BLEKINGE TEKNISKA HÖGSKOLA

BTH

DEGREE PROJECT FOR MASTER OF SCIENCE IN ENGINEERING

COMPUTER SECURITY

Static Vulnerability Analysis of Docker Images

Michael Falk | Oscar Henriksson

Blekinge Institute of Technology, Karlskrona, Sweden, 2017

Supervisor: Emiliano Casalicchio, Department of Computer Science and Engineering, BTH

Abstract

Docker is a popular tool for virtualization that allows for fast and easy deployment of applications and has been growing increasingly popular among companies. Docker also include a large library of images from the repository Docker Hub which mainly is user created and uncontrolled. This leads to low frequency of updates which results in vulnerabilities in the images.

In this thesis we are developing a tool for determining what vulnerabilities that exists inside Docker images with a Linux distribution. This is done by using our own tool for downloading and retrieving the necessary data from the images and then utilizing Outpost24's scanner for finding vulnerabilities in Linux packages. With the help of this tool we also publish statistics of vulnerabilities from the top downloaded images of Docker Hub.

The result is a tool that can successfully scan a Docker image for vulnerabilities in certain Linux distributions. From a survey over the top 1000 Docker images it has also been shown that the amount of vulnerabilities have increased in comparison to earlier surveys of Docker images.

Keywords: Docker, Containerization, Vulnerability analysis, Vulnerability scanning

Sammanfattning

Docker är ett populärt verktyg för virtualisering som används för att snabbt och enkelt sätta upp applikationer och har vuxit sig populärt bland företag. Docker inkluderar även ett stort bibliotek av images från datakatalogen Docker Hub vilket huvudsakligen består av användarskapat och okontrollerat innehåll. Detta leder till en låg frekvens av uppdateringar vilket i sin tur resulterar i sårbarheter i images.

I denna uppsats utvecklar vi ett verktyg för att bestämma vilka sårbarheter som existerar inuti Docker images med Linux distributioner. Detta möjliggörs genom vårt utvecklade verktyg för att ladda ner och hämta ut nödvändig data från images som sedan använder Outpost24s skanner för att hitta sårbarhter i Linux paket. Med hjälp utav detta verktyget publicerar vi även statistik över sårbarheter från images med mest nedladdningar på Docker Hub.

Resultatet är ett verktyg som kan skanna en Docker image efter sårbarheter i vissa Linux distributioner. Från en undersökning av topp 1000 Docker images har det också visats att antalet sårbarheter har ökat i jämförelser med tidigare studier av Docker images.

Nyckelord: Docker, Containerisering, Sårbarhetsanalys, Sårbarhetsskanning

Preface

This thesis is a collaboration between two students at the programme *Master in Science of Engineering: Computer security* at Blekinge Institute of Technology in Karlskrona in southern Sweden. We have been working together with Outpost24, a company with headquarter in Karlskrona that specialize in vulnerability management technology and services. With their help we researched and developed the company's first steps in the area of vulnerability management in Docker.

Acknowledgements We want to thank our supervisor Emiliano Casalicchio who supported us through the project and helped us with the structure of the thesis. We also want to make a special thank to Martin Jartelius, John Stock and Mattias Thidell at Outpost24 who gave us ideas and feedback on our work, we would never have made it without the breakfast provided at the office.

Nomenclature

Acronyms

API	Application Programming Interface
CVE	Common Vulnerabilities and Exposures
CVSS	Common Vulnerability Scoring System
NVD	National Vulnerability Database
OS	Operating system
SDK	Standard Development Kit
VM	Virtual Machine

List of Figures

3.1 The difference between a virtual machine and container 7
3.2 The different modules of the developed software 12

List of Tables

4.1 Distributions among Docker Hub images 14
4.2 Percentage and number of images with vulnerabilities in different Linux distributions 15
4.3 Number of vulnerabilities in different Linux distributions 15
4.4 Comparison of a normal Ubuntu image and a vulnerable Ubuntu image 15

Table of Contents

Abstract i

Sammanfattning (Swedish) iii

Preface v

Nomenclature vii

Acronyms vii

List of Figures viii

List of Tables ix

Table of Contents xi

1 Introduction 1

1.1 Introduction 1

1.2 Background 1

1.3 Objectives 2

1.4 Delimitations 2

1.5 Thesis question and technical problem 2

1.6 Outline 3

2 Theoretical Framework 5

2.1 Virtualization and containers 5

2.2 Internal security of Docker 5

2.3 External threats to Docker 5

2.4 Security in the Docker repository 6

3 Method 7

3.1 Docker Engine 7

3.2 Extracting image data 8

3.3 Data collection 9

3.4 Extracting Package information 9

3.5 Scanning for vulnerabilities 11

3.6 Program design 11

3.7 Validation 12

4 Results 13

4.1 Images from Docker Hub 13

4.2 Extracting OS distribution 13

4.3 Extracting file system 14

4.4 Reading report files 14

4.5 Validation of result 15

4.6 Report file 15

5 Discussion 17

6 Conclusions 19

7 Recommendations and Future Work 21

References 23

1 INTRODUCTION

1.1 Introduction

Containerization is a type of virtualization technology that recently has grown in users much thanks to the Docker project. The purpose of containers in Docker is to be able to easily set up virtual instances on top of the host operating system and letting them use the host operating systems kernel, while still remaining isolated from each other and the host. This results in lightweight virtual environments that can be deployed in a short period of time, which may be appealing in use cases that involve development and deployment of applications.

