Student Thesis

Level: Bachelor

Docker Container Images

Concerns about available container image scanning tools and image security

Authors: Michael Andersson & Robert Hysing Berg
Supervisor: Hans Jones
Examiner: Yves Rybarczyk
Subject/main field of study: Microdata Analysis
Course code: GMI2C8
Credits: 15
Date of examination: 2022-06-01

Abstract:

With the growing use of cloud computing and need for resource effectiveness, the use of container technology has increased compared to virtual machines. This is since containers require fewer resources and are significantly faster to start up. A popular container platform is Docker which lets users manage and run containers. The containers are run from images that can be downloaded from different sources, Docker Hub being a popular choice. Because of container technology sharing the OS-kernel with the host, there is a great need to increase and monitor the security of containers and the images they are run from. To find vulnerabilities in images, there are image scanning tools available. In this dissertation, we study 5 different image scanning tools and their performance. Twenty-five random images were selected from popular images on Docker Hub and were then scanned for vulnerabilities with the tools in the study. We aimed to answer the following questions: (1) Are there any clear differences between the number of vulnerabilities found by different image vulnerability scanning tools? (2) Are there any differences between the types of vulnerabilities found by different image vulnerability scanning tools? (3) What is the relative effectiveness of different image vulnerability scanning tools? The results show that there are considerable differences between different container image scanning tools regarding the number of found vulnerabilities. We also found that there were differences regarding the severity-grading of found vulnerabilities between the tested tools. When using our proposed metric for calculation of relative effectiveness, we discovered that the tool with the highest relative effectiveness could still miss approximately 39 percent of the vulnerabilities in images. The tool with the lowest relative effectiveness could miss approximately 77 percent of the vulnerabilities in images.

Keywords:

Docker, Image, Container, Security, Scanning, Tools

Contents

1 Introduction

1.1 Background

In our modern society, computers play an increasingly important role in almost every aspect of our lives, whether directly or indirectly. This has led to a need for scalability of the services they can provide, and cloud computing is one of the answers to this need.

With the growing use of cloud computing, the use of virtual machines has grown with it since it provides a way to run several different operating systems on a single device. However, virtual machines use a lot of resources from their hosts and thus there is a limit to how many virtual machines can run on a single host (Ahmad, Dimitriou & Sultan, 2019). This is where the use of container technology come into the picture since containers require fewer resources compared to virtual machines while still providing isolation.

The growing use of containers has led to an increased need regarding the examination of security related issues around them. It is of particular interest that containers are more tightly integrated with the OS of the host compared to virtual machines. Due to this there is a risk that sensitive data from other containers or the host system itself could be accessed if a container is compromised (Martin, Raponi, Combe & Di Pietro, 2018).

In Reeves, Tian, Bianchi & Celik (2021), there are examples of different Linux kernel mechanisms that can be used to increase the security when running containers. Javed & Toor (2021a, 2021b) examines different tools that can be used to scan java-based container images for vulnerabilities. The authors then conclude that the available tools were inadequate regarding how accurately they could find vulnerabilities in Docker container images.

The usage of packages in container images hosted on Docker Hub was analysed in Zerouali, Mens & De Roover (2021). Their conclusion was that many container images were missing important security updates to their packages. Lin, Nadi & Khazaei (2020) collected most of the container images that were available on Docker Hub at the time. One of the properties they measured was the popularity of different programming languages.

There are multiple signs that the use of container technology and Docker is increasing in popularity. For example, Matt Carter writes in Matt (October 7, 2021) that the number of Docker Desktop installations has risen from 3.3 million in February 2021 to 4.7 million in early October 2021. Another sign is that when searching on the internet with terms such as “most popular container platforms 2022”, Docker is consistently mentioned when found articles are compared, even if the placement varies (Hiter, February 25, 2022; Aqua Security, n.d.; Software Testing Help, April 4, 2022; Jenna, 2021).

When talking about container images and containers, there is a slight risk of confusion regarding these terms and their use. A container image (in the rest of the study referred to as image) is a file that contains the source code, libraries, dependencies, and tools required to run an application. The image file itself consists of several layers, where each layer builds upon the previous. A container on the other hand can be considered as a read-write copy of an image with a container layer at the top, which allows for modification. In essence, images and containers are in much the same thing. The difference being that the image can be seen as an immutable snapshot of an application and its virtual environment, while a container is created/run with an image as a blueprint, while adding read-write possibilities (PhoenixNAP, October 31, 2019).

1.2 Problem Formulation and Aim

Much of the previous research on containers, images, and scanning tools has focused on the different mechanisms that can be used to harden a container or measuring the accuracy of image scanning tools. Another noteworthy part about previous work is that some have had narrow choices of images in their evaluation of container scanning tools. An example of this is seen in Javed & Toor (2021b) where only java-based images are evaluated which could have led to biased results. Due to this, it is of interest to perform a broader evaluation of image scanning tools with a wider variety of images.

Furthermore, after investigating what tools are available, it was discovered that some of the tools tested in previous work is no longer available or supported. This also shows that this scene is in constant change since some of the tools mentioned in Javed & Toor (2021a, 2021b) are no longer supported or developed. Since some tools are no longer supported or developed, a newer overview on which tools are available and a comparison between them is of interest.

The availability and functionality of tools for evaluating potential vulnerabilities of Docker images and containers are an important part of their secure usage. This leads to a need regarding an updated evaluation of the current state of available tools and their functionality. At the same time the current state of popular images available from Docker Hub regarding security and vulnerabilities should be studied since Docker Hub is an active and trusted source of Docker images.

From the above-mentioned points of interests, the following questions were formulated for this study to focus on:
1. Are there any clear differences between the number of vulnerabilities found by different image vulnerability scanning tools?
2. Are there any differences between the types of vulnerabilities found by different image vulnerability scanning tools?
3. What is the relative effectiveness of different image vulnerability scanning tools?

The aim of the study is to investigate different image vulnerability scanning tools to explore differences in results and effectiveness between the tools.

1.3 Delimitations

While there exists many more platforms for containers, we decided to only look at the Docker platform due to its popularity and to limit the scope. Likewise, only container image scanning tools capable of scanning Docker images were selected. There are also multiple container image scanning tools available and to limit the scope commercial tools were excluded from our study because of limited funding. This was decided since the trial versions often are simplified, or miss full functionality, and therefore are not suited for testing and comparing the ability of the tool. Among the free-to-use tools available, there are differences in the scanning performed by the tools. In this study we have focused on the tools that perform a static scan on a Docker image rather than the dynamic scan or monitoring of a container.

2 Theory

2.1 Virtual Machines and Containers

Virtual machines, commonly abbreviated as VM:s, are a simulation of physical hardware that can be run on any host machine that have the required software. The software that creates and runs virtual machines is called a hypervisor and it allows the user to run multiple virtual machines on a single host machine, (VMware, n.d.). Each virtual machine contains its own full copy of an operating system and various other memory allocations for libraries, applications, and storage (Docker, n.d.).

According to Docker (n.d.), the key difference between containers and virtual machines is that containers are abstracted at the application layer rather than the hardware level. Just like with virtual machines, there can be multiple containers running on the same host machine. Though there is a major difference, containers that run on the same host share the operating system kernel with other containers and the host instead of each container having their own (Wenhao & Zheng, 2020). The lack of individual operating system kernels leads to a significant decrease in resources needed for running multiple containers compared to running multiple virtual machines. This also leads to, as mentioned in Ahmad, Dimitriou & Sultan (2019), that containers are significantly faster to start up than a virtual machine equivalent.

There is another consequence of the containers and the host system sharing the kernel, the isolation between them is much weaker when compared to a virtual machine (Martin et al., 2018). Due to this, there are concerns regarding the security of containers.

The difference between Containerized Applications and Virtual Machines can be seen in *figure 1*. This also highlights how the hypervisor manages the hardware and VM:s with their own OS:s, as opposed to the same host OS kernel being used by Docker Containers.

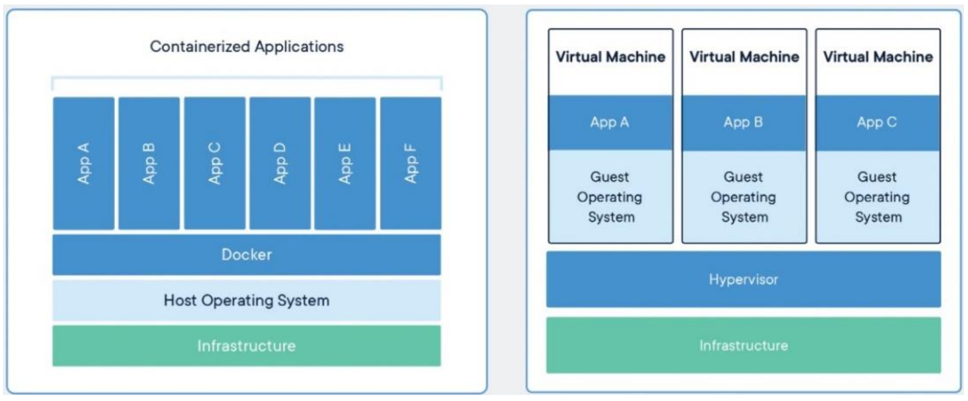


Figure 1 – Comparison between Containerized Applications and Virtual Machines (Docker, n.d.).

2.2 Docker

Docker is a widely used container manager and is one of the most popular container manager technologies with 7,3 million accounts registered with Docker (Docker Newsroom, n.d.). There are many parts of Docker such as the Docker Engine which is the core part of Docker and is what allows a developer to build and run containers (Docker docs, n.d.a). There is also Docker Hub which allows developers to share their container images with each other and a wider audience (Docker docs, n.d.b). In Linux, an image can be pulled from Docker Hub with the command "docker pull <image name>:<tag>" if the host has a network connection and the Docker Engine installed. One or more containers can then be run from the downloaded image. This image file is where all the code, configuration mapping and packages needed to run an application is contained (Mullinix, Konomi, Townsend & Parizi, 2020). This also means that there lies a great responsibility with the developer of the image to follow best practices and guidelines when building the image to increase security and avoid misconfiguration.

