Investigating the inner workings of container image vulnerability scanners

Mehdi Zarei

Thesis submitted for the degree of
Master in Applied Computer and Information Technology - ACIT
(Cloud-based Services and Operations)
30 credits

Department of Computer Science
Faculty of Technology, Art and Design

Oslo Metropolitan University — OsloMet

Spring 2022

Investigating the inner workings of container image vulnerability scanners

Mehdi Zarei

Investigating the inner workings of container image vulnerability scanners

http://www.oslomet.no/

Printed: Oslo Metropolitan University — OsloMet

Abstract

The use of container technology as a main part of software development increasing exponentially. Containers do not only provide a huge benefit for Integration/Continuous Delivery (CI/CD) pipelines, but also simplify shipping problems. However, the security of container images is a primary concern. Exploitation of a single vulnerability in an image could have huge consequences and result in loss of CIA (Confidentiality, Integrity, Availability) in an application. While there are a variety of image scanners that create vulnerability reports informing the security teams, there is a lack of knowledge about the inner workings of container images and how they interact with different types of images.

First, this thesis describes the history of containers, tools, and technology related to containers. Second, we discuss some of the most popular container image scanners and have selected two which are both open-source and highly ranked. Next, the thesis explains how scanners detect packages and vulnerabilities. Finally, a few experiments are conducted with three different types of containers; standard container images, distroless and images that have been slimmed down. These kinds of images are scanned using the image scanners and the results are compared. Our findings reveal that:

1. Both selected images scanners use roughly the same algorithm to detect vulnerabilities
2. Trivy supports more OS and application packages
3. The majority of the detected vulnerabilities are unfixed vulnerabilities
4. None of the tested scanners were able to detect vulnerabilities when using slimmed down images.

Acknowledgments

I would like to express my appreciation for following people who help and support me throughout this master thesis and made the entire project more interesting, educational and fun.

Hårek Haugerud my supervisor for motivation and support me during doing my master thesis and has made this thesis possible.

Ismail Hasan as a co-supervisor for encouragement, help and guild lines to overcome the challenges.

Emilien Socchi who submitted this proposal and provide amazing support during this project. Thank you for all the help and productive feedback.

Kyrre Begnum who holds perfect classes and courses during the master program which have changed my view of cloud computing.

I would like to express my gratitude to Oslo Metropolitan University (OsloMet) for offering me an opportunity in this wonderful master's program and thanks all of our professors and lecturer for their guidance and support me to complete the master's degree.

Finally, special thanks to my dearest family and Friends for all the encouragement and support during the master's program.

Sincerely, Mehdi

Contents

List of Figures

List of Tables

Abbreviations

The following acronyms are used in this report:

- IT - Information Technology
- AWS - Amazon Web Services
- CI - Continuous Integration
- CD - Continuous Delivery
- OS - Operating System
- CIA - Confidentiality, Integrity, Availability
- OCI - Open Container Initiative
- VMM - Virtual Machine Monitor
- LXC - Linux Containers
- QOS - Quality OF Service
- CVSS- Common Vulnerability Scoring System
- CAS - Content-Addressable Storage
- DL - Deep Learning
- CVE - Common Vulnerabilities and Exposures
- CNCF - Cloud Native Computing Foundation
- CLI - Command Line Interface
- JSON - JavaScript Object Nation
- NIST - National Institute of Standard and Technology
- SHA - Secure Hash Algorithm
- RHEL - Red Hat Enterprise Linux
- API - Application Programming Interface
- IaaS - Infrastructure as a Service
- NVD - National Vulnerability Database
- VM - Virtual Machine
- IaC - Infrastructure as Code
- SCAP - Security Content Automation Protocol
- CAS - Content-addressable storage

Chapter 1

Introduction

Last decade the use of virtualization technology increased rapidly which allowed for having multiple isolated virtual environments in a single system. OS-level Virtualization technologies are divided into two different approaches. The former was Hypervisor-based virtualization and the recent one is container-based virtualization which is known as lightweight virtualization or containers [9].

Containers were known in the Linux system for a long time and the first initial release of a Linux container was around 2008 LXC (Linux Containers project)[28]. Containers provide an isolated environment that contains all the packages, libraries, files and dependencies that need to run their application, so can easily set up in a new environment and run without any dependencies. They not only accelerate workflow and improved the application deployment process, but also leverage scalability and flexibility. It allows the creation of multiple servers and replication in a short time.

Container technologies have become a new trend and most companies use containers to produce their application. Based on Gartner's prediction, more than 70% of global companies have two or more applications that use container technologies by 2023 [52]. Also the same year, the container market is predicted to increase from 1.2 billion in 2018 to 4.9 billion(USD)[6].

1.1 Motivation

Most organizations decide to use the core benefit of containers such as lightweight, portable and considered as a replacement for VMs. They take into account as a standard way for deploying microservices[44]. A single container can run small microservices or large applications. Giant companies like Amazon, Spotify, Netflix and Twitter used micro-services to deliver their product[44].

1.2 Problem statement

The introduction of containers improved software agility, flexibility and a new way to organize microservices. While they are easier and faster to deploy, revolutionized Continuous Integration/Continuous Delivery (CI/CD) pipelines. However they fit into agile and DevOps practices, but security is the main concern for the system development life cycle.

DevSecOps mindset is delivering better and faster code, without ignoring the security. While agile development aims to reduce the number of cycles, DevSecOps try to integrate security in the software development cycle To produce secure software. Scanner tools could be a part of the CI/CD pipeline and stop the image push on vulnerabilities. It's difficult to ensure that all packages and images are up to date and they lack malware and vulnerabilities. In addition, a vulnerability in one single image as a core component of a container could cause a problem in the CIA triad(Confidentiality, Integrity, Availability) and have a huge consequence for application.

As far as images are stored, manage and distribute publicly and programmer use them to build infrastructure, criminals can easily place malware into images and upload the polluted image to the repository. Fortinet and Kromtech, two security software organizations, found 17 affected docker images In June 2018. Affected images contain a crypto-mining program and these images were downloaded 5 million times[49].

What's more, based on several surveys about image security, around 46% of developers accept that they do not have any idea whether used images have any vulnerabilities or not [31]. Therefore, more research about the improvement of security in all components of application especially images as a main part of microservices is desirable.

This study aims to improve the security of applications that use microservices. In this thesis, we focus on an efficient way to improve security on container images which is a core component of microservices. We will address the following research questions:

1. **How do container image scanners detect vulnerabilities?**

2. **To what extent are current open source container image scanners able to detect vulnerabilities?**

1.3 Thesis outline

The remain part of the essay is structured as the following layout:

Chapter 2 (Background and related Work): The second chapter propose an important concept of container security and describe more about some of the famous image scanner which is more popular and has a big community. In other words, prepared some helpful information related to container, images scanner and vulnerabilities which give a broad view to the

reader.

Chapter 3 (Approach): This part represents the methodology that we used to solve the problem and how the researcher approach the task.

Chapter 4 (Results): Will describe the result of the research, install and compare the vulnerability scanner and explain some metrics found in this thesis.

Chapter 5 (Discussion): Discuss some challenges that face and how much this approach was practical. This chapter explains some methods that help to improve the security of image and proposes future work.

Chapter 6 (Conclusion): Finally in this chapter we answer the question statements and briefly summarise the whole essay.

Chapter 2

Background and related work

This chapter will elaborate on the tools and technologies used to answer the problem statement. Moreover, the chapter intends to describe some general concepts related to security and containers.

2.1 Virtual machines

A virtual machine is a technology that helps businesses to run multiple separate OS on one single computer. It's a technique to enable their use of resources more effectively, while every virtual machine has a separate OS, CPU, RAM, Network[43].

2.1.1 Virtual machines and hypervisors

Virtual machines provide a high-level secure system and every VMs has its own OS, and dedicated hardware. VMs create and run by VMM (Virtual Machine Monitor) which is a software layer. VMM or hypervisor helps to manage VMs and are placed between a physical machine and a virtual machine. Hypervisors have two different types: type 1 whereas the bare metal would install OS directly onto a physical server. The bare metal hypervisor was introduced in the 1960s by IBM[36]. VMware ESXi and Hyper-V are some instances of this type. Another one is type 2 or hosted. This type needs to run other software on the host and then run a virtual machine on them. VMware Workstation or Oracle VM VirtualBox are some clear examples of hosted types.

2.1.2 Virtual machines vs containers

Containers, like VMs, share resources from a host and can deploy many applications in an isolated environment and can solve resource problems. VMs isolate OS and resources from each other. Table 2.1 represent the comparison between containers and VMs. VMs have some drawbacks such as being large in size and time consuming in both creation and boot. On other hand, containers boot and start within a second while using the same kernel of OS[44]. They provide isolation at the process level by some Linux

Table 2.1: Comparison between VMs and containers

Parameter	Virtual Machine	Container
Contenet	Container networking, storage, memory, library and config file	Files, apps and libraries
Size	Heavyweight and need a multi GB fore files	They are lightweight and MB in size
Boot time	Take minutes to load	Take few second to start
Security	VM have own OS and offer better protection	All share the host OS
Cost Benefit	Use resources maximize	Not all resources use, Lead to waste
Isolation	Fully isolated	Provide process level isolation
Providers	VMware , Virtual Box, Hyper - V	Docker, LXC, LXD

Kernel features like cgroups and namespaces. VMs provide full isolation and they are more secure compared to containers. The main differences between containers and VMs could be seen in table 2.1 The main drawback of containers is security. Since containers use the same OS as a host, they consequently share a kernel with a host which provides less security. In contrast, VMs have their own virtual kernel. Fig. 2.1 shows the architecture of VMs and containers.

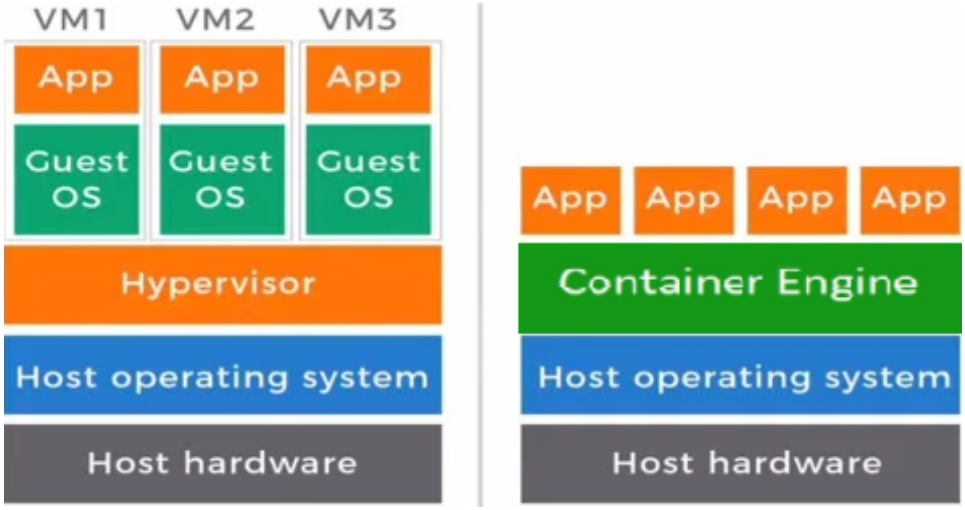


Figure 2.1: VM vs container architecture

2.1.3 Namespaces and cgroups

In a Linux kernel, there are some features that isolate processes from each other and assign resources to a process. As has previously been described, all containers use the same kernel and share all resources like CPU, RAM, network and disk[34]. It is clear that when malware exploits on one of the containers or hosts, as far as they use the same kernel could result in

problems for all other containers and platforms. By namespaces, every process sees the set of its own resources. Linux provides namespaces including ipc, mnt, net, pid, time, user and uts that each have their own properties. For instance, the user namespace has own user id and group id. Fig. 2.2 represent the namespaces in Linux kernel 5.13.0.

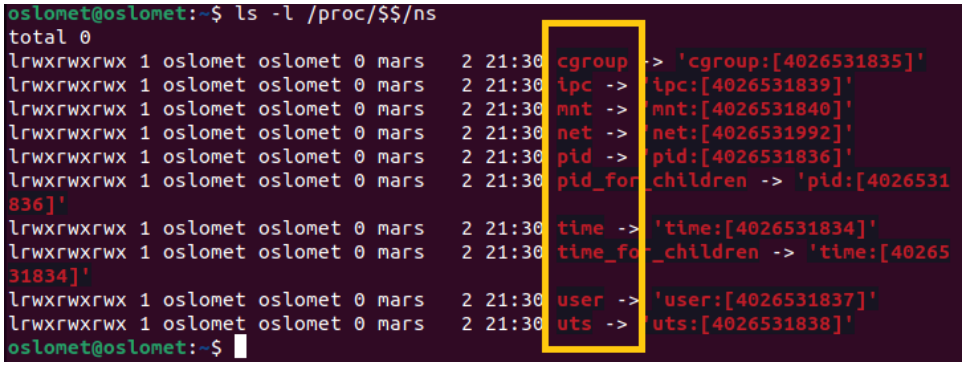


Figure 2.2: Namespaces in Linux kernel 5.13.0

Namespaces provide each process with its own view of the system. Cgroups(abbreviated as Control groups) are another feature that limits and isolated resources like CPU, RAM, Disk I/O, and network for collection of the process. There is more than one process running in every container. So, cgroups limit how much each process could use and it's one of the key components in container security. Fig. 2.3 shows the cgroups in Linux kernel 5.13.0.

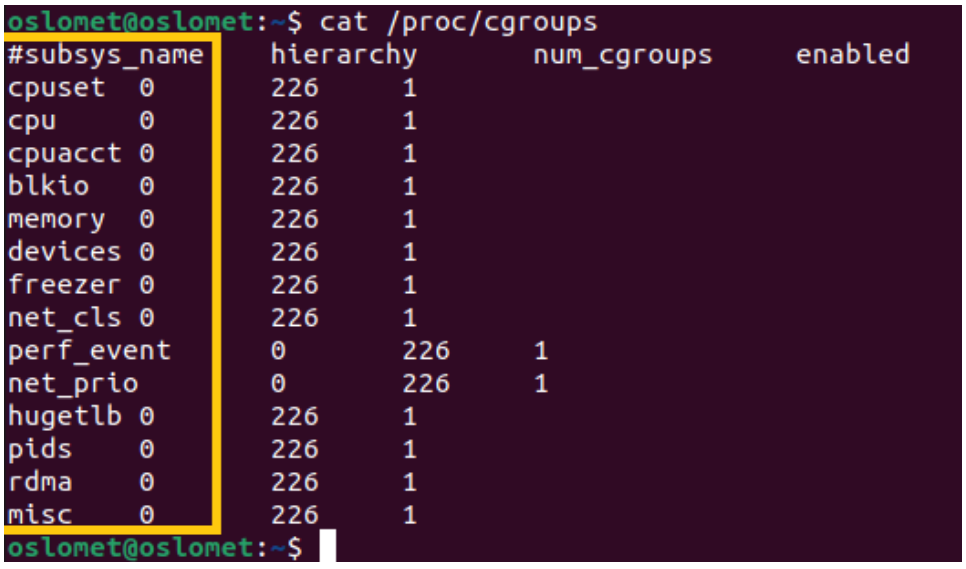


Figure 2.3: Cgroups in Linux kernel 5.13.0

2.2 Container technology

Container technology was introduced in 2008, while Docker was introduced 5 years later in 2013 [46] using LXC as a environment for execution. One year later, in 2014, version 0.9 of Docker was released using its own component named libcontainer, written in the Go language[45] [50].

2.2.1 Container images and registries

Images are a core component of containers and include one or several different layers. Images are immutable and have a static file. The static file includes every parameter that needed to run the container. It is possible to reuse one image to run multiple containers and thereby improve the performance of containers and reduce the size. Container images need a shared place to store and distribute which is named a container registry. The container registry helps developers to upload (push) and download (pull) into another system or cluster. A container registry divided into 2 different types: public and private. Public registry is used by individual users and they can directly control the container content. Docker hub is the most popular container registry among the public type. Another type of registry is private registry could be on-premises and improve the level of security. Most cloud providers like Google, AWS and Azure, provide a private registry[15].

Every container registry contain one or more collections of repositories. A collection of container images in addition to artifact create a repository. If several repositories have the same base image, only one of the images store in the registry. Tag is one of the artifacts that specify the version of image and each repository is a collection of artifact as can be seen in Fig. 2.4

2.2.2 Docker containers

There existed several container technologies before the introduction of docker. However, Docker just provided better tools and standards and then become dominant in the container market. Docker is open source project Platform as a service that helps developer to build, run, manage, update, stop and even remove containers. A Docker container is an instance that could be created by any user and uploaded on registry like a Docker hub. There are two different methods to able to create a Docker image. Either with OS images like Linux, Fedora which is called a base image or by using docker file which has some commands in a file and can create image[41]. Dockerfile is a simple text file that includes some instructions and a list of commands that the Docker engine will run to build an image. Images define by the Docker file and every line in the Docker file could represent a separate layer[16].

Each Docker contains 3 different elements. First is the Docker client which interacts with the user and container. It helps users to run some commands and interact with Docker Daemon. The second is Docker Daemon which gets a command from the Docker client and then mange the

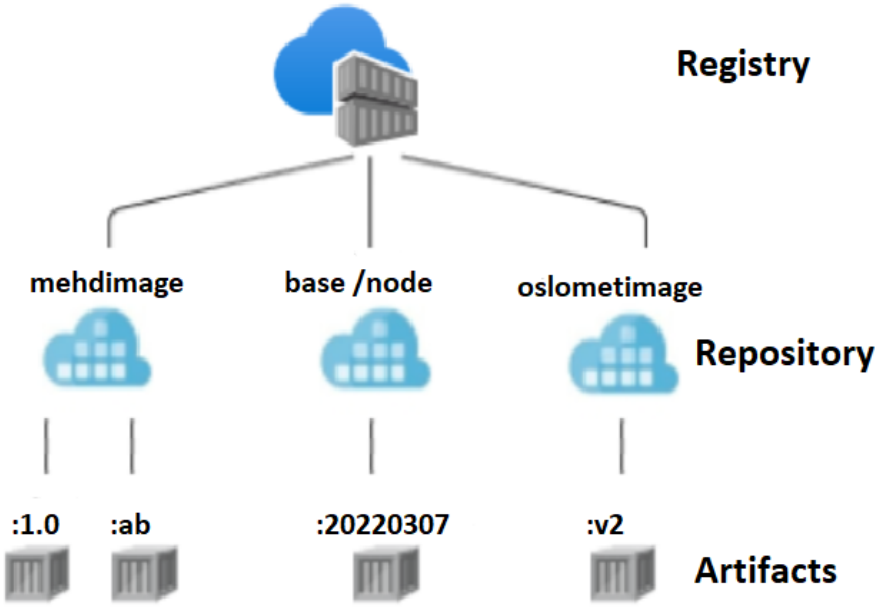


Figure 2.4: Architecture of container registry, repository and artifacts

container and Docker hub. The last element is the Docker register which shares Docker images and it is a place to store images[30].

