docker images such as Clair scanner, therefore benefiting the whole research community. Moreover, a detailed comparative study of the main vulnerability scanners available for Docker images should be conducted prior to the investigation of alternative approaches, in order to get a better understanding of the different scanning and detection algorithms used in today's solutions.

Secondly, a more thorough analysis of the new types of repositories introduced by Docker Inc. in December 2018 is needed, as only the most recent image in each repository was analyzed in this study. Although such images should consist of a repository's most secure image due to their most up-to-date property, there is a possibility that it is not the case as argued in 8.1. Conducting a more thorough analysis of the Certified and Verified repositories may therefore reveal that the most recent images on Docker Hub might not be the most secure ones. Furthermore, the inclusion of Windows repositories as well as images available for multiple processor architectures may constitute an interesting research orientation, in order to see whether Docker Hub's security landscape varies with different platforms and processor architectures. Note also that conducting a small analysis of the paid Verified and Certified repositories listed in Appendix A.1 might also constitute an interesting research orientation in order to see whether such repositories are worth the extra money when it comes to security.

Thirdly, although multiple experiment were conducted at different times of the day, the re-conduction of the same or a similar experiment at different timestamps during a longer period of time may reveal some variations in Docker Hub's effective security landscape due to non-linear repository updates.

Finally, investigating the actual impact of such a large number of vulnerabilities found in many images hosted on Docker Hub constitutes a fundamental research question which needs to be answered. Indeed, many Docker images contain a large number of vulnerabilities of which many have a high severity level. As pointed out in 8.1 however, many of the reported vulnerabilities exploitable in a traditional environment may not be exploitable when containerized. Investigating the actual proportion of exploitable vulnerabilities in this study as well as similar previous research using the CVSS system or something similar may therefore constitute an interesting research topic, which may relativize the currently alarming security landscape of the Docker Hub platform.

Chapter 9

Conclusion

Based on previous research about Docker Hub’s security landscape and the security mechanisms introduced by Docker Inc. in response to those investigations, this thesis addressed the following research questions:

1. Have the security measures introduced by Docker Inc. in response to previous research improved Docker Hub’s security landscape and to what extent?
2. Are vulnerabilities still inherited from images’ parent(s) and in what proportion?
3. How are discovered vulnerabilities distributed across repository types?

First, our result and analysis show that the security measures introduced by Docker Inc. in the form of two new kinds of Certified and Verified repositories have not improved the overall Docker Hub’s security landscape in a way that is significant. Indeed, less Certified repositories are completely free of vulnerabilities compared to Verified, Official and Community repositories, although they contain far less vulnerabilities than their other peers in average even for the most vulnerable Certified repositories (6.2.1, 6.2.4, 7.1.2). Nonetheless, Certified images tend to contain vulnerabilities of higher severity, especially when it comes to critical vulnerabilities (6.2.2, 6.2.3). Furthermore, Verified and Official repositories contain a similar and significantly lower number of unique vulnerabilities than Community images, with a higher chance of being of a lower severity level (6.2.1, 6.2.2, 7.1.3). However, Verified images with less than 180 vulnerabilities are more likely to hold a more important number of total vulnerabilities than images in the same case from one of the three other types of repositories.

Secondly, previous research such as [11] conducted in 2016 pointed out that a large amount of vulnerabilities commonly propagated from parent to child images, with an average of 180 inherited vulnerabilities in average, against 20 new ones which are introduced by child images. Our results show that the average number of inherited vulnerabilities across all types of repositories has dropped to 108 since 2016, while the average number of introduced vulnerabilities has completely exploded from 20 to 160, indicating a global increasing trend of the number of disclosed vulnerabilities found in images hosted on the Docker Hub platform (6.3, 7.2.2). Moreover, it was found that the most influential parent images on Docker Hub are all Official images and that although vulnerabilities in the platform are still inherited in a highly manner, they do not tend to be introduced by the top root parents on Docker Hub, as suggested by Shu et al. in their paper’s future work section [11]. Note however that our analysis took only into consideration the

most updated image in each repository, while Shu et al. analyzed multiple images within many Official and Community repositories. Moreover, the results presented in this thesis are proceeding from the analysis of four types of repositories, while Shu et al. focused only on Official and Community, as Certified and Verified repositories were not a part of the Docker Hub platform at the time of the conduction of their experiment.

Thirdly, the measurements and analysis conducted in this thesis revealed that the majority of images across Official, Community and Certified repositories contain less than 75 vulnerabilities in total, while the majority of Verified images hold less than 180 vulnerabilities (7.1.3). Verified images with less than 180 vulnerabilities are therefore more likely to hold a more important number of total vulnerabilities than images in the same case from one of the three other types of repositories. Furthermore, the number of Official images with a higher number of contained vulnerabilities than the 180 threshold decreases less rapidly than the three other types of repositories, indicating that Official images are more likely to contain a higher number of total vulnerabilities than their peers. The average number of unique vulnerabilities contained in Community images is expected to increase with a rate of approximately 191 vulnerabilities per year between 2019 and 2025 if Docker Hub's security landscape continues evolving the same way, while that same number for Official, Verified and Certified repositories will grow with approximately 96, 115 and 17 vulnerabilities per year respectively (7.1.5). Thus, the future vulnerability trend is an increase of the number of unique vulnerabilities found across all types of repositories, leading therefore to a deterioration of Docker Hub's global security landscape when only the total number of unique vulnerabilities is taken into account.

Fourthly, multiple contributions have been made to the research community, besides the analytical result of our study. Indeed, the discovery and detailed documentation of Docker Hub's hidden REST APIs, as well as a developed Python API for Docker Hub, making future interactions with the platform easier for the research community have been described in 5.2 and Appendix E.1.

Finally, note that this research is entirely reproducible thanks to our developed open source software name DAZER (Docker imAge analyZER), providing a solid tool base for other studies surrounding the analysis of Docker images, as well as the Docker Hub platform.

References

- [1] C. Pettey, "6 Best Practices for Creating a Container Platform Strategy," Oct. 31, 2017. [Online]. Available: <https://www.gartner.com/smarterwithgartner/6-best-practices-for-creating-a-container-platform-strategy/>. [Accessed Mar. 20, 2019].
- [2] Docker Inc., "Build and Ship any Application Anywhere," 2019. [Online]. Available: <https://hub.docker.com/>. [Accessed Jan. 20, 2019].
- [3] Docker Inc., "Our Company," 2019. [Online]. Available: <https://www.docker.com/company>. [Accessed Mar. 20, 2019].
- [4] Docker Inc., "Official Images on Docker Hub," 2019. [Online]. Available: https://docs.docker.com/docker-hub/official_images/. [Accessed Apr. 05, 2019].
- [5] Red Hat, "The State Of Containerization - How, Where, And Why Are Containers Leveraged In The Software Development Life Cycle?," Jun. 2016. [Online]. Available: <https://www.redhat.com/cms/managed-files/forrester-tap-state-of-containerization-analyst-paper-201610-en.pdf>. [Accessed Mar. 20, 2019].
- [6] Security Center, "Cryptojacking invades cloud. How modern containerization trend is exploited by attackers," Jun. 12, 2018. [Online]. Available: <https://kromtech.com/blog/security-center/cryptojacking-invades-cloud-how-modern-containerization-trend-is-exploited-by-attackers>. [Accessed Mar. 20, 2019].
- [7] Docker Inc., "Create a base image," 2019. [Online]. Available: <https://docs.docker.com/develop/develop-images/baseimages/>. [Accessed Jan. 20, 2019].
- [8] Docker Inc., "Docker Security Scanning," 2018. [Online]. Available: <https://docs.docker.com/v17.12/docker-cloud/builds/image-scan/>. [Accessed Mar. 20, 2019].
- [9] J. Morgan, "Introducing the New Docker Hub - Docker Blog", Dec. 13, 2018. [Online]. Available: <https://blog.docker.com/2018/12/the-new-docker-hub/>. [Accessed Mar. 20, 2019].
- [10] J. Gummaraju, T. Desikan and Y. Turner, 'Over 30% of official images in docker hub contain high priority security vulnerabilities', in *Technical Report*, BanyanOps, 2015.
- [11] R. Shu, X. Gu and W. Enck, 'A study of security vulnerabilities on docker hub', in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, ACM, 2017, pp. 269–280.
- [12] NIST, "vulnerability," 2019. [Online]. Available: <https://csrc.nist.gov/glossary/term/vulnerability>. [Accessed Apr. 05, 2019].
- [13] A. Arora, R. Krishnan, A. Nandkumar, R. Telang and Y. Yang, 'Impact of vulnerability disclosure and patch availability - an empirical analysis', in *Third Workshop on the Economics of Information Security*, vol. 24, 2004, pp. 1268–1287.

- [14] Mitre, "Common Weakness Enumeration," Apr. 03, 2018. [Online]. Available: <https://cwe.mitre.org/>. [Accessed Apr. 05, 2019].
- [15] FIRST, "Forum of Incident Response and Security Teams," 2019. [Online]. Available: <https://www.first.org/>. [Accessed Apr. 05, 2019].
- [16] NIST, "NVD Vulnerability Severity Ratings," 2019. [Online]. Available: <https://nvd.nist.gov/vuln-metrics/cvss>. [Accessed May. 06, 2019].
- [17] Atlassian, "Severity Levels for Security Issues," 2019. [Online]. Available: <https://www.atlassian.com/trust/security/security-severity-levels>. [Accessed May. 01, 2019].
- [18] The UNIX and Linux Forums, "Linux and UNIX Man Pages - Unix Version 7 - man page for chdir (v7 section 2)," 2019. [Online]. Available: <https://www.unix.com/man-page/v7/2/chdir/>. [Accessed Apr. 05, 2019].
- [19] M. Tosatti, "Summary of changes from v2.4.19-rc4 to v2.4.19-rc5," 2002. [Online]. Available: <https://mirrors.edge.kernel.org/pub/linux/kernel/v2.4/ChangeLog-2.4.19>. [Accessed Apr. 05, 2019].
- [20] M. Kerrisk, "Linux Programmer's Manual, cgroups - Linux control groups," May. 05, 2019. [Online]. Available: <http://man7.org/linux/man-pages/man7/cgroups.7.html>. [Accessed May. 11, 2019].
- [21] Docker Inc., "What is a Container?," 2019. [Online]. Available: <https://www.docker.com/resources/what-container>. [Accessed Jan. 20, 2019].
- [22] Docker Inc., "dotCloud, Inc. is now Docker Inc.," 2019. [Online]. Available: <https://www.docker.com/docker-news-and-press/dotcloud-inc-now-docker-inc>. [Accessed Jan. 20, 2019].
- [23] RightScale Inc., "RightScale 2018. State of the Cloud Report," 2018. [Online]. Available: <https://assets.rightscale.com/uploads/pdfs/RightScale-2018-State-of-the-Cloud-Report.pdf>. [Accessed Jan. 20, 2019].
- [24] Docker Inc., "About Docker Engine," 2019. [Online]. Available: <https://docs.docker.com/engine/>. [Accessed Apr. 03, 2019].
- [25] Docker Inc., "Dockerfile reference," 2019. [Online]. Available: <https://docs.docker.com/engine/reference/builder/>. [Accessed Jan. 20, 2019].
- [26] Docker Inc., "Docker overview," 2019. [Online]. Available: <https://docs.docker.com/engine/docker-overview/>. [Accessed Jan. 20, 2019].
- [27] Docker Inc., "Docker Registry," 2019. [Online]. Available: <https://docs.docker.com/registry/>. [Accessed Jan. 20, 2019].
- [28] Docker Inc., "Develop with Docker Engine SDKs and API," 2019. [Online]. Available: <https://docs.docker.com/develop/sdk/>. [Accessed Apr. 03, 2019].
- [29] Docker Inc., "Docker image introduction," Apr. 26, 2015. [Online]. Available: <http://docs.docker.com/terms/image/#base-image-def>. [Accessed via the Internet Archive Apr. 03, 2019].
- [30] Docker Inc., "Migrate to Engine 1.10," 2017. [Online]. Available: <https://docs.docker.com/v17.09/engine/migration/>. [Accessed Apr. 03, 2019].
- [31] Docker Inc., "Image Manifest V2, Schema 2," 2019. [Online]. Available: <https://docs.docker.com/registry/spec/manifest-v2-2/>. [Accessed Apr. 03, 2019].
- [32] Docker Inc., "docker tag," 2019. [Online]. Available: <https://docs.docker.com/engine/reference/commandline/tag/>. [Accessed Apr. 03, 2019].