2.2.1 Image Tagging

Docker Hub uses tags to identify different versions of images in the form of "<image name>:<tag>", for example "alpine:3.15.4". Each version of an image also has a digest to uniquely identify that image and the possibility to ensure the integrity of the image. When using the command "-docker pull <image name>:<tag>", the tag identifies which version of the image is to be pulled from Docker Hub. If no tag is given, the default tag of "latest" will be used. The "latest" tag does not stick to the same version of the image as the other tags, but instead can be set by the author to point at any version of the image. Usually the "latest" tag points to the latest version of an image. Tagging is of importance when the user must use a specific version of an image, but also when testing an image or scanning tools because of the ability to trace versions and their capabilities or results.

2.2.2 Docker Compose

Docker Compose is a tool that uses a YAML file to define services to be used when starting a container (Docker docs, n.d.d). Docker Compose can also be used to harden a container and increase the security by setting up different limitations for the container in the YAML file.

2.3 Vulnerabilities

A vulnerability is, according to NIST (n.d.), a

"w*eakness in an information system, system security procedures, internal controls, or implementation that could be exploited or triggered by a threat source*".

In this study we use the term vulnerability to describe a registered CVE, bug, or other weakness in an image or container that could be exploited by a threat actor or give rise to unwanted behaviour that could compromise the security of the image or container.

2.3.1 CVE
CVE stand for Common Vulnerabilities and Exposures. According to The MITRE Corporation (n.d.), it is a system to identify, define, and catalogue publicly disclosed cybersecurity vulnerabilities. The system is maintained by The Mitre Corporation and the CVE ID:s are listed in Mitre's system and in the US National Vulnerability Database (NVD). Each CVE has a unique name, starting with "CVE", followed by the year of registration and a number. An example of a CVE would be CVE-2014-0160, which is the "Heartbleed" vulnerability. Each severity should also be graded based upon a CVSS-score.

2.3.2 CVSS
CVSS stand for Common Vulnerability Scoring System. This provides a way to produce a numerical score reflecting the severity of a vulnerability. The numerical score can then be translated into a representation of the severity, such as low, medium, high, or critical (Forum of Incident Response and Security Teams, n.d.).

2.3.3 Vulnerable Images on Docker Hub
In a study from 2020, cyber security firm Prevasio investigated container images on Docker Hub (Prevasio, 2020). They found that out of approximately 4 million images scanned, more than half of the containers from the images had one or more critical vulnerabilities. Their analysis also showed that 0.16 percent, or 6432 images, of the total images scanned, were malicious or potentially harmful. Their study was done by performing dynamic analysis on the behaviour of the containers and scanning the files of running containers. According to Prevasio (2020), not all vulnerabilities or harmful behaviour could be found or identified unless a container was started from an image. They found that some images, when run as a container, could download and execute malicious code at runtime. Each container was run and analysed in an isolated environment.

2.4 Container Security Mechanisms

Since the containers can communicate directly with the kernel of the host, there is a risk of an attacker performing an escape attack in order to escape the container and compromise the host (Martin et al., 2018). If an attacker manages to escape a container there is the potential that a privilege escalation or DoS attack can be executed. Even if an attacker is not able to escape the container, there is a risk that an attacker, by exploiting a bug or misconfiguration, can run arbitrary code outside of the container (Reeves et al., 2021). Running arbitrary code outside of the container, from within the container, effectively means that the attacker in a way has escaped the container. Thus, there is a need for security mechanisms to secure a container and host.

Because the use of Docker and containers are tightly coupled with the OS kernel of the host, many of the mechanisms for increased security have to do with different settings regarding the Linux kernel. These mechanisms are mainly set up and utilized when starting/running a container from an image. The different mechanisms will be explained in the next sections regarding isolation and increasing defences.

2.4.1 Isolation
In Reeves et al. (2021), the authors highlight two mechanisms of the Linux kernel that provides an increased security by isolating a container and limiting the resources that can be used by a container. The two mechanisms are namespaces and cgroups.

Namespaces
Namespaces works by confining processes into different groups to limit access from the container. There are eight types of namespaces available in Linux (Man7.org, August 27, 2021c):

- Cgroup
- IPC
- Network
- Mount
- PID
- Time
- User
- UTS

If a container process is, for example, put into its own PID namespace, it can only see other processes that share the same PID namespace (Reeves et al., 2021). This can be used to isolate the processes of different containers from each other and from the host. There is also the possibility to use the User namespace to map the root ID of a container to a regular user on the host, which, according to Reeves et al. (2021), can prevent a malicious process that has gained root access within a container to gain root permissions on the host. According to the authors, this, for example, effectively prevents a vulnerability called runC CVE-2019-5736 from executing.

Cgroups
Control groups are referred to as cgroups. In the Linux kernel this is a feature that allow processes to be organized in a hierarchical manner into groups. The groups usage of different resources can then be limited and monitored (Man7.org, August 27, 2021b). According to Reeves, Tian, Bianchi, & Celik (2021), the three main cgroups for containers are CPU, memory, and I/O. Using these cgroups, the amount of CPU load, memory, and I/O operations can be limited and monitored.

2.4.2 Increasing Defences
There are also, according to Reeves et al. (2021), three security mechanisms in the Linux kernel that a host can use to secure/defend itself against malicious execution from containers. These are Mandatory Access Control (MAC), Seccomp, and Capabilities.

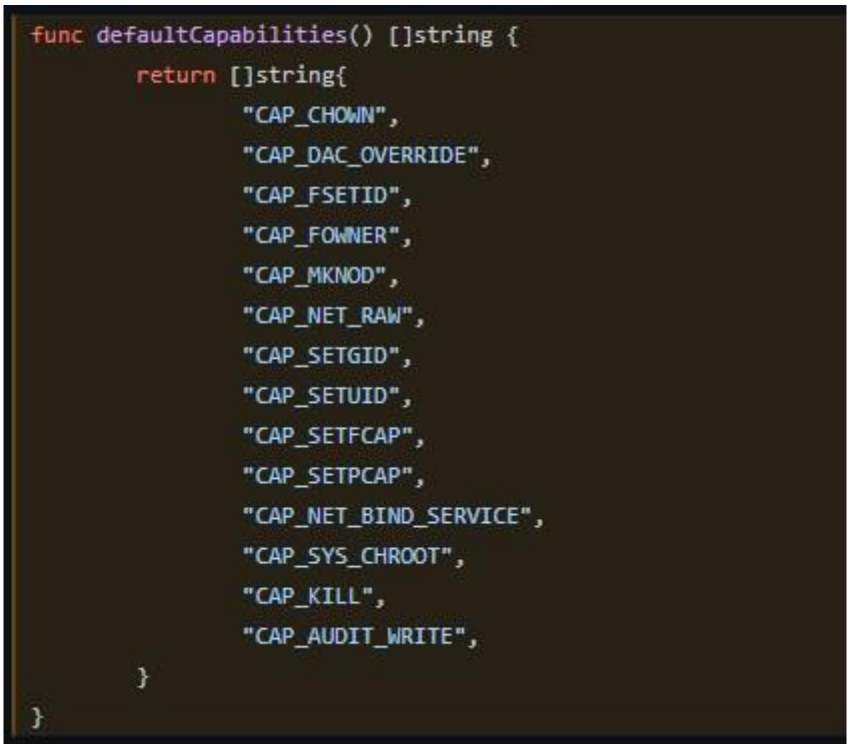
MAC
Mandatory Access Control is a form of policy frameworks that can be implemented as Linux Security Modules (LSM:s). Every action gets tested against the rules in the policy framework, and if the action passes the rules in the policy framework it is allowed, otherwise the action is not allowed (Reeves et al., 2021). According to the authors, examples of policy frameworks are SELinux and AppArmor.

Seccomp

Seccomp stands for secure computing and is a feature to restrict the calls a process can make to the kernel (Reeves et al., 2021 & Man7.org, August 27, 2021d). According to Docker docs (n.d.c), seccomp has to be configured in the kernel and Docker has to be built with seccomp. A seccomp profile can then be created as a json-file which in short is an allow list that denies or allows system calls. When seccomp is activated, there is a default profile loaded when a container is run, but the default profile can be overrun with a user-created profile. According to Docker docs (n.d.c), seccomp is paramount for running containers with least privilege.

Capabilities

Capabilities is a form of separation of the Linux root permissions into separate units that can be used to set a more precise permission model regarding constraints for what a user can do (Reeves et al., 2021). According to Man7.org (August 27, 2021a), there are 41 capabilities implemented in Linux. There are 14 default capabilities granted when building an image (Moby, 2022), and according to Reeves et al. (2021), these capabilities should be managed in order to reduce the attack surface. The capabilities granted by default can be seen in *figure 2*.



```
func defaultCapabilities() []string {
        return []string{
                "CAP_CHOWN",
                "CAP_DAC_OVERRIDE",
                "CAP_FSETID",
                "CAP_FOWNER",
                "CAP_MKNOD",
                "CAP_NET_RAW",
                "CAP_SETGID",
                "CAP_SETUID",
                "CAP_SETFCAP",
                "CAP_SETPCAP",
                "CAP_NET_BIND_SERVICE",
                "CAP_SYS_CHROOT",
                "CAP_KILL",
                "CAP_AUDIT_WRITE",
        }
}
```

Figure 2 – Capabilities granted to an image by default (Moby, 2022*).*

2.5 Tools

A simple search on Google with the term “Docker image scanner” leads to a multitude of different tools for scanning images and containers. There are tools for static scanning of images alongside more dynamic tools for monitoring running containers. Some of the more popular tools can be seen below with a short description of the tool. The tools listed in this section are the tools selected for the study based on the tool selection criteria mentioned in section 3.2.2 Selection of Tools.

Javed & Toor (2021a), studied different tools for vulnerability detection regarding OS and non-OS packages in images. The tools studied was Clair, Anchore, and Microscanner. Microscanner is now deprecated and Trivy seems to have replaced Microscanner, whereby we choose to include Trivy in our study (Aqua Security, 2022). Other popular scanning tools recommended by various instances are Dagda and Grype. Finally, there is Dockers own scanning tool called Docker scan, which is available through Docker Desktop and Docker CLI.

