NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

POSTGRADUATE STUDIES PROGRAM
”COMPUTER SYSTEMS: SOFTWARE AND HARDWARE”

MSc THESIS

Container Image Vulnerability Management in Kubernetes

Marios M. Levogiannis

Supervisors: **Alex Delis**, Professor NKUA
Dimitris Mitropoulos, Assistant Professor NKUA

ATHENS

OCTOBER 2024

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
”ΥΠΟΛΟΓΙΣΤΙΚΑ ΣΥΣΤΗΜΑΤΑ: ΛΟΓΙΣΜΙΚΟ ΚΑΙ ΥΛΙΚΟ”

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Διαχείριση Ευπαθειών Εικόνων Κοντέινερ σε Kubernetes

Μάριος Μ. Λεβογιάννης

Επιβλέποντες: **Αλέξης Δελής**, Καθηγητής ΕΚΠΑ
Δημήτρης Μητρόπουλος, Επίκουρος Καθηγητής ΕΚΠΑ

ΑΘΗΝΑ

ΟΚΤΩΒΡΙΟΣ 2024

MSc THESIS

Container Image Vulnerability Management in Kubernetes

Marios M. Levogiannis
S.N.: M1539

SUPERVISORS: **Alex Delis**, Professor NKUA
Dimitris Mitropoulos, Assistant Professor NKUA

EXAMINATION COMMITTEE: **Alex Delis**, Professor NKUA
Dimitris Mitropoulos, Assistant Professor NKUA
Mema Roussopoulos, Professor NKUA

Examination Date: 22 October 2024

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Διαχείριση Ευπαθειών Εικόνων Κοντέινερ σε Kubernetes

Μάριος Μ. Λεβογιάννης
Α.Μ.: Μ1539

ΕΠΙΒΛΕΠΟΝΤΕΣ: **Αλέξης Δελής**, Καθηγητής ΕΚΠΑ
Δημήτρης Μητρόπουλος, Επίκουρος Καθηγητής ΕΚΠΑ

ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΤΡΟΠΗ: **Αλέξης Δελής**, Καθηγητής ΕΚΠΑ
Δημήτρης Μητρόπουλος, Επίκουρος Καθηγητής ΕΚΠΑ
Μέμα Ρουσσοπούλου, Καθηγήτρια ΕΚΠΑ

Ημερομηνία Εξέτασης: 22 Οκτωβρίου 2024

ABSTRACT

Kubernetes and containers have revolutionized the way applications are deployed in production environments. To provide portability and reproducibility, every container image incorporates all the dependencies required by the application. Although this is an advantage from the deployment standpoint, it also expands the attack surface as different versions of the same dependencies with different vulnerabilities can be bundled in multiple container images.

To tackle this problem we will develop an approach to effectively manage container image vulnerabilities in Kubernetes. Our approach will include (1) scanning of both existing and to-be-deployed container images for known vulnerabilities, (2) visibility to the current status of vulnerable container images in a Kubernetes cluster and (3) policy enforcement for new container images by rejecting deployment of container images affected by multiple vulnerabilities, all in a Kubernetes-native way. The solution will leverage existing open-source components, where applicable (e.g. container image scanners).

SUBJECT AREA: Vulnerability Management, Kubernetes

KEYWORDS: Kubernetes, Container Image, Application Dependency, Vulnerability, Vulnerability Management, Vulnerability Scanning

ΠΕΡΙΛΗΨΗ

Το Kubernetes και τα κοντέινερ έχουν φέρει την επανάσταση στον τρόπο με τον οποίο εγκαθίστανται οι εφαρμογές σε παραγωγικά περιβάλλοντα. Για να προσφέρουν φορητότητα και αναπαραγωγιμότητα, κάθε εικόνα κοντέινερ ενσωματώνει όλες τις εξαρτήσεις που απαιτούνται από την εφαρμογή. Παρότι αυτό είναι ένα πλεονέκτημα από την οπτική της εγκατάστασης, επεκτείνει επίσης την επιφάνεια επίθεσης καθώς διαφορετικές εκδόσεις των ίδιων εξαρτήσεων με διαφορετικές ευπάθειες μπορεί να περιλαμβάνονται σε πολλαπλές εικόνες κοντέινερ.

Για την αντιμετώπιση αυτού του προβλήματος θα θα αναπτύξουμε μια προσέγγιση για την αποτελεσματική διαχείριση των ευπαθειών εικόνων κοντέινερ σε Kubernetes. Η προσέγγισή μας θα περιλαμβάνει (1) σάρωση τόσο των υπάρχοντων όσο και των προς-εγκατάσταση εικόνων κοντέινερ για γνωστές ευπάθειες, (2) ορατότητα στην τρέχουσα κατάσταση των ευπαθών εικόνων κοντέινερ σε ένα σύμπλεγμα Kubernetes και (3) επιβολή πολιτικής για νέες εικόνες κοντέινερ με την απόρριψη της εγκατάστασης εικόνων κοντέινερ που επηρεάζονται από πολλαπλές ευπάθειες, όλα με έναν εγγενή ως προς το Kubernetes τρόπο. Η λύση θα αξιοποιήσει υπάρχοντα στοιχεία ανοικτού λογισμικού, όπου αυτό είναι εφικτό (π.χ. σαρωτές εικόνων κοντέινερ).

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Διαχείριση Ευπαθειών, Kubernetes

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Kubernetes, Εικόνες Κοντέινερ, Εξάρτηση Εφαρμογής, Ευπάθεια, Διαχείριση Ευπαθειών, Σάρωση Ευπαθειών

Στους γονείς μου

ACKNOWLEDGMENTS

I would like to thank my supervisors, Alex Delis and Dimitris Mitropoulos, for the chance they gave me to work on this project and also for their comments and overall cooperation while working on this thesis.

Furthermore, I would like to thank my co-workers at GRNET's Security Department for their valuable input during the course of this work.

CONTENTS

1 INTRODUCTION 27

1.1 Problem Description **27**

1.2 Our Contribution **27**

1.3 Thesis Structure **27**

2 BACKGROUND AND RELATED WORK 29

2.1 Application Dependencies **29**
2.1.1 Operating System Packages 30
2.1.2 Language Packages 31

2.2 Microservices **32**
2.2.1 Decoupled Development and Deployment 32
2.2.2 API-Based Communication 33
2.2.3 Scalability 33
2.2.4 High Availability 34

2.3 Containers **34**
2.3.1 Container Image 35
2.3.1.1 Registry 35
2.3.1.2 Repository 35
2.3.1.3 Tag 35
2.3.1.4 Digest 35
2.3.2 Container Runtime 36

2.4 Kubernetes **36**
2.4.1 Cluster 36
2.4.1.1 Control Plane 36
2.4.1.2 Nodes 37
2.4.2 Objects 38
2.4.2.1 Namespace 39
2.4.2.2 Names and UIDs 39
2.4.2.3 Owner and Controller References 39
2.4.2.4 Garbage Collection 39
2.4.3 Workload 39
2.4.3.1 Pod 39
2.4.3.2 Deployment and ReplicaSet 40

2.4.3.3 StatefulSet . . . 41
2.4.3.4 DaemonSet . . . 41
2.4.3.5 Job and CronJob . . . 42
2.4.4 Configuration . . . 43
2.4.4.1 Config Map . . . 43
2.4.4.2 Secret . . . 44
2.4.5 Role-Based Access Control . . . 45
2.4.5.1 Users, Groups and Service Accounts . . . 45
2.4.5.2 Roles and ClusterRoles . . . 45
2.4.5.3 RoleBindings and ClusterRoleBindings . . . 46
2.4.6 Extending Kubernetes . . . 46
2.4.6.1 Custom Resource Definition . . . 47
2.4.6.2 Controller . . . 48
2.4.6.3 Admission Webhook . . . 49
2.4.6.4 Operator Pattern . . . 49
2.4.7 Tools . . . 50
2.4.7.1 Kubectl . . . 50
2.4.7.2 Kubebuilder . . . 50

2.5 Vulnerabilities . . . 50
2.5.1 Common Vulnerabilities and Exposures . . . 50
2.5.2 Common Vulnerability Scoring System . . . 51
2.5.2.1 v3.X . . . 51
2.5.2.2 v4.0 . . . 52

2.6 Vulnerability Scanners . . . 55
2.6.1 Kubernetes-native . . . 56
2.6.2 Limitations . . . 56

2.7 Challenges . . . 56

2.8 Related Work . . . 57
2.8.1 Trivy Operator . . . 57
2.8.2 Trivy Enforcer . . . 57
2.8.3 Kubescape . . . 58
2.8.4 Commercial and Vendor-specific Solutions . . . 58

3 DESCRIPTION 59

3.1 Visibility . . . 59
3.1.1 Definitions . . . 59
3.1.1.1 Workload . . . 59
3.1.1.2 Image and Image Target . . . 60
3.1.1.3 Resource and Resource Target . . . 60
3.1.1.4 Target . . . 61

3.1.1.5 Tracked Vulnerability 61
3.1.1.6 Lifecycle 61
3.1.2 Architecture 62
3.1.2.1 Container Image Scanning 62
3.1.2.2 Lifecycle Tracking 62
3.1.2.3 External Data Store and Dashboard 63
3.1.2.4 Access Control 63
3.1.2.5 Controller 63

3.2 Admission 64
3.2.1 Definitions 64
3.2.1.1 Admission Scan 64
3.2.1.2 Admission Scan Result 64
3.2.1.3 Admission Policy 64
3.2.2 Architecture 64
3.2.2.1 Custom Resource Definitions 65
3.2.2.2 Controller 65
3.2.2.3 Admission Webhook 65

4 IMPLEMENTATION 67

4.1 KCIVM Visibility Operator 67
4.1.1 Introduction 67
4.1.2 Preliminaries 67
4.1.3 Container Image Scanning 67
4.1.3.1 Trivy 67
4.1.3.2 Trivy Operator 69
4.1.3.3 Vulnerability Report CRD 69
4.1.4 Vulnerability Report Controller 72
4.1.5 Ignore DB 73
4.1.6 Exporter 75
4.1.7 External Data Store 76
4.1.8 Dashboard 80

4.2 KCIVM Admission Operator 81
4.2.1 Introduction 81
4.2.2 Preliminaries 81
4.2.3 Container Image Scanning 81
4.2.4 Admission Scan CRD and Controller 82
4.2.5 Admission Scan Result CRD 84
4.2.6 Admission Policy CRD 86
4.2.7 Pod Validating Webhooks 88

5 EVALUATION 91

5.1 KCIVM Visibility Operator **91**

5.2 KCIVM Admission Operator **99**
5.2.1 Performance Impact 104

6 CONCLUSION AND FUTURE WORK **107**

ABBREVIATIONS - ACRONYMS **109**

A EVALUATION **111**

APPENDICES **112**

REFERENCES **113**

LIST OF FIGURES

2.1 The control plane and nodes of a Kubernetes cluster [1]. 37
2.2 CVSS v3.1 Metric Groups [2]. 53
2.3 CVSS v4.0 Metric Groups [3]. 54

4.1 KCIVM Visibility Operator Overview. 68
4.2 Overview of the Tracked Vulnerability. 74
4.3 KCIVM Admission Operator Overview. 82

5.1 Kibana Dashboard (1/2). 93
5.2 Kibana Dashboard (2/2). 94
5.3 Elasticsearch Document (1/2). 95
5.4 Elasticsearch Document (2/2). 96
5.5 Kibana Dashboard, after myapp’s vulnerabilities had been fixed. 97
5.6 Elasticsearch Document, after the vulnerability had been fixed. 98
5.7 Admission Policy Evaluation Mean Times. 105

LIST OF TABLES

2.1 CVSS Qualitative Severity Rating Scale. 51
5.1 Admission Policy Evaluation Mean Times. 105

PREFACE

This thesis was completed as part of the Postgraduate Studies Program ”Computer Systems: Software And Hardware” at the Department of Informatics and Telecommunications of the National and Kapodistrian University of Athens during the fall of 2024.

The work of this thesis was conducted in part during my employment at the Security Department of the Greek National Infrastructures for Research and Technology (GRNET). The idea behind this subject was driven by the requirements of the organization, which manages several Kubernetes clusters hosting high-profile, in-house developed services (e.g. governmental services). The result of this work will help the organization maintain a good security hygiene in its Kubernetes clusters.

1. INTRODUCTION

Kubernetes has become a cornerstone for container orchestration, enabling organizations to efficiently deploy, manage and scale their applications. However, as with any rapidly evolving technology, security risks and vulnerabilities are a constant concern. Effective container image vulnerability management in Kubernetes is crucial to ensure that workloads remain secure while maintaining operational efficiency.

1.1 Problem Description

Container images, which are the smallest executable unit in a Kubernetes cluster, incorporate all the dependencies required by an application in order to provide portability and reproducibility. Although this is an advantage from the deployment standpoint, it expands the attack surface as different versions of the same dependencies affected by different security vulnerabilities may be bundled in different container images. As a result, the following needs arise:

1. Tracking of vulnerable packages in already deployed container images to ensure that they are patched in a timely manner.
2. Blocking deployment of new container images with multiple vulnerable packages as a proactive measure to prevent exploitation.

1.2 Our Contribution

Motivated by the aforementioned challenges, we have designed and implemented two Kubernetes Operators that:

1. Provide visibility to the known vulnerabilities affecting each container image in a Kubernetes cluster and track the lifecycle of each vulnerability between deployments. Each vulnerability is tracked separately by package, container image, workload and namespace.
2. Disallow the deployment of new container images that are affected by known vulnerabilities as motivation for developers to maintain a good security hygiene in their container images.

1.3 Thesis Structure

This rest of this document is structured as follows:

- Chapter 2 lays the groundwork for the subsequent chapters. It provides the fundamental concepts that are essential for the understanding of the problem and the proposed solution.
- Chapter 3 presents a comprehensive description of the design of our solution.
- Chapter 4 demonstrates the technical details of the implementation, including any assumptions made during the development and limitations that had to be dealt with.
- Chapter 5 presents the demo instances of the operators and evaluates their performance impact on a Kubernetes cluster.
- Chapter 6 concludes the thesis and suggests potential future work.

2. BACKGROUND AND RELATED WORK

This chapter outlines the key concepts relevant to the subject of this thesis. It provides definitions of the key components of modern cloud-native applications with regard to their dependencies and the management of their vulnerabilities. Information presented in this chapter is essential for the understanding of the problem and our approach to solve it.

2.1 Application Dependencies

Application dependencies refer to the tools, libraries, external APIs or other software components that an application relies on to function. Dependencies allow developers to build applications more efficiently by leveraging pre-existing components and avoiding the costs of implementing, testing and maintaining the necessary functionalities.

Using dependencies in applications comes with a range of advantages and disadvantages. Dependencies can significantly accelerate development and add robust features, but they also introduce complexity and risks that need to be managed carefully.

Advantages of using dependencies:

- Code reusability: Dependencies allow developers to leverage pre-built libraries, frameworks or packages, reducing the need to write common functionality from scratch.
- Access to specialized functionality: Dependencies often provide highly specialized functionality (e.g., encryption libraries, machine learning frameworks, image processing tools, etc) that would be difficult and time-consuming even for large development teams to implement on from scratch.
- Continuous development: Many dependencies, especially open-source ones, are maintained and continuously improved by large developer communities. This means that applications benefit from ongoing bug fixes, performance improvements and new features.
- Security: Well-maintained dependencies are often more secure and stable than custom-built solutions. Established libraries have undergone extensive testing and security vulnerabilities are regularly patched.

Disadvantages of using dependencies:

- Versioning issues: As dependencies have other dependencies themselves, the more dependencies an application has the higher the chances of conflicts. For example, two libraries requiring a different versions of another library.

- Security vulnerabilities: Dependencies can introduce security risks, especially if they are outdated, unmaintained or come from untrusted sources. If a vulnerability is found in a dependency, the entire application can be compromised.
- Supply chain attacks: Adversaries may inject malicious code into dependencies, leading to a complete comprise of the application that uses them. Dependencies are a great target for such attacks as popular ones are generally utilized by multiple applications.
- Increased attack surface: Each dependency added to an application increases the attack surface, potentially opening new vectors for security vulnerabilities or exploits.
- Maintenance: Dependencies often require frequent updates to fix bugs or security vulnerabilities. While updates can bring improvements, they can also break compatibility or introduce new bugs, creating a maintenance burden.
- Reliance on third-parties: When relying on third-party dependencies without a support contract, developers have less control over the quality, security or future direction of the dependency. If it is abandoned by its maintainers or becomes deprecated, it may leave the application vulnerable or incompatible with future platforms.

In this section we will focus on dependencies that are bundled with or are installed alongside applications and are usually distributed in the form of packages. Packages are usually managed by package managers, which are responsible for installing, upgrading, configuring and managing packages. Package managers utilize centralized repositories as their source of packages. Furthermore, they perform automatic dependency resolution and recursively install all required packages of a package.

2.1.1 Operating System Packages

Operating System (OS) packages are tools or libraries that are installed and managed at the operating system level. These packages are typically but not necessarily compiled code and installed on the system to provide functionality to applications or other packages on the system.

Characteristics of Operating System packages:

- Tied to the Operating System: Operating system packages are platform-specific and the way they are managed, installed and updated depends on the operating system.
- Global Installation: Operating system packages are installed globally, meaning they are accessible to all users and applications on the system.

Advantages of using Operating System packages:

- Centralized management: They provide a unified way to manage all tools and libraries across the system, reducing complexity in managing individual applications.
- System integration: Since these packages are tied to the operating system, they have better integration with system resources such as low-level system components and device drivers.
- Security: Official operating system repositories are usually curated to include trusted libraries and tools.

Disadvantages of using Operating System packages:

- Version compatibility: Because operating system packages are globally installed, only a single version of a package may be installed. Applications must be compatible with the version of the dependent package available on each operating system they intend to support.
- Limited versions: The versions of packages in operating system repositories usually lag behind the latest upstream versions.

Examples of common operating system package managers for Linux-based operating systems are `apt` (Debian, Ubuntu, etc), `dnf` (RHEL, Fedora, etc) and `apk` (Alpine). The respective package formats that the aforementioned package managers handle are `.deb` (Debian, Ubuntu, etc), `.rpm` (RHEL, Fedora, etc) and `.apk` (Alpine).

2.1.2 Language Packages

Language packages are libraries or modules specifically designed to be used within a particular programming language. They are generally distributed through language-specific package managers and their scope is limited to the programming environment rather than the entire operating system.

Characteristics of Language packages:

- Specific to the Language runtime: Language packages are tailored to a particular programming language. They cannot generally be reused by other applications implemented in different languages.
- Virtual environments: Language runtimes often support virtual environments that may be employed to isolate an application and its dependencies from other applications using the same language runtime. This ensures that dependencies do not conflict across different applications. Virtual environments create self-contained directories for each project, storing libraries and packages without affecting global packages (e.g. operating system packages).

Advantages of using Language packages:

- Isolation: With the use of virtual environments, language packages do not interfere with the global packages, eliminating the possibility of version conflicts.
- Fine-Grained version specification: Applications can specify exactly which versions of language packages they require, without making compromises to achieve compatibility with other applications or operating systems.
- Platform independence: Language-specific packages often run the same across different operating systems, as long as a compatible language runtime is present.

Disadvantages of using Language packages:

- Security concerns: It is common practice to utilize community-driven package repositories which may contain unmaintained or malicious packages.