- [33] Docker Inc., "Explore - Docker Hub," 2019. [Online]. Available: <https://hub.docker.com/search?q=&type=image>. [Accessed Apr. 05, 2019].
- [34] Docker Inc., "Repositories," 2019. [Online]. Available: <https://docs.docker.com/docker-hub/repos/>. [Accessed Apr. 05, 2019].
- [35] Docker Inc., "Docker glossary," 2019. [Online]. Available: <https://docs.docker.com/glossary/?term=parentimage>. [Accessed Jan. 20, 2019].
- [36] T. Lauinger, A. Chaabane, S. Arshad, W. Robertson, C. Wilson and E. Kirda, 'Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web', *arXiv preprint arXiv:1811.00918*, 2018.
- [37] R. G. Kula, D. M. German, A. Ouni, T. Ishio and K. Inoue, 'Do developers update their library dependencies?', *Empirical Software Engineering*, vol. 23, no. 1, pp. 384–417, 2018.
- [38] Docker Inc., "HTTP API V2: Docker Registry HTTP API V2," 2019. [Online]. Available: <https://docs.docker.com/registry/spec/api/>. [Accessed Apr. 05, 2019].
- [39] X. Lin, L. Lei, Y. Wang, J. Jing, K. Sun and Q. Zhou, 'A measurement study on linux container security: Attacks and countermeasures', in *Proceedings of the 34th Annual Computer Security Applications Conference*, ACM, 2018, pp. 418–429.
- [40] A. Zerouali, T. Mens, G. Robles and J. Gonzalez-Barahona, 'On the relation between outdated docker containers, severity vulnerabilities and bugs', *arXiv preprint arXiv:1811.12874*, 2018.
- [41] T. Kuznets, "Docker Security Scanning safeguards the container content lifecycle," May. 10, 2016. [Online]. Available: <https://blog.docker.com/2016/05/docker-security-scanning/>. [Accessed Apr. 05, 2019].
- [42] Docker Inc., "Docker Security Scanning," 2018. [Online]. Available: <https://docs.docker.com/v17.12/docker-cloud/builds/image-scan/>. [Accessed Apr. 05, 2019].
- [43] B. Peterson, "PEP 373 – Python 2.7 Release Schedule," Nov. 3, 2008. [Online]. Available: <https://www.python.org/dev/peps/pep-0373/>. [Accessed Apr. 01, 2019].
- [44] CoreOS, "Vulnerability Static Analysis for Containers: Clair," 2019. [Online]. Available: <https://github.com/coreos/clair>. [Accessed Apr. 01, 2019].
- [45] A. Coralic, "Clair scanner: Docker containers vulnerability scan," 2019. [Online]. Available: <https://github.com/arminc/clair-scanner>. [Accessed Apr. 01, 2019].
- [46] CoreOS, "Understanding drivers, their data sources, and creating your own," 2019. [Online]. Available: <https://github.com/coreos/clair/blob/master/Documentation/drivers-and-data-sources.md>. [Accessed Apr. 01, 2019].
- [47] Mitre, "CWE-295: Improper Certificate Validation," Dec. 27, 2018. [Online]. Available: <https://cwe.mitre.org/data/definitions/295.html>. [Accessed Apr. 08, 2019].
- [48] The Computer Incident Response Center Luxembourg, "CIRCL – Computer Incident Response Center Luxembourg – CSIRT – CERT," 2018. [Online]. Available: <http://circl.lu>. [Accessed Apr. 09, 2019].
- [49] Gephi, "The Open Graph Viz Platform," 2017. [Online]. Available: <https://gephi.org/>. [Accessed May. 18, 2019].
- [50] NIST, "NVD CWE Slice," 2019. [Online]. Available: <https://nvd.nist.gov/vuln/categories>. [Accessed May. 06, 2019].

Appendices

Appendix A

Excluded repositories

This appendix provides a list of all the repositories skipped during the conducted experiments.

A.1 Paid repositories

Note that all paid repositories are of type Verified.

- store/koekiebox/fluid (Certified)
- store/klokantech/tileservers-maptiler
- store/avinetworks/seprivate
- store/softwareag/adabasmanager-ce
- store/portworx/oci-monitor (Certified)
- store/gitpitch/desktop
- store/esystemstech/liferay-cc (Certified)
- store/datastax/dse-server
- store/aretera/foopipes (Certified)
- store/site24x7/docker-agent (Certified)
- store/codecov/enterprise
- store/sysdig/agent (Certified)
- store/datastax/dse-server
- store/codecov/enterprise
- store/avinetworks/seprivate
- store/datastax/dse-server
- store/codecov/enterprise

A.2 Manifest not found error

- `mcr.microsoft.com/iot/opc-twin-registry`
- `mcr.microsoft.com/iot/opc-twin-onboarding`
- `mcr.microsoft.com/azureiotedge/cameracapturee`
- `mcr.microsoft.com/aiforearth/blob-py`
- `mcr.microsoft.com/windows`
- `mcr.microsoft.com/azure-stream-analytics/simulated-sensor`
- `mcr.microsoft.com/oryx/nodejs`
- `mcr.microsoft.com/hpcacm`
- `mcr.microsoft.com/azuredocs/appservice/samples/multicontainerwordpress`
- `store/kaazing/gateway`
- `mcr.microsoft.com/iot/opc-twin-webui:1.0.1`
- `store/gitlab/gitlab-ee`
- `mcr.microsoft.com/cntk/nightly:2.7-rc0.dev`
- `store/softwareag/commandcentral:10.3.0.2-alpine`
- `mcr.microsoft.com/azure-stream-analytics/azureiotedge`
- `store/aerospike/aerospike`
- `store/appdynamics/java:4.5_centos7`
- `store/buddy/updater:1.4.12`
- `store/hpsoftware/oa_store`
- `store/appdynamics/machine:4.5_centos7`
- `store/portworx/px-dev`

A.3 No matching manifest or incompatible platform error

- `mcr.microsoft.com/windows/nanoserver:1809`
- `mcr.microsoft.com/dotnet/framework/wcf:4.7.2`
- `mcr.microsoft.com/windows/iotcore:1809`
- `mcr.microsoft.com/windows:1809`
- `mcr.microsoft.com/dotnet/framework/sdk:4.7.2`
- `mcr.microsoft.com/dotnet/framework/runtime:4.7.2`
- `mcr.microsoft.com/aivision/visionmodule:0.1`
- `mcr.microsoft.com/windows/servercore:ltsc2019`

- mcr.microsoft.com/windows/servercore/iis:windowsservercore-ltsc2019
- mcr.microsoft.com/dotnet/framework/samples:dotnetapp

A.4 Pull access denied error

- microsoft-dotnet-framework
- microsoft-dotnet-core
- microsoft-azure-pipelines-vsts-agent
- microsoft-azure-cognitive-services
- microsoft-osa-cli
- microsoft-windows-base-os-images
- microsoft-mcrdemoproductfamily
- microsoft-dotnet-core-nightly
- microsoft-cntk

A.5 Manual checkout of repositories (kept)

Certified repositories

- Oracle Java 8 SE (Server JRE)
- MySQL Server Enterprise Edition
- Db2 Developer-C Edition
- Oracle WebLogic Server
- Oracle Fusion Middleware Infrastructure
- IBM WebSphere Application Server Liberty
- Oracle Instant Client
- Oracle Coherence
- Sematext Agent
- IBM Security Access Manager
- Oracle Database Enterprise Edition

Verified repositories

- softwareag/apama-correlator
- IBM Db2 Warehouse client container
- softwareag/apama-builder

- Splunk Enterprise
- softwareag/aris-octopus
- softwareag/aris-loadbalancer
- softwareag/sample-ehcache-client
- IBM Tivoli Netcool/OMNIBus Probe for Email
- IBM MQ Advanced
- softwareag/adabas-ce
- softwareag/aris-kibana
- Data Studio (IBM)
- Data Server Manager Developer-C Edition (IBM)
- Blackfire
- Semaphore CI
- FileCloud
- softwareag/apama-correlator
- IBM Db2 Warehouse
- softwareag/aris-accserver
- softwareag/webmethods-microservicesruntime
- softwareag/terracotta-server-oss
- IBM Queryplex
- Puppet Agent Ubuntu
- Percona Server
- FullArmor HAPI File Share Mount
- Anaconda

A.6 Summary

| A summary of skipped repositories in this study | |
|---|--|
| Repository type | Reason |
| All | Failed to download image |
| All | Change in API (both v1 and v2) |
| All | Manifest not found due to internal server error, failed login, busy server etc. |
| All | Not supported by Clair (incompatible platform) |
| Verified and Certified | Pull access denied, requires checkout (typically for commercial repositories) |
| Verified and Certified | Incompatible platform (e.g. Windows) |
| Verified and Certified | Not contain images or explicit pulling instructions (especially Microsoft repositories) |
| Verified and Certified | Duplicate repositories (especially Microsoft repositories) |
| Official | Not containing real images, e.g. scratch and rocket.chat |
| Official | Missing tags on Docker Hub |

Table A.1: A summary of repositories which are not included in this study

Appendix B

Scripts

The most straight-forward way to setup the environment is to install the software from scratch on your VM (make sure to fulfill the requirements described in [5.4](#)), and also make sure the script is executable and then copy-paste this below.

B.1 Installing the required tools for the VMs

```
1  #!/bin/bash
2
3  # HOW-TO: Make an executable file in terminal
4  # Run this by typing . ./scriptname.sh
5  (Yes, there are two dots and a space between them)
6
7  # Install Docker Engine
8  curl -sSL https://get.docker.com/ | sudo sh;
9
10 # Add user to Docker group
11 sudo usermod -a -G docker $USER;
12
13 # Install Git and Clair scanner binary file for Linux
14 sudo apt-get update;
15 sudo apt-get -y install git;
16 wget https://github.com/arminc/clair-scanner/releases/download/v8/clair-scanner_linux_amd64;
17 mv clair-scanner_linux_amd64 clair-scanner && chmod +x clair-scanner;
18
19 # Install and launch Clair and Clair DB containers
20 sudo docker run -u root -p 5432:5432 -d --name db arminc/clair-db:latest;
21 sudo docker run -u root -p 6060:6060 --link db:postgres -d
22 --name clair arminc/clair-local-scan:latest;
23
24 # Pip3 is required
25 sudo apt update && sudo apt install python3-pip -y;
```

B.2 Setup of the environment

Note that DAZER is an open source software publicly available via its own Github repository located at <https://github.com/dockalyzer/dazer>.

B.2.1 Requirements for *.nix

- Ubuntu 16.04.5 LTS*: <http://releases.ubuntu.com/16.04/> (required for Linux)
- Git: <https://git-scm.com/downloads> (required)
- Python 3.6.x: <https://www.python.org/downloads/> (required)
- Docker: <https://www.docker.com/get-started> (required)
- Clair scanner: <https://github.com/arminc/clair-scanner> (required)
- Valid Docker Hub credentials: <https://hub.docker.com/signup> (required)
- MongoDB: <https://resources.mongodb.com/getting-started-with-mongodb> (recommended)

Note: more recent Ubuntu versions and other Debian-based distributions should also work but they have not been tested.

B.2.2 Requirements for Windows

- Git: <https://git-scm.com/downloads> (required)
- Docker: <https://hub.docker.com/editions/community/docker-ce-desktop-windows> (required)
- Anaconda: <https://www.anaconda.com/distribution/> (strongly recommended)
- Clair scanner: <https://github.com/arminc/clair-scanner> (required)
- Valid Docker Hub credentials: <https://hub.docker.com/signup> (required)
- MongoDB: <https://resources.mongodb.com/getting-started-with-mongodb> (recommended)

B.2.3 Prerequisite

It is up to you whether you want to use the Clair binary (recommended) or install it from source on your local machine. We demonstrate this process for Linux only.

