TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Oliver Vanamets 212032IVCM

DOCKER IMAGE SECURITY SCANNING USING OPEN SOURCE TOOLS

Master's Thesis

Supervisor: Mauno Pihelgas
PhD

Tallinn 2025

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Oliver Vanamets 212032IVCM

DOCKERI TÕMMISTE TURVAANALÜÜS KASUTADES AVATUD LÄHTEKOODIGA TÖÖRIISTU

Magistritöö

Juhendaja: Mauno Pihelgas
PhD

Tallinn 2025

Author's Declaration of Originality

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

Author: Oliver Vanamets

02.01.2025

Abstract

Docker has become the dominant force in the software deployment world. The accelerated development cycle that Docker's infrastructure-agnostic design allows makes securing such environments a difficult task.

This thesis aims to help practitioners utilize static vulnerability scanning tools to reduce the potential attack surface of Docker deployments. This reduction is achieved through the detection and remediation of known vulnerabilities found in Docker containers.

First, a brief overview of Docker architecture and the general vulnerability scanning procedure is given. Different vulnerability scanners that are either used by practitioners or mentioned in previous research will be looked at. Scanners that are open source, allow free commercial use, and are actively maintained are selected for testing.

The testing section gives detailed information about how the scanners will be tested and the data gathered. Possible custom configuration options are also explored. Selected scanners are then tested and the findings are presented and explained in the results section. The usability aspects and additional functionality of the tools are also discussed.

Analysis identified Dependency Track as the leading solution for static vulnerability scanning. Some recommendations are also given on how practitioners can begin managing the large volume of results to obtain useful insights into their security posture.

The thesis is written in English and is 56 pages long, including 11 chapters, four figures, and seven tables.

Annotatsioon
Dockeri tõmmiste turvaanalüüs kasutades avatud lähtekoodiga tööriistu

Docker on saanud tarkvaraarenduse ja -halduse maailmas domineerivaks töövahendiks. Kiirendatud tarkvaraarenduse protsess mida Docker võimaldab on muutnud tarkvara turvalisuse tagamise keeruliseks.

Selle lõputöö eesmärk on aidata küberturbe spetsialistidel kasutusele võtta staatilise turvaanalüüsi tööriistu, et vähendada potentsiaalseid ründevektoreid Dockerile ehitatud süsteemides. Seda eesmärki aitab saavutada Dockeri konteinerite teadaolevate turvanõrkuste tuvastamine ja kõrvaldamine.

Kõigepealt antakse põgus ülevaade Dockeri arhitektuurist ning üldisest haavatavuste skännerite tööpõhimõttest. Vaadeldakse erinevaid skännereid mis on praegu teadaolevalt selles valdkonnas kasutusel või mida on uuritud varasemates uurimistöödes. Skännerid, mis on avatud lähtekoodiga, lubavad tasuta kommertskasutust ning on jätkuvalt aktiivses arenduses võetakse testimisele.

Testimise peatükis antakse detailne kirjeldus kogu testimise protsessist ja sellest kuidas andmeid kogutakse. Uuritakse ka võimalikke eriseadeid ja konfiguratsioone mida skännerid pakuvad. Seejärel testitakse väljavalitud skännereid ning tulemuste peatükis esitletakse ja selgitatakse testimise tulemusi. Samuti arutletakse ja võrreldakse skännerite kasutusmugavust ja lisafunktsionaalsust.

Analüüsi tulemusel selgus, et parim lahendus staatiliseks haavatavuste tuvastamiseks on Dependency Track. Lisaks antakse spetsialistidele mõningaid näpunäiteid kuidas tulla toime suure hulga leitud haavatavustega ning saada kasulikku informatsiooni oma süsteemide turvalisuse hetkeseisu kohta.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 56 leheküljel, 11 peatükki, nelja joonist ja seitset tabelit.

List of Abbreviations and Terms

API	Application Programming Interface
SBOM	Software Bill of Materials
CI/CD	Continuous Integration/Continuous Delivery
OCI	Open Container Initiative
CPE	Common Platform Enumeration
SWID	Software Identification Tag
PURL	Package URL
URL	Uniform Resource Locator
AWS	Amazon Web Services
ECS	Elastic Compute Service
RHEL	Red Hat Enterprise Linux
BID	Bugtraq ID
RHSA	Red Hat Security Advisories
RHBA	Red Hat Bug Advisories
OWASP	Open Worldwide Application Security Project
CVE	Common Vulnerabilities and Exposures
IaC	Infrastructure as Code
NVD	National Vulnerability Database
GHSA	GitHub Security Advisories
OSS	Open Source Software
JSON	JavaScript Object Notation
XML	Extensible Markup Language
DT	Dependency Track
SLES	SUSE Linux Enterprise Server
EPSS	Exploitation Probability Scoring System

Table of Contents

List of Figures

List of Tables

1. Introduction

The traditional method of software development involves long release cycles, often spanning months or even years. Each release goes through extensive testing - security and otherwise - and is then packaged and deployed on physical servers or virtual machines (VMs). Usually, a set of dependencies or other software is also deployed along with the main application. This, however, creates a few issues especially on the operations side of things.

Deploying an application and its dependencies on physical servers or VMs requires significant time and effort for provisioning, configuring, and maintaining the underlying infrastructure. Scaling an application involves adding more servers or VMs, leading to higher costs and more complexity. Changes to software were typically slow and deployments caused downtime, requiring delicate rollback procedures if issues emerged. Some of these issues can be mitigated by the use of containers.

The faster release cycle induced by containerization created a new problem: it left little time for extensive security testing. Agile and continuous software delivery paradigm prioritizes rapid feature development and frequent releases. This pace does not allow for thorough security reviews, which traditionally requires manual and time-consuming testing. As development teams push code to production more frequently, it becomes difficult to perform extensive security tests on every update. To mitigate this, security testing needs to evolve with automation to ensure security checks happen continuously, without slowing down development.

1.1 Motivation

The use of containers as a software deployment method started gaining popularity after 2013 with the introduction of Docker, an open platform for developing and deploying containerized applications. Around 50% of global organizations ran containerized applications in 2023 and it is estimated that this will increase to 95% by 2029 [1]. Combining the already high popularity of containers with the fact that an estimated 87% of containers running in production have a known high or critical severity vulnerability makes container security an even more pressing issue [2]. Most of these known vulnerabilities could be fixed as 71% of the vulnerabilities have a fix available that has not yet been applied [2].

In May 2021, The U.S. President's Executive Order 14028 on improving the nation's cyber security was issued. This was, in part, a response to the SolarWinds software supply chain attack. SolarWinds attack was discovered near the of 2020 and targeted U.S. government institutions and collaterally affected many private sector organizations as well [3]. The order outlines actions to be taken by several U.S. government agencies and defines cyber security requirements for government contractors [4]. Part of this executive order was directed at defending against software supply chain attacks. This order initiated the standardization of SBOMs and their use in the government sector. This also had implications in the private sector. Since then a growing number of organizations are focusing on remediating software supply chain attacks. This, of course, has influenced the whole industry to pay more attention to such threats and to figure out possible remediation methods. This is partly the motivation behind this research.

Patching the known vulnerabilities themselves is a rather trivial process when there is already a fixed version of the vulnerable component available. Analyzing the containers and identifying their vulnerable components before and during use is a more difficult endeavor. An example of this is the Log4J vulnerability. Although Apache released the final patch for it about a month after its discovery in November 2021, it was still one of the most commonly exploited vulnerabilities in May 2023 [5]. U.S. Department of Homeland Security predicted that due to the prolific use of Apache software and lackluster patching practices, Log4J may remain exploitable for a decade [6]. Static analysis of container images could identify the presence of such vulnerabilities.

While open-source tools exist to carry out such analysis, choosing the most suitable one is not a straight-forward process. Practical usefulness of the scanning results, compatibility with CI/CD pipelines, integration into existing security processes, and general ease-of-use must all be taken into consideration. All this, along with the ever-changing threat landscape and set of possible tools to use necessitates formal and up-to-date research into this topic.

1.2 Problem Statement

Most container image scanners work in a similar fashion. A given container image is broken down into its components: the base image and dependencies or other included software packages needed to run the software in the container. A scanner then compares the base image and the list of dependencies/packages to a list of known vulnerabilities or CVE's (*Common Vulnerabilities and Exposures*) to determine if any of the known vulnerabilities apply to any of the components contained in the image. The quality of the results is hypothesized to be largely dependent on the scanners' ability to accurately discover what is included in a given image and the public vulnerability databases that the scanner uses to compare against.

Previous research has shown that different container image scanners give drastically different results when scanning the same image. This highlights the need to further investigate the scanning tools themselves and the results that they give to explain the discrepancies.

1.3 Research Questions

The main research question is: How to reduce the attack surface of Docker deployments using container image scanning?

This leads to the following sub-questions:

- What open-source and free to use tools are available for this task?
- What causes the discrepancies in the results that different scanners give for the same image?
- How to configure the tools and interpret their results to maximize their usefulness?

1.4 Scope

The focus of this study is a comparative analysis of available static container vulnerability scanning tools. It aims to evaluate vulnerability scanners in relation to one another. It is important to note that this thesis does not seek to evaluate the concept or efficacy of static vulnerability scanning as a whole.

Security scanning during runtime is excluded in favor of a deeper analysis of static vulnerability scanning since the former is highly dependent on the peculiarities of the software stack onto which it is implemented. Custom security policies which are paramount to the success of runtime scanning must be tailored to the specific situation at hand. As such, overarching recommendations for runtime scanning are difficult to give and such suggestions would lack the broader applicability that this thesis aims to have.

While there are other types of application containers out there other than Docker, its container format became the cornerstone of OCI's (*Open Container Initiative*) standardized container image specification. This means that scanning tools compatible with Docker images are also compatible with other OCI-compliant container image types. As such, the comparison of the tools can be done using Docker images and the results will be applicable to other OCI-compliant container images, at least to some extent. Docker images are preferred as Docker has about 83% market share as a containerization tool [7]. This allows for a wide range of different Docker images to be used for testing.

As 97% of Docker containers use some form of Linux as their base image, the images chosen for this research are also Linux-based [2]. Tests are also conducted on a Linux host but since this research focuses on static analysis, there is no need to actually run any of those containers.