Examples of common language package managers are `pip` for Python, `npm` for JavaScript and Node.js, `composer` for PHP, `maven` and `gradle` for Java, `cargo` for Rust, `gem` for Ruby and `nuget` for .NET.

2.2 Microservices

Microservices architecture is a software design approach in which an application is structured as a collection of small, loosely coupled and independently deployable services. Each service in a microservices architecture is assigned a specific business responsibility and communicates with other services through well-defined APIs. Unlike monolithic architectures where all application components are tightly integrated and deployed as a single unit, microservices allow for independent development and scaling of services.

2.2.1 Decoupled Development and Deployment

Microservices are designed as small, independent services, each responsible for a specific task, such as user authentication, document processing, etc. These services are loosely coupled, meaning changes to one service do not directly affect others, and they can be developed, deployed and scaled independently.

Advantages:

- Independent deployment: Each service can be deployed separately without impacting the entire system, allowing for faster and more agile updates. This also reduces downtime during deployments.

- Independent development: Services can be developed and tested independently, enabling faster development cycles.

Disadvantages:

- Deployment overhead: While services can be deployed independently, managing the deployment of many services can be complex, requiring advanced tools (e.g. Kubernetes, CI/CD pipelines) to streamline the process.

2.2.2 API-Based Communication

Microservices communicate with each other over the network through well-defined APIs. These APIs define how services interact, making it easy for services to communicate across platforms and languages.

Advantages:

- Interoperability: APIs enable microservices to communicate regardless of the underlying technology stack, providing greater flexibility and compatibility.
- Modular development: Each service's functionality is accessed through well-defined interfaces, which promotes clear separation of concerns and modular development.

Disadvantages:

- Network latency: API-based communication introduces additional network latency compared to internal function calls within a monolithic system.
- Communication Failures: Since microservices rely on network communication, they are vulnerable to issues such as timeouts, network outages and service unavailability.

2.2.3 Scalability

Microservices enable independent scaling, meaning each service can be scaled based on its specific performance and load requirements. This allows resources to be allocated more effectively, ensuring critical services receive the resources they need without unnecessarily scaling the entire system.

Advantages:

- Efficient resource use: Services can be scaled horizontally based on demand.

Disadvantages:

- Complexity: Managing the independent scaling of multiple services requires sophisticated orchestration tools (e.g. Kubernetes), which can introduce additional operational overhead.

2.2.4 High Availability

Since services are isolated, failures in one service are less likely to affect the entire application. This isolation provides resilience, allowing the application to remain functional even when one or more services encounter issues.

Advantages:

- Resilience: The system remains operational even if one or more services fail, as other services continue to function independently. This is a significant advantage for large, mission-critical applications where uptime is crucial.

Disadvantages:

- Inter-Service dependencies: Even though services are isolated, there are still dependencies between them. A cascading failure can occur if one critical service causes multiple other services to fail

2.3 Containers

Containers are lightweight, isolated runtime environments that encapsulate an application along with all its necessary dependencies, ensuring consistent behavior across different environments. Containers belong to the category of OS-level virtualization, unlike virtual machines (VMs). Containers share the host operating system's kernel, which makes them faster and more resource-efficient. They achieve isolation using Linux features like namespaces, which isolate the containers processes, network and file system, and cgroups, which control resource usage such as CPU, memory and disk I/O. Security-wise, if a container is compromised, it has limited access to the host or other containers. However, since containers share the host's kernel, any vulnerabilities in the kernel could allow a compromised container to escape to other containers or the host.

Containers and microservices are closely connected because containers are an ideal way to package, deploy and manage microservices.

The Open Container Initiative (OCI) defines standards for container runtimes and images, ensuring compatibility across different tools and platforms [4]. The OCI Specification was based on the Docker Image Specification, the latter is now a superset of the former.

2.3.1 Container Image

A container image is an immutable, read-only file system that defines what is inside the container. It contains everything required for the application to run, such as the application code, runtime, libraries, etc. This allows them to run consistently across different infrastructures as the same container image can be deployed on different environments without modifications.

Images are built in layers, with each change (such as adding dependencies or application code) creating a new layer on top of the previous one. This layering allows for efficient use of disk space, as common layers can be shared across multiple images. When a container is started, the image is used as a template to create the running instance, with a top writable layer added for runtime operations.

A container image is identified its image reference. The format of the image reference is:

```
[REGISTRY_HOST[:PORT]/]REPOSITORY_NAME[:TAG|@DIGEST]
```

2.3.1.1 Registry

Registry is the hostname and optionally the port of the container image registry. If omitted a system-specific default is assumed, usually Docker Hub for historical reasons. Examples: `docker.io`, `registry.example.com:12345`

2.3.1.2 Repository

Repository is the name of the container image in the container image registry. It may include a namespace (e.g. a username or an organization name). This is the only mandatory part of an image reference. Examples: `my-app`, `my-org/my-app`

2.3.1.3 Tag

Tag is a user-defined marker for the image, usually indicating a version or variant. If omitted, `latest` is assumed. Tags are mutable, meaning the image they point to can change over time. Either a tag or a digest may be specified, but not both. Examples: `latest`, `v1.0`

2.3.1.4 Digest

Digest is a repository-specific cryptographic hash that uniquely identifies a specific image, which, unlike tag, is immutable. Either a tag or a digest may be specified, but not both.

Example:
`sha256:6e141e3beb39217d4d4b0d48cefdcc8e54dd915b4b261fb26efe463169f3b8e3`

2.3.2 Container Runtime

A container runtime is software that runs and manages the lifecycle of containers, handling tasks like starting, stopping and isolating containers on a host system. It works at a lower level to execute the container by creating the required processes, applying resource constraints and ensuring the application runs as defined by its container image. Common container runtimes include containerd, CRI-O, Docker and Podman.

2.4 Kubernetes

Kubernetes (often abbreviated as K8s) is an open-source container orchestration system for automating software deployment, scaling and management. It abstracts the complexities of managing infrastructure for modern, cloud-native applications and provides a robust system for orchestrating containers. Furthermore, it is highly modular and follows a declarative model, meaning users define the desired state of the system and Kubernetes works to maintain that state. Kubernetes, as a container orchestration platform, is well-suited for deploying and scaling microservices, which are typically composed of small, loosely coupled and independently deployable components.

Kubernetes is written in Go and is designed to run on a wide variety of platforms, including cloud providers as well as on-premises infrastructure. It was originally developed by Google and is now maintained by the Cloud Native Computing Foundation (CNCF).

In the following sections we will discuss only those Kubernetes characteristics required for this thesis.

2.4.1 Cluster

A Kubernetes *cluster* is the foundational element of the Kubernetes architecture, consisting of a control plane and a set of worker machines that run containerized applications [1]. Every cluster needs at least one worker node.

2.4.1.1 Control Plane

The *control plane* is responsible for maintaining the desired state by making global decisions about the cluster. It is composed of several components:

- kube-apiserver: It exposes the Kubernetes API. The API server is the front end for the Kubernetes control plane.

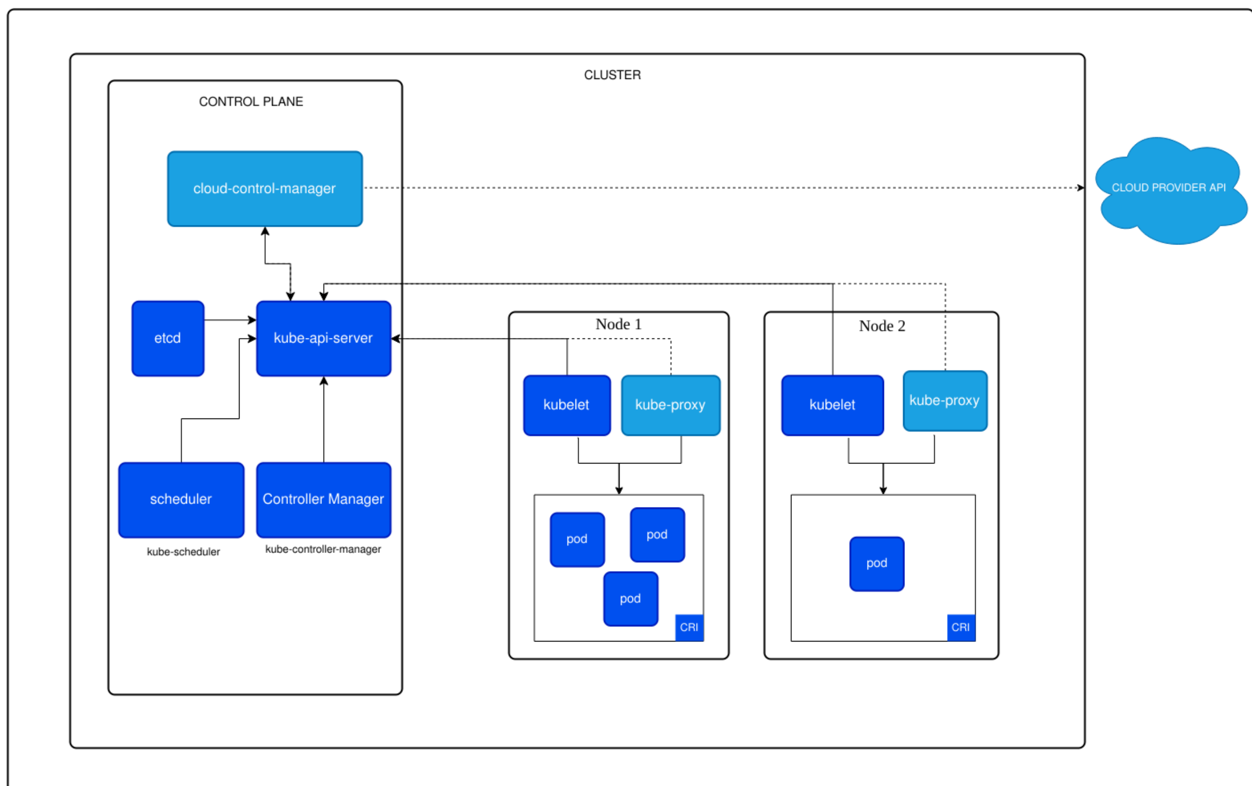


Figure 2.1: The control plane and nodes of a Kubernetes cluster [1].

- etcd: A consistent and highly-available key value store used as a backing store for all cluster data.
- kube-scheduler: Schedules workloads to nodes based on constraints like resource availability, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, etc.
- kube-controller-manager: Runs the controller processes, which ensure that tje desired state of objects matches the actual state of the cluster.
- cloud-controller-manager: Optional component for integration of a Kubernetes cluster to a specific cloud provider.

2.4.1.2 Nodes

Nodes are the worker machines in a Kubernetes cluster. They may be a physical or a virtual machines, depending on the cluster. Each node consists of several components:

- Kubelet: An agent that runs on each node. It oversees the creation and health of containers by communicating with the Control Plane and pulling container specifications.

- Kube-Proxy: A network proxy that maintains network rules on each node, enabling communication between Pods across the cluster.
- Container Runtime: The software that runs the containers, such as containerd or CRI-O. Kubernetes uses the container runtime to create and manage containers.

2.4.2 Objects

In Kubernetes, *objects* represent the persistent entities in the Kubernetes system [5]. Objects describe the desired state of a cluster, such as what containerized applications are running and the resources available to them. Kubernetes ensures that the cluster's current state matches the desired state as specified by objects. Objects are typically defined in YAML configuration files, called manifests. The object spec is what describes the desired state.

Example of a Kubernetes object definition:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: my-container
    image: my-image:latest
```

A Kubernetes object consists of several required fields that define its behavior and configuration. These are:

- API Version: Defines which version of the Kubernetes API is used for the object. Example values: `v1`, `apps/v1`, `networking.k8s.io/v1`, etc.
- Kind: Defines the type of the object. Common kinds: `Pod`, `Deployment`, `Secret`, etc.
- Metadata: Data that helps uniquely identify the object, including a name string, UID and optional namespace.
- Spec: The desired state. The format of this field is different for every Kubernetes object kind.
- Status: An optional sub-resource that provides the actual state of the object as observed by the Kubernetes control plane.

2.4.2.1 Namespace

Namespaces provide a logical isolation mechanism for resources within a single cluster. Namespaces are intended for use in multi-tenant environments. Namespaces provide a scope for names, where the name of an object is unique within a namespace, but not across namespaces.

2.4.2.2 Names and UIDs

Each object in a cluster has a *name* that is unique for that type of resource within a namespace and a *UID* that is unique across the whole cluster.

2.4.2.3 Owner and Controller References

Objects in Kubernetes link to each other through *owner references*. Owner references are metadata of an object which indicate which object owns that object. A special owner reference is the *controller reference*. An object may be owned by multiple objects but controlled only by one. Cross-namespace owner references are disallowed by design.

2.4.2.4 Garbage Collection

Garbage Collection is a mechanism in Kubernetes that automatically deletes objects when their owner object is deleted, based on the owner references. This gives the control plane the opportunity to clean up related resources before deleting an object and helps prevent orphaned objects. This process is called cascading deletion. There are two types of cascading deletion, foreground cascading deletion where the dependents are deleted before the owner object and background cascading deletion where the owner is deleted immediately and dependents at a later time.

2.4.3 Workload

A *workload* is a resource that represents an application running on Kubernetes [6]. Ultimately, workloads define how containers should be run.

2.4.3.1 Pod

A *Pod* is the most basic and smallest unit in Kubernetes, representing a single instance of a running process in the cluster. A Pod encapsulates one or more containers that share the same network namespace and storage. Typically, a Pod runs a single container, but it can also run multiple containers that work closely together. In the latter case, the extra

containers may be *init containers* that are run and must complete successfully before the main container is run (e.g. to perform application specific initialization) or *sidecar containers* that run along the main container to perform complementary operations (e.g. log collection).

Pods are designed to be short-lived. If a Pod dies, it will not be restarted on its own. Kubernetes relies on higher-level controllers like Deployments or ReplicaSets (that will be discussed in the following sections) to manage Pod lifecycles.

Example of a Pod definition:

```
apiVersion: v1
kind: Pod
metadata:
  namespace: my-namespace
  name: my-pod
spec:
  containers:
  - name: my-container
    image: my-app:latest
```

2.4.3.2 Deployment and ReplicaSet

A *ReplicaSet* is a resource which ensures that a specified number of Pod replicas are running at any given time. It allows for declarative updates to applications (e.g. rolling updates, rollbacks), providing a powerful way to manage stateless applications and maintain their availability even during changes. A *Deployment* is a higher-level resource to manage ReplicaSets.

Example of a Deployment definition:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  namespace: my-namespace
  name: my-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
```

```
    containers:
    - name: my-container
      image: my-app:latest
```

Example of a ReplicaSet definition (e.g. auto-generated by the Deployment above):

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  namespace: my-namespace
  name: my-deployment-abc123
spec:
  replicas: 2
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
      - name: my-container
        image: my-app:latest
```

2.4.3.3 StatefulSet

A *StatefulSet* works similar to a ReplicaSet, but is suited for stateful applications that require persistent storage or stable network addresses.

2.4.3.4 DaemonSet

A *DaemonSet* ensures that a copy of a Pod is running on all nodes in a Kubernetes cluster. Its purpose is analogous to a daemon on a traditional server, e.g. to perform operations fundamental to the operation of your cluster.

Example of a DaemonSet definition:

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
namespace: my-namespace
  name: my-daemonset
spec:
```

```
  selector:
    matchLabels:
      name: my-daemonset
  template:
    metadata:
      labels:
        name: my-daemonset
    spec:
      containers:
      - name: my-container
        image: my-daemon:latest
```

2.4.3.5 Job and CronJob

A *Job* creates one or more Pods and ensures that a specified number of them successfully complete. Unlike Deployments or ReplicaSets (which ensure Pods are continuously running), Jobs are designed for short-lived, one-time tasks. Once the Pods complete their tasks, the Job ends. If a Pod fails, the Job can retry by creating a new Pod until the task is successfully completed.

Example of a Job definition:

```
apiVersion: batch/v1
kind: Job
metadata:
  namespace: my-namespace
  name: my-job
spec:
  template:
    spec:
      containers:
      - name: my-container
        image: my-app:latest
        command: [...]
      restartPolicy: OnFailure
  completions: 1
```

A *CronJob* is used to run Jobs on a scheduled basis. The schedule is defined in cron format. The CronJob ensures that Jobs are created at the specified times.

Example of a CronJob definition:

```
apiVersion: batch/v1
kind: CronJob
metadata:
```

```
  namespace: my-namespace
  name: my-cronjob
spec:
  schedule: ”*/5 * * * *” # Runs every 5 minutes.
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: my-container
            image: my-app:latest
            command: [...]
          restartPolicy: OnFailure
```

2.4.4 Configuration

Kubernetes provides mechanisms to manage configuration separately from the application code [7]. This separation allows configuration changes without rebuilding container images. Configuration can be injected to Pods as files, command line arguments or environment variables.

2.4.4.1 Config Map

A *ConfigMap* is a resource that allows to store non-confidential data in key-value pairs.

Example of a ConfigMap definition:

```
apiVersion: v1
kind: ConfigMap
metadata:
  namespace: my-namespace
  name: my-config
data:
  service-url: https://example.com/
  log-level: INFO
```

Example of how it can be used as environment variables:

```
apiVersion: v1
kind: Pod
metadata:
  namespace: my-namespace
  name: my-pod
spec:
```

```
containers:
- name: my-container
  image: my-image:latest
  env:
  - name: LOG_LEVEL
    valueFrom:
      configMapKeyRef:
        name: my-config
        key: log-level
```

2.4.4.2 Secret

A *Secret* is a resource used to store sensitive information, such as passwords, API keys, SSH keys, TLS certificates and tokens. Unlike ConfigMaps, Secrets are treated as sensitive data by Kubernetes, however they are stored unencrypted in the underlying data store (etcd). Values are stored Base64 encoded.

There are several Secret types:

- Opaque: Used to store generic key-value pairs (e.g. passwords, tokens).
- Docker Config: Used to store credentials for pulling images from private Docker registries.
- TLS: Stores a TLS certificate and private key.
- Service Account Token: Provides an authentication token for communicating with the Kubernetes API server.

Example of a Secret definition:

```
apiVersion: v1
kind: Secret
metadata:
namespace: my-namespace
  name: my-secret
type: Opaque
data:
  api-key: dmVyeSBzZWNyZXQgYXBpIGtleQo=
```

Example of how it can be used as environment variables:

```
apiVersion: v1
kind: Pod
metadata:
```

```
  namespace: my-namespace
  name: my-pod
spec:
  containers:
  - name: my-container
    image: my-image:latest
    env:
    - name: API_KEY
      valueFrom:
        secretKeyRef:
          name: my-secret
          key: api-key
```

2.4.5 Role-Based Access Control

Role-Based Access Control (RBAC) is a feature in Kubernetes that enables administrators to define fine-grained access controls for users and service accounts. The following sections briefly describe the key concepts around RBAC.