Clair binaries can be obtained here: <https://github.com/arminc/clair-scanner/releases>

1. Download the appropriate binary from the link above (e.g. for Ubuntu: `clair-scanner_linux_amd64`) using the following command:

```
wget https://github.com/arminc/clair-scanner/releases/download/v8/clair-scanner_
```

`linux_amd64`

2. Set write permission to the downloaded binary and move it to your home directory with the following name:

```
chmod +x clair-scanner_linux_amd64
mv clair-scanner_linux_amd64 $HOME/clair-scanner
```

3. Deploy the Clair database with the following command:

```
docker run -d --name db arminc/clair-db:latest
```

4. Deploy the Clair scanner with the following command:

```
docker run -p 6060:6060 --link db:postgres -d --name clair arminc/clair-local-scan:latest
```

Important: make sure Clair scanner and the Clair database are using the "latest" tag, otherwise DAZER will try to delete them.

B.2.4 Getting Started

1. Clone this repository

```
git clone https://github.com/dockalyzer/dazer.git
```

2. Add your Docker Hub credentials to the credentials.yml file.
3. Navigate to its root directory and install all the necessary Python packages using the following command:

```
pip install -e .
```

4. Run DAZER as followed:

```
./main.py <official|certified|verified|community> [<x_images>]
```

Examples:

Gathering metadata and vulnerability information for all Certified images:

```
./main.py certified
```

Gathering metadata and vulnerability information for all Verified images:

```
./main.py verified
```

Gathering metadata and vulnerability information for all Official images:

```
./main.py official
```

Gathering metadata and vulnerability information for 100 random Community images among the most popular ones:

```
./main.py community 100
```

5. Import the exported Json files to a noSQL database for further analysis (e.g. MonogDB):

```
mongoimport -db analyzed_images -collection images --file  
$HOME/DAZER/DAZER/json/vulnerabilities_2019-05-20_14-58-00.json
```

Appendix C

Research queries

Some of the research queries cannot be directly translated using a single-lined MongoDB query. Therefore, JavaScript code along with Mongo queries are utilized to fully develop the research queries.

C.1 MongoDB queries

RQ1: Have the security measures introduced by Docker Inc. in response to previous research improved Docker Hub’s security landscape and to what extent?

As described in 5.5.2, RQ1 consists of several detailed research questions defined in section 4.5 which only apply to Certified and Verified images. Those detailed research questions cannot be directly translated into research queries, as they essentially consist of comparing and identifying correlations between the results of RQ2 and RQ3 for Certified and Verified images with other types of images.

RQ2: Are vulnerabilities still inherited from images’ parent(s) and in what proportion?

RQ2.1: What proportion of images depends on a parent?

```
// Change to the desired type of repository
var image_type = "certified";

print("What proportion of " + image_type + " images depends on a parent? ")
+ db.data.find({$and: [ {"type": {"$eq": image_type }}, {"parent": {"$ne" : ""}} ]},
{ _id: 0, name: 1, parent: 1 }).count());
```

RQ2.2: What proportion of images contain inherited vulnerabilities?

```
// TIPS: Place this script below into a new file
// Change to the desired type of repository
```

```

var image_type = "certified";
var result = [];

var parent = function(image_name) {
  var image_sliced = image_name.split(":")[0];
  var tag_sliced = image_name.split(":")[1];
  var parent_name = db.data.findOne({$and: [ {name: {$eq: image_sliced }},
    {tag: {$eq: tag_sliced }}]}).parent;

  if (parent_name) {
    var child_vuln = db.data.findOne({$and: [{name: {$eq: image_sliced }},
      {tag: {$eq: tag_sliced }}]}, {name: 1, tag: 1, vulnerabilities: 1});
    var parent_vuln = db.data.findOne({$and: [ {name: {$eq: parent_name.split(":")[0] }},
      {tag: {$eq: parent_name.split(":")[1]}}]}, {name: 1, tag: 1, vulnerabilities: 1});

    if (parent_vuln === null)
    {
      return result;
    }

    var merged = child_vuln.vulnerabilities.filter(value => -1 !==
parent_vuln.vulnerabilities.indexOf(value));

    for (let index = 0; index < merged.length; index++) {
      let el = merged[index];
      if (result.indexOf(el) === -1) {
        result.push(el);
      }
    }
    return parent(parent_name);
  }
  return result;
};

var total_image = 0;
var contain_inherited_vuln = 0;
var dont_contain_vuln = 0;
var total_inherited_vuln = [];

db.data.find().forEach( function(image) {
  let image_and_name = image.name + ":" + image.tag;
  total_image++;
  a = parent(image_and_name)
  if (a.length > 0) {

```

```

        contain_inherited_vuln++;
        total_inherited_vuln.push(a);
        result = [];
    }
    else {
        dont_contain_vuln++;
    }
})

var avg_inherited_vuln = [];
total_inherited_vuln.forEach(function(element) {
    avg_inherited_vuln.push(element.length);
});

const average = arr => arr.reduce( ( p, c ) => p + c, 0 ) / arr.length;
const result_inherited_average = average(avg_inherited_vuln);

print("Total " + image_type + " images contain inherited vulnerabilities "
+ contain_inherited_vuln);
print("Total " + image_type + " images that dont contain inherited vulnerabilities: "
+ dont_contain_vuln);
print("How many vulnerabilities do " + image_type + " images inherit in average: "
+ Math.ceil(result_inherited_average) + " (Rounded up, e.g. 1.1 -> 2)");

```

RQ2.3: What proportion of images introduce vulnerabilities?

```

// Change to the desired type of repository
var image_type = "certified";

// TIPS: Place this script below into a new file
var result = [];

var parent = function(image_name) {
    var image_sliced = image_name.split(":")[0];
    var tag_sliced = image_name.split(":")[1];
    var parent_name = db.data.findOne({$and: [ {name: {$eq: image_sliced }},
    {tag: {$eq: tag_sliced }}]}).parent;
    var child_vulnerabilities = db.data.findOne({$and: [ {name: {$eq: image_sliced }},
    {tag: {$eq: tag_sliced }}]}).vulnerabilities;

    if (parent_name) {
        var child_vuln = db.data.findOne({$and: [ {name: {$eq: image_sliced }},
        {tag: {$eq: tag_sliced }}]}, {name: 1, tag: 1, vulnerabilities: 1});
    }
}

```

```

var parent_vuln = db.data.findOne({$and: [{name: {$eq: parent_name.split(":")[0]}},
{tag: {$eq: parent_name.split(":")[1]}}]}, {name: 1, tag: 1, vulnerabilities: 1});

if (parent_vuln === null)
{
    return result;
}

var merged = child_vuln.vulnerabilities.filter(value => -1 !==
parent_vuln.vulnerabilities.indexOf(value));

for (let index = 0; index < child_vulnerabilities.length; index++) {
    let el = child_vulnerabilities[index];
    let not_contain = merged.includes(el);
    if (!(not_contain)) {
        result.push(el);
    }
}
return parent(parent_name);
}
return result;
};

var total_image = 0;
var contain_inherited_vuln = 0;
var dont_contain_vuln = 0;
var total_inherited_vuln = [];

db.data.find().forEach( function(image) {
    let image_and_name = image.name + ":" + image.tag;
    total_image++;
    a = parent(image_and_name)
    if (a.length > 0) {
        contain_inherited_vuln++;
        total_inherited_vuln.push(a);
        result = [];
    }
    else {
        dont_contain_vuln++;
    }
})

var avg_inherited_vuln = [];
total_inherited_vuln.forEach(function(element) {
    avg_inherited_vuln.push(element.length);

```



```
});
```

```
const average = arr => arr.reduce( ( p, c ) => p + c, 0 ) / arr.length;
const result_inherited_average = average(avg_inherited_vuln);

print("Total " + image_type + " images that introduce vulnerabilities: "
+ contain_inherited_vuln);
print("Total " + image_type + " images that dont introduce vulnerabilities: "
+ dont_contain_vuln);
print("How many vulnerabilities do " + image_type + " images introduced in average: "
+ Math.ceil(result_inherited_average) + " (Rounded up, e.g. 1.1 -> 2)");
```

RQ2.4: How many vulnerabilities do images inherit in average?

Already answered in RQ2.3.

RQ2.5: How many vulnerabilities do images introduce in average?

Already answered in RQ2.4.

Q3: How are discovered vulnerabilities distributed across repository types?

RQ3.1: What proportion of images contain no vulnerabilities?

```
// Change to the desired type of repository
var image_type = "certified";

print("Proportion of " + image_type + " images that contain no vulnerabilities: "
+ db.data.find({$and: [ {"type": {"$eq": image_type }}, {"total_vulnerabilities":
{"$eq" : 0}} ]}, { _id: 0, name: 1, vulnerabilities: 1 }).count());
```

RQ3.2: What proportion of images contain at least one vulnerability?

```
// Change to the desired type of repository
var image_type = "certified";

print("Proportion of images that contain one or several vulnerabilities: " +
db.data.find({$and: [{"type": {"$eq": image_type }}, {"total_vulnerabilities":
{"$ne" : 0}}]}}, { _id: 0, name: 1, vulnerabilities: 1 }).count());
```

Note: Unlike find() function in MongoDB, aggregation() function does not support retrieving the value of a property, and therefore it must be calculated manually by dividing the number of vulnerabilities with total analyzed images.

RQ3.3: How many vulnerabilities do images contain in average?

```

// Change to the desired type of repository
var image_type = "certified";

print("How many vulnerabilities do " + image_type + " images contain in total
(remember to calculate average): ");
db.data.aggregate({ $group: { _id: "$type", sumTotalVulnerabilities:
{ $sum: "$total_vulnerabilities" } } },
{$project: {"_id": 0, "sumTotalVulnerabilities": 1}} );
// Average to be calculated manually

```

RQ3.4: How many unique vulnerabilities do images contain in total?

```

// Change to the desired type of repository
var image_type = "certified";

print("How many unique vulnerabilities " + image_type + " images contain in total
(remember to calculate average): ");
db.vuln.aggregate([ { $group : { _id: { cve_number : {$gt:["$cve_number", null]}},
count : { $sum : 1 } } } ]); // Average to be calculated manually

```

RQ3.5: How are unique vulnerabilities distributed per severity among images?

```

// Change to the desired type of repository
var image_type = "certified";

print("How are unique vulnerabilities distributed per severity among "
+ image_type + " images")
print("Critical: " + db.vuln.find({"severity": {"$eq" : "Critical"}}),
{ _id: 0, severity: 1 }).count())

print("High: " + db.vuln.find({"severity": {"$eq" : "High"}}),
{ _id: 0, severity: 1 }).count())

print("Medium: " + db.vuln.find({"severity": {"$eq" : "Medium"}}),
{ _id: 0, severity: 1 }).count())

print("Low: " + db.vuln.find({"severity": {"$eq" : "Low"}}),
{ _id: 0, severity: 1 }).count())

print("Negligible: " + db.vuln.find({"severity": {"$eq" : "Negligible"}}),
{ _id: 0, severity: 1 }).count())

print("Unknown: " + db.vuln.find({"severity": {"$eq" : "Unknown"}}),
{ _id: 0, severity: 1 }).count())

```

RQ3.6: How are unique vulnerabilities distributed per year among images?

```
// Change to the desired type of repository
var image_type = "certified";

print("How are unique vulnerabilities distributed per year among " + image_type + " images")
print("2019: " + db.vuln.find( { cve_number: { $regex: '(CVE-2019|RHSA-2019|ELSA-2019)' } }
).count())
print("2018: " + db.vuln.find( { cve_number: { $regex: '(CVE-2018|RHSA-2018|ELSA-2018)' } }
).count())
print("2017: " + db.vuln.find( { cve_number: { $regex: '(CVE-2017|RHSA-2017|ELSA-2017)' } }
).count())
print("2016: " + db.vuln.find( { cve_number: { $regex: '(CVE-2016|RHSA-2016|ELSA-2016)' } }
).count())
print("2015: " + db.vuln.find( { cve_number: { $regex: '(CVE-2015|RHSA-2015|ELSA-2015)' } }
).count())
print("2014: " + db.vuln.find( { cve_number: { $regex: '(CVE-2014|RHSA-2014|ELSA-2014)' } }
).count())
print("2013: " + db.vuln.find( { cve_number: { $regex: '(CVE-2013|RHSA-2013|ELSA-2013)' } }
).count())
print("2012: " + db.vuln.find( { cve_number: { $regex: '(CVE-2012|RHSA-2012|ELSA-2012)' } }
).count())
print("2011: " + db.vuln.find( { cve_number: { $regex: '(CVE-2011|RHSA-2011|ELSA-2011)' } }
).count())
print("2010: " + db.vuln.find( { cve_number: { $regex: '(CVE-2010|RHSA-2010|ELSA-2010)' } }
).count())
```