As there is usually less technical information and community support on closed-source software publicly available, open-source tools are chosen for this research. Also, since this thesis is aimed at practitioners working in enterprise environments, any tools under a license that does not allow unlimited commercial use will be excluded.

1.5 Goals and Main Outcomes

Amidst the rapid growth of the use of containerization technologies it is imperative to secure these environments against the ever-changing threat landscape. This thesis aims to help the software development and operations community by providing a comprehensive evaluation of static vulnerability scanners. Through thorough comparative analysis of their usability and effectiveness, this research seeks to guide practitioners in making informed decisions about the security tools best suited for their environments. The goal is to provide actionable insights for organizations selecting among existing tools while balancing coverage, usability, and cost considerations.

The main outcomes of this thesis are the following: a recommendation of a scanning tool for the enterprise environment based on comprehensive testing of currently available tools and an investigation into the root cause(s) of discrepancies between the results of different scanners.

1.6 Research Methods

The main scanning functionality of the tools will be tested using experimental research methods such as scanning a set of container images and analyzing the results. The set of images will be compiled based on what is used at a medium to large scale public-sector organization. In addition to that - to reduce organizational bias - 200 most popular Docker Official or Verified Publisher images will be added. Altogether, almost 400 images will be analyzed. Each tool will be used to scan all the images in that combined set. The results across all scanners are then aggregated and compared.

The tools' effectiveness will partly be judged according to their coverage or sensitivity. Coverage or sensitivity in this case means the relative amount of found vulnerabilities (true positives) compared to all relevant vulnerabilities (sum of true positives and false negatives). In order to assess the coverage of a given tool, the set of all relevant vulnerabilities and the set of false positives for the scanned images must be established.

Given that the scope of this thesis is to comparatively analyze available scanning tools and not the efficacy of static vulnerability scanning in general, it will be assumed that there is no such vulnerability that none of the scanners could find. This means that the set of all relevant vulnerabilities will be assumed to be equal to the set of all unique vulnerabilities that all the scanners combined could find.

The set of false positives in this study are established by examining all positives found in a subset of the images used in testing. Positives that are obviously not applicable are regarded as false positives. Such vulnerabilities may be related to an OS or architecture which is not used in a given container. While false positives in practice are all such vulnerabilities which are not applicable to the specific configuration and use case of a container, such false positives can not be assessed during static analysis and thus will not be considered in this research.

The usability of the tools will be scored subjectively based on ease of set-up, ease of use, and the ease of integration into a CI/CD workflow. The usability testing and comparative analysis of the results of experimentation are combined to comprehensively evaluate the chosen image scanners and to give a recommendation as to which tool is best suited for the task.

2. Existing Solutions and Related Work

Existing research into this topic mainly focuses on the fact that most available Docker container images in public repositories have numerous known vulnerabilities. Malhotra et al. looked at 30 Dockerhub's official and verified images [8]. Their research focused on the general security posture of publicly available images. Although the results from different scanners were compared, possible reasons for variations between them were not discussed. The authors specifically name the discrepancies between different scanners' results a topic that needs further investigation in future research.

Mills et al. created a custom tool OGMA [9]. OGMA combines the results of six scanners into one report and helps with the visualization of the vulnerability landscape. The tools used within OGMA were Clair, Dagda, Docker Scan, Grype, Sysdig, and Trivy. The discrepancies of the scanning results are attributed to different vulnerability sources that each scanner uses. While the custom tool showed promising results, the use of such custom tools in an enterprise setting is dubious at best given the intermittent maintenance they receive (if any) from their creators. The testing procedure involved 25 popular container images.

In another paper, the same research group (Mills et al.) compiled a longitudinal assessment of Docker images over time, scanning 380 container images over three testing periods [10]. Their focus was on how the set of vulnerabilities changes over time, are new vulnerabilities added more often than they are patched. The issue of differing results of the scanners was avoided by the use of the custom tool OGMA. Their findings indicate that container images tend to grow in size over time and the added components tend to expand the set of vulnerabilities present. They also found that Debian-based container images had consistently more vulnerabilities than any other types of images. As testing was carried out using OGMA, the usability and efficacy of each individual scanner was not discussed.

Jain et al. give a general overview of the threat landscape in the context of container images [11]. Different tools and their use cases are discussed in relation to mitigating the identified attack mechanisms. The tools themselves are not compared or used in practice, no actual static vulnerability scanning was conducted within their research.

Javed et al. compared four static vulnerability scanners in terms of coverage and accuracy [12]. The core of their research was the fact that static scanners are better at identifying OS package rather than application package vulnerabilities. They scanned 59 Java application containers with Clair, Anchore, and Microscanner. They also obtained the source code for all the 59 Java applications and scanned the code with SpotBug's security plugin to find Java-related vulnerabilities that the static scanners did not find. Coverage was assessed by comparing the application packages that the scanners could identify with the application packages that the SpotBug plugin could identify. The accuracy or Detection-Hit Ratio was calculated by dividing the number of vulnerabilities found by a given tool with the total number of vulnerabilities identified by all the tools and the SpotBug's plugin combined.

While their approach illustrates one of the shortcomings of static scanning in general, the practical usability of the tools and extra features they might have are not discussed.

Research conduced by Andres Pihlak compared different static and dynamic vulnerability analysis tools available at the time [13]. While the discrepancies of the results of static vulnerability scanning were pointed out, the reasons which may have caused them were not discussed. The use and efficacy of dynamic scanning tools was found to be highly dependent on the specific software stack onto which it is implemented. Due to this, no specific recommendation of a dynamic scanning tool was given.

Majumder et al. proposed a custom scanning tool that combines the results of Grype, Snyk, Trivy, and CVE-Bin-Tool for static vulnerability scanning [14]. The CVE-Bin-Tool was used to identify packages that were present in the 111 images that were scanned, but were missed by the three static vulnerability scanners. Similarly to Javed et al., this research focused on non-OS or application package vulnerability detection. The available scanners were not compared in terms of practical usability. In terms of practical usability in an enterprise setting, the proposed custom tool is similar to OGMA meaning the same criticisms apply.

The usability of Grype, Trivy, and Snyk was investigated through heuristic evaluation by Kim et al. [15]. Tests were conducted in which a set of users were tasked with assessing the usability of the three scanners. The usefulness of presented information, filtering capabilities, and general readability of the results were compared. This research did not mention the technical prowess of the tools nor did it include any evaluation of the ease of use in an enterprise setting (integration into CI/CD or other workflows).

Vollmer et al. analyzed static and dynamic vulnerability scanning tools with the focus of identifying the general shortcomings of both approaches [16]. While the results of static scanning tools were compared, the possible reasons behind it were not mentioned. No evaluation of practical usability was given either. The main idea conveyed by this research is the fact that both static and dynamic vulnerability scanning have significant shortcomings and further research and better tools are needed.

The previous research presented here is up to five years old. Given the growing popularity of containerization as a whole and the security research therein, the set of possible tools to use for static vulnerability scanning is changing rapidly. Some of the research mentioned here could already be considered out-of-date as at least some of the tools discussed within them have reached end-of-life and have been replaced with newer options [11, 10, 12, 9, 13, 15].

3. Background

Containerization and process isolation in and of itself is not new. Early developments in this regard can be traced back to the 1970s. At that time, this was limited to sandbox-like segregated environments. This was to ensure that applications could be used or tested without interfering with other processes on the same host machine.

The popularity of containerization really started with the introduction of Docker in 2013. Docker is an open-source platform which enables developers and system administrators to build and run applications in a containerized fashion.

Container can be thought of as a lightweight form of a virtual machine that packages together an application and all its dependencies. Dependencies include libraries, system tools, and configuration files that an application needs during use. This allows an application within the container to run consistently across different environments, be it on a developer's computer, a testing environment, or a production server.

Docker's efficiency comes from its use of OS-level virtualization. Unlike virtual machines, which need a separate guest operating system, Docker containers share the host machine OS kernel. This shared kernel method reduces resource overhead and enables multiple containers to run on the same host. Figure 1 illustrates the architectural difference between virtualization and containerization.

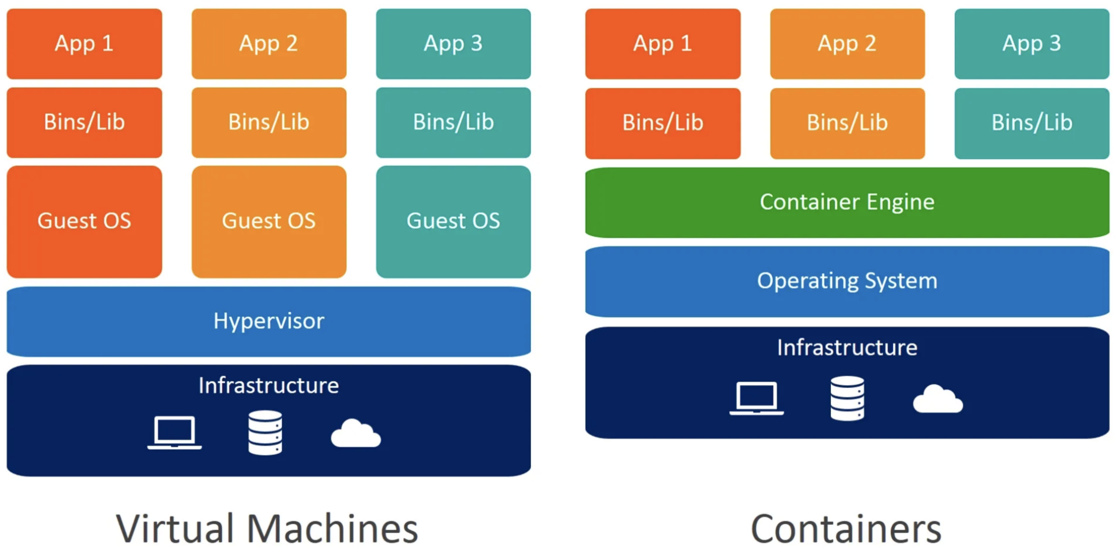

Figure 1. *Virtualization Vs. Containerization* [17]

Docker uses Linux kernel namespaces and cgroups features to keep containers isolated. Namespaces are used for isolating the view of process IDs, network interfaces, and file systems. Cgroups are used for allocating and limiting resources like CPU and memory for each container.