All image scanning tools have in common that they scan the images for already known vulnerabilities. This is because they compare the packages in the image against different vulnerability databases where vulnerable packages, versions, and related vulnerabilities are stored. Some tools are capable of also examining binaries in the images to be able to find vulnerabilities if regular package managers are not used in the images. There are also tools that utilize antivirus technology to be able to scan for viruses and trojans within the images. Some of the tools had information available regarding the database(s) used in the scanning process and where the tool got its information from. All the tools mentioned in this section could scan both OS and non-OS packages, whereby the results of the tests should not differ considerably between the tools. The different tools and their capabilities/traits can be seen in *table 1*.

Tool	**OS packages**	**Non-OS packages**	**AntiVirus engine**	**Database Information**
Docker Scan	X	X		
Trivy	X	X		X
Anchore	X	X		
Grype	X	X		X
Dagda	X	X	X	X
Clair	X	X		X

Table 1 – Image scanning tools and their capabilities.
OS packages: If the tool can scan OS packages for vulnerabilities.
Non-OS packages: If the tool can scan non-OS packages, i.e., dependencies or other packages, for vulnerabilities.
AntiVirus engine: If the tool has an antivirus engine that can scan for malicious code inside an image.
Database Information: If there is information available regarding where the tool’s database collects information.

When inspecting *table 1* above and the descriptions of the tools below, there seems to be a slight edge in favour of Dagda. Dagda has the most features, with even an antivirus engine available, to be able to find a high number and diverse set of vulnerabilities when scanning images. There is also a wide range of sources for the vulnerability database presented. This leads to Dagda being the most promising tool in advance of the actual tests.

Docker Scan (Snyk)

Docker scan is a Docker official free-to-use scanning tool for local images and can detect vulnerabilities in OS and non-OS packages. This tool runs on Snyk engine to perform vulnerability scanning of images. Snyk is an open-source free-to-use scanning tool/engine. It uses the Snyk Vulnerability Database, but it is unclear where the database gets the data from.

Trivy

Trivy is an open-source free-to-use image scanner that can detect vulnerabilities in OS and non-OS packages. Trivy can also scan for example Dockerfiles for configuration issues. The tool can also scan for hardcoded passwords, API keys, or tokens. It uses the Aqua Vulnerability Database that collects data from NVD, software vendor advisories, and Kube-Hunter.

Anchore

Anchore Engine is an open-source free-to-use image scanner that can identify vulnerabilities, malware, misconfigurations, and secrets. It can detect vulnerabilities in OS and non-OS packages. The vulnerability database draws data from the National Vulnerability Database, but no further information could be retrieved.

Grype

Grype is an open-source free-to-use tool made by Anchore to scan images for vulnerabilities. It can detect vulnerabilities in OS and non-OS packages. The vulnerability database draws data from Alpine Linux SecDB, Amazon Linux ALAS, RedHat RHSAs, Debian Linux CVE Tracker, Github GHSAs, National Vulnerability Database, Oracle Linux OVAL, RedHat Linux Security Data, Suse Linux OVAL, and Ubuntu Linux Security.

Dagda

Dagda is an open-source free-to-use scanning tool for images. It performs a static analysis of images to identify vulnerabilities, trojans, viruses, malware, and other malicious threats. Dagda can detect vulnerabilities in OS and non-OS packages. It can also scan containers and monitor running containers and Docker daemon for anomalous activities. Dagda creates a MongoDB instance to which known vulnerabilities (CVE:s), Bugtraq ID:s, Red Hat Security Advisories, and Red Hat Bug Advisories, and known exploits from Offensive Security Database are imported. This database is then used when a scan is performed/active. Dagda also uses ClamAV antivirus engine to detect trojans, viruses, malware, and other malicious threats.

Clair

Clair is an open-source free-to-use scanning tool. It scans images layer by layer to provide a notification of vulnerabilities that may be a threat. Clair can detect vulnerabilities in OS and non-OS packages. Vulnerabilities found are based on the CVE database and similar databases from Red Hat, Ubuntu, and Debian. Clair was supposed to be part of our study, but we could not get Clair to work properly in a reasonable timeframe and therefore had to exclude this tool from our study.

2.6 Mitigation

When studying the security of images/containers and available tools for examining images, there arises the question regarding mitigation of risks and the possibility to harden the images/containers. In Yasrab (2021), the author points out the fact that containers directly communicate with the host kernel, which can lead to an attacker having direct access to the host kernel. According to the author there is the possibility to execute SELinux or AppArmor while running Docker Engine, which should greatly reduce the attack surface. Other suggested actions are to use trusted images and to make use of the security mechanisms built into the Linux kernel (Yasrab, 2021).

According to Yasrab (2021) and Shen & Yu (2020), increasing the security of a container can be divided into two parts. The first part is to ensure that the image is secure and trusted when downloading it from the source or creating your own, but also before actual usage. The second part is to secure/harden the container when it is run from the image. Most of the Linux kernel security mechanisms is only possible to set up when the container is started, why it is an important step that, according to us, can easily be forgotten or overlooked by users new to the usage of container technology.

3 Method/Methodology

3.1 Methodology

This study was based on an explorative method with a quantitative approach regarding data collection and analysis.

The specific data collected using the scanning tools were total number of vulnerabilities found, severity, CVE and the date of discovery/registration. These would then be analysed to determine any differences between the vulnerability scanning tools. A more thorough explanation regarding the steps taken to collect this data is explained in section 3.3 Testing Process.

This data was also useful when considering the analysis of the current state of vulnerabilities present within popular images from Docker Hub. The number of vulnerabilities as well as the distribution of vulnerabilities could be investigated to get an overview of the vulnerabilities on Docker Hub at the time of data collection. It could also be determined which vulnerability severity is most common and how fast different severity vulnerabilities get fixed on average by analysing the vulnerabilities date of discovery.

3.2 Prerequisites

In this section we list the software and hardware that was used during the tests of different tools for scanning Docker images. There is also an explanation regarding the selection of tools to test and the selection of the images to use the tools on.

3.2.1 Host Software & Hardware

Virtualization

For testing purposes, we choose to set up virtual machines where Docker images and tools could be tested in a safe sandbox environment. There are several different tools available for virtualization, but since our university can offer VMWare for free and it is a well-documented and tested solution for virtualization, VMWare was chosen as the platform for running virtual machines.

Host

Ubuntu 20.04.4 LTS OR Ubuntu 21.10
Kernel: 5.13.0-40-generic
CPU: 2 x Intel(R) Core(TM) i5-8300H CPU @ 2.30GHz
Memory: 6GB OR 8GB

Docker software

Docker version 20.10.7, build 20.10.7-0ubuntu5~20.04.2
Docker-compose version 1.25.0, build unknown

3.2.2 Selection of Tools

The tools selected for evaluation and testing with the images were selected based on the criteria that they needed to be free of charge, popular, and, preferably, had been used in an earlier study. The free of charge criteria was decided since trial versions of paid tools often do not contain the fully functional tool and therefore is not suitable for comparison with a fully functional tool. When selecting popular tools, we aimed at including tools that were in active use and therefore regularly updated and maintained. The criteria where the tools preferably have been tested in an earlier study was chosen because of the possibility to compare the findings in this study with that of an earlier study of the tool.

As can be seen in section 2.5 Tools, Anchore had been used in earlier studies and was therefore included in our study. Another tool name Microscanner, that had been used in earlier studies, had been replaced by Trivy, why Trivy was chosen to be included in our study. Dagda had been mentioned in earlier studies but was not tested there since it suffered from inaccuracy regarding vulnerability detection (Javed & Toor, 2021a). Since Dagda provided the most features according to section 2.5 Tools, and there had been some updates to the tool, we chose to include Dagda in our study. When looking into Anchore, we found that there was another tool available from Anchore by the name Grype (Grype, 2022). This tool was included in our study because it was released by the same company (Anchore) that released Anchore and therefore it would be interesting to see if the performance was similar. Prior to this study, Docker had announced that their own image scanning tool by the name Docker Scan now worked with Docker CLI. Since this tool is released and supported by Docker themselves, it was of interest to see how it performed against other tools. Therefore, Docker Scan was included in our study.

Although the results from our experiments were analysed and assessed based upon metrics that were not necessarily the same as in earlier studies, there was still some elements that could be compared. For example, the accuracy or effectiveness of the tools could be compared to what earlier studies have discovered.

3.2.3 Selection of Images

When selecting images, we looked at both official and verified images (verified publisher) at Docker Hub. The verified images were all from a publisher verified by Docker. The images chosen to be used in this study were subject to the following criteria:

- The image had to be popular, which in this case meant having at least 1 000 000 downloads from Docker Hub. This was because popular images with a high number of downloads are the ones with the highest probability of being used.
- There had to be a mix between official and verified images in the sample because not all recommended images are Docker official images.

For our study we selected 25 random images from the first five pages of recommended images on Docker Hub. Each downloaded image was documented with image name, tag, and alias. Due to ethical considerations we chose to show the aliases of the images instead of their names in this study.

3.3 Testing Process

3.3.1 Metrics for Assessing Tools

Javed & Toor (2021b) uses a system of metrics to grade the tools that they test, and by that assess the quality of the tool. The system is based on the number of vulnerabilities found by a tool, the number of vulnerabilities missed, and a calculation between the two. This also builds on a static analysis of the images source code itself, if available, and from this a detection hit ratio is calculated. The static analysis of the source code could be one of the reasons that only java-based images were tested in that study.