2.4.5.1 Users, Groups and Service Accounts

Users, Groups and Service Accounts are the entities to which permissions are granted. Like everything else in Kubernetes, they are represented as resources. An example service account manifest follows:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  namespace: my-namespace
  name: my-service-account
```

2.4.5.2 Roles and ClusterRoles

A Role defines permissions within a specific namespace. It specifies what actions (verbs) can be performed on which resources (e.g. Pod, Deployment, ConfigMap, etc) within that namespace. Roles are created as a Kubernetes resources.

A ClusterRole is similar to a Role but applies cluster-wide. It can define permissions across all namespaces or for cluster-scoped resources.

The following example role allows to `GET`, `LIST` and `WATCH` the built-in Deployment resources in the specified namespace:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: my-namespace
  name: my-role
rules:
  - apiGroups: ["apps/v1"]
    resources: ["deployments"]
    verbs: ["get", "list", "watch"]
```

2.4.5.3 RoleBindings and ClusterRoleBindings

A *RoleBinding* grants the permissions defined in a Role to a specific user or group within a particular namespace. RoleBindings can reference either a Role or a ClusterRole. Similarly, a *ClusterRoleBinding* grants the permissions defined in a ClusterRole to a user, group or service account across all namespaces.

The following example binds the role `my-role` of the previous section to user `user`:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  namespace: my-namespace
  name: my-role-binding
subjects:
  - apiGroup: rbac.authorization.k8s.io
    kind: User
    name: user
roleRef:
apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: my-role
```

2.4.6 Extending Kubernetes

Kubernetes is highly extensible by design, allowing to extend its core functionality without modifying the underlying code. There are several key mechanisms for extending Kubernetes, such as Custom Resource Definitions (CRDs), Controllers, Admission Controllers and the Operator pattern. These components allow Kubernetes to manage new kinds of resources or perform custom tasks.

The Kubernetes core as well as other official components are implemented in Go.

2.4.6.1 Custom Resource Definition

A *Custom Resource Definition (CRD)* is an extension of the Kubernetes API. Custom resources function similarly to native Kubernetes resources (e.g Pods). Several built-in Kubernetes resources (e.g. Deployments and Jobs) are implemented as custom resources.

Custom Resource Definitions, like native resources, are a declarative API where the desired state of the cluster is described using a YAML manifest. They are identified by their API Version and Kind. An example Custom Resource Definition follows:

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: my-crds.my-group.example.com
spec:
  group: my-group.example.com
  names:
    kind: MyCRD
    listKind: MyCRDList
    plural: my-crds
    singular: my-crds
  scope: Namespaced
  versions:
  - name: v1alpha1
    schema:
      openAPIV3Schema:
        properties:
          apiVersion:
            type: string
          kind:
            type: string
          metadata:
            type: object
          spec:
            properties:
              my-field:
                type: string
            required:
            - field
            type: object
        type: object
    served: true
    storage: true
```

This Custom Resource Definition defines a new resource called `MyCRD` at version `v1alpha1`

under the API group `my-group.example.com`. `MyCRD` has a single field named `my-field`. An example instance of the Custom Resource Definition above follows:

```
apiVersion: my-group.example.com/v1alpha1
kind: MyCRD
metadata:
  name: my-crd
spec:
  my-field: my-value
```

2.4.6.2 Controller

A *Controller* in Kubernetes is a control loop that watches the state of Kubernetes resources and takes action to ensure the system's current state matches the desired state. Controllers use a declarative model where users specify the desired state of resources and the controller continuously works to make the current state converge to that desired state.

Informers: Kubernetes controllers use informers to monitor resource changes in the API server. Informers are responsible for watching Kubernetes objects (like Pods, Deployments, Jobs, etc) and generating events (Create, Update, Delete) whenever there is a change to a resource. Informers also maintain local caches of these objects for efficiency.

A Kubernetes controller may watch only a single type of resource, but may triggered by changes to related resources, too. A Kubernetes resource may be watched by multiple Kubernetes controllers.

Queue: Controllers use a queue to manage the incoming events generated by the informers. The events are queued and the controller processes each event one by one. This allows controllers to process changes asynchronously, ensuring that multiple updates can be handled efficiently.

Reconciliation Loop: At the core of a controller's logic is the reconciliation loop. This loop is responsible for aligning the actual state with the desired state. The reconciliation logic is invoked based on an event that corresponds to a resource being created, updated or deleted:

- Create Event: The controller is called to reconcile when a new object is created. It observes the new desired state and ensures the actual state is updated accordingly. For example, in a ReplicaSet controller, the reconciliation logic ensures the desired number of Pods are created.

- Update Event: When an object is updated, the controller observes the new desired state and reconciles the actual state accordingly. For example, if the number of replicas in a Deployment changes, the controller ensures the correct number of Pods are created or deleted.
- Delete Event: After an object is deleted, the controller is triggered, e.g. to perform cleanup or other finalizing tasks. If the object has a registered Finalizer, deletion becomes a two step process: First, it is marked for deletion which is observed by the controller as an Update event and after the Finalizer has been removed it is finally deleted.

The Finalizer mechanism is employed to ensure that a controller can be triggered before a watched object is deleted, ensuring that the controller cas still access its data. While an object has at least one Finalizer registered, it remains in a ”terminating” state until all Finalizers have been removed. Responsible for removing a Finalizer is the controller that registered it. If a Finalizer is not removed properly (e.g. due to controller crashes or bugs), it can prevent objects from being fully deleted.

Leader Election: To ensure high availability and fault tolerance, multiple instances of a Kubernetes controller may live in a Kubernetes cluster. Leader election is crucial that only one instance is performing reconciliation operations at a time.

2.4.6.3 Admission Webhook

An *Admission Webhook* is a Kubernetes component that intercepts create, update and delete requests to the Kubernetes API server prior to persistence of the respective object, but after the request is authenticated and authorized. Kubernetes supports two types of admission webhooks, which are always executed in that order:

1. Mutating Admission Webhook: Modifies requests before the changes are persisted.
2. Validating Admission Webhook: Validates requests and decide whether they are allowed or denied.

Admission Webhooks provide significant flexibility and customization in enforcing cluster-level governance, security policies or injecting necessary data.

2.4.6.4 Operator Pattern

The combination of custom resources, controllers and admission webhooks is referred to as a Kubernetes *Operator*. This concept allows to extend the cluster’s behavior by linking custom controllers to one or more custom resources. Operators are clients of the Kubernetes API that act as controllers for custom resources.

2.4.7 Tools

2.4.7.1 Kubectl

Kubectl is a command line tool for communicating with a Kubernetes cluster's control plane, using the Kubernetes API. Among others, it can manage (retrieve, create, update, delete) all kinds of Kubernetes resources and get or stream container logs, making it an essential tool for managing a Kubernetes cluster.

For example, the following command invocation retrieves all pods in the specified namespace:

```
$ kubectl -n my-namespace get pods
```

The following command invocation applies (creates or updates) the resources specified in the supplied manifest:

```
$ kubectl apply -f deployment.yaml
```

2.4.7.2 Kubebuilder

Kubebuilder is an SDK for building Kubernetes APIs using CRDs in Go. It simplifies the process of creating, managing and deploying Kubernetes operators and controllers. It can generate boilerplate code, scaffolding the project structure necessary for creating controllers and CRDs. This significantly reduces the time and effort required to set up a new operator.

2.5 Vulnerabilities

A security vulnerability is a flaw, weakness or exposure that can be exploited by an attacker to compromise the confidentiality, integrity or availability (CIA) of a system. Vulnerabilities can arise from coding errors, design flaws or misconfigurations, potentially leading to a range of security threats such as unauthorized access, data theft or service disruption.

2.5.1 Common Vulnerabilities and Exposures

Common Vulnerabilities and Exposures (CVE) is a program that provides a standardized method for identifying publicly known security vulnerabilities [8]. The CVE List is maintained by the MITRE Corporation and allows organizations to better communicate and share information about vulnerabilities.

A CVE record is a vulnerability tracked by the CVE program and is identified by CVE ID [9]. The CVE ID consists of three parts: The `CVE` prefix, the year when the CVE ID was

assigned and a sequence number that distinguishes the specific vulnerability from others in the same year. An example CVE ID is `CVE-2024-12345`.

Apart from the CVE ID, a CVE record includes a vulnerability description, a list of references (e.g. URLS of advisories or patches) and the CVE Numbering Authority (CNA) that assigned the CVE ID.

2.5.2 Common Vulnerability Scoring System

The *Common Vulnerability Scoring System (CVSS)* is an open industry standard used to assess and quantify the severity of security vulnerabilities [10]. It provides a numeric score (ranging from 0.0 to 10.0), which represents the overall risk posed by the vulnerability, which is commonly used to prioritize remediation efforts based on the risk level. All CVSS scores are mapped to the qualitative ratings shown in table 2.1.

Table 2.1: CVSS Qualitative Severity Rating Scale.

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

CVSS has evolved over time and five versions have been released, specifically v1, v2, v3.0, v3.1 and v4. v4 is the latest version, but v3.X is still widely used today.

2.5.2.1 v3.X

CVSS v3.x, introduced in 2015, contains three sets of group metrics: Base, Temporal and Environmental [2]. The overall score is derived primarily from the Base metrics, which measure the intrinsic qualities of a vulnerability, while the Temporal and Environmental metrics allow adjustments based on factors like exploit availability or specific conditions in an organization's environment.

The group metrics, also presented in figure 2.2, are described below:

- Base Metrics:
  - Attack Vector (AV): Describes how the vulnerability can be exploited. Possible values are: Network (N), Adjacent (A), Local (L), Physical (P)
  - Attack Complexity (AC): Reflects how complex it is to exploit the vulnerability. Possible values are: Low (L), High (L)

  - Privileges Required (PR): Describes the level of privileges an attacker must have to exploit the vulnerability. Possible values are: None (N), Low (L), High (L)
  - User Interaction (UI): Indicates whether exploiting the vulnerability requires interaction from the victim. Possible values are: None (N), Required (R)
  - Scope (S): Refers to whether a successful exploit affects resources beyond the vulnerable component. Possible values are Unchanged (U), Changed (C)
  - Confidentiality (C): Measures the impact to the confidentiality of data. Possible values are: None (N), Low (L), High (L)
  - Integrity (I): Measures the impact to the integrity of data. Possible values are: None (N), Low (L), High (L)
  - Availability (A): Measures the impact to the availability of the affected component. Possible values are: None (N), Low (L), High (L)
- Temporal Metrics:
  - Exploit Code Maturity (E): Indicates the availability of exploit code. Possible values are: Not Defined (X), Unproven (U), Proof-of-Concept (P), Functional (F), High (H)
  - Remediation Level (RL): Indicates the availability of a patch or workaround. Possible values are: Not Defined (X), Official Fix (O), Temporary Fix (T), Workaround (W), Unavailable (U)
  - Report Confidence (RC): Level of confidence in the existence of the vulnerability. Possible values are: Not Defined (X), Unknown (U), Reasonable (R), Confirmed (C)
- Environmental Metrics:
  - Requirement Metrics: These metrics allow the Confidentiality, Integrity and Availability scores to be modified depending on the importance of the affected resource is the assessed scope (e.g. organization).
  - Modified Metrics: These metrics allow all metrics to be modified to reflect specific security needs.

2.5.2.2 v4.0

CVSS v4.0, released in 2023, introduced new group metrics and more granular scoring mechanisms to overcome limitations criticized in the previous versions [3]. The overall score is again derived primarily from the Base metrics, while the Threat Environmental and Supplemental metrics allow adjustments based on factors like exploit availability or specific conditions in an organization's environment.

The group metrics, also presented in figure 2.3, are described below:

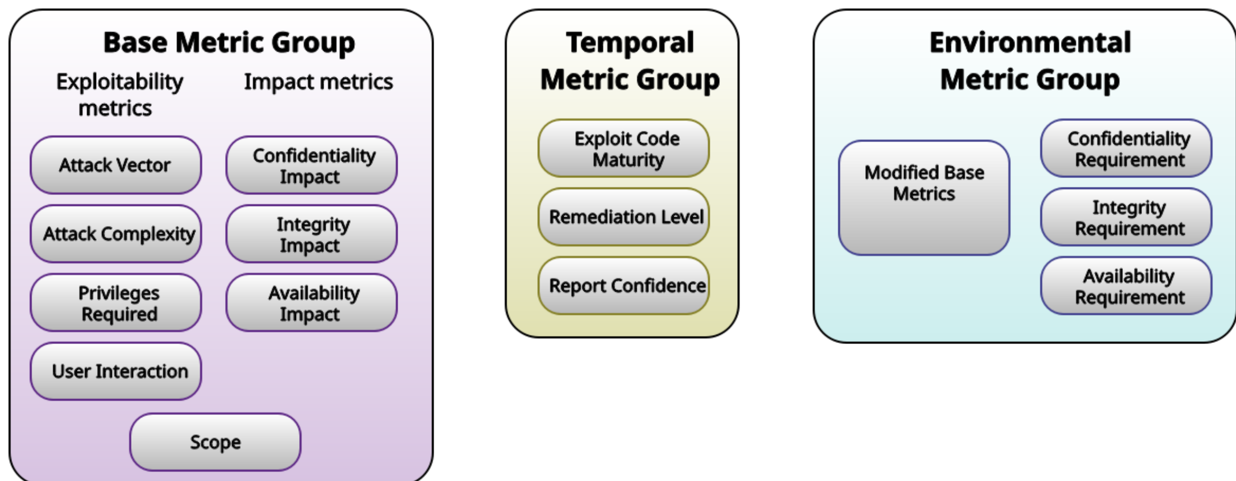


Figure 2.2: CVSS v3.1 Metric Groups [2].

- Base Metrics:
  - Exploitability Metrics:
    * Attack Vector (AV): Describes how the vulnerability can be exploited. Possible values are: Network (N), Adjacent (A), Local (L), Physical (P)
    * Attack Complexity (AC): Reflects how complex it is to exploit the vulnerability. Possible values are: Low (L), High (L)
    * Attack Requirements (AC): Reflects specific prerequisites required to exploit the vulnerability. Possible values are: None (N), Present (P)
    * Privileges Required (PR): Describes the level of privileges an attacker must have to exploit the vulnerability. Possible values are: None (N), Low (L), High (L)
    * User Interaction (UI): Indicates whether exploiting the vulnerability requires interaction from the victim. Possible values are: None (N), Required (R)
  - Vulnerable and Subsequent System Metrics:
    * Confidentiality (C): Measures the impact to the confidentiality of data. Possible values are: None (N), Low (L), High (L)
    * Integrity (I): Measures the impact to the integrity of data. Possible values are: None (N), Low (L), High (L)
    * Availability (A): Measures the impact to the availability of the affected component. Possible values are: None (N), Low (L), High (L)
- Threat Metrics:
  - Exploit Code Maturity (E): Indicates the likelihood of the vulnerability being exploited in the wild. Possible values are: Attacked (S), POC (P), Unreported (F)

- Environmental Metrics:
  - Requirement Metrics: These metrics allow the Confidentiality, Integrity and Availability scores to be modified depending on the importance of the affected resource is the assessed scope (e.g. organization).
  - Modified Metrics: These metrics allow all metrics to be modified to reflect specific security needs.
- Supplemental Metrics:
  - Safety (S): Measures the potential for the vulnerability to cause physical harm or safety risks, especially relevant in sectors like healthcare, automotive, industrial control systems and critical infrastructure. Possible values are: Not Defined (X), Negligible (N), Present (P)
  - Automatable (AU): Evaluates whether the exploitation of the vulnerability can be automated. Possible values are: Not Defined (X), No (N), Yes (Y)
  - Recovery (R): Measures how difficult it is to recover from a successful exploitation of the vulnerability. Possible values are: Not Defined (X), Automatic (A), User (U), Irrecoverable (I)
  - Value Density (V): Assesses the amount and sensitivity of the data or assets at risk due to the vulnerability. Possible values are: Not Defined (X), Diffuse (D), Concentrated (C)
  - Vulnerability Response Effort (RE): Describes the level of effort required to respond to and mitigate the vulnerability. Possible values are: Not Defined (X), Low (L), Moderate (M), High (H)
  - Provider Urgency (U): Describes the level of urgency as specified by a provider along the software supply chain. Possible values are: Not Defined (X), Clear (C), Green (G), Amber (A), Red (R)

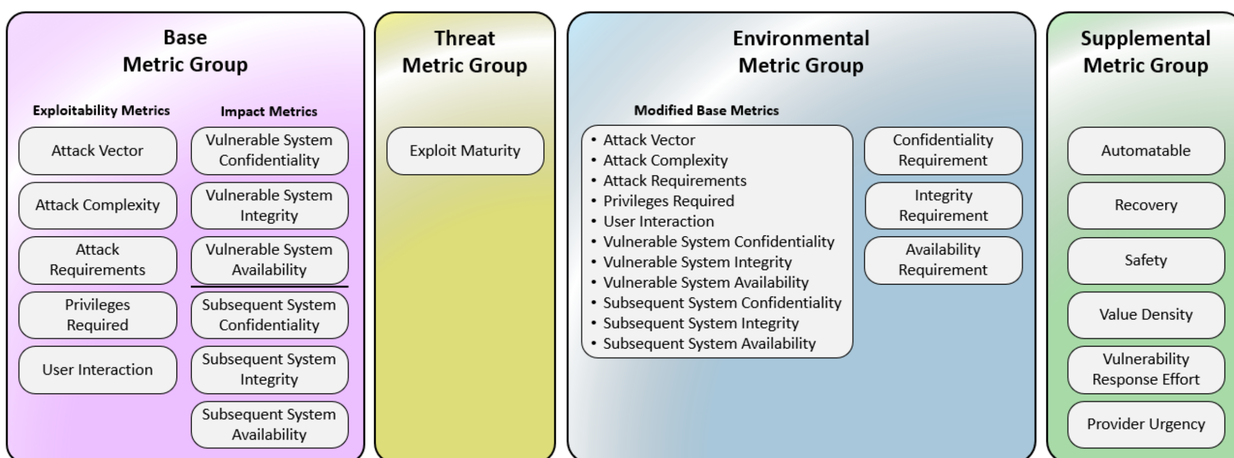


Figure 2.3: CVSS v4.0 Metric Groups [3].

2.6 Vulnerability Scanners

A *Vulnerability Scanner* for dependencies is a specialized tool designed to analyze dependencies of an application, with the goal of identifying known vulnerabilities. As discussed in section 2.1 Application Dependencies, dependencies are usually distributed in the form of OS or language packages. The job of a vulnerability scanner is to identify the exact versions of the dependencies of an application, either specified in a package manager-specific dependency list or from the installed packages, and cross-reference them against a vulnerability database to determine if any of them are susceptible to known security vulnerabilities.

Thinking of container images as ”packages” that bundle an application and all its dependencies, a vulnerability scanner may also scan whole container images for known vulnerabilities. In this scenario, the scanning scope includes both OS packages and language packages. The rest of this section will focus on vulnerability scanning of container images.

A container image vulnerability scan starts with package identification, where the vulnerability scanner scans the image for various sources of packages. Such sources are OS package managers (unless the container image uses a ”distroless” base image) and application projects with language-specific package managers. These may reside anywhere inside a container image and as such the vulnerability scanner needs to traverse the whole file system hierarchy in order to identify them.

After the vulnerability scanner has collected all packages and their versions it looks up its configured vulnerability sources. These may include public (e.g. vendor-, distro- or package-specific) or scanner-specific vulnerability databases. Packages are matched by their name and version.

Several tools that can perform container image vulnerability scanning exist, both open-source and commercial. The following lists present some well known ones (in no particular order).