2.2.3 OCI format for container images

The Open Container Initiative (OCI) is a structure that was launched by containers leader in the industry like Docker and CoreOS in June 2015[2]. This project is under the Linux Foundation which provides a standard format for containers and is part of the Cloud Native Computing Foundation (CNCF). OCI includes two different specifications. The first one which is runtime-spec defines the lifecycle, configuration and running environment of each image container. In this specification, there are setting like Linux namespaces, cgroups and mount location. It shows how to execute the "filesystem bundle" after downloading and unpacking the image on the disk. This means that every image should contain sufficient information like a command, environment variables, etc that container engines expect to run this image.

OCI runtime-spec and lifecycle

Lifecycle represent the timeline of container and have the following elements[38];

ociVersion() that shows the version of Open Container Initiative Runtime Specification.

id()represent the container id and should be unique among all containers.

status() describe state of container and could be one of creating,created, running, stopped values.

pid() its a container process id and its a required item in Linux, but in other platform could be optional.

bundle() it shows the path of container's directory on the host and user can find the configuration file and file-system of container in this path.

annotations() is optional value and show the property or comment that associated with container.

Following Fig. 2.5is an example of JSON format for OCI images.

```
{
    "ociVersion": "0.2.0",
    "id": "oci-container1",
    "status": "running",
    "pid": 4422,
    "bundle": "/containers/redis",
    "annotations": {
        "myKey": "myValue"
    }
}
```

Figure 2.5: JSON format for OCI lifecycle

The second specification is image-spec that name shows define how we can create an OCI image and what specification need to have. This part should include filesystem (layer), manifest, serialization and standard configuration and dependencies. OCI image is a combination of image configuration, image manifest and files system serialization.[39].

2.3 Software vulnerabilities

Software vulnerabilities can find application weaknesses and a pollute system environment. There is some method that which this software could access the system. They can install themselves, install by packet manager(apt, yum,..) or application package. They can exploit by attackers or hackers to get access to files or systems.

2.3.1 Common Vulnerabilities and Exposures (CVE)

There are two different types of vulnerabilities which are known vulnerabilities and unknown vulnerabilities. The US launched a system in 1999 to provide reference-method for known information-security vulnerabilities and exposures[40]. It is a reference method and standard way for known vulnerability and exposure.

The syntax for vulnerabilities is CVE prefix + Year + Arbitrary digits. A year is a year of disclosure of software vulnerability. Some of the companies that report the vulnerability such as Oracle or red Hat have their own ID to recognize the malware. They try to give each vulnerability one CVE, but some of them need to assign a large number of CVEs[14]. It is a

Table 2.2: Common vulnerability scoring system[42]

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

common name dictionary that helps to share data across network security and identify them. The second one is unknown vulnerabilities which are undisclosed and not wildly known to the public, but if you find them and report them to the public, after they become a known vulnerability.

2.3.2 Common Vulnerability Scoring System(CVSS)

Known vulnerabilities are the target of the scanner and they have ID and severity. Some of the security vendors have their own system to score the severity of the vulnerability. CVSS (Common Vulnerability Scoring System) is the scoring system that has some open standards and assigns a number to vulnerability and evaluates the severity. This standard is used by CERT and NVD to show the impact of the vulnerability.

CVSS are different types and evaluate the severity of vulnerabilities, then give them scores according to some metrics[12]. Common Vulnerability Scoring System changes during the year and the current version is 3.1 which was released in June 2019[42]. There is a different kind of metric that shows how much impact the assets of the organization if the vulnerability is exploited. The Score range is different based on the level of impact on the system. This range defines from 0 to 10 and the highest scope is a higher degree of vulnerability severity.

As could be seen in table 2.2 there are different levels of severity and it is very crucial to understand the severity level to do risk analysis and take proper decision[47].

The Level in the scoring system includes: Critical: This vulnerability could result in root-level permission in the system and infrastructure. At this level, hackers do not need any information about a target and do not need any credentials.

High: Exploitation of this level could result in permission to the system and could cause downtime or loss of data. However, is more difficult to exploit in comparison to the critical level.

Medium: In this level, an attacker needs some information about victims or needs to get access local network. Some hackers do social engineering for manipulating victim users. Exploitation needs user permission and provides limited access.

low: In this level, exploitation has little impact on the organization and needs physical access to the asset or be on the local place[25].

None: There is no potential risk to target the system.

2.3.3 National Vulnerability Database (NVD)

National Vulnerability Database is a repository of standard vulnerability which uses Security Content Automation Protocol (SCAP) to enable the management of vulnerability and security. NVD is a database for reported known vulnerabilities using Common Vulnerability Scoring System (CVSS) to score CVEs. They provide information related to vulnerabilities and are used by the organization to find out the priority of vulnerabilities and what should be done to keep their infrastructure safe. CVE is a list of vulnerabilities and NVD is more database that builds based on the CVE list and each update on the CVE list appears on NVD[37].

2.3.4 Static and dynamic analysis

Cybercriminals are innovative and try to create new malware(short-term malicious software). They looking for a way to take control of the target program and access confidential information. Since attackers create software vulnerabilities to take control of the system, There are two different techniques to find vulnerabilities and protect the software. These approaches are static and dynamic detection. However, some people tried to integrate dynamic detection with static analysis to improve the efficiency [54] and some others developed automated detection for vulnerability in software by using Deep Learning(DL) [10].

Static detection is more simple and fast, so it is an effective way to be a part of workflow during the development process. The static method uses some samples and signatures and tries to analyze vulnerabilities based on their attributes in the repository. In this technique, try to find out malware behavior without running code [4]. static detection has some challenges like updating the repository regularly. What's more, some of the malware uses obfuscation techniques like changing the code and making it difficult to detect them by static detection.

Another detection is dynamic which runs the malware in a controlled environment and then evaluates the malware behavior. In the dynamic analysis method, first, run the virus, then monitor the process and virus behavior. Sometimes need a debugger to analyze malware functions. This dynamic approach is more accurate however both methods have their own advantages and drawbacks[3].

2.4 Container image security and vulnerability scanning

Image is the main component of the container and some of them can be infected by viruses or Trojans. This problem happens more frequently if you run outdated images[53]. Image scanners evaluate the security of images and containers. There are many scanning tools for scan images and finding vulnerabilities. These scanners usually collect package information of applications and help us to protect software and improve security. They

Name	Stars	Contributers	Pulls	Commits	Downloads	Last Commit
Trivy	11056	176	25	750	3547330	31.03.2022
Clair	8568	102	8	1508	9397	22.03.2022
Snyk	3831	172	57	4761	2 915 490	31.03.2022
Anchore	3166	31	4	596	1 271 220	30.03.2022
Dagpa	945	7	5	267	*NA	27.07.2021

Table 2.3: Comparison container scanners based on Github insights visited 31. Mars 2021 (NA= Not Available)

do the binary scan in components by finding each layer of image [17]. There is a variety of image scanner tools like Trivy, Snyk, Dagpa, Clair and Anchore.

The thesis has aimed to evaluate the most popular scanners on the market. To compare scanners together, access their source code and compare some of GitHub's factors. In GitHub stars, meaning as an appreciation and most stars represent the quality of a project. This table 2.3 compares these scanner based on GitHub feature like the number of download, community, pull request and stars. The download number for Clair is low because it's just the number of Clair v4.4 new version of Clair. The number of downloads for the Dagpa scanner is not available because as you can be seen in table2.3 shows that not update for more than 8 months. The following list presents some of the popular image security scanners;

2.4.1 Clair

An open-source tool available on macOS, Windows and Linux. Clair scanner is written by Go programming language using static analysis for finding vulnerabilities in application and container images [1]. This tool which was created by CoreOS used for finding the security issue in images by using CVEs. First Clair detects all the layers in the container for vulnerabilities. There are some resources like (Debian Security Bug Tracker, Ubuntu CVE Tracker and Red Hat Security Data)[11] which Clair uses these databases as a reference list for searching vulnerabilities and exposures.

Clair v4.0 engine is a ClairCore package that has some library and fetches layer and detects them. Clair has an API that communicates with Clair microservice[11].

2.4.2 Anchore

Image open-source scanner for security that can run alone or in orchestration platforms such as K8s, Rancher and Amazon ECS. It can use as a plugin in Jenkins to scan CI/CD pipeline. In addition, It is possible to access throw command line or API and customize security policy to evaluate images[5]. Anchore uses static analysis to evaluate the container images and some policy-based compliance systems. Here is the process of analyzing image in Anchore:

In the first step, fetch the image and extract it without execution. second, will analyze the image with an Anchore analyzer and separate metadata and classify them. third, will save the result in a database to use in the future and for audit. Then try to check the result and find vulnerabilities and update the data and image analysis result. Finally, notify the user about vulnerability matches[29].

2.4.3 Dagpa

An open-source image scanner that is more than 99% written in Python. It performs static analysis to find vulnerabilities and malware. To perform scanning, first imported all of the known vulnerabilities into MongoDB and then searched in this database to find vulnerabilities and exploits during analysis. As a result of storing reports and analyses in MongoDB, you can easily see and maintain the history of images and containers.

Dagpa for discovering vulnerabilities using ClamAV- is an antivirus engine- for detecting the malicious threat [13]. Dagpa support different type of Linux: Redhat/ CentOS/ Fedora, Debian/ Ubuntu, Alpine. Dagpa is also able to analyze dependencies based on OWASP dependency check (is software that checks and detects malware in application dependencies) and Retire.js(use a different library in JavaScript to find vulnerabilities) Java, Python, PHP, Nodejs [19].

It also uses Falco (cloud-native run-time security for detecting suspect behavior in real-time)for monitoring running containers with help of Docker daemon[24].

2.4.4 Trivy

In 2019, another open-source image scanner was created by Teppei Fukuda and developed by Aqua Security. Trivy, not only detect vulnerability in Os packages and applications but is also able to scan IaC(Infrastructure as Code) file like Terraform and k8s to find misconfiguration which could be a potential threat for your deployment[7]. Trivy easily installs and can start scan containers, just download and run[8]. These features of the Trivy scanner are described in the following list:

Simple installation: Just with a few command in package manager like yum, you can install it and update it later.

Fast and accurate: It takes a few seconds to scan the image and show the vulnerabilities ID and fixed version. Trivy downloads all of the security advisors to GitHub and removes duplication. Then make DB in GitHub and update it every 12 hours. The database file is a light use key/value database and does not have any setup. Trivy check GitHub Advisory Database and other data sources in another repository which include security patches[8].

Support variety of vendors: Trivy support most OS like alpine, Red Hat, CentOS, Debian, Ubuntu, Oracle, openSUSE, Photon OS, and has a plan to support Fedora and windows in the future. Also, accept multiple formats

for scanning an image in container registries like Tar file, local image, OCI image format or docker file.

It is possible to set up client/server mode for your environment. Server cache some same repetitive layer in the image that scanned before and does not need to do it again for all of the containers in software or application. It is possible to configure the port number in the server to make it more secure[8].

2.4.5 Snyk

Snyk is another scanner that provides different products such as Synk code, Synk container, Synk infrastructure as code. Its also able to integrate with IDE, CI/CD pipeline, registry and management tools. There is two way to use Synk. First, by using Synk CLI which is a command-line to scan for vulnerabilities, It is possible to install CLI with npm, scoop or manually. when running the command to scan container with Snyk, first check locally in docker daemon if it is not already available, then download images from the registry. After that check, the software installed in the image and then Synk service returns the vulnerabilities list.

The second one is Snyk API and is available for paid plans. This method which is only available on HTTPS needs a token from Snyk , to use API, need to register with Snyk and use a token for authorization. Synk has a different pricing plan and is only free for individual users and some limited tests. Big companies and organizations should buy a business plan or enterprise plan which has advanced control and report [51].

2.5 Related Work

This section provides some related work on this technology.

Emilien socchi and jonathan luu in 2019 during their thesis[23] " A deep dive into Docker Hub's security landscape " developed tools called Docker imAge analyZER(DAZER) that analyze different types of docker images. Their software which was written by Python found out most of the vulnerabilities handed in from parent to child image. They evaluate the security of Docker Hub by using their analyzer to collect metadata from any type of available image on four different repositories as Official image, Community, Verified and Certified. They by using DAZER discovered that there are fewer vulnerabilities in the majority of Official, Community and Certified images compare to Verified images.

in [33] Kaur, Mathieu Dugre, Aiman Hanna and Tristan Glatard at "An analysis of security vulnerabilities in container images for scientific data analysis" present how vulnerabilities on images increase security risk and compare 4 different image scanners- Vuls, Anchore, Clair and Singularity Tools (Stools) to scan container images that used in neuroscience data analysis. They describe how outdated packages and unused packages cause

vulnerabilities. The solution to reducing the number of vulnerabilities in images could be to update the software and remove the unused package.

Another paper is[32] that Mubin Ul Haque† and M. Ali Babar in “An Empirical Study of Exploitability and Impact of Base-Image Vulnerabilities” present how one vulnerability in an image could drive a security attack in different software. They check the largest docker registry - Docker Hub and evaluate base images. They used Anchore which uses more sources for the vulnerability database and update every 6 hours to do in-depth vulnerability analysis. They discovered that vulnerability on 261 base images used on 4,681 container applications in GitHub.

Markus Linnalampi in his master thesis [35] "outdated software in container images" in 2021 discuss how container become most popular in software development. Then explained how to detect outdated software from container images by scanners and best practices to make them more secure. Finally provide a new pipeline for scanning containers in the CI pipeline.

Chapter 3

Methodology

In this chapter we will explain about required action to address the problem statement question" most popular open-source image scanners interact with images and detect vulnerabilities? What could be done to make them more efficient and improve the level of image security?". Also, describe some flowcharts, the design phase and some key aspects to do the project.

3.1 Objectives

The objective of this project is to put all of the tools and technologies from the background into the frame to make a prototype. To make clear prototype thesis approach is divided into 3 different steps. The first step is to design the model The second phase is to do some experiments and in final step analysis the result. As mentioned before, methodology in this project needs three phases that present in Fig. 3.1.

1- Design

(a) Create an environment to install multiple VMs with same specification to install a container scanners which imply in this research.

(b) Define which number of container scanners include in this project.

(c) Define a set of images involve in this research.

2- Experiment

(a) Identify hardware and infrastructure that need to install VMs and container scanners.

(b) Install the container vulnerability scanners and other tools.

(c) Perform scanning and collect quire data.

3- Measurement and analysis

(a) Run task and install all of the scanners to evaluate and test requirements and the environment.

(b) Perform a scanner for images and evaluate the accuracy.

(c) Collect metadata, and vulnerability from the database.

(d) Collect data and Compare results from each experiment.

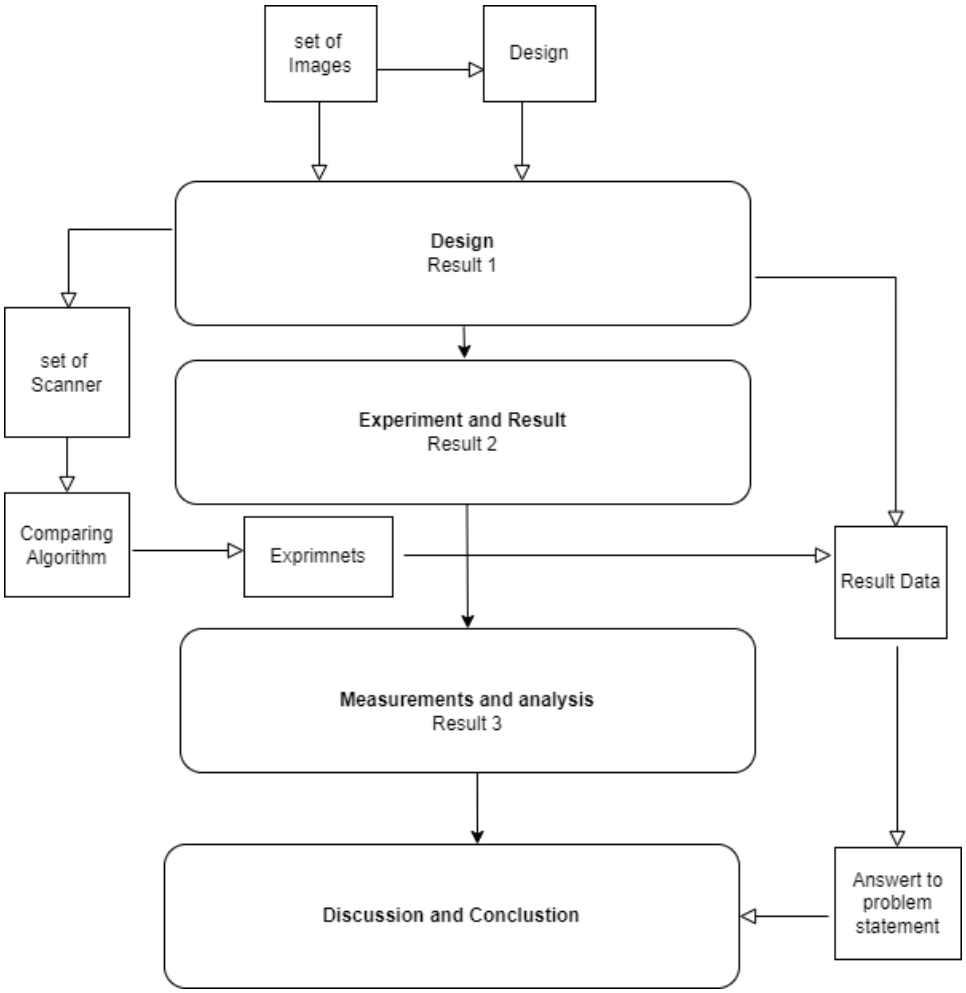


Figure 3.1: Thesis methodology

3.1.1 Data set and prototype

In this step need to decide on the vulnerability scanner and OCI container image that need to use to conduct this project. There were some criteria to choose vulnerability scanners. The most important factor was being an open-source scanner and having a big community. As can be seen in this table 2.3 there is 5 popular container scanner including Trivy, Clair, Snyk, Anchore and Dagpa. Based on our criteria, Snyk which is also used in Docker Hub is a commercial scanner for an enterprise company and it is out of scope in this essay. Moreover, some scanners like dagpa have old commits as you can see in Fig. 2.3. In addition, Dagpa does not have a big community behind it compared to others and seems that not actively supported. Clair, Anchore and Trivy were founded in 2015, 2016 and 2019 respectively. To invest in this research, have chosen Clair as a First developed image scanner and Trivy as a fast growing scanner with lots of features.

