BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

VULNERABILITY ASSESSMENT OF CONTAINER IMAGES
DETEKCE ZRANITELNOSTÍ V KONTEJNEROVÝCH OBRAZECH

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR   Bc. MICHAL FINDRA
AUTOR PRÁCE

SUPERVISOR   Ing. JIŘÍ PAVELA
VEDOUCÍ PRÁCE

BRNO 2024

Master's Thesis Assignment

Institut:	Department of Intelligent Systems (DITS)	157039

Student:	**Findra Michal, Bc.**
Programme:	Information Technology and Artificial Intelligence
Specialization:	Cybersecurity
Title:	**Vulnerability Assessment of Container Images**
Category:	Security
Academic year:	2023/24

Assignment:

1. Study the topic of vulnerability assessment and detection within container images.
2. Get acquainted with existing tools for vulnerability assessment in container images, such as Syft, Grype, TheLateSyft or Vulntron.
3. Design and implement automated vulnerability assessment of container images within the Vulntron tool. The tool must periodically retrieve, parse and scan images found within a given namespace, and report the detected vulnerabilities through an API or other tools within the Red Hat pipeline (e.g., DefectDojo).
4. Evaluate the scalability of the tool, that is, whether it manages to analyze all namespace images within the analysis time frame (e.g., 24 hours). Discuss the advantages, shortcomings and possible future work of the resulting tool.

Literature:
- Repozitář nástroje Syft: https://github.com/anchore/syft
- Repozitář nástroje Grype: https://github.com/anchore/grype
- Repozitář nástroje TheLateSyft: https://github.com/xxlhacker/TheLateSyft

Requirements for the semestral defence:
First two points of the assignment.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Pavela Jiří, Ing.**
Consultant:	Gábor Bürgés
Head of Department:	Hanáček Petr, doc. Dr. Ing.
Beginning of work:	1.11.2023
Submission deadline:	17.5.2024
Approval date:	6.11.2023

Abstract

The work focuses on the problem of automated security analysis of container images in a distributed environment. It describes present vulnerabilities in these environments and tools that deal with the analysis of container images that serve as a template for deploying a specific container. The process involves acquiring an environment description and subsequently processing it into a format meaningful for Vulntron tool developed as part of this thesis. Vulntron automates this process, performs a security analysis of individual components of the container image, and generates a report in a visually and technically processable format. The thesis also includes practical integration in form of Vulntron deployment into various types of development processes within the Red Hat company.

Abstrakt

Práca sa zaoberá automatizovaným bezpečnostným rozborom kontajnerových obrazov v distribuovanom prostredí. Popísané sú aktuálne zraniteľnosti v týchto prostrediach a nástojenie, ktoré sa zaoberajú analýzou kontajnerových obraz, slúžiacich ako vzor na vytvorenie daného kontajneru. Popísané je získanie popisu prostredia, následného spracovania do formátu zmysluplného pre vyvíjaný nástroj Vulntron. Vulntron slúži na automatizáciu tohoto procesu, bezpečnostnú analýzu jednotlivých komponentov kontajnerového obrazu a následný report do vizuálnej aj technicky ďalej spracovateľnej podoby. Súčasťou implementácie bude aj praktické nasadenie nástroja do rôznych typov vývojového procesu vrámci firmy Red Hat.

Keywords

Vulntron, Container image, Container security, Security analysis, Grype, Syft, SBOM, Vulnerability detection, CI-CD

Klíčová slova

Vulntron, Kontajnerový obraz, Kontajnerová bezpečnosť, Bezpečnostná analýza, Grype, Syft, SBOM, Detekcia zraniteľností, CI-CD

Reference

FINDRA, Michal. *Vulnerability Assessment of Container Images*. Brno, 2024. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Jiří Pavela

Rozšířený abstrakt

Táto práca sa zaoberá automatizovaným bezpečnostným rozborom kontajnerových obrazov s cieľom identifikovať, analyzovať a riešiť aktuálne zraniteľnosti a bezpečnostné hrozby, ktoré sa môžu vyskytnúť v komplexných a dynamických distribuovaných prostrediach. S ohľadom na rýchly rozvoj a nasadenie kontajnerizovaných aplikácií vo viacerých odvetviach je nevyhnutné zabezpečiť, aby boli tieto aplikácie chránené pred potenciálnymi útokmi a zraniteľnosťami, ktoré by mohli ohroziť ich integritu a dostupnosť. Okrem analýzy rizík a hrozieb práca skúma aj aktuálne požiadavky zamerané na zabezpečenie kontajnerových technológií, čo poskytuje základ pre implementáciu bezpečnostných opatrení.

Prvá časť práce sa venuje štúdiu a analýze existujúcich nástrojov a metód pre hodnotenie zraniteľností v kontajnerových obrazoch. Podrobne rozoberá nástroje ako Syft, Grype a ďalšie, ktoré umožňujú efektívne vyhľadávanie a klasifikáciu bezpečnostných nedostatkov. Popisuje tiež rôzne typy zraniteľností, ktoré môžu kontajnerové obrazy obsahovať, a metodiky ich detekcie. V táto časť sa taktiež zaoberá porovnaním efektivity a presnosti týchto nástrojov, čím poskytuje praktické odporúčania pre ich využitie vo vývojových procesoch.

Druhá časť sa zameriava na návrh a implementáciu nástroja Vulntron, ktorý automatizuje proces identifikácie a spracovania zraniteľností. Nástroj Vulntron integruje analýzu zraniteľností priamo do procesov vývoja softvéru a umožňuje dynamické sledovanie a reagovanie na nové hrozby, čím zvyšuje bezpečnosť kontajnerových aplikácií. Nástroj Vulntron bol navrhnutý tak, aby bol schopný efektívne spracovávať veľké množstvo dát a integrovať sa s ďalšími nástrojmi používanými vo vývojovom procese, ako sú CI/CD pipeline. Detailné technické opisy implementácie nástroja Vulntron ukazujú, ako boli riešené požiadavky od tímov a nasadenie nástroja do produkčného prostredia.

Tretia časť práce predstavuje praktické nasadenie a evaluáciu systému Vulntron vo vývojovom prostredí spoločnosti Red Hat. Hodnotí účinnosť systému v rôznych scenároch a analyzuje jeho schopnosť integrovať sa s existujúcimi nástrojmi a procesmi. Práca taktiež obsahuje štúdiu prípadov, kde bol Vulntron úspešne aplikovaný na reálne projekty a jeho dopad na zlepšenie bezpečnostných postupov pri vývoji softvéru.

Záver práce sumarizuje dosiahnuté výsledky a navrhuje možné smerovania pre ďalší vývoj systému Vulntron. Diskutuje o potenciálnych vylepšeniach a rozšíreniach, ktoré by mohli zvýšiť jeho efektivitu a adaptabilitu na meniace sa bezpečnostné hrozby v distribuovaných systémoch. V návrhoch na zlepšenie sa zameriava na rozšírenie funkcionalít systému o nové analytické nástroje a zlepšenie používateľského rozhrania pre intuitívnejšiu administráciu a monitorovanie bezpečnostných udalostí.

Vulnerability Assessment of Container Images

Declaration

I hereby declare that this master's thesis was prepared as an original work by the author under the supervision of Ing. Jiří Pavela. The supplementary information was provided by Mr. Gábor Bürgés. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
Michal Findra
May 14, 2024

Acknowledgements

I would like to express my heartfelt gratitude to my supervisor, Ing. Jiří Pavela, for all the guidance and valuable feedback. I would also like to extend my appreciation to consultant Mr. Gábor Bürgés for their assistance. Additionally, I am deeply thankful to my close friends and family for their unwavering support and encouragement throughout this journey. A FITting verse that provided inspiration: "For I know the plans I have for you, plans to prosper you and not to harm you, plans to give you hope and a future." — Jeremiah 29:11.

Contents

Chapter 1

Introduction

The last few years have seen a dramatic shift in software development and deployment due to the widespread adoption of containerization technologies, best represented by platforms such as Red Hat OpenShift and Docker. Applications can be efficiently packaged and distributed across a wide range of computing environments with the help of containers that are lightweight, portable, and effective. However, as more and more businesses take advantage of containerized applications, possible security issues concerning container images are coming to light.

The main goal of this thesis is to create and implement an automated pipeline that can monitor container images in advance and seamlessly integrate vulnerability assessment tools into the deployment process. By automating this process, the security posture of containerized environments is strengthened, as it addresses the time-sensitive nature of vulnerability identification and remediation. A distinct set of vulnerabilities are introduced by container images that enclose not only the program but also its dependencies and runtime environment. Comprehending and addressing these vulnerabilities is crucial to ensure the availability, confidentiality, and integrity of the programs installed in containers.

The suggested system introduces a user-friendly interface to visualize and comprehend vulnerabilities that have been identified, in addition to streamlining the assessment process. By providing developers and security experts with a clear and user-friendly way to comprehend and react to possible threats, this interface is intended to empower all relevant parties.

In the following chapters, the current state of containerization will be explored, the unique security risks that container images present will be discussed, state-of-the-art vulnerability assessment techniques will be reviewed, and new insights will be provided on strengthening the security posture of containerized applications.

Chapter 2

Containers and Container Orchestration

Particularly in the context of the Linux ecosystem, containers constitute a fundamental paradigm in contemporary computing. These compact, lightweight and effective units contain applications and their dependencies, making them consistent with the deployment in a variety of computing environments. Isolating applications from the underlying system allows for increased flexibility and scalability, which is the core idea of containers.

Docker and Linux Containers (LXC) [3][9] are two well-known containerization technologies. Due to Docker being built on top of LXC and offers a higher-level abstraction, it makes containerization processes easier for developers to handle. LXC offers a low-level container management toolset. Through the use of Linux kernel features like `cgroups` and `namespaces`, containers are able to achieve isolation, which guarantees that every container runs independently while sharing resources with the host system and keeping itself separate from other containers.

The efficiency of containerized applications lies in their ability to share the host OS kernel, optimizing resource utilization. This contrasts with traditional virtualization, where each virtual machine requires a separate OS instance. Containerization streamlines deployment, fosters rapid development cycles, and supports microservices architectures, making it a cornerstone in contemporary software development and deployment practices within the Linux ecosystem.

An image of a container is required initially to utilize a Linux container. It is a file (or files) that contains the environment, which includes configuration files, runtimes, and libraries, along with the application. Dockerfiles, text files with a human-readable description of the image, are the foundation upon which container images are built. In the majority of use cases, users construct their images over the preexisting ones, saving them from having to start from scratch. Starting with a fully-fledged container image, like `Ubuntu` or `UBI8`, which contains all the necessary Linux utilities and package managers, like `DNF` or `APT`, can sometimes be a better option when building the container image. Nevertheless, engineers occasionally utilize lighter (also referred to as minimal) images, like `ubi8-minimal` or `alpine`, when optimizing for image sizes. These images have fewer features, but still enable the installation of all required tools while consuming fewer resources.

When the container image is prepared, it can be unpacked by calling a container engine like Docker or Podman. It initiates what is known as a container process by making an API call to the Linux kernel. The `clone()` system call is used to create this container process.

Due to this, the container is really just a Linux process with additional isolation thanks to kernel namespaces. Only the filesystem and processes created within the same namespace are visible to the Linux container. Apart from the data structures that represent processes and namespaces, the Linux kernel does not contain any data structures that represent containers.

There has been an attempt to standardize every aspect of the workflow as container technologies gained popularity and a plethora of new tools have been developed. The main goals of this standardization have been to guarantee that all of these tools generate and use the same artifacts, that switching container tooling is feasible without causing major problems, and that the containerization investment is independent of vendors.

2.1 Container Orchestration

The following section describes container orchestration with descriptions and definitions of technologies that are closely related to the topic of this thesis. The content of this chapter is derived from current Kubernetes[6][7] and Red Hat OpenShift[14][2] documentations.

Applications are typically deployed on top of a single operating system as a standard procedure. A physical host can be divided into multiple virtual hosts by using virtualization. However, in terms of scalability and operational efficiency, using virtual instances on shared resources might not be the best option. Since a virtual machine (VM) uses resources exactly like a physical machine does, allocating resources like CPU, RAM, and storage to a VM comes at a high cost. The utilization of virtual instances on shared resources may degrade the performance of the application.

Containerization offers a solution by segregating applications within a containerized framework. Containers, similar to virtual machines, possess distinct attributes such as filesystems, virtual CPUs (vCPUs), memory allocations, and dependencies. They are portable across diverse environments and exhibit reduced resource overhead compared to full-fledged operating systems, functioning as lightweight, isolated processes atop the operating system kernel. In contrast, VMs, acting as abstractions of physical hardware, have slower boot times and rely on a hypervisor for execution within a singular machine environment.

The comparison of individual container technologies throughout the years and their structure is shown in Figure 2.1 with more detailed description of individual eras:

Traditional deployment era. In the past, companies used actual servers to run their applications. Resource allocation problems resulted from the inability to define resource boundaries for applications on a physical server. When several applications are running on a single physical server, for instance, there may be times when one of the applications uses all the resources and the others perform worse. Running every application on a different physical server would be one way to solve this. However, because resources were not fully utilized and maintaining numerous physical servers was costly for enterprises, this did not scale.

Virtualized deployment era. Enables to use the CPU of a single physical server to run multiple Virtual Machines. Applications can be isolated within virtual machines thanks to virtualization, which also adds a layer of security by preventing unauthorized access to one application's data by other applications. Better scalability and resource utilization

in a physical server are made possible by virtualization, which also lowers hardware costs and makes it easier to add or update applications. A group of disposable virtual machines can be presented as a cluster of physical resources through the use of virtualization. On top of virtualized hardware, every virtual machine is a complete machine running every component, including its own operating system.

The era of container deployment. Containers share the operating system with other applications due to their minimal isolation properties compared to virtual machines. Containers are therefore regarded as lightweight. A container shares the same CPU, memory, process space, filesystem, and other resources as a virtual machine. They are cross-cloud and cross-OS distribution compatible because they are separated from the underlying infrastructure. A non-exhaustive list of benefits of using container deployments:

- Agile application creation and deployment. Simplified and expedited creation of container images compared to utilizing VM images.

- Continuous development, integration, and deployment. Facilitation of dependable and frequent building and deployment of container images with swift and effective rollbacks, owing to the immutability of images.

- Environmental consistency across development, testing, and production. Uniform operation across diverse environments, ensuring equivalence between local and cloud-based executions.

- Cloud and OS distribution portability. Compatibility with a multitude of operating systems such as Ubuntu, Red Hat Enterprise Linux, CoreOS, spanning on-premises setups, major public cloud platforms, and beyond.

- Loosely coupled, distributed, elastic, liberated micro-services. Fragmentation of applications into smaller, autonomous units, enabling dynamic deployment and management, in contrast to a monolithic configuration reliant on a single-purpose machine.

- Resource isolation. Assured consistency in application performance through isolation of resources.

- Resource utilization. Enhanced efficiency and density in resource utilization.

2.2 Kubernetes

Kubernetes is an open source container orchestration tool developed by Google. Kubernetes can be used to run and manage workloads that are container-based. Using Kubernetes to build cloud-native applications by deploying a collection of interconnected microservices is the most common use case. Kubernetes clusters that span hosts across public, private, on-premise, and hybrid clouds can be established.

Kubernetes facilitates sharing resources, orchestrating containers across multiple hosts, installing new hardware configurations, running health checks and self-healing applications, and scaling containerized applications.
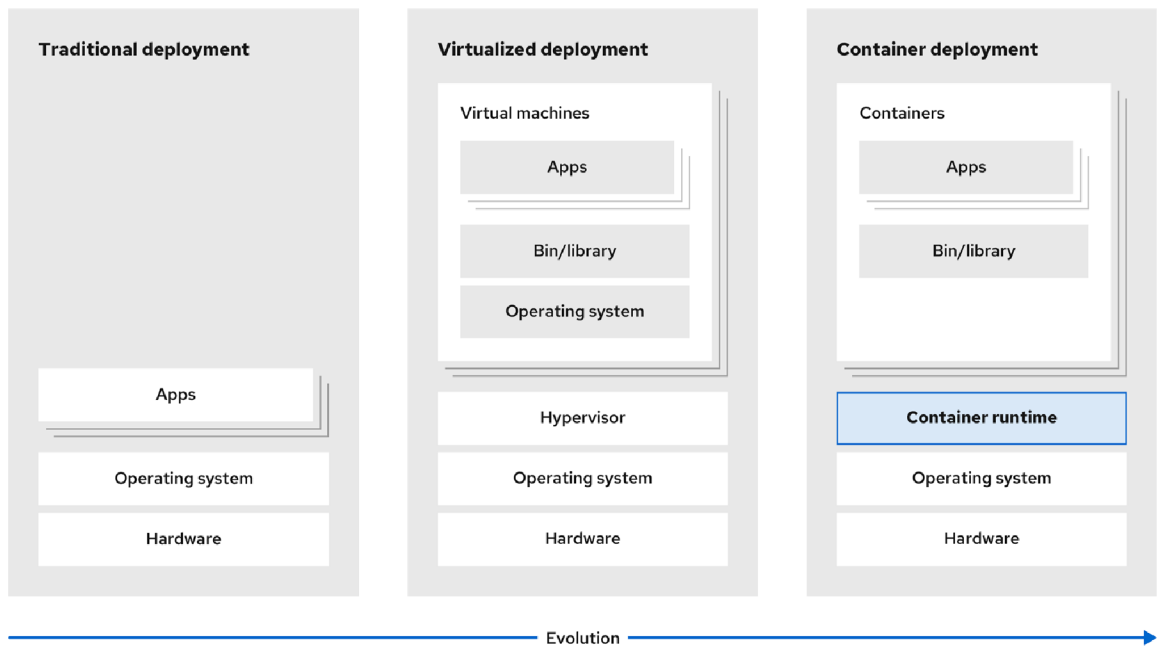
Figure 2.1: Comparison of container technologies throughout the years [14].

2.2.1 Kubernetes Objects and Architecture

Kubernetes objects are persistent entities in the Kubernetes system. Kubernetes uses these entities to represent the state of your cluster. They can describe:

- What containerized applications are running (and on which nodes).

- The resources available to those applications.

- The policies around how those applications behave, such as restart policies, upgrades, and fault-tolerance.

Manipulating a Kubernetes object (create, modify, or delete) is done using the Kubernetes API.

The object specification `spec` and the object `status` are two embedded object fields that determine the configuration of almost all Kubernetes entities. When an object is created, the specification, which is present in objects with a defined spec, must be configured to clarify the desired properties and state of the resource. On the other hand, the status represents the object's current state at that moment, which the Kubernetes system and its component parts keep updated. The Kubernetes control plane continuously and proactively arranges for each object's actual state to match the user's specified desired state.

List 2.1 shows an example initial manifest for a Kubernetes deployment.

```
1  apiVersion: applications/v1
2  kind: Deployment
3  metadata:
4    name: tomcat-deployment
```

```
 5  spec:
 6    selector:
 7      matchLabels:
 8        app: tomcat
 9    replicas: 3
10    template:
11      metadata:
12        labels:
13          app: tomcat
14      spec:
15        containers:
16        - name: tomcat
17          image: tomcat:9.0.46
18          ports:
19          - containerPort: 8080
```

Listing 2.1: An example Kubernetes manifest for deployment of a `Tomcat` application.