In our study, there was not enough time to conduct an as extensive analysis as in Javed & Toor (2021b). Therefore, the tools used in this study was compared based on, for example, differences in number of found vulnerabilities and vulnerability severity. Furthermore, we propose a new way of investigating the accuracy/effectiveness of a scanning tool. In our study we collected data from several tools and their scans of 25 popular images from Docker Hub. By calculating the total number of unique vulnerabilities found by all tools for each image (*nTot*), and then comparing each tools number of found unique vulnerabilities for each image *(nTool)* to the total number of found unique vulnerabilities for that image *(nTot),* we could get a measurement of each tool's share of found vulnerabilities for each image *(pTool).* The effectiveness of each tool would then be the total mean value of that tools share of vulnerabilities found *(pTot)* for all images scanned *(n).* The highest possible value for relative effectiveness would be 1,0 since this would mean that all vulnerabilities found were also found by that tool. This could be an effective way of investigating the effectiveness of the scanning tools. The formula for calculating the relative effectiveness is presented below:

i = Image
n = Number of images
pTot = Share of found vulnerabilities (Relative effectiveness)
pTool = Share of found vulnerabilities by a specific tool for a specific image
pSum = Sum of Ptool for all images for a specific tool
nTool = Found vulnerabilities by a specific tool for a specific image
nTot = Total number of found unique vulnerabilities by all tools for a specific image

$$pTool = \frac{nTool}{nTot}$$

$$pSum = \sum_{i=1}^{n} pTool_i \qquad (pSum = pTool_1 + pTool_2 + \dots pTool_n)$$

$$pTot = \frac{pSum}{n}$$

An example calculation is presented below:
Tool 1 has 300, 110, 150 and 400 unique vulnerabilities found for images 1, 2, 3 and 4
Tool 2 has 200, 55, 60 and 200 unique vulnerabilities found for images 1, 2, 3 and 4
Tool 3 has 250, 500, 300 and 70 unique vulnerabilities found for images 1, 2, 3 and 4

Assuming all tools share 50 of the unique vulnerabilities per image:
Tool 1 $pSum = 1.724 \approx \frac{300}{250+150+200+50} + \frac{110}{60+5+450+50} + \frac{150}{100+10+250+50} + \frac{400}{350+150+20+50}$
Tool 2 $pSum = 0.902 \approx \frac{200}{250+150+200+50} + \frac{55}{60+5+450+50} + \frac{60}{100+10+250+50} + \frac{200}{350+150+20+50}$
Tool 3 $pSum = 2.124 \approx \frac{250}{250+150+200+50} + \frac{500}{60+5+450+50} + \frac{300}{100+10+250+50} + \frac{70}{350+150+20+50}$

This gives the final relative effectiveness for each tool as:
Tool 1 pTot ≈ 0.431 (1.724 / 3)
Tool 2 pTot ≈ 0.226 (0.902 / 3)
Tool 3 pTot ≈ 0.531 (2.124 / 3)

3.3.2 Steps for Performing Vulnerability Scans of Images

Steps taken to prepare for the process of testing:

1. Start virtual machine
2. Download and install Docker
    a. - sudo apt update
    b. - sudo apt install docker
    c. - sudo apt install docker-compose (needed for Anchore)
3. Download the set of selected images
    a. Docker pull <image name>:<tag>

The “latest” tag was not used because of the possibility that the latest tag can be linked to a new version of the image if there is a need to scan the image again later. All tags used in this study were linked to a specific version of that image.

Steps taken for each tool during the process of testing:
(The process for each tool is described in section 3.5 Data collection)

1. Download and install the tool
2. Start the scan of the image
3. Save and document the results of the scan
    a. Image name
    b. Tool name
    c. Vulnerabilities found

3.4 Ethical Considerations

Since this study was related to security flaws existent within images that are in active use, it was important to anonymize those images so to not create a guide to potential exploits. This was accomplished by selecting some, but not all, of the most popular images and calling them by pseudonyms such as alias_1 or alias_2. This might be seen as trivial since results in this study should be replicable and any would be hacker could perform the same scans. However, the process can be made slightly more tedious by not making it trivial to find exploits in specific public images by reading the results of an academic dissertation. Therefore, we chose to anonymize the images so that the results of this study could not be used to identify specific images and their vulnerabilities.

Due to these ethical considerations, the results of this study will not be future-proof and might very well lose all relevance in the coming years. Though the future-proof aspect can also be said about some previous studies since some of the tools tested there did not exist anymore at the time of this study.

3.5 Data Collection

In the following sections, the process of testing the different image scanning tools is described. The installation and the function/use of the tools is documented and explained. To be able to extract data from the reports generated from all the tools, we created a python program/tool. This would generate the desired data regarding found vulnerabilities, their distribution, and occurrence. Together with our python tool, we used Microsoft Excel to work with different Excel-sheets for presentation of the data.

Some of the tools had built-in functionality to export and save the results to a report in various formats. Since every chosen tool supported output in json format and that json is easy to work with, we chose to export the results in json format. Several programs however did not have a built-in functionality to export the results of their scans to a file, and we had to pass the generated output of the results to a file manually with the terminal functionality for passing output to a file and saving.

After the reports were saved, we used a python tool that we created to extract information from the reports and compare the results. The python tool was based on a loop where it checked all reports for a certain tool, recorded the vulnerabilities found in said report into nested dictionaries that were then saved into CSV files. The reason for the nested dictionaries was that it made it easy to create a relation of image-package-vulnerability-count&severity. These CSV files were then imported into a Microsoft Excel worksheet and further analysed so that the data could be summarized and put into tables and figures.

3.5.1 Docker Scan

Docker scan is the official vulnerability scanning tool for Docker images and comes with Docker Desktop or can be installed as an individual module for Docker. The installation instructions for Docker scan are well documented and there is a guide available in the Docker documentation for how to install it and use it.

To scan an image the following command was entered into the terminal:
- *docker scan <image_name>:<tag>*

There are additional options available for the command such as *--json* which gives you the output in json format and the command looks like this with the option:
- *docker scan --json <image_name>:<tag>*

The above command was used in this study to gather the vulnerabilities of images obtained from Docker Hub. Another feature of Docker Scan is that if an image is not locally available, it will attempt to pull it from Docker Hub if it exists there.

According to the documentation, there seemed to be no way to export the results of the scans to a file with Docker Scan. The results of the scans could be output to the terminal in json format though, and thus could be saved to a json file through the terminal with the command:
- *docker scan --json <image_name>:<tag> > ~/DockerScan/<report_name>.json*

3.5.2 Trivy

Installation was a straightforward install from Github with the command:
- *curl -sfL https://raw.githubusercontent.com/aquasecurity/trivy/main/contrib/install.sh | sudo sh -s -- -b /usr/local/bin*

To scan an image we entered the command:
- *trivy image <image_name>:<tag>*

The images were scanned quickly, and the results got printed to the terminal for the user to view. There was also a summary for each packet that is scanned with the total number of vulnerabilities found and the number of vulnerabilities for each severity (unknown, low, medium, high, critical).

To save the report from the scan in different formats, for example json or xml, we could use the commands:
- *trivy image -f json -o <report_name>.json <image_name>:<tag>*
- *trivy image -o <report_name>.xml <image_name>:<tag>*

Reports from the scans were saved in both xml and json formats. The xml format was more human readable while the json format was used as input for our report-analyser tool.

3.5.3 Grype

Grype was easy to install with the following command that they recommended at the time of testing:
- *curl -sSfL https://raw.githubusercontent.com/anchore/grype/main/install.sh | sh -s -- -b /usr/local/bin*

To scan an image with Grype, the following command was entered into the terminal:
- *grype <image_name>:<tag>*

There are also additional options such as explicitly defining some parameters, for example the image being a docker image, by preceding the "image:tag" with docker:
- *grype docker:<image_name>:<tag>*

When performing a scan of an image, the output to the terminal can be selected by using the following command:
- *grype <image_name>:<tag> -o json*

There was no documented way to export the results of the scans to a file with Grype. The results of the scans could be output to the terminal in json format though, and thus could be saved to a json file through the terminal with the command:
- *grype <image_name>:<tag> -o json > ~/Grype/<report_name>.json*

3.5.4 Dagda

The first important notice is that Dagda requires at least 6GB of memory to work. Before Dagda could be downloaded and started, there was a long list of dependencies that had to be downloaded and installed on the system. Most of the dependencies had to do with python3, but all the dependencies could be installed through “apt install”, for example:
- *sudo apt install python3-requests*

When all the dependencies had been installed, mongoDB had to be pulled from Docker Hub and started with the commands:
- *sudo docker pull mongo*
- *sudo docker run -d -p 27017:27017 mongo*

Next step was to download Dagda from Github and unzip. Starting the Dagda service was then a matter of navigating to the directory where the dagda.py file was located and start it with the command:
- *sudo python3 dagda.py start*

The rest of the process had to be done in a second terminal. Firstly, two variables had to be declared with the commands:
- *export DAGDA_HOST='127.0.0.1'*
- *export DAGDA_PORT=5000*

After this the vulnerability database hade to be initialized and updated with the command:
- *python3 dagda.py vuln –init*

The initialization and update process of the database took very long time the first time (20-30 minutes), and there was no message to alert us that the initialization was complete. The status of the initialization could be viewed with the command:
- *python3 dagda.py vuln –init_status*

This command had to be entered manually several times during the initialization of the vulnerability database before the initialization was complete.

When the database was fully initialized, the scanning process could be started with the command:
- *python3 dagda.py check –docker_image <image_name>:<tag>*

The scanning process did not alert us when the scan was completed. We had to manually enter the following command to see the results of the scan:
- *python3 dagda.py history <image_name>:<tag>*

The status of the output changed from “Analyzing” to a json output with the results when the scan was complete. We had to enter the above command manually several times before the scan was complete and the output changed to a json output with the results.

There seemed to be no way to export the results of the scans to a file with Dagda. The results of the scans were output to the terminal in json format though, and thus could be saved to a json file through the terminal with the command:
- *python3 dagda.py history <image_name>:<tag> > ~/Dagda/<report_name>.json*

One of the images could not be scanned as this generated an api-error in Dagda and crashed the service. We tried several times but could not get Dagda to scan that image.

3.5.5 Anchore

Installation started with the download of a YAML file for Docker Compose with the following command:
- *sudo curl -O https://engine.anchore.io/docs/quickstart/docker-compose.yaml*

This also showed that Anchore is a container that must be started from the downloaded docker-compose-file.

To start the Anchore engine, we entered the following command:
- *sudo docker-compose up –d*

After the container engine had started, the status of the system could be viewed with the command:
- *sudo docker-compose exec api anchore-cli system status*

Before Anchore could be used, the engine needed to sync the vulnerability data. To see the sync status of the vulnerability data, we had to enter the command:
- *sudo docker-compose exec api anchore-cli system feeds list*

This printed the list of vulnerabilities and the last sync date to the terminal.

To scan an image, Anchore first needed to download a separate copy of the image with the command:
- *sudo docker-compose exec api anchore-cli image add <image_name>:<tag>*

This downloaded the image and started analysing the content. Anchore does not alert the user automatically when the scan is complete, instead we had to use the following command to initiate a waiting process that asks the engine every 5 seconds if the scan was completed or not:
- *sudo docker-compose exec api anchore-cli image wait <image_name>:<tag>*

When the scan was complete, the waiting process stopped and generated information regarding the scanned image and the "Analysis Status" changed from "analyzing" to "analyzed".