3.2 Design phase

This phase is a combination of several models to install selected container scanners on VMs and evaluate the scanner container performance. Fig. 3.2 represent the design phase that includes installing different technology to get the result. There are different layers and all of them are connected and need to work properly to achieve the goal. The first layer is the hardware host system, then VMM which controls hardware and assigns hardware to VMs. Then we install Trivy and Clair as selected image scanners for the experiment.

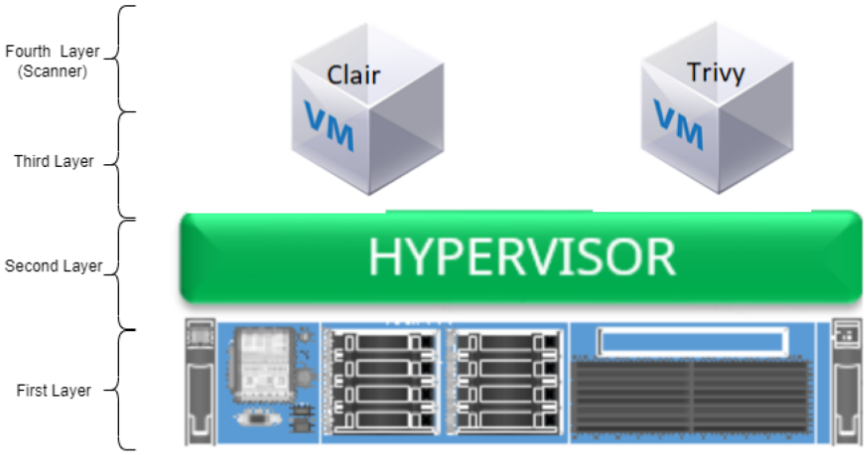


Figure 3.2: Overview on design phase

1. Hardware Layer: The first layer is the physical server which is situated in Oslo Metropolitan University(OsloMet) server room.

2. Hypervisor: Is an OpenStack cloud computing platform in OsloMet which provide IaaS and fulfills the requirement to run instances on it.

3. Virtual Machine: Created 2 different VM and installed Ubuntu server LTS 20.04 for this purpose.

4. Container Scanner: selected container security scanner was Trivy and Clair to install and evaluate their performance.

3.2.1 Tools and technologies

This section explains about tools and technologies used in this project. This project is based on the OsloMet university environment and hardware provided by computer department.The following tools used in this research:

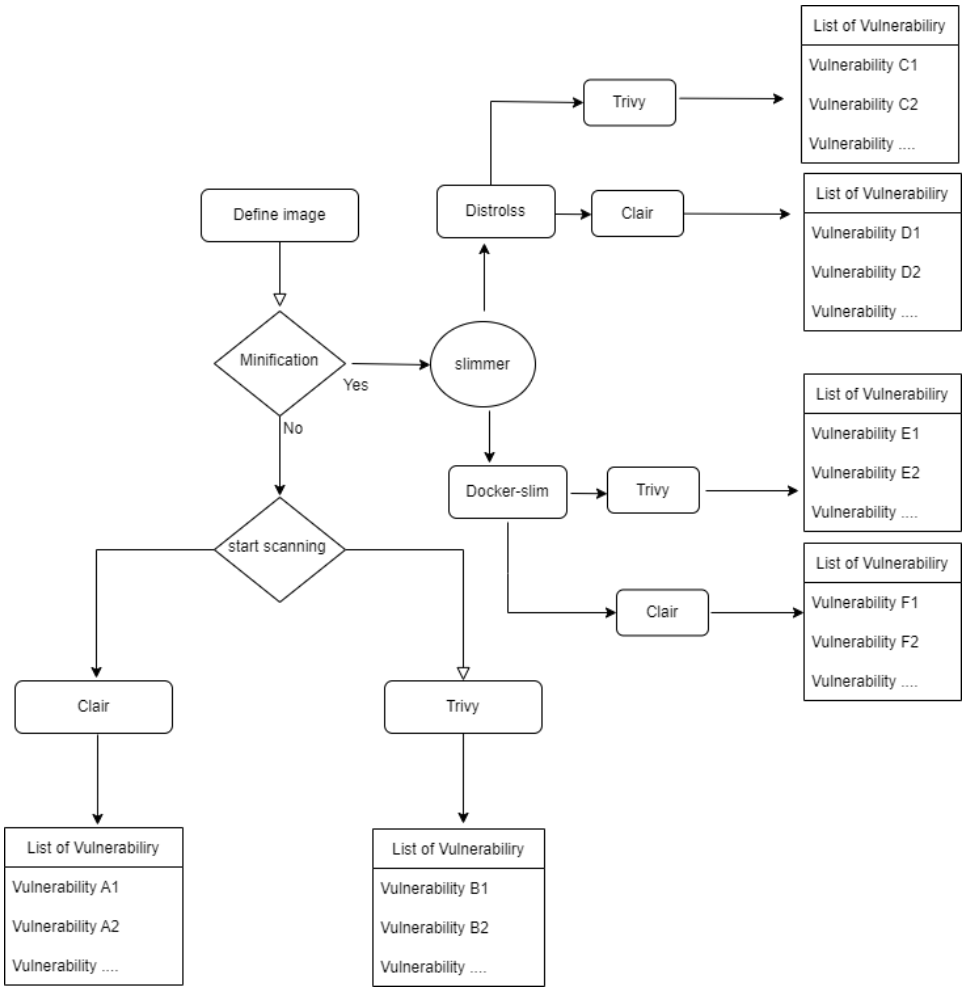


Figure 3.3: The design workflow

1. Go version go1.16.15: Mostly refer as Golang is an open-source programming language designed by Google in 2007 and announced in 2009. Go influenced by the C language, but more simple. It is designed to improve the programming productivity in some areas like a networked machine or multi-core[21]. It could be seen that most container security scanners were written by Go.

2. Ubuntu 20.04.1 LTS

3. Docker Engine - Community version 20.10.14

4. Clair scanner

5. Aqua Trivy

6. Docker slimmer

3.2.2 Selected images

In this part, we select some images to analyze how accurate container scanners find out vulnerabilities. We do not focus on any special images and applications. As far as this study is not related to the particular company and has a general purpose, In this research tries to select the most popular official images that have more than 1 billion downloads and more than 10k stars. In addition, picked some application packages to compare both selected scanners together. This image was widely used and its variety of operating systems and application packages to evaluate scanner images. The following list is some selected sets of image containers that will evaluate in this study.

1. alpine: This is an open-source Linux Operating System, it is also a lightweight images base that is small and more secure. It chooses by most companies as a base image due to its lack of vulnerability.
2. redis: As the most popular open-source key-value data store. Redis is an abbreviation of REmote DIctionary Server.
3. mySQL: This is another open-source software that is a leading relational database for web-based programs.
4. node.js: Application which is written in javaScript and use for networking applications.
5. nginx: Not only provide an open source proxy server but is also use for load balancing and a web server.
6. python: High-level programming language that has a simple syntax and is similar to the English language.
7. debian: One of the open-source Linux distributions is the free operating system.
8. ubuntu: Is a Debian-base Linux OS that is free and has a different edition.
9. java: High-level and object-oriented programming language.
10. PostgreSQL: Is a relational database to store data and retrieve it later. It also is a default database for the Clair container scanner.

these image was widely used and its variety of operating system and application packages to evaluate scanner images.

3.3 Expected result

Image scanners use static analysis and expected that Trivy and Clair have the same algorithm to find out vulnerabilities. They have their database

and some sort of repository as an updater. For scan images, the scanner connected to the images registry gets a different layer. After that, Scan layers, and release the operating system and application packages. then update their database based on the resources and finally match them with the local vulnerabilities database. In addition to OS, the application package could be affected by vulnerabilities like log4j which discover in December 2021 in the java Library. As long as Clair and Trivy are the popular vulnerability scanner, I expected both of them to analyze operating system packages and application packages. It supposes that the main approach for discovering malware is checking the package manager in the operating system and finding out the list of installed packages. Then, detect these packages version and check them for vulnerabilities.

Trivy and Clair rely on their database to match and find the vulnerability. Both of them need their database to store vulnerabilities from all distributions. Every operating system has a different version and includes lots of information and source. Another expectation could be based on the verity of the package, they need any other 3rd tools for a running database.

Finally, in the experiment, we scan some standard images, distroless images and slimmed-down images. The slim image only contains kernel and runtime dependencies. They do not have any package manager or shell. After scanning both It is expected that the vulnerability scanner was not able to scan slim images or not find any vulnerabilities.

Chapter 4

Experiments and results

The first part explains how Clair and Trivy work, and find OS packages, application packages and vulnerabilities. What's more, will be conducted some experiment that is into 3 different categories. the first category is to scan standard images by both scanners and evaluate their accuracy. The second type of experiment is to scan Distroless images and compare the result to normal images. In the last part, slimmed down the standard images and scan them again to see the result.

4.1 Analyzing Clair scanner

Clair software is available on different operating systems like Windows, Linux, and macOS platforms. Also, it comes in different implementation methods. It's possible to install based on client-server which makes it complex in installation and it is not a convenient way to always have a dedicated server[27]. Another method is standalone which is more suitable for our experiment. This standalone is maintained by CoreOS and actively supported at this time of writing. Clair detects malware with known vulnerabilities or weaknesses. The first step they need to identify layers, packages and images version, then check with the vulnerability database as a reference and detect them.

This section dig into source code and explain some of the module and functions. It's important to understand how they connect to resources to update the vulnerability database and how they fetch layers and find vulnerabilities in packages.

4.1.1 Comprehensive analysis

Clair scanner is modular design and relay on ClairCore package. This image scanner was developed by CoreOs and acquired by RedHat in 2018.

ClairCore as a core engine of Clair implements the security scanner and uses the Postgres database which is a free and relational open-source database. In Fig. 4.2 can be seen that ClairCore have 3 main components including LibIndex, postgresSQL and LibVuln. Libindex is a service that gets the manifest and indexes it. Index report is an internal data structure

and represents the container images that this report can be sent and fed to libVuln to check with matcher and find out the vulnerabilities and produce a vulnerability report. Both LibIndex and LibVuln daemons save their information such as distributions, vulnerabilities and packages from their source to postgresSQL database as a content addressable.

Content addressed storage

The PostgreSQL database stores all of the data from manifests and layers. Additionally, it stores the vulnerabilities found by the LibVulne module. The point is that all of the data will be saved as content addressable. In Content-Addressable Storage, an abbreviation named CAS is a method that saves data based on their content, not their address or location. It helps to retrieve data with high speed from fixed content like document[18].

Some of the images like Ubuntu are using in many containers as a base layer, To create a new container, it just needs to add other layers on top of that. When Clair starts to scan the container, and store all of the layers in the database, and when fetches the layers recognize that the Ubuntu layer has in the database. It means that just download from the registry once base layer and use it again.

```
    #https://github.com/quay/claircore/blob/main/cmd/cctool/inspector.go
1   func Inspect(ctx context.Context, r string)
    (*claircore.Manifest, error)
2   ref, err := name.ParseReference(r)

3   repo := ref.Context()
4   auth, err := authn.DefaultKeychain.Resolve(repo)

5   rt, err := transport.New(repo.Registry, auth,
    http.DefaultTransport, []string{repo.Scope("pull")})
6   desc, err := remote.Get(ref, remote.WithTransport(rt))

7   img, err := desc.Image()
8   h, err := img.Digest()

9   ccd, err := claircore.ParseDigest(h.String())

10  out := claircore.Manifest{
    Hash: ccd,
11   return &out, nil
}
```

The following step happen when start to scan image with Clair vulnerability scanner:

1- In spect function is the function that send images to registyr and get the manifest as an output. Inspect function use name package from golang which parse the input image by reference interface. This interface have some data including repository context.

```
type Reference interface {
fmt.Stringer
// Context accesses the Repository context of the reference.
Context() Repository
// Identifier accesses the type-specific portion of the reference.
Identifier() string
// Name is the fully-qualified reference name.
Name() string
// Scope is the scope needed to access this reference.
Scope(string) string
}
```

As can be seen every manifest include digest and layers which both of them are json file. Every image have their unique digest and could be identify among other images.In the first step, claircore will check the image registry to get information from the container and image layer and metadata. ClairCore uses inspec.go file that inspects the registry for specified image manifest with a pointer to layer. This file uses the container registry library in golnag to work with resources in the container registry like images and layers. Inspect Function as you can see in the file, need two input which is "ctx" as a context type and "r" string. Context is a package in golang which listen to an event and notice if cancels this event or passes request data. This data after pass could have tag along or trace ID for monitoring and logging this variable. Another input is the name of images with a tag as an input string. The output is an array of a manifest. each manifest includes hash digest and layers in json files. The manifest as output defines as a struct which is a variable type in golang. Its abbreviation of structure contains a collection of fields. The manifest result show the layers and how able to retrieve them. here is manifest structure:

```
type Manifest struct {

 Hash Digest ‘json:"hash"

 Layers []*Layer ‘json:"layers"
   }
```

After parsing image, Name get string and return a repository reference(ref, err := name.ParseReference(r)) line 2. Then save this repository in repo variable and authenticate it by keychain library. Its important to recognize that names and layers are valid. in line 5 transport package handle and pull repo and do handshake authentication. you can see the digest number when you run the pull command in your system.

```
mehdi@ubuntu:~# docker pull ubuntu:20.04
20.04: Pulling from library/ubuntu
```

```
e0b25ef51634: Pull complete
Digest: sha256:9101220a875cee98b016668342c489ff0674f247f6ca20dfc91b91c0f28581ae
Status: Downloaded newer image for ubuntu:20.04
docker.io/library/ubuntu:20.04
```

Then in line 6 by get method from remote package have access to image descriptor and save in int desc variable. In line 7 by image method this descriptor change to images and in line 8 this images save in hash digest. Then use claircore parsDigest to make sure this digest is well-formed.

Digest is a hash of data and use by claircore. In line 10 produce manifest which is final output of this function and sent as a input to libvul function. Fig. 4.1 shows that each manifest contain multi layers and some configuration data which hashed. Image manifest is a json file which is not only content addressable but also include layers order and link to retrieve them from registry.

Figure 4.1: Manifest of haproxy

Each container builds up several layers and each of them presents metadata. Each layer could change except the first layer which is read-only and all other layers build on it. All the layers stick together and create a container.

the latest release for Clair is v4.4.1 at this time which was released last month in April 2022. After version 4 Clair uses authentication itself. Its keyserver protocol which use to sign and verify requests. former versions of Clair used another service for authentication called JWT Proxy which could configure to sign the outgoing request and authenticate the incoming request from other services.

2- The second step is sending the manifest to LibIndex package in Clair-Core. In this step As you can see in Fig 4.2 Library index is a function that fetches the container layers and identify the packages index the content and then provides the index report. This report includes the packages, repositories, and distributions. This step is also called indexing which submits a manifest, fetch layers, index the content, and then provide an index report. This operation is performed by the Index function of the "libindex.go" file. Index performs a scan and index of each layer within the provided Manifest. This package has different functions that will take options and dependencies, system locks of instance as an input, and returns libindex as output. This package includes different functions such as a scan and new to get the package and produce the Libindex report.

```
#https://github.com/quay/claircore/blob/main/libindex/libindex.go
   1   func New(ctx context.Context, opts *Opts, cl *http.Client)
       (*Libindex, error) {
   2   ctx = zlog.ContextWithValues(ctx, "component",
       "libindex/New")
   3   err := opts.Parse(ctx)
   4   dbPool, err := initDB(ctx, opts)
   5   store, err := initStore(ctx, dbPool, opts)
   6   ctxLocker, err := ctxlock.New(ctx, dbPool)
   7   l := &Libindex{
      Opts:  opts,
      store: store,
      client: cl,
       cl:    ctxLocker,
  }
   8   l.fetchArena.Init(cl, os.TempDir())

   9   pscnrs, dscnrs, rscnrs, err :=
       indexer.EcosystemsToScanners(ctx,  opts.Ecosystems,
       opts.Airgap)

   10  vscnrs := indexer.MergeVS(pscnrs, dscnrs, rscnrs)
      Hash: ccd,
   11   l.Opts.vscnrs = vscnrs
       return l, nil
}
```

in line 2 add key-value pairs of the relevant context. in this part using zerolog via context all logging in Clair core is done by zlog.

Then the "initDB" function of init.go library in libindex call to initialize a Postgres pool.Pool based on the given libindex.Opts.

LibIndex will run linindex.opts as a connection string and store data.

After that the "initStore" function of "claircore/libindex/init.go" call to initialize a indexer.Store . "Store" object which is an interface for dealing with objects libindex needs to persist:

Then Locker provides context-scoped locks and stores them into ctx-

Locker in line 6 and after that libindex is valued for scanning and indexing a Manifest. Line 8 an instance of "fetchArena" will be created by init function. this instance keeps trace of all layer which fetched. Line 9 After that, the "EcosystemsToScanners" of "claircore/internal/indexer/ecosystem.go" file will call to extracts multiple ecosystems and returns their discrete scanners and store them into array of Package Scanner as "pscnrs", array of Distribution Scanner as "dscnrs" and an array of Repository Scanner as "rscnrs". Then the function "MergeVS" of "claircore/internal/indexer-/versionedscanner.go" merges these lists of scanners into a single list of VersionedScanner as "vscnrs". The with the "RegisterScanners" of "claircore/internal/indexer/postgres/registerscanners.go" file, the results will store into postqres database. LibIndex is also going to store this information in its postgres database. This is an optimization because next time LibIndex is asked to scan the same layers that it already scanned before. It will instead of going to the image registry download the layers and perform the scanning. It will instead be able to find the previous results in its database and use those results to construct a new index report and send it quickly back to the client. and in the last line scanner version will be set into L which is the Index report.