To build the containers Docker uses images which can be created and uploaded to public registries such as Docker Hub by both organizations and individual users [1]. Docker Hub is the official registry of images that is hosted by Docker, and with over 650,000 registered users this is the largest host of public Docker images [2]. When uploading images to the Docker Hub registry users may choose to store it in a private repository for only selected users to be able to use or in a public repository for anyone to fetch. This also means that the images are updated and maintained only by the users who created the repository. Docker Hub also features official repositories which are reviewed by Docker and are promoted in their registry, however the number of unofficial repositories greatly outnumbers the official ones.

Since the repositories on Docker Hub are only updated and maintained by users this creates a problem with the security of the images due to the lack of control from Docker on how often the libraries and applications in these images are updated. The images can go months without an update and if a image is using outdated libraries or applications it could contain vulnerabilities that potentially could compromise the container system or the host operating system. This is especially true in the case of containers since it works much closer to the host operating system than a normal Virtual Machine (VM) would do. Furthermore if an attacker uploads an image containing malware this could allow the attacker to remotely access the application a user have deployed.

1.2 Background

Due to Docker rising in popularity the need for monitoring and managing vulnerabilities in Docker is becoming greater. Earlier studies have shown over 30% of the official image repositories contains vulnerabilities[9] and if a developer would use Docker as a solution they could have a vulnerable system which creates the need for assessing the security flaws in the Docker images. This problem have been examined by Banyanops who have come to the conclusion that the images in Docker Hub needs to be analyzed in real-time. This will be covered in more detail in the theoretical framework. The current recommended way by Docker to get a overview for vulnerabilities in a specific image is to use their web service, called Docker Security Scanning, which today only supports scanning images directly on Docker Hub and is a paid service by Docker [3].

Outpost24 is a company that provides vulnerability management for other companies to protect their computer and network systems. The company has given the task to produce a method that directly scans a large number of images and detects vulnerabilities in these. This tool should also be automated and regularly check for new vulnerabilities, and for each image summarizes the found vulnerabilities for simplifying the process of assessing the security flaws contained

in Docker images. This work has the potential to help provide a more secure environment for developers launching their applications using Docker by providing a better way to assess the security of an image.

1.3 Objectives

The objective of our research is to create a method for automating the scanning of Docker images and test them for vulnerabilities, and also summarize the results of the scans in order to make security assessment of different systems using Docker more efficient. We want to produce a solution that automatically analyses a set of images from Docker Hub's library and then returns a report over vulnerabilities that are found in them. The plan is to do this for the most popular images in the library and to use a predefined list of vulnerabilities for scanning. Then we will publish statistics of the scanning to give a good overview of the security flaws in the images from Docker's repository. In order to test our solution a prototype will be developed which will then be tested against the images in the Docker Hub and have the results of the found vulnerabilities summarized for each image in a format that is easy to read to simplify the process of assessing the security of a Docker image.

1.4 Delimitations

The study conducted in this thesis is limited to the vulnerabilities listed in the Common Vulnerabilities and Exposures (CVE) database and also limited to the top 1000 images in the repository. Also images with a Linux Operating system (OS) will be the only type of image included in the work and of these only Ubuntu, Debian, Alpine and CentOS will be supported. There are also different tags for an image specifying different versions, for the images in this thesis the *latest* tag will be used to only include the newest version. The thesis mainly focuses on the method for detecting vulnerabilities in Docker images and collecting the results, the security of the Docker infrastructure and configuration will not be researched in our work but will only be reviewed from related work. Furthermore the vulnerabilities to be scanned are the installed packages on the image, executable files in other places will not be included.

1.5 Thesis question and technical problem

From the researched problems we have identified the need for surveying the security flaws in Docker images. We want to know how a solution for detecting vulnerabilities in Docker images can be conducted more efficiently and how the result of this scan can be presented in a manner that is more clear and easier to access.

The research questions that will be answered is.

1. How can a method for automating the extraction and vulnerability scanning of packages in a Docker image be constructed?
2. How can we present the results of a vulnerability scan of a Docker image in a format that is clear and detailed?

Our hypothesis for the first question is that it is possible to to detect vulnerabilities that may exist in an Docker image and summarize them without the need to browse Docker Hub and without

the need to start up a container.
Our hypothesis of the second question is that it is possible to present the result of a vulnerability scan in a data format that may then be processed to produce a report.

1.6 Outline

In chapter 2 research related to the subject will be reviewed as the methods and information in these will be used as a base to build our own method to further develop a solution to the problem. Chapter 3 explains our method for building the tool required for our research. In this chapter the method for collection of data and presentation of result is also explained. In chapter 4 verification of the method and the results will be presented. In chapter 5 there will be a discussion about the results and possible improvements along with the answer to the research questions. In chapter 6 the thesis will be concluded. In chapter 7 recommendations for future work that can improve the study will be suggested.

2 THEORETICAL FRAMEWORK

2.1 Virtualization and containers

Virtualization has long been the back end for different cloud systems utilizing virtual machine technology such as Virtualbox, VMware or Hyper-V according to Manu et al. who in this study explored security and isolation in the cloud with the help of Docker [4]. They pointed out that that virtualization emulates hardware in the host OS of a physical machine, and there also exists a hypervisor layer that is used to help with resource management between VMs. On the emulated hardware a separate guest OS can be run creating isolated machines. The authors of this study concluded that in contrast containers run directly run on top of the host OS making them more lightweight but also less secure as a compromised container could potentially allow an attacker to gain access to the host OS which is less likely to happen on a VM.

2.2 Internal security of Docker

The internal security of Docker is an important part when motivating the vulnerability scanning on parts of this software. One study made by Bui[5] tackles this problem by analyzing the different parts of Docker such as how it isolates processes, file system and network, reviewing the technologies that Docker uses to make this possible. The author also examines different low-level security systems which may be implemented in a Linux environment to further isolate Docker containers from the host system. Results of this work shows that Docker provides a relative high level of isolation. However the author notes that some issues are found in the network isolation where Docker may be vulnerable to certain attacks between containers, and another potential security problem may be if a container is run with high privileges.