After Kubernetes is deployed, e.g. using the deployment manifest similar to one shown in Listing 2.1, a cluster is instantiated.

Cluster. Consists of a group of worker computers, or nodes, that are responsible for running containerized applications. Every cluster includes a minimum of one worker node. The hosting environment for Pods, which make up the functional elements of the application workload, is provided by the worker node. Simultaneously, the control plane coordinates the cluster's worker node and Pod management. To provide fault-tolerance and high availability, a cluster usually consists of multiple nodes, whereas the control plane in an operational context usually spans multiple computing units.

Control Plane Components

The components of the control plane detect and respond to cluster events (e.g., starting up a new Pod when a Deployment's `replicas` field is unsatisfied) and make global decisions about the cluster (e.g., scheduling). Any machine in the cluster can run the components of the control plane. Setup scripts usually start all control plane components on the same machine for simplicity.

The following are the descriptions of the most vital control plane components. The components are also shown in Figure 2.2.

- `kube-apiserver`. The API server is a component of the Kubernetes control plane that exposes the Kubernetes API. The API server is the front-end for the Kubernetes control plane.

- `etcd`. Consistent and highly-available key value store used as Kubernetes' backing store for all cluster data.

- `kube-scheduler`. Control plane component that watches for newly created Pods with no assigned node, and selects a node for them to run on.
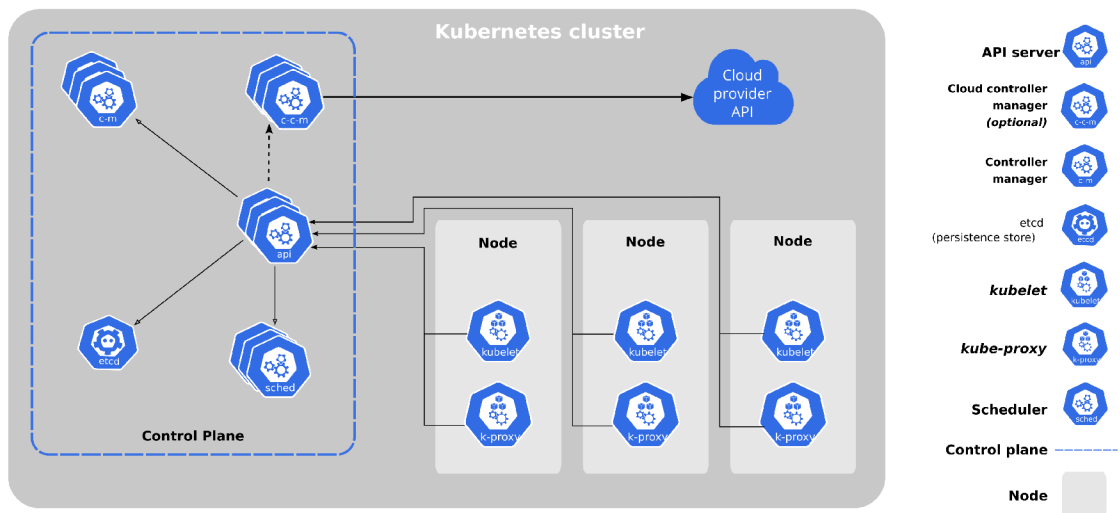
- `kube-controller-manager`. Runs controller process.

7

Figure 2.2: A schema of the Kubernetes cluster architecture [6].

Node Components

Every node has node components that operate on it to maintain Pod operations and provide the Kubernetes runtime environment.

The following are the descriptions of the most vital node components. The components are also shown graphically in Figure 2.2.

- `kubelet`. An agent that runs on each node in the cluster. It makes sure that containers are running in a Pod.

- `kube-proxy`. A network proxy that is individual for each node and maintains network rules on nodes.

- Container runtime. An essential part that enables Kubernetes to efficiently run containers. In the Kubernetes environment, it is in charge of overseeing the execution and lifecycle of containers.

Nodes

The workload is orchestrated by Kubernetes through the deployment of containers inside of Pods, which are then run on nodes. Depending on how the cluster is set up, these nodes could be virtual or real computers. Every node has the necessary services to carry out Pods and is overseen by the control plane.

2.2.2 Containers in Kubernetes

Due to dependencies that promote standardization and guarantee consistent behavior across a range of deployment environments, every executed container embodies repeatability. With containers, it is easier to deploy applications across different cloud platforms or operating systems because they help to detach applications from the host's underlying infrastructure. Individual nodes within a Kubernetes cluster run the containers that make up the Pods

that are assigned to them. To encourage co-location and concurrent execution, containers in a Pod are arranged and scheduled jointly on a single node.

Container images. A pre-built software package that includes all the necessary parts to run an application, such as the codebase, runtime environment, relevant application and system libraries, and default settings for important parameters. Due to their stateless and immutable nature, containers are not meant to have their codebase modified while in use. If there are situations where changes must be made to a containerized application, the recommended process is to create a new image with the necessary changes and then recreate the container to start using the updated image.

Container runtimes. An essential element that enables Kubernetes to efficiently manage containers. It oversees the Kubernetes environment's container execution and lifecycle management.

Container security. The most common attacks on containers [17] include malicious actors who aim to escape the container. It is more common for attackers to exploit known vulnerabilities that remain unpatched. Administrators of containerized environments must regularly scan their container images to ensure that they are free from known and exploitable vulnerabilities. Many of these vulnerabilities are straightforward to exploit, often through automated scripted attacks. Various types of attackers exist, including those who script attacks to identify or exploit vulnerabilities. These individuals may relay information about potential exploits to more sophisticated attackers who assess whether a vulnerability provides a pathway into a container and whether it might allow an escape. However, the initial concern should be the vulnerable dependencies that grant attackers access.

The concept of *defense in depth* involves multiple layers of security between an attacker and the assets being protected. This strategy ensures that breaching one layer does not immediately compromise sensitive data. In the context of container security, numerous layers can be implemented. For example, one might start with a network security layer that restricts specific types or sources of traffic into a deployment. Following this, a network policy could limit the traffic reaching specific containers or groups of containers. Additionally, runtime enforcement can be applied to containers to ensure that they execute only authorized programs and prevent activities like unauthorized cryptomining. Implementing a read-only file system is another layer that prevents attackers who penetrate a container from modifying files.

Another critical security measure is configuring containers to operate non-privileged users instead of running as root. By default, containers operate with root privileges identical to the root user on the host machine. This means that if a container is compromised and the attacker manages to escape, they gain root-level access to the host, potentially leading to significant security breaches.

2.2.3 Workloads

Workloads are an essential part that enables Kubernetes to efficiently run containers. In the Kubernetes environment, it is in charge of overseeing the execution and life cycle of containers.

Pods

Pods are the smallest deployable computing units that you can create and manage in Kubernetes.

A Pod is a collection of one or more containers that share network and storage resources and operating guidelines. The contents of a Pod are always executed in a shared context, co-located, and co-scheduled. A Pod, which consists of one or more closely coupled application containers, represents a *logical host* that is specific to a given application. Applications running on the same physical or virtual machine are comparable to cloud applications running on the same logical host in non-cloud scenarios.

Pods management. Even singleton Pods don't require direct creation. Workload resources like `Deployment` or `Job` are used for their creation. `StatefulSet` resource is used if Pods need to track state.

Only one instance of a particular application is intended to run on each Pod. Multiple Pods, one for each instance, should be used if the application is scaled horizontally (i.e. to provide more overall resources by running more instances). This is commonly referred to as `replication` in Kubernetes. Replicated Pods are typically created and controlled by a workload resource and controller.

A Pod in Kubernetes clusters may be used in two ways:

- Single container Pod. The most popular Kubernetes use case is the „one-container-per-Pod" model, where a Pod acts as a wrapper around a single container, with Kubernetes handling Pod management instead of container management directly.

- Multiple container Pod. An application made up of several co-located containers that must share resources and are closely coupled can be contained within a single Pod. When co-located containers work together, they create a unified unit of service. For example, a container may serve publicly accessible data from a shared volume, while a separate sidecar container updates or refreshes the files. These containers, storage assets, and a transient network identity are all bundled into a single unit by the Pod.

Pods are more like single application containers in that they are temporary rather than permanent. Pods are created, given a unique identifier (UID), and assigned to nodes where they remain until they are terminated, unless the restart policy is followed or until they are removed voluntarily. When a node fails, the Pods associated with that node are deleted after a predetermined amount of time.

Pods are not inherently capable of self-healing. If a Pod is connected to a broken node or experiences a resource shortage that requires eviction, it is immediately removed. Kubernetes uses a higher-level abstraction called a controller to oversee the orchestration of these transient Pod instances in order to manage the dynamic lifecycle of Pods. A Pod identified by its UID cannot be rescheduled to a different node. Alternatively, in the event that it becomes necessary, the Pod might be swapped out for a brand-new, strikingly similar instance that might have a different UID but the same name. When an entity, like a volume, is marked to survive with a Pod, it means that the entity will remain for the duration of the life of that particular Pod, depending on its UID. As such, the end of the corresponding Pod, independent of any further replication, triggers the destruction and subsequent reconstruction of the corresponding entity, represented in this case by a volume.

Storage in Pods. An array of shared storage volumes can be specified by a Pod. The shared volumes in the Pod are accessible to all containers, enabling data sharing amongst them. Additionally, volumes enable the survival of persistent data within a Pod in the event that one of its containers needs to be restarted.

Pod phases. The phase field of an object called `PodStatus` serves as the status field for a Pod.

The Pod phase provides a concise, comprehensive representation of the Pod's lifecycle stage at that particular moment. It is not intended to be an exhaustive state machine, nor to include a comprehensive compilation of container or Pod state observations.

- **Pending.** The Kubernetes cluster has accepted the Pod, but one or more of the containers are not yet configured and operational.

- **Running.** Every container has been instantiated and the Pod has been bound to a node. At least one container is running right now, or it is starting or stopping right now.

- **Succeeded.** Every container in the Pod has completed its activities successfully and will not be restarted.

- **Failed.** Every container in the Pod has shut down, and at least one of them did so in an ineffective manner. This means that the system will either terminate it or provide an exit with a non-zero status.

- **Unknown.** For some reason, it was not possible to determine the condition of the Pod. Errors in communication with the node where the Pod is supposed to run usually result in this phase.

Pod networking. A distinct IP address is given to each Pod for every address family. The IP address and network ports of each container within a Pod are shared by all of the containers in the Pod. `Localhost` is used by the containers that are part of the same Pod for communication between them. The use of shared network resources (like ports) by containers in a Pod must be coordinated when they communicate with entities outside the Pod. Containers can locate one another using localhost and share an IP address and port space within a Pod.

An example network created between one Pod with two containers and the second one with a container is shown in Figure 2.3. Localhost is used to communicate between two containers, and Virtual bridge to communicate between two Pods.

Route. A route is a way to give a service an external hostname so that it can be accessed from the outside. Each route is composed of a service selector, a designated name, and a security configuration. A router can use a pre-established path in conjunction with the endpoints that are recognized by the service to which it is connected in order to provide a unique identifier that allows clients from outside the network to access applications. Even though it is easy to deploy a full multi-tier application, without the routing layer, incoming traffic coming from sources outside the environment cannot reach the application directly.
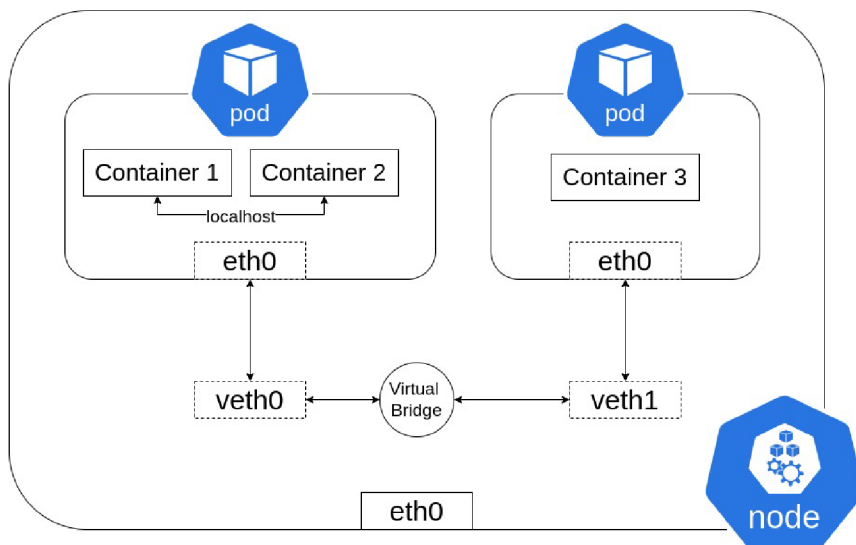
Figure 2.3: Container to Container and Pod to Pod networking. (created by the author)

Management

There are numerous built-in APIs provided by Kubernetes that are specifically designed to enable declarative management of workloads and their parent components. Applications run inside containers that are stored inside Pods. However, it would require a significant amount of work to manually manage each individual Pod. It would be better to instantiate a new Pod in its place, for example, in the event of Pod failure. These tasks are automated by Kubernetes, which reduces the load. Kubernetes control plane takes over the role of managing Pod entities, following the guidelines specified in the workload object that was created.

- **ReplicaSet.** The main goal of a ReplicaSet is to guarantee that a specified set of backup Pods is always present and functioning at all times. For this reason, it is frequently used to guarantee the constant availability of a predetermined number of identical Pods. Several defining fields define a ReplicaSet:

  1. a selector that specifies the criteria for identifying Pods that are available for acquisition;
  2. a replica count that indicates how many Pods the set should support;
  3. a Pod template that specifies the details of newly instantiated Pods that are required to satisfy the preset replica count.

  A ReplicaSet then goes about doing its job, creating and deleting Pods in order to reach the desired number. When conditions call for the creation of new Pods, the ReplicaSet uses the assigned Pod template.

- **Deployments.** Deployments make declarative changes to Pods and ReplicaSets easier. In a deployment, you specify the desired configuration, and the deployment controller modifies the existing state to conform to this specification while controlling the changeover at a controlled speed. ReplicaSets can be instantiated by Deployments, or they can be set up to replace other Deployments by moving their resources to new instances.

- **StatefulSets.** The StatefulSet is a key component of the workload API and is used as a supervisor for stateful applications. Its principal purpose is to coordinate and scale a specific group of Pods while providing guarantees for the orderly deployment and unique occurrence of these Pods. Similar to a deployment, a StatefulSet manages Pods based on the same container specification. However, it deviates from the deployment paradigm in that it gives each of its component Pods a persistent identity. Due to the fact that these Pods are based on a single specification, they are not interchangeable. Instead, they are all assigned a unique identifier that will not change even in the event of a rescheduling.

- **Jobs.** Job coordinates the formation and operation of one or more Pods, continuously attempting to run them until a predefined quantity of them successfully complete their missions. The Job keeps a close eye on the Pods' successful completions throughout this process. The Job is deemed fulfilled when the required number of successful completions is reached. The cleanup procedure for the related Pods that a Job deleted is triggered. Furthermore, when a Job is suspended, its active Pods are removed until the Job is reactivated. In a simple case, one Job instance is used to guarantee the consistent running of one single Pod until it is finished. If the first Pod fails or is removed (for example, due to a reboot or hardware failure of the node), the Job quickly starts a new Pod to carry out the task.

2.2.4 Security

A range of APIs and security features are included in Kubernetes, along with techniques for defining policies that support the management of information security procedures.

Following the Cloud Native Computing Foundation's[1] recommendations for best practices in information security in cloud-native environments, Kubernetes follows a cloud-native architectural framework.

Limiting access to the Kubernetes API is a crucial security feature for any Kubernetes cluster.

Secrets

A Secret is an object that contains a restricted amount of sensitive information, such as cryptographic keys, tokens, or passwords. Otherwise, a Pod's specification might incorporate such sensitive data, or a container image might contain it. Using Secrets eliminates the need to directly include sensitive information in application code. The process of creating Secrets on their own, apart from the Pods that depend on them, reduces the possibility that the Secret and any related information will be accidentally revealed. This process includes the Pod creation, inspection, and modification workflow.

Pod Security Standards

The three policies that make up the Pod Security Standards are designed to address security issues in a multifaceted manner. Taken as a whole, these policies range from extremely minimal to noticeably restrictive, depending on how permissive they are.

- Privileged: Having the widest range of permissions and an unrestricted policy. This policy permits the escalation of known privileges.

---

[1] https://www.cncf.io/

- Baseline: Serving as a placeholder for a minimally restrictive rule meant to prevent known privilege increases. Allows the minimal standard configuration of Pods.

- Restricted: Including a severely limited policy that conforms to industry best practices and modern Pod hardening principles.

Service Accounts

A Service Account is any type of non-human account that has a distinct identity inside a Kubernetes cluster. The credentials linked to a specific Service Account can be used by Application Pods, system parts, and entities inside and outside the cluster to assume the identity of that Service Account. This feature is used for a number of things, such as implementing identity-based security measures and authenticating API servers.

2.3 Red Hat OpenShift Container Platform

Red Hat OpenShift Container Platform is a Kubernetes environment designed to manage container-focused applications and their dependencies in a variety of computing environments, such as cloud, virtualized, on-premise and bare metal. With a focus on usability, stability, and component customization, it makes the deployment, configuration, and administration of containers easier. This platform runs on nodes, which are computing resources that are equipped with Red Hat Enterprise Linux CoreOS (RHCOS), a secure and lightweight operating system based on Red Hat Enterprise Linux (RHEL). The overview of the Red Hat OpenShift Container Platform is shown in Figure 2.4.

A node obtains a container runtime, such as CRI-O or Docker, upon booting and setup, enabling it to administer and carry out container workloads that are scheduled for it. These workloads are scheduled and node registration and workload details are managed by the Kubernetes agent, or kubelet. The networking, load balancing, and routing within the cluster are managed by Red Hat OpenShift Container Platform. It has cluster services for logging, managing upgrades, and keeping an eye on performance and health of the cluster.

The platform also includes features like OperatorHub and the container image registry, which provide community-developed software and certified Red Hat products to deliver a range of application services within the cluster. These services include database management, front-end development, user interface design, application runtimes, business automation, and developer tools for testing and developing container applications.

Within the cluster, application management can be handled manually by setting up container deployments from pre-existing images or by using specialized resources known as Operators. Additionally, users can use pre-built images and source code to create custom images that they can store locally in internal, private, or public registries. Moreover, the Multicluster Management layer allows for the distribution of workloads among several clusters and their deployment, configuration, compliance, and management all from a single console interface.