The following list shows the structure of the Index Report.

```
 type IndexReport struct {
1 Hash Digest ‘json:"manifest_hash"‘
2 State string ‘json:"state"‘
3 Packages map[string]*Package ‘json:"packages"‘
4 Distributions map[string]*Distribution ‘json:"distributions"‘
5 Repositories map[string]*Repository ‘json:"repository"‘
6 Environments map[string][]*Environment ‘json:"environments"‘
7 Success bool ‘json:"success"‘
8 Err string ‘json:"err"‘
}
```

1. This is manifest hash key that describe IndexReport
2. Shows the state of this Index
3. Shows the discovered package in this particular manifest key with package ID
4. Represent the discovered distribution in this particular manifest key with distribution ID
5. Shows the discovered repositories in this particular manifest key with repository ID
6. Demonstrate list of environment that a package was discovered
7. Check that index operation finished successfully.

8. show the error in case of index operation not able to finish

In the last step, IndexReport receives data that contains full information about what content was located on the image. Libvuln is connected to the updater and performs fetch and parsing information from security database from sources like RedHat security data, Ubuntu trackers, Debian trackers, Oracle Linux security data, Alpine security database, and National Vulnerability Database(NVD). .[11] . This updater has two different interface parsers and fetcher. fetcher is called when a new security advisory is available. parser is an interface that is called to check the contents of security advisory and read data from them. Then pars them into the array to put into ClairCore vulnerability. This library is connected directly to the database and stores vulnerabilities for further matching.

The vulnerability will match this content with the information from its vulnerability database and based on this matching LibVuln will produce a vulnerability report and send it back to the client. The Vulnerability report contains the information from the IndexReport, which means the content of the image as well as the list of all vulnerabilities.In addition, Claircore will set a severity base on vulnerability. All of the company have their own severity database and ClairCore map them to these 6 categories. Unknown, Negligible, Low, Medium, High, Critical. For some companies that do not provide severity in vulnerability, ClairCore maps them to unknown.

. Fig. 4.2 shows the big picture of the container scanner architecture.

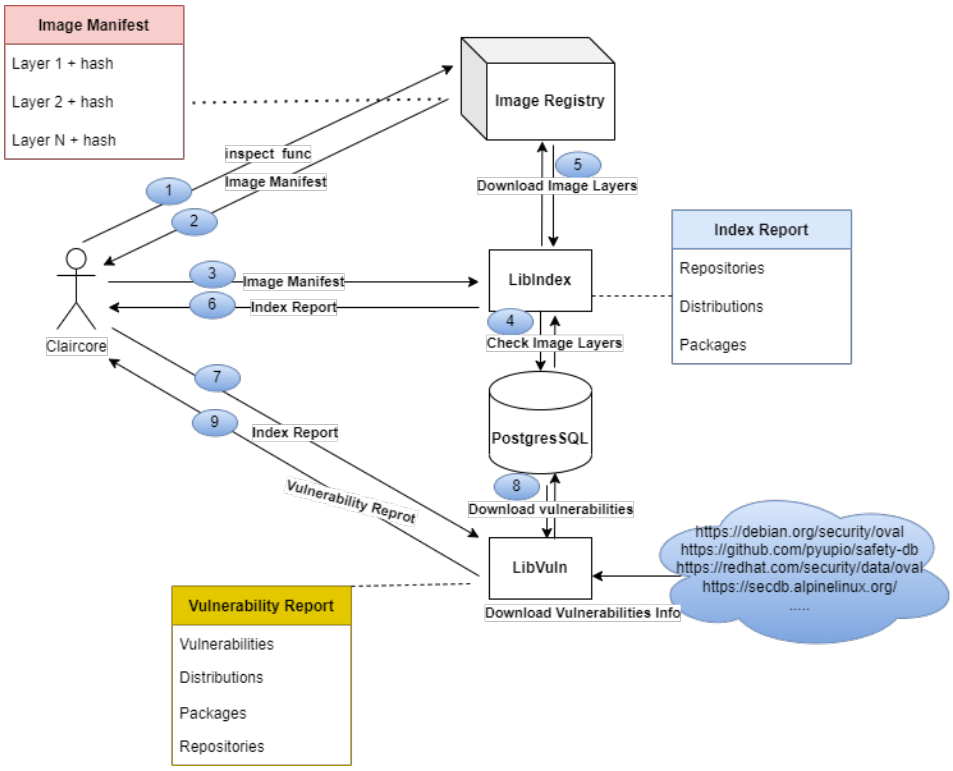


Figure 4.2: The Clair scanner container architecture

in summery:

Table 4.1: Clair resources updater

Num	packages	Resources Updater
1	Alpine	https://cve.mitre.org/
2	Ubuntu	https://people.canonical.com/~ubuntu-security/oval/
3	Oracle	https://linux.oracle.com/security/oval/
4	Debian	https://www.debian.org/security/oval/
5	RHEL	https://access.redhat.com/security/data/oval/
6	Suse	https://support.novell.com/security/oval
7	Photon	https://packages.vmware.com/photon/photon_oval_definitions/
8	AWS Linux	https://alas.aws.amazon.com/
9	Python (app)	https://github.com/pyupio/safety-db/

1. The Clair scanner uses the CLI interface that uses the inspect function to connect to the image registry

2. Images manifest is the output of this function which contain layer and digest. Fig. 4.1 is an example of manifest. This manifest also shows where this layer could be download.

3. In this step, the image manifest which is Json file will send to LibIndex and by reading, the data will understand where should download layers.

4. ibIndex check database for layer and if not able to find them in the local database,

5. Download layer from images registry. After download will do image scanning to find out which kind of content is in these image layers. This content includes package, distribution, and packages. Also, these data are stored in the postgres database for the future. For example, if next time the layer of the same image comes to LibIndex, first will check the database and don't need to download it from the images registry. This database is used as a cache for next time.

6. Return index report that includes repository, distribution, and packages.

7. Index report will send to LibVuln packages.

8. LibVuln uses a matcher which is a check updater and updates the database regularly. when the first time installs the Clair, it takes around 30 minutes to download the vulnerabilities from the updater. updater is a vulnerability database from different packages on the internet that could be seen in 4.1

9. Vulnerability report which is the final report will send to ClairCore.

4.1.2 Identifying The Operating System(OS)

Clair is able to find vulnerabilities in images that use one of the following packages or distributions including Ubuntu, Debian, RHEL, Suse, Oracle, Alpine, AWS Linux and VMWare Photon. At the time of writing the report, Python packages are the only application package for which Clair can find vulnerabilities.

It defines path and fallbackpath which are two locations that could be found os-release. then using "Scan" function to report information about distribution in the provided layer. This function use zlog package in line 2 which is the logging package by zerolog. It also generates the log context. Line 4 uses a reader which is a function that looks at a layer closer and opens the layer. in line 5 create a tar file system from the layer otherwise give an error. After that, start looking for the os-release file base on provided value in line 6. In this part try to open the tar file system and by Debug function read the name it is the same as an os-release. Finally, if find the file close the file in line 7 and save the os-release file in the rd variable.There is a list of supported ubuntu packate in 7.1.1. Then in line 7 use the toDist function that returns the distribution, version from the provided file. This function will use parse and split the os-release file content into key-value pairs. The output of the function is OS distribution. 7.1.3 represent the scanner.go source code.

```
#https://github.com/quay/claircore/blob/main/osrelease/scanner.go
const (
   Path         = ‘etc/os-release‘
   FallbackPath = ‘usr/lib/os-release‘
)

1 func (s *Scanner) Scan(ctx context.Context, l *claircore.Layer)
    ([]*claircore.Distribution, error) {
2  defer trace.StartRegion(ctx, "Scanner.Scan").End()
   ctx = zlog.ContextWithValues(ctx,
      "component", "osrelease/Scanner.Scan",
      "version", s.Version(),
      "layer", l.Hash.String())
3  zlog.Debug(ctx).Msg("start")
   defer zlog.Debug(ctx).Msg("done")

4  r, err := l.Reader()
5  sys, err := tarfs.New(r)
   var rd io.Reader
6  for _, n := range []string{Path, FallbackPath} {
      f, err := sys.Open(n)
      if err != nil {
         zlog.Debug(ctx).
            Str("name", n).
            Err(err).
            Msg("unable to open file")
         continue
      }
```

```
7     defer f.Close()
8     rd = f
      break
   }
   if rd == nil {
      zlog.Debug(ctx).Msg("didn't find an os-release file")
      return nil, nil
   }
7  d, err := toDist(ctx, rd)
   }
8  return []*claircore.Distribution{d}, nil
}
```

4.1.3 Identifying the packages

Clair uses the Scan function to find packages. This function looks for "status" file in directories because installed packages it can find in this file. If it is not able to find any database for packages, will return null. This function gets an instance of Context and a pointer of Claircore's layer as input and returns a pointer to an array of Claircore's package. In line 2 have used defer statements to delay the execution of this package. Trance packages have some facility to generate traces and loge the event during execution. In line 3 trace will log the layer along with additional information. ContextWithValues is a helper that takes pairs of strings and adds them to the Context via the baggage package. In line 4, it gets the component, the version and the layer and adds them to the ctx. In lines 5 and 6 Debug starts and "done" message ending the operation with the debug level. In line 7 the Reader method of "layer.go" calls to read the layer and store it in "rd". In line 8, package new from tarfs creates a file from the rd. From lines to 9 to 15 at first, a map keyed by the directory is created as "loc" then if the name of the package is "status" and "d" which "fs.DirEntry" is not a directory then one score will add one score to "loc[dir]" also, if the name of the package is "info" and "d" is a directory will add one score to loc[dir]. and finally in line 15 a "score" of 2 of "loc[dir]" means this is almost certainly a "dpkg" database. This for loop will repeat until find the directory or give the error . After finding the status file will open and read data such as name, package and version. At line 30 returns array of Claircore's package

```
#https://github.com/quay/claircore/blob/main/dpkg/scanner.go
1func (ps *Scanner) Scan(ctx context.Context, layer
    *claircore.Layer) ([]*claircore.Package, error) {

2  defer trace.StartRegion(ctx, "Scanner.Scan").End()
3  trace.Log(ctx, "layer", layer.Hash.String())
4  ctx = zlog.ContextWithValues(ctx,
      "component", "dpkg/Scanner.Scan",
      "version", ps.Version(),
      "layer", layer.Hash.String())
5  zlog.Debug(ctx).Msg("start")
```

```
6  defer zlog.Debug(ctx).Msg("done")
7  rd, err := layer.Reader()
   if err != nil {
      return nil, fmt.Errorf("opening layer failed: %w", err)
   }
   defer rd.Close()
8  sys, err := tarfs.New(rd)
   }
9  loc := make(map[string]int)
10 walk := func(p string, d fs.DirEntry, err error) error {
      }
      switch dir, f := filepath.Split(p); {
      case f == "status" && !d.IsDir():
         loc[dir]++
13    case f == "info" && d.IsDir():
         loc[dir]++
      }
   }

   if err := fs.WalkDir(sys, ".", walk); err != nil {
      return nil, err
   }
   zlog.Debug(ctx).Msg("scanned for possible databases")

   var pkgs []*claircore.Package
   for p, x := range loc {
15    if x != 2 { // If we didn't find both files, skip this
    directory.
         continue
      }
      ctx = zlog.ContextWithValues(ctx, "database", p)
      zlog.Debug(ctx).Msg("examining package database")

      // We want the "status" file.
      fn := filepath.Join(p, "status")
      db, err := sys.Open(fn)
      switch {
      case errors.Is(err, nil):
      case errors.Is(err, fs.ErrNotExist):
         zlog.Debug(ctx).
            Str("filename", fn).
            Msg("false positive")
         continue
      default:
20       return nil, fmt.Errorf("reading status file from layer
    failed: %w", err)
      }

21    found := make(map[string]*claircore.Package)

      tp := textproto.NewReader(bufio.NewReader(db))
   Restart:
      hdr, err := tp.ReadMIMEHeader()
      for ; err == nil && len(hdr) > 0; hdr, err =
```

```
        tp.ReadMIMEHeader() {
        name := hdr.Get("Package")
        v := hdr.Get("Version")
        p := &claircore.Package{
           Name:     name,
           Version:  v,
           Kind:     claircore.BINARY,
           Arch:     hdr.Get("Architecture"),
           PackageDB: fn,
        }
        if src := hdr.Get("Source"); src != "" {
           p.Source = &claircore.Package{
              Name: src,
              Kind: claircore.SOURCE,
               Version:  v,
              PackageDB: fn,
           }
        }

        found[name] = p
30      pkgs = append(pkgs, p)
     }
```

4.1.4 Identifying vulnerabilities in discovered packages

Libvul is a package that using Index Report and create a Vulnerability report. This package have some function like Scan and EnrichedMatch function. EnrichedMatch receives an IndexReport and creates a VulnerabilityReport containing matched vulnerabilities. In line 2 extract the IndexReport and save in records.IndexRecords returns a list of IndexRecords derived from the IndexReport and store them in 'recoreds'. At line 3 GOMAXPROCS sets the maximum number of CPUs that can be executing simultaneously. Lines 4 a pool creates to run matchers to match group. At line 6 WithContext method returns a new group to 'mg' and an associated Context derived from ctx to 'mctx'. from line 6 to 20 match group with context by using workers. The rest of the function pool will set up to watch the matchers and attach results to the report and finally use atomic to track closing the results channel. The final claircore Vulnerability Report is compose of Hash, Packages, Environments, Distributions, Repositories, map[string]*claircore.Vulnerability, map[string][]string for PackageVulnerabilities. The source code is available on 7.1.4

```
vr := &claircore.VulnerabilityReport{
     Hash:                 ir.Hash,
     Packages:             ir.Packages,
     Environments:         ir.Environments,
     Distributions:        ir.Distributions,
     Repositories:         ir.Repositories,
     Vulnerabilities:      map[string]*claircore.Vulnerability{},
     PackageVulnerabilities: map[string][]string{},
  }
```

```
#https://github.com/quay/claircore/blob/main/internal/matcher/match.go
1 func EnrichedMatch(ctx context.Context, ir
    *claircore.IndexReport, ms []driver.Matcher, es
    []driver.Enricher, s Store) (*claircore.VulnerabilityReport,
    error) {
2  records := ir.IndexRecords()
3  lim := runtime.GOMAXPROCS(0)
4  mCh := make(chan driver.Matcher)
5  vCh := make(chan map[string][]*claircore.Vulnerability, lim)
6  mg, mctx := errgroup.WithContext(ctx)
   for i := 0; i < lim; i++ {
      mg.Go(func() error { // Worker
         var m driver.Matcher
         for m = range mCh {
            select {
            case <-mctx.Done():
               return mctx.Err()
            default:
            }
            vs, err := NewController(m, s).Match(mctx, records)
            if err != nil {
               zlog.Error(ctx).
                  Err(err).
                  Msg("matcher error")
               continue
            }
            vCh <- vs
         }
         return nil
      })
20 }
   // Set up a pool to watch the matchers and attach results to the
       report.
   var vg errgroup.Group
   vg.Go(func() error { // Pipeline watcher
   Send:
      for _, m := range ms {
         select {
         case <-mctx.Done():
            break Send
         case mCh <- m:
         }
      }
      close(mCh)
      defer close(vCh)
      if err := mg.Wait(); err != nil {
         return err
      }
      return nil
   })
   vg.Go(func() error { // Collector
```

```
    for pkgVuln := range vCh {
       for pkg, vs := range pkgVuln {
          for _, v := range vs {
             vr.Vulnerabilities[v.ID] = v
             vr.PackageVulnerabilities[pkg] =
                 append(vr.PackageVulnerabilities[pkg], v.ID)
          }
       }
    }
    return nil
 })
```

4.2 Analyzing Aqua Trivy scanner

Trivy is a vulnerability image scanner that analyzes the operating system and application packages and artifacts. In addition, Its is fast and even able to scan misconfiguration in Iac. It's integrated with CI/CD pipeline. Trivy doesn't rely on any particular database and has its own compact database in Github called Trivy-db.

4.2.1 Comprehensive analysis

Trivy can Scan different artifacts such as images, file systems like host machines, and Git repositories. In addition, able to work in both Standalone or Client/server mode[8]. In client/server mode, which is the best choice if you like to scan images in a different location and do not like to download databases in each place. Trivy download vulnerability database in the server, and then all of the clients should connect to the server. Trivy clients are connected to the container registry and to the Trivy server to get vulnerability results.

In standalone mode, first, download vulnerability information from GitHub Trivy-database Ttivy database. Trivy-db is an internal database that gets information from the following list and updates automatically every 6 hours. hours[48] Trivy-db uses a bolt to store vulnerabilities which is a key/ value database. Bolt is a Go language project that doesn't need a full database server like Postgress and provides a fast and reliable database. This database saves data as a bucket. The database contains different buckets like vulnerability, severity, and vulnerability ID supplied by vendors. As previously mentioned, NVD has numerous vulnerabilities unrelated to any Specific package o language. This database does not store an NVD database to reduce the size of the database.

1. Trivy download the vulnerability information from Trivy-db.

2. Trivy has a cache inside. in this step, check layers with this cache situated in home/.cache and include both fanal and db directory and then will pull the missing layer from a container registry.