According to Combe, Martin and Di Pietro[6] the Docker images composed of different layers and there will always be a base layer. When a modification to a image is made the changes is stored in a new layer and each layer also contains metadata. There are also several vulnerabilities related to Docker isolation, first the local default configuration is considered relatively secure but it is possible to change the configuration so that the system becomes less secure. One example that the authors give is that one may change the network configuration so containers no longer gets isolated which gives the container full access to the host network. It is also possible to change the configuration so that the host and container is under the same UTS namespace allowing the container to see the hosts name and domain. The most serious flaw in the configuration is the possibility to change the privilege of a container e.g. a container can be set with capabilities which can make the container manipulate the sensitive paths /proc and /write directories as well as changing the host kernel parameters. This paper concludes that improper configuration can result in data leakage and denial of service.

2.3 External threats to Docker

Mohallel et al. conducted experiments in their work regarding the security of Docker containers[7]. By making a comparison in security of a web server installed in a container and on a base operating system they wanted to know the difference in attack surfaces between the two. In this experiment they installed Apache, Nginx and MySQL on a Debian server for the base operating system. Since Docker Hub provides different images for these services the researchers used three different containers, one for each of the before mentioned. The test itself was done by using a network scanner locally to detect vulnerabilities. The results gotten from this research showed

that a Docker container has increased attack surfaces due to the Docker base image exposing certain vulnerabilities, with the conclusion that using a container may possibly open up for security risks.

In their work Combe et al. are taking a deep dive into the potential vulnerability issues that different scenarios of running Docker may have[8]. This is done by analyzing several use cases with the focus on vulnerabilities in containers, images, code, configuration and the host kernel. The authors are making interesting points regarding Docker images where the vulnerability may be in two parts of it, one being where a vulnerability lies in the base image, such as Debian or Ubuntu, which may contain outdated packages that can be exposed to attacks. The other possibility for vulnerabilities to exist is according to the authors the external code that a user may add to the base image when creating their own image, including vulnerable programs in the new image.

2.4 Security in the Docker repository

The images distributed via Docker Hub are a source of vulnerabilities according to Combe et al[8]. Docker supports automated builds which automatically fetches the latest version of a image on GitHub into the Docker Hub repository. In the event of a compromised GitHub account an attacker could implement malware in a image which is then automatically sent to the repository. When the image is then pulled and launched as a container it could put the host machine at risk.

A study made by Gummaraju, Desikan, and Turner at BanyanOps shows that over 30% of the official images contain vulnerabilities with high severity, and if public images are also considered the amount of vulnerable images rises to 40%[9]. The study have been conducted by pulling images from Docker Hub and then the installed packages have been compared to databases such as the National Vulnerability Database and MITRE using the package name and the package version. In these databases lists of CVE are maintained which describes the vulnerabilities and in the case of National Vulnerability Database (NVD) the vulnerability is also assigned a score which determines if vulnerability is low, medium or high. The distribution of the OS e.g. Ubuntu, Debian is also taken into consideration due to the vulnerability may vary from different distributions. Due to images containing a large number of vulnerabilities and the design of Docker infrastructure the containers spawned from an image could be vulnerable to remote attacks and other exploits. Furthermore due to the ease of launching a large number of containers in a cloud environment it can make the vulnerabilities hard to find and greatly increases the number of vulnerabilities to manage. This problem brings the need for analyzing the Docker images in real time and also to flag a image whenever it is in need of a rebuild to properly maintain the security of systems using Docker.

3 METHOD

Due to being tied to a non-disclosure agreement the source code for the application is not shared here. Data input that is specific to internal systems that belong to Outpost24 will also not be displayed. The remaining methods will however be described in detail so the functionality of the application can be understood.

3.1 Docker Engine

Docker is a platform for developing and running applications. The applications are developed as images and then distributed on Docker Hub where users can download and deploy the applications. The images are run as containers which is a lightweight virtual machine where the instances are run directly on the host OS by utilizing the Docker engine which has a base OS kernel, each container then uses this kernel to run their own libraries and binary files. The containers are isolated by namespaces and control groups rather than emulating hardware and running a guest OS like a virtual machine, the difference of a VM and a container is visualized in figure 3.1. The containers and images can be managed by using a command line interface and be used for e.g. downloading images, starting containers or removing containers[10]. For the back end of the application developed Docker was chosen. Its abilities to download and extract files from images is used as a base for extracting package data and as well for extracting info about the images repository.