RQ3.7: Is there a correlation between the most popular images (most pulled) and the most vulnerable ones?

```
// Change to the desired type of repository
var image_type = "certified";

print("Most pulled top 10: ");
db.data.find({$and: [ {"type": {"$eq": image_type }}, {"name": {"$ne" : ""}}]},
{ _id: 0, name: 1, total_pulled: 1, total_vulnerabilities: 1}).sort({total_pulled:-1}
).limit(10);

print("Most vulnerable top 10: ");
db.data.find({$and: [ {"type": {"$eq": image_type }}, {"name": {"$ne" : ""}}]},
{ _id: 0, name: 1, total_vulnerabilities: 1 }).sort({total_vulnerabilities:-1}
).limit(10);
```

RQ3.8: Is there a correlation between the last updated images and the most vulnerable ones?

```

// Change to the desired type of repository
var image_type = "certified";

print("Most recently updated top 10: ");
db.data.find({$and: [ {"type": {"$eq": image_type }}, {"name": {"$ne" : ""}}]},
{ _id: 0, name: 1, last_updated: 1, total_vulnerabilities: 1}).sort({last_updated:-1}
).limit(10);

print("Most vulnerable top 10: " );
db.data.find({$and: [ {"type": {"$eq": image_type }}, {"name": {"$ne" : ""}}]},
{ _id: 0, name: 1, total_vulnerabilities: 1 }).sort({total_vulnerabilities:-1}
).limit(10);

```

RQ3.9: Is there a correlation between base images and the most vulnerable ones OR base images and the most popular ones OR base images and the last updated ones?

```

// Change to the desired type of repository
var image_type = "certified";

print("Most vulnerable top 10 (base images): " );
db.data.find({$and: [ {"type": {"$eq": image_type }}, {"parent": {"$eq" : ""}}]},
{ _id: 0, name: 1, total_vulnerabilities: 1 }).sort({total_vulnerabilities:-1}
).limit(10);

print("Most pulled top 10 (base images): ");
db.data.find({$and: [ {"type": {"$eq": image_type }}, {"parent": {"$eq" : ""}}]},
{ _id: 0, name: 1, total_pulled: 1, total_vulnerabilities: 1}).sort({total_pulled:-1}
).limit(10);

print("Most recently updated top 10 (base images): ");
db.data.find({$and: [ {"type": {"$eq": image_type }}, {"parent": {"$eq" : ""}}]},
{ _id: 0, name: 1, last_updated: 1, total_vulnerabilities: 1}).sort({last_updated:-1}
).limit(10);

```

RQ3.10: What are the top 10 vulnerabilities?

```

// Change to the desired type of repository
var image_type = "certified";

print("Top 10 vulnerabilities for " + image_type + " images: ");
db.data.aggregate([ { $match: { type: image_type } }, {$unwind: "$vulnerabilities" },
{$group: {_id: {id: "$vulnerabilities"}, count: {$sum : 1}}}, { $sort: { count: -1 }},
{ $limit: 10 } ] );

```

RQ3.11: Are vulnerabilities found in base images correlated with non-base images in some way?

```
// Change to the desired type of repository
var image_type = "certified";

print("Top 10 vulnerabilities for base images: ");
db.data.aggregate([ { $match: { type: image_type, "$and": [ { "parent": { "$eq": "" } } ] } }, { $unwind: "$vulnerabilities" }, { $group: { _id: { id: "$vulnerabilities" }, count: { $sum : 1 } } }, { $sort: { count: -1 } }, { $limit: 10 } ] );

print("Top 10 vulnerabilities for not-base images: ");
db.data.aggregate([ { $match: { type: image_type, "$and": [ { "parent": { "$ne": "" } } ] } }, { $unwind: "$vulnerabilities" }, { $group: { _id: { id: "$vulnerabilities" }, count: { $sum : 1 } } }, { $sort: { count: -1 } }, { $limit: 10 } ] );
```

C.2 Miscellaneous MongoDB queries

In this study, we would also like to survey not only the vulnerabilities across repository types, but also all the vulnerabilities on Docker Hub in general. When merging multiple JSON vulnerability files in MongoDB, there are many JSON objects that occur several times. These needed to be removed, and therefore the script below may be executed via MongoDB shell to remove the duplicate MongoDB documents. Note that we experienced that this script needs to be run several times before all the duplicates are removed.

```
// Get unique vulnerability objects
var bulk = db.vuln.initializeOrderedBulkOp(), count = 0;

// Specify the most unique property in the entire collection, in this case the CVE number.
db.vuln.aggregate([
  { "$group": {
    "_id": {
      "cve_number" : "$cve_number",
    },
    "ids": { "$push": "$_id" },
    "count": { "$sum": 1 }
  } },
  { "$match": { "count": { "$gt": 1 } } }
],
{ "allowDiskUse": true }).forEach(function(doc) {
  doc.ids.shift(); // removes the first match
  bulk.find({ "_id": { "$in": doc.ids } }).remove();
  count++;
}
```

```

    if ( count % 1000 == 0 ) {
        bulk.execute();
        bulk = db.testkdd.initializeOrderedBulkOp();
    }
});

if ( count % 1000 != 0 )
    bulk.execute();

```

Note: Similar for the "data" collection, we observed hundreds of duplicate images in the collection for Community images. These need to be removed using this script below:

```

// Get unique image objects
var bulk = db.data.initializeOrderedBulkOp(), count = 0;

// Specify the most unique property in the entire collection, in this case the image ID.
db.data.aggregate([
    { "$group": {
        "_id": {
            "image_id" : "$image_id",
        },
        "ids": { "$push": "$_id" },
        "count": { "$sum": 1 }
    }},
    { "$match": { "count": { "$gt": 1 } } }
],
{ "allowDiskUse": true}).forEach(function(doc) {
    doc.ids.shift(); // removes the first match
    bulk.find({ "_id": { "$in": doc.ids } }).remove();
    count++;

    if ( count % 1000 == 0 ) {
        bulk.execute();
        bulk = db.testkdd.initializeOrderedBulkOp();
    }
});

if ( count % 1000 != 0 ) {
    bulk.execute();
}

```

Appendix D

Result data

D.1 Top ten most vulnerable repositories across image types

Official repositories

- (K_1) rails (1530 vulnerabilities)
- (K_2) django (1474 vulnerabilities)
- (K_3) glassfish (645 vulnerabilities)
- (K_4) erlang (589 vulnerabilities)
- (K_5) elixir (588 vulnerabilities)
- (K_6) python (586 vulnerabilities)
- (K_7) hylang (586 vulnerabilities)
- (K_8) gcc (586 vulnerabilities)
- (K_9) perl (586 vulnerabilities)
- (K_{10}) buildpack-deps (586 vulnerabilities)

Community repositories

- (A_1) wfmdigital/php-worker (1792 vulnerabilities)
- (A_2) iron/images (1792 vulnerabilities)
- (A_3) nolan/concourse-rancher-compose-resource (1741 vulnerabilities)
- (A_4) robophred/concourse-svn-resource (1741 vulnerabilities)
- (A_5) jinlee/counter-resource (1518 vulnerabilities)
- (A_6) inquicker/kaws-etcd-ebs-backup (1350 vulnerabilities)

- (A_7) modeanalytics/concourse-mapping-resource (1327 vulnerabilities)
- (A_8) onswb/php-base (1300 vulnerabilities)
- (A_9) therightplace/bedboard2-sidekiq (1277 vulnerabilities)
- (A_{10}) springcloud/spring-pipeline-m2 (1218 vulnerabilities)

Verified repositories

- (D_1) mcr.microsoft.com/azuredocs/azure-vote-front (1531 vulnerabilities)
- (D_2) store/klokantech/openmaptiles-server-dev (1171 vulnerabilities)
- (D_3) mcr.microsoft.com/ospo/ghcrawler-dashboard (773 vulnerabilities)
- (D_4) store/filecloud/filecloud (759 vulnerabilities)
- (D_5) store/discourse/discourse (654 vulnerabilities)
- (D_6) mcr.microsoft.com/oryx/build (643 vulnerabilities)
- (D_7) mcr.microsoft.com/ospo/ghcrawler (639 vulnerabilities)
- (D_8) store/ibmcorp/icam-service-composer-ui (598 vulnerabilities)
- (D_9) store/rocketchat/rocket.chat (556 vulnerabilities)
- (D_{10}) mcr.microsoft.com/aiforearth/base-r (434 vulnerabilities)

Certified repositories

- (G_1) store/opsani/skopos (166 vulnerabilities)
- (G_2) store/oracle/database-enterprise (106 vulnerabilities)
- (G_3) store/hpsoftware/sitescope_store (79 vulnerabilities)
- (G_4) store/bleemeo/bleemeo-agent (78 vulnerabilities)
- (G_5) store/datadog/agent (55 vulnerabilities)
- (G_6) store/sematext/agent (46 vulnerabilities)
- (G_7) store/ibmcorp/websphere-liberty (44 vulnerabilities)
- (G_8) store/ibmcorp/db2_developer_c (43 vulnerabilities)
- (G_9) store/oracle/database-instantclient (42 vulnerabilities)
- (G_{10}) store/oracle/coherence (40 vulnerabilities)

D.2 Top ten most pulled repositories across image types

Official repositories

- (L_1) nginx ($1.650 * 10^9$ pulls, 82 vulnerabilities)
- (L_2) redis ($1.496 * 10^9$ pulls, 43 vulnerabilities)
- (L_3) alpine ($1.196 * 10^9$ pulls, 0 vulnerabilities)
- (L_4) httpd ($1.116 * 10^9$ pulls, 135 vulnerabilities)
- (L_5) ubuntu ($1.018 * 10^9$ pulls, 41 vulnerabilities)
- (L_6) postgres ($9.293 * 10^8$ pulls, 102 vulnerabilities)
- (L_7) node ($8.075 * 10^8$ pulls, 79 vulnerabilities)
- (L_8) mysql ($7.913 * 10^8$ pulls, 59 vulnerabilities)
- (L_9) memcached ($7.066 * 10^8$ pulls, 43 vulnerabilities)
- (L_{10}) registry ($6.135 * 10^8$ pulls, 0 vulnerabilities)

Community repositories

- (B_1) jtarchie/pr ($2.060 * 10^9$ pulls, 16 vulnerabilities)
- (B_2) pivotalcf/pivnet-resource ($1.951 * 10^9$ pulls, 333 vulnerabilities)
- (B_3) cfcommunity/slack-notification-resource ($1.749 * 10^9$ pulls, 0 vulnerabilities)
- (B_4) kope/protokube ($5.559 * 10^8$ pulls, 133 vulnerabilities)
- (B_5) pivotalpa/maven-resource ($2.718 * 10^8$ pulls, 40 vulnerabilities)
- (B_6) navicore/teams-notification-resource ($2.403 * 10^8$ pulls, 178 vulnerabilities)
- (B_7) swce/keyval-resource ($1.898 * 10^8$ pulls, 0 vulnerabilities)
- (B_8) bitnami/mongodb ($1.894 * 10^8$ pulls, 74 vulnerabilities)
- (B_9) datadog/agent ($1.799 * 10^8$ pulls, 47 vulnerabilities)
- (B_{10}) patrickcrocker/maven-resource ($1.606 * 10^8$ pulls, 40 vulnerabilities)

Verified repositories

- (E_1) mcr.microsoft.com/dotnet/core/runtime-deps ($3.489 * 10^8$ pulls, 54 vulnerabilities)
- (E_2) mcr.microsoft.com/service-fabric/reverse-proxy ($2.473 * 10^8$ pulls, 104 vulnerabilities)
- (E_3) mcr.microsoft.com/dotnet/core-nightly/runtime-deps ($3.029 * 10^7$ pulls, 54 vulnerabilities)
- (E_4) mcr.microsoft.com/mssql-tools ($4.910 * 10^6$ pulls, 132 vulnerabilities)
- (E_5) mcr.microsoft.com/azureiotedge-agent ($3.617 * 10^6$ pulls, 0 vulnerabilities)

- (E_6) mcr.microsoft.com/azuredocs/aci-helloworld ($2.707 * 10^6$ pulls, 3 vulnerabilities)
- (E_7) mcr.microsoft.com/cntk/release ($2.293 * 10^6$ pulls, 211 vulnerabilities)
- (E_8) mcr.microsoft.com/azureiotedge-hub ($1.436 * 10^6$ pulls, 0 vulnerabilities)
- (E_9) mcr.microsoft.com/azureiotedge-simulated-temperature-sensor ($1.103 * 10^6$ pulls, 0 vulnerabilities)
- (E_{10}) mcr.microsoft.com/azuredocs/aci-tutorial-sidecar ($9.472 * 10^5$ pulls, 22 vulnerabilities)