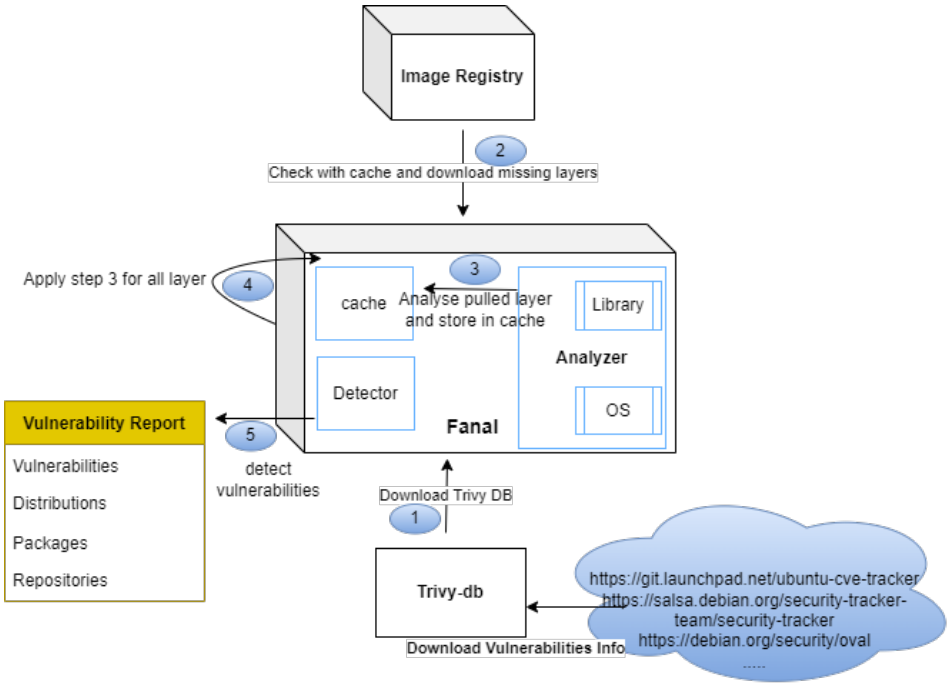


Figure 4.3: Trivy architecture

3. After that, analyze one layer and store information into the cache; this applies to all layers

4. Continue step 3 for other layers until the end.

5. Detect vulnerability with detector module and report it.

4.2.2 Identifying the Operating System(OS)

Trivy support many OS packages and version. For example, Trivy support all version of Linux Ubuntu. Every operating system has its own variable, eolDates, and contains a list of all versions that support it. for Ubuntu, Its located in https://github.com/aquasecurity/trivy/blob/main/ pkg/-detector/ospkg/ubuntu/ubuntu.go

```
var (
   eolDates = map[string]time.Time{
      "4.10": time.Date(2006, 4, 30, 23, 59, 59, 0, time.UTC),
      "5.04": time.Date(2006, 10, 31, 23, 59, 59, 0, time.UTC),
      "5.10": time.Date(2007, 4, 13, 23, 59, 59, 0, time.UTC),
      "6.06": time.Date(2011, 6, 1, 23, 59, 59, 0, time.UTC),
      "6.10": time.Date(2008, 4, 25, 23, 59, 59, 0, time.UTC),
      "7.04": time.Date(2008, 10, 19, 23, 59, 59, 0, time.UTC),
      "7.10": time.Date(2009, 4, 18, 23, 59, 59, 0, time.UTC),
      "8.04": time.Date(2013, 5, 9, 23, 59, 59, 0, time.UTC),
      "8.10": time.Date(2010, 4, 30, 23, 59, 59, 0, time.UTC),
      "9.04": time.Date(2010, 10, 23, 23, 59, 59, 0, time.UTC),
      "9.10": time.Date(2011, 4, 29, 23, 59, 59, 0, time.UTC),
```

```
      "10.04": time.Date(2015, 4, 29, 23, 59, 59, 0, time.UTC),
      "10.10": time.Date(2012, 4, 10, 23, 59, 59, 0, time.UTC),
      "11.04": time.Date(2012, 10, 28, 23, 59, 59, 0, time.UTC),
      "11.10": time.Date(2013, 5, 9, 23, 59, 59, 0, time.UTC),
      "12.04": time.Date(2019, 4, 26, 23, 59, 59, 0, time.UTC),
      "12.10": time.Date(2014, 5, 16, 23, 59, 59, 0, time.UTC),
      "13.04": time.Date(2014, 1, 27, 23, 59, 59, 0, time.UTC),
      "13.10": time.Date(2014, 7, 17, 23, 59, 59, 0, time.UTC),
      "14.04": time.Date(2022, 4, 25, 23, 59, 59, 0, time.UTC),
      "14.10": time.Date(2015, 7, 23, 23, 59, 59, 0, time.UTC),
      "15.04": time.Date(2016, 1, 23, 23, 59, 59, 0, time.UTC),
      "15.10": time.Date(2016, 7, 22, 23, 59, 59, 0, time.UTC),
      "16.04": time.Date(2024, 4, 21, 23, 59, 59, 0, time.UTC),
      "16.10": time.Date(2017, 7, 20, 23, 59, 59, 0, time.UTC),
      "17.04": time.Date(2018, 1, 13, 23, 59, 59, 0, time.UTC),
      "17.10": time.Date(2018, 7, 19, 23, 59, 59, 0, time.UTC),
      "18.04": time.Date(2028, 4, 26, 23, 59, 59, 0, time.UTC),
      "18.10": time.Date(2019, 7, 18, 23, 59, 59, 0, time.UTC),
      "19.04": time.Date(2020, 1, 18, 23, 59, 59, 0, time.UTC),
      "19.10": time.Date(2020, 7, 17, 23, 59, 59, 0, time.UTC),
      "20.04": time.Date(2030, 4, 23, 23, 59, 59, 0, time.UTC),
      "20.10": time.Date(2021, 7, 22, 23, 59, 59, 0, time.UTC),
      "21.04": time.Date(2022, 1, 22, 23, 59, 59, 0, time.UTC),
      "21.10": time.Date(2022, 7, 22, 23, 59, 59, 0, time.UTC),
      "22.04": time.Date(2032, 4, 23, 23, 59, 59, 0, time.UTC),
   }
)
```

Trivy has an Analyse function to identify OS type by looking at the specific filename and realizing the operating system. For instance, Ubuntu OS type could find in etc/lsb-release. The first step defines a variable with the file name that shows the operating system and version. In the following examples, every operating system has a file that shows the release and version. The following is some example of files and information inside the files.

```
var requiredFiles = []string{"etc/lsb-release"}
var requiredFiles = []string{"etc/alpine-release"}
var requiredFiles = []string{"etc/os-release"}


ubuntu@trivy:~$ cat /etc/lsb-release
DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=20.04
DISTRIB_CODENAME=focal
DISTRIB_DESCRIPTION="Ubuntu 20.04.1 LTS"
ubuntu@trivy:~$


Or for Debian
```

```
debian@oslomet:/etc$ cat os-release
PRETTY_NAME="Debian GNU/Linux 11 (bullseye)"
NAME="Debian GNU/Linux"
VERSION_ID="11"
VERSION="11 (bullseye)"
VERSION_CODENAME=bullseye
ID=debian
HOME_URL="https://www.debian.org/"
SUPPORT_URL="https://www.debian.org/support"
BUG_REPORT_URL="https://bugs.debian.org/"
debian@oslomet:/etc$
```

Trivy defines a variable with this path and then uses a regular expression to find this file by Analyse function. At first, set isubuntu to false. Then in line 2, the NewScanner function uses the Scan function, which looks for the file and checks the file path for requiredFiles which is a string and ists equal to "etc/lsb-release." At the top, lsb-release contains some information like DISTRIB_ID=Ubuntu, and in line 4, check if it's equivalent to ubuntu or not. If the condition is true, continue and check the release version. Then in line 7, type all of the characters after the 16th character, which is the version. In top example is 20.04. Finally, line 8 will return the ubuntu and version.

```
#https://github.com/aquasecurity/fanal/blob/main/analyzer/os/ubuntu/ubuntu.go
1func (a ubuntuOSAnalyzer) Analyze(_ context.Context, input
    analyzer.AnalysisInput) (*analyzer.AnalysisResult, error) {
   isUbuntu := false
2  scanner := bufio.NewScanner(input.Content)
3  for scanner.Scan() {
      line := scanner.Text()
4     if line == "DISTRIB_ID=Ubuntu" {
         isUbuntu = true
         continue
      }

5     if isUbuntu && strings.HasPrefix(line, "DISTRIB_RELEASE=") {
6        return &analyzer.AnalysisResult{
            OS: &types.OS{
               Family: aos.Ubuntu,
7              Name:   strings.TrimSpace(line[16:]),
            },
   }
8  return nil, xerrors.Errorf("ubuntu: %w", aos.AnalyzeOSError)
}
```

Table 4.2shows the different operating system packages that Trivy support:

Number	OS packages	Resources updater
1	Alpine	https://secdb.alpinelinux.org
2	AWS Linux	https://alas.aws.amazon.com/,
3	Debian	https://salsa.debian.org/security-tracker-team/security-tracker
4	RHEL	https://www.redhat.com/security/data/metrics/
5	Oracle	https://linux.oracle.com/security/oval/
6	Ubuntu	https://git.launchpad.net/ubuntu-cve-tracker
7	Suse	https://ftp.suse.com/pub/projects/security/cvrf
8	Photon OS	"https://packages.vmware.com/photon/photon_cve_metadata/
9	CBL-Mariner	https://github.com/microsoft/CBL-MarinerVulnerabilityData
10	AlmaLinux	https://errata.almalinux.org/
11	Rocky Linux	https://download.rockylinux.org/pub/rocky

Table 4.2: Trivy resources OS pdater

4.2.3 Identifying packages

Trivy will find vulnerabilities in application and operating system packages. Trivy uses files and a directory to find out a package. To get information about the package, should check some directories located in "var/lib/dpkg/" The following const shows the directories that Trivy will parse and look for package information.

Trivy uses two different functions to return the list and status of packages. The first function is "parseDpkgInfoList" that used to get the list of packages. This function get file path and parses " /var/lib/dpkg/info/*.list". Then by the scan function in line 2, check the file path. This scan function uses (Regular Expression)RegEX to check the file path. This function has different rules. For example, file content contains a keyword and should scan, or the file path should scan by this special rule until able to find the secret. This function adds a file if it is still a directory. In line 4, this function checks if the path is equal to the "/" and continues until the file is not a directory and in Line 5, check the installed file. In line 8, add the last file. In line 10, return the list of installed files.

Second function is parseDpkgStatus function that will parses the " /var/lib/dpkg/status or /var/lib/dpkg/status/*". source version and name of package are available of dpkg and Trivy use it to find package and version. for Alpine, this file is situated in "lib/apk/db/installed".

```
# https://github.com/aquasecurity/fanal/blob/main/analyzer/pkg
/dpkg/dpkg.go
const (
   version = 2
   statusFile = "var/lib/dpkg/status"
   statusDir = "var/lib/dpkg/status.d/"
   infoDir   = "var/lib/dpkg/info/"
)
1 func (a dpkgAnalyzer) parseDpkgInfoList(scanner *bufio.Scanner)
    (*analyzer.AnalysisResult, error) {
```

```
   var installedFiles []string
   var previous string
2  for scanner.Scan() {
3     current := scanner.Text()
4     if current == "/." {
         continue
      }
5     if !strings.HasPrefix(current, previous+"/") {
6        installedFiles = append(installedFiles, previous)
      }
7     previous = current
   }

   // Add the last file
8  installedFiles = append(installedFiles, previous)

9  if err := scanner.Err(); err != nil {
      return nil, xerrors.Errorf("scan error: %w", err)
   }

10 return &analyzer.AnalysisResult{
      SystemInstalledFiles: installedFiles,
   }, nil
}
```

4.2.4 Identifying vulnerabilities in discovered application packages

Line 2 returns a driver type according to library type by NewDriwer function to detect vulnerabilities in libraries. For example, if the library type is pip or pipenv, the related driver type is a python package. If was not able to find packages, return the error. Line 4 uses a detect function that gets the driver and filetypes package and returns the vulnerability. This detect function with Small "d" uses another function to find vulnerability based on the type and version of the driver. Finally, detect function will return the vulns variable, which is the output of Detect function.

```
#https://github.com/aquasecurity/trivy/blob/main/pkg/detector/
library/detect.go
1 func Detect(libType string, pkgs []ftypes.Package)
    ([]types.DetectedVulnerability, error) {
2  driver, err := NewDriver(libType)
3  if err != nil {
      return nil, xerrors.Errorf("failed to new driver: %w", err)
   }

4  vulns, err := detect(driver, pkgs)
5  if err != nil {
6     return nil, xerrors.Errorf("failed to scan %s vulnerabilities:
    %w", driver.Type(), err)
   }
```

Table 4.3: Trivy resources application updater

Number	packages	Resources Updater
1	PHP	https://github.com/FriendsOfPHP/security-advisories https://github.com/advisories(composer)
2	Python	https://github.com/pyupio/safety-db https://github.com/advisories(pip)
3	Ruby	https://github.com/rubysec/ruby-advisory-db https://github.com/advisories(Arubygems)
4	node.js	https://github.com/nodejs/security-wg https://github.com/advisories(Anpm)
5	Rust	https://github.com/RustSec/advisory-db
6	Java	https://github.com/advisories(Amaven) https://gitlab.com/gitlab-org/advisories-community
7	Go	https://gitlab.com/gitlab-org/advisories-community
8	.Net	https://gitlab.com/gitlab-org/advisories-community

```
  return vulns, nil
}
```

4.2.5 Identifying vulnerabilities in discovered OS packages

After discovering the operating system and repository, and package type, Trivy tries to detect vulnerabilities. In line 2, use the newDriver function to recognize the operating system family, supported or not. If there is a supported OS, then put it in the driver variable; otherwise, it returns an unsupported OS error. In line 4, the supported version uses IsSupportedVersion function that gets the operating system family and name and checks that this OS family can scan using the operating system scanner. In other words, this function checks whether this OS version is in the EOL list or not. For example, Ubuntu has its scanner and detect function in their file. If this operating system is on the list, then call detect function for this special operating system detect function for each operating system will get OS version, repository and package type as input and return detected vulnerability as an output.

```
#https://github.com/aquasecurity/trivy/blob/main/pkg/detector/
ospkg/detect.go
1 func (d Detector) Detect(_, osFamily, osName string, repo
    *ftypes.Repository, _ time.Time, pkgs []ftypes.Package)
    ([]types.DetectedVulnerability, bool, error) {
2  driver, err := newDriver(osFamily)
   if err != nil {
3     return nil, false, ErrUnsupportedOS
   }

4  eosl := !driver.IsSupportedVersion(osFamily, osName)
```

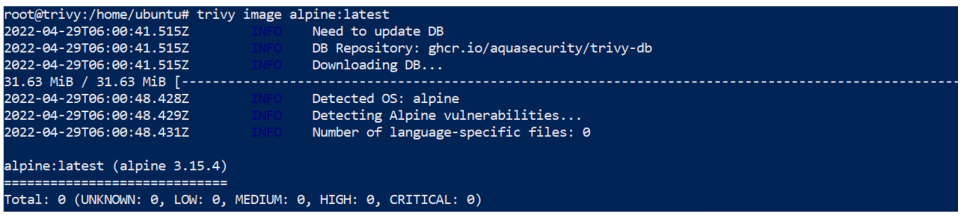
```
5  vulns, err := driver.Detect(osName, repo, pkgs)
6  if err != nil {
       return nil, false, xerrors.Errorf("failed detection: %w", err)
   }

7  return vulns, eosl, nil
}
```

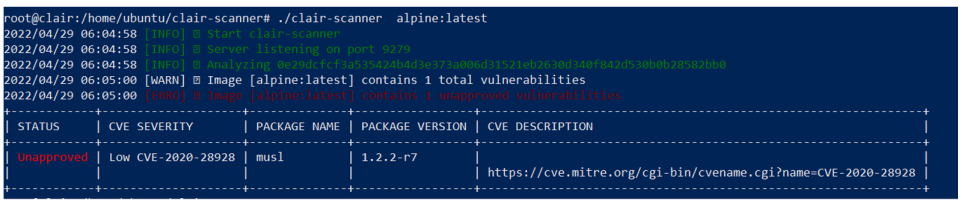
4.3 Experiments

One of the main approaches to evaluating software is installing and doing experimental analysis. This section has tried to set up some different kinds of images and scan them with image scanners. There are three different types of images selected for scanning and experiment. Standard images, Distroless images, and slimmed down one. The goal for choosing these different kinds of images was popularity and security. The first standard images are widely used images from registries.

The second set is Distroless images which are lightweight without a package manager. The main reason to select Distroless images was that most image scanners use the package manager to find packages and then vulnerabilities in packages. We tried to evaluate the scanner. Scanners can find vulnerabilities or how they interact with this kind of image. The third type, a slim one, removes the package manager and some files that most scanners are looking for to find the package. The slim version of the containers has been used docker-slim to remove the extra layer. The result represents how the scanner interacts with images and finds vulnerabilities.



A)- Alpine:latest scanned by Trivy



B)- Alpine:latest scanned by Clair

Figure 4.4: Number of vulnerabilities in Alpine images

Table 4.4: Number of vulnerabilities by image scanner

Image name	Tag	Trivy	Clair	ignore-unfixed
alpine	latest	0	1	0
redis	latest	76	76	2
mysql	latest	182	182	1
node.js	latest	980	980	20
nginx	latest	130	130	2
python	latest	1023	1023	20
debian	latest	76	76	2
ubuntu	latest	17	17	4
java	latest	1612	12	754
postgres	latest	109	109	2

4.3.1 Scanning standard images

In this part, some selected images will be provided to both image scanners and compare their results. All of the images are the latest tag. Table 4.4shows the number of vulnerabilities that Trivy and Clair, as a vulnerability scanner, detect. It's clear that most of the results are the same, and both found the same number of vulnerabilities in images. One notable number is Alpine vulnerability which is different. Clair found one vulnerability which is CVE-2020-28928, with low severity. Trivy scans the image and not able to detect any vulnerability. Trivy shttps://secdb.alpinelinux.org as a data source for vulnerability in Alpine, while Clair uses https://cve.mitre.org/ as a resource for matching vulnerabilities. They are Alpine Linux databases and contain the vulnerable package; however, they include a different number of vulnerabilities.

The significant difference in the number of vulnerabilities is in the Java image. Trivy detected more than 1600 vulnerabilities, while Clair's result shows that this image contains 12 vulnerabilities. This result was predictable because Trivy can scan the application package and have a data source. Clair supports python programming language and detects a vulnerability in Python packages. Trivy uses GitHub Advisory Database for Python packages, a security vulnerability database from open-source software. Clair uses safety DB as a database for python package vulnerabilities.

4.3.2 Scanning distroless images

Google provides distroless images that don't contain package managers, shells, or extra applications. Most distroless images use Debian as a base operating system[20]. The use of images in the application lifecycle will increase security. These so-called distroless images remove as much as possible from OS but still have two major components. In the container, these components include userspace, like the C library(glibc, muslc)and encryption libraries.