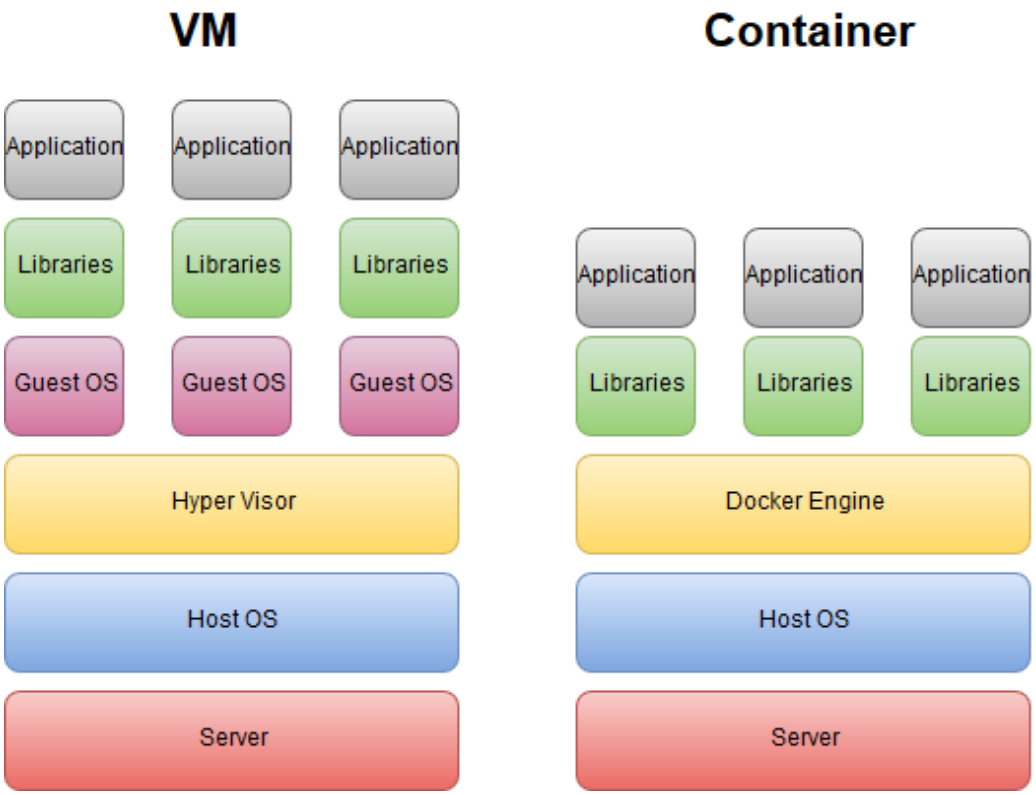

Figure 3.1: The difference between a virtual machine and container

3.1.1 Docker API

Docker provides an Application Programming Interface (API) for communicating with the installed version of Docker and several Standard Development Kit (SDK) are also available in many different programming languages[11]. For the application that was developed the Python SDK was chosen due to Python being a simple yet powerful language[12]. The API provides several different functions that integrates with the Docker engine which is used from the SDK as a back end for the application. Images can be pulled down by using the API call *pull* which takes the name of an image and then downloads a image if it exists. The API call *save* takes a image object and converts it to a archive file. Lastly the low level API provides the function *images* which provides information about images such as name, ID and creation timestamps, this function is used to retrieve the information needed to form a report for the scanned images. There are several more functions in the API that allows for control of containers and network but these are not relevant for the application developed.

3.1.2 CVE and NVD

CVE is a dictionary of common vulnerabilities[13]. A CVE is identified with a unique identifier and has a standardized description for each ID. Furthermore the CVE has its severity rated using the Common Vulnerability Scoring System (CVSS). Using CVE can help with evaluating computer systems and identifying security flaws. NVD is a database that is synchronized with the CVE dictionary[14]. NVD provides additional analysis for each vulnerability. NVD updates as soon as a new CVE is published so that it always updated. In the developed application CVE was used in order to detect what vulnerabilities that were present in Docker images.

3.1.3 Image layers

An image in Docker consists of one or more layers[15]. When a image is first created it has one layer and if an modification to the image is made a new layer is created containing the changes relevant to the previous layer. This provides the ability to see the history of the changes made to an image. Each layer has a file containing metadata such as user, hostname and id. Each layer except for the base layer also has a parent layer which creates a hierarchical structure of layers. The metadata of the image itself also contains information on which layer is the latest making it possible to determine the correct order of the layers. This is important when extracting the data due to if a older layer is extracted this may result in a newer file being overwritten by an older version.

3.2 Extracting image data

When using the API to save the file system of an image the result is saved as a tar archive file. Contained in this file are the image layers in their own tar archives which each contains files that build up the file system of the container. For extracting the archive files a built-in module for Python called TarFile was used, which held the functionality of extracting the archive to a file object rather than writing the extracted files to the storage. From the file object only the files required for the scan were extracted in the application created, something that minimized the storage requirements for the software.

3.3 Data collection

The data that was first was collected was the types of OS distributions that exists among the images. To achieve this the OS release files contained in the images was read by a script which determined what type of distribution that was used. The data set used for this analysis was the top 1000 most downloaded images on Docker Hub. The list of images was acquired by using the Docker Hub website where all images could be sorted by most downloads. The result of this was determining the data set which was used for the scanning.

The data set used for the scanning was determined by three criteria. First was the capabilities of the scanner and which Linux distributions it supported. The other two were the frequency of the different Linux distributions among the top 1000 downloaded images and if the *latest* tag was existing in the image. The reason behind only including the *latest* tag was the restrictions that the Docker SDK had in regards to retrieving the tag of the latest build.

Other data that was collected was the severity level of each CVE that was detected. This was achieved by using a external program *cve-search*[16] which can take a CVE ID from the result of a scan and then return the detailed information. A small bash script was created to go through each report file and for each CVE found run it against *cve-search* and take the severity level which is measured by a value between 1.0 and 10.0 where higher value means a more severe vulnerability. Based on the rules for severity level in the NVD, the severity levels found were then classified as low, medium, high or critical and then the amount of each level were counted so that statistics of vulnerabilities in Docker images could be constructed. Another script was also created to sort the severity levels by OS distribution so statistics of the different OS distributions could be achieved.

3.4 Extracting Package information

From the data collected it could be seen that the dominant distributions were Debian, Ubuntu and Alpine Linux, and because of this support for these distributions were prioritized. Support for the Red Hat-based distribution CentOS was also included due to the reason that the method for all Red Hat distributions is the same, and by including CentOS support for all other Red Hat distributions is achieved. Other types of distributions were found but support for these were not included. To scan for vulnerabilities the scanner required a list of the installed packages in a specific format. The list of packages is usually generated by using the package manager, however in this case the analysis was preformed on a image that was not yet started as a container resulting in that executable files like the package managers from the images could not be used. An alternative method designed for static analysis of installed packages was therefore needed to be created. The method for extracting the installed packages was different in each type of distribution.