The following sections will build on the knowledge about Kubernetes that was acquired in the previous sections. They will discuss the features and important differences between Kubernetes technology and the Red Hat OpenShift Container Platform technology that are used within the created implementation.
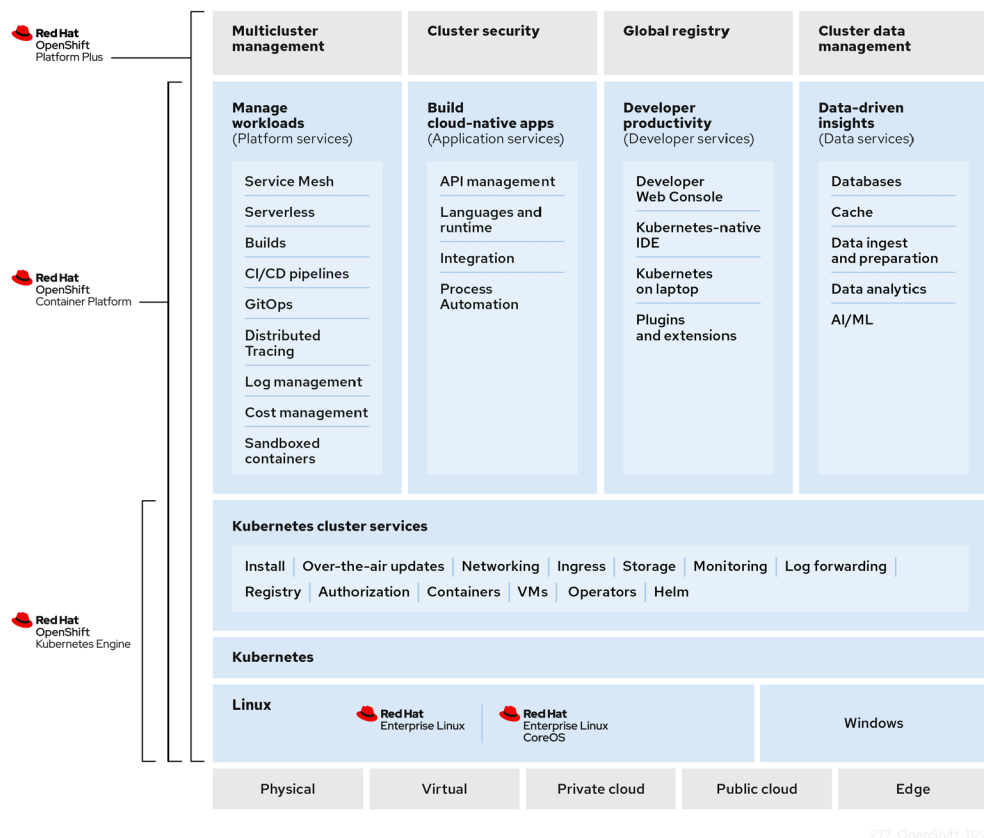
Figure 2.4: Red Hat OpenShift Container Platform overview [14].

2.3.1 Networking

Red Hat OpenShift Networking offers a wide range of features, plugins, and sophisticated networking functionalities that are intended to improve Kubernetes networking. Whether the cluster consists of one single entity or several hybrid clusters, this ecosystem is designed to meet its complex networking needs. Ingress, egress, load balancing, high-performance throughput, security protocols, and intra- and inter-cluster traffic management are just a few of the components that are seamlessly integrated within this framework. Role-based observability tools are also included to help simplify the inherent complexities.

Some of the key features commonly utilized within Red Hat OpenShift Networking are:

- Management of network plugins through the Cluster Network Operator.

- TLS encryption for web traffic via the Ingress Operator.

- Name assignment functionalities provided by the DNS Operator.

- Traffic load balancing on bare metal clusters enabled by the MetalLB Operator.

- Support for IP failover to ensure high availability.

- Expanded hardware network support through various Container Network Interface (CNI) plugins, including `macvlan`, `ipvlan`, and `SR-IOV`.

15

- Provision for IPv4, IPv6, and dual-stack addressing.

- Compatibility with hybrid Linux-Windows host clusters accommodating Windows-based workloads.

- Integration with Red Hat OpenShift Service Mesh for service discovery, load balancing, service-to-service authentication, failure recovery, and comprehensive metrics and monitoring.

Routes and Ingress

Pods and services that run within Red Hat OpenShift Container Platform cluster are given unique IP addresses during the cluster's setup. While other proximate Pods and services can reach these IP addresses, external clients cannot access them. The Ingress Operator runs the IngressController API, making services within the Red Hat OpenShift Container Platform cluster accessible from the outside. When one or more HAProxy-based Ingress Controllers are deployed and managed, the Ingress Operator handles routing, allowing external clients to communicate with your service. Traffic routing via Kubernetes Ingress and Red Hat OpenShift Container Platform Route can be configured by using the Ingress Operator. Furthermore, Ingress Controller configurations provide ways to publish Ingress Controller endpoints, such as specifying the type of `endpointPublishingStrategy` and controlling internal load balancing.

The Red Hat OpenShift Container Platform's Kubernetes Ingress feature is powered by an Ingress Controller that is enabled by a shared router service that runs as a Pod inside the cluster. This Ingress Controller, which can be scaled and replicated like any other standard Pod, is the main tool used to manage Ingress traffic. HAProxy is an open-source load balancing solution that is used by this router service.

The route mechanism in the Red Hat OpenShift Container Platform enables Ingress traffic to cluster services. Routes provide additional features, such as TLS re-encryption, TLS passthrough, and traffic splitting for blue-green deployments, that are not always possible with standard Kubernetes Ingress Controllers. Through routes, ingress traffic is able to access cluster services. While handling external requests and routing them according to predetermined criteria are features that both Ingress and routes have in common, Ingress is only capable of accepting three different connection types:

- HTTP/2,

- HTTPS with server name identification (SNI),

- and TLS with certificates.

Routes on the Red Hat OpenShift Container Platform are created dynamically in response to the Ingress resource's specifications.

Chapter 3

Vulnerability Detection

Cybersecurity field requires having a thorough understanding of vulnerabilities and implementing efficient detection techniques is essential to protecting digital environments from possible threats. The complexities of vulnerabilities are explored in this chapter, along with their nature, sources, and importance of early detection.

Before exploring the issues presented, it is essential to clarify the commonly used terms. The following definitions are based on [19]. This book discusses the topic of cybersecurity of information systems. However, the issues and definitions presented are largely similar to those mentioned in this thesis, so they will be incorporated in their translated paraphrased form as stated in the book.

Weak point. A weak point in an information system (IS) is susceptible to exploitation that causes harm or losses through an attack on the IS. The existence of weak points is a consequence of errors, failures in analysis, design, and/or implementation of the IS.

A weak point can also be the result of the high density of stored information, software complexity, the presence of hidden channels for information transmission, and other factors.

Types of a weak point:

- **Physical.** For example, the location of the IS physical infrastructure is in an area easily accessible to sabotage or vandalism, or vulnerable to power outages.

- **Natural.** Objective factors such as floods, fires, earthquakes, or lightning.

- **In Hardware or Software.** Vulnerabilities can exist in the physical components or the software systems.

- **Physical Attacks.** This includes radiations, attacks during communication for message exchange, or attacks on communication channels.

- **Human Factor.** Representing the greatest vulnerability among all possibilities.

Threat. Weak points are characteristics or elements of an information system that make it vulnerable to threats from the environment in which it functions. Threat is a potential for an attacker to target weak areas in an IS in order to damage its assets.

Threats are categorized as follows.

- Objective.

    - Natural, physical: fire, flood, power outage, malfunctions, etc.; An emergency plan must be created in situations like these, where prevention is challenging and attention must be paid to minimizing the effects through a suitable recovery plan.

    - Physical: e.g. electromagnetic radiation.

    - Technical or logical: memory failure, software *backdoors*, improper connection of otherwise secure components, theft, or destruction of storage media, or incorrect deletion of information.

- Subjective (human factor threats).

    - Unintentional: actions of untrained user or administrator.

    - Intentional: posed by the existence of external attackers (spies, terrorists, criminal elements, competitors, hackers) and internal attackers. It is estimated that 80% of IT attacks are conducted from within, by attackers who could be dismissed, disgruntled, blackmailed, or greedy employees. In terms of attack management, the cooperation of both types of attackers is very effective.

The essence of a threat lies in its origin, whether external or internal, and the motivations driving potential attackers, such as financial gain or gaining a competitive edge. In situations where prevention is difficult and it is necessary to focus on minimizing impacts through an appropriate recovery plan, an emergency plan must be developed. A threat is also characterized by its frequency and criticality of application. Common risks to information technology include data alteration without authorization, message interception and modification, illegal access through eavesdropping, and vulnerability exploitation through the use of harmful software or electromagnetic radiation. The gathering of private data, the unauthorized use of resources, the denial of service, and the abdication of accountability for actions pertaining to security are additional ways that threats might appear.

Attack. An attack, also known as a security incident, may consist of a deliberate use of a weak point, i.e., either the unintentional execution of an action that results in asset damage or the use of a vulnerable point to cause losses or damage to IS assets. The manifestation of computer crime, potential attack forms, attackers' identities, the risks associated with using information technologies, and attack defense strategies are typical issues to consider when analyzing potential forms of IT attacks.

Attacks can take the following forms:

- **Interruption.** An intentional attack on availability, including loss, unavailability, damage to assets, peripheral malfunction, data erasure, program deletion, and operating system issues.

- **Interception.** A passive attack on confidentiality, in which an unauthorized party gains unauthorized access to assets, such as a program or data copy.

- **Modification.** An active attack on integrity, in which an unauthorized party alters an asset, such as by changing stored and/or transmitted data or by adding functionality to a program.

- **Fabrication.** An active attack on authenticity or integrity, in which an unauthorized party creates something (transaction forgery, provision of false data).

Since detecting interception is very difficult, prevention is a good way to defend against passive interception attacks. Total attack prevention is unachievable, so typical defense (particularly against active attack forms) depends on identifying attacks and then restoring activity. Whether a heuristic is active, detection-based, preventive, or based on hypotheses, it is crucial to apply knowledge gained from observed facts and experiences to improve protections. An attack may occur accidentally, intentionally or randomly. Identified are the following types of attacks.

- Attacks on hardware:

  - Interruption: natural disasters, impacts, intentional attacks by theft, destruction.
  - Interception: theft of processor time or memory space.
  - Modification: changing operational modes.

- Attacks on software:

  - Interruption: unintentional attacks may include software deletion caused by incorrect configuration systems or archival systems, use of untested programs, operator errors; intentional attacks may include deliberate software deletion.
  - Interception: unauthorized copying of software, piracy.
  - Modification: using *backdoors* (non-public startup procedures from the time of software creation).
  - Fabrication: embedding trojans, viruses, worms, logic bombs.

- Attacks on data:

  - Attacks on data are much more dangerous because data can be read and interpreted by virtually anyone; the value of data is characterized by its temporality, and the market value of data is not the sole cost consideration, as it must also include the cost of data reconstruction, re-creation, etc.
  - Interruption: unintentional deletion, such as accidental data erasure, deliberate deletion, sabotage.
  - Interception: breach of confidentiality, theft of copies.
  - Modification: integrity violation, unauthorized data modifications.
  - Fabrication: repeated unauthorized partial withdrawals from a bank account (salami attack), transaction generation, etc.

Attacker. An attacker can be external, but often, there is also an internal attacker within the organization. Generalization based on their knowledge and resources:

- Weak strength attackers: amateurs, random attackers exploiting randomly discovered vulnerabilities during routine work; these are random, often unintentional attacks, where attackers have limited knowledge, opportunities, and means. Relatively weak security measures, which are inexpensive, are sufficient for protection against them.

- Moderate strength attackers: hackers whose frequent creed is to gain access to unauthorized areas; these are common attacks, attackers often have a lot of knowledge, but usually lack obvious opportunities for attacks and have limited means; moderate security measures are taken against them.

- Significant strength attackers: skilled criminals, typically with experience in the computer industry, who possess a high degree of expertise, sufficient financial resources, and sufficient time to execute an attack. They execute attacks that depart from standard procedure, and strong security measures are necessary to protect against them.

Risk. Risk, or the likelihood of taking advantage of weaknesses in an information system, is implied by the existence of threats. A threat is said to materialize with a certain degree of probability. Risks can be defined as potential harm inflicted by a security incident as well as the likelihood of one occurring.

3.1 Vulnerabilities

Weakness in an information system, system security procedures, internal controls, or implementation that could be exploited or triggered by a threat source [10].

Recognizing that vulnerabilities can manifest in various forms is crucial for developing effective mitigation strategies.

Vulnerabilities come in various forms and types, each representing a distinct facet of potential weaknesses within digital systems.

1. **Buffer Overflow.** This occurs when a program writes more data to a block of memory, or buffer, than it can hold. Exploiting this vulnerability allows attackers to overwrite adjacent memory, potentially leading to execution of malicious code.

2. **Injection Attacks.** Injection vulnerabilities, such as SQL injection and cross-site scripting (XSS), involve injecting malicious code into input fields or scripts. Successful exploitation can result in unauthorized access, data manipulation, or execution of arbitrary code.

3. **Cross-Site Scripting (XSS).** XSS vulnerabilities enable attackers to inject malicious scripts into web pages viewed by other users. This can lead to a theft of sensitive information, session hijacking, or website defacement.

4. **Cross-Site Request Forgery (CSRF).** CSRF exploits trust in a user's authenticated session to perform unauthorized actions on their behalf. Attackers trick users into unwittingly submitting requests, leading to actions such as changing passwords or making financial transactions.

5. **Security Misconfigurations.** Improperly configured settings, default passwords, or unnecessary services can expose vulnerabilities. Attackers exploit these misconfigurations to gain unauthorized access or disrupt services.

6. **Insecure Direct Object References (IDOR).** IDOR vulnerabilities occur when an application provides direct access to objects based on user-supplied input. Exploiting this allows attackers to access unauthorized data.

7. **Privilege Escalation.** Privilege escalation vulnerabilities enable attackers to elevate their permissions beyond what is intended. This could involve gaining administrative access or higher-level privileges within a system.

8. **Denial of Service (DoS) and Distributed Denial of Service (DDoS).** DoS vulnerabilities involve overwhelming a system or network with traffic, causing service interruptions. DDoS attacks amplify this by coordinating attacks from multiple sources, making mitigation more challenging.

9. **Zero-Day Vulnerabilities.** These are vulnerabilities that are unknown to the software vendor or the public. Attackers exploit these vulnerabilities before a patch or fix is available.

10. **Physical Security Vulnerabilities.** Physical vulnerabilities involve weaknesses in the physical security of systems, such as unauthorized access to hardware, theft, or tampering.

11. **Third-Party Software Vulnerabilities.** Dependencies on third-party libraries or software components can introduce vulnerabilities. Attackers may exploit weaknesses in these components to compromise the overall system.

12. **Social Engineering.** While not strictly a technical vulnerability, social engineering exploits human psychology to manipulate individuals to reveal sensitive information or perform actions that compromise security.

This extended list based on [4] is not exhaustive, and new vulnerabilities emerge as technology evolves. Addressing these vulnerabilities requires a multifaceted approach, combining robust coding practices, regular security assessments, and user education.

3.2 Container Image Vulnerabilities

A vulnerability [8], in the context of container images, refers to a weakness or flaw that could be exploited by malicious entities to compromise the confidentiality, integrity or availability of the deployed application.

Container image vulnerabilities [21] can take many different forms, including misconfigured software, insecure dependencies, and software flaws. To fully comprehend, one must acknowledge that vulnerabilities include more than just code-level flaws. They also include the configuration settings, runtime environments, and dependencies present in a containerized program.

Regarding container security, a vulnerability is any weakness, opening, or incorrect setup that an unauthorized party might be able to take advantage of to jeopardize the security of a container image or the containerized environment as a whole. It is crucial to realize that as technology develops, new threats emerge and vulnerabilities change constantly.

Information about the following example container vulnerabilities is sourced from the National Vulnerability Database, NIST. More details can be found on their official website[1].

- **CVE-2019-5736**: Exploit allows attackers to overwrite the host `runc` binary and gain root-level code execution on the host. This affects most containers that use the default container runtime.

---

[1]`https://nvd.nist.gov/`

- **CVE-2018-15664**: A vulnerability in Docker that can allow a path traversal attack, enabling an attacker to execute arbitrary commands inside the container and potentially escape to the host.

- **CVE-2020-14386**: A serious issue in the Linux kernel that could allow an attacker in a container to cause a heap-based buffer overflow, leading to privilege escalation and root access on the host.

- **CVE-2021-30465**: Vulnerability in Open Containers `runc` that allows a malicious container to escape to the host system due to improper handling of file descriptors.

- **CVE-2019-16884**: Vulnerability in `containerd` allows attackers to use a specially crafted image that, when pulled and unpacked, leads to a container escape and arbitrary command execution on the host.

3.2.1 Dynamic Nature of Container Vulnerabilities

Containers, by design, promote agility and rapid deployment, but this dynamism introduces unique challenges regarding security. The dynamic nature of containers means that the vulnerability landscape evolves rapidly. Continuous integration, continuous deployment (CI/CD) pipelines [1], and frequent updates contribute to a fluid environment where new vulnerabilities may arise with each iteration.

It is critical to stay on top of the constantly shifting threat landscape in this environment and to quickly patch any vulnerabilities. Understanding the dynamic nature of vulnerabilities is important for putting in place efficient security measures, especially as containerized applications become increasingly integrated into modern computing.

3.3 Container Image Security

The following section is derived from *Container image security: Going beyond vulnerability scanning* whitepaper by Red Hat [13].

In cloud-native environments, container images are the common application delivery format. A new set of best practices is needed to ensure the integrity of container images given their widespread distribution and deployment. Although running image scans to look for known vulnerabilities in language packages and operating systems is still a crucial component of image security, it is merely one of many security measures you must take to protect your environments. Decisions about image infrastructure and handling to improve and maintain your organization's security will be informed by an understanding of the risks at each stage of a container's life cycle.

3.3.1 Build process

A container image that has malicious software in it already presents a risk while running. In the context of continuous integration build infrastructure to stop outside vulnerabilities from getting into your images, security in continuous integration build infrastructure is just as crucial as it is in production environment. Steps to help improve securing build infrastructure and pipelines are as follows:

- Limit administrative access to the build infrastructure.

- Permit only necessary network ingress.

- Handle any required secrets with caution and only give the bare minimum of access.

- Make sure to thoroughly inspect any external websites from which sources or other files are retrieved and use network firewalls to restrict list access to reliable sources.

- Use a vulnerability scanner on the resulting images, but in order to obtain reliable results, it is also necessary to make sure that the scanner is secure and reliable.

Base image. In the process of constructing images based on pre-existing, third-party base images, several critical factors warrant consideration:

- Trustworthiness of the base image's provenance and hosting infrastructure. Verifying the legitimacy of the open-source community or source organization that is providing the base image is essential. It is crucial to confirm that the image is hosted on a reliable registry and comes from a respectable source. Transparency and security are further strengthened by making the Dockerfile and underlying source code of every component in the image accessible.

- Update frequency. Choosing base images that are updated frequently is essential, especially when relevant vulnerability disclosures occur. Images that are not updated regularly present increased risks and can expose systems to known vulnerabilities.

- Default software packages installed. A more stable and controllable deployment approach is produced by starting with a base image that is minimalistic and carefully adding the necessary tools and libraries based on the requirements of the application. By eliminating the necessity to determine which packages can be securely removed from an existing image, this method reduces the possibility of complications and dependencies.