Certified repositories

- (H_1) mcr.microsoft.com/java/jre-headless (1947 pulls, 0 vulnerabilities)
- (H_2) store/sematext/logagent (0 pulls, 0 vulnerabilities)
- (H_3) store/veritasnetbackup/client (0 pulls, 22 vulnerabilities)
- (H_4) store/sematext/agent (0 pulls, 46 vulnerabilities)
- (H_5) store/ibmcorp/isam (0 pulls, 34 vulnerabilities)
- (H_6) store/elastic/packetbeat (0 pulls, 2 vulnerabilities)
- (H_7) store/ibmcorp/db2_developer_c (0 pulls, 43 vulnerabilities)
- (H_8) store/oracle/fmw-infrastructure (0 pulls, 29 vulnerabilities)
- (H_9) store/dynatrace/oneagent (0 pulls, 1 vulnerability)
- (H_{10}) store/elastic/heartbeat (0 pulls, 2 vulnerabilities)

D.3 Top ten last updated repositories across image types

Official repositories

- (N_1) swift: Saturday, 30-Mar-19 00:24:43 UTC (139 vulnerabilities)
- (N_2) xwiki: Friday, 29-Mar-19 23:13:55 UTC in RFC 2822 (189 vulnerabilities)
- (N_3) geonetwork: Friday, 29-Mar-19 23:10:01 UTC (186 vulnerabilities)
- (N_4) websphere-liberty: Friday, 29-Mar-19 23:03:03 UTC (43 vulnerabilities)
- (N_5) maven: Friday, 29-Mar-19 22:51:23 UTC (0 vulnerabilities)
- (N_6) clojure: Friday, 29-Mar-19 22:41:29 UTC (201 vulnerabilities)
- (N_7) tomcat: Friday, 29-Mar-19 22:29:12 UTC (0 vulnerabilities)
- (N_8) pypy: Friday, 29-Mar-19 22:21:07 UTC (119 vulnerabilities)
- (N_9) teamspeak: Friday, 29-Mar-19 22:17:25 UTC (0 vulnerabilities)
- (N_{10}) rabbitmq: Friday, 29-Mar-19 22:15:26 UTC (32 vulnerabilities)

Community repositories

- (C_1) circleci/php: Sunday, 21-Apr-19 00:45:05 UTC (451 vulnerabilities)
- (C_2) circleci/golang: Sunday, 21-Apr-19 00:32:48 UTC (439 vulnerabilities)
- (C_3) drone/drone: Sunday, 21-Apr-19 00:27:34 UTC (0 vulnerabilities)
- (C_4) drone/agent: Sunday, 21-Apr-19 00:27:03 UTC (0 vulnerabilities)
- (C_5) circleci/mysql: Sunday, 21-Apr-19 00:05:47 UTC (68 vulnerabilities)
- (C_6) bitnami/minideb: Saturday, 20-Apr-19 22:33:36 UTC (44 vulnerabilities)
- (C_7) graze/php-alpine: Saturday, 20-Apr-19 20:30:38 UTC (0 vulnerabilities)
- (C_8) titpetric/netdata: Saturday, 20-Apr-19 19:34:29 UTC (128 vulnerabilities)
- (C_9) elicocorp/odoo-china: Saturday, 20-Apr-19 19:21:54 UTC (173 vulnerabilities)
- (C_{10}) kpsys/portaro: Saturday, 20-Apr-19 18:16:50 UTC (90 vulnerabilities)

Verified repositories

- (F_1) mcr.microsoft.com/oryx/python-3.7: Saturday, 30-Mar-19 19:12:28 UTC (180 vulnerabilities)
- (F_2) mcr.microsoft.com/oryx/build: Saturday, 30-Mar-19 19:11:50 UTC (643 vulnerabilities)
- (F_3) mcr.microsoft.com/dotnet/core-nightly/runtime-deps: Friday, 29-Mar-19 21:41:55 UTC (54 vulnerabilities)
- (F_4) store/ibmcorp/security_information_queue: Thursday, 28-Mar-19 21:58:58 UTC (161 vulnerabilities)
- (F_5) mcr.microsoft.com/dotnet/core/runtime-deps: Wednesday, 27-Mar-19 20:48:07 UTC (54 vulnerabilities)
- (F_6) mcr.microsoft.com/mssql/server: Thursday, 21-Mar-19 02:56:20 UTC (39 vulnerabilities)
- (F_7) store/softwareag/apama-correlator: Tuesday, 19-Mar-19 18:08:53 UTC (39 vulnerabilities)
- (F_8) mcr.microsoft.com/azure-functions/base: Tuesday, 19-Mar-19 17:56:07 UTC (115 vulnerabilities)
- (F_9) mcr.microsoft.com/iotedgedge/opc-client: Friday, 15-Mar-19 14:44:48 UTC (0 vulnerabilities)
- (F_{10}) mcr.microsoft.com/iotedgedge/opc-publisher-nodeconfiguration: Friday, 15-Mar-19 13:05:04 UTC (0 vulnerabilities)

Certified repositories

- (I_1) store/elastic/packetbeat: Tuesday, 02-Apr-19 16:32:37 UTC (2 vulnerabilities)

- (I_2) store/elastic/auditbeat: Tuesday, 02-Apr-19 15:21:37 UTC (2 vulnerabilities)
- (I_3) store/elastic/metricbeat: Tuesday, 02-Apr-19 15:13:29 UTC (2 vulnerabilities)
- (I_4) store/elastic/heartbeat: Tuesday, 02-Apr-19 15:06:11 UTC (2 vulnerabilities)
- (I_5) store/elastic/filebeat: Tuesday, 02-Apr-19 15:03:06 UTC (2 vulnerabilities)
- (I_6) store/elastic/apm-server: Tuesday, 02-Apr-19 14:56:32 UTC (2 vulnerabilities)
- (I_7) store/ibmcorp/websphere-liberty: Wednesday, 27-Mar-19 02:30:44 UTC (44 vulnerabilities)
- (I_8) store/sematext/agent: Wednesday, 20-Mar-19 14:23:43 UTC (46 vulnerabilities)
- (I_9) store/dynatrace/oneagent: Wednesday, 20-Mar-19 08:00:59 UTC (1 vulnerability)
- (I_{10}) store/intersystems/iris: Friday, 15-Mar-19 14:36:43 UTC (37 vulnerabilities)

D.4 All base repositories across image types - sorted by popularity

Official repositories

1. alpine
2. ubuntu
3. centos
4. debian
5. amazonlinux
6. iojs
7. oraclelinux
8. notary
9. glassfish
10. sl
11. java

Community repositories (only top ten)

1. jtarchie/pr
2. cfcommunity/slack-notification-resource
3. kope/protokube
4. pivotalpa/maven-resource
5. swce/keyval-resource

6. [bitnami/mongodb](#)
7. [patrickcrocker/maven-resource](#)
8. [ymedlop/npm-cache-resource](#)
9. [mesosphere/aws-cli](#)
10. [jrsc/letsencrypt-nginx-proxy-companion](#)

Verified repositories

1. [mcr.microsoft.com/mssql-tools](#)
2. [mcr.microsoft.com/azureiotedge-agent](#)
3. [mcr.microsoft.com/azuredocs/aci-helloworld](#)
4. [mcr.microsoft.com/azureiotedge-hub](#)
5. [mcr.microsoft.com/azureiotedge-simulated-temperature-sensor](#)
6. [mcr.microsoft.com/azuredocs/aci-tutorial-sidecar](#)
7. [mcr.microsoft.com/mssql/server](#)
8. [mcr.microsoft.com/azureiotedge-functions-binding](#)
9. [mcr.microsoft.com/azureiotedge/modbus](#)
10. [mcr.microsoft.com/iotedge/opc-publisher](#)
11. [mcr.microsoft.com/azuredocs/aci-wordcount](#)
12. [mcr.microsoft.com/iotedge/opc-proxy](#)
13. [mcr.microsoft.com/azure-functions/base](#)
14. [mcr.microsoft.com/k8s/aad-pod-identity/mic](#)
15. [mcr.microsoft.com/azuredocs/aci-hellofiles](#)
16. [mcr.microsoft.com/azure-cognitive-services/luis](#)
17. [mcr.microsoft.com/azureiotedge-testing-utility](#)
18. [mcr.microsoft.com/azure-oss-db-tools/pgbouncer-sidecar](#)
19. [mcr.microsoft.com/iotedge/opc-plc](#)
20. [mcr.microsoft.com/iotedge/opc-publisher-diagnostics](#)

Certified repositories

1. [mcr.microsoft.com/java/jre-headless](#)
2. [store/veritasnetbackup/client](#)

3. store/ibmcorp/isam
4. store/ibmcorp/db2_developer_c
5. store/coscale/coscale-agent
6. store/instana/agent
7. store/oracle/fmw-infrastructure
8. store/oracle/mysql-enterprise-server
9. store/oracle/weblogic
10. store/oracle/serverjre
11. store/datadog/agent
12. store/amirsharif/enforcerd
13. store/weaveworks/cloud-agent
14. store/nvpublic/allinone
15. store/oracle/coherence
16. store/oracle/database-enterprise
17. store/opsani/skopos
18. store/oracle/database-instantclient
19. store/hpsoftware/sitescope_store

D.5 Top ten most vulnerable base repositories across image types

Official repositories

1. glassfish (645 vulnerabilities)
2. iojs (197 vulnerabilities)
3. ubuntu (41 vulnerabilities)
4. debian (28 vulnerabilities)
5. centos (2 vulnerabilities)
6. alpine (0 vulnerabilities)
7. java (0 vulnerabilities)
8. oraclelinux (0 vulnerabilities)
9. sl (0 vulnerabilities)
10. amazonlinux (0 vulnerabilities)

Community repositories

1. wfmdigital/php-worker (1792 vulnerabilities)
2. iron/images (1792 vulnerabilities)
3. nolan/concourse-rancher-compose-resource (1741 vulnerabilities)
4. jinlee/counter-resource (1518 vulnerabilities)
5. inquicker/kaws-etcd-ebs-backup (1350 vulnerabilities)
6. modeanalytics/concourse-mapping-resource (1327 vulnerabilities)
7. onswab/php-base (1300 vulnerabilities)
8. therightplace/bedboard2-sidekiq (1277 vulnerabilities)
9. pbardzinskismarsh/task-image (1121 vulnerabilities)
10. kpacha/mesos-influxdb-collector (951 vulnerabilities)

Verified repositories

1. mcr.microsoft.com/azuredocs/azure-vote-front (1531 vulnerabilities)
2. store/klokantech/openmaptiles-server-dev (1172 vulnerabilities)
3. store/discourse/discourse (654 vulnerabilities)
4. store/ibmcorp/icam-service-composer-ui (598 vulnerabilities)

5. store/rocketchat/rocket.chat (556 vulnerabilities)
6. store/sysdig/sysdig (423 vulnerabilities)
7. mcr.microsoft.com/k8s/aad-pod-identity/mic (309 vulnerabilities)
8. mcr.microsoft.com/windowscamerteam/onnxconverter (243 vulnerabilities)
9. store/influxdata/influxdb (197 vulnerabilities)
10. store/ibmcorp/voice-gateway-mr (190 vulnerabilities)

Certified repositories

1. store/opsani/skopos (166 vulnerabilities)
2. store/oracle/database-enterprise (106 vulnerabilities)
3. store/hpsoftware/sitescope_store (79 vulnerabilities)
4. store/datadog/agent (55 vulnerabilities)
5. store/ibmcorp/db2_developer_c (43 vulnerabilities)
6. store/oracle/database-instantclient (42 vulnerabilities)
7. store/oracle/coherence (40 vulnerabilities)
8. store/ibmcorp/isam (34 vulnerabilities)
9. store/amirsharif/enforcerd (30 vulnerabilities)
10. store/oracle/fmw-infrastructure (29 vulnerabilities)