Docker container is in essence a stack of layers. Bottom or base layer is the OS layer onto which the rest of the container is built. Each layer may add or remove any parts of the previous layers. The top layer or the container layer is the only writable layer which allows ephemeral changes to be made during runtime. Figure 2 illustrates the layered architecture of a Docker container using an Alpine Linux based Apache container as an example.

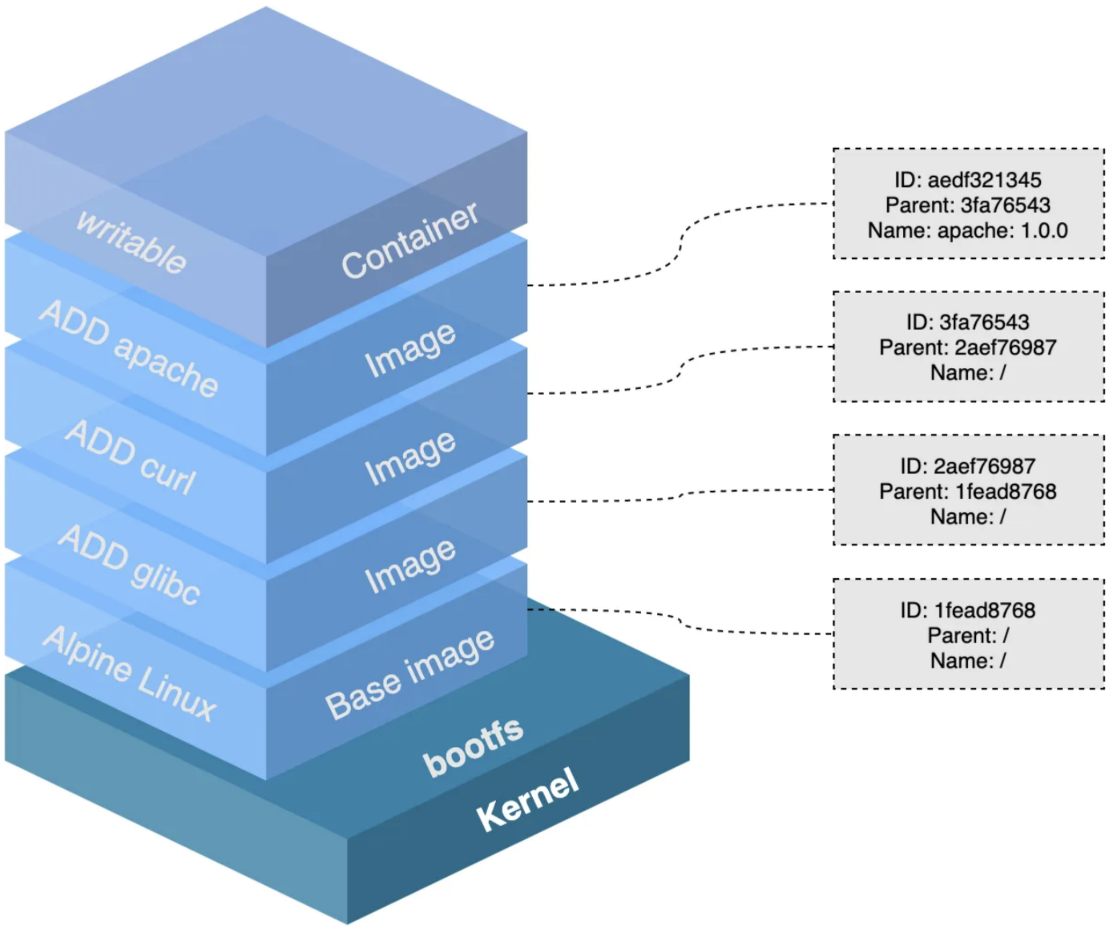

Figure 2. *Layered Architecture* [18]

Docker's two main concepts are containers and container images. A Docker container image is an immutable blueprint of a container that describes all the layers needed to set up a given container. This includes the base operating system layer and application code layers. A container is a running instance of a given container image.

3.1 Supply Chain Issues

Consider the following scenario: an IT department of an organization uses some standard retail software in their daily workflow. The developer of said software has included other, third-party components into their product which is a common practice in the industry. In some cases, this chain of component inclusion may even involve fourth- or fifth-parties. The true breath of software in use is obscured for the end user by this chain of supply. Without explicitly analyzing the supply chain, the end user essentially leaves their security in the hands of every party further up the chain. This leads to different types of issues, one of which are explained in the following real-life example.

The Log4Shell or Log4j vulnerability was discovered in November 2021. The name comes from the Apache Log4j logging library which it affects. This vulnerability is considered catastrophic by researchers as it is widespread and allows hackers to take total control of systems using a vulnerable version of Log4j [19]. The 34% increase of vulnerability exploitation between 2020 and 2021 is mainly attributed to this vulnerability [20].

It is estimated that it will take at least a decade to fix every vulnerable instance of Log4j due to its widespread use and inadequate patching processes [6]. The issue for software end users in this case is that without knowing the complete list of components included in the software that they use, it is difficult to determine if Log4j affects them, and if so, to patch all vulnerable Log4j instances. The ubiquity and severity of this vulnerability drew wide attention from the industry to vulnerability scanning and supply chain issues in general.

4. Vulnerability Scanners

The tracking of known vulnerabilities dates back to 1999. That is when the CVE or Common Vulnerabilities and Exposures program started. In September 1999, the original 321 CVE records were created [21]. Having a list of known vulnerabilities is only half the battle though - finding and patching them is a whole other issue.

Vulnerabilities can be introduced into a Docker container at any level. Vulnerabilities may emerge in base images or in any other layers which a container consists of. In order to find a vulnerability, one must know which components are used in the system that they are tasked with keeping secure. The problem of keeping track of all those packages, libraries, or other dependencies which may be vulnerable is what led to the development of vulnerability scanners.

Static vulnerability scanners analyze each layer in a Docker image by examining its contents — such as system libraries and dependencies — for known security issues. By scanning each layer, these tools help identify vulnerabilities introduced at any stage of the container's build process.

When a component is identified in a container image, scanners use either its CPE, SWID, or PURL value to find relevant vulnerabilities from a vulnerability database. Although CPE is considered deprecated, it is still widely used in the software industry today [22].

The selection of container image scanning tools is based on what has been used in previous research and what is currently being used by the community. As mentioned before, some of the tools used in previous research are end-of-life but some of those have an official successor in which case the successor is chosen. The comparison of the chosen scanners on the basis of OS and language-specific package support and vulnerability database support can be found at the end of this chapter.

4.1 Anchore Engine

Anchore Engine is a Docker container scanner that runs in a Docker container itself which can be used as standalone or within an orchestration platform such as Kubernetes, Amazon ECS, or others. While it does support a wide range of operating systems and programming languages, it is no longer maintained since 2023 and no future versions will be released.

As such, it will be excluded from testing done in this thesis. Since users seeking similar functionality are advised to use Grype, that will be tested instead [23].

4.2 Clair

Clair is a vulnerability scanner made by ProjectQuay. Clair runs on Docker containers and is split into multiple microservices: indexer, matcher, notifier, and a PostgreSQL database. The first three can be run as a single combined container.

After the initial startup, vulnerabilities are fetched from different sources specified in the configuration file. By default, no vulnerability sources are explicitly specified and information is thus pulled from all supported sources. Populating the PostgreSQL database with fetched vulnerabilities takes 10-30 minutes depending on available bandwidth. After the database is populated, locally available images can be scanned.

Results are printed into console by default but can be saved as a JSON or an XML file. Clair has the added functionality of remembering which components have been present in previous scans and notifies the user when an applicable vulnerability emerges at a later date [24] [25]. A webhook can be configured to enable this functionality.

The tests were run on Clair v4.8.0 released on October 9th, 2024. Clair is distributed under the Apache 2.0 license which allows both personal and commercial use.

4.3 Dagda

Dagda is a tool that along with static vulnerability scanning can perform anomaly detection on containers during runtime as well. On startup, Dagda populates an included MongoDB database with vulnerability information. This database is used for finding vulnerabilities during image scanning [26].

The latest version of Dagda was released in July 2021. It is not compatible with the latest versions of Red Hat or Debian based Linux distributions and thus will not be tested in this research.

4.4 Trivy

Trivy is an actively maintained vulnerability scanner that can also scan remote Git repositories, VM images, Kubernetes clusters and AWS configurations for CVEs, IaC issues, secrets, and software licenses [27].

Trivy runs as a single binary or a Docker container. It is also available through RHEL/Debian package managers for easy installation in Linux environments. The vulnerability database is downloaded automatically from GitHub Container registry (GHCR) and is updated every six hours. [28].

Testing was done on Trivy version 0.55.2 released on September 17th, 2024. Trivy is distributed under the Apache 2.0 license.

4.5 Snyk

Snyk is an umbrella term encompassing a number of different security software. Most relevant to this thesis are Snyk Open Source (SCA) for scanning dependencies and Snyk Container for base image security. Specific technical information on either of those products is difficult to find. There are severe limitations imposed on the free personal tier and pricing for team and enterprise tiers is not explicitly provided. For these reasons, it will not be included in the testing and comparison carried out in this thesis.

4.6 Grype

Grype is the successor to Anchore Engine's static vulnerability scanning functionality. Grype is a rather focused tool with little extra functionality apart from scanning container images or filesystem paths for vulnerabilities. In order to scan a container image, the included Syft tool is used for creating an SBOM. While Grype allows other SBOM creation tools to be used, Syft is the included and recommended method for best results [29].

Installation of Grype is relatively simple using an installation script available on GitHub. Vulnerability information is fetched automatically during runtime and the data in it is updated every 24 hours [29].

Testing was done on Grype version 0.80.2 released on September 24th, 2024. Grype is distributed under the Apache 2.0 license.