Among the growing number of safe and simple base image choices, a few choices are worth taking into account. For example, Google Distroless[2] provides incredibly simple base images that are already set up for popular programming languages. Even though these images don't come with a package installer, you can add more software by copying files to them. Furthermore, the Red Hat Universal Base Image (UBI), which is based on Red Hat Enterprise Linux, offers a feasible alternative that can be accessed without requiring a Red Hat subscription. UBI supports a variety of language ecosystems and is offered in several tiers that include standard platform, multi-service, and minimalistic versions. A custom base image can also be created from scratch, providing more control over the composition and security posture of the image.

Securing Container Images. Reducing the number of potential attack vectors becomes crucial in the event that hostile actors are able to successfully compromise the security of an operating container. Starting with a simple base image establishes a strong basis, but this benefit is compromised by the use of a common set of flexible tools for all Dockerfiles. Keeping the image small enough to minimize its footprint has two advantages: it makes it less likely that zero-day vulnerabilities will be included inside the image, and it also makes the image smaller, which speeds up storage and retrieval.

---

[2]`https://github.com/GoogleContainerTools/distroless`

Using an approach that restricts the image to necessary binaries, libraries, and configuration files provides the best protection. In particular, installing (or uninstalling) the following tools should be avoided: package managers (like apt, yum, and apk), Unix shells (like bash and sh), compilers, and debuggers (removing them also prevents shell scripts from running in runtime).

If it becomes necessary to use these tools for application debugging in production images, workflows and practices should be reevaluated. It is wise to consider creating temporary debugging images in order to address diagnostic challenges, at the very least. Notably, *ephemeral containers* are given initial support by Kubernetes starting with version 1.16, which makes it easier to integrate them into already existing Pods and speed up debugging efforts.

Ephemeral containers are temporary and lightweight containers that can be added to a running Kubernetes pod, primarily used for debugging purposes or testing without affecting the Pod's operation. They are designed to execute and terminate without leaving any state or configuration behind.

Secrets. It is essential that private information is not inserted into images, even if they are intended for internal use. Key elements including SSH private keys, cloud provider credentials, Transport Layer Security (TLS) certificate keys, and database passwords are all included in this type of sensitive data. It is critical to acknowledge that any individual who has access to the image has the ability to extract these private elements. A safe strategy is to only supply sensitive data during runtime, which makes it easier to use the same image in different runtime environments, each requiring different credentials. This approach simplifies the update of compromised or expired secrets without requiring the image to be rebuilt. Providing Secrets to Kubernetes Pods as Kubernetes secrets or using a different secret management system are workable substitutes for embedding secrets inside images.

Image security scanning. Images that contain software that has security flaws can be exploited while the program is running. When building an image using the CI pipeline, image scanning becomes a necessary precondition to ensure build runs are completed successfully. Insecure images must never be allowed to register in the production-accessible container registry. It is important to remember that while there are many free and paid image scanning programs available, along with cloud-based scanning services, their coverage effectiveness varies. Although some scanners only check installed operating system packages, others also check installed runtime libraries related to particular programming languages.

Some scanners can enhance their capabilities by incorporating additional procedures, such as binary fingerprinting or other types of file content analysis.

Careful consideration should be given to the scanner's ability to meet the required coverage parameters, support the package installer database of the base image, and work with the programming languages used in the application ecosystem when choosing one to integrate into the continuous integration pipeline. It is the practitioner's responsibility to define appropriate risk thresholds, which may include defining vulnerability severity thresholds and creating a protocol for managing builds that have fixable vulnerabilities that are more severe than predefined thresholds.

3.3.2 Storing process

Once the secure container image is constructed, it becomes necessary to choose a location for its storage. Selecting a private, internal registry offers a significant advantage for strengthening security protocols and enabling personalized setups. This method requires close supervision of the infrastructure and access protocols of the registry. A lot of cloud service providers, on the other hand, offer managed registry services that are linked into the cloud's access control system. Private repositories can benefit from these service-based registries, which also significantly reduce administrative burdens. Thus, depending on their unique security requirements and available infrastructure resources, security engineers and build engineers must carefully determine which solution best meets their organization's needs.

3.4 Vulnerability Management

In order to lower the risks of cyberattacks and security breaches, vulnerability management is an IT security practice that entails finding, evaluating, and fixing security flaws in devices, networks, and applications. Vulnerability management is seen by security experts as a crucial component of security automation. This Section is derived from [16].

Common Vulnerabilities and Exposures (CVEs) is a system used by the security industry to catalog vulnerabilities found by IT vendors and security researchers. Vulnerability management is a continuous process due to the constant emergence of new CVEs. Security teams can automate vulnerability scanning and patching, among other detection and remediation procedures, with the use of a vulnerability management program.

Common Vulnerability Scoring System (CVSS) is the industry standard for grading CVEs. The vulnerability is assessed by means of a formula that considers various aspects, including the degree of complexity, the possibility of remote attack, and the need for user intervention. A base score, ranging from 0 (no impact) to 10 (highest base impact), is assigned by the CVSS to each CVE. This score by itself does not provide a thorough evaluation of risk. A more comprehensive CVSS analysis can be formed with the aid of two additional review types: temporal and environmental.

A temporal review includes information about the methods of exploitation that are currently in use, the presence of attacks that take advantage of the vulnerability, and the accessibility of fixes or workarounds for the flaw.

An environmental review contributes details unique to the organization concerning mission-critical information, systems, or controls that may be present in the environment of the end user and have the potential to change the likelihood or impact of an attack being carried out successfully. Researchers and vendors may also employ other scales in addition to CVSS scores. Red Hat Product Security, for instance, offers a four-point severity scale to assist users in assessing security risks.

- Critical impact: Vulnerabilities that could be quickly and easily used by an unauthenticated remote attacker to compromise the system without requiring user input.

- Important impact: Vulnerabilities that could compromise resource availability, confidentiality, or integrity.

- Moderate impact: Vulnerabilities that might be harder to take advantage of, but in some cases, could still compromise resource availability, confidentiality, or integrity.

- Low impact: Any other security-related vulnerabilities, such as those that are thought to require unusual circumstances to be exploited or those for which a successful exploit would have little repercussions.

3.4.1 Vulnerability management

Vulnerability management is a cyclical process [18]. The cycle is represented graphically in Figure 3.1.

Figure 3.1: Vulnerability management cycle [18].

The cycle goes through a certain number of steps before repeating. The steps are as follows:

- **Discover vulnerabilities.** A security breach is more likely to occur the longer a vulnerability goes undiscovered. Conduct weekly network scans, both internal and external, to find new and old vulnerabilities. This procedure includes scanning systems that are accessible over a network, determining which ports and services are open on those systems, obtaining system data, and comparing those data with known vulnerabilities.

- **Prioritize assets.** After learning which assets are being used, an assessment of the worth of each asset can be performed in relation to its use or alignment with the goals of the organization. Including this contextual understanding in the system list makes it easier to assess how urgently vulnerabilities need to be fixed. Determining, for example, whether the compromised system is related to a business laptop, a customer support desk terminal, an application, or a web server that provides essential services to a distinguished clientele helps prioritize security fixes appropriately.

- **Assess vulnerabilities.** Assessment is a systematic examination aimed at capturing the condition of applications and systems within a given environment.

- **Prioritize vulnerabilities.** Prioritizing vulnerabilities based on their possible impact on the company, employees, and customers is crucial as soon as vulnerabilities are found during the scanning process. Traditional vulnerability management systems provide a variety of integrated metrics that are intended to aid in the evaluation and prioritization of vulnerabilities. But it is also crucial to supplement this process with background knowledge about company operations, current threats, and related risks that come from both internal and external sources. Finding vulnerabilities that are highly relevant, have the potential to have a substantial impact, and are likely to be exploited is the main goal. It might not be possible to patch every vulnerability in the organizational ecosystem due to the multiplicity of devices, services, and software. A practical strategy for navigating this reality is to identify and rank the most important and likely targets of future attacks.

- **Remediate vulnerabilities.** The remediation and/or mitigation of these identified weaknesses is the natural progression that occurs after vulnerabilities are identified, prioritized, and cataloged. It is important to recognize that those responsible for understanding the risks associated with vulnerabilities may not often be in a position to implement corrective actions within an organization. Keeping this discrepancy in mind, companies should work to ensure that their security operations, IT operations, and system administration teams have a common vocabulary, set of standards for making decisions, and set of procedures.

- **Verify remediation.** Confirming the vulnerability's resolution is a critical but frequently overlooked step. After carrying out the actions described above, it is wise to carry out a follow-up scan with the objective of verifying that the risks that are of the highest priority have been successfully resolved or mitigated. This final procedural step allows the incident to be formally closed within the tracking system and makes it easier to evaluate key performance indicators like the mean time to remediate (MTTR) or the number of critical vulnerabilities that are still present.

- **Report on status.** Especially after major events like a widely publicized software bug or the use of a zero-day vulnerability, questions from management, executives, and board members might come up about how well vulnerability assessment and mitigation efforts are working inside company infrastructure. Reports that provide information about vulnerabilities, related risks, and the effectiveness of vulnerability management practices can be used to support requests for staffing or purchase of relevant tools. Leading vulnerability management platforms, for example, provide features like interactive dashboards that are customized to fit different user profiles, stakeholder interests, and analytical viewpoints, as well as automated reports.

Chapter 4

Software Bill of Materials and Advanced Scanning Techniques

The adoption of *Software Bill of Materials* (SBOM) has become a pivotal practice in enhancing transparency and security in software supply chains.

An SBOM [11] functions as a detailed inventory, listing the ingredients, software components, along with their identification, information, and supply chain relationships. The richness of information in an SBOM varies based on industry needs and consumer requirements.

When creating an SBOM, the focus is on establishing a baseline with the minimum necessary information, allowing for swift adoption and subsequent evolution. This approach avoids imposing an exhaustive set of attributes initially, ensuring a practical starting point.

SBOMs are dynamic and relate each component through intricate supply chains, offering a comprehensive view of software systems. The capture and exchange of these links are some of the crucial features.

Structured data formats and exchange protocols are essential for a functional SBOM, enabling machine readability and automation, especially for large organizations managing diverse data sources.

Importantly, SBOMs are not isolated; they connect with other data sources. For instance, in vulnerability management, SBOMs rely on a catalog of known vulnerabilities (e.g., CVE) and associations of vulnerabilities to components (e.g., Vultron project will use default libraries from Grype). Similarly, for license management, mapping licenses to components is essential for compliance.

In essence, SBOMs serve as dynamic and interconnected tools, delineating software compositions and facilitating efficient data management, scalability, and alignment with crucial data sources in the software ecosystem.

Crucial role of SBOM as a catalyst for achieving transparency in software supply chains. SBOM promises to mitigate cybersecurity risks and streamline costs by [11]:

- **Identifying Vulnerabilities.** Enhancing the recognition of vulnerable software components to reduce cybersecurity incidents.

- **Streamlining Supply Chains.** Reducing unproductive work from convoluted supply chains and fostering efficient development and maintenance processes.

- **Market Differentiation.** Allowing SBOM supporting vendors to distinguish themselves, fostering trust and accountability.

- **Standardization Efforts.** Reducing duplication through the promotion of standardized formats, enhancing unity across different sectors.

- **Counterfeit Detection.** Facilitating the identification of suspicious or counterfeit software components, fortifying the integrity of the entire system.

4.1   Key Components of an SBOM

The cornerstone of an effective SBOM lies in its ability to uniquely and unambiguously identify components and establish their relationships. The following baseline information serves as the foundational elements for an SBOM entry, providing essential details for comprehensive identification:

- **Author Name.** The author of the SBOM entry, not necessarily the supplier. This attribution ensures clarity regarding the source of the SBOM information.

- **Supplier Name.** This signifies the name or identity of the component supplier in the SBOM entry. In cases where the author and supplier are the same, it designates a first-party authoritative component. When different, it indicates a claim or assertion about a component from an alternative supplier.

- **Component Name.** One or more names of the component are specified, allowing flexibility for multiple names or aliases. Component names can include supplier information and may be expressed using a generic `namespace:name` construct.

- **Version String.** Version information is crucial for component identification. Syntax specifics are not prescribed, but basic consistency and logic, such as *Semantic Versioning*, are expected.

- **Component Hash.** The cryptographic hash of the component serves as a precise identifier. While highly accurate, other baseline identification information is considered useful and sometimes necessary.

- **Unique Identifier.** A version 4 or 5 UUID can be generated and utilized as a unique identifier for components, enhancing the identification process.

- **Relationship.** Inherent in the SBOM design, relationships are established. The default relationship type is *includes*, but for clarity, this document advocates for inverting the direction of the relationship to *included in*. This inversion ensures a consistent representation.

4.2   Syft

Syft[1] is a powerful open-source tool dedicated to enhancing transparency in container security. Specializing in generating Software Bill of Materials (SBOMs) for container images, Syft provides a detailed inventory of software components, dependencies, and their respective versions. By fostering a clear understanding of the containerized application's composition, Syft significantly contributes to improving security practices in the dynamic landscape of containerized environments.

---

[1]`https://github.com/anchore/syft`

4.2.1   Key Features and Capabilities

Syft offers a host of features and capabilities that distinguish it in the realm of container security:

- **SBOM Generation.** Syft's primary function is generating comprehensive SBOMs, providing detailed insights into the components, dependencies, and versions within a container image.

- **Support for Multiple Image Formats.** Syft is versatile in supporting various container image formats, ensuring compatibility with a wide range of containerized applications.

- **Integration with CI/CD Pipelines.** Syft can be seamlessly integrated into CI/CD pipelines, enabling automated SBOM generation as part of the software development lifecycle.

- **Detection of Licenses and Vulnerabilities.** Beyond generating SBOMs, Syft helps to identify software licenses associated with components and detecting potential vulnerabilities, contributing to a holistic security assessment.

4.2.2   Syft SBOM Output

Syft generates a comprehensive Software Bill of Materials output that provides detailed information about the components and their relationships within a container image. The output may be structured as a JSON document with key fields, each serving a specific purpose. All example outputs will be from analysis of the `tomcat`[2] image.

Artifacts and Relationships

The SBOM output includes a detailed list of artifacts and their relationships, providing insights into the container's composition. Each artifact entry includes essential information, such as Common Platform Enumeration (CPE) details, identification methods, and relationships with other components within the image. Fields in `artifacts` section include:

- **cpes** – *Common Platform Enumeration* entries representing the component.

- **foundBy** – the tool or method that identified the component.

- **id** – a unique identifier of the component.

- **language** – the programming language of the component.

- **licenses** – information about the licenses associated with the component.

- **locations** – paths and layer information where the component is located.

- **metadata** – additional metadata about the component, such as author, version, and platform.

- **metadataType** – the type of metadata associated with the component (e.g., *python-package*).

---

[2]`https://hub.docker.com/_/tomcat/`

- **name** – the name of the component (e.g., *Jinja2*).

- **purl (Package URL)** – a standardized format for expressing metadata about software packages.

- **type** – the type of component (e.g., *python*).

- **version** – the version of the component (e.g., *2.11.3*).

An example of *artifacts* specification listing from the `tomcat` image is shown in Listing 4.1

```
1
2   "artifacts": [
3      {
4        "cpes": [
5          "cpe:2.3:a:apache:tomcat-annotations-api:2.1.1:*:*:*:*:*:*:*",
6        ],
7        "foundBy": "java-archive-cataloger",
8        "id": "b3436ecc39b02f05",
9        "language": "java",
10       "licenses": [
11         <snip>
12       ],
13       "locations": [
14         {
15           "accessPath": "/usr/local/tomcat/lib/annotations-api.jar",
16           "annotations": {
17             "evidence": "primary"
18           },
19           "layerID": "sha256:6b31b525a<snip>",
20           "path": "/usr/local/tomcat/lib/annotations-api.jar"
21         }
22       ],
23       "metadata": {
24         "digest": [
25           {
26             "algorithm": "sha1",
27             "value": "4d9f537d2a349621b949485c54aa1e550febe273"
28           }
29         ],
30         "manifest": {
31             <snip>
32         },
33         "virtualPath": "/usr/local/tomcat/lib/annotations-api.jar"
34       },
35       "metadataType": "java-archive",
36       "name": "annotations-api",
37       "purl": "pkg:maven/org.apache.tomcat-annotations-api/annotations-
               api@2.1.1",
38       "type": "java-archive",
```

```
39        "version": "2.1.1"
40      }
41    ]
```

Listing 4.1: An *artifacts* example from the `tomcat` package.

The `artifactRelationships` section specifies relationships between artifacts, such as containment and evidence associations.

Fields in this section include:

- **child** – the unique identifier of the artifact considered as the child in the relationship. This points to a specific component within the container image.

- **parent** – the unique identifier of the artifact considered as the parent in the relationship. This points to another component or the container image itself.

- **type** – describes the nature of the relationship between the child and parent artifacts. Common relationship types include:

    1. **contains** – indicates that the parent artifact contains the child artifact.

    2. **evident-by** – highlights a relationship where evidence supporting a claim is provided by the child artifact.

Example of *artifactRelationships* from the same `tomcat` image is shown in Listing 4.2.

```
1  "artifactRelationships": [
2    {
3      "child": "263a32391686f3c9",
4      "parent": "00085d3c60fed8fd",
5      "type": "evident-by"
6    },
7    {
8      "child": "43efc68c767736ea",
9      "parent": "051c283e855f0daf",
10      "type": "contains"
11    }
12  ],
```

Listing 4.2: An *artifactRelationships* example from the `tomcat` package.

Descriptor, Distro, and Schema Information

The SBOM output encapsulates key details about the container, including its descriptor, underlying distribution, and schema information. This metadata aids in understanding the context and environment in which the containerized application operates.

- **descriptor** – contains the name and version of the tool generating the SBOM (e.g., *syft 0.96.0*).

- **distro** – information about the underlying distribution, including name, version and relevant URLs.

- **schema** – details about the schema version and URL used in the SBOM.

Descriptor, *distro*, *files* and *schema* examples from the same `tomcat` image are shown in Listing 4.3.

```
 1  "descriptor": {
 2      "name": "syft",
 3      "version": "0.96.0"
 4    },
 5  "distro": {
 6      "bugReportURL": "https://bugs.launchpad.net/ubuntu/",
 7      "homeURL": "https://www.ubuntu.com/",
 8      "id": "ubuntu",
 9      "idLike": [
10        "debian"
11      ],
12      "name": "Ubuntu",
13      "prettyName": "Ubuntu 22.04.3 LTS",
14      "privacyPolicyURL": "https://www.ubuntu.com/legal/
15                          terms-and-policies/privacy-policy",
16      "supportURL": "https://help.ubuntu.com/",
17      "version": "22.04.3 LTS (Jammy Jellyfish)",
18      "versionCodename": "jammy",
19      "versionID": "22.04"
20    },
21  "files": [
22      {
23        "id": "a582cbda17469f59",
24        "location": {
25          "layerID": "sha256:8ceb9643fb36a<snip>",
26          "path": "/etc/alternatives/README"
27        }
28      },
29      {
30        "id": "a23bbaa4efaee7e7",
31        "location": {
32          "layerID": "sha256:8ceb9643fb36a<snip>",
33          "path": "/etc/apt/apt.conf.d/01-vendor-ubuntu"
34        }
35      }
36    ],
37  "schema": {
38      "url": "https://raw.githubusercontent.com/
39              anchore/syft/main/schema/json/schema-12.0.1.json",
40      "version": "12.0.1"
41    },
```

Listing 4.3: *Descriptor*, *distro*, *files* and *schema* examples from the `tomcat` package.