D.6 Top ten most used parent images across image types

Official

1. debian:9-slim (18 child images)
2. openjdk:8-jre-alpine (7 child images)
3. centos:7 (6 child images)
4. debian:latest (5 child images)
5. alpine:latest (5 child images)
6. ubuntu:xenial (5 child images)
7. ubuntu:latest (4 child images)
8. buildpack-deps:stretch (4 child images)
9. php:7.3.3-fpm-stretch (4 child images)
10. alpine:3.8 (4 child images)

Community

1. alpine:latest (15 child images)
2. centos:7 (14 child images)
3. alpine:3.8 (11 child images)
4. ubuntu:xenial (10 child images)
5. ubuntu:latest (10 child images)
6. alpine:3.9.2 (7 child images)
7. ubuntu:bionic-20181204 (7 child images)
8. ubuntu:trusty-20181217 (5 child images)
9. ubuntu:bionic-20190204 (4 child images)
10. debian:stretch-20181226-slim (4 child images)

Verified

1. ubuntu:xenial-20180808 (3 child images)
2. debian:stretch-20181226-slim (3 child images)
3. ubuntu:bionic-20190204 (3 child images)
4. ubuntu:xenial-20180525 (3 child images)
5. debian:9-slim (2 child images)
6. alpine:3.7 (2 child images)
7. ubuntu:xenial-20180417 (1 child image)
8. ubuntu:xenial-20181005 (1 child image)
9. debian:jessie-20190204 (1 child image)
10. node:8.12 (1 child image)

Certified

1. centos:7 (6 child images)
2. ubuntu:xenial (1 child image)
3. ubuntu:bionic-20190204 (1 child image)
4. ubuntu:bionic-20180710 (1 child image)
5. alpine:latest (1 child image)

6. node:8.11-alpine (1 child image)
7. debian:stable-20190228-slim (1 child image)

D.6.1 Top ten most used parent images across all repository types

1. centos:7 (26 child images, 4 vulnerabilities)
2. debian:9-slim (26 child images, 47 vulnerabilities)
3. alpine:3.8 (25 child images, 0 vulnerabilities)
4. alpine:latest (23 child images, 0 vulnerabilities)
5. ubuntu:xenial (18 child images, 43 vulnerabilities)
6. ubuntu:latest (14 child images, 33 vulnerabilities)
7. java:openjdk-8-jre (11 child images, 271 vulnerabilities)
8. debian:latest (10 child images, 60 vulnerabilities)
9. ubuntu:bionic-20190204 (8 child images, 34 vulnerabilities)
10. ubuntu:bionic-20181204 (7 child images, 39 vulnerabilities)

D.7 Top ten most vulnerable packages

D.7.1 Across all repository types

1. firefox
2. linux
3. imagemagick
4. binutils
5. php5
6. tcpdump
7. mysql-5.5
8. tiff
9. openjdk-7
10. openssl

D.7.2 Across the most popular parents

1. glibc
2. ncurses
3. systemd
4. gnutls28
5. curl
6. pcre3
7. openssl
8. krb5
9. nss
10. sqlite3

D.8 CWE vulnerability categories

The following is a list of the CWE-IDs categorizing the vulnerabilities found in the ten most vulnerable packages discovered across all types of repositories during the conducted experiments, with an associated category name and definition retrieved from NVD's website [50].

- **CWE-19: Data Handling**
Weaknesses in this category are typically found in functionality that processes data.
- **CWE-20: Input Validation**
The product does not validate or incorrectly validates input that can affect the control flow or data flow of a program.
- **CWE-93: Improper Neutralization of CRLF Sequences ('CRLF Injection')**
The software uses CRLF (carriage return line feeds) as a special element, e.g. to separate lines or records, but it does not neutralize or incorrectly neutralizes CRLF sequences from inputs.
- **CWE-113: Improper Neutralization of CRLF Sequences in HTTP Headers ('HTTP Response Splitting')**
The software receives data from an upstream component, but does not neutralize or incorrectly neutralizes CR and LF characters before the data is included in outgoing HTTP headers.
- **CWE-119: Buffer Errors**
The software performs operations on a memory buffer, but it can read from or write to a memory location that is outside of the intended boundary of the buffer.
- **CWE-121: Stack-based Buffer Overflow**
A stack-based buffer overflow condition is a condition where the buffer being overwritten is allocated on the stack (i.e., is a local variable or, rarely, a parameter to a function).
- **CWE-122: Heap-based Buffer Overflow**
A heap overflow condition is a buffer overflow, where the buffer that can be overwritten is allocated in the heap portion of memory, generally meaning that the buffer was allocated using a routine such as malloc().
- **CWE-125: Out-of-bounds Read**
The software reads data past the end, or before the beginning, of the intended buffer.
- **CWE-190: Integer Overflow or Wraparound**
The software performs a calculation that can produce an integer overflow or wraparound, when the logic assumes that the resulting value will always be larger than the original value. This can introduce other weaknesses when the calculation is used for resource management or execution control.
- **CWE-200: Information Leak / Disclosure**
An information exposure is the intentional or unintentional disclosure of information to an actor that is not explicitly authorized to have access to that information.
- **CWE-254: Security Features**
Software security is not security software. Here we're concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management.

- **CWE-264: Permissions, Privileges, and Access Control**
Weaknesses in this category are related to the management of permissions, privileges, and other security features that are used to perform access control.
- **CWE-270: Privilege Context Switching Error**
The software does not properly manage privileges while it is switching between different contexts that have different privileges or spheres of control.
- **CWE-271: Privilege Dropping / Lowering Errors**
The software does not drop privileges before passing control of a resource to an actor that does not have those privileges.
- **CWE-338: Use of Cryptographically Weak Pseudo-Random Number Generator (PRNG)**
The product uses a Pseudo-Random Number Generator (PRNG) in a security context, but the PRNG is not cryptographically strong.
- **CWE-367: Time-of-check Time-of-use (TOCTOU) Race Condition**
The software checks the state of a resource before using that resource, but the resource's state can change between the check and the use in a way that invalidates the results of the check. This can cause the software to perform invalid actions when the resource is in an unexpected state.
- **CWE-393: Return of Wrong Status Code**
A function or operation returns an incorrect return value or status code that does not indicate an error, but causes the product to modify its behavior based on the incorrect result.
- **CWE-399: Resource Management Errors**
Weaknesses in this category are related to improper management of system resources.
- **CWE-400: Uncontrolled Resource Consumption ('Resource Exhaustion')**
The software does not properly restrict the size or amount of resources that are requested or influenced by an actor, which can be used to consume more resources than intended.
- **CWE-470: Use of Externally-Controlled Input to Select Classes or Code ('Unsafe Reflection')**
The application uses external input with reflection to select which classes or code to use, but it does not sufficiently prevent the input from selecting improper classes or code.
- **CWE-787: Out-of-bounds Write**
The software writes data past the end, or before the beginning, of the intended buffer.
- **CWE-835: Loop with Unreachable Exit Condition ('Infinite Loop')**
The program contains an iteration or loop with an exit condition that cannot be reached, i.e., an infinite loop.

D.9 Predicting an estimation of total vulnerabilities across repository types between 2019 and 2025

| Estimating total vulnerabilities for Official images | | | |
|--|--------------------------------------|-----------|-----------|
| | Estimated (total vulnerabilities) | Lower 95% | Upper 95% |
| 2019 | 759 | 391 | 1127 |
| 2020 | 855 | 443 | 1266 |
| 2021 | 951 | 499 | 1402 |
| 2022 | 1047 | 559 | 1535 |
| 2023 | 1143 | 621 | 1665 |
| 2024 | 1239 | 685 | 1793 |
| 2025 | 1335 | 750 | 1919 |
| Increase rate: $y = 96x$ | | | |

Table D.1: Detailed numbers of unique vulnerabilities estimated for Official repository

| Estimating total vulnerabilities for Community images | | | |
|---|--------------------------------------|-----------|-----------|
| | Estimated (total vulnerabilities) | Lower 95% | Upper 95% |
| 2019 | 1491 | 923 | 2059 |
| 2020 | 1682 | 1047 | 2317 |
| 2021 | 1873 | 1176 | 2569 |
| 2022 | 2064 | 1311 | 2816 |
| 2023 | 2255 | 1449 | 3059 |
| 2024 | 2446 | 1590 | 3300 |
| 2025 | 2637 | 1734 | 3538 |
| Increase rate: $y = 191x$ | | | |

Table D.2: Detailed numbers of unique vulnerabilities estimated for Community repository

| Estimating total vulnerabilities for Verified images | | | |
|--|--------------------------------------|-----------|-----------|
| | Estimated (total vulnerabilities) | Lower 95% | Upper 95% |
| 2019 | 891 | 484 | 1298 |
| 2020 | 1006 | 458 | 1554 |
| 2021 | 1121 | 461 | 1781 |
| 2022 | 1236 | 480 | 1991 |
| 2023 | 1350 | 510 | 2191 |
| 2024 | 1465 | 548 | 2383 |
| 2025 | 1580 | 591 | 2569 |
| Increase rate: $y = 114.85x$ | | | |

Table D.3: Detailed numbers of unique vulnerabilities estimated for Verified repository

| Estimating total vulnerabilities for Certified images | | | |
|---|--------------------------------------|-----------|-----------|
| | Estimated (total vulnerabilities) | Lower 95% | Upper 95% |
| 2019 | 175 | 115 | 234 |
| 2020 | 191 | 107 | 275 |
| 2021 | 206 | 104 | 311 |
| 2022 | 224 | 105 | 343 |
| 2023 | 241 | 107 | 374 |
| 2024 | 257 | 111 | 403 |
| 2025 | 274 | 116 | 432 |
| Increase rate: $y = 16.5x$ | | | |

Table D.4: Detailed numbers of unique vulnerabilities estimated for Certified repository

Appendix E

Source code

Due to a limitation of number of pages for the entire thesis, this appendix only contains a part of the source code designed in chapter 4 and implemented in chapter 5 as the Docker vulnerability AnalyZER (DAZER) software used to conduct experiments. Note that DAZER is an open source software made publicly available in its entirety via a dedicated Github repository located at <https://github.com/dockalyzer/dazer>.

E.1 dockahub_api.py

```
1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  """
4      This module contains functions interacting with bother version 1 and 2 of Docker Hub's
5      ↪ API.
6  """
7
8  import json
9  import logging
10 import os
11 import random
12 import re
13 import requests
14 from DAZER import utils
15 from requests.adapters import HTTPAdapter
16 from urllib3.util import Retry
17
18 dockahub_api_v1 = "https://hub.docker.com/api/content/v1/"
19 dockahub_api_v2 = "https://hub.docker.com/v2/"
20 headers = {
21     'User-Agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_1) AppleWebKit/537.36 (KHTML,
22     ↪ like Gecko) '
```



```

22         'Chrome/39.0.2171.95 Safari/537.36'}
23 # -- Retrying the request three times before an exception raises
24 session = requests.Session()
25 retries = Retry(connect=3, backoff_factor=30)
26 adapter = HTTPAdapter(max_retries=retries)
27 session.mount('http://', adapter)
28 session.mount('https://', adapter)
29
30
31 def search_query_v1(query):
32     """
33         Interrogates the version 1 of Docker Hub's search API with the passed query.
34
35         Args:
36             query (string): the query to be searched for
37
38         Returns:
39             dict: the query's result in Json format when successful (note: may
40 ↪ be empty), None otherwise
41
42     """
43     query_result = None
44     search_api = dockerhub_api_v1 + "products/search?q="
45     request = search_api + query # -- e.g. certification_status=certified&page_size=1"
46     response = requests.get(request, headers=headers)
47
48     if response.ok:
49         query_result = response.json()
50
51         if not query_result.get("summaries"):
52             logging.warning("Empty result from Docker Hub's search API version 1")
53         else:
54             logging.error("Request to Docker Hub's search API version 1 failed (change in the
55 ↪ API?)")
56
57     return query_result
58
59
60 def get_repository_query_v1(repository, is_insecure=False):
61     """
62         Retrieves the passed repository using Docker Hub's image API version 1.
63
64         Note 1: The images retrieved using version 1 of Docker Hub's image API contains a
65 ↪ lot more details than the
66         ones retrieved using version 2.
67
68         Note 2: The version 1 of Docker Hub's image API is NOT able to retrieve
69 ↪ 'community' images (only 'official',