When the scan was complete, we could see the results with the command:
- *sudo docker-compose exec api anchore-cli image content <image_name>:<tag> all*

All the found vulnerabilities were printed to the terminal with information regarding severity, name, a URL to the vulnerability, and more. A downside was that there was no summary regarding found vulnerabilities, and it seems that it is up to the user to count the found vulnerabilities manually. For this purpose, we build a short python program that could search through each report and output a summary with the different severities and found vulnerabilities.

There seemed to be no way to export the results of the scans to a file with Anchore. The results of the scans could be output to the terminal in json format though, and thus could be saved to a json file through the terminal with the command:
- *sudo docker-compose exec api anchore-cli --json image vuln <image_name>:<tag> all > <report_name>.json*

Because the images are downloaded to Anchore separated from images that are downloaded with “docker pull”, every image also had to be deleted manually within Anchore after every scan. Before an image could be deleted however, there were auto-initiated subscriptions for that image that had to be unsubscribed to. After the subscriptions had been unsubscribed, the image could be deleted with the command:
- *sudo docker-compose exec api anchore-cli image del <image_name<:<tag>*
The subscriptions could seemingly not be deleted in the version of Anchore used for this study, although the subscriptions are deactivated and the image is deleted, the subscriptions are still there. It seems the subscriptions are possible to delete when using the enterprise or UI versions of Anchore though.

3.5.6 Tool Statistics
Using python, we could extract the data from the reports generated by the tested tools. We could see the total number of found vulnerabilities by the tools and dive deeper into the reports to see differences in found vulnerabilities regarding scanned images and packets inside the images. The extraction of the information from the generated reports gave us the possibility to compare the tested tools and their results to investigate if there were any differences between them regarding the number, or type, of vulnerabilities found.

3.5.7 Vulnerability Statistics
With the data collected we could also see which of the found vulnerabilities were the most common. Because of the way that CVE:s are named, we could extract the year of registration for each found CVE and use that information to see the number of found CVE-vulnerabilities for each year.

4 Results

When conducting the tests with the tools, there was an obvious difference between the tools regarding the installation and vulnerability scanning process. As can be seen in section 3.5 Data collection, Docker Scan, Grype, and Trivy had a rather short installation process, while Anchore and Dagda needed more steps and preparation for installation and setup. The scanning process was started with a single command when using Docker Scan, Grype, and Trivy. For Anchore and Dagda, there were some more commands needed for the scanning process to be started and a report being saved. Anchore also downloaded its own copy of the image that it was told to scan, and that image needed to be erased from within Anchore.

When extracting and showing the results from the data in the generated reports, we used the python tool that we had created.

During the tests there were some great differences between the tools regarding vulnerabilities found. Although there were several images that had zero vulnerabilities according to all tools, the results were very different. One big difference was that not all tools reported the findings in the same way. The different severity levels reported by the tools can be seen in *table 2.* This meant that the results from the tools could not always be compared to each other in a straight fashion. Though most of the data reported was similar, some of the tools had other or no severity levels mixed into the report. Dagda, for example, had different bugs reported in the report. These bugs could have a CVE-identifier, but sometimes a BugtrackID or some other form of ID could be used as identifier. This made it somewhat difficult to grade some of the vulnerabilities reported, as there was no grading for severity available for some of the vulnerabilities found. We labelled these severity levels as “Other” for Dagda. Dagda also had a section for Malware in the generated reports, though no malware was found in the scanned images.

Severity	Anchore	Dagda	Docker	Grype	Trivy
Negligible	1812			1811	
Low	25	14	1922	253	2105
Medium	120	18	2236	546	545
High	164	0	1643	666	606
Critical	21	0	138	101	73
Unknown	134	83	0	302	4
Malware		0			
Other		541			
Totals	2276	656	5939	3679	3333

Table 2 – Tested tools and their reported total number of vulnerabilities found from scanning the 25 images, filtered by reported severity level and tool.

According to *tables 2 & 3*, the total results from the scans showed great differences regarding the number of found vulnerabilities. There was a great difference between Docker Scan that found the highest number of vulnerabilities (5939), and the rest of the tools. Dagda, that found the lowest number of vulnerabilities (656), found a considerably lower number of vulnerabilities than the rest of the tools. There were also great differences between the tools regarding the reported severity levels of the vulnerabilities found *(table 2 & figure 3).*

Alias	**Anchore**	**Dagda**	**Docker Scan**	**Grype**	**Trivy**
alias_0	0	0	0	0	0
alias_1	0	0	0	47	23
alias_2	88	3	353	176	170
alias_3	106	4	424	231	225
alias_4	0	0	0	0	0
alias_5	9	0	12	34	8
alias_6	36	56	65	97	28
alias_7	58	59	148	77	76
alias_8	0	0	0	0	0
alias_9	0	4	0	0	0
alias_10	120	59	472	169	75
alias_11	0	0	1	40	0
alias_12	58	59	151	80	79
alias_13	74	59	199	102	98
alias_14	89	5	375	187	182
alias_15	380	63	743	519	512
alias_16	60	3	254	134	134
alias_17	84	59	226	114	109
alias_18	664	88	1564	1028	1020
alias_19	58	59	148	78	76
alias_20	0	4	0	22	0
alias_21	17	5	45	21	17
alias_22	0	4	1	16	0
alias_23	0	0	0	0	0
alias_24	375	63	758	507	501
Total	2276	656	5939	3679	3333

Table 3 – Tools and their reported number of found vulnerabilities from the scans, filtered by image and tool. The name for each image is masked by an alias representing the image name.

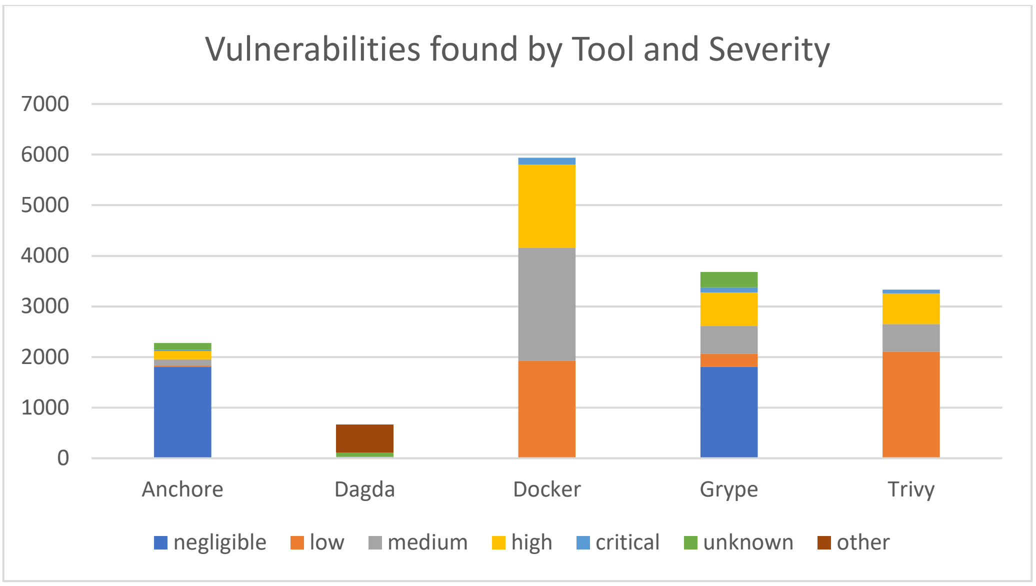


Figure 3 – Total number of vulnerabilities found for each tool, color-coded by reported severity level.

Upon comparing the total number of vulnerabilities reported by the tools against the total number of unique vulnerabilities reported by the tools, the results were according to *figure 4*. The total number of vulnerabilities found shows all vulnerabilities found including vulnerabilities found several times in the same image and packet (duplicates). Unique vulnerabilities found in same package shows the total number of found vulnerabilities, but not counting duplicate vulnerabilities found in the same package more than once. Unique vulnerabilities found in same image shows the total number of found vulnerabilities, but only counting unique vulnerabilities per image.

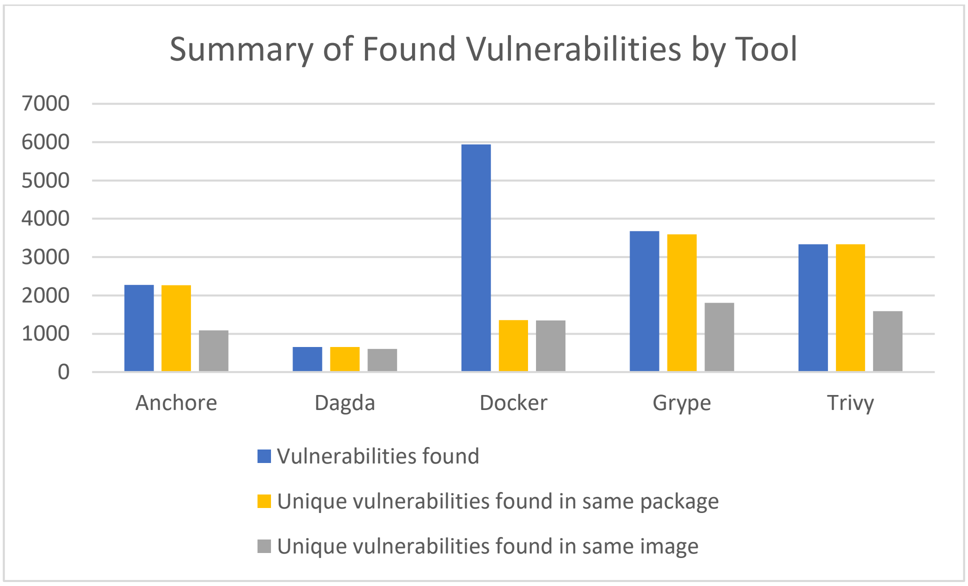


Figure 4 – Number of found vulnerabilities for each tool, color-coded by filtering of the vulnerabilities found.

When investigating the different vulnerabilities reported by the image scanning tools and their occurrences, the most frequently found vulnerabilities were according to *table 4*. The most found vulnerability, CVE-2022-0563, was found 319 times in total by all the tools tested. This number was significantly higher than the rest of the vulnerabilities found. This number is representing the total number of CVE-vulnerabilities found by all tools but excluding duplicate CVE-vulnerabilities in the same package.