The second component is the kernel that runs on hardware or

Table 4.5: Size and Vulnerabilities in distroless images

Distroless Image	Size(MB)	Detected by Trivy	Detected by Clair
base-debian11	124	79	79
gcr.io/distroless/base-debian11	20.03	14	0
Python3	920	1038	1038
gcr.io/distroless/python3	54.20	47	0
java	643	1612	12
gcr.io/distroless/java	210	33	0
nodejs	995	980	980
gcr.io/distroless/nodejs	166	14	0

VM. In other words, distroless images have necessary packages in the image. Therefore, they are minimal in size. The smallest one is gcr.io/distroless/static-debian11, around 2 MB, while the Debian image is 124 MB [20]. Smaller size means lower CPU, RAM, and resources. This smaller size has a considerable benefit in cloud computing, such as fewer network costs and faster pull and deployment.

First, to experiment, pull some examples of distroless images from The Google cloud platform and compare them with the same standard images. After pulling dirstroless images into the system have tried to scan them with both Trivy and Clair scanners.

Table 4.5 compares four different container images group in some features like size and number of vulnerabilities. A Trivy scanner scanned these images. It shows that Trivy analyzes files and doesn't just look at installed packages. Distroless images do not have status file in /var/lib/dpkg/status.but they do still contain glibc, openssl, libssl, etc.

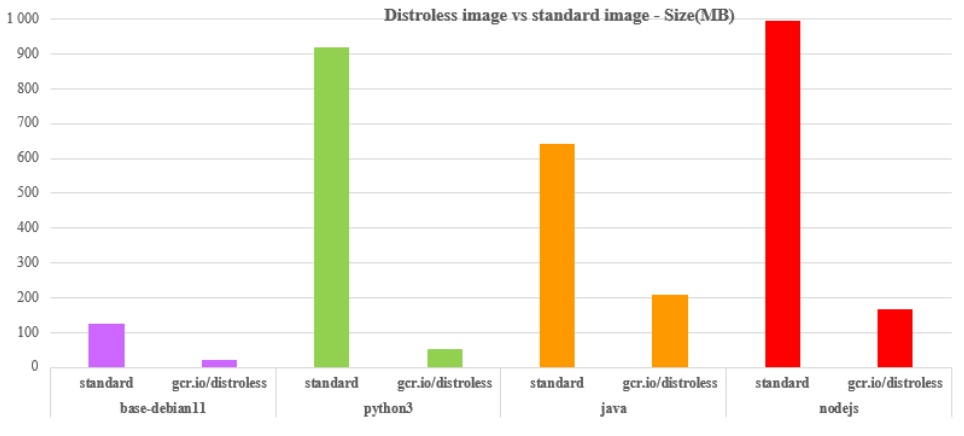


Figure 4.5: Distroless image size(MB)

When compare normal and distroless images, Its clear that in all of the distoless images size and vulnerabilities reduced significantly. in normal Application images java have the most number of vulnerability

which is more than 1600 and following by python packages with 1038 vulnerabilities. In normal images debian with 79 vulnerabilities are the lowest while decrease to 14 in distroless images. The highest number of vulnerability in distroless images is 47 which is 22 vulnerabilities less than debian as a lowest number in normal images. This 4.6 bar chart compares the number of vulnerabilities detected in distroless images with standard images by Trivy.

After that, pulled the images into the system to experiment with the Clair scanner. When starting a scan with Clair, It not able to detect vulnerabilities. Clair uses a package manager to detect vulnerabilities, and as far as distroless images don't contain any package manager or even shell, Clair cannot discover any malware. Figure . 4.7 represents the result of scanning the distroless image with Clair. When comes to size all of 4 distroless images are less than nodejs image. The bar chart . 4.5 compares the size of standard images with distroless images.

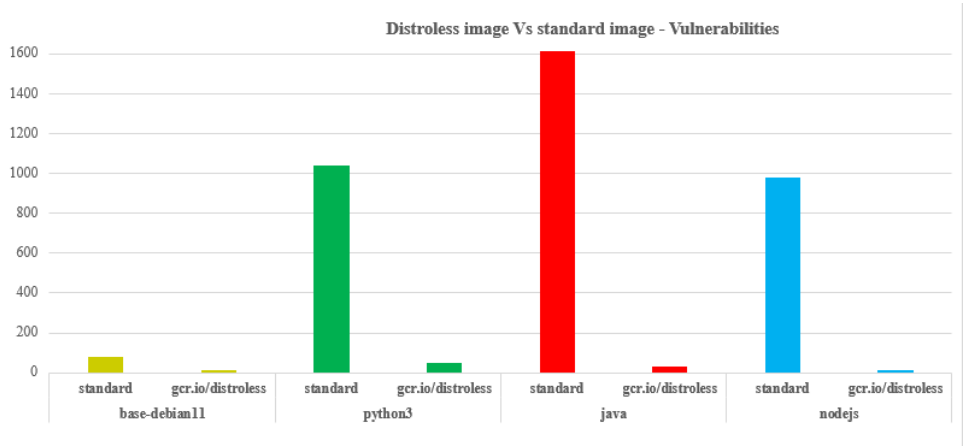


Figure 4.6: Distroless image vulnerabilities

```
ubuntu@clair: $ sudo su
root@clair:/home/ubuntu# cd clair-scanner/
root@clair:/home/ubuntu/clair-scanner# ./clair-scanner  gcr.io/distroless/static-debian11
2022/05/09 07:48:27 [INFO] ▶ Start clair-scanner
2022/05/09 07:48:28 [INFO] ▶ Server listening on port 9279
2022/05/09 07:48:28 [INFO] ▶ Analyzing bfc8fe66ac7cf1f9cd06d1461a1073af51f94fc8b61db01af62362fc6df7b2de
root@clair:/home/ubuntu/clair-scanner#
```

Figure 4.7: Distroless image scan by Clair

4.3.3 Scanning slimmed images

One of the primary techniques to increase the security in the container images is using some slimmer tools to remove the extra layers. They optimize the image container, which causes them smaller and more secure. There is some different application that use analysis technique to optimize container images. Fig. 4.8 shows that the nginx container was 142MB in size and, after using slimmer, decreased to 21.1MB, around 11X smaller than

usual. Some of this application like docker slim minimize containers up to 30X [22].

```
root@trivy:/home/ubuntu# docker images
REPOSITORY    TAG       IMAGE ID       CREATED         SIZE
nginx.slim    latest    3cafc578d2bc   3 minutes ago   12.1MB
nginx         latest    fa5269854a5e   11 days ago     142MB
root@trivy:/home/ubuntu#
```

Figure 4.8: Slim nginx conatiner with Docker-slim

Hardening images for all organizations that use containers are a part of IT security strategy. One approach could be using images with more minor vulnerabilities or adding image scanners in CI/CD pipeline. Distroless images improve security with less layer and vulnerabilities. Instead, another problem arise with them and their maintenance. As we mentioned, they don't have any package manager or bash. Still, they need to update base dependencies and proper maintenance. as mentioned before, the Alpine base image is around 5 Mb in size and is considered the most secure image. It is possible to install the library and customize your own images based on your need.

Chapter 5

Discussion

5.1 Comparison of Trivy and Clair

In this chapter, we first compare some features in Trivy and Clair, then discuss experiments and results. Both Trivy and Clair are open-source scanners that analyze images and provide vulnerability reports. Some factors could be important when comparing them.

5.1.1 Accuracy

Image scanner detects vulnerabilities and reports them along with severity. The accuracy rate represents how well it could use for the security team to find vulnerabilities and patch them. Accuracy is an essential factor for the scanner to give a proper report. when compare the result of scanner together, It shows that both of them are accurate. It's clear that most of the results are the same, and both found the same number of vulnerabilities in images. One notable number is Alpine vulnerability which is different. Clair found one vulnerability which is CVE-2020-28928, with low severity. Trivy scans the image and not able to detect any vulnerability. Trivy shttps://secdb.alpinelinux.org as a data source for vulnerability in Alpine, while Clair uses https://cve.mitre.org/ as a resource for matching vulnerabilities. They are Alpine Linux databases and contain the vulnerable package; however, they include a different number of vulnerabilities.

The significant difference in the number of vulnerabilities is in the Java image. Trivy detected more than 1600 vulnerabilities, while Clair's result shows that this image contains 12 vulnerabilities. This result was predictable because Trivy can scan the application package and have a data source. Clair supports python programming language and detects a vulnerability in Python packages. Trivy uses GitHub Advisory Database for Python packages, a security vulnerability database from open-source software. Clair uses safety DB as a database for python package vulnerabilities. When it comes to distroless images, Clair not able to find vulnerabilities while Trivy support these kind of images. Trivy using the file scanning instead of looking for special location.

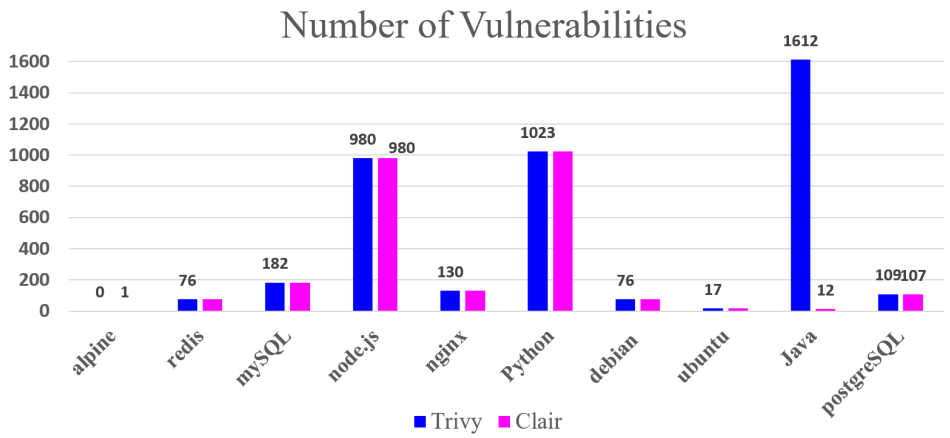


Figure 5.1: Number of detected vulnerabilities by Clair and Trivy

5.1.2 Unfixed vulnerabilities

Container security scanners detect a lot of vulnerabilities, and as could be seen in table 4.4 applications like java and python have more than 1000 vulnerabilities. After scanning the images, some Linux distributions like Debian latest have 76 vulnerabilities. As we mentioned before, some Linux distributions like Ubuntu and Red Hat will provide information related to CVEs and how to patch them. After the investigation into detected vulnerabilities is released, most of them are unfixed. It means that the vendors, regardless of upstream, did not provide any patch for vulnerabilities. These vulnerabilities have low severity or do not significantly affect the application.

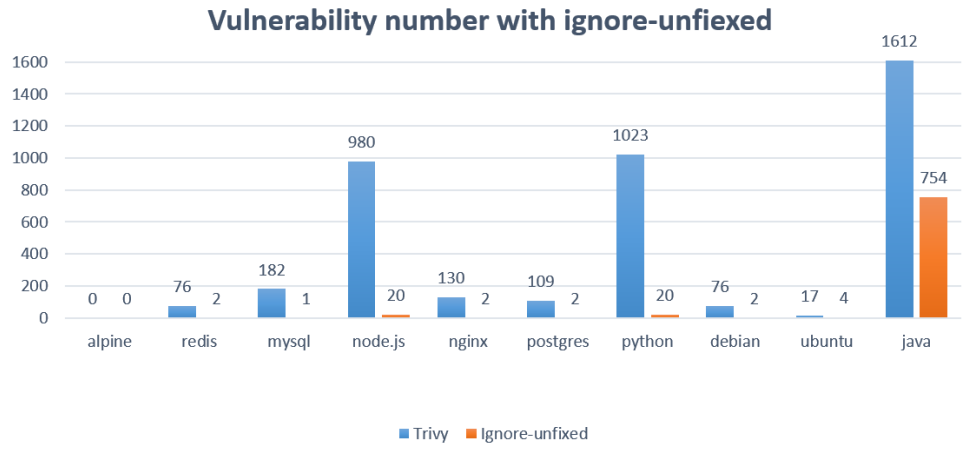


Figure 5.2: Number of vulnerabilities with ignore-unfixed option

Trivy provides options like "–ignore-unfixed" that can compare the number of vulnerabilities without this option. Here is an example of using this option:

```
root@trivy:/home/ubuntu#  trivy image --ignore-unfixed ubuntu:latest
```

Fig. 5.2 represents by using the –ignore-unfixed option, the number of vulnerabilities will decrease significantly. The highest decreasing number of vulnerabilities is related to nodejs and python. Both have around 1000 in the number of vulnerabilities and have reduced to 20 with unfixed. By using the unfixed option, all of the vulnerabilities in images are reduced to 20 or less except java image. It shows that most of the detected vulnerabilities by the scanner are unfixed. Vulnerability reports provide all vulnerabilities, and the security team in every organization could review the list and ensure that the image could leave with unpatched issues.

5.1.3 Number of support packages

There is essential to support a variety of vendors from OS packages and application packages. Clair detects vulnerabilities in OS packages that are present in table 4.1. Clair can only find vulnerabilities in python packages, while Trivy has more data sources and support more application packages. Trivy supports most of the OS packages and has a plan to support windows. The list of supported OS packages is present in table4.2. Trivy is also able to detect vulnerabilities in distroless images. The list of application packages that Trivy support can be see in table4.3.

5.1.4 Third party databases

Database plays a vital role in every application, especially scanning tools. All the packages will check with the database and return report. Clair relay on Postgres database, which is content based storage. It is a fundamental element in Clair scanner to deliver the service, and if the database goes down, it is likely unable to provide service anymore. Red Hat recommends having automatic replication and fail-over for Postgres database[26]. Running another PostgresSQL instance needs more resources and maintenance. Trivy uses Trivy-db, which is available on GitHub. This CLI tool has all vulnerabilities from different sources like NVD, Debian, and update resources that are mentioned in 4.3and manipulates the vulnerability database. Trivy-db is a key-value database and the internal Update is every 6 hours. When you start scanning images, the first step is updating the build-in policy and then scanning the images.

5.1.5 Future threats

Trivy and Clair are open-source projects which do static analysis in container images to detect vulnerabilities. Clair was the first developed scanner and was created in 2015. Trivy was developed in 2019, and the same year was acquired by Aqua security. One of the main advantages of open source software is that it doesn't involve copyright and is free to use. Although Aqua is a cloud security company and tries to secure a cloud-native Environment, It's a potential risk that this company will change its vision in the future and not provide Trivy as an open-source license. They

can remove the code and then start to offer a commercial to the users and organizations.

5.2 Future work

During research about this project and experimenting with some tools, there are the following steps for future work.

Future work could be using a vulnerability scanner in the software development lifecycle. Both Trivy and Clair have a standalone mode suitable for CI/CD pipeline. This method can automate the process and perform scanning images that build. Integrating them within the CI/CD could be interesting, and check which ones fit into this pipeline.

Finally, during the experiment and scan images, when using a slimmer scanner not able to find vulnerabilities. However, Trivy provides some new features compared to the old scanner, such as scanning file systems and Git repositories. There seems to be a strong need for an effective vulnerability scanner that relies not only on particular files.

Chapter 6

Conclusion

The main goal of this thesis was to address the following research questions:

1. **How do container image scanners detect vulnerabilities?**
2. **To what extent are current open source container image scanners able to detect vulnerabilities?**

Scanning the image at first glance is very simple. but there are many images, and each of them has an unsimilar kernel, shell, package, and other components and needs a different implementation for each distribution. The results show that most available images have vulnerabilities and need to scan before using them in software development. Clair uses Claircore as the main component and a Postgres database for saving the results. Trivy main component is fanal and uses Trivy-db, a separate repository located on GitHub. Both Trivy and Clair are modular and use similar algorithms to detect vulnerabilities. They look at specific files and retrieve information about the Os and application packages. Python package is the only application that Clair can find vulnerabilities.

Most of the vulnerabilities reported by Trivy and Clair are unfixed vulnerabilities. As these vulnerabilities don't considerably impact the application, vendors do not provide patches for them. Most likely, they do not significantly affect the systems or applications. The algorithms reveal that image scanners rely on the same file systems to find vulnerabilities. Experiments confirm that when using some slimmer, like docker-slim, which removes the extra libraries, neither Trivy nor Clair can scan the images and find vulnerabilities.

in conclusion, there is no doubt that the container images should be unthreatened. The solution could be using some secure image like alpine as a base image and optimizing it.

Bibliography

[1] */security/what-is-cve*. 2021. URL: https://www.redhat.com/en/topics/security/what-is-cve (visited on 23/02/2022).

[2] *About the OCI*. 2021. URL: https://opencontainers.org/about/overview// (visited on 19/02/2022).

[3] Ashish Aggarwal and Pankaj Jalote. 'Integrating static and dynamic analysis for detecting vulnerabilities'. In: *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*. Vol. 1. IEEE. 2006, pp. 343–350.

[4] Amr Amin et al. 'Androshield: Automated android applications vulnerability detection, a hybrid static and dynamic analysis approach'. In: *Information* 10.10 (2019), p. 326.

[5] *Anchore Image Scanner*. 2019. URL: https://anchore.com/ (visited on 09/02/2022).

[6] *Application container market"*. 2021. URL: https://www.marketsandmarkets.com/Market-Reports/%20application-container-market-182079587.htm (visited on 19/01/2022).

[7] *aquasecurity.CNCF.io*. 2022. URL: https://www.cncf.io/online-programs/trivy-open-source-scanner-for-container-images-just-download-and-run/ (visited on 06/02/2022).

[8] *aquasecurity.github.io*. 2022. URL: https://aquasecurity.github.io/trivy/v0.23.0/ (visited on 06/02/2022).

[9] Thanh Bui. 'Analysis of docker security'. In: *arXiv preprint arXiv:1501.02967* (2015).

[10] Saikat Chakraborty et al. 'Deep learning based vulnerability detection: Are we there yet'. In: *IEEE Transactions on Software Engineering* (2021).

[11] *Clair Documentation*. 2022. URL: https://quay.github.io/clair/whatis.html (visited on 21/02/2022).

[12] *clair-scanner*. 2020. URL: https://github.com/arminc/clair-scanner (visited on 21/02/2022).

[13] *clamav engine/*. 2022. URL: https://www.clamav.net/ (visited on 06/02/2022).

[14] *Common Vulnerabilities and Exposures*. 2022. URL: https://en.wikipedia.org/wiki/Common_Vulnerabilities_and_Exposures (visited on 09/03/2022).

[15] *Container images and registries*. 2022. URL: https://www.redhat.com/en/topics/cloud-native-apps/what-is-a-container-registry (visited on 06/03/2022).