Open-source:

- Aqua Security Trivy[1]
- Quay Clair[2]
- Anchore Grype[3]

Commercial:

- Snyk Container[4]

[1] https://aquasecurity.github.io/trivy/
[2] https://github.com/quay/clair
[3] https://github.com/anchore/grype
[4] https://docs.snyk.io/scan-using-snyk/snyk-container

- Qualys Container Security[5]
- Synopsys Black Duck[6]

2.6.1 Kubernetes-native

Kubernetes-native vulnerability scanners also exist. They usually leverage an existing vulnerability scanner to perform the actual vulnerability scan, but integrate them in Kubernetes using controllers and CRDs. Two well-known examples are Trivy Operator[7] which uses Trivy and Kubescape[8] which uses Grype.

2.6.2 Limitations

Although Vulnerability Scanners have the important advantage of being application agnostic, they also come with several limitations.

First of all, a vulnerability scanner can identify packages only of known (to the vulnerability scanner) package managers. Dependencies managed by unsupported package managers or embedded in the application itself (e.g. by copying their source code) cannot be handled by vulnerability scanners.

Secondly, vulnerabilities in packages identified by vulnerability scanners my not be exploitable in the specific context of an application, as the vulnerable functionality of a dependency may not be used by the application at all or the conditions required for the vulnerability to be exploitable may not be met at runtime.

Finally, a vulnerability scanner can identify only packages that have been installed at the build time of a container image. Although an anti-pattern, a container may install extra packages at runtime which will go undetected by the vulnerability scanner.

2.7 Challenges

In the section, we briefly discuss the challenges that our solution aims to overcome:

- View to the current state of vulnerable container images in a Kubernetes cluster.
- Tracking of the lifecycle of each vulnerability, separately for every image and workload in a Kubernetes namespace.

[5] https://www.qualys.com/apps/container-security/

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

[7] https://aquasecurity.github.io/trivy-operator/

[8] https://kubescape.io/

- Enforcement of policies at the deployment time of container images.
- Configurable policies for what is considered to be a vulnerable container image.
- Platform-agnostic solution (e.g. do not depend on a specific container image registry for the vulnerability scans).
- Implemented in a Kubernetes-native way.

These challenge had a strong influence on the design and implementation decisions that we discuss in the following chapters.

2.8 Related Work

There are several projects with similar of partially overlapping goals to that of this thesis, i.e. vulnerability management in Kubernetes-native way. As with our solution, most build upon existing container image vulnerability scanners, like those described in section 2.6.

2.8.1 Trivy Operator

The Trivy Operator leverages Trivy to continuously scan a Kubernetes cluster for security issues, including container image vulnerabilities [11]. The vulnerability scans are summarized in security reports as Kubernetes CRDs and can be accessed through the Kubernetes API. Furthermore, it offers a Grafana Dashboard that exposes metrics from the reports.

In comparison to the visibility-part of our solution, it only provides a view to the current state of the cluster without vulnerability tracking or lifecycle management.

2.8.2 Trivy Enforcer

Trivy Enforcer[9] is an experimental Kubernetes Operator that aims to protect from the deployment of unsafe container images. As of today, the project is no longer maintained.

It has a very similar goal to the admission-part of our solution, but using a different approach. Specifically, it depends on an instance of the Harbor container image registry to perform the vulnerability scans which queries when evaluating the admission policies.

[9] https://github.com/aquasecurity/trivy-enforcer

2.8.3 Kubescape

Kubescape is a broader security platform that also offers a Kubernetes operator to scan container images and report vulnerabilities in a Kubernetes cluster. It leverages Grype for the vulnerability scans.

In comparison to the visibility-part of our solution, it only provides a view to the current state of the cluster without vulnerability tracking or lifecycle management.

2.8.4 Commercial and Vendor-specific Solutions

Many major cloud providers that offer managed Kubernetes clusters as-a-service also offer proprietary vulnerability management and policy enforcement solutions. Furthermore, commercial offerings that can be installed in self-managed Kubernetes clusters also exist. However, due to the closed nature of all the aforementioned offerings, it is impossible to compare them to our solution.

3. DESCRIPTION

In this chapter we will provide an in-depth description of our solution. We will begin by introducing the key components and constructs involved in our implementation and we will proceed by describing how they interact with each other to achieve our goals.

3.1 Visibility

The goal of the *Visibility* component of our solution is to provide visibility to the state of known vulnerabilities that affect container images of existing workloads in a Kubernetes cluster. This does not only refer to the state of vulnerabilities at a specific point in time (which as stated in section 2.8 is already offered by other projects), but includes tracking of the lifecycle of each vulnerability between different deployments of the same workload. The lifecycle management involves tracking of the vulnerability from the moment it is first identified till the moment it is resolved, including potential regressions.

3.1.1 Definitions

Before we proceed with the architecture, it is important to provide definitions for the terms that will be used later on in the description of the solution. It should be noted that some terms overlap with similar terms in different contexts, but what is defined in the following sections refers to our implementation.

3.1.1.1 Workload

In Kubernetes, a Workload is a running application, as discussed earlier in section 2.4.3. Built-in Workloads include the Deployment, ReplicaSet, Job, etc, but custom Workloads defined using CRDs may also exist. The goal of a Workload is to orchestrate the containers that comprise the application.

A container does not exist in its own but is always part of a Pod, which is the smallest deployable compute object in Kubernetes. A Pod may exist autonomously, but it is usually controlled by another resource.

The controlling relation between two resources is denoted using a controller reference, which is a special owner reference, as discussed in section 2.4.2.3. A resource may directly control a Pod or it may control another resource which controls a Pod, thus forming a chain of controlled resources.

We define as *Workload* the root resource in a chain of controlled resources that control a Pod. We provide examples of Workloads using Kubernetes built-in resources to clarify the definition:

- Job: A Job directly controls a Pod. If that Job is not controlled by another resource, then it is a Workload.
- CronJob: A CronJob controls a Job which controls a Pod. If that CronJob is not controlled by another resource, then it is a Workload. In this example, the Job is not a Workload as it is controlled by the Cronjob.
- ReplicaSet: A ReplicaSet directly controls a set of Pods. If that ReplicaSet is not controlled by another resource, then it is a Workload.
- Deployment: A Deployment controls a ReplicaSet which controls a set of Pods. If that Deployment is not controlled by another resource, then it is a Workload. In this example, the ReplicaSet is not a Workload as it is controlled by the Deployment.

Since a Workload is a Kubernetes object, it is identified by its API Version, Kind and Name in a Namespace and additionally by its Namespace's Name in a Kubernetes cluster.

3.1.1.2 Image and Image Target

As described in section 2.3.1, a container image reference consists of three parts: registry, repository and tag or digest. Image and Image Target decompose the reference to two parts:

- An *Image* consists of and is identified by the registry and the repository of a container image registry. It represents the set of different versions of a specific container image.
- An *Image Target* consists of and is identified by the tag or the digest of a container image reference. It represents a specific version of a specific container image.

Combining the Image and the Image Target gives us the container image reference, which, as discussed earlier, identifies a container image.

3.1.1.3 Resource and Resource Target

An installed package is identified by four values: class, type, name and version. *Resource* and *Resource Target* group these values into two parts:

- A *Resource* consists of and is identified by the the package's class (e.g. OS or Language package), type in the class (e.g. Debian or Python) and name. It represents the set of different versions of a specific package
- A *Resource Target* consists of and is identified by the package's detected version. It represents a specific version of a specific package.

Combining the Resource and the Resource Target gives us an installed package in a container image. Resources in this context should not be confused with Kubernetes resources.

3.1.1.4 Target

A *Target* is an Image Target - Resource Target pair, i.e. a specific version of a package in a specific container image.

3.1.1.5 Tracked Vulnerability

As mentioned earlier, our goal is to track the lifecycle of each vulnerability. Tracking a lifecycle just per vulnerability ID does not provide fine enough granularity, as a single vulnerability may affect multiple resources. Furthermore, a vulnerable resource may be present in multiple container images, a container image may be deployed by multiple workloads and similar workloads may exist in multiple namespaces and clusters.

Our solution tracks each vulnerability separately for each ”cluster, namespace, workload, image and resource” tuple it affects. This tuple forms the package ID, which is identified by the entities that comprise it. Specifically:

- Cluster: An arbitrary *Name* for the cluster.
- Namespace: The *Name* of the namespace.
- Workload: The *API Version*, *Kind* and *Name* of the workload.
- Image: The *Registry* and *Repository* of the image.
- Resource: The *Class*, *Type* in class and *Name* of the resource.

A *Tracked Vulnerability* tracks a package ID - vulnerability ID pair. This allows us to track separate lifecycles for the same vulnerable packages in different container images, for the same container image in different workloads and for similarly-named workloads in different namespaces and clusters. Furthermore, it can be deduced from the definitions so far that a tracked vulnerability tracks the set of Targets that correspond to a specific package ID - vulnerability ID pair.

3.1.1.6 Lifecycle

A *Lifecycle* is a continuous period of time in which a tracked vulnerability had been or has been open. A Tracked Vulnerability is considered to be open at a specific point in time if it tracks one or more Targets and closed if it does not track any targets.

3.1.2 Architecture

The architecture of this component revolves around the following work flow: Container images of already-deployed workloads are scanned to identify their installed packages, the versions of the installed dependencies are checked against vulnerability databases to detect which are affected by known vulnerabilities, the vulnerabilities along with their tracking information are persisted to a data store and finally a dashboard provides access to these data (e.g. metrics, detailed information, etc), ultimately providing visibility to the state of vulnerabilities in the Kubernetes cluster.

This component is designed in a Kubernetes-native fashion, utilizing Controllers (and CRDs, indirectly).

3.1.2.1 Container Image Scanning

Our approach of identifying the versions of the application's dependencies is project-agnostic. Instead of relying on a project-specific mechanism to retrieve these versions, it relies on scanning the resulting container image to identify the versions of installed packages. This comes with several benefits as well as drawbacks that affect the rest of our solution's design.

First of all, scanning the resulting container image allows to retrieve with certainty the versions of the packages that will be used by this build of the application. On the other hand, retrieving the versions from the application project would require resolving the specified packages and their dependencies (unless the project utilizes package locking), which might not resolve to the same versions as at the time of building.

Moreover, being project-agnostic has the advantage of not requiring support from the project itself, making the solution universal. On the other hand, this approach has the disadvantage that it will only identify packages installed by known package managers. Even if we assume that a project does not utilize an exotic package manager for its dependencies, it may still manually include dependencies (e.g. by copying their source code) which will go undetected by the container image scanning.

Finally, an application may also depend on libraries or tools provided by the underlying OS. Since we are scanning container images, we can also take OS packages in the container image into account (unless the container image is using a ”distroless” base). This would not have been possible if the dependency identification took place on the project level, as the runtime OS is not available at that point.

3.1.2.2 Lifecycle Tracking

An important contribution of our solution is managing the lifecycle of a Tracked Vulnerability. A Tracked Vulnerability may have one or more lifecycles, which represent the different periods of time in which it had been or has been open. It is obvious that only the latest

lifecycle may be open and that all previous ones must be closed. This mechanism is used to track regressed vulnerabilities in our tracking unit. The current state (open or closed) of a Tracked Vulnerability is determined only by the latest lifecycle.

Each lifecycle stores the timestamps of when it was opened and closed. This allows to extract metrics such as the number of hours that a Tracked Vulnerability has been open since the last regression or in total by summing the hours of all lifecycles.

3.1.2.3 External Data Store and Dashboard

Our solution relies on an external data store to store and manage Tracked Vulnerabilities and their related entities. This is a deliberate design decision instead of using CRDs and the Kubernetes API for storage and management operations, as explained below.

Since tracking of the resources and vulnerabilities is done at a very fine level, this would have required to maintain thousands of CRD instances, putting a lot of load on Kubernetes' etcd (e.g. storage space, indices, frequent write operations, etc). Furthermore, the user interface of our solution is a dashboard that presents metrics, statistics and details of Tracked Vulnerabilities, which requires frequent read access to all data. All this would put a lot of stress on both etcd and the Kubernetes API server, potentially affecting negatively its performance.

The design is pluggable, meaning the it is not tied to a specific data store or dashboard. It can potentially support arbitrary data stores, as long as the respective backend is implemented.

3.1.2.4 Access Control

As discussed earlier, each Tracked Vulnerability is bound to a specific Kubernetes namespace. Our design specifies that access to a Tracked Vulnerability affecting a specific Kubernetes namespace should be given only to the ”owner” of that namespace. This can be useful in multi-tenant scenarios, where each tenant is isolated in its own Kubernetes namespace.

3.1.2.5 Controller

To trigger the scan of deployed container images, our approach is based on Kubernetes controllers. Specifically, we employ Kubernetes controllers to watch for changes to Pods and trigger a vulnerability scan whenever a container image is changed.

This asynchronous mechanism allows to react in very short time to changes in container images and update the set of Tracked Vulnerabilities, without impacting the deployment process or harming the performance of the Kubernetes cluster.

3.2 Admission

The goal of the *Admission* component of our solution is to enforce that vulnerable container images are not allowed to be deployed in a Kubernetes cluster, forcing developers to maintain a good security hygiene in their dependencies and subsequently in the resulting container images.

3.2.1 Definitions

Before we proceed with the architecture, we will provide definitions for the terms that will be used later on in the description of the solution.

3.2.1.1 Admission Scan

An *Admission Scan* is a request made by an entity that intends to deploy a new container image to the Kubernetes cluster (e.g. CI/CD operator, user that performs manual operations, etc) to request a set of container images to be scanned for vulnerabilities.

3.2.1.2 Admission Scan Result

An *Admission Scan Result* is a response to a request described in previous section and it contains the results of the vulnerability scan of a single container image. It is obvious that an Admission Scan generates one or more Admission Scan Results, as the former refers to a set of container images and the later a single container image.

3.2.1.3 Admission Policy

An *Admission Policy* is a configurable policy that defines what a ”vulnerable” container image is. It uses the Admission Scan Result that corresponds to the container image as its input for the evaluation.

3.2.2 Architecture

The architecture of this component revolves around enforcing policies on container images before they are deployed to the Kubernetes cluster thus preventing potential known vulnerability exploitation.

The goal of this component is to be setup-agnostic and not depend on external services, such as external container image scanning services. Consequently, the scanning of the container image has to be done by the operator itself. However, container image scanning

is a time-consuming process, as it requires fetching the container image from a remote registry, detecting the installed packages and querying a vulnerability database, and cannot be performed at the time of policy evaluation. As a result, the work flow of the Admission Operator was designed to have two distinct phases, the *scan phase* and the *policy evaluation phase*.

This component is also designed in a Kubernetes-native fashion, utilizing CRDs, Controllers and Admission Webhooks.

3.2.2.1 Custom Resource Definitions

The Admission Scan, Admission Scan Result and Admission Policy presented in the previous sections are backed by Kubernetes CRDs.

An Admission Scan is intended to be created by the deploying actors (e.g. CI/CD operator, user that performs manual operations, etc) that deploy workloads in a Kubernetes namespace. As such, the Kubernetes service account of these entities should be allowed to create instances of that CRD in that namespace.

An Admission Scan Result is the response to an Admission Scan and is created by the operator in the same namespace as its related Admission Scan. The deploying actors may be allowed read-only access to an Admission Scan Result, but must not have permissions to modify it, as in that case they could falsify the results.

An Admission Policy is intended to be created by both the deploying actors and the cluster administrator. The former can create Admission Policies only in the namespace they own and the policy will be evaluated only for container images in that namespace. These policies are called namespace-local policies. The cluster administrator can create Admission Policies in a special namespace and can specify for which namespaces they will be evaluated. These are called global policies.

3.2.2.2 Controller

A Kubernetes controller is employed to watch for instances of the Admission Scan CRD (i.e. vulnerability scan requests) and trigger the vulnerability scan. The results of the vulnerability scan are used to create the related instances of the Admission Scan Result CRD.

3.2.2.3 Admission Webhook

Policy evaluation takes place in a validating admission webhook that watches for Pods. Whenever a Pod is created or updated, the validating admission webhook:

1. Gathers all namespace-local and global Admission Policies that should be evaluated

for container images in this Kubernetes namespace.

2. Retrieves the Admission Scan Result that corresponds to that container image. If one does not already exist, the admission of the Pod is denied.

3. Evaluates all gathered Admission Policies for the retrieved Admission Scan Result. If all Admission Policies pass, the the admission of the Pod is allowed. Otherwise, if at least one fails, the admission of the Pod is denied.

4. IMPLEMENTATION

In this chapter, we will present the technical details of our implementation, challenges and assumptions made during the development process.

4.1 KCIVM Visibility Operator

4.1.1 Introduction

The *KCIVM*[1] *Visibility Operator* is the implementation of the design presented in section 3.1 and follows the Kubernetes Operator pattern described in section 2.4.6.4.

Figure 4.1 presents an overview of the design of the KCIVM Visibility Operator.

4.1.2 Preliminaries

The operator was implemented in Go, utilizing the official Kubernetes libraries where possible. To bootstrap the development process, we used Kubebuilder to generate a basic project structure:

```
$ kubebuilder init \
   --project-name kcivm-visibility-operator \
   --domain security.grnet.gr \
   --repo gitlab.grnet.gr/security/kcivm-visibility-operator
```

4.1.3 Container Image Scanning

4.1.3.1 Trivy

For scanning the container images for packages and their vulnerabilities we decided to use *Trivy*[2], an open-source security scanner. Trivy supports several ”targets” (what it can scan) and ”scanners” (what it can find) [12], including the Container Image target, the OS Packages scanner, the Language Packages scanner and the Known Vulnerabilities scanner.

The aforementioned functionalities cover the all cases specified in the work flow of our design: It can scan container images for both OS and language packages and check if the detected versions are affected by known vulnerabilities.