3.4.1 Dpkg

Dpkg is the standard package manager for Debian and Ubuntu. On a live system the way to extract the installed packages would be to execute following into a command line.

dpkg-query -W -f='${Package} |${Source} |${Version} |${Status}'

However *dpkg-query* is a executable file and can not be used when statically extracting the data from a image. To overcome this problem the method used was to parse the file *status* contained

in */var/lib/dpkg*. this file contains the information about installed packages such as name, version and source, and by parsing it the same output was generated in order to make a scanner accept the list as input.

3.4.2 RPM

RPM Package Manager is the standard package manager for Red Hat distributions, CentOS included. On a live system the way of listing installed packages would be

rpm -qa –qf '%{NAME} %{EPOCH}:%{VERSION}-%{RELEASE}'

Reading the package list from a static image was in this case more complex. RPM stores the installed packages in a Berkeley DB database which is a non relational database and consists of several files. To read the RPM database the Python API *rpm-python3* was chosen. The API requires the location of the extracted RPM files that will be read. The database then needs to be rebuilt using the API command *rebuild()*, this is because when the files are extracted from an image to another OS there will be conflicts with the database version and the rebuild command will solve the conflicts without altering the data. After the database is rebuilt it can be opened and then it is possible to iterate over each entry in the database and extract the data of each entry to reconstruct the output needed. Each entry is read to a dict object for this entries in this dict *name, epoch,version and release* is read to form the same output as the package manager. When all objects have been read the database is closed and the path reset to default.

3.4.3 apk

Alpine Linux package management is the standard package manager for Alpine Linux. On a live system the way to list installed packages is to use the command

apk info -v

To extract the data from a static image a similar method to extract packages for dpkg was used. The file *installed* contained in */lib/apk/db* contains a list of all installed packages. By parsing it the package names and versions could be extracted to create the same output as the command, the data used in *installed* was the field *P:* which states the name of the package, the second field used was *V:* which states the version of the package. These fields were combined for each package listed in the file to match the output from the package manager. Alpine Linux also required additional information about the architecture used on the OS this was achieved by extracting the file *arch* located in */etc/apk/arch*. This file identifies the CPU architecture used on the OS e.g. X86_64.

3.4.4 Operating system version

In addition to the list of installed packages the scanner also needed to know what Linux distribution and OS version the Docker images used. This also required a method for detecting the OS on a static image. The version of a Linux system can usually be detected by reading the *os-release* file, located in */etc*. This file contains information such as the name of the distribution, the version number and code name for the release. For Red Hat-based distributions there is a file called *redhat-release* which indicates which type of distribution is used, e.g. CentOS. Debian and Ubuntu may also have a file called *lsb-release* which contains information similar to *os-release*. All these files contains the information necessary to identify the OS distribution so during the

extraction phase of the application these files were the ones to be extracted if they existed. For the cases where none of the files for detecting the OS were available the image was not included in the scan and also excluded from the statistics. For Debian and Ubuntu the files extracted were then parsed to only include the relevant information such as the name, version and codename for the distribution. For Alpine Linux the entire *os-release* file was read and for CentOS only the *redhat-release* file was read.

3.5 Scanning for vulnerabilities

Several different scanners were examined in order to choose one that worked well with the application. First OpenSCAP[17] was considered as it had support for Docker images and containers, however this was only available on Red Hat distributions. Another scanner considered was Vulners[18] which is an online search engine that takes a package list and returns a list of vulnerabilities if any is found. It does have a API for scanning, however this did not have support for Alpine Linux which is one of the major OS distributions used for Docker images. In the end the scanner used at Outpost24 was chosen. The scanner is a tool used internally on the company's network which requires that a JSON file is built using a specific structure and contains the information needed. This is then sent to the scanner which then returns a report containing a list of the found vulnerabilities in the form of CVE identifiers as well as findings according to an internal system. In this research the CVE ID that was returned from the scanner was used to determine vulnerabilities.

3.5.1 JSON scanning object

In order to scan packages for vulnerabilities the scanner first needs to know what distribution the OS has. The type of the distribution along with name and version was identified with the *os-release* file. The scanner also needed the list of installed packages and the command that is used to generate the package list. This data was then used to construct a JSON file necessary to run against the scanner. The JSON files were then sent to the scanner which matched the packages against Outpost24's internal database. After the scan is complete a report is generated containing a list of found CVE's along with the name and version of the package.

3.5.2 Report from scan

The report from the scan is a list containing what package, version and what vulnerabilities that was found for that package. This is not enough for scanning of Docker images as it only refers to what OS the CVE was found on. In this case it is also preferable to have a report that states what Docker image that the CVE's where found on as the OS can be the same for several images. To achieve this the Docker API was used to get information about each image. The information that was used to identify the image is the image name, the image ID and the time stamp that indicates when the image was last updated on Docker Hub. All of this data was then assembled into a new JSON file. The reason for choosing JSON format for the final report is because it becomes easier to read and it is possible to parse it, making it easier to integrate the application with other systems.

3.6 Program design

The application consists of several different modules which then are connected, visually presented in figure 3.2. The first module utilizes the Docker API to download images, getting information about images and getting the images as a image object. The second module extracts the image

layers in the correct order and then extracts the files that identifies the OS and installed packages. The third module parses the extracted files in different ways depending on the OS distribution. The fourth module takes the extracted data and constructs a JSON object which is then used for scanning. The final and fifth module takes all reports and creates a JSON object containing the report and information about the image. All modules are connected to automate the process of taking a list of Docker images and then scan the installed packages on the images, and in the end generate a report for each image scanned.