4.7 Docker Scout

Docker Scout is a vulnerability scanning tool included with Docker Desktop or can be used separately as Scout-CLI which allows local and repository scanning. Docker Scout is not included with Docker Engine and is not under the same Apache 2.0 license. Docker Scout is free only for personal and small business use (fewer than 250 employees and less than $10 million (USD) in annual revenue). Paid subscriptions are also required for government entities of any size. This excludes a considerable portion of the possible organizations to which this research is aimed at. For this reason, Docker Scout will not be tested in this thesis [30, 31].

4.8 Dependency Track

OWASP's Dependency-Track (DT) is a component analysis platform which helps keep track of vulnerabilities, policy compliance, software licenses and outdated components. DT consists of an API server and a frontend service which can be deployed on Docker or on a Kubernetes cluster using a Helm chart. Integrations with Trivy, VulnDB and Snyk analyzers can be enabled, though VulnDB and Snyk are both subscription services offered by third parties [32].

One drawback at least for testing purposes is the fact that DT requires SBOMs to be created beforehand. During testing, Syft was used as the SBOM creator which is also used by Grype, although automatically. To initiate a scan, an SBOM must be uploaded through the API after which the results can be found using the web interface. Scanning results can also be downloaded from the API which simplifies the integration process for CI/CD pipelines.

The web interface gives an easy-to-read overview of all scanned container images, scanning results, and result changes over time. The user is notified when a new vulnerability emerges for an existing, previously scanned component. This gives the user a comprehensive up-to-date overview of the current security posture, even without repeated scanning.

Testing was done on Dependency Track v4.11.5 released on August 14, 2024. Dependency Track is distributed under the Apache 2.0 license.

4.9 jFrog X-ray

jFrog X-ray is a vulnerability scanner built into jFrog's Artifactory repository software. X-ray compares scanned images to vulnerability data found in OSS index which includes data from NVD, GitHub, Ubuntu, Debian, Red Hat and PHP advisories. jFrog claims to give better insight into binaries present on a scanned image than the competitors.

While jFrog's X-ray would be relatively easy to use for those already using jFrog's container registry software, the X-ray functionality is only available for a few of the top subscription levels. Licensing and the tight integration with Artifactory which some organizations may not already use are the reasons for excluding this product from this research [33].

4.10 Package and Vulnerability Database Support

The following list of OS packages are supported by all four chosen scanners:

- Alpine
- Amazon
- Busybox
- CentOS
- Azure
- Debian
- Distroless
- Oracle
- SUSE
- RHEL
- Ubuntu
- Wolfi
- Chainguard
- Alma
- Rocky
- Photon

Table 1 shows which language-specific packages each of the chosen scanners support:

Language	Clair	Trivy	Grype	DT
Ruby	●	●	●	●
Java	●	●	●	●
JavaScript		●	●	●
Python	●	●	●	●
.NET		●	●	●
Golang	●	●	●	
PHP		●	●	●
Rust		●	●	●
Elixir		●		●
C/C++		●		
Dart		●		
Swift		●		
Julia		●		

Table 1. Language-Specific Analysis Support

Table 2 describes the supported vulnerability databases for each chosen scanner. It should be noted, that many of the databases shown contain overlapping data. OS-specific vulnerability databases, for example, contain vulnerabilities found in NVD/CVE databases but that relate to the given OS. In rare cases, the OS-specific database can have entries that are not yet listed in the general CVE database, but will be listed there eventually. NVD database is closely related to the general CVE database, but includes more specific and practical information about any given vulnerability. The general CVE database is where a vulnerability gets its CVE identifier which helps the industry reference and address known vulnerabilities consistently.

Database	Clair	Trivy	Grype	DT
Ubuntu	●	●	●	
Debian Security Tracker	●	●	●	
RHEL	●	●	●	
SUSE	●	●	●	
Oracle	●	●	●	
Alpine SecDB	●	●	●	
VMWare PhotonOS DB	●	●		
AWS	●	●	●	
OSV DB	●	●		●
GitHub Advisories		●	●	●
CVE/NVD		●		●
Chainguard SecDB			●	
Wolfi SecDB			●	
OSS Index				●

Table 2. Supported Vulnerability Databases

5. Testing

Each scanner was downloaded from their respective official source. Clair's source code was downloaded from Quay's Clair GitHub repository and compiled locally [34]. Both of Dependency Track's API server and frontend components were downloaded from DT's website and deployed as a single Docker Compose file [32]. Grype's binary was downloaded from Anchore's official Grype GitHub repository [29]. Trivy's binary was sourced from Aquasecurity's official Trivy repository [27]. Tests were conducted on a virtual machine with eight processing cores and 16 GB of RAM running Oracle Linux 9 operating system. Table 3 describes each scanner's version and release date that was used during testing. The list of images that are used for scanning can be found in the references of this thesis [35]. A total of 387 images will be scanned.

Scanner	Version	Release Date
Clair	4.8.0	2024/10/09
Dependency Track	4.12.0	2024/10/01
Grype	0.80.2	2024/09/24
Trivy	0.55.2	2024/09/17

Table 3. Tested Versions

Scanning results shown later in this thesis are from the last week of November 2024. This was done to ensure that the set of possible vulnerabilities would be as similar as possible for each scanner. Some vulnerabilities may be added and testing different scanners at vastly different times may cause discrepancies in the results.

Initial testing was done with each scanner in its default configuration. The only exception to this is Dependency Track for which the OSS Index Analyzer was turned on. This is the recommended option as per DT's documentation for scanning dependencies and not just operating systems and applications [36].

5.1 Data Collection Procedure

For Clair, Grype, and Trivy, testing was carried out by automated scripts which instructed the scanners to report vulnerabilities of each image as a JSON file. The findings from each scanner were then compiled into CSV files for later processing. Since Dependency Track scans an SBOM file and not the image itself, an SBOM creation tool was needed. Initially, Syft was used to generate SBOMs of each image in XML format conforming to

CycloneDX standard version 1.6. Syft is the recommended and included SBOM creation tool for Grype and supports the CycloneDX standard which Dependency Track requires.

The SBOMs created by Syft were uploaded to Dependency Track's API endpoint which initiated the scanning procedure. Results of each scan were then downloaded from DT's API. Dependency Track presents findings in its native FPF or Finding Packaging Format which is essentially a JSON file with specific fields tailored for representing vulnerability findings. Similarly to the other scanners, findings from those FPF files were compiled into a CSV file for later processing.

The CSV files compiled from the results of each scanner were then aggregated into an Excel spreadsheet for statistical comparison.

5.2 Configuration Options

Clair's configuration options that would affect the results were limited to reducing the number of vulnerability sources that it uses as all of them are used by default. This would have had a negative effect, if any, on the coverage of the scanner so this option was not used.

Similarly to Clair, Grype also uses all of its available vulnerability sources by default, so the range of sources can not be improved by custom configuration. While Grype does support searching for vulnerabilities by analyzing SBOMs created by software other than Syft, it is officially recommended to use the included SBOM creator for best results [37].

Same is true for Trivy. While it allows for SBOMs created by other tools to be used for scanning, the official documentation warns against it. This is because using alternative SBOM creation tools may result in inaccurate vulnerability detection due to custom properties Trivy includes in SBOMs when creating them [38].

Dependency Track however, has a wider range of scanning configuration options. Different analyzers that compare the uploaded SBOMs with data from vulnerability databases can be enabled. Dependency Track supports Sonatype's OSS index, Snyk, VulnDB and Trivy analyzers. OSS Index was already enabled in the first round of testing as this was the recommended minimal configuration for finding vulnerabilities in dependencies. Snyk and VulnDB analyzers are commercial services and were excluded from testing due to requiring a paid subscription. Trivy analyzer can be enabled with relatively little effort as it only consists of adding Trivy service to the Docker compose file used to run Dependency Track in the first place. When enabled, Trivy will run alongside of Dependency Track as a third container and its API endpoint will automatically be used by Dependency Track when analyzing an SBOM.

Officially approved configuration options that would widen the coverage for Clair, Grype, and Trivy are lacking. For this reason, the next phases of testing only affect the results of Dependency Track.

5.3 Testing Custom Configurations

In the second phase of testing, the Trivy analyzer is enabled for Dependency Track. It should be noted that the SBOMs used are still the ones from the first phase, created by Syft and in CycloneDX 1.6 (XML) format. For best results Dependency Track recommends using Trivy as the SBOM creator when Trivy analyzer is enabled, but using Syft's SBOMs allows for separate analysis of the effect that Trivy's added vulnerability sources may have on the results.

For the third phase, Trivy was used to create the SBOMs that were uploaded to Dependency Track for analysis. The SBOMs were still conforming with CycloneDX 1.6 standard but now in JSON format as Trivy does not currently support the creation of CycloneDX (XML) SBOMs. The uploading of SBOMs and downloading of the results was procedurally identical in all three phases of testing.

5.4 Finding False Positives

Vulnerabilities that are found but are not actually exploitable due to a given container's use case should be considered as false positives in practice. Although the static nature of the analysis carried out in this thesis does not allow counting such vulnerabilities, some obviously inapplicable vulnerabilities could still be counted as false positives. This allows for some false-positive rate to be given to each scanner without knowing the particular use case of any of the container images scanned.

A set of 20 images will be randomly chosen from the test set. All vulnerabilities present in those 20 images will be analyzed individually to assess whether or not a given vulnerability could be considered a false positive without knowing the exact use case of the containers. When the set of found but inapplicable vulnerabilities is established, a percentage will be calculated for each of the scanners to show how many of the findings were known inapplicable findings.

6. Results

The results presented here, for all three testing phases, are based on data from which duplicate image-vulnerability combinations have been removed and vulnerability aliases have been replaced with their respective "CVE-" counterparts if available. A more detailed discussion on duplicate entries and aliases can be found in the duplicate entries and aliases section.

In the initial testing phase, 108706 unique image-vulnerability combinations were found in 378 container images. Nine images were considered vulnerability-free by all the scanners.

Clair identified the largest amount of vulnerabilities, 63091. Trivy found 61266, Grype 50661, and Dependency-Track 16601 vulnerabilities. Although Clair identified more vulnerabilities than other scanners, Grype had the largest coverage for most base operating systems. Debian is the only base OS for which Clair had the best coverage. For other popular base OS's such as Alpine, SLES, Ubuntu, busybox and Red Hat, Grype had the best coverage.