[1] Kubernetes-native Container Image Vulnerability Management
[2] https://aquasecurity.github.io/trivy/

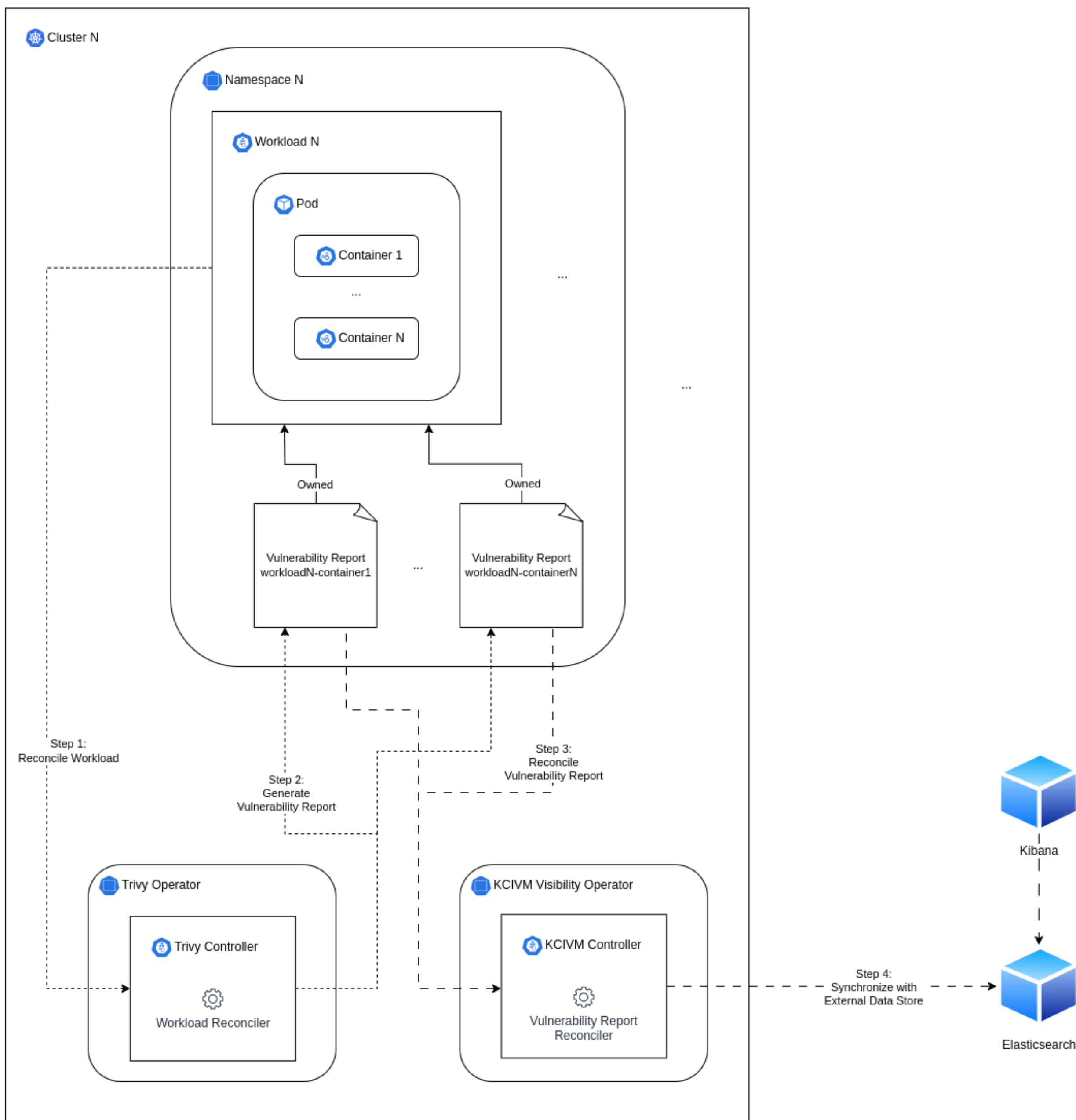


Figure 4.1: KCIVM Visibility Operator Overview.

It supports a wide range of OS package managers, including Alpine, Debian, Ubuntu, RHEL, etc, and language package managers, including PHP, Python, Ruby, Node.js, Java, Go, Rust, .Net, C/C++, etc. An exhaustive and up-to-date list can be found in the official documentation[3].

[3]https://aquasecurity.github.io/trivy/latest/docs/scanner/vulnerability/

4.1.3.2 Trivy Operator

As described in section 2.8.1, Trivy Operator leverages Trivy to ”continuously scan a Kubernetes cluster for security issues”. These scans include scanning container images for known vulnerabilities, making it a perfect fit for our implementation. In the following sections, when referencing Trivy Operator we will refer only to its container image scanning functionality, ignoring other functionalities that are irrelevant to our implementation.

The container image scanning functionality of Trivy Operator is built around the Vulnerability Report CRD and works as follows: A Kubernetes controller watches Pods and workload resources (e.g. Deployment, Job, etc) and if an instance of the Vulnerability Report CRD does not exist for every container image they reference, it triggers a new scan to create the missing vulnerability report.

Trivy is utilized to perform the scanning, in a Kubernetes-native way. Instead of running a Trivy process in its controller's container, Trivy Operator starts a Kubernetes Job of the Trivy container image to run Trivy asynchronously.

Furthermore, Trivy is configured to work in client-server mode. Trivy requires access to a vulnerability DB and downloading it in the ephemeral Job containers would have been inefficient. To overcome this, a Trivy server instance continuously runs in the background to perform the scans submitted by Trivy clients in the jobs.

4.1.3.3 Vulnerability Report CRD

As discussed in the previous section, a *Vulnerability Report* is the result of the vulnerability scanning of a container image. It contains metadata about the container image that was scanned, metadata about the scanner itself, the list of vulnerabilities that were identified and a summary of all vulnerabilities. The API group and kind of the CRD are `aquasecurity.github.io/v1alpha1` and `VulnerabilityReport`, respectively. The specification of the CRD follows:

- `updateTimestamp`: A UTC timestamp representing when this vulnerability report was last updated.
- `scanner`: Metadata about the scanner that generated this vulnerability report.
  - `name`: The name of the scanner (Trivy).
  - `vendor`: The vendor of the scanner (Aqua Security).
  - `version`: The version of the scanner.
- `registry`: The registry from which the container image was pulled.
  - `server`: The FQDN of the registry.
- `artifact`: Information about the container image, as defined in sections 2.3.1 and 3.1.1.2.

  - `repository`: The repository of the container image.
  - `digest`: The digest of the container image.
  - `tag`: The tag of the container image.
- `os`: Information about the OS of the container image.
  - `family`: The OS family (e.g. `debian`).
  - `name`: The version (numeric or codename) of the OS.
  - `eosl`: A flag indicating whether this OS version has reached End of Service Life (EOSL).
- `summary`: A summary of the number of vulnerabilities by severity.
  - `criticalCount`: The number of vulnerabilities with Critical severity.
  - `highCount`: The number of vulnerabilities with High severity.
  - `mediumCount`: The number of vulnerabilities with Medium severity.
  - `lowCount`: The number of vulnerabilities with Low severity.
  - `noneCount`: The number of vulnerabilities with None severity.
  - `unknownCount`: The number of vulnerabilities with Unknown severity.
- `vulnerabilities`: The list of vulnerabilities affecting OS or language packages identified in the container image.
  - `vulnerabilityID`: The identifier of the vulnerability (e.g. CVE ID).
  - `resource`: The name of the vulnerable package.
  - `class`: The class of the package, as defined in section 3.1.1.3.
  - `packageType`: The type in class of the package, as defined in section 3.1.1.3.
  - `installedVersion`: The installed version of the resource.
  - `fixedVersion`: The version of the resource in which this vulnerability is fixed.
  - `publishedDate`: The date/time this vulnerability was published.
  - `lastModifiedDate`: The date/time this vulnerability was last modified.
  - `title`: The title or summary of the vulnerability.
  - `description`: A detailed description of the vulnerability.
  - `severity`: The severity of the vulnerability.
  - `cvss`: One or more CVSS scores assigned to this vulnerability, per vendor that made the assignment.
  - `cvsssource`: The name of the vendor whose CVSS score was used to extract the severity.
  - `primaryLink`: The primary link for the vulnerability.

- `links`: Additional links for the vulnerability.
- `packagePath`: The path of the vulnerable package in the filesystem, optional.
- `target`: The image and package version (similar to our definition of target in section 3.1.1.5).

Vulnerability reports have an expiration time, which by default is 24 hours. They are deleted after they have expired, which triggers a re-scan of the respective container images. This way, vulnerability reports are kept up-to-date as new vulnerabilities emerge.

A sample part of a vulnerability report follows:

```
Artifact:
  Digest:   sha256:903↩
      d3225acecaa272bbdd7273c6c312c2af8b73644058838d23a8c9e6e5c82cf
  Repository: library/debian
Os:
  Family: debian
  Name:  12.7
Registry:
  Server: index.docker.io
Scanner:
  Name:   Trivy
  Vendor: Aqua Security
  Version: 0.50.1
Summary:
  Critical Count: 1
  High Count:  1
  Low Count:   57
  Medium Count: 13
  None Count:  0
  Unknown Count: 0
Update Timestamp: 2024-10-01T13:42:10Z
Vulnerabilities:
  Class: os-pkgs
  Cvss:
    Nvd:
      V2Score:     4.3
      V2Vector:    AV:N/AC:M/Au:N/C:N/I:P/A:N
      V3Score:     3.7
      V3Vector:    CVSS:3.1/AV:N/AC:H/PR:N/UI:N/S:U/C:N/I:L/A:N
  Description:    It was found that apt-key in apt, all versions, ↩
      do not correctly validate gpg keys with the master keyring, ↩
      leading to a potential man-in-the-middle attack.
  Fixed Version:
  Installed Version: 2.6.1
```

```
  Last Modified Date: 2021-02-09T16:08:18Z
  Links:
    https://access.redhat.com/security/cve/cve-2011-3374
    https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=642480
    https://people.canonical.com/~ubuntu-security/cve/2011/CVE↩
       -2011-3374.html
    https://seclists.org/fulldisclosure/2011/Sep/221
    https://security-tracker.debian.org/tracker/CVE-2011-3374
    https://snyk.io/vuln/SNYK-LINUX-APT-116518
    https://ubuntu.com/security/CVE-2011-3374
  Package Type:   debian
  Primary Link:   https://avd.aquasec.com/nvd/cve-2011-3374
  Published Date: 2019-11-26T00:15:11Z
  Resource:       apt
  Score:          3.7
  Severity:       LOW
  Target:         debian@sha256:903↩
     d3225acecaa272bbdd7273c6c312c2af8b73644058838d23a8c9e6e5c82cf↩
      (debian 12.7)
  Title:          It was found that apt-key in apt, all versions, ↩
     do not correctly valid ...
  Vulnerability ID: CVE-2011-3374
...
```

4.1.4 Vulnerability Report Controller

Our implementation revolves around the *Vulnerability Report Controller*, a Kubernetes Controller that watches the Vulnerability Report CRD defined by Trivy Operator. The design and work flow are tightly coupled around the reconciliation logic of the controller.

Kubebuilder was again used to generate a stub controller implementation. However, Kubebuilder does not support creating controllers that watch resources in different API groups than the controller. The command below creates a controller that watches a `VulnerabilityReport` kind in the `kcivm.security.grnet.gr/v1alpha1` API group:

```
$ kubebuilder create api \
   --group kcivm \
   --version v1alpha1 \
   --kind VulnerabilityReport \
   --controller \
   --resource=false
```

The stub implementation was then manually patched to change the kind's API group to `aquasecurity.github.io/v1alpha1`, which is the correct API group for the Vulnera-

bility Report CRD.

The workflow of our controller is the following:

1. Before the controller is started:
   (a) The list of all existing Vulnerability Reports in the Kubernetes cluster is retrieved.
   (b) Targets in the external data store that reference a report not in that list are deleted.
2. When a Vulnerability Report is created or updated, the reconciler:
   (a) Retrieves the Namespace and calculates the Workload from the controller references using the Kubernetes API.
   (b) For every vulnerability in the report:
      i. Extracts the Image, Resource and Vulnerability ID to create the respective Tracked Vulnerability and adds it to the set of Tracked Vulnerabilities.
      ii. Extracts the Image Target and Resource Target pair to create the respective Target and associates it with the corresponding Tracked Vulnerability.
   (c) Synchronizes the Targets with the external data store, creating the new ones, updating the existing ones and deleting the non-existent ones.
3. When a Vulnerability Report is deleted, the reconciler:
   (a) Deletes all Targets in the external data store that reference this report.

Step 1 is required to compensate for reconciliations of deleted instances that were missed while the controller was not running. Moreover, updating the existing Targets in that step is not required, as all Vulnerability Reports will be reconciled immediately after the controller is started.

Furthermore, it should be noted that in step 2b that (1) the size of the set of Tracked Vulnerabilities is less than the number of vulnerabilities in the report, (2) there is a one-to-one correspondence between Targets and vulnerabilities in the report and (3) each Tracked Vulnerability has one or more Targets associated, in accordance to what is described in section 3.1.1.5.

An overview of the Tracked Vulnerability is presented in figure 4.2.

4.1.5 Ignore DB

Our solution can be configured to ignore specific vulnerabilities. A vulnerability may need to be ignored because it is considered to be false positive or has been triaged and is deemed not to affect an application.

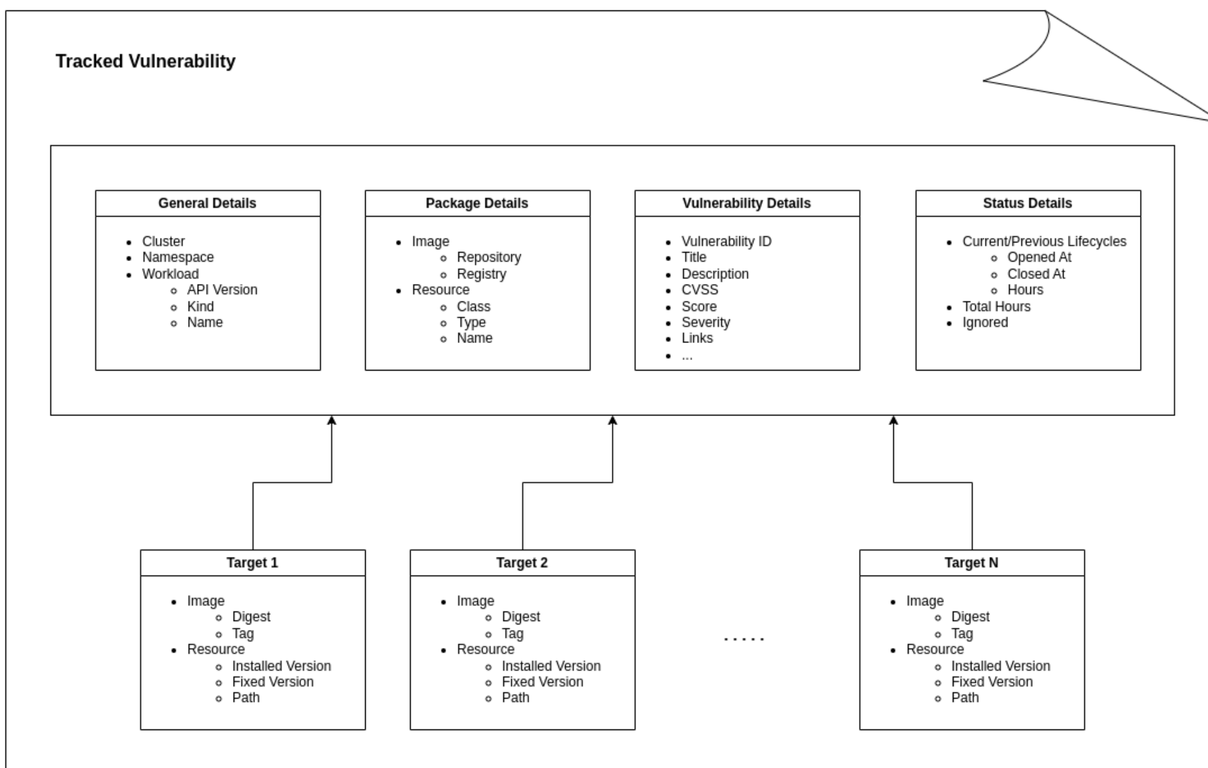


Figure 4.2: Overview of the Tracked Vulnerability.

Furthermore, vendors may not acknowledge that a reported vulnerability is indeed valid or affects their software and opt to ignore it. For example, OS vendors may ignore vulnerabilities in their packages if they consider that the conditions required for the vulnerability to exploitable are not met in their specific configuration.

The list of ignored vulnerabilities can be specified using JSON files via ConfigMaps. The following is an excerpt of such file, showcasing the format:

```
{
  ”os-pkgs”: {
    ”debian”: {
      ”389-ds-base”: [
        ”CVE-2012-0833”,
        ”CVE-2012-2678”,
        ...
      ]
    }
  }
}
```

This list in the example was extracted from the Debian Security Bug Tracker and includes

all vulnerabilities that Debian considers not to affect their product.

4.1.6 Exporter

As described in the design, the Vulnerability Report Controller does not maintain any state in the Kubernetes cluster, but relies on an external data store. To decouple the controller from a specific external data store, we have defined the *Exporter* interface which declares the functionalities that the client of the external data store must implement:

```
type Exporter interface {
    // Create a new or update an existing report target.
    CreateOrUpdate(ctx context.Context, workload client.Object, ↩
        vulnerabilityReport *aquasecurityv1alpha1.↩
        VulnerabilityReport, ignoreDB *ignoredb.IgnoreDB) error

    // Delete the specified report target.
    Delete(ctx context.Context, namespace string, reportName ↩
        string) error

    // Delete all report targets NOT in the specified list.
    SyncDeleted(ctx context.Context, namespace string, reportNames↩
         []string) error
}
```

This interface declares three APIs:

- `CreateOrUpdate`: Synchronize the specified vulnerability report with the external data store:
  - Targets in the vulnerability report that do not exist in the external data store must be created.
  - Targets in the vulnerability report that also exist in the external data store must be updated.
  - Targets in the external data store must be updated that do not exist in the vulnerability report must be deleted.
- `Delete`: All targets in the external data store that reference the specified vulnerability report must be deleted.
- `SyncDeleted`: All targets in the external data store that do not reference one of the specified vulnerability reports must be deleted.

It can be seen that there is a one-to-one correspondence between the APIs and the work flow steps of our controller.

4.1.7 External Data Store

The *External Data Store* that was selected to be used in our solution is Elasticsearch[4]. Elasticsearch is an open-source[5] search engine built on Apache Lucene. It is designed for full-text search, enabling fast and scalable data retrieval across large datasets. It has a HTTP-based API and uses schema-free JSON documents.

We used the official `elasticsearch` Go library[6] to implement the Exporter interface described in section 4.1.6. Our implementation is compatible with both supported stable versions of Elasticsearch (as of the time of writing), 7 and 8.

Each Tracked Vulnerability is represented by an Elasticsearch Document, the format of which is shown in the code snippet below:

```
{
 ”kcivm”: {
   ”cluster”: <string>,
   ”namespace”: <string>,
   ”workload”: {
     ”apiVersion”: <string>,
     ”kind”: <string>,
     ”name”: <string>
   },
   ”image”: {
     ”registry”: <string>,
     ”repository”: <string>
   },
   ”resource”: {
     ”name”: <string>,
     ”class”: <string>,
     ”packageType”: <string>
   },
   ”vulnerability”: {
     ”vulnerabilityID”: <string>,
     ”severity”: <string>,
     ”score”: <float>,
     ”lastModifiedDate”: <string>,
     ”description”: <string>,
     ”primaryLink”: <string>,
     ”links”: [
       <string>,
       ...
     ],
```

[4]https://www.elastic.co/elasticsearch
[5]https://www.elastic.co/blog/elasticsearch-is-open-source-again
[6]https://github.com/elastic/go-elasticsearch

```
  ”publishedDate”: <string>,
  ”title”: <string>,
  ”cvss”: {
    <string>: {
      ”V3Vector”: <string>,
      ”V3Score”: <float>,
      ...
    },
    ...
  }
},
”targets”: [
  {
    ”image”: {
      ”digest”: <string>,
      ”tag”: <string>
    },
    ”resource”: {
      ”fixedVersion”: <string>,
      ”installedVersion”: <string>,
      ”packagePath”: <string>
    },
    ”report”: {
      ”os”: {
        ”name”: <string>,
        ”family”: <string>
      },
      ”containerName”: <string>,
      ”name”: <string>,
      ”scanner”: {
        ”vendor”: <string>,
        ”name”: <string>,
        ”version”: <string>
      },
      ”updateTimestamp”: <string>
    }
  }
],
”status”: {
  ”ignored”: <bool>,
  ”currentLifecycle”: {
    ”hours”: <int>,
    ”closedAt”: <string>,
    ”openedAt”: <string>
```

```
      },
      ”previousLifecycles”: [
        {
          ”hours”: <int>,
          ”closedAt”: <string>,
          ”openedAt”: <string>
        },
        ...
      ],
      ”totalHours”: <int>
    }
  }
}
```

On the Elasticsearch side, we have defined an Elasticsearch Index Template[7] with type mappings for all fields. Some fields (e.g. hours, as described below) are defined as runtime fields and are calculated on document retrieval.