Source Information

The SBOM output delves into details about the source of the container image, including its identifier, metadata, layers, and user-provided information. This section aids in understanding the origin, configuration, and context of the container image.

- **id** – a unique identifier of the container image.

- **metadata** – additional metadata about the container image, including architecture, labels, and URLs.

- **name** – the name of the container image.

- **type** – the type of the source (e.g., *image*).

- **version** – the version or digest of the container image.

Source example from the same `tomcat` image is shown in Listing 4.4.

```
1   "source": {
2       "id": "5c98b22b571d494ea1907b5c<snip>",
3       "metadata": {
4         "architecture": "amd64",
5         "config": "eyJhcmNoaXRl<snip>",
6         "imageID": "sha256:e7652758<snip>",
7         "imageSize": 453582300,
8         "labels": {
9           "org.opencontainers.image.ref.name": "ubuntu",
10          "org.opencontainers.image.version": "22.04"
11        },
12        "layers": [
13          {
14            "digest": "sha256:8ceb9643<snip>",
15            "mediaType": "application/vnd.docker.image.rootfs.
16                         diff.tar.gzip",
17            "size": 77845857
18          }
19        ],
20        "manifest": "ewogICAic<snip>",
21        "manifestDigest": "sha256:5c98b22b<snip>",
22        "mediaType": "application/vnd.docker.distribution.
23                     manifest.v2+json",
24        "os": "linux",
25        "repoDigests": [
26          "index.docker.io/library/tomcat@sha256:fe38c5d0<snip>"
27        ],
28        "tags": [],
29        "userInput": "tomcat"
30      },
31      "name": "tomcat",
32      "type": "image",
```

```
33      "version": "sha256:5c98b22b<snip>f"
34    }
```

Listing 4.4: A *source* example from the `tomcat` package.

4.3 Grype

The open source vulnerability scanner Grype[3] is specifically made for container images. By utilizing the vulnerability database made available by the vulnerability-as-code project as its foundation, it enables users to perform comprehensive scans of containerized applications. Grype guarantees a thorough approach to security assessment by concentrating on finding vulnerabilities in software dependencies, configurations, and other crucial elements.

4.3.1 Key Features and Capabilities

Grype excels in providing a range of features that contribute to its effectiveness in container image scanning:

- **Comprehensive Database Integration.** Grype's strength lies in its integration with well-established vulnerability databases, ensuring up-to-date and accurate vulnerability information.

- **Flexible and Extensible.** The tool supports multiple image formats and allows for extensibility, enabling users to tailor scans to their specific needs and adapt to diverse container environments.

- **Integration with CI/CD Pipelines.** Grype seamlessly integrates into CI/CD pipelines, enabling automated and continuous vulnerability assessments throughout the software development lifecycle.

- **Rich Output Formats.** Grype offers versatile output formats, including JSON and SPDX, facilitating easy integration with other tools and platforms.

4.3.2 Grype output

All example outputs will be from an analysis of the `tomcat`[4] image.

The output of Grype provides detailed information about vulnerabilities and related artifacts. Outline of the sections present in the JSON output of Grype `tomcat` image analysis consists of parts described in the following sections.

Vulnerability Information

- **id** – CVE identifier for the vulnerability.

- **dataSource** – source of vulnerability information.

- **namespace** – namespace providing additional information about the vulnerability.

- **severity** – severity level of the vulnerability.

---

[3]`https://github.com/anchore/grype`
[4]`https://hub.docker.com/_/tomcat/`

35

- **urls** – URLs providing additional details or references related to the vulnerability.

- **cvss** – Common Vulnerability Scoring System metrics if available.

- **fix** – information about fixes for the vulnerability.

- **advisories** – advisories related to the vulnerability.

A *vulnerability* example from the `tomcat` image is shown in Listing 4.5.

```
1  "vulnerability": {
2      "id": "CVE-2022-3715",
3      "dataSource": "http://people.ubuntu.com/~ubuntu-security/
4                      cve/CVE-2022-3715",
5      "namespace": "ubuntu:distro:ubuntu:22.04",
6      "severity": "Low",
7      "urls": [
8          "http://people.ubuntu.com/~ubuntu-security/cve/CVE-2022-3715"
9      ],
10     "cvss": [],
11     "fix": {
12         "versions": [],
13         "state": "not-fixed"
14     },
15     "advisories": []
16 }
```

Listing 4.5: A *vulnerability* example from the `tomcat` image.

Related Vulnerabilities

Information about vulnerabilities related to the main vulnerability. For each related vulnerability, the fields are the same as for the main vulnerability. A *relatedVulnerabilities* example from the same `tomcat` image is shown in Listing 4.6.

```
1  "relatedVulnerabilities": [
2  {
3      "id": "CVE-2022-3715",
4      "dataSource": "https://nvd.nist.gov/vuln/detail/CVE-2022-3715",
5      "namespace": "nvd:cpe",
6      "severity": "High",
7      "urls": [
8          "https://bugzilla.redhat.com/show_bug.cgi?id=2126720"
9      ],
10     "description": "A flaw was found in the bash package, where
11                     a heap-buffer overflow can occur in valid
12                     parameter_transform. This issue may lead
13                     to memory problems.",
14     "cvss": [
15         {
```

```
16              "source": "nvd@nist.gov",
17              "type": "Primary",
18              "version": "3.1",
19              "vector": "CVSS:3.1/AV:L/AC:L/PR:L/UI:N/S:U/C:H/I:H/A:H",
20              "metrics": {
21                  "baseScore": 7.8,
22                  "exploitabilityScore": 1.8,
23                  "impactScore": 5.9
24              },
25              "vendorMetadata": {}
26          }
27      ]
28  }]
```

Listing 4.6: A *relatedVulnerabilities* example from the `tomcat` package.

Match Details

Information about how the vulnerability was matched in the system.

- **type** – type of match (e.g., *exact-direct-match*).

- **matcher** – matcher used for the match.

- **searchedBy** – information about the search criteria.

- **found** – information about the artifact found.

A *matchDetails* example from the same `tomcat` image is shown in Listing 4.7.

```
1  "matchDetails": [
2      {
3          "type": "exact-direct-match",
4          "matcher": "dpkg-matcher",
5          "searchedBy": {
6              "distro": {
7                  "type": "ubuntu",
8                  "version": "22.04"
9              },
10             "namespace": "ubuntu:distro:ubuntu:22.04",
11             "package": {
12                 "name": "bash",
13                 "version": "5.1-6ubuntu1"
14             }
15         },
16         "found": {
17             "versionConstraint": "none (deb)",
18             "vulnerabilityID": "CVE-2022-3715"
19         }
20     }
21  ]
```

Listing 4.7: A *matchDetails* example from the `tomcat` package.

Artifact Information

Information about the software artifact affected by the vulnerability.

- **id** – unique identifier of the artifact.

- **name** – name of the software (e.g., *bash*, *binutils*).

- **version** – version of the software.

- **type** – type of the artifact (e.g., *deb*).

- **locations** – locations of the artifact in the system.

- **language** – programming language associated with the artifact.

- **licenses** – licenses associated with the artifact.

- **cpes** – *Common Platform Enumeration* (CPE) entries associated with the artifact.

- **purl** – package URL providing a unique identifier for the artifact.

- **upstreams** – information about upstream sources for the artifact.

An *artifact* example from the same `tomcat` image is shown in Listing 4.8.

```
1  "artifact": {
2      "id": "66e585599046d91e",
3      "name": "bash",
4      "version": "5.1-6ubuntu1",
5      "type": "deb",
6      "locations": [
7          {
8              "path": "/usr/share/doc/bash/copyright",
9              "layerID": "sha256:8ceb9643fb36a8ac<snip>"
10         },
11         {
12             "path": "/var/lib/dpkg/info/bash.conffiles",
13             "layerID": "sha256:8ceb9643fb36a8ac<snip>"
14         },
15         <snip>
16     ],
17     "language": "",
18     "licenses": [
19         "GPL-3"
20     ],
21     "cpes": [
22         "cpe:2.3:a:bash:bash:5.1-6ubuntu1:*:*:*:*:*:*:*"
23     ],
```

```
24      "purl": "pkg:deb/ubuntu/bash@5.1-6ubuntu1?arch=
25             amd64\u0026distro=ubuntu-22.04",
26      "upstreams": []
27 }
```

Listing 4.8: An *artifact* example from the `tomcat` package.

Understanding the output structures of image scanning tools is essential when designing and implementing the Vulntron tool. This knowledge enables the creation of more robust and efficient security solutions, as it allows for the seamless integration of these tools into the Vulntron tool framework, thus enhancing its capabilities to accurately identify and mitigate vulnerabilities in containerized environments.

Chapter 5

Existing Tools for Vulnerability Detection

This chapter looks in-depth at a number of well-known solutions that are essential for locating, evaluating, and dealing with possible security risks in software systems. The significance of these tools in protecting digital environments against diverse cyber threats is emphasized by examining their workings and effectiveness. These tools provide an essential layer of defense that improves overall cybersecurity resilience by helping to identify and mitigate vulnerabilities.

5.1 Clair

Clair[1] is an open-source project that offers a tool designed to monitor container security by performing static analyzes of vulnerabilities within applications and Docker containers.

Using API-driven analysis engines, Clair meticulously examines container layers, identifying and assessing known security flaws. This approach enables the creation of services that facilitate the continuous monitoring of potential vulnerabilities in containerized environments.

Clair streamlines the process of building robust security measures, ensuring a vigilant stance against emerging threats throughout the life cycle of containerized applications. This Section is derived from [12].

5.1.1 How Clair works

Clair operates by meticulously scanning each layer of a container, identifying potential vulnerabilities that pose a threat. Leveraging databases such as the Common Vulnerabilities and Exposures (CVE), as well as repositories from Red Hat, Ubuntu, and Debian, Clair provides notifications based on known security flaws.

Given that container layers may be shared across multiple containers, a crucial aspect of Clair's functionality involves introspection to establish an inventory of packages and match them against known CVEs.

Automatic detection of vulnerabilities not only enhances awareness but also encourages the adoption of best security practices within development and operations teams. Clair's real-time awareness of newly announced vulnerabilities, without the need for rescanning,

---

[1]https://github.com/quay/clair

enables swift notification about existing vulnerable layers. Despite its efficiency, Clair does not dive into nuanced threat analysis, leaving teams responsible for more in-depth assessments when specific conditions for vulnerability exploitation are a concern.

5.1.2 Clair and Kubernetes

As an integral component of the open-source *Project Quay*, Clair seamlessly integrates with Kubernetes, particularly within the Red Hat OpenShift platform. The Container Security Operator, a Kubernetes Operator, facilitates Clair's deployment and utilization within Red Hat Quay.

5.1.3 Red Hat Quay

Red Hat Quay[2][15] is an enterprise-level container registry service designed to support cloud-native and DevSecOps development models. It enables the efficient construction, distribution, and secure deployment of containerized applications in global data center and cloud environments. Red Hat Quay provides robust security features and scalability, demanded by modern software development needs.

When integrated with Red Hat OpenShift, the Quay Container Security Operator enhances the security of container image repositories through comprehensive automation, sophisticated authentication, and robust authorization systems. This integrates seamless DevSecOps workflows by enabling automatic security scanning of container images, helping to identify and mitigate potential security vulnerabilities before they are deployed.

Red Hat Quay is versatile, available as a component of Red Hat OpenShift and as a standalone option. It provides advanced features that include georeplication to support distributed environments, continuous security scanning of container images, and extensive support for rollback capabilities. These features ensure that images are not only distributed efficiently across multiple locations, but also maintained with high integrity, allowing organizations to revert to previous versions of container images if needed.

A key component of Quay's security framework is Clair described in Section 5.1.

Red Hat Quay can be utilized as a hosted service or as a user-installable software, providing flexibility according to organizational needs. In either deployment model, Red Hat Quay, in combination with Clair, ensures a secure, efficient, and highly scalable environment for containerized applications. This makes it an indispensable tool for enterprises aiming to leverage the power of containerization while maintaining stringent security standards.

5.2 Trivy

Trivy[4] is an essential security tool, focusing on securing software deployment. As more companies embrace containers like Docker and Kubernetes, Trivy is an open-source vulnerability scanner that plays a key role in detecting and fixing security risks early on.

By easily integrating into development pipelines, it assists developers in detecting and addressing vulnerabilities, ensuring that modern software remains robust and secure. Its versatility is demonstrated by its ability to scan container images, filesystems, both local and remote Git repositories, virtual machine images, Kubernetes configurations, and AWS

---

[2]`https://www.redhat.com/en/technologies/cloud-computing/quay`
[3]`https://redhatgov.io/workshops/secure_software_factory/lab17/`
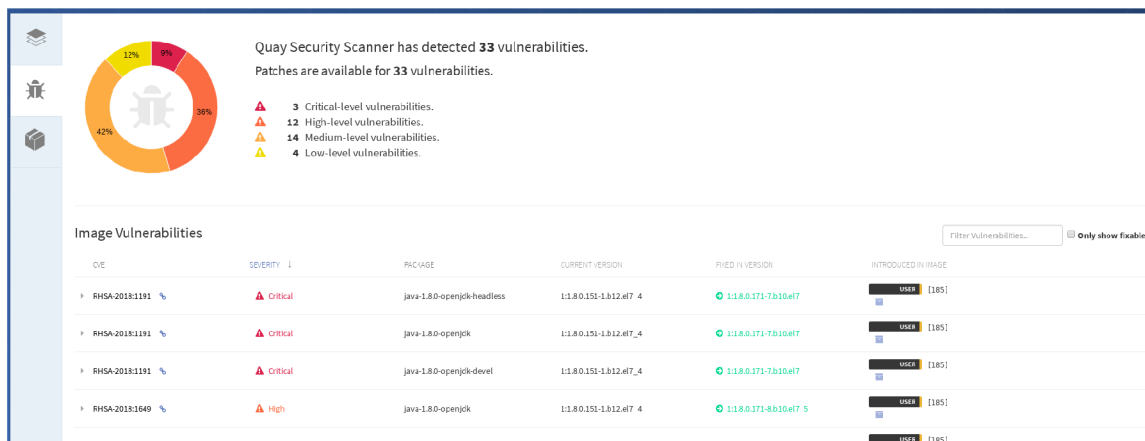[4]`https://trivy.dev`

Figure 5.1: Clair security scan as a part of the Red Hat Quay[3].

setups. This wide-ranging support underscores Trivy's relevance in diverse development and deployment scenarios.

When it comes to scanning, Trivy goes beyond a singular focus. It adeptly identifies OS packages, software dependencies, and Software Bill of Materials, providing a transparent view of the software stack. Its meticulous search extends to known vulnerabilities, including Common Vulnerabilities and Exposures, allowing for proactive risk mitigation.

Trivy doesn't stop at surface-level scans; it delves deeper into Infrastructure as Code (IaC) issues and misconfigurations, ensuring the integrity of deployment environments. Additionally, it proves invaluable in locating sensitive information and secrets, thus fortifying applications against potential security breaches.

Finally, Trivy extends its scope to software licenses, enabling organizations to maintain compliance and legal standards within their software ecosystems. An example Trivy security scan as a part of Gitlab pipeline is in Figure 5.2.

5.3 Nessus

Nessus[6], renowned for its network vulnerability scanning capabilities, extends its functionality to containers, providing a robust means of identifying and mitigating security risks within containerized environments.

When utilized as a container scanner, Nessus follows a systematic process to analyze and evaluate the security posture of container images. The workflow encompasses the following key steps [20]:

1. **Image Ingestion.** Ingesting container images from designated repositories or other sources, including container registries, local repositories, or specified network locations.

2. **Static Analysis.** Conducts a static analysis of the container image, examining its file system, configuration files, and metadata to identify potential vulnerabilities or misconfigurations.

---

[5]https://aquasecurity.github.io/trivy/v0.48/tutorials/integrations/gitlab-ci/
[6]https://www.tenable.com/products/nessus
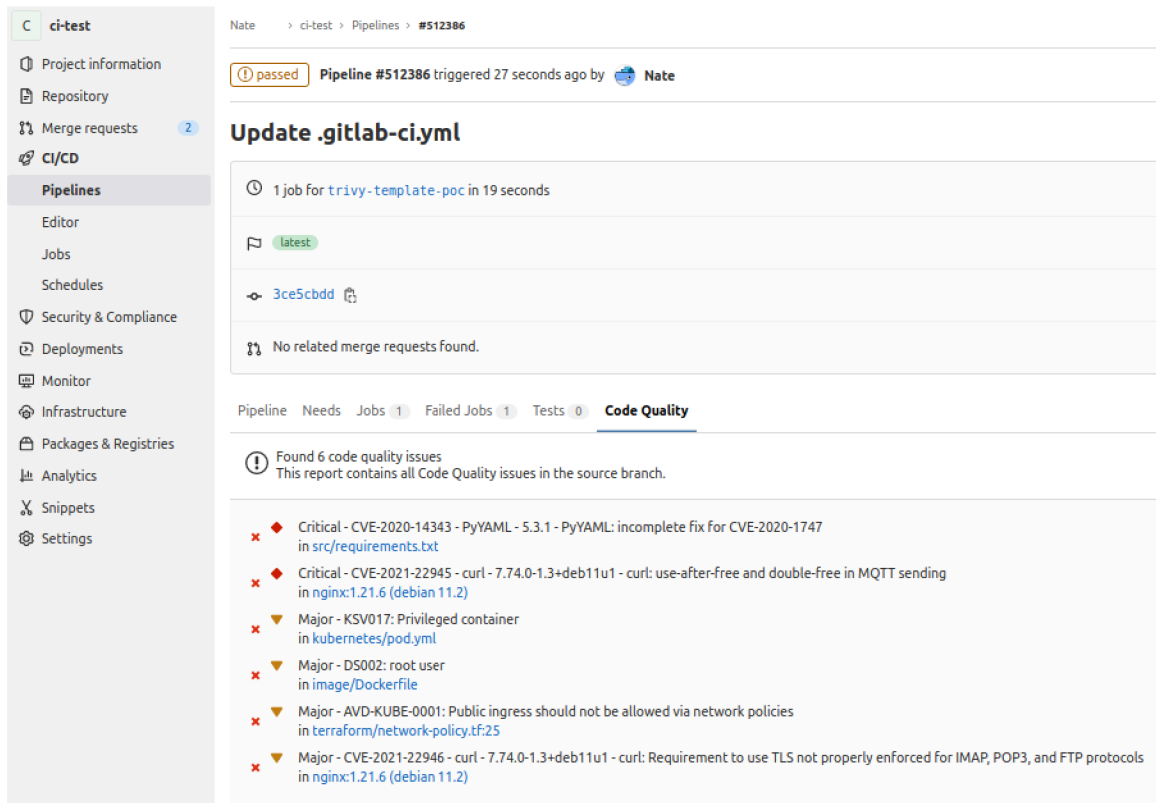
Figure 5.2: Trivy security scan as a part of Gitlab pipeline[5].