```

```

69         'certified' and 'verified').
70
71         Note 3: For special purposes, this function may be called in a manner that is
↪ insecure by omitting the logging
72         of unexpected events.
73
74
75         Args: repository           (string):           the repository to be retrieved (note:
↪ the API id of the repository to be
76         retrieved is also accepted - e.g. 3567eb02-06cf-48e2-978f-cbd86cc3e61d)
↪ is_insecure   (bool):   whether the
77         function should log unexpected events
78
79
80         Return:
81         dict:           the query's result in Json format when successful (note: may
↪ be empty), None otherwise
82
83         """
84         query_result = None
85         image_api = dockerhub_api_v1 + "products/images/"
86         request = image_api + repository
87         response = requests.get(request, headers=headers)
88
89         try:
90             session.get(request)
91
92             if response.ok:
93                 query_result = response.json()
94
95                 if "message" in query_result:
96                     logging.warning(repository + " got empty result from Docker Hub's
↪ image API version 1")
97
98             elif not is_insecure:
99                 logging.error("Request to Docker Hub's image API version 1 failed (change
↪ in the API?)")
100
101         except requests.exceptions.ConnectionError:
102             logging.exception("Error for request: " + request)
103
104         return query_result
105
106
107 def search_query_v2(query):
108     """
109         Interrogates the version 2 of Docker Hub's search API with the passed query.
110
111
112         Args:
113         query           (string):           the query to be searched for

```

```

114
115
116             Returns:
117                 dict:         the query's result in json format when successful (note: may
↪ be empty), None otherwise
118
119             """
120             query_result = None
121             search_api = dockerhub_api_v2 + "search/repositories/?query="
122             request = search_api + query # -- e.g. library&is_official=true&page=1"
123             response = requests.get(request, headers=headers)
124
125             try:
126                 session.get(request)
127                 if response.ok:
128                     query_result = response.json()
129
130                     if not query_result.get("results"):
131                         logging.warning("Empty result from Docker Hub's search API
↪ version 2")
132                 else:
133                     logging.error("Request to Docker Hub's search API version 2 failed
↪ (change in the API?)")
134
135             except requests.exceptions.ConnectionError:
136                 logging.exception("Error for request: " + request)
137
138             return query_result
139
140
141 def get_repository_query_v2(repository, is_insecure=False):
142     """
143         Retrieves the passed repository using Docker Hub's image API version 2.
144
145
146         Note 1: The images retrieved using version 2 of Docker Hub's image API contain
↪ less details than the ones
147             retrieved using version 1.
148
149         Note 2: The version 2 of Docker Hub's image API is NOT able to retrieve
↪ 'certified' and 'verified' images (
150             only 'official' and 'community').
151
152
153         Args: repository         (string):         the repository to be retrieved (note:
↪ the API id of the repository to be
154             retrieved is also accepted - e.g. 3567eb02-06cf-48e2-978f-cbd86cc3e61d)
155         is_insecure (bool): whether the
↪ function should log unexpected events
156
157

```

```

158             Return:
159                 dict:         the query's result in Json format when successful (note: may
↪ be empty), None otherwise
160
161         """
162         query_result = None
163         image_api = dockerhub_api_v2 + "repositories/"
164         request = image_api + repository
165         response = requests.get(request, headers=headers)
166
167         try:
168             session.get(request)
169
170             if response.ok:
171                 query_result = response.json()
172
173                 if "detail" in query_result:
174                     logging.warning(repository + " got empty result from Docker Hub's
↪ image API version 2")
175
176                 elif not is_insecure:
177                     logging.error("Request to Docker Hub's image API version 2 failed (change
↪ in the API?)")
178
179         except requests.exceptions.ConnectionError:
180             logging.exception("Error for request: " + request)
181
182         return query_result
183
184
185 def get_repository_tags_query_v2(repository):
186     """
187         Retrieves all the available tags for the passed repository using Docker Hub's
↪ tags API.
188
189
190         Args:
191             repository         (string):         the repository to be retrieved for
↪ tags
192
193
194         Returns:
195             list:         a list of retrieved tags for the passed repository
196
197     """
198     tags = []
199     tags_api = dockerhub_api_v2 + "repositories/"
200     request = tags_api + repository + "/tags/?page_size=1"
201     response = requests.get(request, headers=headers)
202
203     try:

```

```

204         session.get(request)
205
206     if response.ok and not response.json().get("detail"):
207         total_images = response.json().get("count") # -- the total number of
208             ↳ images to be fetched
209         fetching_size = 50 # -- the number of json objects to fetch per request
210         total_pages = total_images // fetching_size + 1 \
211             if total_images % fetching_size > 0 else total_images //
212             ↳ fetching_size
213         # -- the total number of pages to request for complete retrieval
214
215         for page in range(1, total_pages + 1):
216             # -- Retrieving one page
217             request = tags_api + repository + "/tags/?page_size=" +
218                 ↳ str(fetching_size) + "&page=" + str(page)
219             response = requests.get(request, headers=headers)
220
221             if response.ok and response.json().get("results"):
222                 images = response.json().get("results")
223
224                 for image in images:
225                     tags.append(image.get("name"))
226             else:
227                 logging.warning(repository + " got empty result from
228                     ↳ Docker Hub's tags API")
229                 logging.warning("Sent request: " + request)
230
231         else:
232             logging.error("Request to Docker Hub's tags API failed (change in the
233                 ↳ API?)")
234
235     except requests.exceptions.ConnectionError:
236         logging.exception("Error for request: " + request)
237
238     return tags
239
240 def get_repository_tag_query_v2(repository, tag):
241     """
242     Retrieves the passed tag object for the passed repository using Docker Hub's tags
243     ↳ API.
244
245     Args:
246         repository (string): the repository to retrieve the passed
247         ↳ tag for
248         tag (string): the tag to retrieve
249
250     Returns:
251         dict: a dictionary representing the retrieved tag object
252 
```

```

248     """
249     tag_json = ""
250     tags_api = dockerhub_api_v2 + "repositories/"
251     request = tags_api + repository + "/tags/" + tag
252     response = requests.get(request, headers=headers)
253
254     try:
255         session.get(request)
256         if response.ok and not response.json().get("detail"):
257             tag_json = response.json()
258         else:
259             logging.error("Request to Docker Hub's tags API failed (change in the
↳ API?)")
260
261     except requests.exceptions.ConnectionError:
262         logging.exception("Error for request: " + request)
263
264     return tag_json
265
266
267 def has_repository_tag_query_v2(repository, tag):
268     """
269         Verifies whether the passed repository has a the passed tag using Docker Hub's
↳ tags API.
270
271         Args:
272             repository (string): the repository to be retrieved for the
↳ passed tag
273             tag (string): the tag to be verified for
↳ existence
274
275         Returns:
276             bool: True if the passed repository has the passed tag, False otherwise
277
278     """
279
280     has_latest_tag = False
281
282     tags_api = dockerhub_api_v2 + "repositories/" + repository + "/tags/"
283     request = tags_api + tag
284     response = requests.get(request, headers=headers)
285
286     try:
287         session.get(request)
288
289         if response.ok:
290             has_latest_tag = True
291
292     except requests.exceptions.ConnectionError:
293         logging.exception("Error for request: " + request)
294

```

```

295
296         return has_latest_tag
297
298
299 def get_repository_type(repository):
300     """
301     Retrieves the type of the passed repository as being of of the following types: official,
↪ certified, verified,
302     community.
303
304
305     Args:
306         repository (string): the repository to be retrieved
307
308
309     Returns:
310         string: the type of the passed repository if successful,
↪ an empty string otherwise
311
312     """
313     image_type = ""
314     image = get_repository_query_v1(repository, is_insecure=True)
315
316     if image:
317         default_plan = image.get("plans")[0]
318         namespace = default_plan.get("repositories")[0].get("namespace")
319
320         if namespace == "library":
321             image_type = "official"
322
323         elif namespace == "store":
324             certification_status = default_plan.get("certification_status")
325             image_type = "certified" if certification_status == "certified" else
↪ "verified"
326
327     else:
328         repository = get_repository_query_v2(repository, is_insecure=True)
329
330         if repository is not None and "namespace" in repository:
331             image_type = "community"
332
333     return image_type
334
335
336 def get_latest_versioned_tag(repository):
337     """
338     Retrieves the most recent versioned tag (last pushed) of the passed repository.
339
340
341     Note:
342         A versioned tag may have different formats such as:

```

```

343             - 1
344             - 1.2
345             - 1.2.3
346             - v1.2.3
347             - 1.2.3-alpine
348             - v1.2.3-alpine
349
350
351         Args:
352             repository (string): the repository to be retrieved for its
↪      most recent versioned tag
353
354
355         Returns:
356             string: the most recent versioned tag of the passed
↪      repository
357
358         """
359         tag = ""
360         tags = get_repository_tags_query_v2(repository)
361
362         if len(tags) != 0:
363             tag = tags[0]
364
365         return tag
366
367
368     def get_official_images():
369         """
370             Retrieves the name, latest tag and slug name of all the official images available
↪      on Docker Hub.
371
372
373         Note 1:
374             The retrieval is executed iteratively by fetching 50 Json objects (i.e.
↪      images) per request.
375
376
377         Note 2:
378             Official repositories are located in the '/library' namespace.
379
380
381         Returns:
382             list: a list of dictionaries with the retrieved information
383
384         """
385         images = []
386         excluded_repositories = ["scratch", "rocket.chat"] # -- repositories which are indexed
↪      by the Docker Hub API but
387         # do not contain real images
388

```



```

389     query = "library&is_official=true&page_size=1"
390     result = search_query_v2(query)
391
392     if result is not None and result.get("results"):
393         total_images = result.get("count") # -- the total number of images to be
394         ↪ fetched
395         fetching_size = 50 # -- the number of json objects to fetch per request
396         total_pages = total_images // fetching_size + 1 \
397             if total_images % fetching_size > 0 else total_images // fetching_size
398         # -- the total number of pages to request for complete retrieval
399
400         for page in range(1, total_pages + 1):
401             # -- Retrieving one page
402             query = "library&is_official=true&page_size=" + str(fetching_size) +
403             ↪ "&page=" + str(page)
404             result = search_query_v2(query)
405
406             if result and result.get("results"):
407                 for image in result.get("results"):
408                     repository = str(image.get("repo_name"))
409                     tag = ""
410
411                     if repository not in excluded_repositories:
412                         # -- Retrieving the necessary information for
413                         ↪ each of the repository on the current page
414                         tag = get_latest_versioned_tag("library/" +
415                         ↪ repository)
416
417                     if tag:
418                         images.append({
419                             "name": repository,
420                             "tag": tag,
421                         })
422                     else:
423                         logging.info("%s - Image retrieval
424                         ↪ skipped (missing tag)", repository)
425
426     return images
427
428 def get_certified_images():
429     """
430     Retrieves the name, latest tag and slug name of all the certified images
431     ↪ available on Docker Hub.
432
433     Note 1:
434     The retrieval is executed iteratively by fetching 50 Json objects (i.e.
435     ↪ images) per request.
436
437     Note 2: Certified repositories are located in the '/store/<username>' namespace
438     ↪ and use unpredictable tags (

```

```

433         e.g. '/store/ibmcorp/websphere-liberty:microProfile2')
434
435
436     Returns:
437         list:         a list of dictionaries with the retrieved information
438
439     """
440     images = [] # -- the list of dictionaries containing all the certified images' names,
441                 ↳ tags and slug names
442     query = "&type=image&certification_status=certified&page_size=1"
443     result = search_query_v1(query)
444
445     if not result.get("message") and result.get("summaries"):
446         total_images = result.get("count") # -- the total number of images to be
447                 ↳ fetched
448         fetching_size = 50 # -- the number of json objects to fetch per request
449         total_pages = total_images // fetching_size + 1 \
450             if total_images % fetching_size > 0 else total_images // fetching_size
451         # -- the total number of pages to request for complete retrieval
452
453         for page in range(1, total_pages + 1):
454             # -- Retrieving one page
455             query = "&type=image&certification_status=certified&page_size=" +
456                 ↳ str(fetching_size) + "&page=" + str(page)
457             result = search_query_v1(query)
458
459             if result is not None and result.get("summaries"):
460                 for image in result.get("summaries"):
461                     # -- Retrieving name and tag
462                     image_name = image.get("slug")
463                     name = ""
464                     tag = ""
465                     result = get_repository_query_v1(image_name)
466
467                     if not result.get("message"):
468                         # -- Determining the retrieving method
469                         if re.search("microsoft", image_name):
470                             # -- Microsoft specific retrieval
471                             description =
472                                 ↳ result.get("full_description")
473                             name = re.search("docker pull
474                                 ↳ ([.\\w-]+)", description)
475                             # -- e.g. 'docker pull
476                                 ↳ mcr.microsoft.com/oryx/nodejs' or
477                                 ↳ 'docker pull microsoft-mssql-tools'
478
479                             if not name:
480                                 logging.info(
481                                     "Repository skipped ('" +
482                                     ↳ image_name + "' does
483                                     ↳ not contain images
484                                     ↳ or explicit pulling
485                                     ↳ instructions)")