- **Cluster:** A name for the Kubernetes cluster, to distinguish Tracked Vulnerabilities of different Kubernetes clusters. The Kubernetes API does not have the notion of ”cluster name” and this value is a configuration option of the operator.
- **Namespace:** The name of the Namespace.
- **Workload:** API version, kind and name of the Workload.
- **Image:** The registry and repository of the Image.
- **Resource:** The class, type in class and name of the Resource.
- **Vulnerability:** The ID and additional details of the Vulnerability.
- **Targets:** A list of Image Target - Resource Target pairs, along with extra metadata:
  - **Image Target:** The digest and/or tag of the Image.
  - **Resource Target:** The installed version, fixed version and path of the package.
  - **Report:** Metadata of the Vulnerability Report, as specified in Trivy Operator’s CRD.
- **Status:** The status of the Tracked Vulnerability:
  - **Ignored:** Indicates whether this Tracked Vulnerability should be ignored or not, as specified in the Ignore DB.

[7] https://www.elastic.co/guide/en/elasticsearch/reference/current/index-templates.html

- **Current Lifecycle:** The currently tracked lifecycle, including the date/time it was first detected to affect the resource, the date/time it was detected no longer affect the resource and the number of hours it has or had been affecting the resource (calculated dynamically).
- **Previous Lifecycles:** A list of previously tracked lifecycles, which have the same format as the currently tracked lifecycle.
- **Total hours:** The sum of hours of all lifecycles (calculated dynamically).

The name of the Elasticsearch Index, under which the documents are stored, has the following structure: `kcivm-{cluster}-{namespace}`, where `{cluster}` is the cluster's name and `{namespace}` is the namespace's name. This allows us to perform access control on a per-namespace basis by giving access to users only to documents that belong to a specific namespace.

For the implementation of the required work flows in the controller, we utilized the following Elasticsearch APIs:

- **Index:** The Index APIs[8] are used to manage Elasticsearch indices. Specifically:
  - **Exists:** The Index Exists API is used to verify the existence of one or more indices without fetching any data.
  - **Create:** The Create Index API allows you to create an index.
  - **Refresh:** The Refresh API ensures that changes to an index (e.g. new documents, updates or deletions) are immediately visible to search queries. Normally, Elasticsearch automatically refreshes indices periodically (every second by default), but the Refresh API is useful when immediate consistency is needed.
- **Point in Time:** The Point in Time API[9] is used to provide a consistent view of an index at a specific point in time, allowing for reliable pagination and search without interference from ongoing changes. It is implemented by creating a snapshot of the index at the requested point in time.
- **Search:** The Search API[10] is used to perform queries and retrieve documents from the indices. It is used in conjunction with the Point in Time API. All performance recommendations in the documentation are followed (e.g. pagination, unordered retrieval, etc).
- **Bulk:** The Bulk API[11] allows multiple operations (e.g. create or update) to be processed in one call, significantly reducing network overhead.

[8] https://www.elastic.co/guide/en/elasticsearch/reference/current/indices.html
[9] https://www.elastic.co/guide/en/elasticsearch/reference/current/point-in-time-api.html
[10] https://www.elastic.co/guide/en/elasticsearch/reference/current/search-search.html
[11] https://www.elastic.co/guide/en/elasticsearch/reference/current/docs-bulk.html

Finally, it should be noted that Elasticsearch does not offer a ”document locking” or other synchronization mechanism, which would be required in order to process Vulnerability Reports in parallel, as vulnerabilities in different reports may be tracked by the same Tracked Vulnerability. As a result, the implemented controller is configured to reconcile at most one Vulnerability Report at a time (`MaxConcurrentReconciles=1`).

4.1.8 Dashboard

Since we have opted to use Elasticsearch as the data store, using Kibana to implement the data visualization dashboard is the obvious choice. Kibana is a data visualization and exploration tool that works with Elasticsearch. It provides a graphical interface to analyze, explore and visualize data stored in Elasticsearch. Kibana is commonly used for dashboards, monitoring logs, etc.

Our dashboard consists of several panels that provide useful metrics about the state of vulnerabilities in a Kubernetes cluster. Specifically:

- Number of tracked vulnerabilities.
- Number of unique vulnerability IDs.
- Number of tracked vulnerability by namespace.
- Number of tracked vulnerability by image target.
- Percentage of tracked vulnerabilities by severity.
- Percentage of tracked vulnerabilities by class.
- Percentage of targets by fixability.
- Table of the number of the top 100 vulnerability IDs.
- Table of the number of tracked vulnerabilities by cluster - namespace - workload - image - resource.
- Average hours of all lifecycles by severity.
- Average hours of the current lifecycles by severity.
- Table with the details of all vulnerabilities, including but not limited to vulnerability details, status and lifecycles, etc.

By default, the dashboard only shows tracked vulnerabilities that are open (i.e. that have at least one target) and are not ignored. An example dashboard is presented in the evaluation chapter.

Furthermore, Kibana dashboards support interactive filters and queries. Users can apply filters that affect all visualizations on the dashboard. e.g. limit the data to a specific cluster, namespace, workload, image, resource or any of their subfields, etc.

Finally, no special access control management is required on the dashboard level. A user's dashboard is populated with data from Elasticsearch indices they can access. For example, in a multi-tenant Kubernetes setup, a tenant's dashboard will be populated with tracked vulnerabilities in the namespace they own.

4.2 KCIVM Admission Operator

4.2.1 Introduction

The *KCIVM*[12] *Admission Operator* is the implementation of the design presented in section 3.2 and follows the Kubernetes Operator pattern described in section 2.4.6.4.

Figure 4.3 presents an overview of the design of the KCIVM Admission Operator.

4.2.2 Preliminaries

The operator was implemented in Go, utilizing the official Kubernetes libraries where possible. To bootstrap the development process, we used Kubebuilder to generate a basic project structure:

```
$ kubebuilder init \
   --project-name kcivm-admission-operator \
   --domain security.grnet.gr \
   --repo gitlab.grnet.gr/security/kcivm-admission-operator
```

4.2.3 Container Image Scanning

Scanning the container images for packages and their vulnerabilities is again performed using Trivy, as described in the implementation of the Visibility Operator (section 4.1.3). However, Trivy Operator cannot be used in the Admission Operator, as it scans container images after the respective Pods have been created. On the other hand, the requirement of the Admission Operator is to block deployment of container images (i.e. creation of Pods) meaning that the scanning of container images has to be performed beforehand.

[12] Container Image Vulnerability Management in Kubernetes

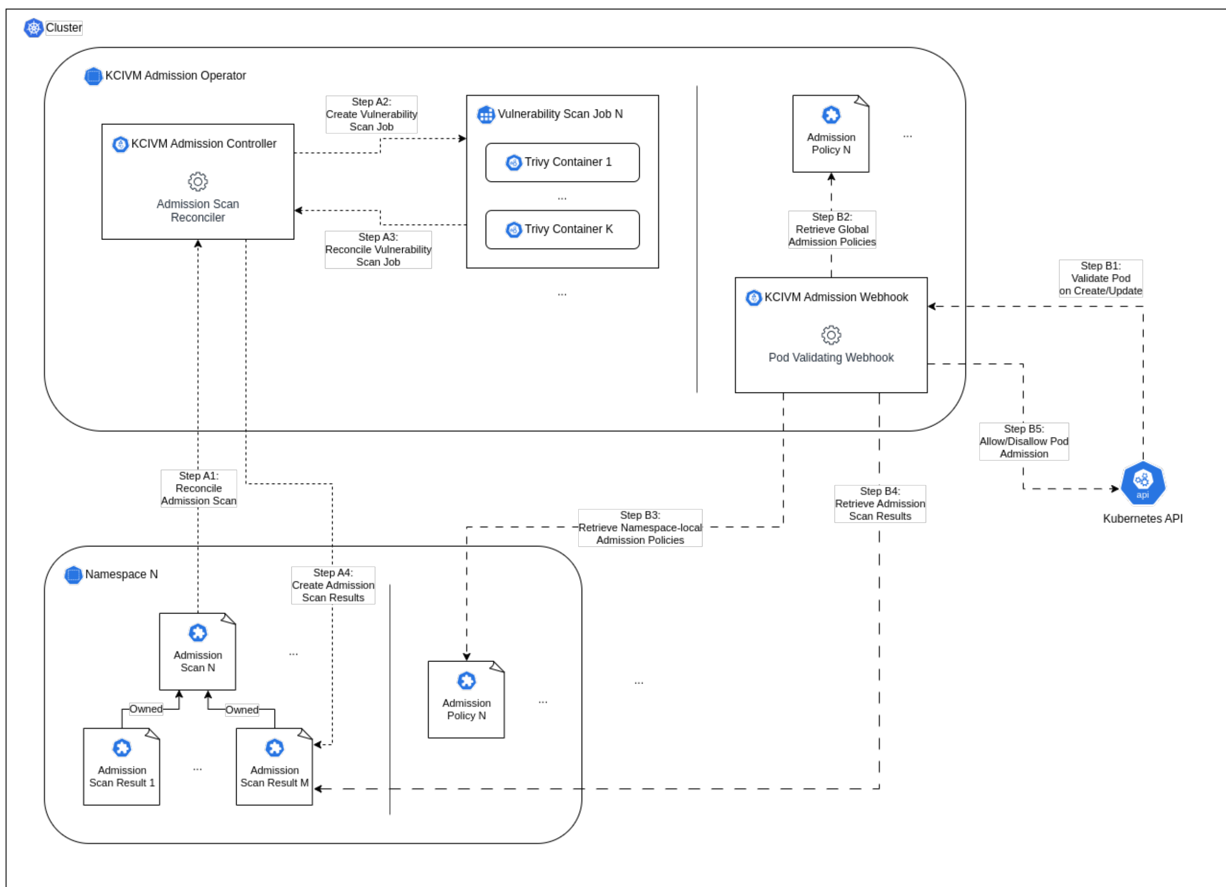


Figure 4.3: KCIVM Admission Operator Overview.

4.2.4 Admission Scan CRD and Controller

The implementation of the Admission Scan functionality is backed by the *Admission Scan CRD* and the *Admission Scan Controller*. The design and work flow are tightly coupled around the reconciliation logic of the controller.

Kubebuilder was again used to generate stub CRD and controller implementations. The command below was used to create the desired CRD and controller in the `kcivm.security.grnet.gr/v1alpha1` API group:

```
$ kubebuilder create api \
   --group kcivm \
   --version v1alpha1 \
   --kind AdmissionScan \
   --controller \
   --resource
```

The specification of the CRD follows:

- `images`: An array of references to container image to scan. The references must be specified using a digest.
- `imagePullSecrets`: An array of references to Kubernetes Secrets of type `kubernetes.io/dockerconfigjson` in the same namespace. The credentials of these secrets are used to retrieve container images from registries that require authentication.

It is important to explain why the container image references in the Admission Scan must be specified using a digest. Each container image will be retrieved from the container registry at different points in time (1) to be scanned by the operator and (2) to be deployed by Kubernetes. Consequently, a container image reference with a tag could possibly resolve to different container images at those point in time. This could allow a malicious entity to bypass the enforcement policies. On the other hand, reference with a digest ensures that the same image that was scanned is the one that will be deployed.

The controller utilizes the Trivy container image to implements the container image scanning (similar to how Trivy Operator works). It watches for Admission Scans and triggers news scans when one is created or updated. The work flow follows:

1. It extracts from the Kubernetes secrets the registry credentials that correspond to each container image reference.
2. It creates in its own namespace a new Kubernetes secret of type `Opaque` that holds the aforementioned container image reference - registry credential pairs.
3. It creates in its own namespace a Kubernetes job with multiple Trivy containers, one for each container image to scan. The registry credentials are passed to each Trivy container using environment variables backed by the aforementioned secret.
4. It watches the job until it successfully completes. Then it parses each Trivy container's output and creates the corresponding Admission Scan Result.

The operator's Kubernetes service account must have the following permissions in all namespaces:

- Get and Watch to Admission Scans.
- Create to Admission Scan Results.
- Get to Secrets.

It should be noted that the job and the corresponding secret are created in the operator's namespace and not the namespace where the Admission Scan exists, so that the owner of that namespace cannot tamper with the scan.

Finally, our implementation includes a validating webhook for the Admission Scan CRD that ensures that the container image references specified during creation or update are indeed specified using a digest. If not, the admission scan request is denied.

An example Admission Scan follows:

```
apiVersion: kcivm.security.grnet.gr/v1alpha1
kind: AdmissionScan
metadata:
  name: myapp-backend
  namespace: myapp
spec:
  images:
    - registry.docker.grnet.gr/security/kcivm/myapp/myapp-app-↩
      release@sha256:9↩
      deba42c8cbc8caa9fa9d3695a172d9de4668fcaa911190fc27b00c8aee199f9↩

  imagePullSecrets:
    - name: myapp-registrycreds
```

4.2.5 Admission Scan Result CRD

The *Admission Scan Result CRD* holds the results of the scan of a single container image.

Kubebuilder was used to generate a stub CRD implementation. The command below was used to create it in the `kcivm.security.grnet.gr/v1alpha1` API group:

```
$ kubebuilder create api \
    --group kcivm \
    --version v1alpha1 \
    --kind AdmissionScanResult \
    --controller=false \
    --resource
```

The specification of the CRD closely resembles Trivy Operator's Vulnerability Report CRD:

- `image`: The container image reference of the scanned container image, as specified in the related Admission Scan.
- `os`: Information about the OS of the container image.
  - `family`: The OS family (e.g. `debian`).
  - `name`: The version (numeric or codename) of the OS.
  - `eosl`: A flag indicating whether this OS version has reached End of Service Life (EOSL).

- `vulnerabilities`: The list of vulnerabilities affecting OS or language packages identified in the container image.
  - `vulnerabilityID`: The identifier of the vulnerability (e.g. CVE ID).
  - `severity`: The severity of the vulnerability.
  - `cvss`: One or more CVSS scores assigned to this vulnerability, per vendor that made the assignment.
  - `publishedDate`: The date/time this vulnerability was published.
  - `lastModifiedDate`: The date/time this vulnerability was last modified.
  - `resource`: The name of the vulnerable package.
  - `class`: The class of the package.
  - `packageType`: The type in class of the package.
  - `installedVersion`: The installed version of the resource.
  - `fixedVersion`: The version of the resource in which this vulnerability is fixed.
  - `status`: The status of the vulnerability as defined by the vendor (e.g. affected, not affected, will not fix, etc).

Since an Admission Scan Result targets a single container image and an Admission Scan specifies multiple container images, multiple Admission Scan Results correspond to a single Admission Scan. This relation is implemented using Kubernetes objects' owner references, where each Admission Scan Result is owned by an Admission Scan.

Permissions to create Admission Scan Results in any namespace should be granted only to the operator. Namespace owner may optionally be granted permissions to read Admission Scan Results, but this is not required for the functionality of the operator.

Finally, Admission Scan Results have a configurable expiration time, which by default is 12 hours. After that time has passed, they are disregarded during policy evaluation and eventually deleted. This mechanism is employed to ensure that scan results are updated for new vulnerabilities.

A sample part of an Admission Scan Result follows:

```
Image: registry.docker.grnet.gr/security/kcivm/myapp/myapp-app-↩
   release@sha256:9↩
   deba42c8cbc8caa9fa9d3695a172d9de4668fcaa911190fc27b00c8aee199f9
Os:
  Family: debian
  Name:  12.7
Vulnerabilities:
  Class: os-pkgs
  Cvss:
    Nvd:
      V2Score:     4.3
```

```
   V2Vector:    AV:N/AC:M/Au:N/C:N/I:P/A:N
   V3Score:     3.7
   V3Vector:    CVSS:3.1/AV:N/AC:H/PR:N/UI:N/S:U/C:N/I:L/A:N
 Installed Version: 2.6.1
 Last Modified Date: 2021-02-09T16:08:18Z
 Package Type:   debian
 Published Date: 2019-11-26T00:15:11Z
 Resource:       apt
 Severity:       LOW
 Status:         affected
 Vulnerability ID: CVE-2011-3374
 ...
```

4.2.6 Admission Policy CRD

The *Admission Policy CRD* is used to allow definition of configurable admission policies. Policies can be specified by both namespace owners, called namespace-local policies, and by system administrators, covering one or more namespaces, called global policies.

For the policy definition, we decided to use *Rego*[13], a powerful but easy to read and write policy language. The implementation utilizes the official Go library[14].

Kubebuilder was used to generate a stub CRD implementation. The command below was used to create it in the `kcivm.security.grnet.gr/v1alpha1` API group:

```
$ kubebuilder create api \
   --group kcivm \
   --version v1alpha1 \
   --kind AdmissionPolicy \
   --controller=false \
   --resource
```

The specification of the CRD is the following:

- `rego`: The Rego policy.
- `namespaces`: A list of namespaces, where this policy will be applied. It can only be set for global policies.

Our implementation also includes a validating webhook for the Admission Policy CRD that parses the specified Rego policy and ensures that it is valid before it is persisted.

[13] https://www.openpolicyagent.org/docs/latest/policy-language/
[14] https://github.com/open-policy-agent/opa

In the design section 3.2.2.1 we discussed that global policies can only be created by cluster administrators in a "special namespace". That special namespace was chosen to be the operator's namespace, which by default is `kcivm-admission-operator-system`. Furthermore, the operator does not enforce policies in its own namespace, meaning that deployments of the operator's pods are never disallowed.

As discussed in section 3.2.1.3, the input to the specified Rego policy is an Admission Scan Result. As the output, our implementation demands that the Rego policy has a rule named `allow`, whose evaluated value will be used to determine whether admission policy should be allowed or not. Furthermore, we have implemented a policy library with utilities that are available to be used by Rego policies.

An example global admission policy follows:

```
apiVersion: kcivm.security.grnet.gr/v1alpha1
kind: AdmissionPolicy
metadata:
  name: vuln-count-policy
  namespace: kcivm-admission-operator-system
spec:
  rego: |
    package kcivm.policy

    import rego.v1
    import data.kcivm.library

    default vulnerability_ignored(_) := false
    vulnerability_ignored(vulnerability) if {
      vulnerability.status == "not_affected"
    }
    vulnerability_ignored(vulnerability) if {
        vulnerability.status == "will_not_fix"
    }

    vulnerabilities := [vulnerability |
      vulnerability := input.vulnerabilities[i]
      not vulnerability_ignored(vulnerability)
    ]
    vulnerability_summary := library.make_vulnerability_summary(↩
        vulnerabilities)

    default allow := false
    allow if {
        vulnerability_summary["CRITICAL"] == 0
        vulnerability_summary["HIGH"] < 5
        vulnerability_summary["MEDIUM"] < 10
```

```
  }

namespaces:
  - myapp
  - yourapp
  - theirapp
```

This Rego policy works as follows:

- Iterates over all vulnerabilities in the input and filters out all vulnerabilities that have a status of `not_affected` or `will_not_fix`, utilizing the `vulnerability_ignored` function.
- Calls the `make_vulnerability_summary` function from the policy library, to create a summary of vulnerability counts per severity.
- Evaluates the `allow` rule to true only if the summary of vulnerabilities is below the specified thresholds, in this case no vulnerabilities of critical severity, less than 5 vulnerabilities of high severity and less than 10 vulnerabilities of medium severity.