3. **Dependency Scanning.** Examine the image dependencies, such as operating system packages and software libraries, cross-referencing this information with known vulnerability databases to identify outdated or insecure components.

4. **Vulnerability Detection.** Utilizes a comprehensive vulnerability database to detect known vulnerabilities within the container image, including those with assigned Common Vulnerabilities and Exposures identifiers.

5. **Compliance Checks.** Verifies whether the container image adheres to predefined security and compliance standards, considering organization-specific policies, industry-specific regulations or best practices for secure container configurations.

6. **Configuration Assessment.** Assesses configuration settings within the container image, identifying deviations from security best practices, including settings related to network configurations, user privileges, and access controls.

7. **Report Generation.** After completing the analysis, generates a comprehensive report detailing identified vulnerabilities, misconfigurations, and compliance issues, serving as a valuable resource for security teams and developers.

8. **Integration with Workflows.** Integrates with existing CI/CD pipelines and orchestration tools, facilitating the automation of container security assessments and ensuring that security checks are integral to development and deployment processes.
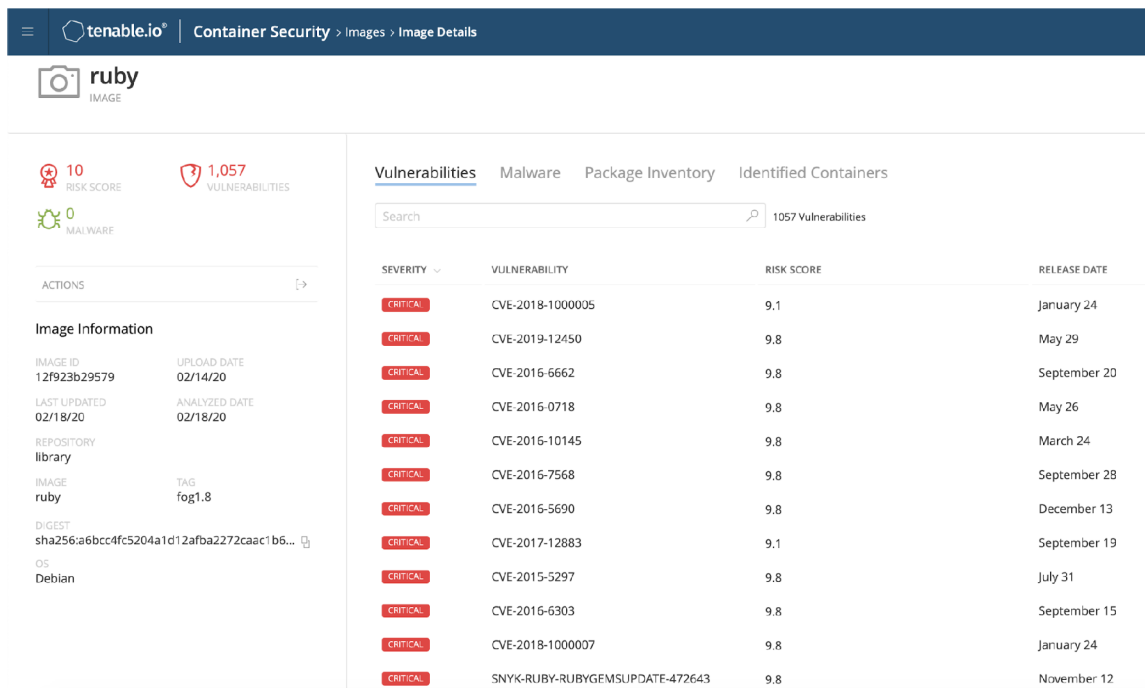
Figure 5.3: An example of a security scan by Nessus[7].

By following this methodical approach, Nessus, as a container scanner, empowers organizations to proactively identify and remediate security concerns within their containerized environments, contributing to a more resilient and secure application ecosystem. An example of a security scan done by Nessus is in Figure 5.3.

5.4   Jfrog Xray

JFrog Xray [5] was unveiled at swampUP on May 23, 2016. With this tool, organizations can now see into the contents of their software components, which is a major step toward improving the efficacy of DevOps, InfoSec and development teams, as well as optimizing the continuous delivery (CD) pipeline.

With the introduction of universal impact analysis by JFrog Xray, companies can now fully comprehend all of their software packages, binary artifacts, and container images, regardless of the large number and variety of components used in the software build and distribution process. Strong features like implications for production and continuous integration environments, a comprehensive dependencies graph that makes it simple to identify vulnerabilities or compliance issues, and an open API that supports integration with different component scanning technologies for customized scanning capabilities are just a few of the insightful information that JFrog Xray offers its users.

JFrog Xray provides radical transparency into every component used within an organization by leveraging integration with vulnerability and license compliance databases like

---

[7]https://www.tenable.com/blog/tenable-bolsters-container-security-to-capture-open-source-vulnerabilities
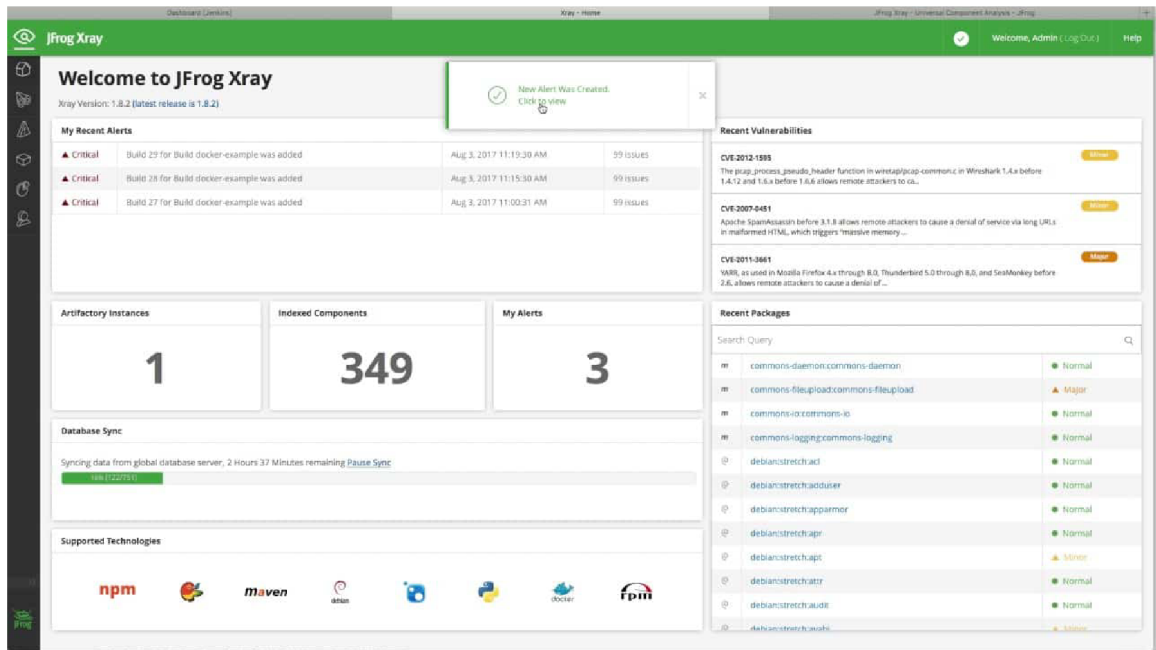
Figure 5.4: An example of a security scan by Jrog Xray[11].

VersionEye[8], Black Duck[9], and Mend[10]. Due to its smooth integration with JFrog Artifactory, organizations can perform a thorough metadata analysis and find interdependencies throughout their entire data. An example of a security scan done by JFrog Xray is in Figure 5.4.

---

[8]https://www.versioneye.com/

[9]https://www.synopsys.com/software-integrity/software-composition-analysis-tools/black-duck-sca.html

[10]https://www.mend.io/

[11]https://www.applicationsecsanta.com/jfrog-xray

Chapter 6

Vulntron

As a part of this thesis, a tool *Vulntron* will be designed and implemented. In its core the Vulntron tool is an image SBOM vulnerability scanner and analyzer based on Syft and Grype, with a reporting UI using the DefectDojo reporting system[1] that will run inside the Red Hat OpenShift cluster.

6.1 Design

Vulntron tool will operate in two distinct modes.

Kafka mode. Firstly, it will facilitate the creation of a Kafka listener configured to monitor a specific topic, capturing and consuming messages. Upon detection of a new image build message, the tool initiates a security scan for the specified image. Once the scan is complete, the results, accompanied by a timestamp, are stored in the database. In the event of subsequent image pushes occurring within the next 24 hours, an automatic rescan is triggered. Alternatively, if the 24-hour timer expires without new image activity, a rescan will be initiated.

In summary, the Kafka mode is designed to monitor a specific Kafka topic for messages indicating new image builds. The workflow is as follows:

1. Capture messages signaling the creation of new images.

2. Initiate security scans for the identified images.

3. Store scan results with timestamps in the database.

4. Trigger automatic rescans within a given time frame if new images are pushed or upon timer expiration.

Automatic mode. In the automatic mode, the *Vulntron* tool functions autonomously to monitor and scan images within a designated namespace, without relying on Kafka messages. It dynamically retrieves information on all images utilized in deployments within the namespace by employing Red Hat OpenShift API interface. The description will be retrieved using API interface similar to the Red Hat OpenShift CLI `describe` commands. Subsequently, the tool parses the obtained output and proceeds to perform scans on all

---

[1] `https://www.defectdojo.org/`

parsed images. The steps, mirroring those in the Kafka scan, involve storing the scan results with timestamps in the database. This process ensures a comprehensive and automated assessment of images within the specified namespace, contributing to a proactive security stance.

To summarize, the Automatic mode will autonomously scan images within a designated namespace without relying on Kafka messages but only on the specified cluster, compared to Kafka mode, that will be more focused on collecting images from optionally more than one cluster. The Automatic mode workflow is as follows:

1. Utilize the Red Hat OpenShift API to retrieve information on images in deployments within the namespaces within the cluster.

2. Parse the output to identify and capture all images.

3. Configure image scanners that should be used to scan the captured images.

4. Configure notifications and channels that should be used to notify interested teams (eg. developers, quality engineers, product security team, etc.) about found vulnerabilities.

5. Conduct image analysis scans for the captured images using the selected scanners.

6. Store scan results in the database, following the same process as in the Kafka Mode.

7. Execute optimization procedures such as vulnerability deduplications and vulnerability reduction inside the database and skipping scans of already scanned images.

Figure 6.1 shows a high-level overview of the integration of the Vulntron tool inside the Red Hat OpenShift cluster.
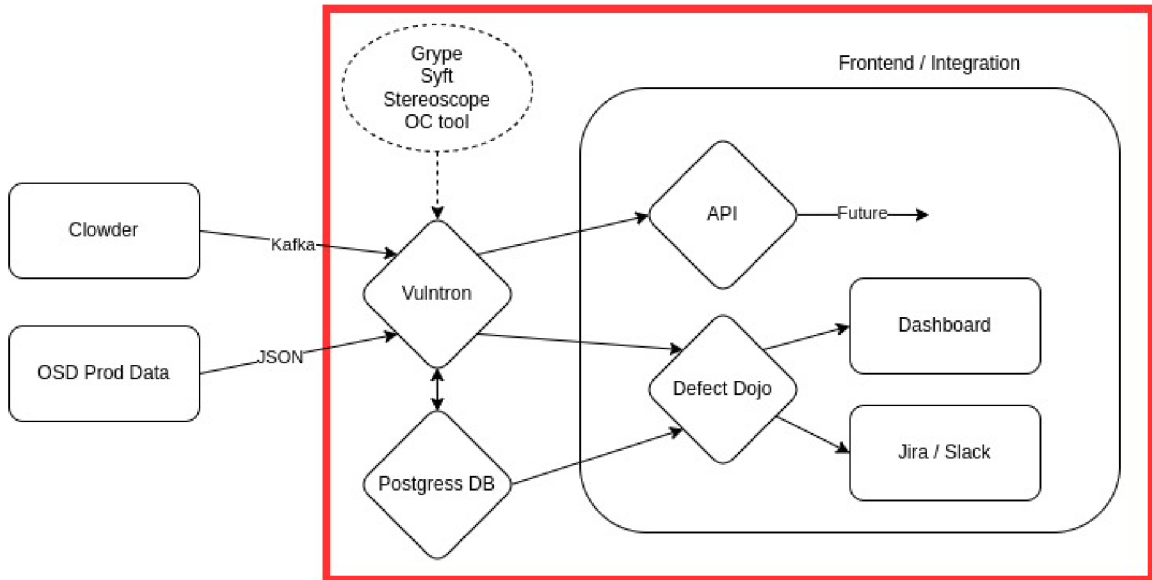
Figure 6.1: A high level overview of the Vulntron tool integration. (created by the author)

6.2 Implementation

Implementation was done using Golang v1.21.8[2] with Makefile to leverage the automatic build, run and clean process. The following steps cover mostly the Automatic mode as the Kafka mode was postponed, as explained in Section 6.7.

6.2.1 Project structure

A break down of the project file structure within the *Vulntron* directories with a short description of each folder or file:

- **bin/** Contains executable files after they are build.
    - **clean_dd_db** Script to clean the database.
    - **vulntron** Main executable for the Vulntron tool.

- **config.yaml** Configuration file for setting up the project environment.

- **deployment.yaml** YAML file for deploying the application in a containerized environment.

- **Dockerfile** Dockerfile to build the production ready image.

- **go.mod** and **go.sum** Define the module's properties and dependencies.

- **internal/** Application-specific modules.
    - **config/**
        * **config.go** Manages configurations loaded from config.yaml file.
    - **utils/**
        * **utils.go** Provides utility functions used across the project.
    - **vulntron_auto/**
        * **vulntron_auto.go** Handles the Automatic mode in the Vulntron system.
    - **vulntron_dd/**
        * **vulntron_dd.go** Manages DefectDojo reporting system API calls.
    - **vulntron_grype/**
        * **vulntron_grype.go** Implements the Grype tool vulnerability scanner used by the Vulntron tool.
    - **vulntron_kafka/**
        * **vulntron_kafka.go** Handles the Kafka mode in the Vulntron tool.
    - **vulntron_scanner_stats/**
        * **vulntron_scanner_stats.go** Gathers and reports scanning statistics.
    - **vulntron_syft/**
        * **vulntron_syft.go** Implements the Syft tool scanner for SBOM generation used by the Vulntron tool.

---

[2]`https://go.dev/doc/devel/release#go1.21.minor`

– **vulntron_trivy/**

    ∗ **vulntron_trivy.go** Implements the Trivy tool scanner for detecting vulnerabilities used by the Vulntron tool.

- **LICENSE** The license file of the project.

- **main.go** The main entry point of the application.

- **Makefile** Scripts to automate the compilation and installation of the project.

- **README.md** Provides a detailed description of the project, its dependencies, and how to run it.

- **scripts/**

  – **clean_dd_db.go** Go script for cleaning the database, part of the automation scripts.

6.2.2 Data preparation

The description of the namespace in JSON format can be retrieved from the Red Hat OpenShift cluster by executing the following command:

```
$ oc get pod -o json -n <name of namespace>
```

The JSON output file structure with descriptions of each field:

- **apiVersion**: (String) Specifies the API version, e.g., `v1`.

- **items**: (Array) A list of items, each representing a Red Hat OpenShift Pod configuration.

  – **apiVersion**: (String) Specifies the API version of an item.
  – **kind**: (String) The type of the Red Hat OpenShift resource, e.g., *Pod.*
  – **metadata**: (Object) Metadata of the Pod.
    ∗ **name**: (String) The name of the Pod.
    ∗ **namespace**: (String) The namespace of the Pod.
    ∗ **labels**: (Object) Key-value pairs that are attached to objects, often used to organize and to select subsets of objects.
    ∗ **annotations**: (Object) Key-value pairs that store non-identifying auxiliary data, used by tools and libraries.
  – **spec**: (Object) JSON list with the specification of the desired behavior of the Pod.
    ∗ **containers**: (Array) A list of containers included in the Pod.
      · **name**: (String) The name of the container.
      · **image**: (String) The Docker image for the container.
      · **ports**: (Array) The ports exposed by the container.
      · **env**: (Array) Environment variables available to the container.
      · **resources**: (Object) The resources required by the container.
      · **volumeMounts**: (Array) Mount paths for volumes.

* **volumes**: (Array) Volumes that can be mounted by containers.
  - **status**: (Object) Most recently observed status of the Pod.
    * **phase**: (String) The phase of the Pod (e.g., Running).
    * **conditions**: (Array) The current service state of the Pod.
    * **containerStatuses**: (Array) A list of containers and their status.
      · **containerID**: (String) A unique ID of the container.
      · **image**: (String) The image of the container.
      · **imageID**: (String) The image of the container with its hash.
      · **name**: (String) The name of the image.
      · **ready**: (Bool) Information if the container is ready to be used.
      · **started**: (Bool) Information if the container has already started.
      · **state**: (Object) The state of the container with the time of start.
    * **hostIP**: (String) IP address of the host to which the Pod is assigned.
    * **podIP**: (String) IP address assigned to the Pod.
    * **startTime**: (String) Start time of the Pod.

Since other deeper level fields are not used in the implementation, they have been purposefully left out of the previous description. Details are limited to the most important and relevant components.

In order to facilitate more effective scanning and eliminate the need to pass the entire JSON file through the program, the implementation uses data structures to store information about each namespace.

The `PodInfo` structure contains the information about each Pod and the structure is shown in Listing 6.1.

```
1  type PodInfo struct {
2      Pod_Name string `json:"Pod_Name"`
3      Namespace string `json:"Namespace"`
4      StartTime string `json:"StartTime"`
5      Containers []ContainerInfo `json:"Containers"`
6  }
```

Listing 6.1: PodInfo structure.

Likewise, information about every container in a Pod is stored in the `ContainerInfo` structure that is shown in Listing 6.2.

```
1  type ContainerInfo struct {
2      Container_Name string `json:"Container_Name"`
3      Image string `json:"Image"`
4      ImageID string `json:"ImageID"`
5  }
```

Listing 6.2: ContainerInfo structure.

Each `PodInfo` contains information about the namespace to which it belongs. Namespaces are not implemented as standalone structures. Data about the Pod's name, start time, and containers are also stored in this structure.