Vulnerability	**Times Reported**
CVE-2022-0563	319
CVE-2018-5709	203
CVE-2004-0971	177
CVE-2021-39537	175
CVE-2022-1304	158
CVE-2010-4756	121
CVE-2018-20796	121
CVE-2019-1010022	121
CVE-2019-1010023	121
CVE-2019-1010024	121
CVE-2019-1010025	121
CVE-2019-9192	121
CVE-2022-29458	121

Table 4 – 14 most found vulnerabilities in descending order, unique vulnerabilities found per tool per image per package by all tools.

There were some old vulnerabilities present amongst the findings from the image scanning tools. Some vulnerabilities did not have a CVE ID attached and did not have a date in their ID. *Table 5* shows the number of found unique vulnerabilities per package, regarding vulnerabilities with a CVE ID and without a CVE ID. The total share of vulnerabilities that did not have a CVE ID was approximately 1 percent and were excluded in *table 6* due to lacking easily retrievable dates. The oldest vulnerabilities found were from 1999, although most of the vulnerabilities were from recent years, as can be seen in *table 6* and *figure 5*.

Vulnerabilities	**Anchore**	**Dagda**	**Docker Scan**	**Grype**	**Trivy**
With CVE	2258	626	1336	3527	3330
Without CVE	6	27	16	68	3

Table 5 – Found unique vulnerabilities per package. CVE and non-CVE.

Year	Anchore	Dagda	Docker Scan	Grype	Trivy	Total
1999	2	0	0	2	0	4
2000	0	0	0	0	0	0
2001	8	0	2	8	8	26
2002	0	0	0	0	0	0
2003	24	0	6	24	24	78
2004	59	0	12	59	58	188
2005	35	20	16	35	35	141
2006	1	41	0	1	0	43
2007	81	40	36	81	81	319
2008	40	11	12	40	40	143
2009	5	60	1	5	2	73
2010	65	11	28	65	65	234
2011	73	43	39	73	73	301
2012	10	1	3	10	10	34
2013	63	27	33	66	64	253
2014	14	83	4	14	12	127
2015	23	162	12	32	24	253
2016	98	93	64	121	119	495
2017	225	1	116	230	228	800
2018	278	10	97	313	305	1003
2019	321	8	175	451	440	1395
2020	98	0	146	302	270	816
2021	347	12	333	931	875	2498
2022	388	3	201	664	597	1853

Table 6 – Found CVE-vulnerabilities by year and tool. The values in the tool columns are the sum of unique CVE IDs per package per year. Total is the sum of the tool columns.

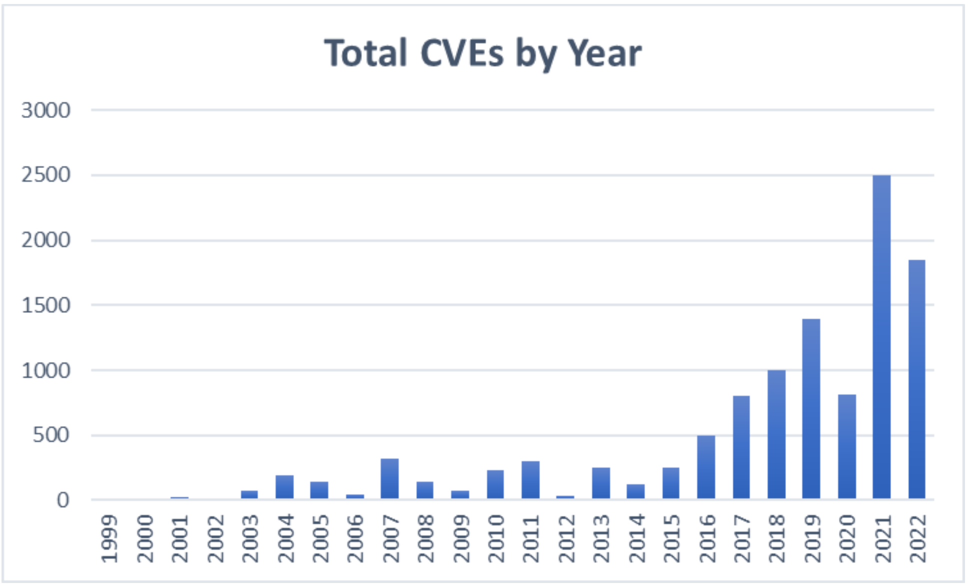


Figure 5 – Total found CVE-vulnerabilities and the year they were registered.

Table 7 shows the number of unique vulnerabilities found per image by each tool. At the bottom of the table the share of unique vulnerabilities found by each tool is displayed (Relative effectiveness). This number was generated by using the calculation described in section 3.3.1 Metrics for assessing tools. There is also a mean value presented that calculates the mean value for each image and the total share mean.

Alias	Anchore	Dagda	Docker Scan	Grype	Trivy	Mean
alias_0	0	0	0	0	0	0
alias_1	0	0	0	43	23	13,2
alias_2	42	3	91	92	92	64,0
alias_3	46	4	103	104	101	71,6
alias_4	0	0	0	0	0	0
alias_5	1	0	4	24	8	7,4
alias_6	27	51	23	41	22	32,8
alias_7	29	54	42	42	41	41,6
alias_8	0	0	0	0	0	0
alias_9	0	4	0	0	0	0,8
alias_10	63	54	71	81	62	66,2
alias_11	0	0	1	19	3	4,6
alias_12	30	54	45	45	44	43,6
alias_13	38	54	62	62	59	55,0
alias_14	43	5	88	89	87	62,4
alias_15	178	58	197	294	289	203,2
alias_16	32	3	70	70	70	49,0
alias_17	35	54	51	52	49	48,2
alias_18	220	78	257	354	349	251,6
alias_19	29	54	42	43	42	42,0
alias_20	0	4	0	17	5	5,2
alias_21	9	5	13	13	9	9,8
alias_22	0	4	1	14	1	4
alias_23	0	0	0	1	1	0,4
alias_24	175	58	184	281	276	194,8
Share	0,24	0,23	0,38	0,61	0,47	0,39

Table 7 – Number of unique vulnerabilities found per image and tool. Total share regarding found vulnerabilities at the bottom (Relative effectiveness).

Table 8 shows the results from the scans of an image with a pre-known set of vulnerabilities. The digit "1" marks if the tool found the vulnerability and the digit "0" marks if the tool did not find the vulnerability. The row at the bottom of the table is the share of vulnerabilities found by each tool.

Vulnerabilities	**Anchore**	**Dagda**	**Docker Scan**	**Grype**	**Trivy**
CVE-2014-4877	1	1	1	1	1
CVE-2015-1038	0	0	0	0	0
CVE-2016-3170	0	0	0	0	0
CVE-2016-3169	0	0	0	0	0
CVE-2016-3168	0	0	0	0	0
CVE-2016-3164	0	0	0	0	0
CVE-2016-3163	0	0	0	0	0
CVE-2016-3092	1	1	1	1	1
CVE-2013-4590	1	1	0	1	0
CVE-2014-0119	1	1	1	1	1
CVE-2014-0099	1	1	1	1	1
CVE-2014-0096	1	1	1	1	1
CVE-2014-0075	1	1	1	1	1
CVE-2016-3956	1	0	0	1	1
CVE-2016-2216	0	0	0	0	0
CVE-2016-2086	0	0	0	0	0
CVE-2014-7187	0	0	0	0	0
CVE-2014-7186	0	0	0	0	0
CVE-2014-7169	0	0	0	0	0
CVE-2014-6278	0	0	0	0	0
CVE-2014-6277	0	0	0	0	0
CVE-2014-6271	0	0	0	0	0
CVE-2016-3610	1	0	1	1	1
CVE-2016-3606	1	0	1	1	1
CVE-2016-3598	1	0	1	1	1
CVE-2016-3587	1	0	1	1	1
CVE-2016-3550	1	0	1	1	1
CVE-2016-3508	1	0	1	1	1
CVE-2016-3500	1	0	1	1	1
CVE-2016-3485	0	0	0	0	0
Share	0,5	0,23	0,43	0,5	0,47

Table 8 – Results from the scans of an image with a pre-known set of vulnerabilities.

5 Discussion

5.1 Discussion of the Results

Some of the tools tested had a very complex setup- and run-process. These tools were very prone to errors and were not perceived as user-friendly. Dagda was one such tool, that had a very complex setup-process and was very prone to errors during the setup of the tool and scanning of images. Overall, Grype, Trivy and Docker Scan was perceived as the most user-friendly tools to work with, with a slight edge in favour of Grype and Trivy. This was because they did not need the user to login to any account or install any dependencies or other actions prior to running the tool. All that was needed for us to do was to install Grype and Trivy, then download the desired image and start the scanning process. Trivy alerted us when the scan was complete and did have the option to save the generated report in various formats, something that all other tools missed.

Regarding the reports generated by the tools, there was some difference expected. Surprisingly, most reports generated by the tools had a high amount of similarity and was mostly easy to work with when extracting information. Again, Dagda marked itself as a little different than the other tools. Dagda had a more complex report where the desired information was not always in the same place. Dagda did not always report severity-level for all CVE-vulnerabilities, whereas the severity-level for a vulnerability could not always be retrieved.

The different severity-levels reported and the number of vulnerabilities for each severity-level differed between the tools *(table 2 & figure 3).* This is of course a result of the tools reporting a different number of found vulnerabilities. There is however a slight disturbance here. We noticed that some of the tools reported different severity-levels for duplicate vulnerabilities on several occasions. This meant that the same vulnerability could be reported with, the most extreme found, a severity level spanning from “low” to “critical”. This was perceived by us as an odd behaviour since the vulnerabilities we looked up had a fixed severity-level. This could be due to the tools using multiple different sources for their vulnerability data, but the same severity should not be able to obtain such an extreme spread in severity-level.

When looking at Anchore and Grype, we could see that the “negligible” and “low” severities roughly correspond to the “low” severities reported by Trivy. This could be the result of Anchore and Grype being the only tools reporting the “negligible” metric for severity and that the “negligible” metric could be translated to “low”. Docker Scan, Grype, and Trivy reported a considerably higher number of vulnerabilities with a severity-level of “high” and “critical” compared to Anchore and Dagda.