Other examples of policies that may be implemented are:

- Disallow if any package or a specific package is affected by a vulnerability ID.
- Disallow if any package or a specific package is affected is by a vulnerability with a specific CVSS metric (e.g. critical integrity).

In all cases, the list of vulnerabilities that will be taken into account can be narrowed down. For example:

- Take into account only non-vendor-ignored vulnerabilities (as in the example above).
- Take into account only vulnerabilities that were published after a specific date/time.
- Take into account only vulnerabilities that have been assessed by a specific trusted vendor.

4.2.7 Pod Validating Webhooks

The implementation of the policy evaluation phase, as described in the architecture section 3.2.2, revolves around validating webhooks for Pods. The webhooks are called before a Pod is created or updated to evaluate the specified admission policies. Policy evaluation takes place on the Pod-level as this is the smallest deployable compute object in Kubernetes and inherently covers all workload scenarios.

We did not use Kubebuilder to scaffold the webhook implementation as the latest version at the time of the development used deprecated interfaces from the Kubernetes runtime[15][16].

The work flow of both the create and update validating webhooks it the following:

1. Retrieve all Admission Policies that should be evaluated for the Pod's namespace:
   - Namespace-local ones, defined in that namespace.
   - Global ones, which target that namespace.
2. For every container specified in the Pod:
   (a) Retrieve the container image reference.
   (b) Retrieve the latest non-expired Admission Scan Result that correspond to this container image reference.
   (c) For each policy:
      i. Evaluate the policy using the scan result as its input.
      ii. If policy evaluation fails, deny Pod admission.
3. If all policies for all containers have been evaluated and their result was true, allow Pod admission.

[15]https://github.com/kubernetes-sigs/kubebuilder/pull/4060
[16]https://github.com/kubernetes-sigs/kubebuilder/pull/4150

5. EVALUATION

In this chapter, we present qualitative and quantitative evaluations of the Visibility and Admission operators, respectively, conducted in a real-world environment to ensure the results reflect practical performance and behavior. The tests were performed on a multi-node Kubernetes cluster, with external services that the operator depends on deployed on remote hosts and accessed over the network. This setup simulates typical production environments, where network latency, multi-node coordination and external service dependencies affect the operators' performance.

Experimental Setup: The experimental setup consists of a two-node Kubernetes cluster running the K3s[1] distribution. Each node runs on a virtual machine equipped with a 4-core Intel Xeon E5-2650 processor, 16 GB of RAM and a 1GbE network interface. The image registry and the Elasticsearch instance used as the data store are deployed on remote hosts and are accessed over the network.

Furthermore, we created a vulnerable application to be used in the evaluation of both operators. It consists of multiple workloads, including Kubernetes Deployments (e.g. web frontend, backend server, database server, etc), Kubernetes Jobs and Kubernetes CronJobs. The aforementioned workloads use both custom images, with extra OS and language packages of diverse types installed, and unmodified ones. In both cases, the images and the packages that were used were deliberately chosen to be old versions that are known to be affected by multiple vulnerabilities.

5.1 KCIVM Visibility Operator

In this section, we present a demo instance of the KCIVM Visibility Operator, which was set up to evaluate its operation.

Kubebuilder, which as already discussed was used to bootstrap development of the operator, provides a Makefile with several targets that can be used for managing, building and deploying the operator.

The next command is used to build the operator's container image and push it to a private image registry.

```
$ make docker-build docker-push IMG=registry.docker.grnet.gr/↩
   security/kcivm/visibility-operator:latest
```

Then, the following command is used to deploy the previously-built container image to a Kubernetes cluster. Under the hood, the Makefile uses Kustomize[2] for customizing the manifests before deploying.

[1] https://k3s.io/

[2] https://github.com/kubernetes-sigs/kustomize

```
$ make deploy IMG=registry.docker.grnet.gr/security/kcivm/↩
   visibility-operator:latest
```

Finally, the next command can be used to undeploy the operator from the cluster.

```
$ make undeploy
```

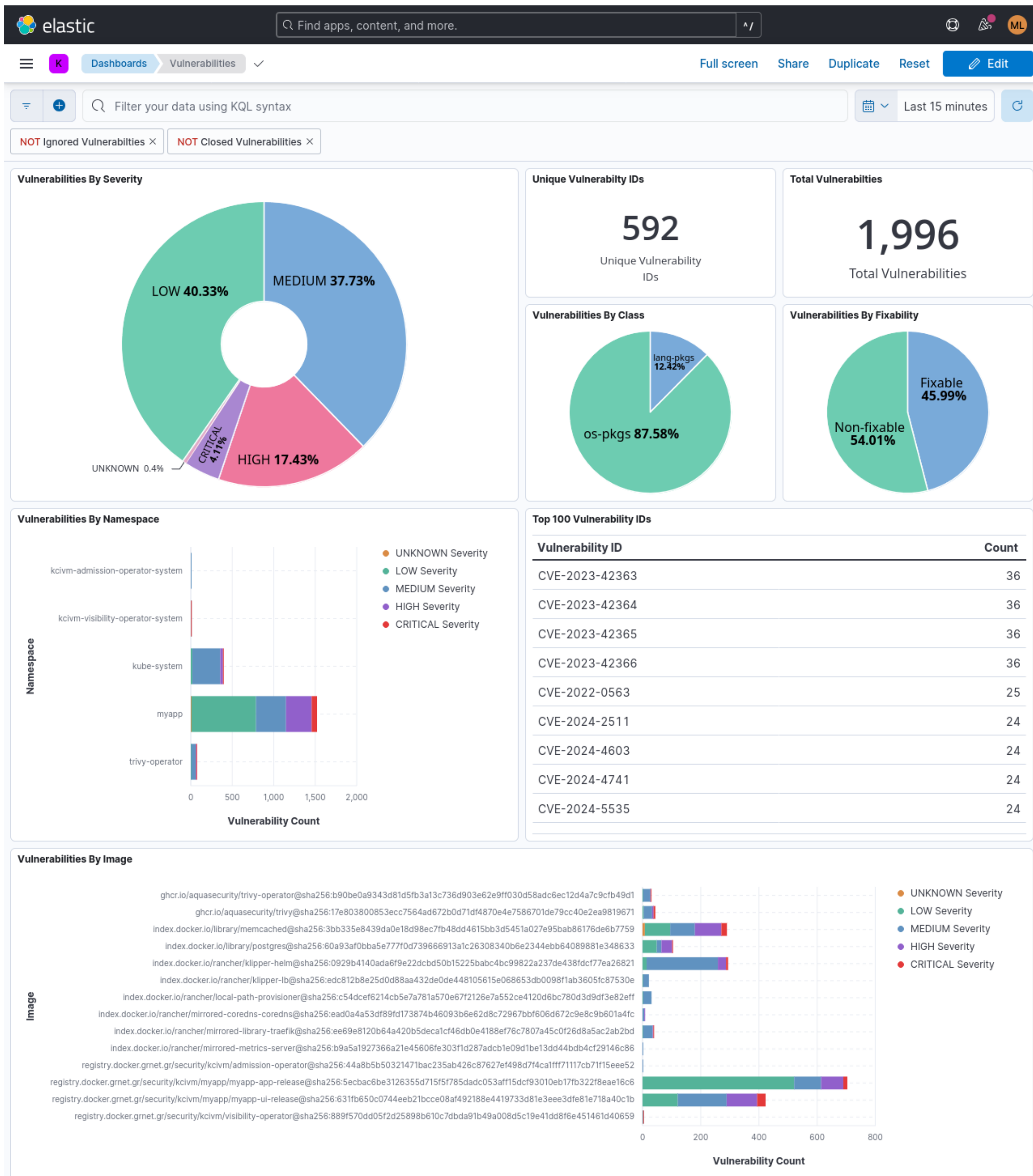
After the operator has been deployed, it immediately starts reconciling existing vulnerability reports and synchronizing tracked vulnerabilities and targets with the external data store.

Figures 5.1 and 5.2 present the default Kibana dashboard. The Elasticsearch user that we use to access Kibana has access to indices that correspond to all namespaces in the cluster.

Figures 5.3 and 5.4 present the Elasticsearch document of a tracked vulnerability. It can be seen that it includes all information discussed in the description and implementation sections.

Figure 5.5 presents the same dashboard, filtered to include results of `myapp` only and after all vulnerabilities affecting `myapp`'s workloads had been fixed. The screenshot has been annotated to mark the filter.

Figure 5.6 presents the previous document, but after the vulnerability had been fixed. The screenshot has been annotated to mark the change (lifecycle close timestamp).



Vulnerability ID	Count
CVE-2023-42363	36
CVE-2023-42364	36
CVE-2023-42365	36
CVE-2023-42366	36
CVE-2022-0563	25
CVE-2024-2511	24
CVE-2024-4603	24
CVE-2024-4741	24
CVE-2024-5535	24

Figure 5.1: Kibana Dashboard (1/2).

Vulnerabilities By Resource

Cluster	Namespace	Workload API Version	Workload Kind	Workload Name	Image Registry	Image Repository	Resource Class	Resource Type	Resource Name	Count
testing	myapp	apps/v1	Deployment	myapp-backend	registry.docker.	security/kcivm/n	os-pkgs	debian	vim	92
testing	myapp	apps/v1	Deployment	myapp-backend	registry.docker.	security/kcivm/n	os-pkgs	debian	vim-common	92
testing	myapp	apps/v1	Deployment	myapp-backend	registry.docker.	security/kcivm/n	os-pkgs	debian	vim-runtime	92
testing	myapp	apps/v1	Deployment	myapp-backend	registry.docker.	security/kcivm/n	os-pkgs	debian	xxd	92
testing	myapp	apps/v1	Deployment	myapp-backend	registry.docker.	security/kcivm/n	os-pkgs	debian	libc-bin	14
testing	myapp	apps/v1	Deployment	myapp-backend	registry.docker.	security/kcivm/n	os-pkgs	debian	libc6	14
testing	myapp	apps/v1	Deployment	myapp-backend	registry.docker.	security/kcivm/n	os-pkgs	debian	libsystemd0	13
testing	myapp	apps/v1	Deployment	myapp-backend	registry.docker.	security/kcivm/n	os-pkgs	debian	libudev1	13
testing	myapp	apps/v1	Deployment	myapp-backend	registry.docker.	security/kcivm/n	os-pkgs	debian	libpython3.7-mi	12
testing	myapp	apps/v1	Deployment	myapp-backend	registry.docker.	security/kcivm/n	os-pkgs	debian	libpython3.7-stc	12

Figure 5.2: Kibana Dashboard (2/2).

elastic | Find apps, content, and more.

Discover > kcivm-testing-myapp#18cd0dbb9f9d4e2aad0cb132211d2097

Table JSON

Search field names

Field	Value
_id	18cd0dbb9f9d4e2aad0cb132211d2097
_ignored	-
_index	kcivm-testing-myapp
_score	0
kcivm.cluster	testing
kcivm.image.registry	registry.docker.grnet.gr
kcivm.image.repository	security/kcivm/myapp/myapp-app-release
kcivm.namespace	myapp
kcivm.resource.class	os-pkgs
kcivm.resource.name	libc6
kcivm.resource.packageType	debian
kcivm.status.currentLifecycle.hours	1
kcivm.status.currentLifecycle.openedAt	Oct 4, 2024 @ 17:28:00.000
kcivm.status.ignored	false
kcivm.status.totalHours	1
kcivm.targets.image.digest	sha256:5ecbac6be3126355d715f5f785dadc053aff15dcf93010eb17fb322f8eae16c6
kcivm.targets.image.reference	registry.docker.grnet.gr/security/kcivm/myapp/myapp-app-release@sha256:5ecbac6be3126355d715f5f785dadc053aff15dcf93010eb17fb322f8eae16c6
kcivm.targets.image.tag	1d39863d
kcivm.targets.report.containerName	myapp-backend
kcivm.targets.report.name	replicaset-myapp-backend-5dd7bb8fc-myapp-backend
kcivm.targets.report.os.eosl	true
kcivm.targets.report.os.family	debian
kcivm.targets.report.os.name	10.13
kcivm.targets.report.scanner.name	Trivy
kcivm.targets.report.scanner.vendor	Aqua Security

Rows per page: 25 < 1 2 >

Figure 5.3: Elasticsearch Document (1/2).

elastic Find apps, content, and more.

Discover kcivm-testing-myapp#18cd0dbb9f9d4e2aad0cb132211d2097

Table JSON

Search field names

Field	Value
kcivm.targets.report.scanner.version	0.50.1
kcivm.targets.report.updateTimestamp	Oct 4, 2024 @ 17:39:38.000
kcivm.targets.resource.installedVersion	2.28-10+deb10u3
kcivm.targets.resource.target	registry.docker.grnet.gr/security/kcivm/myapp/myapp-app-release:1d39863d (debian 10.13)
kcivm.vulnerability.cvss.nvd.V2Score	4
kcivm.vulnerability.cvss.nvd.V2Vector	AV:N/AC:L/Au:S/C:N/I:N/A:P
kcivm.vulnerability.cvss.redhat.V2Score	5
kcivm.vulnerability.cvss.redhat.V2Vector	AV:N/AC:L/Au:N/C:N/I:N/A:P
kcivm.vulnerability.description	The glob implementation in the GNU C Library (aka glibc or libc6) allows remote authenticated users to cause a denial of service (CPU and memory consumption) via crafted glob expressions that do not match any pathnames, as demonstrated by glob expressions in STAT commands to an FTP daemon, a different vulnerability than CVE-2010-2632.
kcivm.vulnerability.lastModifiedDate	Sep 1, 2021 @ 15:15:07.000
kcivm.vulnerability.links	[http://cxib.net/stuff/glob-0day.c, http://securityreason.com/achievement_securityalert/89, http://securityreason.com/exploitalert/9223, https://access.redhat.com/security/cve/CVE-2010-4756, https://bugzilla.redhat.com/show_bug.cgi?id=681681, https://bugzilla.redhat.com/show_bug.cgi?id=CVE-2010-4756, https://nvd.nist.gov/vuln/detail/CVE-2010-4756, https://www.cve.org/CVERecord?id=CVE-2010-4756]
kcivm.vulnerability.primaryLink	https://avd.aquasec.com/nvd/cve-2010-4756
kcivm.vulnerability.publishedDate	Mar 2, 2011 @ 22:00:01.000
kcivm.vulnerability.severity	LOW
kcivm.vulnerability.title	glibc: glob implementation can cause excessive CPU and memory consumption due to crafted glob expressions
kcivm.vulnerability.vulnerabilityID	CVE-2010-4756
kcivm.workload.apiVersion	apps/v1
kcivm.workload.kind	Deployment
kcivm.workload.name	myapp-backend

Rows per page: 25 < 1 2 >

Figure 5.4: Elasticsearch Document (2/2).

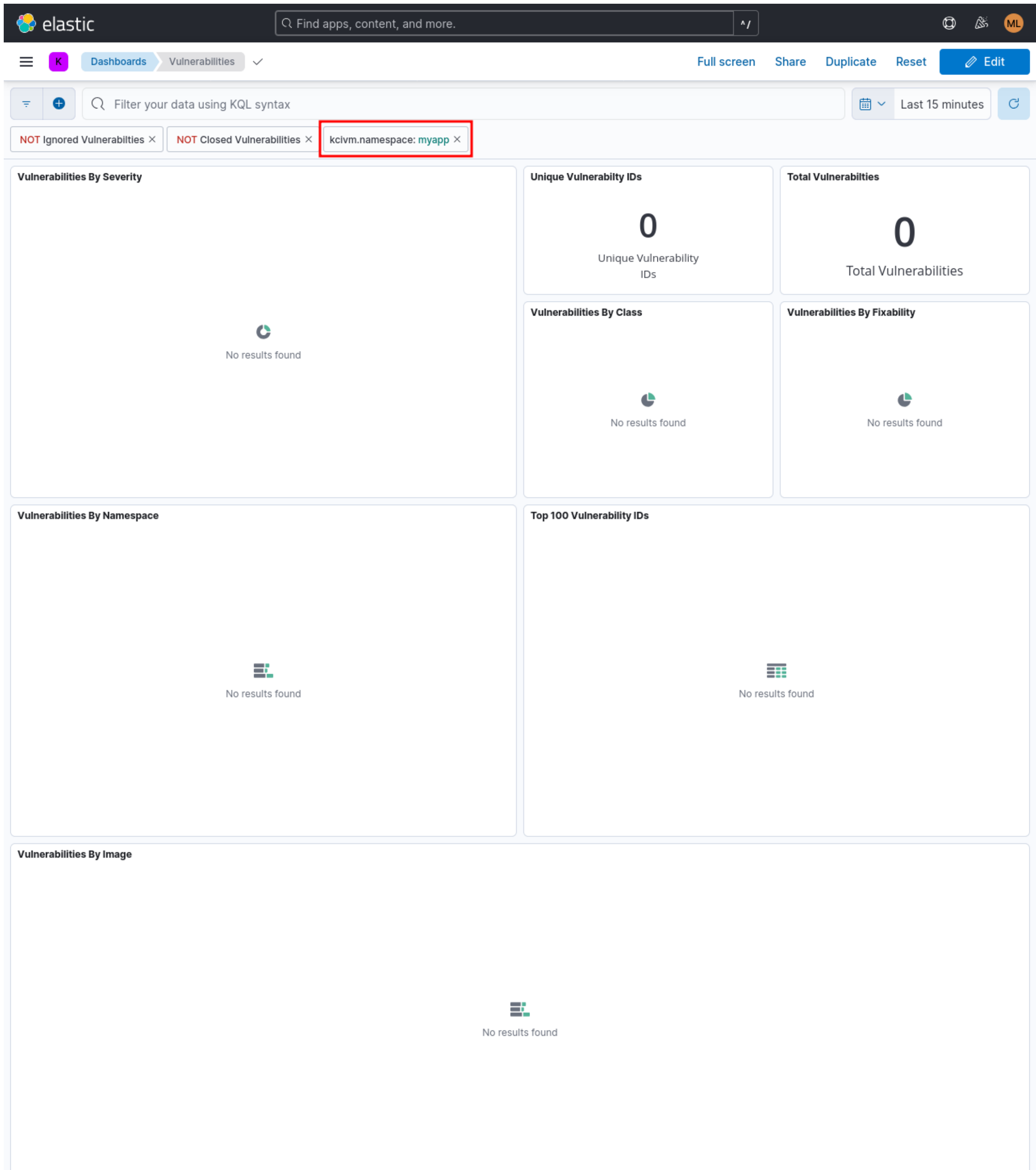


Figure 5.5: Kibana Dashboard, after myapp's vulnerabilities had been fixed.

elastic

Find apps, content, and more.