To help with the readability of the results, the coverage percentages in the tables 4 and 5 are given in relation to the final total number of vulnerabilities for each base image. This final set includes vulnerabilities that were not found in the first testing phase and were only identified in the second or third testing phases. This is the reason why Clearlinux is present in table 4, although none of the scanners could identify any vulnerabilities in that image during the first testing phase.

Base OS	No. of images	Clair	DT	Grype	Trivy	Best
Debian	147	53%	8%	29%	45%	Clair
Alpine	71	38%	38%	66%	49%	Grype
Ubuntu	67	37%	16%	42%	41%	Grype
Busybox	21	25%	54%	70%	53%	Grype
unknown	19	19%	61%	76%	54%	Grype
Red Hat	18	50%	20%	68%	64%	Grype
SLES	17	16%	24%	55%	29%	Grype
Amazon	5	17%	11%	39%	53%	Trivy
Almalinux	3	2%	4%	74%	3%	Grype
Fedora	2	0%	5%	95%	0%	Grype
CentOS	2	0%	10%	89%	90%	Trivy
Oracle	2	6%	18%	49%	45%	Grype
Cirros	1	0%	81%	96%	0%	Grype
Arch	1	23%	46%	0%	46%	DT/Trivy
Wolfi	1	14%	21%	21%	14%	DT/Grype
Photon	1	0%	50%	0%	0%	DT
ClearLinux	1	0%	0%	0%	0%	N/A

Table 4. Coverage per Base OS, Phase 1

All scanners were able to complete the scan on each image tested, but as previously mentioned, nine images did not have any vulnerabilities identified by any scanner.

The second and third phases of testing resulted in one additional vulnerable image being found. This is the previously mentioned ClearLinux image. Note that abbreviations "DT2" and "DT3" in table 5 indicate the second and third phase configurations of Dependency Track respectively. Other scanners' configuration has not changed since the first phase.

Base OS	Total vuln.	Clair	DT	DT2	DT3	Grype	Trivy	Best
Debian	91128	53%	8%	16%	85%	29%	45%	DT3
Alpine	5819	38%	38%	51%	55%	66%	49%	Grype
Ubuntu	23776	37%	16%	24%	58%	42%	41%	DT3
Busybox	1246	25%	54%	57%	57%	70%	53%	Grype
unknown	1415	19%	61%	64%	64%	76%	54%	Grype
Red Hat	5769	50%	20%	28%	70%	68%	64%	DT3
SLES	2586	16%	24%	31%	33%	55%	29%	Grype
Amazon	388	17%	11%	12%	18%	39%	53%	Trivy
Almalinux	665	2%	4%	4%	30%	74%	3%	Grype
Fedora	1390	0%	5%	5%	5%	95%	0%	Grype
CentOS	1047	0%	10%	87%	87%	89%	90%	Trivy
Oracle	283	6%	18%	19%	58%	49%	45%	DT3
Cirros	26	0%	81%	85%	85%	96%	0%	Grype
Arch	13	23%	46%	46%	46%	0%	46%	DT(X)
Wolfi	14	14%	21%	64%	79%	21%	14%	DT3
Photon	2	0%	50%	50%	50%	0%	0%	DT(X)
ClearLinux	1	0%	0%	100%	100%	0%	0%	DT2/3

Table 5. Coverage per Base OS, Phases 2 and 3

A total of 134954 unique image-vulnerability combinations were found in 379 images. DT2 identified 29009 and DT3 102422 unique image-vulnerability combinations. Eight images tested had no vulnerabilities present according to every scanner. The final coverage per scanner was as follows: Clair 47%, DT 12%, DT2 21%, DT3 76%, Grype 38%, and Trivy 45%.

The change from DT to DT2 was to add Trivy as another vulnerability analyzer while not changing the SBOMs that were used as input. These results show that this change increased coverage from 12% to 21%. Changing the SBOM creation tool from Syft to Trivy (from DT2 to DT3) increased coverage again from 21% to 76%. This indicates that while both aspects are important, more detailed SBOM creation influences the results more.

6.1 Image OS and Size

Table 6 compares the base OS's of the scanned images based on average size and the average amount of vulnerabilities identified. Base OS's which had fewer than five images present in the tested set were excluded as it is difficult to draw meaningful conclusions from smaller sample sizes.

Base OS	Avg. size (MB)	Avg. vuln.	Sample size
Ubuntu	690	355	67
Red Hat	460	321	18
Debian	458	620	147
Amazon	420	78	5
SLES	386	152	17
Alpine	231	81	71
Busybox	54	59	21

Table 6. Average Sizes and Vulnerabilities

It may be hypothesized that larger images should, as a general rule, contain more vulnerabilities considering that larger images tend to have more packages and dependencies within them for which known vulnerabilities may be found. Data gathered during these tests shows that the correlation between image size and the amount of vulnerabilities found is 0.28. Figure 3 describes the correlation in more detail.

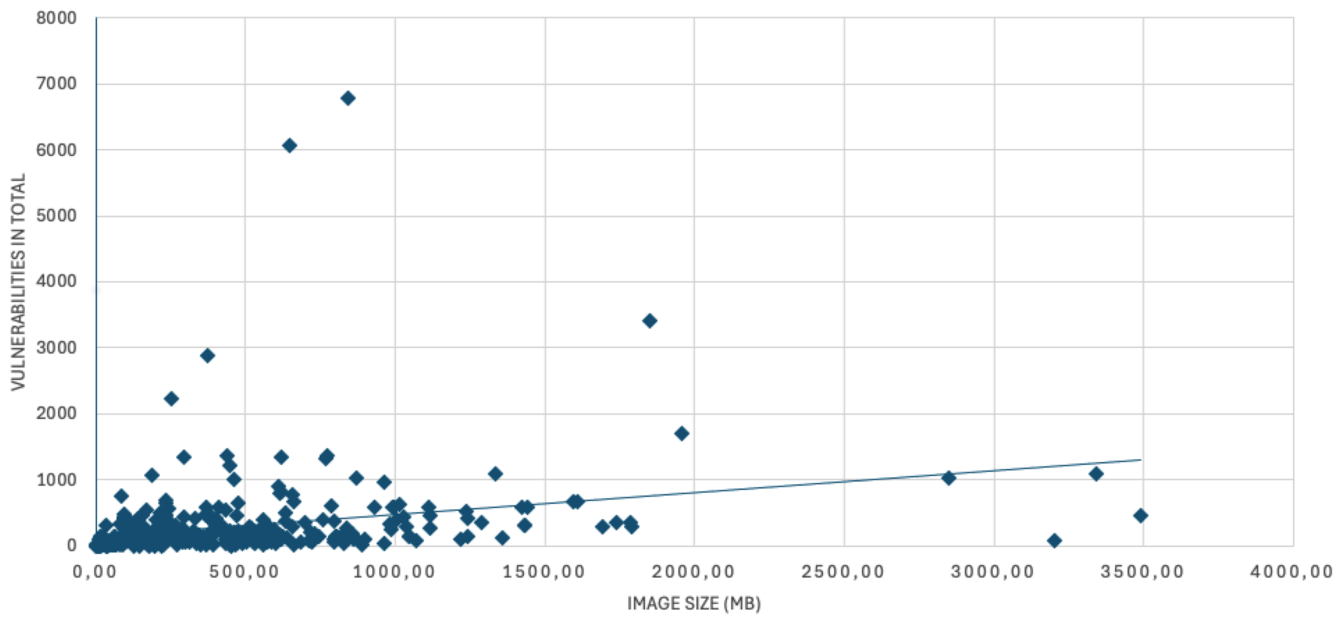

Figure 3. *Distribution of Image Sizes and Number of Vulnerabilities*

6.2 Duplicate Entries and Aliases

In this thesis, an entry is considered a duplicate if the same vulnerability is reported on the same container image more than once. The results from tests conducted in this thesis show a large variance between the scanners, as shown in table 7.

Metric	Clair	DT	DT2	DT3	Grype	Trivy
Total vulnerabilities reported	74732	22402	33724	229560	108386	106901
Without duplicates	63091	16601	29009	102422	50661	60703
Percentage of duplicates	16%	26%	14%	55%	53%	43%

Table 7. Duplicate Entries

In these tests, it was never the case that the same vulnerability for the same component of an image was reported more than once. Sometimes however, SBOM creation tools separate some packages into smaller pieces which is technically the correct thing to do, but such separation may have little real-world value. Take curl for example, a command line tool for transferring data with URLs. Curl depends on a library called libcurl without which curl does not work. The same library can be used by other programs as well. When a vulnerability is discovered in libcurl, the vulnerability is also appended to curl since curl uses the same library. The result of cases like this is that libcurl's vulnerability CVE-2024-6874, for example, is reported twice on every image that has curl installed - one instance for curl and another for the libcurl library. Such double-reporting is more prevalent when Trivy SBOMs are used.

Vulnerabilities with identifiers such as "GHSA-..." or "SUSE-SU..." can sometimes have aliases in the more common form "CVE-...". Aliases are not an issue within the results of a single scanner since one scanner shows results which have aliases as either "CVE-..." or another form, not both at the same time. This is only an issue when aggregating results across many scanners since one may identify a vulnerability as "GHSA-..." and another may use "CVE-..." while conveying the same exact vulnerability. Dependency Track handles aliases automatically when aggregating results across more than one vulnerability analyzer. When aggregating results in this study, all alternative identifiers were replaced with their corresponding "CVE-..." identifier where available.

6.3 False Positives

Through the manual analysis of the vulnerabilities found in 20 randomly chosen images, 13 vulnerabilities were identified to be inapplicable vulnerabilities and as such, false positives. 8 were Windows-related, 3 were only applicable to 32-bit environments, 1 was only for ARM, and 1 for PowerPC architectures. Considering that all tested container images use

Linux as their base OS, are all for x86 architecture, and will probably be used on 64-bit hosts, such vulnerabilities can be assumed to be false positives. Since a scanner cannot know whether a 32 or 64-bit host will be used, finding 32-bit vulnerabilities cannot be considered a mistake on the scanner's part. For this reason, the 3 32-bit vulnerabilities will be excluded from the false positives list, although in practice they could be discarded (if all container images are run in 64-bit environments, of course).