Information such as image name, image ID, and container name are all contained in the structure `ContainerInfo` that holds details about individual containers. When an image is tagged as `image:latest`, it is not helpful for the image scanning process as there is no exact identification of this image, so both fields `Image` and `ImageID` are included. Each image is uniquely identified by its hash and image name, which are provided in the `ImageID` option. Figure 6.2 shows the source of each data field that was loaded into the structures.
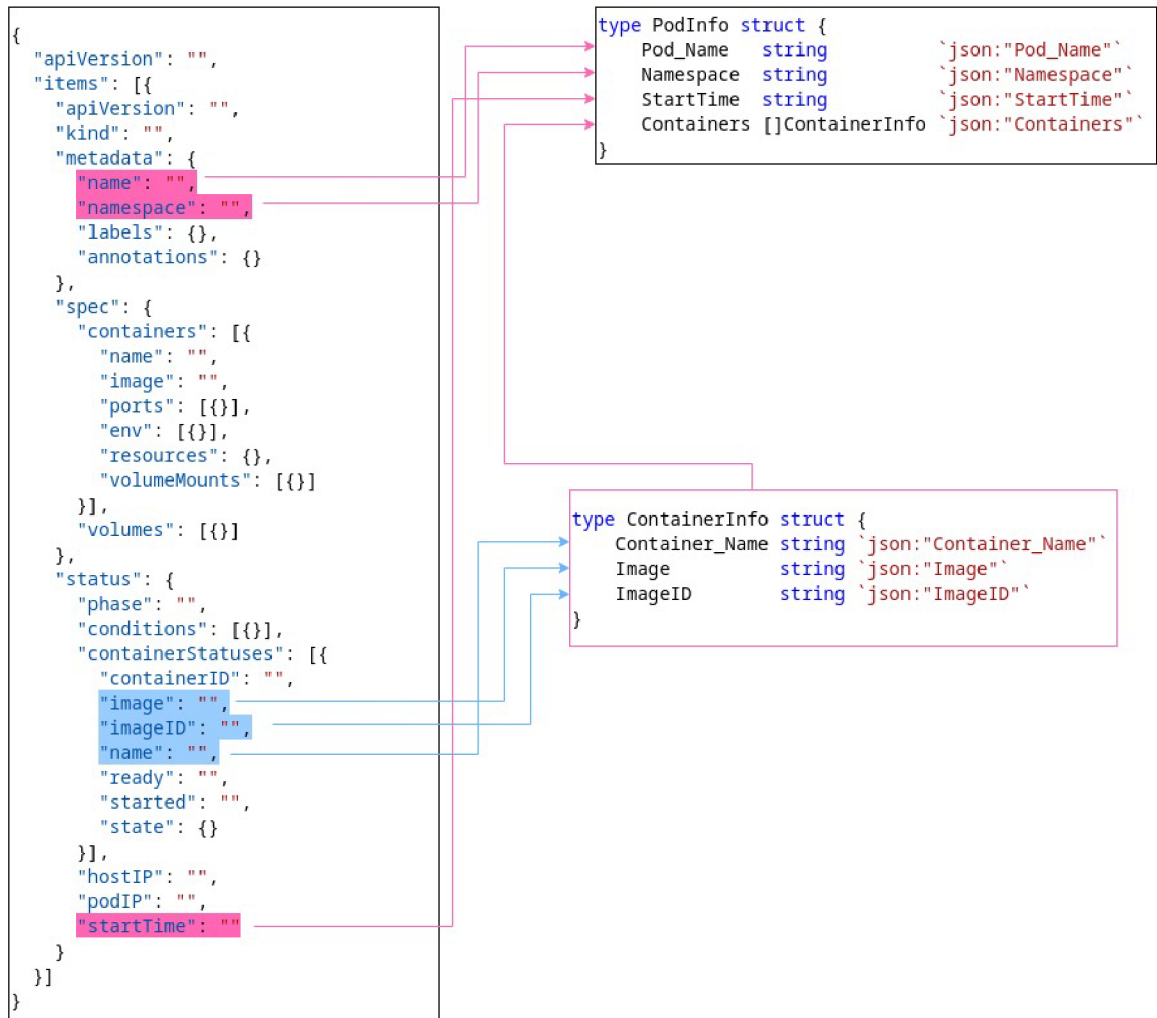
Figure 6.2: Relation between the cluster description output and custom data structures. (created by the author)

6.2.3 Scanning

After the data are ready, the first step is to set up a scanning environment that is specific to the selected scanning types.

The Vulntron tool is both a vulnerability collector and a vulnerability scanner. It gathers and filters the results of every test scan and then loads them into the DefectDojo reporting system. Through this interface, vulnerabilities can be managed, tickets can be automatically generated and loaded into Jira using webhooks, and notifications about crit-

ical vulnerabilities in particular namespaces can be sent automatically to designated Slack channels.

Modularity

Through the following steps, a new scanning tool may be imported into the Vulntron tool which makes it easier to automate the execution of scans and loading the results into the DefectDojo reporting portal. The steps are as follows:

1. Add the necessary configuration for the new scanning tool into `config.yaml` file.

2. Create an entry inside the `config.yaml` file within `scan_types` list in the format shown in Listing 6.3. The following are fields inside new scanner configuration.

   - **name** (string) – the name of the scan type compatible with the DefectDojo supported scan types[3].
   - **engName** (string) – the name of the imported engagement. It should be clear what type of scans this engagement contains. There are no strict naming rules.
   - **function** (string) – the name of the scanning function that must be imported into the `vulntron_auto.go` file and must match the schema shown in Listing 6.4. This function should accept two arguments: configuration (which could require adjustments in other configuration structures within `config.yaml`) and `imageID` (a string containing data about the image to be scanned). The function must return either an error, if the scanning or any setup steps fail, or a string that specifies the path to the scan results.
   - **enabled** (bool) A switch that disables the scan type from being executed.

3. Add the entry into `scanFunctionMap` that will map the real function to the one from configuration file as shown in Listing 6.5.

If all steps are correctly implemented, the automated scan should recognize the new scan type, execute the scan on the selected images, and import the scan results under the selected engagement name within each scanned namespace.

```
1  scan_types:
2    - name: "Anchore Grype"
3      engName: "Grype_eng"
4      function: "RunGrype"
5      enabled: true
```

Listing 6.3: A scan type import example.

```
func RunTrivy(cfg config.Config, imageID string) (string, error)
```

Listing 6.4: A scan function signature.

---

[3]`https://defectdojo.github.io/django-DefectDojo/integrations/parsers/file/`

```
var scanFunctionMap = map[string]ScanFunction{
        "RunGrype": vulntron_grype.RunGrype,
        // "RunAnotherScanner": vulntron_other.RunAnotherScanner
}
```
Listing 6.5: A scan function map example.

6.2.4 DefectDojo reporting

Scan reports are automatically imported into the DefectDojo reporting system. To ensure that the scans are correctly imported, the Vulntron tool ensures, before each scan, that the desired settings are set within the DefectDojo instance used.

Environment variables that are needed for the Vulntron tool to operate:

- **DEFECT_DOJO_USERNAME** Username of the user that will own and import the scans into DefectDojo reporting system.

- **DEFECT_DOJO_PASSWORD** Password of the **DEFECT_DOJO_USERNAME** user. This value should be filled inside Secrets and loaded into environment within the Red Hat Open-Shift cluster.

- **DEFECT_DOJO_URL** The URL of the DefectDojo reporting system that recognizes the user specified above and where the scans results should be uploaded.

- **DEFECT_DOJO_SLACK_CHANNEL** The slack channel that should be notified by Defect-Dojo reporting system.

- **DEFECT_DOJO_SLACK_OAUTH** The slack DefectDojo reporting system bot token to access the notification channel.

DefectDojo internal setup

Using the credentials provided in the environment variables, the Vulntron tool will request an API token on the specified URL.

After a successful API token retrieval, all of the next API calls will be performed using the retrieved API token.

The file `config.yaml` in the section `defect_dojo` contains a configuration for the DefectDojo reporting system, which should be set before each scanning session to ensure that the desired behavior will be configured. The current complete DefectDojo reporting system configuration is shown in Listing 6.6.

```
1  defect_dojo:
2    enable_deduplication: true
3    delete_duplicates: true
4    max_duplicates: 0
5    slack_notifications: true
```
Listing 6.6: The default DefectDojo reporting system configuration.

All currently used system settings options are accessible in the UI of DefectDojo reporting system navigating to System Settings page. There is also an option to change the settings using an API call to `GET /api/v2/system_settings/` endpoint.

The settings that are set by default using the Vulntron tool are:

- **enable_deduplication** – enable finding duplicate vulnerabilities after importing automatically by the DefectDojo reporting system.

- **delete_duplicates** – found duplicate vulnerabilities will be removed.

- **max_duplicates** – the number of duplicates to keep before deleting.

- **slack_notifications** – a switch to enable sending Slack notifications. For more information, refer to Section 6.2.4.

Adding new system settings for the DefectDojo reporting system requires updating the section above in `config.yaml` and updating the function `UpdateSystemSettings()` within the `vulntron_dd.go` file to include the newly added configuration.

Internal notifications

The notification or alerts in terms of the DefectDojo reporting system are enabled by default and can be manually configured using the UI interface by navigating to `Settings -> Notifications` on the sidebar.

Slack integration

Slack integration must be installed as an application in the selected Slack workspace following the DefectDojo reporting system *Slack integration tutorial*[4].

The OAUTH API token and channel name should be stored as a Secret and loaded into the environment as an environment variable, as mentioned in Section 6.2.4.

The types of selected notifications that should be received and also the types of alerts that should be shown inside the DefectDojo reporting system is configurable by navigating to `Settings -> Notifications` as shown in Figure 6.3.

The types of notifications are customizable for individual users, but for the Vulntron tool use case, the `System` notification settings should be changed.

After the Slack notification on-boarding, the automatic step in scanning pipeline takes care of checking the `System Settings` of DefectDojo reporting tool, and if the settings were changed, they are automatically updated to match the configuration from the environment variables.

A successful on-boarding of the Slack notifications to scan an image using two scanning tools can be seen in Figure 6.4.
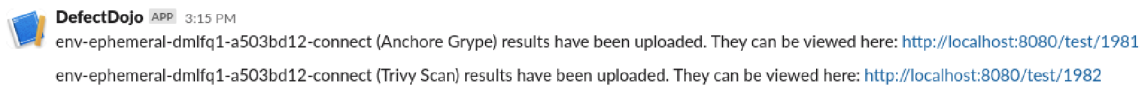
Figure 6.4: An automatic Slack notifications for importing image scans.

Management scripts

The process of unlinking and removing all scans from all the engagements, and engagements from product types, takes a long time. It became necessary to automate the entire database clearing process during the Vulntron tool's development and testing phase to save time compared to the manual method.

---

[4]`https://support.defectdojo.com/en/articles/8944899-configure-a-slack-integration`

Figure 6.3: Notification selection for System administrators.

For this purpose, a script `scripts/clean_dd_db.go` was created. The script is built using the command `make build-clean-db` and can be run using the command from the root of the project (which is the default location when interacting with a Pod in the Red Hat OpenShift debugging cluster) `./bin/clean_dd_db`.

The script leverages the environment variables similar to the main Vulntron application and after the command finishes successfully all data should be removed. This process may take some time to complete.

An example of logs from the database cleaning process are shown in Figure 6.5.

6.3   Usage
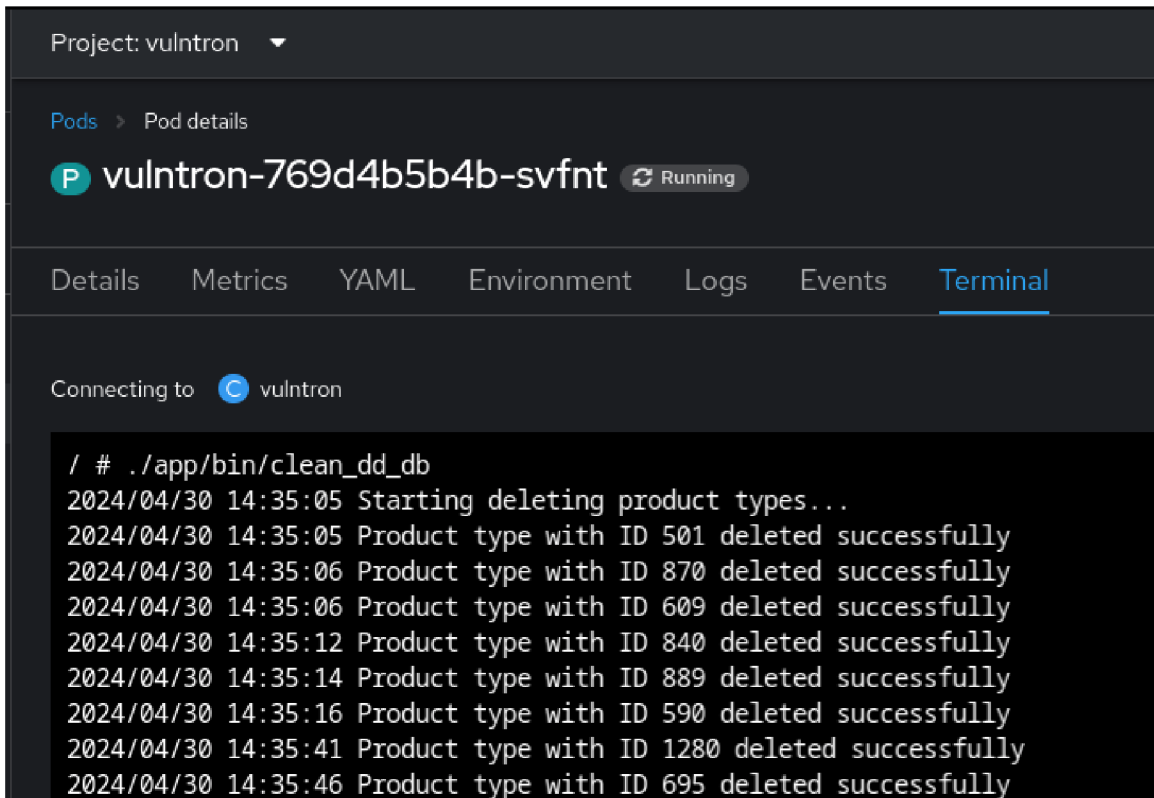
The Vulntron tool was developed to be portable and to be used in the Red Hat OpenShift cluster as a global image security scanner and reporter, and also locally for individual security personnel to monitor the selected Red Hat OpenShift cluster from a local computer and take appropriate actions based on the results.

6.3.1   Local setup

To setup the Vulntron tool locally, the following steps are required:

1. Set up the DefectDojo reporting system.

   - Download the DefectDojo reporting system from GitHub[5] and install it following the provided instructions.

---

[5]`https://github.com/DefectDojo/django-DefectDojo`

Figure 6.5: Database cleaning logs using the `clean_dd_db` script.

- Start the DefectDojo reporting system on `loaclhost` and port 8080 using the command `./dc-up-d.sh postgres-redis`.
- Set the local environment variables `DEFECT_DOJO_USERNAME`, `DEFECT_DOJO_PASSWORD`, `DEFECT_DOJO_URL`, `DEFECT_DOJO_SLACK_CHANNEL` and `DEFECT_DOJO_SLACK_OAUTH` as described in Section 6.2.4.

2. Login into the desired Red Hat OpenShift cluster and retrieve a token from a user or service account. The user or service account should has cluster administrator privileges on level of listing the Pods and accessing the secrets needed.

3. Set the local environment variables for the Red Hat OpenShift access:

- `OC_TOKEN` An Red Hat OpenShift token used to access the cluster.
- `OC_NAMESPACE_LIST` A comma separated list of namespaces to be scanned.
- `OC_NAMESPACE_REGEX` A regex to match the namespaces to be scanned. If not empty the regex takes precedence before the `OC_NAMESPACE_LIST`.

4. Update the `config.yaml` configuration file present in the program root folder to match the desired criteria.

5. Build the Vulntron tool binary from the root folder with `make build`.

6. Run the program manually using `./bin/vulntron -config config.yaml`.

7. Or run and build the program automatically with `make run`.

An example of setting all the variables that can be used in the setup steps is shown in Listing 6.7. This setup will scan all ephemeral namespaces in the selected cluster that follows the specified regex and store the scans in the DefectDojo reporting system running on `localhost:8080`.

```
export DEFECT_DOJO_URL="http://localhost:8080"
export DEFECT_DOJO_USERNAME="admin"
export DEFECT_DOJO_PASSWORD="password"
export DEFECT_DOJO_SLACK_OAUTH="<oauth token>"
export DEFECT_DOJO_SLACK_CHANNEL="vulntron-notifications"
export OC_TOKEN="<Red Hat OpenShift token>"
export OC_NAMESPACE_REGEX="^ephemeral-[a-zA-Z0-9]{6}$"
```

Listing 6.7: An example setup of enviroment variables for local deployment.

6.3.2 Red Hat OpenShift Setup

Steps to set up and install the Vulntron tool in the Red Hat OpenShift cluster are similar to the local setup:

1. Create a new namespace that will store the Pods of the Vulntron tool and the DefectDojo report portal.

2. Install the DefectDojo reporting system on the desired cluster in the newly created Vulntron namespace. Installation may be done using Helm charts[6] or any other method mentioned in the installation guide[7].

3. Get access to an OpenShift user or service account. The user or service account should has cluster administrator privileges on level of listing the Pods and accessing the Secrets needed.

4. Set up the secrets containing all the environment variables mentioned in steps for a local deployment in Section 6.3.1.

5. Edit the last line of the provided `Dockerfile` in the project root with time interval that the Vulntron tool should run in. The default value is 2 hours.

6. Update the second part of the `Dockerfile` to contain the binaries needed for additional scanners besides Grype, if they use not only Go libraries but also CLI tools.

7. Build the Vulntron tool image using the provided `Dockerfile`. The `Dockerfile` will firstly build the binary using the `golang:alpine` image that contains all the needed binaries. After the binary is built the whole project is copied to `alpine` image and this process will greatly reduce the size of the image from 2GB or more, to less then 200MB.

8. Tag and store the created image on an image storage platform (eg. Quay.io).

9. Edit the `deployment.yaml` file in the project root to match the created Secretes from step 4 and specify the path to the Vulntron tool image.

---

[6] `https://github.com/DefectDojo/django-DefectDojo/tree/dev/helm/defectdojo`
[7] `https://defectdojo.github.io/django-DefectDojo/getting_started/installation/`

10. Login into the Red Hat OpenShift cluster from the local terminal and select the working Vulntron namespace with DefectDojo reporting system running.

11. Deploy the Vulntron tool to the Red Hat OpenShift cluster using the command `oc apply -f deployment.yaml`.

If everything is set up correctly, the Vulntron tool will begin the scanning process. An example on a successful deployment is shown in Figure 6.6.
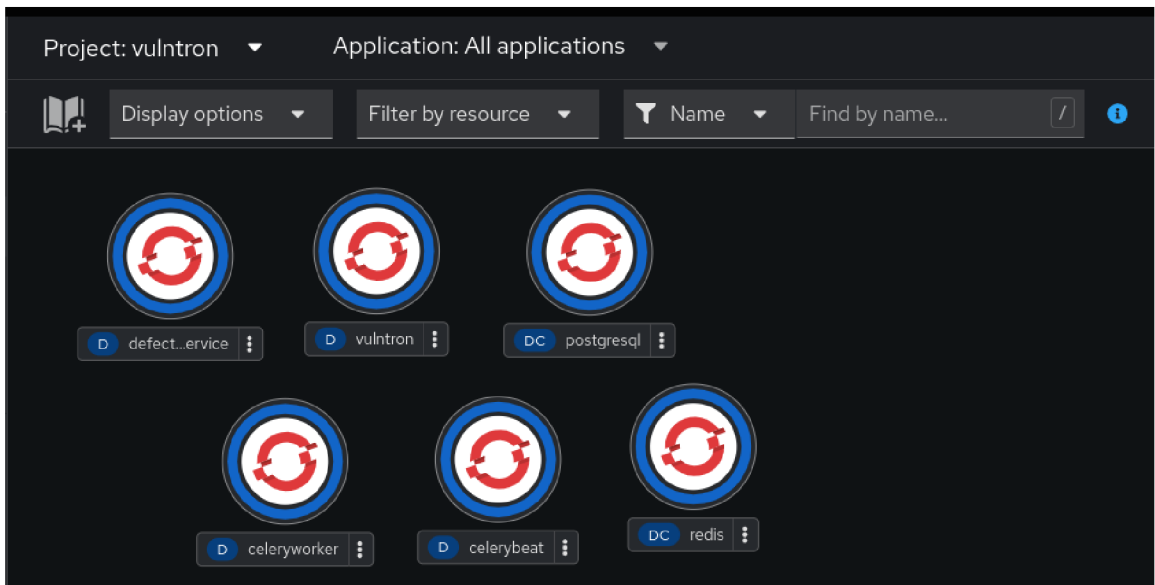

Figure 6.6: Vulntron tool deployed in the Red Hat OpenShift cluster.