[16] *Container images and registries*. 2022. URL: https://docs.microsoft.com/en-us/azure/container-registry (visited on 06/03/2022).

[17] *container-security-scanners/*. 2021. URL: https://geekflare.com/container-security-scanners/ (visited on 29/01/2022).

[18] *Content-addressable storage*. 2022. URL: https://en.wikipedia.org/wiki/Content-addressable_storage (visited on 31/03/2022).

[19] *dagda image scanner*. 2022. URL: https://github.com/eliasgranderubio/dagda/ (visited on 26/01/2022).

[20] *Distroless Container Images*. 2022. URL: https://github.com/GoogleContainerTools/distroless (visited on 07/05/2022).

[21] *Docker containers vulnerability scan*. 2022. URL: https://go.dev/doc/ (visited on 25/04/2022).

[22] *DockerSlim in Githubn*. 2022. URL: https://github.com/docker-slim/docker-slim (visited on 03/05/2022).

[23] jonathan luu Emilien socchi. 'A deep dive into Docker Hub's security landscape "'. In: (2019).

[24] *Falco, the cloud-native runtime security project,/*. 2022. URL: https://falco.org/ (visited on 09/02/2022).

[25] Laurent Gallon. 'On the impact of environmental metrics on CVSS scores'. In: *2010 IEEE Second International Conference on Social Computing*. IEEE. 2010, pp. 987–992.

[26] *Get Postgres and Clair*. 2022. URL: https://access.redhat.com/documentation/en-us/red_hat_quay/2.9/html/manage_red_hat_quay/clair-initial-setup (visited on 04/05/2022).

[27] *Go programming language*. 2022. URL: https://github.com/arminc/clair-scanner (visited on 21/04/2022).

[28] *History container*. 2020. URL: https://www.redhat.com/en/blog/history-containers (visited on 02/02/2022).

[29] *how Anchore Enterprise work*. 2022. URL: https://docs.anchore.com/current (visited on 22/02/2022).

[30] Delu Huang et al. 'Security analysis and threats detection techniques on docker container'. In: *2019 IEEE 5th International Conference on Computer and Communications (ICCC)*. IEEE. 2019, pp. 1214–1220.

[31] Vipin Jain et al. 'Static Vulnerability Analysis of Docker Images'. In: *IOP Conference Series: Materials Science and Engineering*. Vol. 1131. 1. IOP Publishing. 2021, p. 012018.

[32] Omar Javed and Salman Toor. 'Understanding the Quality of Container Security Vulnerability Detection Tools'. In: *arXiv preprint arXiv:2101.03844* (2021).

[33] Bhupinder Kaur et al. 'An analysis of security vulnerabilities in container images for scientific data analysis'. In: *GigaScience* 10.6 (2021), giab025.

[34] Si-yao Liu, Qiang Li and Bin Li. 'Research on isolation of container based on Docker technology'. In: *Computer Engineering & Software* 36 (2015), pp. 110–113.

[35] MarkusLinnalampi. 'outdated software in container images "'. In: (2021).

[36] Shannon Meier et al. 'IBM systems virtualization: Servers, storage, and software'. In: *IBM Redbook, May* (2008).

[37] *National Vulnerability Database*. 2022. URL: https://nvd.nist.gov/ (visited on 21/02/2022).

[38] *OCI Runtime and Lifecycle*. 2022. URL: https://github.com/opencontainers/runtime-spec/blob/main/runtime.md (visited on 03/03/2022).

[39] *OCI specefication*. 2022. URL: https://github.com/opencontainers/image-spec (visited on 19/02/2022).

[40] *Overview of CVE history*. 2022. URL: https://www.cve.org/About/History#Overview (visited on 09/03/2022).

[41] Amit M Potdar et al. 'Performance evaluation of docker container and virtual machine'. In: *Procedia Computer Science* 171 (2020), pp. 1419–1428.

[42] *Qualitative severity rating scale*. 2022. URL: https://www.first.org/cvss/specification-document (visited on 21/02/2022).

[43] Sogand Shirinbab, Lars Lundberg and Emiliano Casalicchio. 'Performance evaluation of container and virtual machine running cassandra workload'. In: *2017 3rd International Conference of Cloud Computing Technologies and Applications (CloudTech)*. IEEE. 2017, pp. 1–8.

[44] Sari Sultan, Imtiaz Ahmad and Tassos Dimitriou. 'Container security: Issues, challenges, and the road ahead'. In: *IEEE Access* 7 (2019), pp. 52976–52996.

[45] Chris Swan. *Docker drops lxc as default execution environment*. 2019.

[46] *The first introudction of docker*. 2020. URL: https://www.docker.com/ (visited on 26/01/2022).

[47] Anshu Tripathi and Umesh Kumar Singh. 'On prioritization of vulnerability categories based on CVSS scores'. In: *2011 6th International Conference on Computer Sciences and Convergence Information Technology (ICCIT)*. IEEE. 2011, pp. 692–697.

[48] *Trivy Database in Githubn*. 2022. URL: https://github.com/aquasecurity/trivy-db (visited on 29/04/2022).

[49] Olufogorehan Tunde-Onadele et al. 'A study on container vulnerability exploit detection'. In: *2019 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE. 2019, pp. 121–127.

[50] Steven J Vaughan-Nichols. 'Docker libcontainer unifies Linux container powers'. In: *ZDNet, June* (2014).

[51] *What is Snyk*. 2022. URL: https://docs.snyk.io/ (visited on 10/03/2022).

[52] Katrine Wist, Malene Helsem and Danilo Gligoroski. 'Vulnerability analysis of 2500 docker hub images'. In: *Advances in Security, Networks, and Internet of Things*. Springer, 2021, pp. 307–327.

[53] Robail Yasrab. 'Mitigating docker security issues'. In: *arXiv preprint arXiv:1804.05039* (2018).

[54] Ruoyu Zhang et al. 'Combining static and dynamic analysis to discover software vulnerabilities'. In: *2011 Fifth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*. IEEE. 2011, pp. 175–181.

Chapter 7

Appendix

7.1 Clair source code

7.1.1 Identiy the OS

Listing 7.1: Ubuntu as an example

```
#https://github.com/quay/claircore/blob/main/ubuntu/releases.go
package ubuntu

import (
   "github.com/quay/claircore"
)

type Release string

const (
   Artful  Release = "artful" // deprecated
   Bionic  Release = "bionic"
   Cosmic  Release = "cosmic"
   Disco   Release = "disco"
   Precise Release = "precise" // deprecated
   Trusty  Release = "trusty"
   Xenial  Release = "xenial"
   Eoan    Release = "eoan"
   Focal   Release = "focal"
   Impish  Release = "impish"
)

var AllReleases = map[Release]struct{}{
   Artful: struct{}{},
   Bionic: struct{}{},
   Cosmic: struct{}{},
   Disco:  struct{}{},
   Precise: struct{}{},
   Trusty: struct{}{},
   Xenial: struct{}{},
   Eoan:   struct{}{},
   Focal:  struct{}{},
```

```
    Impish: struct{}{},
}

var ReleaseToVersionID = map[Release]string{
    Artful: "17.10",
    Bionic: "18.04",
    Cosmic: "18.10",
    Disco:  "19.04",
    Precise: "12.04",
    Trusty: "14.04",
    Xenial: "16.04",
    Eoan:   "19.10",
    Focal:  "20.04",
    Impish: "21.10",
}

var artfulDist = &claircore.Distribution{
    Name:           "Ubuntu",
    Version:        "17.10 (Artful Aardvark)",
    DID:            "ubuntu",
    PrettyName:     "Ubuntu 17.10",
    VersionID:      "17.10",
    VersionCodeName: "artful",
}

var bionicDist = &claircore.Distribution{
    Name:           "Ubuntu",
    Version:        "18.04.3 LTS (Bionic Beaver)",
    DID:            "ubuntu",
    PrettyName:     "Ubuntu 18.04.3 LTS",
    VersionID:      "18.04",
    VersionCodeName: "bionic",
}

var cosmicDist = &claircore.Distribution{
    Name:           "Ubuntu",
    Version:        "18.10 (Cosmic Cuttlefish)",
    DID:            "ubuntu",
    VersionID:      "18.10",
    VersionCodeName: "cosmic",
    PrettyName:     "Ubuntu 18.10",
}

var discoDist = &claircore.Distribution{
    Name:           "Ubuntu",
    Version:        "19.04 (Disco Dingo)",
    DID:            "ubuntu",
    VersionID:      "19.04",
    VersionCodeName: "disco",
    PrettyName:     "Ubuntu 19.04",
}

var preciseDist = &claircore.Distribution{
    Name:       "Ubuntu",
```

```
    Version:    "12.04.5 LTS, Precise Pangolin",
    DID:        "ubuntu",
    VersionID: "12.04",
    PrettyName: "Ubuntu precise (12.04.5 LTS)",
}

var trustyDist = &claircore.Distribution{
    Name:       "Ubuntu",
    Version:    "14.04.6 LTS, Trusty Tahr",
    DID:        "ubuntu",
    PrettyName: "Ubuntu 14.04.6 LTS",
    VersionID: "14.04",
}

var xenialDist = &claircore.Distribution{
    Name:           "Ubuntu",
    Version:        "16.04.6 LTS (Xenial Xerus)",
    DID:            "ubuntu",
    PrettyName:     "Ubuntu 16.04.6 LTS",
    VersionID:      "16.04",
    VersionCodeName: "xenial",
}

var eoanDist = &claircore.Distribution{
    Name:           "Ubuntu",
    Version:        "19.10 (Eoan Ermine)",
    DID:            "ubuntu",
    PrettyName:     "Ubuntu 19.10",
    VersionID:      "19.10",
    VersionCodeName: "eoan",
}

var focalDist = &claircore.Distribution{
    Name:           "Ubuntu",
    Version:        "20.04 LTS (Focal Fossa)",
    DID:            "ubuntu",
    PrettyName:     "Ubuntu 20.04 LTS",
    VersionID:      "20.04",
    VersionCodeName: "focal",
}

var impishDist = &claircore.Distribution{
    Name:           "Ubuntu",
    Version:        "21.10 (Impish Indri)",
    DID:            "ubuntu",
    PrettyName:     "Ubuntu 21.10",
    VersionID:      "21.10",
    VersionCodeName: "impish",
}

func releaseToDist(r Release) *claircore.Distribution {
    switch r {
    case Artful:
        return artfulDist
```

```
    case Bionic:
        return bionicDist
    case Cosmic:
        return cosmicDist
    case Disco:
        return discoDist
    case Precise:
        return preciseDist
    case Trusty:
        return trustyDist
    case Xenial:
        return xenialDist
    case Eoan:
        return eoanDist
    case Focal:
        return focalDist
    case Impish:
        return impishDist
    default:
        // return empty dist
        return &claircore.Distribution{}
    }
}
```

7.1.2 Identify the package

Listing 7.2: Identify the package by scanner

```
#https://github.com/quay/claircore/blob/main/dpkg/scanner.go
package dpkg

import (
    "bufio"
    "context"
    "crypto/md5"
    "encoding/hex"
    "errors"
    "fmt"
    "io"
    "io/fs"
    "net/textproto"
    "path/filepath"
    "runtime/trace"
    "strings"

    "github.com/quay/zlog"

    "github.com/quay/claircore"
    "github.com/quay/claircore/internal/indexer"
    "github.com/quay/claircore/pkg/tarfs"
)
```

```
const (
   name    = "dpkg"
   kind    = "package"
   version = "4"
)

var (
   _ indexer.VersionedScanner = (*Scanner)(nil)
   _ indexer.PackageScanner = (*Scanner)(nil)
)

type Scanner struct{}


func (ps *Scanner) Name() string { return name }

func (ps *Scanner) Version() string { return version }

func (ps *Scanner) Kind() string { return kind }
// It does not respect any dpkg configuration files.
func (ps *Scanner) Scan(ctx context.Context, layer
    *claircore.Layer) ([]*claircore.Package, error) {
   // Preamble
   defer trace.StartRegion(ctx, "Scanner.Scan").End()
   trace.Log(ctx, "layer", layer.Hash.String())
   ctx = zlog.ContextWithValues(ctx,
      "component", "dpkg/Scanner.Scan",
      "version", ps.Version(),
      "layer", layer.Hash.String())
   zlog.Debug(ctx).Msg("start")
   defer zlog.Debug(ctx).Msg("done")


   rd, err := layer.Reader()
   if err != nil {
      return nil, fmt.Errorf("opening layer failed: %w", err)
   }
   defer rd.Close()
   sys, err := tarfs.New(rd)
   if err != nil {
      return nil, fmt.Errorf("opening layer failed: %w", err)
   }

   loc := make(map[string]int)
   walk := func(p string, d fs.DirEntry, err error) error {
      if err != nil {
         return err
      }
      switch dir, f := filepath.Split(p); {
      case f == "status" && !d.IsDir():
         loc[dir]++
      case f == "info" && d.IsDir():
         loc[dir]++
      }
```

```
    return nil
}

if err := fs.WalkDir(sys, ".", walk); err != nil {
    return nil, err
}
zlog.Debug(ctx).Msg("scanned for possible databases")

// If we didn't find anything, this loop is completely skipped.
var pkgs []*claircore.Package
for p, x := range loc {
    if x != 2 { // If we didn't find both files, skip this
        directory.
        continue
    }
    ctx = zlog.ContextWithValues(ctx, "database", p)
    zlog.Debug(ctx).Msg("examining package database")

    // We want the "status" file.
    fn := filepath.Join(p, "status")
    db, err := sys.Open(fn)
    switch {
    case errors.Is(err, nil):
    case errors.Is(err, fs.ErrNotExist):
        zlog.Debug(ctx).
            Str("filename", fn).
            Msg("false positive")
        continue
    default:
        return nil, fmt.Errorf("reading status file from layer
            failed: %w", err)
    }

    found := make(map[string]*claircore.Package)

    tp := textproto.NewReader(bufio.NewReader(db))
     Restart:
    hdr, err := tp.ReadMIMEHeader()
    for ; err == nil && len(hdr) > 0; hdr, err =
        tp.ReadMIMEHeader() {
        name := hdr.Get("Package")
        v := hdr.Get("Version")
        p := &claircore.Package{
            Name:      name,
            Version:   v,
            Kind:      claircore.BINARY,
            Arch:      hdr.Get("Architecture"),
            PackageDB: fn,
        }
        if src := hdr.Get("Source"); src != "" {
            p.Source = &claircore.Package{
                Name: src,
                Kind: claircore.SOURCE,
                Version:  v,
```

```
        PackageDB: fn,
      }
   }

   found[name] = p
   pkgs = append(pkgs, p)
}
switch {
case errors.Is(err, io.EOF):
default:
   zlog.Warn(ctx).Err(err).Msg("unable to read entry")
   goto Restart
}

const suffix = ".md5sums"
ms, err := fs.Glob(sys, filepath.Join(p, "info", "*"+suffix))
if err != nil {

   return nil, fmt.Errorf("resetting tar reader failed: %w",
       err)
}
hash := md5.New()
for _, n := range ms {
   k := strings.TrimSuffix(filepath.Base(n), suffix)
   if i := strings.IndexRune(k, ':'); i != -1 {
      k = k[:i]
   }
   p, ok := found[k]
   if !ok {
      zlog.Debug(ctx).
         Str("package", k).
         Msg("extra metadata found, ignoring")
      continue
   }
   f, err := sys.Open(n)
   if err != nil {
      return nil, fmt.Errorf("unable to open file %q: %w", n,
          err)
   }
   hash.Reset()
   _, err = io.Copy(hash, f)
   f.Close()
   if err != nil {
      zlog.Warn(ctx).
         Err(err).
         Str("package", n).
         Msg("unable to read package metadata")
      continue
   }
   p.RepositoryHint = hex.EncodeToString(hash.Sum(nil))
}
zlog.Debug(ctx).
   Int("count", len(found)).
   Msg("found packages")
```

```
    }

    return pkgs, nil
}
```