Vulnerability	Clair	DT3	Grype	Trivy
CVE-2022-41720	0	92	9	91
CVE-2022-41716	0	87	8	86
CVE-2022-41722	0	98	10	97
CVE-2023-45283	16	147	14	146
CVE-2023-45284	16	147	14	146
CVE-2024-5321	29	29	0	29
CVE-2023-47039	49	1	7	0
CVE-2023-4807	0	7	10	0
CVE-2023-1255	13	19	17	13
CVE-2023-6129	35	42	43	36
Total	158	669	132	644

Table 8. False Positives

In table 8, the first 8 rows are Windows-related vulnerabilities, and the last two are ARM and PowerPC-related respectively.

Table 9 compares the total number of vulnerabilities found by each scanner with the amount of known false positives found by each scanner.

Metric	Clair	DT3	Grype	Trivy
Total vuln. found	63091	102422	50661	60703
False positives	158	669	132	644
% of false positives	0.25%	0.65%	0.26%	1.06%

Table 9. False Positive Percentages

The scanner most prone to giving false positives seems to be Trivy. DT3 (which includes the Trivy analyzer) found around 1.7 times the amount of vulnerabilities in total while finding almost the same amount of false positives as Trivy alone.

6.4 Fixed Vulnerabilities

Most effective usage of static vulnerability scanning relates to identifying vulnerable components that have a newer version available in which the vulnerability is already fixed. Out of all the vulnerabilities identified in this research, 49% are for components which have a fixed version available. More importantly, 79% of all "high" or "critical" severity vulnerabilities have a fix available. Table 10 lists a breakdown of the percentages of fixed vulnerabilities by severity.

Severity	% fixed
Unknown/Negligible	83%
Low	16%
Medium	54%
High	78%
Critical	87%

Table 10. Vulnerabilities Fixed

It is possible that upgrading to a newer version of a given component that has fixed the vulnerabilities currently present may introduce new vulnerabilities which do not yet have a fix available. In terms of security posture, it is still generally advised to keep software and dependencies up to date. Such situations can be dealt with in a case-by-case basis when required.

One example of a vulnerability that has a fix available is CVE-2021-45046. This is the original Log4j vulnerability discussed in the background section of this thesis. This CVE was identified to affect one of the tested images which demonstrates the widespread nature of Log4j. The additional vulnerabilities exposed by the consecutive patches of Log4j (CVE-2021-45105 and CVE-2021-44832) were also present in the testing results [19]. All three vulnerabilities were patched within a month of the original Log4j vulnerability, yet they are still found in images in use in 2024.

6.5 Validation

For validation purposes, the list of vulnerability scanners and their respective versions and scanned image names and hash values are provided in the testing section of this thesis. It should be noted that the time-sensitive nature of vulnerability scanning makes validation difficult. The time-sensitivity of this work stems in part from the fact that the exact versions of the scanners themselves may no longer be available. Additionally, new vulnerabilities may emerge for the same set of images at a later date. Also, the specific versions of the images scanned may not be available when validation is attempted.

7. Usability

Grype and Trivy were the simplest to install and use. Both can be downloaded and used as a precompiled binary and initiating a scan is achieved by running a single command. The output of a scan can be written to a file or presented as a table in the command line interface.

The installation and usage of Clair is somewhat of an elaborate procedure. First a PostgreSQL database needs to be deployed using an included docker compose file. Then, Clair's binary needs to be compiled from the included source code. After starting Clair, a connection to the PostgreSQL database is made and the database will be filled with vulnerability information. This step took around 15 minutes to complete.

When the vulnerability data is fetched, Clairctl can be used to interact with the previously started Clair instance to scan images locally. Clairctl can be downloaded as a binary or compiled locally. The source code for it is included with Clair. Results of a scan can be written to a file or shown in the command line interface.

The deployment of Dependency Track is of similar complexity. Both the API server and the front end component are deployed using a docker compose file. Although an embedded H2 database is enabled by default, it is not intended for production use. PostgreSQL is the recommended database solution although MS SQL and MySQL are also supported.

As previously mentioned, an SBOM needs to be created with another tool. SBOMs can be uploaded to Dependency Track either through the API or via a web interface. Results can be viewed in the web interface, downloaded from the same interface, or downloaded through the API.

7.1 CI/CD Integration

Integration into a CI/CD pipeline can be achieved in various ways depending on the scanner in question. As Clair's initial startup took around 15 minutes, it is not feasible to spin up Clair from scratch every time a build pipeline runs. For CI/CD purposes, Clair itself should run separately from the pipeline. Scanning within the pipeline can then be achieved by running only Clairctl which connects to the separate Clair instance.

As Clair does not support SBOM scanning, a built image must be uploaded to a registry of some sort, so Clair can access it, index the components and run the scan. Clair or Clairctl does not include native functionality for failing a pipeline when certain user defined criteria are met. For example, to fail a pipeline when any vulnerability or a vulnerability of a certain severity is encountered, some custom scripting is needed to analyze the report Clairctl returns. Such scripts can be prone to errors and may need manual updating from time to time.

Since Grype can not be used in client-server mode, it needs to be started every time a pipeline runs. While starting Grype is significantly faster than Clair, it is still unreasonable to introduce this to an otherwise fast build pipeline. Grype does have the native functionality for failing a pipeline when a vulnerability of certain severity is found.

Grype can read container image information from local Docker daemon or an external repository, or scan an SBOM created beforehand. If Grype is not provided with an SBOM, Syft is used internally to generate one. Generating one's own SBOMs is useful for developers who wish to ensure their code dependencies are accurately represented in the analyzed SBOM. In such cases, CycloneDX Maven plugin or CycloneDX Python SBOM creation tool can be used for Java or Python projects respectively.

Trivy's client-server mode allows for faster pipelines since the server side which hosts the vulnerability database is hosted separately from the pipeline. This means Trivy server must be set up beforehand, of course. Trivy also has the native functionality of failing a pipeline when a vulnerability of certain severity is found.

To scan an image, Trivy can either be pointed to a registry where the image is present or Trivy's client side can be used to generate an SBOM to be sent to the server for scanning. Trivy also supports CycloneDX and SPDX format SBOMs not created by Trivy which is handy for developers for reasons discussed in the previous paragraph [38].

As Dependency Track already runs as a server, integrating it into a pipeline simply means sending an SBOM to its API. This procedure can be streamlined by using Jenkins's Dependency Track plugin or Dependency Track GitHub action. Otherwise, an SBOM can be sent with a simple curl command.

Failing a pipeline when a certain severity vulnerability is encountered is only available through the use of the previously mentioned Jenkins plugin or GitHub action. In other cases, findings have to be retrieved from the API and custom scripts need to be used to evaluate the results. An SBOM can be created within the pipeline with Syft, Trivy, some language-specific SBOM creation plugin or other tools. The only requirement is that it must conform to the CycloneDX standard, either in XML or JSON format [39].

7.2 Additional Functionality

Clair's additional functionality is fairly limited. In addition to scanning containers and outputting results in console or as JSON or XML files, Clair can notify the user of vulnerabilities that emerge for a previously scanned image. The notifier service can be configured to use a webhook for such notifications. As Clair is not aware which scanned image is the "latest" of its kind, it may notify of vulnerabilities that are not relevant. To avoid this, older index reports must be deleted manually. This is tedious and time-consuming especially in an enterprise environment.

Trivy can scan files inside container images for misconfigurations, secrets, and licenses. Misconfiguration scanning is supported for IaC files found inside a container image, such as Kubernetes YAML or Terraform files. Secret scanning is mainly useful for detecting any credentials that may have been left in environment variables.

Trivy can also scan Kubernetes clusters, more specifically cluster infrastructure components and cluster configuration such as roles and clusterroles. Trivy supports the creation and scanning of KBOMs to detect Kubernetes vulnerabilities. Trivy supports the same range of misconfiguration, secret, and license scanning for projects in local filesystem or local or remote code repositories.

Grype has no support for anything other than vulnerability scanning. Vulnerabilities can be scanned from Docker or Podman daemon, Docker or OCI archives or project files/directories present in the local filesystem.

Similarly to Trivy, Dependency Track can scan SBOMs for licenses, in addition to vulnerabilities. Dependency Track can also alert the user when a new vulnerability emerges for a previously scanned image. Since all SBOMs of similar container images are uploaded to the same project within Dependency Track (repeated uploads for different versions of the same container, for example), it is aware which version of a given container image is the "latest". Because of this, its notifications of vulnerabilities that emerge at a later date are more relevant compared to Clair's. Dependency Track also identifies outdated components even when no vulnerabilities for those components have been found.

For vulnerability management, Dependency Track allows analysis decisions to be attached to vulnerabilities on a per-project basis. This simplifies triaging efforts as vulnerabilities can be marked as "exploitable", "false positive", or "not affected", for example. Much of the noise caused by irrelevant or non-exploitable vulnerabilities can be eliminated through the use of analysis states. Changes to those states are also maintained in an audit history which is highly useful in environments where such security auditing is required.

Dependency Track enables the enforcement of policy compliance. Different policies can be defined in the context of allowed or forbidden licenses, forbidden severities for vulnerabilities, or allowed or forbidden components within container images. Such policies can be applied to all projects across the portfolio or only to specific projects.

Table 11 includes the results of the subjective assessment of the tools. All scores are given out of five.

Aspect	Clair	DT	Grype	Trivy
Ease of set-up	2	3	5	5
Ease of use	2	2	5	5
CI/CD integration	2	4	3	5
Additional functionality	2	5	1	3
Total	8	14	14	18

Table 11. Subjective Assessment

8. Problems

Static vulnerability scanning is currently plagued by a range of different issues. Static scanning tools rely solely on the dependency data present in SBOMs. Sometimes, an SBOM does not include all the relevant dependency data, and thus the scanning results are affected. Even when an SBOM is accurate and all relevant vulnerabilities are found, all of them may not be applicable to the specific use case for which the container image is being validated.