When the total number of vulnerabilities found by the tools was examined, there was a big difference between the tools, as seen in *tables 2 & 3*. Docker Scan found 5939 vulnerabilities in total, as compared to Grype that found the second highest number with 3679 vulnerabilities and Trivy that found 3333 vulnerabilities. Remarkable here was that Dagda, the tool that seemed to have the highest potential when looking into the capabilities of the different tools in section 2.5 Tools, reported only 656 vulnerabilities found in total. This is a great difference in found vulnerabilities that generates some thought as to how there could be such a great difference among the tools regarding the total number of found vulnerabilities. We could also see that there was a considerable difference between Anchore and Grype regarding the number of found vulnerabilities although they are from the same company. This could be due to the fact that Grype is a newer tool than Anchore or that the company put more effort into Grype. When looking at the tools in section 2.5 Tools, there was no noticeable difference between the tools.

Upon examining the number of duplicate vulnerabilities reported by each tool, there was some clarification to the high number of total vulnerabilities found by Docker Scan. With the term "duplicate", we mean a vulnerability that has been found several times in the same image and package. When looking at *figure 4*, we could see that there were some interesting patterns regarding total vulnerabilities found and unique vulnerabilities found. When inspecting the number of vulnerabilities found, we could see that, for example, Docker Scan had a great difference in the number of total vulnerabilities found compared to the number of total package-unique vulnerabilities found. With total vulnerabilities found, we mean all vulnerabilities found when scanning the images. With total package-unique vulnerabilities found, we mean the total number of vulnerabilities found when not counting vulnerabilities found several times in the same package in the same image more than once per package.

Docker Scan showed itself to report a high number of duplicate vulnerabilities and at most we could find the same CVE reported 118 times in the same package. The other tools did not show this pattern and had a rather small difference between total and unique vulnerabilities found. Grype and Trivy had the highest number of unique vulnerabilities found together with a rather small difference in total and unique vulnerabilities found. Dagda had similar small differences as the other tools but had an overall considerably lower number of vulnerabilities found.

When investigating the total number of found vulnerabilities and the year they were registered, we could see that most vulnerabilities found were from recent years *(tables 4 & 5)*. The highest number of vulnerabilities found were from 2021, with 2022 as the year with the second highest number of found vulnerabilities. This could hint at 2022 will soon pass 2021 regarding found vulnerabilities as 2022 was barely mid-way though when this study was conducted. Vulnerabilities from earlier years in images will hopefully decrease as the images are updated. The number of vulnerabilities reported per year did only count vulnerabilities with a CVE ID. As can be seen in *table 5*, the total number of reported unique vulnerabilities per package that did not have a CVE ID was approximately 1 percent. These vulnerabilities were different bugs that did not have a CVE ID attached to them.

The vulnerability found the highest number of times was CVE-2022-0563, which has the current description from https://nvd.nist.gov/vuln/detail/CVE-2022-0563:

"A flaw was found in the util-linux chfn and chsh utilities when compiled with Readline support. The Readline library uses an "INPUTRC" environment variable to get a path to the library config file. When the library cannot parse the specified file, it prints an error message containing data from the file. This flaw allows an unprivileged user to read root-owned files, potentially leading to privilege escalation. This flaw affects util-linux versions prior to 2.37.4."

The other vulnerabilities that were found a high number of times were from mostly recent years, except for CVE-2004-0971 that is significantly older than the other most found vulnerabilities. CVE-2004-0971 has the following description from https://nvd.nist.gov/vuln/detail/CVE-2004-0971:

"The krb5-send-pr script in the kerberos5 (krb5) package in Trustix Secure Linux 1.5 through 2.1, and possibly other operating systems, allows local users to overwrite files via a symlink attack on temporary files."

It was not anticipated by us that we would see such an old vulnerability among the vulnerabilities found the highest number of times. This could mean that this vulnerability is in a sensitive part of the package that is hard to update or re-write. We have not been able to dive deeper into this subject though.

When using our formula for calculating the relative effectiveness of each tool, we found some interesting results *(table 7)*. When looking at the total number of unique vulnerabilities reported by the tools, we found that Docker Scan that had the highest total number of found vulnerabilities (including duplicates), did not score well regarding found unique vulnerabilities in the images. Docker Scan had a relative effectiveness of 0,38, which translates to that approximately 38 percent of the total reported unique vulnerabilities were also found by Docker Scan. On the other hand, Grype managed the highest score with a relative effectiveness of 0,61, which translates to that approximately 61 percent of the total reported unique vulnerabilities were also found by Grype. The mean value of relative effectiveness among the tested tools was 0,39, which puts Grype (0,61) and Trivy (0,47) at a rather high level of relative effectiveness compared to the other tools tested. Dagda (0,23) and Anchore (0,24) were the tools that had the lowest relative effectiveness score and they scored considerably lower than the rest of the tested tools.

When comparing our results to an earlier study (Javed & Toor, 2021b), we could see that they concluded that the highest-ranking tool in their study could miss approximately 34 percent of the vulnerabilities in OS-packages when scanning an image. In our study, the tool with the highest effectiveness score could miss approximately 39 percent of the vulnerabilities in both OS and non-OS packages when scanning an image. In Javed & Toor (2021b), Anchore was the tool that they concluded missed the least number of vulnerabilities with approximately 34 percent of the vulnerabilities missed. In our study, Anchore got a relative effectiveness of 0,24, which translates to that Anchore could miss approximately 76 percent of the vulnerabilities when scanning an image. Notable is that Javed & Toor (2021b) only scanned OS packages, while we, in this study, scanned both OS and non-OS packages.

We also scanned an image where there was a pre-known set of 30 vulnerabilities to investigate how well the different tools could find the pre-known vulnerabilities. As can be seen in *table 8*, none of the tools could find all the pre-known vulnerabilities. Grype and Anchore found 50 percent of the pre-known vulnerabilities while Trivy found approximately 47 percent and Docker Scan found approximately 43 percent. Dagda scored considerably lower than the rest of the tools with only approximately 23 percent of the pre-known vulnerabilities found. This also corresponds to our calculated relative effectiveness of the tools, where Grype had the highest relative effectiveness and Dagda had the lowest relative effectiveness.

5.1.2 Are there any clear differences between the number of vulnerabilities found by different image vulnerability scanning tools?

There was a significant difference between the tested tools regarding found vulnerabilities. Docker Scan found a considerably higher total number of vulnerabilities than the other tools and Dagda found by far the lowest total number of vulnerabilities. When filtering the found vulnerabilities to exclude duplicate vulnerabilities from the same package, we could see that Docker Scan had dropped considerably and was passed by Grype and Trivy regarding the number of found vulnerabilities. Docker Scan was the only tool that showed this characteristic. All the other tools sustained roughly the same number of found vulnerabilities when excluding duplicates from the same package. Why Docker Scan reports such a high number of duplicate vulnerabilities for packages is unclear to us, but it is an aspect that needs to be studied further to fully understand the great differences in the number of found/reported vulnerabilities.

Grype and Trivy had similar results regarding the number of found vulnerabilities and did not show the considerable drop in found vulnerabilities, as Docker Scan did, when excluding duplicates from the same package. They both showed great consistency and a high number of found vulnerabilities throughout. Another remarkable tool was Dagda, that showed great consistency regarding the number of vulnerabilities found but reported a significantly lower number of found vulnerabilities overall than the rest of the tools. There could be many reasons for this, the database Dagda uses could be out of date, or the scanner itself could simply be a bad performing tool.

5.1.3 Are there any differences between the types of vulnerabilities found by different image vulnerability scanning tools?

There were some differences between the tools when looking at the different types of vulnerabilities found. The most obvious difference was that the tools did not all report the same severity levels for the found vulnerabilities. Docker Scan, for example, could report up to 4 different severity-levels for the same found vulnerability, in the same image. Docker Scan, Grype, and Trivy reported a considerably higher number of vulnerabilities with a severity-level of “high” and “critical” compared to Anchore and Dagda. This could be due to Docker Scan, Grype, and Trivy reporting an overall higher number of found vulnerabilities. However, the most important vulnerabilities for a scanning tool to find should be the “high” and “critical” vulnerabilities. There is also a possibility that Anchore and Dagda reported some vulnerabilities to have a lower severity-level than the other tools.

5.1.4 What is the relative effectiveness of different image vulnerability scanning tools?

Using our formula for calculating relative effectiveness, we could see that there were some considerable differences between the tools *(table 7)*. Grype had a relative effectiveness score of 0,61, which translates to approximately 61 percent of the vulnerabilities found when scanning images. Dagda was found to be the tool that had the lowest calculated relative effectiveness with a score of 0,23. This is a considerable difference in relative effectiveness, and it implies that Dagda could miss approximately 77 percent of the vulnerabilities when scanning images. When we scanned a single image that had a pre-known set of vulnerabilities, we could see that the results were similar to our calculated relative effectiveness for the tools. Only Anchore differed slightly more from the relative effectiveness compared to the other tools.

5.2 Usability of the Tested Tools

When testing the image scanning tools, we found out that there was a great difference in useability between the tools. Dagda and Anchore had to be installed and set-up through several steps and with several dependencies. They were also prone to errors while setting up and running the scans. This could lead to users not being able to use the tools or running the tools not as they were intended. Grype and Trivy was the tools that was by far the easiest to install and use since we only had to install the scanner tool, download the image, and start the scanning process. Trivy also had the possibility to select how to save the report and told us when the scan was finished. With all the other tools tested we had to use Linux built-in functionality to save the terminal output (report) to a json-file.

5.3 Limitations

During the study we used our proposed metric of relative effectiveness to measure the share of unique vulnerabilities found by a certain tool out of the total number of unique vulnerabilities found by all tools for a specific image. This was then calculated for all images and a mean value was calculated to set a relative effectiveness score for each tool. This method has its limitations because it does not consider the actual “true” value of existing vulnerabilities in an image. It can only be used to calculate a score of relative effectiveness of a tool and then compare the score of the tools to each other to achieve an “internal” relative effectiveness among the tools. The calculated relative effectiveness gives a score for how effectively a tool could find vulnerabilities in scanned images, within a group of tools that all scanned the same images. This method needs to use data from several tools to be able to calculate relative effectiveness. The more tools that are used, the more accurate should the relative effectiveness score be. This is because, for each added tool, the probability should increase that the found vulnerabilities from the scans more accurately reflect the “real” existing vulnerabilities in the scanned images.