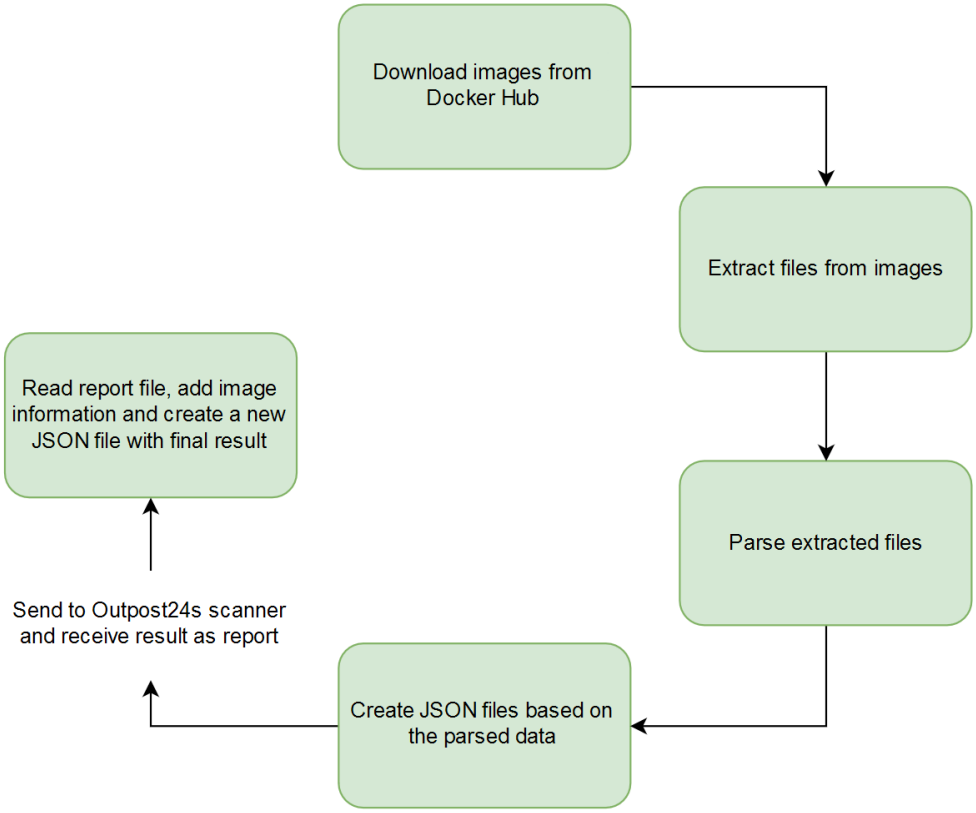

Figure 3.2: The different modules of the developed software

3.7 Validation

For validating that the application works correctly a custom image was created. This image was created with a package containing a known vulnerability installed. To install the vulnerability a Ubuntu image was launched as a container, after this a package containing an old version of OpenSSL was transferred over to the container and installed with dpkg. The file system was then extracted and built as a new image which was then run through the application. The image was then scanned using the application, after the CVE's found in the image was compared to the CVE database to confirm that the application can properly extract the vulnerabilities of a Docker image.

4 RESULTS

The results received from the method described in the previous chapter is presented below. The source code of the application will not be displayed here due to the non-disclosure agreement, however the results from using the application is displayed.

4.1 Images from Docker Hub

The downloading of images from Docker Hub took place 7th April 2017 and after downloading top 1000 images with the latest tag the resulting set was 906 images. This means that 94 images did not have a latest tag and were therefore excluded. When trying to download the images without a latest tag a 404 error was received, for example when trying to download the image *mesosphere/mesos-slave* the following error was caught.

```
404 Client Error:  Not Found ("No such image:  mesosphere/mesos-slave:latest")
```

When looking at the Docker Hub website it was also confirmed that this image did not have a *latest* tag.

4.2 Extracting OS distribution

When extracting the file system the application was capable to correctly find the release files. The application could also properly identify the OS distribution of the image. The cases where the release files were not present the application correctly returned a error message and skipped the remaining steps for that image. The error message caught when the release file for the image *busybox* was not found looked like following.

```
[Errno 2] No such file or directory:
'/home/ubuntu/dockertest/busybox/etc/os-release'
```

It was also confirmed that *busybox* did not have a *os-release* file by starting the container manually and searching for it.

From these statistical results we had regarding the number of different distributions among the downloaded images, we find that the most popular distributions were: Ubuntu, Alpine and Debian with a significant amount as can be seen in table 4.1. From this table there are 75 images missing, these are the ones that did not have the required files for the application to determine OS type.

Linux distribution	Number of occurrences
Ubuntu	289
Alpine	239
Debian	225
Buildroot	36
CentOS	34
Fedora	4
Oracle Linux Server	3
OpenSUSE	2
Raspbian	1
Arch Linux	1
Amazon Linux AMI	1
Total	*831*

Table 4.1: Distributions among Docker Hub images

4.3 Extracting file system

Since the distributions supported by the scanner were Ubuntu, Debian, Alpine and CentOS the set that was used included 787 images. When a unsupported distribution was found, for example Fedora, the following error was returned and the program continued its execution.

```
Fedora currently not supported
```

The required files from the included images were successfully extracted from the images, the parsing of the file successfully extracted the package information in the right format which was confirmed by starting a container of each OS type and compare the package list extracted by the application with the output from the package manager, the data was then put into JSON files. Samples were taken and checked to make sure everything was correct and the result was sent to Outpost24's scanner which successfully returned a list of CVE's if they were found in the images, in the case where a image did not contain any vulnerabilities the resulting list were empty.