8.1 Applicability Problem

Even when running containers with vulnerable components, the mere presence of that component does not yet pose a security risk. Take CVE-2023-45283 as an example. This vulnerability is related to a few specific versions of golang. It is present on 149 of the images scanned during testing and is considered to be of "high" severity. NIST's description of the vulnerability is as follows: "The filepath package does not recognize paths with a \??\prefix as special. On Windows, a path beginning with \??\is a Root Local Device path equivalent to a path beginning with \\?\. Paths with a \??\prefix may be used to access arbitrary locations on the system." [40]. This vulnerability is entirely irrelevant in this case since all of the images scanned here are based on some form of Linux. This vulnerability and seven other similar vulnerabilities were identified as false positives and used to calculate the false positive rate in the results chapter.

Sometimes, a vulnerability requires specific non-default parameters to be set in order to pose an actual risk. Consider CVE-2023-44981, found in seven of the scanned images. NIST describes it as follows: "Authorization Bypass Through User-Controlled Key vulnerability in Apache ZooKeeper. If SASL Quorum Peer authentication is enabled in ZooKeeper (quorum.auth.enableSasl=true), the authorization is done by verifying that the instance part in SASL authentication ID is listed in zoo.cfg server list. The instance part in SASL auth ID is optional and if it's missing, like 'eve@EXAMPLE.COM', the authorization check will be skipped. As a result an arbitrary endpoint could join the cluster and begin propagating counterfeit changes to the leader, essentially giving it complete read-write access to the data tree. Quorum Peer authentication is not enabled by default." [41]. This vulnerability is considered to be of "critical" severity, but applies only to a rather specific use case.

These examples illustrate that the scanning results, however "high" or "critical" the severity, can not be taken at face value, and every occurrence needs further investigation.

8.2 SBOM Accuracy

SBOM creation tools rely on package managers to discover dependencies in a container image. This is unreliable, as some dependencies could be present in an image as a precompiled binary and not installed as a package through a package manager. In some circumstances, it is recommended to not even have a package manager present in the image for security reasons. In such cases, the dependencies and related vulnerabilities would be difficult to discover.

This issue can be remedied relatively easily for images built in-house by one’s own organization. Several SBOM creation tools are available for use in a CI/CD pipeline for creating accurate SBOMs during image build. When it comes to images pulled from the web, an accurate SBOM may be difficult or impossible to obtain. This could leave blind spots in scanning and expose the organization to supply chain attacks. For container images obtained from trusted outside sources, it is sometimes possible to also obtain a trustworthy SBOM with that image. If the source is trusted, the SBOM created by the initial creator of the image should be accurate enough for proper vulnerability scanning.

9. Practical Considerations

This section is dedicated to giving practitioners some practical pointers on how to deal with problems that may emerge when adopting a static vulnerability scanning tool. Some of these recommendations are applicable to any vulnerability scanner. However, Dependency Track does provide useful functionality for solving some of the problems outlined in this research. These tools are especially effective when considering a large-scale enterprise environment where containers and potential vulnerabilities therein are numerous.

9.1 Initial Steps

As is evident from the results, the volume of found vulnerabilities is rather large. It is essential to practice triaging when working with these scanning results. It is common for developers or even security personnel to get alert fatigue if irrelevant or otherwise non-applicable security alerts are not dealt with.

The best starting point for managing this load of vulnerability alerts is to update the images that are currently in use. As the results show, nearly half of the found vulnerabilities could be dismissed by just updating the software included in an image. When the container images themselves are updated, the packages and dependencies within them are usually updated as well.

For container images fetched from outside sources, updating individual components within them and then building them again is not feasible. When there are hundreds of such containers at play, re-building them individually once and then again for every time they receive an update is an uneconomical endevor. This however, is not as big of an issue for containers already built by in-house developers.

Some of the vulnerable containers may be "in-house" containers as in built by development teams within the same organization. Cooperation with developers is essential to maximize the usefulness of static vulnerability scanning. Developers can update individual components within their containers with ease, barring any incompatibility introduced by version differences.

9.2 Triaging

After the images and components are updated, triaging efforts begin. Consider the two examples given in section 8.1. Finding and dismissing such instances can greatly reduce the number of reported vulnerabilities. Dependency Track offers their "Analysis States" functionality precisely for such situations. Labeling irrelevant vulnerabilities can be done either on a per-project basis or across the organization as a whole. Filtering out environment-related vulnerabilities that do not apply such as the Windows-related example can be immediately dismissed across the whole organization.

Vulnerabilities which are only applicable for certain use cases or configurations are more difficult to assess. Determining whether or not the exact configuration of a given container makes it vulnerable may need to be tested during runtime. Runtime testing is outside the specific scope of static vulnerability scanning, but an avenue of exploration worth considering.

Another method is filtering vulnerabilities by severity level. In the early stages of the triaging process it may prove useful to focus only on "high" or "critical" severity vulnerabilities. This helps to keep focus on what may actually be exploited and what needs attending to first.

Exploitability can also be judged by the EPSS or Exploit Prediction Scoring System score. EPSS is managed by FIRST, Forum of Incident Response and Security Teams. EPSS is based on real life data of vulnerability exploitation attempts and thus provides useful insight into which vulnerabilities should be dealt with first [42]. Dependency Track gives an easy-to-read graph of vulnerabilities and their EPSS scores (see Figure 4).

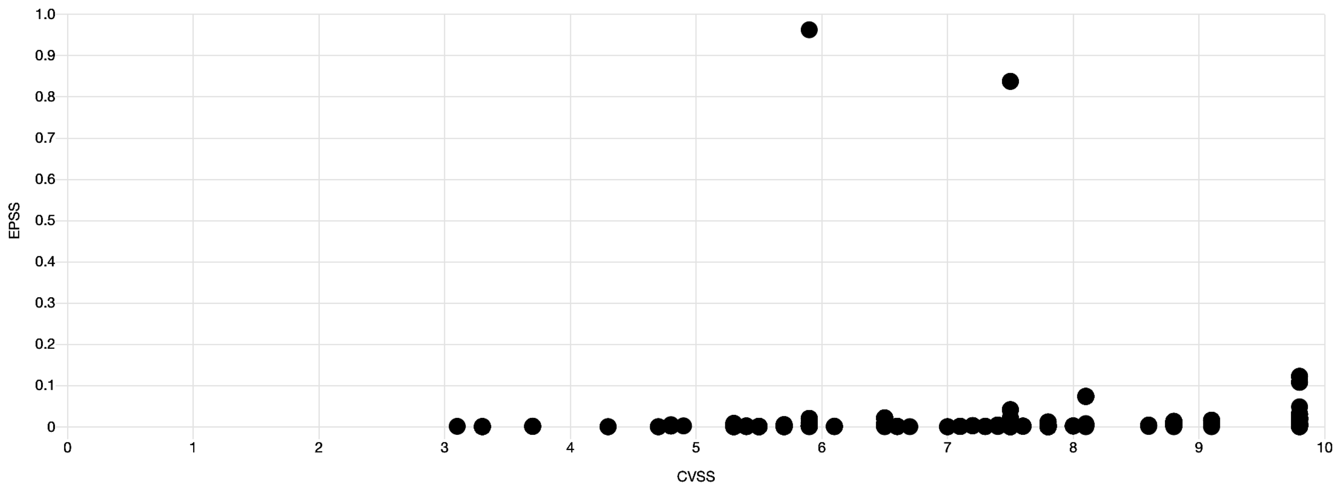

Figure 4. *Dependency Track EPSS Graph*

These scores are helpful and should be utilized by practitioners especially when the vulnerability count is still high after all other measures have been used.

9.3 Minimal Images

As the results show, there is some correlation between image size and the number of vulnerabilities found. Encourage developers to use minimal base images for their containers. It is also advised to remove any extra packages and dependencies that are not essential for the container to work. This, combined with other recommendations given here should ease the management of found vulnerabilities.

10. Summary

Docker has gained the majority of market share in the software deployment world. Its convenience and infrastructure agnostic nature has allowed developers to shorten the time-to-market of software products. Such acceleration of the development cycle has had some negative security implications as security testing was forced to accelerate as well.

This thesis looked at the available static vulnerability scanning tools for Docker container images. A brief overview of the Docker architecture and the basic principles of vulnerability scanning was given. Vulnerability scanners were selected for testing on the basis of what is open source, free to use in enterprise environments, and actively maintained. In total, four tools were selected for testing: Trivy, Grype, Dependency Track, and Clair.

A detailed overview of the testing and data gathering procedure was given in the testing section. Different custom configuration options for the chosen scanners and their inclusion in different testing phases was discussed. Altogether 387 Docker images were tested. The initial testing phase showed drastic differences between the coverage of different scanners. After some configuration changes - most notably the inclusion of Trivy scanning alongside Dependency Track - the coverage improved significantly. In total, 134954 vulnerabilities were identified in 379 images. Eight images were deemed vulnerability-free by all the scanners. The results of the tests indicate that the best option for static vulnerability scanning in enterprise environments is Dependency Track. It achieved the best coverage with the appropriate configuration options. Although it was not the easiest to deploy, its efficacy and the plethora of other helpful features separated it from the competition.

Although enabling Trivy analyzer in Dependency Track increased the false positive rate, the extra coverage provided by Trivy analyzer compensates this issue to some extent. The relatively small amount of such false positives and the fact that all found vulnerabilities need to be manually triaged and reviewed anyway means that Dependency Track in its final configuration is still the final recommendation.

Other notable insight gathered from the results was the fact that half of all vulnerabilities or the majority of "high" or "critical" vulnerabilities can be fixed by updating the corresponding component within the image. Also, that while the vulnerability databases are important, the quality of SBOMs plays a bigger role in improving the results that a scanner gives. The thesis also outlines some problematic aspects of static vulnerability scanning.

First the already mentioned SBOM quality and also the applicability of the results. For example, a vulnerability that only affects hosts running Windows was identified on 149 of the 387 images that were tested. While the scanners were technically correct in identifying this vulnerability, it is still not relevant for Linux based containers.

With regards to the research questions, this thesis outlined the tools and methods by which static vulnerability scanning can be used by practitioners to gain insight into their security posture. This in turn helps practitioners reduce the attack surface of their Docker deployments. Four open source and free to use tools were also identified that can help achieve this goal.

The tests that were conducted concluded that Dependency Track is the best option, especially for enterprise environments. Furthermore, the results and discussions presented in this thesis identified some causes of the discrepancies in the results that the scanners give. These were the quality of SBOMs and the number of vulnerability databases that a scanner uses.