6.4   Performance and Resource Management

During the development of the Vulntron tool, a consideration to resource management was given while deploying within the Red Hat OpenShift cluster. It is important that the tool does not consume all available resources, as security scanning is not the primary function of the cluster.

The first scanning strategy was to scan every image in every Pod. This method would only increase the database size without adding any value, so it was determined to be inefficient. A more efficient solution was to create a system to search for duplicate images before scanning.

Another boost in performance was supposed to be seen by using the built-in Golang concurrency, but this was discarded as concurrency during the scanning phase would be resource intensive and the processes would be rate limited by the Red Hat OpenShift cluster management.

The time gain of using concurrency during the Vulntron tool setup phases would be negligible, so it was omitted as well.

A problem that may occur during the local scanning is that the pulled images are associated with docker images and that is why they are not cleaned by the Grype scanner. To solve this issue, it is recommended to remove unused images regularly using the command `docker system prune -a -f`. This problem does not occur when the Vulntron tool is

deployed inside the Red Hat OpenShift cluster as the deployed image does not contain the `docker` binary and the images are cleaned automatically by the scanner.

The monitoring of memory resources during one scan pass is shown in Figure 6.7. On the left side of the graph the memory usage increases as images are being pulled to be scanned. Moving to the right, there are dips when the images are being cleaned and the vulnerability deduplication takes place.
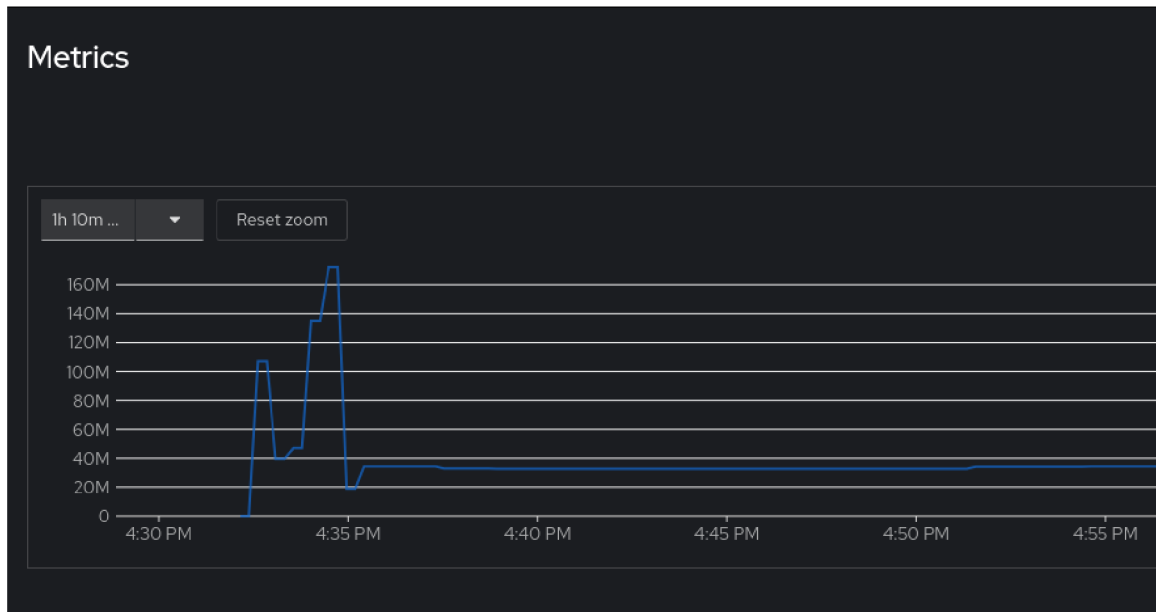

Figure 6.7: Memory resource monitoring during one scan.

6.5 Testing

During the whole development phase the testing was being done in both local and ephemeral cluster environments. The ephemeral cluster is mainly used to test Merge Request build images (which are created automatically for each new Merge Request) and testing images, so it was expected that the number of vulnerabilities discovered may be higher than in a stage or a production environment.

The testing was carried out on weekends and also during peak times to ensure that the scanning process will not interfere with the regular functionality of the cluster. Peak times were chosen by the largest number of namespaces reserved, which was more than 100. Listing 6.8 shows the result of a scan performed during peak time. Observing the memory and CPU resources shows that the Pod was not being rate-limited based on the fact that the implementation was built to do the scans sequentially.

```
2024/05/02 17:13:51 vulntron_scanner_stats.go:36: Scanning complete:
        Scanned 452 pods with 582 images using 2 scanning tools
        in 49m59.86029193s
2024/05/02 17:13:51 vulntron_auto.go:92: Scanning complete!
Every 7200.0s: /app/bin/vulntron --config /app/config.yaml
```

Listing 6.8: A scan pass done during peak time.

The deployment was observed over an extended period and the measured values were reviewed with the lead engineers and management. It was concluded that these values are suitable for the objectives of this project.

6.6   Testing Evaluation

By implementing the Vulntron tool within the Red Hat OpenShift ephemeral cluster environment, the system was configured to conduct a scan of the entire cluster at two-hour intervals over a span of nearly two weeks, as documented at the time of writing. During this period, the Vulntron tool utilized two image scanning tools, Grype and Trivy, enabling the detection of more than 97,000 potential vulnerabilities. The report of this monitoring is shown in Figure 6.8 as a dashboard view of the DefectDojo UI.

It is important to note that a portion of the detected vulnerabilities were duplicates; after consolidating these findings, the adjusted total was approximately 51,000 unique vulnerabilities. This figure does not represent the conclusive count, as a significant number of these vulnerabilities have either been previously resolved or have been classified under the 'will not fix' category, indicating a deliberate decision to not address these issues due to various considerations such as risk assessment outcomes or the obsolescence of the affected components.

6.7   Future Improvements

The implemented part of the Vulntron tool will serve as a base and an example of the individual scanner capabilities and the DefectDojo reporting portal features. After the Vulntron tool is approved, it will start to be adopted by specific teams and more new features will be added.

Kafka mode. There was a lot of interest in the Vulntron tool during its initial design phase from a team that was keen to use Kafka messages to scan new images. However, the ongoing dynamic processes at Red Hat had an effect on this team. As such, there has been a temporary delay in the development of the Kafka mode of Vulntron.

The team is in the process of switching between two versions of Kafka, and the integration of the Vulntron tool cannot be done effectively due to the instability of the current environment.

Having that in mind, the Vulntron Kafka mode is currently effective in subscribing to the selected topic and parsing the messages from the subscribed topic. As the scanning logic was designed to have a slight differences from the Vulntron automatic mode, it is prepared for future completion by leveraging some functionality from the Automatic mode.

Jira automation. As the automated Slack messages are created during the scanning process, the automatic creation of Jira tickets is planned for the future. The tool may be deployed in the production environment, and it is necessary to comply with the Red Hat internal security compliance structures. Currently, the Vulntron tool is being reviewed with no end date specified, so that is why the Jira on-boarding was not implemented in this version.
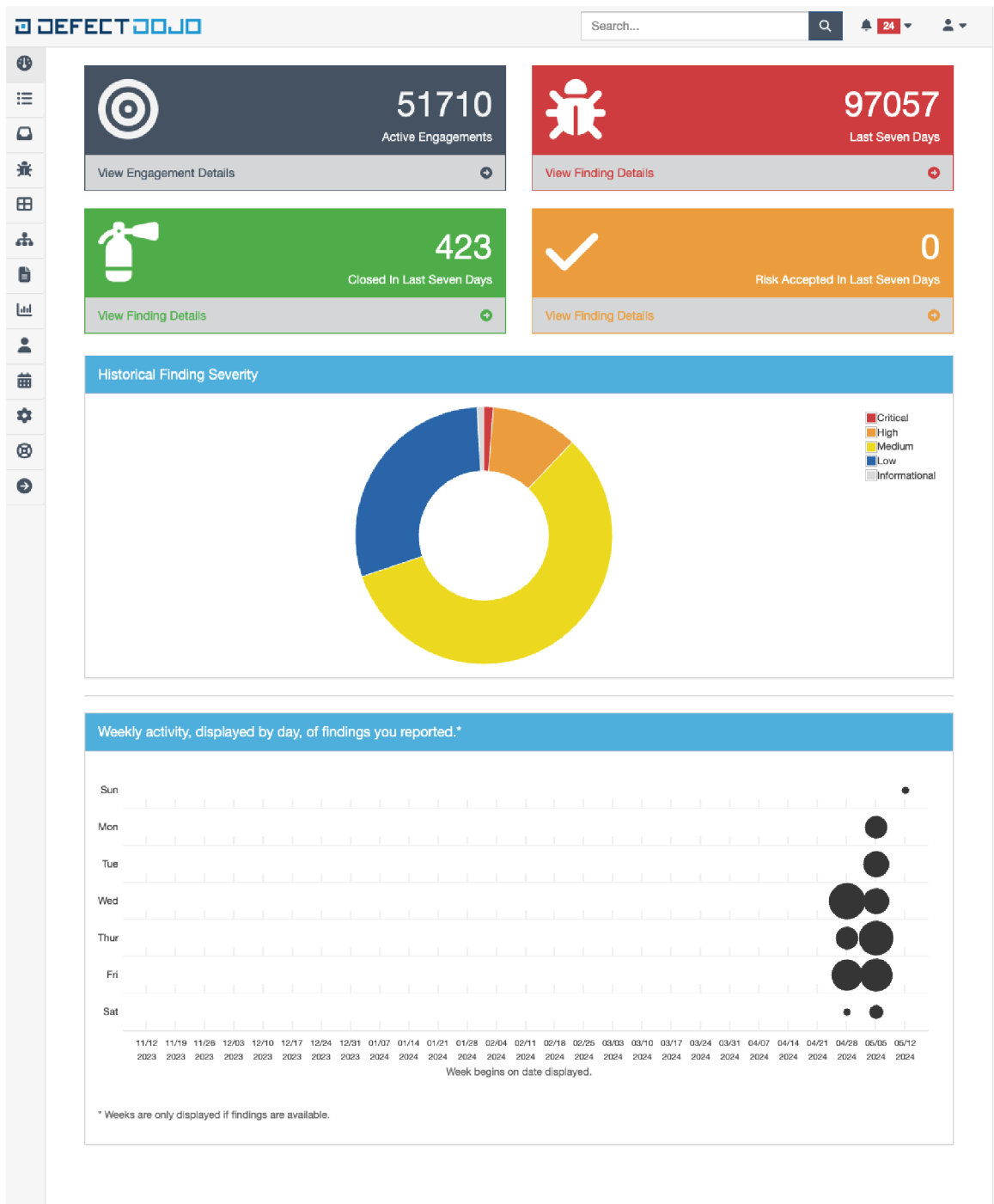
Figure 6.8: The DefectDojo vulnerability report of the ephemeral cluster scanning.

Vulnerability grouping. The point of the last step of the Vulntron tool is to report the vulnerabilities using the selected channel i.e either a Slack notification or automatic Jira card creation.

During the testing phase, the scans of more than 1000 Pods yield more than 65000 vulnerabilities in the criticality range from Informational to Critical. It would be unfeasible to handle that many vulnerabilities and create cards for each of them.

The requirement is to group them by CWE, create a Jira card for each CWE and assign it to teams that own the vulnerable namespace.

A specific reporting procedure will be discussed after the Vulntron tool is on-boarded with the Management and Product security teams.

Chapter 7

Conclusion

This thesis focuses primarily on the design, development, and implementation of the Vulntron tool as well as the integration of the tool into the available environment within the Red Hat company using the Red Hat OpenShift. Vulnerability assessment within container images is a complex field to study as there are many hidden layers that need investigating. The study of the given subject proved to be very beneficial in the development process and also studies of the current state of container security resulted in several areas that will have to be studied and improved in the future, in particular the responsive scanning and reporting.

Contributions. In general, the main output of this work is the Vulntron tool, which extends the security initiative of containerized environments by automatically detecting vulnerabilities and reporting them. The introductory chapters of this thesis contribute to an overall understanding of containers, container orchestration and container security in an extensive analysis of the vulnerabilities that affect container images and the effectiveness of existing tools to manage these vulnerabilities.

Limitations and Future Work. While the Vulntron tool offers a solid framework in the area of vulnerability assessment, there is still a need for more improvements and features. Future work may include extending its coverage to other types of vulnerability scanners and adding more features based on the DefectDojo reporting system, as the implementation of the Vulntron tool explores only a small part of them. There are also potential upgrade opportunities in terms of increasing the performance of the Vulntron tool in larger clusters and more complex environments.

Final Thoughts. Within the software development lifecycle, the increase in container technology usage requires improvements in security measures in multiple areas. This thesis has laid the foundation for further research and development in the area of container image analysis and security, which is really important to ensure that not only the Red Hat products but also other software products are safe from vulnerabilities and attackers.

Bibliography

[1] ABHISHEK, M. K.; RAO, D. R. and SUBRAHMANYAM, K. Framework to Deploy Containers using Kubernetes and CI/CD Pipeline. *International Journal of Advanced Computer Science and Applications*. The Science and Information Organization, 2022, vol. 13, no. 4. Available at: `http://dx.doi.org/10.14569/IJACSA.2022.0130460`. Online; Accessed: Nov 2023.

[2] DUNCAN, J. and OSBORNE, J. *OpenShift in Action*. Manning Publications, 2018. ISBN 9781617294839.

[3] FREYBURG, P. *Module Orchestration of Multitenant Systems*. Brno, Czech Republic, 2023. Master's Thesis. Brno University of Technology, Faculty of Information Technology. Supervisor ALEŠ SMRČKA, P. Available at: `https://www.vut.cz/studenti/zav-prace/detail/142729`. Online, Accessed: April 2024.

[4] HUMAYUN, M.; NIAZI, M.; JHANJHI, N.; ALSHAYEB, M. and MAHMOOD, S. Cyber Security Threats and Vulnerabilities: A Systematic Mapping Study. *Arabian Journal for Science and Engineering*, Jan 2020, vol. 45. Available at: `https://doi.org/10.1007/s13369-019-04319-2`. Online; Accessed: Dec 2023.

[5] JFROG. Introducing JFrog Xray: Enhancing Visibility and Security in Software Component Management. *Press Release*, 2016. Available at: `https://www.jfrog.com/press-releases/jfrog-introduces-jfrog-xray-unprecedented-visibility-software-components/`. Online, Accessed: April 2024.

[6] KUBERNETES AUTHORS. *Kubernetes Documentation*. 2022. Available at: `https://kubernetes.io/docs`. Online, Accessed: April 2024.

[7] LOVE, C. and VYAS, J. *Core Kubernetes*. Manning Publications, 2022. ISBN 9781617297557.

[8] MELL, P.; SCARFONE, K. and ROMANOSKY, S. *The Common Vulnerability Scoring System (CVSS) and its Applicability to Federal Agency Systems*. IR 7435. NIST, 2007. Available at: `https://nvlpubs.nist.gov/nistpubs/ir/2015/NIST.IR.7435.pdf`. Online; Accessed: Nov 2023.

[9] MORAVCIK, M.; SEGEC, P.; KONTSEK, M.; URAMOVA, J. and PAPAN, J. Comparison of LXC and Docker Technologies. In: *2020 18th International Conference on Emerging eLearning Technologies and Applications (ICETA)*. 2020, p. 481–486. ISBN 978-1-6654-2226-0.

[10] NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. *Guide for Applying the Risk Management Framework to Federal Information Systems: A Security Life Cycle Approach.* Special Publication 800-37. U.S. Department of Commerce, National Institute of Standards and Technology, 2020. Available at: `https://csrc.nist.gov/publications/detail/sp/800-37/rev-2/final`. Online, Accessed: April 2024.

[11] NTIA MULTISTAKEHOLDER PROCESS ON SOFTWARE COMPONENT TRANSPARENCY - FRAMING WORKING GROUP. *Framing Software Component Transparency: Establishing a Common Software Bill of Material (SBOM).* National Telecommunications and Information Administration, November 2019. Available at: `https://www.ntia.gov/files/ntia/publications/framingsbom_20191112.pdf`. Online; Accessed: Nov 2023.

[12] RED HAT. *What is Clair?* 2019. Available at: `https://www.redhat.com/en/topics/containers/what-is-clair`. Online; Accessed: Dec 2023.

[13] RED HAT. *Container Image Security Vulnerability Whitepaper.* 2021. Available at: `https://www.redhat.com/en/resources/container-image-security-vulnerability-whitepaper`. Online, Accessed: April 2024.

[14] RED HAT. *OpenShift Container Platform Documentation.* 2022. Available at: `https://docs.openshift.com/container-platform/4.15/`. Online, Accessed: April 2024.

[15] RED HAT. *Red Hat Quay* Datasheet. 2023. Available at: `https://www.redhat.com/en/resources/quay-datasheet`. Online, Accessed: April 2024.

[16] RED HAT. *What is Vulnerability Management.* 2023. Available at: `https://www.redhat.com/en/topics/security/what-is-vulnerability-management`. Online, Accessed: April 2024.

[17] RICE, L. *Container Security: Fundamental Technology Concepts that Protect Containerized Applications.* O'Reilly Media, 2020. ISBN 9781492056676. Available at: `https://books.google.cz/books?id=74biDwAAQBAJ`.

[18] SERVICENOW. *What Is Vulnerability Management?* 2024. Available at: `https://www.servicenow.com/products/security-operations/what-is-vulnerability-management.html`. Online, Accessed: April 2024.

[19] STAUDEK, J. and HANÁČEK, P. *Bezpečnost informačních systémů.* 1st ed. Praha: Úřad pro státní informační systém, 2000. ISBN 80-238-5400-3.

[20] TENABLE. *Nessus Container Security.* 2023. Available at: `https://docs.tenable.com/vulnerability-management/Content/ContainerSecurity/Dashboard.htm`. Online; Accessed: Dec 2023.

[21] WONG, A. Y.; CHEKOLE, E. G.; OCHOA, M. and ZHOU, J. On the Security of Containers: Threat Modeling, Attack Analysis, and Mitigation Strategies. *Computers & Security*, 2023, vol. 128, p. 103140. ISSN 0167-4048. Available at: `https://www.sciencedirect.com/science/article/pii/S0167404823000500`. Online; Accessed: Nov 2023.