4.4 Reading report files

The report files were returned and parsed together with image information to create the final JSON files that Outpost24 later may use, which was one of the goals of this work. Statistics were also calculated to explore how widespread vulnerabilities were among the different Linux distributions. Table 4.2 shows how many images that had vulnerabilities of the different severities. Ubuntu and Debian were highest in percentage while Alpine had few vulnerabilities. In table 4.3 the total number of vulnerabilities in different OS distributions are listed, with a total of over 113 000 in the 787 different images.

The *unknown* label in the tables is not part of the CVSS, but means that the CVE did not have any available score when querying the database.

Linux distribution	Low	Medium	High	Critical	Unknown
Ubuntu	70% (203)	94% (271)	94% (271)	78% (225)	71% (204)
Debian	97% (219)	97% (219)	97% (219)	83% (187)	78% (176)
CentOS	29% (10)	53% (18)	44% (15)	29% (10)	88% (30)
Alpine	7% (16)	13% (32)	18% (44)	0% (0)	10% (25)
Total	*57% (448)*	*69% (540)*	*70% (549)*	*54% (422)*	*55% (435)*

Table 4.2: Percentage and number of images with vulnerabilities in different Linux distributions

Linux distribution	Low	Medium	High	Critical	Unknown	Total
Ubuntu	6943	38 004	14 794	3412	2161	65 314
Debian	4843	28 185	8723	2888	2028	46 667
CentOS	42	421	138	56	117	774
Alpine	20	110	112	0	41	283
Total	*11 848*	*66 720*	*23 767*	*6356*	*4347*	*113 038*

Table 4.3: Number of vulnerabilities in different Linux distributions

4.5 Validation of result

After the vulnerable image was built the package manager was used to confirm that it was installed on the image. After the vulnerability scan it could be seen from the result that new vulnerabilities had appeared on the vulnerable image.

Image	Low	Medium	High	Critical	Unknown	Total
Normal Ubuntu	0	2	0	0	0	2
Vulnerable Ubuntu	3	36	8	4	4	55

Table 4.4: Comparison of a normal Ubuntu image and a vulnerable Ubuntu image

This confirmed that the scanner could successfully detect the new vulnerabilities implemented in the image. It should be noted that this only applies to packages installed with the package manager, applications installed manually from binary files can not be detected with this method.

4.6 Report file

The reports from the scan were successfully placed in a JSON object along with data from the repository. The data that was fetched from the repository was name, image id, creation time and OS type. This creates a object that is parsable which can be integrated into other systems if necessary.

5 DISCUSSION

In this thesis a method for extracting and scanning packages from Docker images has been created by utilizing information from the package managers. The results show that it is possible to extract this information without initiating a container and in the end create a report, all in reasonable time. The software that was built had the capability of handling this in all of the expected scenarios, however we see that the extraction of files from an image could be optimized. This was however not the priority when developing this product and it will be up to Outpost24 if they want to continue the development. Compared to other solutions vulnerability scanning in Docker systems is usually done on containers that are running. The approach used in this thesis was to focus on the images before they are started as containers, as we could instead extract the needed files from the image itself. Our solution also has the advantage of being able to scan a large amount of images automatically when integrated with Outpost24's scanner.

Of all 1000 image scanned 789 were of the supported OS, which meant a coverage of almost 79% of the images. This is an acceptable amount which may be built on and improved in the future. The reason for not including the *latest* tag was the uncertainty of which version being the newest from the API, especially when making the pulling of images automated where no verification was being done. The latest tag made it more clear and always verified that it was the latest build, and this tag is also expected to see more usage from the end user. The remaining absent images were not only disregarded due to them not being supported by the scanner. Some images like e.g. Busybox was very lightweight and were missing crucial files like *os-release*, making it hard to properly determine OS distribution and therefore makes it hard to properly scan them.

When looking at the shares that the different Linux distributions had on Docker Hub one could see that Ubuntu, Debian and Alpine had such a large share that if an existing tool for scanning was to be used it had to support these three distributions. Outpost24 did not have support for Alpine at the beginning of the project, however as they were seeing the benefit of utilizing their own tool in our thesis work they pushed for support that made it shortly after.

The results show that a large number of images contain vulnerabilities, with 70% of images having a high and 54% having a critical severity level. This can be compared to Banyanops study made in 2015[9] where only 37% of the images with *latest* tag that they tested had a vulnerability that was either high or critical. Debian and Ubuntu had by far the most vulnerabilities out of the scanned images, reaching close to each image having at least one vulnerability of each severity level. The reason behind this is most probably because of most images share the same base image distribution, and therefore share the vulnerabilities. Surprising here was the absence of vulnerabilities in Alpine-based images, where there actually were no critical vulnerabilities found. A highly possible explanation for this is the focus that this distribution has on lightweight and security.

The large amount of vulnerabilities is in all likelihood a consequence of poor image management, public images made by users can be several years old resulting in outdated packages, and even official images can go several weeks or even months without a update. For this reason it is also important for users to properly update the packages that comes with a image to reduce the amount of vulnerable packages. The method in this thesis have also only included scanning packages installed by the package manager, if an application or library is installed by other means this method cannot find it. To be able to find vulnerabilities in other parts of the image a binary scan

of the whole file system needs to be implemented. Furthermore the results of the validation also shows that the vulnerabilities in packages installed with the package manager can successfully be detected however because Oupost24 scanner utilizes its own rule engine which mean that the CVE extracted may vary from the results generated from other tools.

The advantage of using a scanning method to this rather than using the Docker security scanner is that instead of logging into Docker Hub and searching up an image to check the vulnerabilities one or more images can instead be scanned with the application and generate a report for the images. Docker security scanner also needs to be used online and is a paid service and can only scan the Docker repository while our application can preform the scan locally. Another advantage is that it is possible to identify vulnerabilities of a image before use. The drawback of using this method is that some vulnerabilities that does not come from the package manager is missed as discussed before.