7.1.3 Identifying the OS-release

Listing 7.3: Claircore OS-release

```
#https://github.com/quay/claircore/blob/main/osrelease/scanner.go
package osrelease

import (
    "bufio"
    "bytes"
    "context"
    "fmt"
    "io"
    "runtime/trace"
    "sort"
    "strings"

    "github.com/quay/zlog"

    "github.com/quay/claircore"
    "github.com/quay/claircore/internal/indexer"
    "github.com/quay/claircore/pkg/cpe"
    "github.com/quay/claircore/pkg/tarfs"
)

const (
    scannerName    = "os-release"
    scannerVersion = "2"
    scannerKind    = "distribution"
)

// Path and FallbackPath are the two documented locations for the
    os-release

const (
    Path         = `etc/os-release`
    FallbackPath = `usr/lib/os-release`
)

var (
    _ indexer.DistributionScanner = (*Scanner)(nil)
    _ indexer.VersionedScanner = (*Scanner)(nil)
)

// Scanner implements a scanner.DistributionScanner that examines
    os-release
```

```
type Scanner struct{}


func (*Scanner) Name() string { return scannerName }


func (*Scanner) Version() string { return scannerVersion }


func (*Scanner) Kind() string { return scannerKind }

// Scan reports any found os-release Distribution information in
    the provided
// layer.

func (s *Scanner) Scan(ctx context.Context, l *claircore.Layer)
    ([]*claircore.Distribution, error) {
   defer trace.StartRegion(ctx, "Scanner.Scan").End()
   ctx = zlog.ContextWithValues(ctx,
      "component", "osrelease/Scanner.Scan",
      "version", s.Version(),
      "layer", l.Hash.String())
   zlog.Debug(ctx).Msg("start")
   defer zlog.Debug(ctx).Msg("done")

   r, err := l.Reader()
   if err != nil {
      return nil, fmt.Errorf("osrelease: unable to open layer: %w",
          err)
   }
   defer r.Close()
   sys, err := tarfs.New(r)
   if err != nil {
      return nil, fmt.Errorf("osrelease: unable to open layer: %w",
          err)
   }

   // Attempt to parse each os-release file encountered. On a
       successful parse,
   // return the distribution.
   var rd io.Reader
   for _, n := range []string{Path, FallbackPath} {
      f, err := sys.Open(n)
      if err != nil {
         zlog.Debug(ctx).
            Str("name", n).
            Err(err).
            Msg("unable to open file")
         continue
      }
      defer f.Close()
      rd = f
      break
   }
```

```
	if rd == nil {
		zlog.Debug(ctx).Msg("didn't find an os-release file")
		return nil, nil
	}
	d, err := toDist(ctx, rd)
	if err != nil {
		return nil, err
	}
	return []*claircore.Distribution{d}, nil
}

// ToDist returns the distribution information from the file
    contents provided on
// r.
func toDist(ctx context.Context, r io.Reader)
    (*claircore.Distribution, error) {
	ctx = zlog.ContextWithValues(ctx,
		"component", "osrelease/parse")
	defer trace.StartRegion(ctx, "parse").End()
	m, err := Parse(ctx, r)
	if err != nil {
		return nil, err
	}
	d := claircore.Distribution{
		Name: "Linux",
		DID:  "linux",
	}
	ks := make([]string, 0, len(m))
	for key := range m {
		ks = append(ks, key)
	}
	sort.Strings(ks)
	for _, key := range ks {
		value := m[key]
		switch key {
		case "ID":
			zlog.Debug(ctx).Msg("found ID")
			d.DID = value
		case "VERSION_ID":
			zlog.Debug(ctx).Msg("found VERSION_ID")
			d.VersionID = value
		case "BUILD_ID":
		case "VARIANT_ID":
		case "CPE_NAME":
			zlog.Debug(ctx).Msg("found CPE_NAME")
			wfn, err := cpe.Unbind(value)
			if err != nil {
				zlog.Warn(ctx).
					Err(err).
					Str("value", value).
					Msg("failed to unbind the cpe")
				break
			}
			d.CPE = wfn
```

```
      case "NAME":
         zlog.Debug(ctx).Msg("found NAME")
         d.Name = value
      case "VERSION":
         zlog.Debug(ctx).Msg("found VERSION")
         d.Version = value
      case "ID_LIKE":
      case "VERSION_CODENAME":
         zlog.Debug(ctx).Msg("found VERISON_CODENAME")
         d.VersionCodeName = value
      case "PRETTY_NAME":
         zlog.Debug(ctx).Msg("found PRETTY_NAME")
         d.PrettyName = value
      case "REDHAT_BUGZILLA_PRODUCT":
         zlog.Debug(ctx).Msg("using RHEL hack")
         d.PrettyName = value
      }
   }
   zlog.Debug(ctx).Str("name", d.Name).Msg("found dist")
   return &d, nil
}

// Parse splits the contents r into key-value pairs as described in
// os-release(5).
//
// See comments in the source for edge cases.
func Parse(ctx context.Context, r io.Reader) (map[string]string,
    error) {
   ctx = zlog.ContextWithValues(ctx, "component", "osrelease/Parse")
   defer trace.StartRegion(ctx, "Parse").End()
   m := make(map[string]string)
   s := bufio.NewScanner(r)
   s.Split(bufio.ScanLines)
   for s.Scan() && ctx.Err() == nil {
      b := s.Bytes()
      switch {
      case len(b) == 0:
         continue
      case b[0] == '#':
         continue
      }
      eq := bytes.IndexRune(b, '=')
      if eq == -1 {
         return nil, fmt.Errorf("osrelease: malformed line %q",
             s.Text())
      }
      key := strings.TrimSpace(string(b[:eq]))
      value := strings.TrimSpace(string(b[eq+1:]))

         switch value[0] {
      case '\'':
         value = strings.TrimFunc(value, func(r rune) bool { return
             r == '\'' })
         value = strings.ReplaceAll(value, `'\''`, `'`)
```

```
      case '"':

         value = strings.TrimFunc(value, func(r rune) bool { return
             r == '"' })
         value = dqReplacer.Replace(value)
      default:
      }

      m[key] = value
   }
   if err := s.Err(); err != nil {
      return nil, err
   }
   if err := ctx.Err(); err != nil {
      return nil, err
   }
   return m, nil
}

var dqReplacer = strings.NewReplacer(
   "\\`", "`",
   `\\`, `\`,
   `\"`, `"`,
   `\$`, `$`,
)
```

7.1.4 Identify the Vulnerabilities

Listing 7.4: Identify the vulnerabilities

```
#https://github.com/quay/claircore/blob/main/internal/matcher/match.go
package matcher

import (
   "context"
   "encoding/json"
   "runtime"
   "sync/atomic"

   "github.com/quay/zlog"
   "golang.org/x/sync/errgroup"

   "github.com/quay/claircore"
   "github.com/quay/claircore/internal/vulnstore"
   "github.com/quay/claircore/libvuln/driver"
)

// Match receives an IndexReport and creates a VulnerabilityReport
    containing matched vulnerabilities
func Match(ctx context.Context, ir *claircore.IndexReport, matchers
    []driver.Matcher, store vulnstore.Vulnerability)
    (*claircore.VulnerabilityReport, error) {
```

```
// the vulnerability report we are creating
vr := &claircore.VulnerabilityReport{
   Hash:                 ir.Hash,
   Packages:             ir.Packages,
   Environments:         ir.Environments,
   Distributions:        ir.Distributions,
   Repositories:         ir.Repositories,
   Vulnerabilities:      map[string]*claircore.Vulnerability{},
   PackageVulnerabilities: map[string][]string{},
}

// extract IndexRecords from the IndexReport
records := ir.IndexRecords()
// a channel where concurrent controllers will deliver
    vulnerabilities affecting a package.
// maps a package id to a list of vulnerabilities.
ctrlC := make(chan map[string][]*claircore.Vulnerability, 1024)
// a channel where controller errors will be reported
errorC := make(chan error, 1024)
// fan out all controllers, write their output to ctrlC, close
    ctrlC once all writers finish
go func() {
   defer close(ctrlC)
   var g errgroup.Group
   for _, m := range matchers {
      mm := m
      g.Go(func() error {
         mc := NewController(mm, store)
         vulns, err := mc.Match(ctx, records)
         if err != nil {
            return err
         }
         // in event of slow reader go routines will block
         ctrlC <- vulns
         return nil
      })
   }
   if err := g.Wait(); err != nil {
      errorC <- err
   }
}()
// loop ranges until ctrlC is closed and fully drained, ctrlC is
    guaranteed to close
for vulnsByPackage := range ctrlC {
   for pkgID, vulns := range vulnsByPackage {
      for _, vuln := range vulns {
         vr.Vulnerabilities[vuln.ID] = vuln
         vr.PackageVulnerabilities[pkgID] =
             append(vr.PackageVulnerabilities[pkgID], vuln.ID)
      }
   }
}
select {
case err := <-errorC:
```

```
      return nil, err
   default:
   }
   return vr, nil
}

// Store is the interface that can retrieve Enrichments and
    Vulnerabilities.
type Store interface {
   vulnstore.Vulnerability
   vulnstore.Enrichment
}

// EnrichedMatch receives an IndexReport and creates a
    VulnerabilityReport
// containing matched vulnerabilities and any relevant enrichments.
func EnrichedMatch(ctx context.Context, ir *claircore.IndexReport,
    ms []driver.Matcher, es []driver.Enricher, s Store)
    (*claircore.VulnerabilityReport, error) {
   // the vulnerability report we are creating
   vr := &claircore.VulnerabilityReport{
      Hash:                 ir.Hash,
      Packages:             ir.Packages,
      Environments:         ir.Environments,
      Distributions:        ir.Distributions,
      Repositories:         ir.Repositories,
      Vulnerabilities:      map[string]*claircore.Vulnerability{},
      PackageVulnerabilities: map[string][]string{},
      // The Enrichments member isn't constructed here because it's
      // constructed separately and then added.
   }
   // extract IndexRecords from the IndexReport
   records := ir.IndexRecords()
   lim := runtime.GOMAXPROCS(0)

   // Set up a pool to run matchers
   mCh := make(chan driver.Matcher)
   vCh := make(chan map[string][]*claircore.Vulnerability, lim)
   mg, mctx := errgroup.WithContext(ctx) // match group, match
       context
   for i := 0; i < lim; i++ {
      mg.Go(func() error { // Worker
         var m driver.Matcher
         for m = range mCh {
            select {
            case <-mctx.Done():
               return mctx.Err()
            default:
            }
            vs, err := NewController(m, s).Match(mctx, records)
            if err != nil {
               zlog.Error(ctx).
                  Err(err).
                  Msg("matcher error")
```

```
            continue
         }
         vCh <- vs
      }
      return nil
   })
}
// Set up a pool to watch the matchers and attach results to the
    report.
var vg errgroup.Group
vg.Go(func() error { // Pipeline watcher
Send:
   for _, m := range ms {
      select {
      case <-mctx.Done():
         break Send
      case mCh <- m:
      }
   }
   close(mCh)
   defer close(vCh)
   if err := mg.Wait(); err != nil {
      return err
   }
   return nil
})
vg.Go(func() error { // Collector
   for pkgVuln := range vCh {
      for pkg, vs := range pkgVuln {
         for _, v := range vs {
            vr.Vulnerabilities[v.ID] = v
            vr.PackageVulnerabilities[pkg] =
                append(vr.PackageVulnerabilities[pkg], v.ID)
         }
      }
   }
   return nil
})
if err := vg.Wait(); err != nil {
   return nil, err
}

// Set up a pool to run the enrichers and attach results to the
    report.
eCh := make(chan driver.Enricher)
type entry struct {
   kind string
   msg  []json.RawMessage
}
rCh := make(chan *entry, lim)
eg, ectx := errgroup.WithContext(ctx)
eg.Go(func() error { // Sender
Send:
   for _, e := range es {
```

```
            select {
            case eCh <- e:
            case <-ectx.Done():
               break Send
            }
         }
         close(eCh)
         return nil
      })
      eg.Go(func() error { // Collector
         em := make(map[string][]json.RawMessage)
         for e := range rCh {
            em[e.kind] = append(em[e.kind], e.msg...)
         }
         vr.Enrichments = em
         return nil
      })
      // Use an atomic to track closing the results channel.
      ct := uint32(lim)
      for i := 0; i < lim; i++ {
         eg.Go(func() error { // Worker
            defer func() {
               if atomic.AddUint32(&ct, ^uint32(0)) == 0 {
                  close(rCh)
               }
            }()
            var e driver.Enricher
            for e = range eCh {
               kind, msg, err := e.Enrich(ectx, getter(s, e.Name()), vr)
               if err != nil {
                  zlog.Error(ctx).
                     Err(err).
                     Msg("enrichment error")
                  continue
               }
               if len(msg) == 0 {
                  zlog.Debug(ctx).
                     Str("name", e.Name()).
                     Msg("enricher reported nothing, skipping")
                  continue
               }
               res := entry{
                  msg: msg,
                  kind: kind,
               }
               select {
               case rCh <- &res:
               case <-ectx.Done():
                  return ectx.Err()
               }
            }
            return nil
         })
      }
```

```
   if err := eg.Wait(); err != nil {
      return nil, err
   }

   return vr, nil
}

// Getter returns a type implementing driver.EnrichmentGetter.
func getter(s vulnstore.Enrichment, name string) *enrichmentGetter {
   return &enrichmentGetter{s: s, name: name}
}

type enrichmentGetter struct {
   s    vulnstore.Enrichment
   name string
}

var _ driver.EnrichmentGetter = (*enrichmentGetter)(nil)

func (e *enrichmentGetter) GetEnrichment(ctx context.Context, tags
    []string) ([]driver.EnrichmentRecord, error) {
   return e.s.GetEnrichment(ctx, e.name, tags)
}
```

7.2 Trivy source code

7.2.1 Identify the OS

Listing 7.5: Ubuntu OS release.

```
#https://github.com/aquasecurity/fanal/blob/main/analyzer/os/ubuntu/ubuntu.go
package ubuntu

import (
   "bufio"
   "context"
   "os"
   "strings"

   "golang.org/x/xerrors"

   "github.com/aquasecurity/fanal/analyzer"
   aos "github.com/aquasecurity/fanal/analyzer/os"
   "github.com/aquasecurity/fanal/types"
   "github.com/aquasecurity/fanal/utils"
)

func init() {
   analyzer.RegisterAnalyzer(&ubuntuOSAnalyzer{})
}
```

```
const version = 1

var requiredFiles = []string{"etc/lsb-release"}

type ubuntuOSAnalyzer struct{}

func (a ubuntuOSAnalyzer) Analyze(_ context.Context, input
    analyzer.AnalysisInput) (*analyzer.AnalysisResult, error) {
   isUbuntu := false
   scanner := bufio.NewScanner(input.Content)
   for scanner.Scan() {
      line := scanner.Text()
      if line == "DISTRIB_ID=Ubuntu" {
         isUbuntu = true
         continue
      }

      if isUbuntu && strings.HasPrefix(line, "DISTRIB_RELEASE=") {
         return &analyzer.AnalysisResult{
            OS: &types.OS{
               Family: aos.Ubuntu,
               Name:   strings.TrimSpace(line[16:]),
            },
         }, nil
      }
   }
   return nil, xerrors.Errorf("ubuntu: %w", aos.AnalyzeOSError)
}

func (a ubuntuOSAnalyzer) Required(filePath string, _ os.FileInfo)
    bool {
   return utils.StringInSlice(filePath, requiredFiles)
}

func (a ubuntuOSAnalyzer) Type() analyzer.Type {
   return analyzer.TypeUbuntu
}

func (a ubuntuOSAnalyzer) Version() int {
   return version
```

7.2.2 Identify the packages

Listing 7.6: Discover the application

```
#parseDpkgStatus parses /var/lib/dpkg/status or
    /var/lib/dpkg/status/*
#https://github.com/aquasecurity/fanal/blob/main/analyzer/pkg/dpkg/dpkg.go
```

```
func (a dpkgAnalyzer) parseDpkgStatus(filePath string, scanner
    *bufio.Scanner) (*analyzer.AnalysisResult, error) {
   var pkg *types.Package
   pkgMap := map[string]*types.Package{}

   for scanner.Scan() {
      line := strings.TrimSpace(scanner.Text())
      if line == "" {
         continue
      }

      pkg = a.parseDpkgPkg(scanner)
      if pkg != nil {
         pkgMap[pkg.Name+"-"+pkg.Version] = pkg
      }
   }

   if err := scanner.Err(); err != nil {
      return nil, xerrors.Errorf("scan error: %w", err)
   }

   pkgs := make([]types.Package, 0, len(pkgMap))
   for _, p := range pkgMap {
      pkgs = append(pkgs, *p)
   }

   return &analyzer.AnalysisResult{
      PackageInfos: []types.PackageInfo{
         {
            FilePath: filePath,
            Packages: pkgs,
         },
      },
   }, nil
}
```

7.2.3 Identify vulnerabilities in the application packages

Listing 7.7: Identify vulnerability in applicaiton package

```
#https://github.com/aquasecurity/trivy/blob/main/pkg/detector/library/detect.go
package library

import (
   "golang.org/x/xerrors"

   ftypes "github.com/aquasecurity/fanal/types"
   "github.com/aquasecurity/trivy/pkg/types"
)

// Detect scans and returns vulnerabilities of library
```

```
func Detect(libType string, pkgs []ftypes.Package)
    ([]types.DetectedVulnerability, error) {
   driver, err := NewDriver(libType)
   if err != nil {
      return nil, xerrors.Errorf("failed to new driver: %w", err)
   }

   vulns, err := detect(driver, pkgs)
   if err != nil {
      return nil, xerrors.Errorf("failed to scan %s vulnerabilities:
          %w", driver.Type(), err)
   }

   return vulns, nil
}

func detect(driver Driver, libs []ftypes.Package)
    ([]types.DetectedVulnerability, error) {
   var vulnerabilities []types.DetectedVulnerability
   for _, lib := range libs {
      vulns, err := driver.DetectVulnerabilities(lib.Name,
          lib.Version)
      if err != nil {
         return nil, xerrors.Errorf("failed to detect %s
             vulnerabilities: %w", driver.Type(), err)
      }

      for i := range vulns {
         vulns[i].Layer = lib.Layer
         vulns[i].PkgPath = lib.FilePath
      }
      vulnerabilities = append(vulnerabilities, vulns...)
   }

   return vulnerabilities, ni
```

7.2.4 Identify vulnerabilities in the OS packages

Listing 7.8: Detect Vulnerability in Ubuntu

```
# Detect function scans and returns the vulnerabilities
#
    https://github.com/aquasecurity/trivy/blob/main/pkg/detector/ospkg/detect.go

func (s *Scanner) Detect(osVer string, _ *ftypes.Repository, pkgs
    []ftypes.Package) ([]types.DetectedVulnerability, error) {
   log.Logger.Info("Detecting Ubuntu vulnerabilities...")
   log.Logger.Debugf("ubuntu: os version: %s", osVer)
   log.Logger.Debugf("ubuntu: the number of packages: %d",
       len(pkgs))
```

```
    var vulns []types.DetectedVulnerability
    for _, pkg := range pkgs {
        advisories, err := s.vs.Get(osVer, pkg.SrcName)
        if err != nil {
            return nil, xerrors.Errorf("failed to get Ubuntu
                advisories: %w", err)
        }

        installed := utils.FormatSrcVersion(pkg)
        installedVersion, err := version.NewVersion(installed)
        if err != nil {
            log.Logger.Debugf("failed to parse Ubuntu installed package
                version: %w", err)
            continue
        }

        for _, adv := range advisories {
            vuln := types.DetectedVulnerability{
                VulnerabilityID: adv.VulnerabilityID,
                PkgName:         pkg.Name,
                InstalledVersion: installed,
                FixedVersion:    adv.FixedVersion,
                Layer:           pkg.Layer,
                Custom:          adv.Custom,
                DataSource:      adv.DataSource,
            }

            if adv.FixedVersion == "" {
                vulns = append(vulns, vuln)
                continue
            }

            fixedVersion, err := version.NewVersion(adv.FixedVersion)
            if err != nil {
                log.Logger.Debugf("failed to parse Ubuntu package
                    version: %w", err)
                continue
            }

            if installedVersion.LessThan(fixedVersion) {
                vulns = append(vulns, vuln)
            }
        }
    }
    return vulns, nil
}

// IsSupportedVersion checks is OSFamily can be scanned using
    Ubuntu scanner
func (s *Scanner) IsSupportedVersion(osFamily, osVer string) bool {
    eol, ok := eolDates[osVer]
    if !ok {
        log.Logger.Warnf("This OS version is not on the EOL list: %s
            %s", osFamily, osVer)
```

```
    return false
  }
  return s.clock.Now().Before(eol)
}
```