Insights into how to configure a tool and interpret the results for maximum usefulness were also given. These include keeping containers and components within them up to date and triaging and filtering results based on applicability and exploitability.

11. Conclusion

As presented in the results section of this thesis, Dependency Track proved to have the highest vulnerability coverage, given some custom configuration on the users part. This is in line with the results of previous research where multiple scanners were combined for best results [9]. The combination of Dependency Track and Trivy on the back end has the added bonus of being maintained and officially supported and does not require extra maintenance on the users part as opposed to the custom combined tool proposed previously [9].

Static vulnerability scanning is not without its issues, though. Testing conducted in this thesis showed that the results can differ widely based on the configuration of a scanner. Even when the tools are properly configured, the effective use of static vulnerability scanning still requires a considerable amount of manual analysis. This makes all the remediation methods such as keeping the images small, packages up to date, and triaging even more important. The results of this research illustrate the importance of updates for managing the number of vulnerabilities identified by the scanners. These tools often generate an overwhelming volume of results, especially when recently introduced, making it critical to reduce the amount to a manageable level. However, nearly 50% of all vulnerabilities found by these scanners are for components which have a fixed version available, and nearly 80% of "high" or "critical" vulnerabilities can also be excluded by updating.

The extra functionality of Dependency Track in the form of vulnerability triaging, license management, policy enforcement, and security auditing is another aspect which highlights Dependency Track among the competition. These tools are especially useful in larger enterprise environments where security duties are usually split between different teams and organizational policies and auditing is commonplace. Having a single interface for managing all these tasks makes Dependency Track a worthwhile tool to adopt.

The comparison of currently available tools and their functionality is the main contribution of this thesis for the cyber security community. Previous works are either out of date or lack the practical discussions on the usage of any given tool. Recommendations and other pointers given in this thesis should be useful to any practitioner looking to adopt any of the tested tools or static vulnerability scanning in general into their workflow.

Future work includes improving the scanning solutions to avoid obvious irrelevant vulnerabilities such as applicability to the container's base operating system. Also, a deeper dive into the specifics of different SBOM creators would be in order.

References

[1] Arun Chandrasekaran and Wataru Katsurashima. *A CTO's Guide to Containers and Kubernetes: Top 10 FAQs*. Last accessed 21 July 2024. 2024. URL: https://www.gartner.com/doc/reprints?id=1-2GI4QT6A&ct=240206&st=sb.

[2] Sysdig. *Sysdig 2023 Cloud-Native Security and Usage Report*. Last accessed 20 July 2024. 2023. URL: https://sysdig.com/blog/2023-cloud-native-security-usage-report/.

[3] Sean Michael Kerner Saheed Oladimeji. *SolarWinds hack explained: Everything you need to know*. Last accessed 13 December 2024. 2023. URL: https://www.techtarget.com/whatis/feature/SolarWinds-hack-explained-Everything-you-need-to-know.

[4] NIST. *Executive Order 14028*. Last accessed 8 December 2024. 2024. URL: https://www.nist.gov/itl/executive-order-14028-improving-nations-cybersecurity.

[5] Check Point Research. *May 2023's Most Wanted Malware*. Last accessed 13 December 2024. 2023. URL: https://www.globalsecuritymag.com/May-2023-s-Most-Wanted-Malware-New-Version-of-Guloader-Delivers-Encrypted-Cloud.html.

[6] Cyber Safety Review Board. *Review of the December 2021 Log4j Event*. Last accessed 13 December 2024. 2022. URL: https://www.cisa.gov/sites/default/files/publications/CSRB-Report-on-Log4-July-11-2022_508.pdf.

[7] 6sense. *Top 5 Containerization technologies in 2024*. Last accessed 20 July 2024. 2024. URL: https://6sense.com/tech/containerization.

[8] Ruchika Malhotra, Anjali Bansal, and Marouane Kessentini. "Vulnerability Analysis of Docker Hub Official Images and Verified Images". In: *2023 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE, 2023.

[9] Alan Mills, Jonathan White, and Phil Legg. "OGMA: Visualisation for Software Container Security Analysis and Automated Remediation". In: *2022 IEEE International Conference on Cyber Security and Resilience (CSR)*. IEEE, 2022.

[10] Alan Mills, Jonathan White, and Phil Legg. "Longitudinal risk-based security assessment of docker software container images". In: *Computers & security* (2023).

[11] Vipin Jain et al. “Static Vulnerability Analysis of Docker Images”. In: *IOP conference series. Materials Science and Engineering* (2021).

[12] Omar Javed and Salman Toor. “Understanding the Quality of Container Security Vulnerability Detection Tools”. In: *arXiv.org* (2021).

[13] Andres Pihlak. “Continuous Docker Image Analysis and Intrusion Detection Based On Open-source Tools”. MA thesis. Tallinn University of Technology, 2020.

[14] Shafayat Hossain Majumder et al. “Layered Security Analysis for Container Images: Expanding Lightweight Pre-Deployment Scanning”. In: *2023 20th Annual International Conference on Privacy, Security and Trust (PST)*. IEEE, 2023.

[15] Taeyoung Kim, Seonhye Park, and Hyoungshick Kim. “Why Johnny Can’t Use Secure Docker Images: Investigating the Usability Challenges in Using Docker Image Vulnerability Scanners through Heuristic Evaluation”. In: *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*. 2023.

[16] Franziska Vollmer et al. “A Comprehensive Analysis and Evaluation of Docker Container Attack and Defense Mechanisms”. In: *2024 IEEE Symposium on Security and Privacy (SP)*. 2024.

[17] Wensen Ma. *Virtual Machines vs. Containers: Navigating the Landscape of Modern Computing*. Last accessed 13 December 2024. 2024. URL: https://www.devskillbuilder.com/virtual-machines-vs-containers-navigating-the-landscape-of-modern-computing-a6944ae3176f.

[18] Joerg Flade. *Docker — What it is, How Images are structured, Docker vs. VM and some tips*. Last accessed 13 December 2024. 2020. URL: https://ragin.medium.com/docker-what-it-is-how-images-are-structured-docker-vs-vm-and-some-tips-part-1-d9686303590f.

[19] IBM. *What is the Log4j vulnerability?* Last accessed 13 December 2024. 2024. URL: https://www.ibm.com/think/topics/log4j.

[20] IBM. *IBM X-Force Threat Intelligence Index 2024*. Last accessed 13 December 2024. 2024. URL: https://www.ibm.com/reports/threat-intelligence.

[21] CVE Program. *CVE History*. Last accessed 8 December 2024. 2024. URL: https://www.cve.org/about/history.

[22] OWASP. *Component Naming Schemes*. Last accessed 8 December 2024. 2024. URL: https://owasp.org/www-project-web-security-testing-guide/latest/5-Reporting/02-Naming_Schemes.

[23] Anchore. *Anchore Engine Gitlab repository*. Last accessed 14 September 2024. 2024. URL: `https://github.com/anchore/anchore-engine`.

[24] Quay. *What is Clair*. Last accessed 14 September 2024. 2024. URL: `https://quay.github.io/clair/whatis.html`.

[25] Red Hat. *What is Clair?* Last accessed 14 September 2024. 2024. URL: `https://www.redhat.com/en/topics/containers/what-is-clair`.

[26] Elías Grande. *Dagda*. Last accessed 14 September 2024. 2024. URL: `https://github.com/eliasgranderubio/dagda`.

[27] Aqua Security. *Trivy*. Last accessed 14 September 2024. 2024. URL: `https://github.com/aquasecurity/trivy`.

[28] Aqua Security. *Trivy*. Last accessed 15 October 2024. 2024. URL: `https://github.com/aquasecurity/trivy-db`.

[29] Anchore. *Grype*. Last accessed 20 October 2024. 2024. URL: `https://github.com/anchore/grype`.

[30] Docker. *Docker License*. Last accessed 29 November 2024. 2024. URL: `https://docs.docker.com/subscription/desktop-license`.

[31] Docker. *Docker Scout License*. Last accessed 29 November 2024. 2024. URL: `https://github.com/docker/scout-cli/blob/main/LICENSE`.

[32] OWASP. *Dependency-Track Documentation*. Last accessed 29 November 2024. 2024. URL: `https://docs.dependencytrack.org/`.

[33] jFrog. *jFrog X-ray Documentation*. Last accessed 29 November 2024. 2024. URL: `https://jfrog.com/help/r/jfrog-security-documentation/jfrog-xray`.

[34] Quay. *Clair GitHub page*. Last accessed 30 November 2024. 2024. URL: `https://github.com/quay/clair`.

[35] Oliver Vanamets. *Scanned Images*. Last accessed 8 December 2024. 2024. URL: `https://github.com/fal0/Scanned-Images/blob/main/scanned_images.txt`.

[36] OWASP. *Dependency Track FAQ*. Last accessed 30 November 2024. 2024. URL: `https://docs.dependencytrack.org/FAQ/`.

[37] Anchore. *Grype Supported Sources*. Last accessed 1 December 2024. 2024. URL: `https://github.com/anchore/grype?tab=readme-ov-file#supported-sources`.

[38] Aqua Security. *Trivy Supported SBOMs*. Last accessed 1 December 2024. 2024. URL: `https://trivy.dev/latest/docs/target/sbom/`.

[39] OWASP. *Dependency Track CI/CD*. Last accessed 1 December 2024. 2024. URL: `https://docs.dependencytrack.org/usage/cicd`.

[40] NIST. *CVE-2023-45283*. Last accessed 2 January 2025. 2023. URL: `https://nvd.nist.gov/vuln/detail/cve-2023-45283`.

[41] NIST. *CVE-2023-44981*. Last accessed 2 January 2025. 2023. URL: `https://nvd.nist.gov/vuln/detail/CVE-2023-44981`.

[42] FIRST. *Exploitation Probability Scoring System*. Last accessed 8 December 2024. 2024. URL: `https://www.first.org/epss/`.

Appendix 1 – Non-Exclusive License for Reproduction and Publication of a Graduation Thesis[1]

I Oliver Vanamets

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis “Docker Image Security Scanning Using Open Source Tools”, supervised by Mauno Pihelgas
   1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
   1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.
2. I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

02.01.2025

[1]The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.