In this thesis the results show a large amount of images which may be vulnerable to attacks, something that an attacker may use when finding targets to exploit. However, by creating awareness of the situation regarding vulnerabilities in images from Docker Hub this may result in more pressure from the public, leading to Docker increasing the security of the images they are hosting. This would in turn lead to future systems using Docker being less exploitable by attackers.

6 CONCLUSIONS

The question for this thesis was how a method for automating the extraction and vulnerability scanning of packages in a Docker image can be constructed, and also how we can present the results of a vulnerability scan of a Docker image in a format that is clear and detailed. The results show that the method can with a acceptable accuracy of images from Docker Hub automate the vulnerability scanning of several Docker images by providing a simple list of names. In addition this is done statically on images while existing methods focus on the images launched as containers. The results also show that enough information can be extracted to create an easy to read and parsable JSON object which clearly identifies the image and its vulnerabilities and makes it possible to integrate the scan result in other systems. The proposed method works well if packages installed by the package manager is the target for the vulnerability scan, should an analysis of areas other than the package manager be required other methods such as binary application scanning is a more appropriate approach.

This work is interesting for future development and research in vulnerability scanning of Docker and the different approached that may be taken to this problem. In this work the proposed solution was to make it possible by utilizing the Docker API for extracting information directly from an image which proved to be successful. In future research the presented data over vulnerabilities in different distributions may also be used as a comparison as Docker still is a relatively new technology and therefore constantly changing.

7 RECOMMENDATIONS AND FUTURE WORK

Our application can successfully detect the vulnerabilities in Docker images for a few chosen distributions. For future development of the application it would be valuable to give support for more Linux distributions to further increase the number of images included in the scan. If more distributions are to be included a method needs to be developed to detect the OS information of the images that were excluded in this work. Furthermore, vulnerabilities in other parts of the file system are not included in this method which could be extended upon. The application developed could probably be optimized in the parts when looking for files in the file system of the image. No execution time was being timed in this thesis, it could however be interesting to calculate the difference when scanning between containers and images. In the future the study of the amount of vulnerabilities in Docker images could be repeated, in this thesis we surveyed the top 1000 images and found a increase in vulnerabilities compared to Banyanops study from 2015. It would be interesting to see how the amount of vulnerabilities will change in the future.

REFERENCES

[1] Docker Inc., "Docker Hub", https://hub.docker.com [Online; accessed 29 Apr. 2017].

[2] M. Marks, "Docker Hub hits 5 billion pulls", https://blog.docker.com/2016/08/docker-hub-hits-5-billion-pulls [Online; accessed 29 Apr. 2017].

[3] Docker Inc., "Docker Security Scanning", https://docs.docker.com/docker-cloud/builds/image-scan [Online; accessed 29 Apr. 2017].

[4] A. R. Manu, J. K. Patel, S. Akhtar, V. K. Agrawal, K. N. B. S. Murthy, "A study, analysis and deep dive on cloud PAAS security in terms of Docker container security", *2016 International Conference on Circuit, Power and Computing Technologies (ICCPCT)*, pp. 1 - 13, 2016.

[5] T. Bui, "Analysis of Docker Security", *arXiv preprint arXiv:1501.02967*, 2015.

[6] T. Combe, A. Martin and R. Di Pietro, "To Docker or Not to Docker: A Security Perspective", *IEEE Cloud Computing, vol. 3, no. 5*, pp. 54-62, 2016.

[7] A. A. Mohallel, J. M. Bass and A. Dehghantaha, "Experimenting with docker: Linux container and base OS attack surfaces", *2016 International Conference on Information Society (i-Society)*, pp. 17-21, 2016.

[8] T. Combe, A. Martin and R. Di Pietro, "Containers: Vulnerability Analysis", *tech. report, Nokia Bell Labs.*, http://ricerca.mat.uniroma3.it/users/dipietro/containers_security.pdf [Online; accessed 29 Apr. 2017].

[9] G. Jayanth, T. Desikan, and Y. Turner, "Over 30% of Official Images in Docker Hub Contain High Priority Security Vulnerabilities", *BanyanOps*, 2015, https://www.banyanops.com/pdf/BanyanOps-AnalyzingDockerHub-WhitePaper.pdf [Online; accessed 29 Apr. 2017].

[10] Docker Inc., "Use the Docker command line", https://docs.docker.com/engine/reference/commandline/cli [Online; accessed 29 Apr. 2017].

[11] Docker Inc., "Docker Engine API and SDKs", https://docs.docker.com/engine/api [Online; accessed 29 Apr. 2017].

[12] Docker Inc., "Docker SDK for Python", https://docker-py.readthedocs.io/en/stable [Online; accessed 29 Apr. 2017].

[13] The MITRE Corporation, "About CVE", https://cve.mitre.org/about [Online; accessed 29 Apr. 2017].

[14] National Institute of Standards and Technology, "National Vulnerability Database", https://nvd.nist.gov [Online; accessed 29 Apr. 2017].

[15] Docker Inc., "Understand images, containers, and storage drivers", https://docs.docker.com/engine/userguide/storagedriver/imagesandcontainers [Online; accessed 29 Apr. 2017].

[16] cve-search, "Cve-search", http://cve-search.github.io/cve-search [Online; accessed 29 Apr. 2017].

[17] OpenSCAP, "Security compliance of RHEL7 Docker containers", https://www.open-scap.org/resources/documentation/security-compliance-of-rhel7-docker-containers [Online; accessed 29 Apr. 2017].

[18] Vulners, "Vulners", https://vulners.com/ [Online; accessed 29 Apr. 2017].

Blekinge Institute of Technology, Campus Gräsvik, 371 79 Karlskrona, Sweden