There is also the possibility that a way to include the reported severity-levels as a metric when calculating the effectiveness of a scanning tool could give a more accurate measurement and description of the “real” effectiveness. This would be an area for further study, as the effectiveness and performance of vulnerability scanning tools is important in the work for increased security.

This study compared several popular scanning tools and their functionality for scanning Docker Container Images from Docker Hub. Due to the constant evolving nature of the scanning tools and Docker Hub, the results of this study will only reflect the studied scanning tools and Docker Hub landscape at the time the study was conducted and for a short time thereafter. The study can be replicated but the results will most likely differ due to updated tools or dependencies. The images where selected with specific tags and therefore would be possible to use again, but since the name of the images was chosen to be set as an alias due to ethical reasons, this has a negative impact on the reproducibility of the study.

6 Conclusion

This study showed that there are considerable differences between tools regarding the number of found vulnerabilities and how they are reported. There were also great differences between the tools regarding the number of duplicate vulnerabilities found. Docker Scan found the same CVE multiple times in the same package in the same image on several occasions and a deeper understanding of this is needed. Furthermore, there seems to be a great inconsistency between the tools regarding the severity level as the same CVE could be reported with up to four different severity-levels from for example Docker Scan. The calculated relative effectiveness score showed that there were considerable differences regarding relative effectiveness between the tools. Although Grype scored significantly higher than most of the other tools, Grype could still miss approximately 39 percent of the vulnerabilities when scanning an image. Dagda that scored the lowest for relative effectiveness could miss approximately 77 percent of the vulnerabilities when scanning an image.

6.1 Future Work

Container image scanning tools are an important part of securing container images. The tools used for scanning the images need to be further studied to be able to understand the effectiveness of the tools and find areas where there is a need for improvement. The different numbers reported regarding duplicate vulnerabilities needs to be studied further to understand why there is such a great difference between the tools and what this means. Further studies also need to be conducted regarding the different severity-levels reported for the same vulnerabilities and where this behaviour comes from. It would be interesting topic to study and compare the reported severity-levels between scanning tools and investigate how the reported severity-levels compare to the severity-level registered in the CVE MITRE database. This could be a part of a tool's total accuracy, the accuracy of the reported severity-levels.

7 Final Words

One thing that got obvious during the study of Docker images, containers, and scanning tools, is that there is a need for an easier way of securing the images and containers. It is hard to secure the images since most of the hardening tools/mechanisms are set up during the start of a container from the image. The different ways of hardening a container have to be set up as parameters/arguments when running/starting the container. This in turn leads to the creators of an image not being able to control the security of the image/container to the extent needed, since the tools or mechanisms needed for that is not available until the image is used to run a container. This also puts a big piece of the container security at the hands of the user. For a user new to the container technology, this is an important part of the security that can easily be forgotten or looked over. Some of the authors provide docker-compose files where security is set up, which, in many ways, makes the process of securing a container easier.

Container image scanning tools should be easy to use and should not show the extreme spread in found vulnerabilities and effectiveness that was seen in this study. There should be a clear documentation available for a tool regarding the different metrics reported and how duplicates and different severity-levels should be read and understood. There is a great need for a standard regarding the grading of vulnerabilities with the aim of vulnerability scanning tools measuring and reporting the same metrics. This would increase the standard and effectiveness of security-enhancing work related to container technology. There is already the CVSS-system, but there still seems to be some disagreement or poor co-operation between different instances regarding the use of a common grading system. There needs to be a clear severity-grading and the extreme spread for the same severity seen in this study is confusing and serious regarding the security of container images.

References

Ahmad, I., Dimitriou, T., & Sultan, S. (2019). Container Security: Issues, Challenges, and the Road Ahead. *IEEEAccess*, 7, 52976-52996.

Aqua Security. (2022). *MicroScanner is now deprecated in favour of Trivy*. From Github. Retreived 21 April, 2022, from https://github.com/aquasecurity/microscanner

Aqua Security. (n.d.). *Container Platforms: 6 Best Practices and 15 Top Solutions*. Retrieved May 16, 2022, from https://www.aquasec.com/cloud-native-academy/container-platforms/container-platforms-6-best-practices-and-15-top-solutions/

Docker. (n.d.). *What is a Container?* Retrieved April 5, 2022, from https://www.docker.com/resources/what-container/

Docker docs. (n.d.a). *Docker Engine overview*. Retrieved May 24, 2022, from https://docs.docker.com/engine/

Docker docs. (n.d.b). *Docker Hub Quickstart*. Retrieved May 24, 2022, from https://docs.docker.com/docker-hub/

Docker docs. (n.d.c). *Seccomp security profiles for Docker*. Retrieved April 7, 2022, from https://docs.docker.com/engine/security/seccomp/

Docker docs. (n.d.d). *Use Docker Compose*. Retrieved May 20, 2022, from https://docs.docker.com/get-started/08_using_compose/

Docker Newsroom. (n.d.). *Docker by the numbers*. Docker. Retrieved April 5, 2022, from https://www.docker.com/company/newsroom/

Forum of Incident Response and Security Teams. (n.d.). *Common Vulnerability Scoring System SIG*. Retrieved May 20, 2022, from https://www.first.org/cvss/

Grype. (2022). *Grype*. From Github. Retreived June 2, 2022, from https://github.com/anchore/grype

Hiter, S. (February 25, 2022). Top Container Software & Platforms 2022. *Enterprise Networking Planet*. Retrieved from https://www.enterprisenetworkingplanet.com/guides/container-software/

Javed, O., & Toor, S. (2021a). Understanding the Quality of Container Security Vulnerability Detection Tools. *ArXiv:2101.03844 [Cs]*. http://arxiv.org/abs/2101.03844

Javed, O., & Toor, S. (2021b). An Evaluation of Container Security Vulnerability Detection Tools. In *2021 5th International Conference on Cloud and Big Data Computing (ICCBDC)* (pp. 95–101). Association for Computing Machinery. https://doi.org/10.1145/3481646.3481661

Jenna, P. (July 15, 2021). Top Container Software Solutions 2022. *Enterprise Storage Forum*. Retrieved from https://www.enterprisestorageforum.com/software/container-software/

Lin, C., Nadi, S., & Khazaei, H. (2020). A Large-scale Data Set and an Empirical Study of Docker Images Hosted on Docker Hub. *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 371–381. https://doi.org/10.1109/ICSME46990.2020.00043

Man7.org. (August 27, 2021a). *capabilities(7)—Linux manual page*. Retrieved April 8, 2022, from https://man7.org/linux/man-pages/man7/capabilities.7.html

Man7.org. (August 27, 2021b). *cgroups(7)—Linux manual page*. Retrieved April 7, 2022, from https://man7.org/linux/man-pages/man7/cgroups.7.html

Man7.org. (August 27, 2021c). *namespaces(7)—Linux manual page*. Retrieved April 7, 2022, from https://man7.org/linux/man-pages/man7/namespaces.7.html

Man7.org. (August 27, 2021d). *seccomp(2)—Linux manual page*. Retrieved April 8, 2022, from https://man7.org/linux/man-pages/man2/seccomp.2.html

Martin, A., Raponi, S., Combe, T., & Di Pietro, R. (2018). Docker ecosystem – Vulnerability Analysis. *Computer Communications*, *122*, 30–43. https://doi.org/10.1016/j.comcom.2018.03.011

Matt, C. (October 7, 2021). *Docker Index Shows Momentum in Developer Community Activity*. Docker. [Blogpost] https://www.docker.com/blog/docker-index-shows-surging-momentum-in-developer-community-activity-again/

Moby. (2022). *The Moby Project*. [Go]. Retrieved April 8, 2022, from https://github.com/moby/moby/blob/5f17312653c3e4dc5474f86692b09f06262a1ebd/oci/defaults.go

Mullinix, S. P., Konomi, E., Townsend, R. D., & Parizi, R. M. (2020). On Security Measures for Containerized Applications Imaged with Docker. *ArXiv:2008.04814 [Cs]*. http://arxiv.org/abs/2008.04814

NIST. (n.d.). *Computer Security Resource Center*. Retrieved May 20, 2022, from https://csrc.nist.gov/glossary/term/vulnerability

PhoenixNAP. (October 31, 2019). *Docker Image VS Container: What is the difference?* Retrieved from https://phoenixnap.com/kb/docker-image-vs-container

Prevasio. (2020). *Operation Red Kangaroo*. Retrieved May 24, 2022, from https://knowledge-base.prevasio.io/pdf.html?file=Red_Kangaroo.pdf

Reeves, M., Tian, D. J., Bianchi, A., & Celik, Z. B. (2021). Towards Improving Container Security by Preventing Runtime Escapes. *2021 IEEE Secure Development Conference (SecDev)*, 38–46. https://doi.org/10.1109/SecDev51306.2021.00022

Shen, Y., & Yu, X. (2020). Docker container hardening method based on trusted computing. *Journal of Physics: Conference Series*, *1619*, 012014. https://doi.org/10.1088/1742-6596/1619/1/012014

Software Testing Help. (April 4, 2022). *Top 10 Best Container Software in 2022*. Retrieved from https://www.softwaretestinghelp.com/container-software/

The MITRE Corporation. (n.d.). *About the CVE Program*. Retrieved May 5, 2022, from https://www.cve.org/About/Overview

VMware. (n.d.). *What is a Hypervisor?* Retrieved May 17, 2022, from https://www.vmware.com/topics/glossary/content/hypervisor.html

Wenhao, J., & Zheng, L. (2020). Vulnerability Analysis and Security Research of Docker Container. *2020 IEEE 3rd International Conference on Information Systems and Computer Aided Education (ICISCAE)*, 354–357. https://doi.org/10.1109/ICISCAE51034.2020.9236837

Yasrab, R. (2021). Mitigating Docker Security Issues. *ArXiv:1804.05039 [Cs]*. http://arxiv.org/abs/1804.05039

Zerouali, A., Mens, T., & De Roover, C. (2021). On the usage of JavaScript, Python and Ruby packages in Docker Hub images. *Science of Computer Programming*, *207*, 102653. https://doi.org/10.1016/j.scico.2021.102653