```

```

476                                     continue
477
478                                     name = name.group(1)
479                                     tag = re.search("docker
↳ pull.*?:([\w\d:-]+)",
↳ description).group(1) if re.search(
"docker pull.*?:([\w\d:-]+)",
↳ description) else "latest"
↳ # -- e.g. '3.2.1', 'v3'
480
481     else:
482         # -- Normal retrieval
483         default_plan = result.get("plans")[0]
484         repository =
↳ default_plan.get("repositories")[0]
485         version = default_plan.get("versions")[0]
486         name = repository.get("namespace") + "/"
↳ + repository.get("reponame")
487         tag = version.get("tags")[0].get("value")
↳ if version.get("tags")[0].get(
"value") else "latest"
488
489
490         images.append({
491             "name": name,
492             "tag": tag,
493             "slug_name": image_name
494         })
495     return images
496
497
498 def get_verified_images():
499     """
500     Retrieves the name, latest tag and slug name of all the verified images available
↳ on Docker Hub.
501
502
503     Note 1:
504     The retrieval is executed iteratively by fetching 50 Json objects (i.e.
↳ images) per request.
505
506
507     Note 2: Verified repositories which are non-Microsoft are located in the
↳ '/store/<username>' namespace and use
508     unpredictable tags (e.g. 'store/saplabs/hanaexpresssa:2.00.033.00.20180925.2').
↳ Microsoft repositories use a
509     complete different namespace scheme proper to them and tend to use the 'latest'
↳ tag for all of their images.
510
511
512     Note 3: Certain Microsoft repositories are listed out with different names on
↳ Docker Hub, but are actually the
513     same as they use the same docker pull command (e.g. the 'Oryx node-x.y'
↳ repositories)

```

```

514
515
516         Returns:
517             list:         a list of dictionaries with the retrieved information
518
519     """
520     images = []
521     query = "&type=image&image_filter=store&page_size=1"
522     result = search_query_v1(query)
523
524     if result is not None and result.get("summaries"):
525         total_images = result.get("count") # -- the total number of images to be
526         ↪ fetched
527         fetching_size = 50 # -- the number of json objects to fetch per request
528         total_pages = total_images // fetching_size + 1 \
529         if total_images % fetching_size > 0 else total_images // fetching_size
530         # -- the total number of pages to request for complete retrieval
531
532         for page in range(1, total_pages + 1):
533             # -- Retrieving one page
534             query = "&type=image&image_filter=store&page_size=" + str(fetching_size)
535             ↪ + "&page=" + str(
536                 page) # -- returns both Official and Verified images
537             result = search_query_v1(query)
538
539             if result is not None and result.get("summaries"):
540                 for image in result.get("summaries"):
541                     # -- Verifying that the image is of type Verified
542                     image_name = image.get("slug")
543                     image_type = get_repository_type(image_name)
544
545                     if image_type is "verified": # or image_type is
546                     ↪ "certified":
547                         # -- Retrieving name and tag
548                         result = get_repository_query_v1(image_name)
549                         name = ""
550                         tag = ""
551
552                     if not result.get("message"):
553                         # -- Determining the retrieving method
554                         if re.search("microsoft", image_name):
555                             # -- Microsoft specific
556                             ↪ retrieval
557                             description =
558                             ↪ result.get("full_description")
559                             name = re.search("docker pull
560                             ↪ ([.\\w-]+)", description)
561                             # -- e.g. 'docker pull
562                             ↪ mcr.microsoft.com/oryx/nodejs'
563                             ↪ or 'docker pull
564                             ↪ microsoft-mssql-tools'

```

```

556
557         if not name:
558             logging.info(
559                 "Repository
                    ↳ skipped ('"
                    ↳ + image_name
560                 + "' does not
                    ↳ contain
                    ↳ images or
                    ↳ explicit
                    ↳ pulling
                    ↳ instructions)")
561             continue
562
563         name = name.group(1)
564         tag = re.search("docker
                    ↳ pull.*?:([.\w\d:-]+)",
                    ↳ description).group(1) if
                    ↳ re.search(
565                 "docker
                    ↳ pull.*?:([.\w\d:-]+)",
                    ↳ description) else
                    ↳ "latest" # -- e.g.
                    ↳ '3.2.1', 'v3'
566
567         # -- Filtering out a potential
568         ↳ duplicate repository
569         is_retrieved = False
570
571         for image in images:
572             if name ==
                    ↳ image.get("name"):
                    ↳ is_retrieved =
                    ↳ True
                    ↳ break
573
574
575         if is_retrieved:
576             continue
577
578     else:
579         # -- Normal retrieval
580         default_plan =
                    ↳ result.get("plans")[0]
581         repository =
                    ↳ default_plan.get("repositories")[0]
582         version =
                    ↳ default_plan.get("versions")[0]
583         name =
                    ↳ repository.get("namespace")
                    ↳ + "/" +
                    ↳ repository.get("reponame")

```

```

584                                     tag =
↳                                     version.get("tags")[0].get("value")
↳                                     if
↳                                     version.get("tags")[0].get(
585                                         "value") else "latest"
586
587                                     images.append({
588                                         "name": name,
589                                         "tag": tag,
590                                         "slug_name": image_name
591                                     })
592     return images
593
594
595 def get_community_images(x_images):
596     """
597     Retrieves the name and latest tag of the passed number of community images among the most
↳     popular ones available
598     on Docker Hub.
599
600
601     Note 1:
602     The retrieval is executed iteratively by fetching 50 Json objects
↳     (i.e. images) per request.
603
604
605     Note 2:
606     Community repositories are located in the '<username>' namespace
↳     (e.g. '/pivotalcf/pivnet-resource:latest')
607
608
609     Note 3: The returned images are chosen randomly between the passed number
↳     of images times 3 for increasing
610     randomness across multiple calls to this function.
611
612
613     Args:
614     x_images      (int):      the base number of images to be retrieved
615
616
617     Returns: tuple:      two lists of dictionaries with the retrieved
↳     information (one with the first passed number
618     of retrieved images, another one with the rest of the retrieval)
619
620     """
621     images = []
622     query = "%2B&is_official=false&ordering=-pull_count&page_size=1"
623     result = search_query_v2(query)
624     excluded = ["bugswarm/artifacts", "microsoft/oms", "programmerq/scaletest",
↳     "newrelic/nrsysmond",
625     "weaveworks/weave-npc"]

```

```

626
627     if result is not None and result.get("results"):
628         total_images = x_images * 3 # -- the total number of images to analyze, original
        ↪ value times three
629         fetching_size = 50 # -- the number of json objects to fetch per request
630         total_pages = total_images // fetching_size + 1 \
631             if total_images % fetching_size > 0 else total_images // fetching_size
632         # -- the total number of pages to request for complete retrieval
633
634         counter = 0 # -- counting for every single image that are being analyzed
635         for page in range(1, total_pages + 1):
636             # -- Retrieving one page
637             query = "%2B&is_official=false&ordering=-pull_count&page_size=" +
        ↪ str(fetching_size) + "&page=" + str(page)
638             result = search_query_v2(query)
639
640             if result is not None and result.get("results"):
641                 for image in result.get("results"):
642                     if counter < total_images:
643                         repository = str(image.get("repo_name"))
644                         tag = get_latest_versioned_tag(repository)
645                         if repository not in excluded:
646                             if tag:
647                                 images.append({
648                                     "name": repository,
649                                     "tag": tag
650                                 })
651
652                             else:
653                                 logging.info("%s - Image
        ↪ retrieval skipped (missing
        ↪ tag)", repository)
        ↪ counter -= 1
654
655                             else:
656                                 logging.info("%s - repository skipped due
        ↪ to DNS resolve problems",
        ↪ repository)
        ↪ counter -= 1
657
658                             else:
659                                 break
660
661             counter += 1
662
663         random.shuffle(images)
664         requested = images[:x_images]
665         remaining = images[x_images:]
666
667         return requested, remaining
668
669
670 def get_image_extrainfo(image_name):

```

```

671         """
672         Retrieves and parses extra information (type and total pulled) for the image with the
↪ passed name using Docker
673         Hub's repository API.
674
675
676         Note 1: Images of type 'community' cannot be retrieved using the version 1 of
↪ Docker Hub's API and are
677         therefore retrieved via version 2. Other image types ('certified', 'verified',
↪ 'official') are retrieved using
678         version 1.
679
680
681         Note 2: The passed image name must be a slug name or a Hub ID for certified and
↪ verified images, as version of
682         the Docker Hub only understands slug names and image ids.
683
684
685         Args:
686         image_name (string): the name of the image to be retrieved
↪ and parsed for extra information
687
688
689         Return:
690         dict: a dictionary containing the parsed information from
↪ retrieved image
691
692         """
693         repository_type = get_repository_type(image_name)
694         total_pulled = 0
695
696         if repository_type != "community":
697             # -- Using version 1 of Docker Hub's repository API
698             result = get_repository_query_v1(image_name)
699
700             if result:
701                 total_pulled = result.get("popularity")
702
703         else:
704             # -- Using version 2 of Docker Hub's repository API
705             result = get_repository_query_v2(image_name)
706
707             if result:
708                 total_pulled = result.get("pull_count")
709
710         return {
711             "type": repository_type,
712             "total_pulled": total_pulled
713         }
714
715

```



```

716 def get_image_parent(repository, image_layers, db_type):
717     """
718     Retrieves the tagged parent of the image with the passed layers, belonging to the passed
↪ repository from the
719     passed type of parent database.
720
721
722     Args:
723         repository          (string):          the name of the repository
↪ that the image to retrieve the parent for belongs to
724         image_layers        (string):          the complete layer combination of
↪ the image to retrieve the parent for
725         db_type             (string):          the type of parent
↪ database to retrieve from ('official' and 'verified')
726
727
728     Returns:
729         dict:                  the parent repository name and tag of the image with
↪ the passed layers or an empty dict if no parent is found
730
731
732     Raises:
733         IOError:              on database file reading failure
734
735     """
736     parent = dict()
737     image_id_length = 12 # -- the length of a single fs layer id (e.g. 6ae821421a7d)
738     image_ids = [image_layers[id:id + image_id_length] for id in range(0, len(image_layers),
↪ image_id_length)]
739     # -- splitting image_layers into a list of single layer ids with a length of
↪ image_id_length
740     home = os.path.dirname(os.path.realpath(__file__))
741     base_dir = os.path.join(home, "json/parent-db")
742
743     if db_type == "official":
744         parent_files = [os.path.join(base_dir, resource) for resource in
↪ os.listdir(base_dir) if
745             re.search("(official)", resource)]
746
747     elif db_type == "verified":
748         parent_files = [os.path.join(base_dir, resource) for resource in
↪ os.listdir(base_dir) if
749             re.search("(verified)", resource)]
750
751     else:
752         return ""
753
754     parent_file = utils.get_most_recent_file(parent_files)
755     # -- the file containing all the unique layers for all the images in the repositories of
↪ the passed type
756

```

```

757     try:
758         with open(parent_file, "r") as file:
759             base_db = json.loads(file.readline())
760
761             repositories = list(base_db.keys())
762             index = repositories.index(repository)
763             repositories = repositories[:repositories.index(repository)] +
764             ↪ repositories[repositories.index(
765                 repository) + 1:] # -- all the repositories apart from the passed one
766
767     except IOError:
768         raise
769
770     except:
771         pass
772
773     current_layers = ""
774
775     # -- Retrieving parent
776     for image_id in image_ids:
777         current_layers = current_layers + image_id
778
779         for repo in repositories:
780             for image in base_db[repo]:
781                 if current_layers == image.get("fs_layers"):
782                     # -- The repository is a parent
783                     parent = {
784                         "name": repo,
785                         "tag": image.get("image_tag")
786                     }
787
788     # -- Continuing, as the image's closest parent is among the lower layers
789     return parent

```