Discover > kcivm-testing-myapp#18cd0dbb9f9d4e2aad0cb132211d2097

Table JSON

Search field names

Field	Value
_id	18cd0dbb9f9d4e2aad0cb132211d2097
_ignored	-
_index	kcivm-testing-myapp
_score	0
kcivm.cluster	testing
kcivm.image.registry	registry.docker.grnet.gr
kcivm.image.repository	security/kcivm/myapp/myapp-app-release
kcivm.namespace	myapp
kcivm.resource.class	os-pkgs
kcivm.resource.name	libc6
kcivm.resource.packageType	debian
kcivm.status.currentLifecycle.closedAt	Oct 12, 2024 @ 18:36:52.000
kcivm.status.currentLifecycle.openedAt	Oct 4, 2024 @ 17:28:00.000
kcivm.status.ignored	false
kcivm.status.totalHours	194
kcivm.vulnerability.cvss.nvd.V2Score	4
kcivm.vulnerability.cvss.nvd.V2Vector	AV:N/AC:L/Au:S/C:N/I:N/A:P
kcivm.vulnerability.cvss.redhat.V2Score	5
kcivm.vulnerability.cvss.redhat.V2Vector	AV:N/AC:L/Au:N/C:N/I:N/A:P
kcivm.vulnerability.description	The glob implementation in the GNU C Library (aka glibc or libc6) allows remote authenticated users to cause a denial of service (CPU and memory consumption) via crafted glob expressions that do not match any pathnames, as demonstrated by glob expressions in STAT commands to an FTP daemon, a different vulnerability than CVE-2010-2632.
kcivm.vulnerability.lastModifiedDate	Sep 1, 2021 @ 15:15:07.000
kcivm.vulnerability.links	[http://cxib.net/stuff/glob-0day.c, http://securityreason.com/achievement_securityalert/89, http://securityreason.com/exploitalert/9223, https://access.redhat.com/security/cve/CVE-2010-4756, https://bugzilla.redhat.com/show_bug.cgi?id=681681, https://bugzilla.redhat.com/show_bug.cgi?id=CVE-2010-4756, https://nvd.nist.gov/vuln/detail/CVE-2010-4756, https://www.cve.org/CVERecord?id=CVE-2010-4756]
kcivm.vulnerability.primaryLink	https://avd.aquasec.com/nvd/cve-2010-4756
kcivm.vulnerability.publishedDate	Mar 2, 2011 @ 22:00:01.000
kcivm.vulnerability.severity	LOW

Rows per page: 25 — 1 2

Figure 5.6: Elasticsearch Document, after the vulnerability had been fixed.

5.2 KCIVM Admission Operator

In this section, we present a demo instance of the KCIVM Admission Operator, which was set up to evaluate its operation.

In addition to the Kubebuilder commands presented in KCIVM Visibility Operator's evaluation section, the following command needs to be used before building the operator in order to generate the CRD manifests (YAML files) from the corresponding Go types.

```
$ make manifests
```

The rest of the commands to build and deploy the operator remain the same.

```
$ make docker-build docker-push IMG=registry.docker.grnet.gr/↩
   security/kcivm/admission-operator:latest
$ make deploy IMG=registry.docker.grnet.gr/security/kcivm/↩
   admission-operator:latest
```

As part of the evaluation, we implemented two admission policies.

Severity Thresholds Policy: The first admission policy ensures that the specified severity thresholds are not exceeded. It takes into account only those vulnerabilities that are not marked as ”not affected” or ”will not fix” by the vendor. The specified thresholds are 0 critical, less than 5 high, less than 10 medium and less than 100 low severity vulnerabilities. This admission policy is installed as a global policy, as it is created in the special `kcivm-admission-operator-system` namespace, and is applied to all namespaces, as no specific namespaces are defined. The admission policy's manifest is shown in the following code snippet.

```
apiVersion: kcivm.security.grnet.gr/v1alpha1
kind: AdmissionPolicy
metadata:
  name: severity-thresholds
  namespace: kcivm-admission-operator-system
spec:
  rego: |
    package kcivm.policy

    import rego.v1
    import data.kcivm.library

    default vulnerability_ignored(_) := false
    vulnerability_ignored(vulnerability) if {
      vulnerability.status == ”not_affected”
    }
```

```
    vulnerability_ignored(vulnerability) if {
       vulnerability.status == "will_not_fix"
    }

    vulnerabilities := [vulnerability |
       vulnerability := input.vulnerabilities[i]
       not vulnerability_ignored(vulnerability)
    ]
    vulnerability_summary := library.make_vulnerability_summary(↩
       vulnerabilities)

    default allow := false
    allow if {
       vulnerability_summary["CRITICAL"] == 0
       vulnerability_summary["HIGH"] < 5
       vulnerability_summary["MEDIUM"] < 10
       vulnerability_summary["LOW"] < 100
    }
```

Vulnerability ID Policy: The second admission policy ensures that a specific package is not affected by a specific vulnerability. In this example, the package is Debian's `libssl1.1` and the vulnerability is CVE-2022-2068. This admission policy is installed as a namespace-local policy, as it is created in `myapp`'s namespace. The admission policy's manifest is shown in the following code snippet.

```
apiVersion: kcivm.security.grnet.gr/v1alpha1
kind: AdmissionPolicy
metadata:
  name: cve-2022-2068
  namespace: myapp
spec:
  rego: |
    package kcivm.policy

    import rego.v1
    import data.kcivm.library

    resource_affected_by_vulnerability(vulnerabilities, class, ↩
       package_type, resource, vulnerability_id) if {
       some vulnerability in vulnerabilities
       vulnerability.class == class
       vulnerability.packageType == package_type
       vulnerability.resource == resource
       vulnerability.vulnerabilityID == vulnerability_id
```

```
  }

  default allow := false
  allow if {
     not resource_affected_by_vulnerability(input.↩
        vulnerabilities, "os-pkgs", "debian", "libssl1.1", "CVE↩
        -2022-2068")
  }
```

The following commands were used to create the admission policies in the testing cluster.

```
$ kubectl apply -f global-admission-policy.yaml
admissionpolicy.kcivm.security.grnet.gr/severity-thresholds ↩
   created
$ kubectl apply -f myapp-admission-policy.yaml
admissionpolicy.kcivm.security.grnet.gr/cve-2022-2068 created
```

Then we attempted to deploy a vulnerable workload, the manifest of which is shown in the following code snippet. The container image that is used is one that is affected by multiple security vulnerabilities.

```
apiVersion: v1
kind: Pod
metadata:
  namespace: myapp
  name: vulnerable-workload
spec:
  containers:
  - name: workload
    image: registry.docker.grnet.gr/security/kcivm/myapp/myapp-ui-↩
       release@sha256:↩
       a763da655ed79c7d2b54c953256cc5477e07cf8d183f9c2989eccb2d0b50270f↩

  imagePullSecrets:
  - name: registrycreds
```

Attempting to deploy the workload, we get an error response. The error response comes from the operator's validating webhook and informs us that an admission scan result does not already exist and thus the policies cannot be validated.

```
$ kubectl apply -f myapp-vulnerable-workload.yaml
The Pod "vulnerable-workload" is invalid: spec.containers[0].↩
   image: Invalid value: "registry.docker.grnet.gr/security/kcivm/↩
   myapp/myapp-ui-release@sha256:↩
   a763da655ed79c7d2b54c953256cc5477e07cf8d183f9c2989eccb2d0b50270f↩
   ": No admission scan result was found for this image.
```

As discussed in the description section, admission scan results are created by the operator as a response to admission scan requests. The following manifest shows an admission scan for the aforementioned image.

```
apiVersion: kcivm.security.grnet.gr/v1alpha1
kind: AdmissionScan
metadata:
  name: vulnerable-workload
  namespace: myapp
spec:
  images:
    - registry.docker.grnet.gr/security/kcivm/myapp/myapp-ui-↩
        release@sha256:↩
        a763da655ed79c7d2b54c953256cc5477e07cf8d183f9c2989eccb2d0b50270f↩

  imagePullSecrets:
    - name: registrycreds
```

The admission scan request is created as follows.

```
$ kubectl apply -f myapp-vulnerable-workload-admission-scan.yaml
admissionscan.kcivm.security.grnet.gr/vulnerable-workload created
$ kubectl -n myapp get admissionscan vulnerable-workload
NAME                PHASE
vulnerable-workload Succeeded
$ kubectl -n myapp get admissionscanresults
NAME                        AGE
admission-scan-result-xd67r 88s
```

After it succeeds, we can check the resulting admission scan result.

```
$ kubectl -n myapp get admissionscanresults
NAME                        AGE
admission-scan-result-xd67r 88s
$ kubectl -n myapp describe admissionscanresult admission-scan-↩
   result-xd67r
Name:        admission-scan-result-xd67r
Namespace:   myapp
Labels:      kcivm.security.grnet.gr/admission-scan-image-ref-hash=↩
   c375aa67
             kcivm.security.grnet.gr/admission-scan-job-name=↩
                admission-scan-job-bzns4
             kcivm.security.grnet.gr/admission-scan-job-namespace=↩
                kcivm-admission-operator-system
             kcivm.security.grnet.gr/admission-scan-job-uid=94d9a1f6↩
                -d42b-4ed7-9752-2fe1c992bb9a
```

```
            kcivm.security.grnet.gr/admission-scan-name=vulnerable-↩
               workload
            kcivm.security.grnet.gr/admission-scan-namespace=myapp
            kcivm.security.grnet.gr/admission-scan-spec-hash=0↩
               c6bb19a
            kcivm.security.grnet.gr/admission-scan-uid=af8cbdae-3↩
               aec-4518-afe8-00528173ad81
Annotations: kcivm.security.grnet.gr/admission-scan-creation-↩
   timestamp: 2024-10-13T17:16:11Z
API Version: kcivm.security.grnet.gr/v1alpha1
Kind:        AdmissionScanResult
Metadata:
  Creation Timestamp: 2024-10-13T17:16:45Z
  Generate Name:  admission-scan-result-
  Generation:     1
  Owner References:
    API Version:      kcivm.security.grnet.gr/v1alpha1
    Block Owner Deletion: true
    Controller:       true
    Kind:             AdmissionScan
    Name:             vulnerable-workload
    UID:              af8cbdae-3aec-4518-afe8-00528173ad81
  Resource Version:  35515480
  UID:               c1af56ac-7a1e-430b-9eab-c214cbf1a28b
Spec:
  Image: registry.docker.grnet.gr/security/kcivm/myapp/myapp-ui-↩
     release@sha256:↩
     a763da655ed79c7d2b54c953256cc5477e07cf8d183f9c2989eccb2d0b50270f↩

  Os:
    Family: debian
    Name:  11.3
  Vulnerabilities:
    Class: os-pkgs
    Cvss:
      Nvd:
        V2Score:     4.3
        V2Vector:    AV:N/AC:M/Au:N/C:N/I:P/A:N
        V3Score:     3.7
        V3Vector:    CVSS:3.1/AV:N/AC:H/PR:N/UI:N/S:U/C:N/I:L/A:N
    Installed Version: 2.2.4
    Last Modified Date: 2021-02-09T16:08:18Z
    Package Type:   debian
    Published Date: 2019-11-26T00:15:11Z
```

```
  Resource:       apt
  Severity:       LOW
  Status:         affected
  Vulnerability ID: CVE-2011-3374
 ...
```

Now we can attempt to redeploy the vulnerable workload.

```
$ kubectl apply -f myapp-vulnerable-workload.yaml
The Pod "vulnerable-workload" is invalid:
* spec.containers[0].image: Invalid value: "registry.docker.grnet↩
   .gr/security/kcivm/myapp/myapp-ui-release@sha256:↩
   a763da655ed79c7d2b54c953256cc5477e07cf8d183f9c2989eccb2d0b50270f↩
   ": Image disallowed by admission policy "myapp/cve-2022-2068".
* spec.containers[0].image: Invalid value: "registry.docker.grnet↩
   .gr/security/kcivm/myapp/myapp-ui-release@sha256:↩
   a763da655ed79c7d2b54c953256cc5477e07cf8d183f9c2989eccb2d0b50270f↩
   ": Image disallowed by admission policy "kcivm-admission-↩
   operator-system/severity-thresholds".
```

It can be seen that both the global and the namespace-local admission policies were evaluated for the specified image's admission scan result. The error response indicates that the container image was blocked by both admission policies.

5.2.1 Performance Impact

In this section, we measure the performance impact the policy evaluation has on pod deployment, i.e. the delay that is introduced by the pod admission webhook. In this experiment, combinations of the following scenarios are measured:

- Pods with a one or two containers.
- One up to four simultaneous admission policies.

The admission policy that will be used is the "Severity Thresholds" one, described in the qualitative evaluation. This is a complex admission policy that iterates over all vulnerabilities in the admission scan result, performs string comparison operations and creates a map of vulnerability counts by severity.

All measurements were repeated 100 times and the mean times were calculated. The results are presented in table 5.1. The script that was used to automate the process is presented in appendix A. All times are from the perspective of the client.

The data demonstrate that the performance impact is minimal, typically within a few milliseconds: 100ms and 220ms per admission policy for the 1- and 2-image workloads, respectively. This linear scaling behavior is clearly shown in figure 5.7.

Table 5.1: Admission Policy Evaluation Mean Times.

Admission Policies	Container Images	Iterations	Mean Time (Seconds)
0	1	100	0.2933
1	1	100	0.4091
2	1	100	0.5008
3	1	100	0.5968
4	1	100	0.7015
0	2	100	0.3681
1	2	100	0.5057
2	2	100	0.7213
3	2	100	0.9331
4	2	100	1.1441

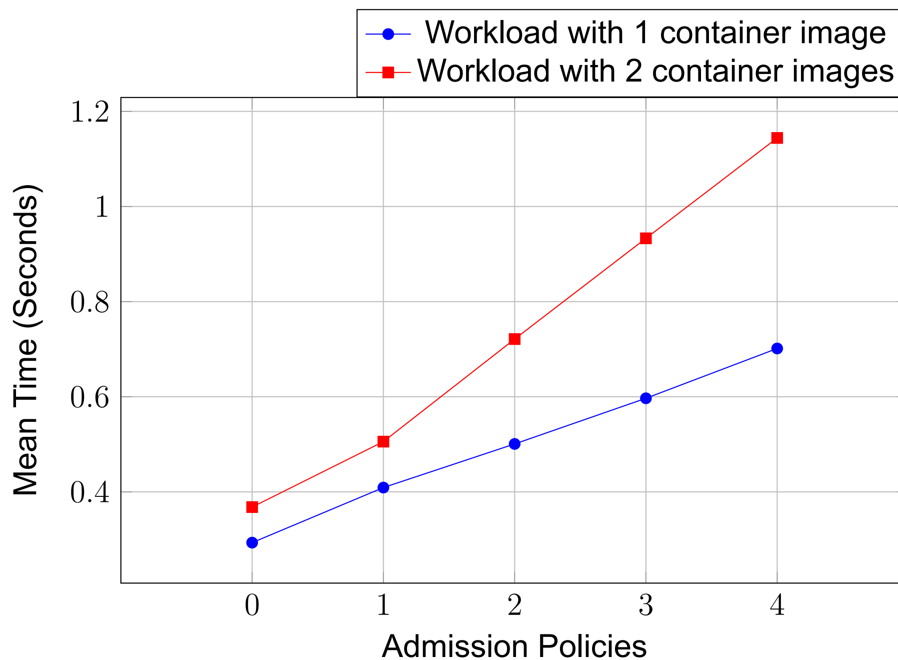


Figure 5.7: Admission Policy Evaluation Mean Times.

6. CONCLUSION AND FUTURE WORK

In this thesis, we presented the design and implementation of two practical Kubernetes operators, discussing the challenges that we encountered, design decisions that had to be made and technical implementation details. The first operator, KCIVM Visibility Operator, deals with the need to monitor known vulnerabilities in a Kubernetes cluster in order to plan their timely patching. The second operator, KCIVM Admission Operator, provides enforcement of policies regarding deployment of vulnerable container images. Together, they form an overall solution to managing container image vulnerabilities in Kubernetes.

Our evaluation showed that KCIVM Admission Operator had minimal performance impact on the deployment operations of the Kubernetes cluster. The KCIVM Visibility Operator was not expected to have any impact at all, as it runs in the background without interfering with other operations.

Future work may explore the possibility of incorporating artificial intelligence in the solution. Artificial intelligence could be used to predict whether a vulnerability in a package actually affects the application in the scanned container images and automatically mark it as ignored, thus reducing the amount of false positive tracked vulnerabilities.

Moreover, additional data stores and dashboards can be implemented for the KCIVM Visibility Operator. The alternatives that we considered during the design and development of the solution were an SQL database as the data store and a Grafana[1] dashboard using that SQL database as its data source.

Finally, the policy evaluation part of the KCIVM Admission Operator could be replaced by a dedicated policy controller for Kubernetes. This could allow our solution to be better integrated into existing environments. Policy controllers that were considered during the development were Gatekeeper[2] and Kyverno[3].

[1] https://grafana.com/
[2] https://open-policy-agent.github.io/gatekeeper/
[3] https://kyverno.io/

ABBREVIATIONS - ACRONYMS

API	Application Programming Interface
CIA	Confidentiality, Integrity and Availability
CNCF	Cloud Native Computing Foundation
CRD	Custom Resource Definition
CVE	Common Vulnerabilities and Exposures
CVSS	Common Vulnerability Scoring System
CWE	Common Weakness Enumeration
DB	Database
EOSL	End of Service Life
FQDN	Fully Qualified Domain Name
ID	Identifier
JSON	JavaScript Object Notation
KCIVM	Kubernetes-native Container Image Vulnerability Management
OCI	Open Container Initiative
OS	Operating System
RBAC	Role-Based Access Control
SDK	Software Development Kit
UTC	Coordinated Universal Time
YAML	Yet Another Markup Language

APPENDIX A. EVALUATION

The following code snippet presents the script that was used to take the measurements for the quantitative evaluation of the KCIVM Admission Operator:

```
import subprocess
import time
from tabulate import tabulate

ITERATION_COUNT = 100


def benchmark_command(iter_count):
   results = []
   for i in range(iter_count):
      # Clean up before the main command.
      subprocess.run(
         ["kubectl", "-n", "myapp", "delete", "pod", "vulnerable-↩
            workload"]
      )

      start_time = time.perf_counter()
      try:
         result = subprocess.run(
            ["kubectl", "apply", "-f", "myapp-vulnerable-workload↩
               .yaml"],
            stdout=subprocess.DEVNULL,
            stderr=subprocess.DEVNULL,
            check=True,
         )
      except subprocess.CalledProcessError as e:
         # Code 1 is returned if request is denied by the ↩
            validating webhook.
         if e.returncode != 1:
            raise e

      end_time = time.perf_counter()
      elapsed_time = end_time - start_time

      results.append(elapsed_time)
   return results


if __name__ == "__main__":
```

```
results = benchmark_command(ITERATION_COUNT)

headers = ["Run #", "Time (seconds)"]
print(tabulate(enumerate(results, start=1), headers=headers, ↩
   tablefmt="grid"))

mean_time = sum(results) / len(results)
print(f"Mean time (seconds): {mean_time}")
```

REFERENCES

[1] Cluster Architecture. Kubernetes. Accessed Sep. 22, 2024. [Online]. Available: https://kubernetes.io/docs/concepts/architecture/

[2] CVSS v3.1 Specification Document. Forum of Incident Response and Security Teams (FIRST). Accessed Sep. 22, 2024. [Online]. Available: https://www.first.org/cvss/v3.1/specification-document

[3] CVSS v4.0 Specification Document. Forum of Incident Response and Security Teams (FIRST). Accessed Sep. 22, 2024. [Online]. Available: https://www.first.org/cvss/v4.0/specification-document

[4] Image Format Specification. Open Container Initiative. Accessed Sep. 22, 2024. [Online]. Available: https://github.com/opencontainers/image-spec/blob/main/spec.md

[5] Objects In Kubernetes. Kubernetes. Accessed Sep. 22, 2024. [Online]. Available: https://kubernetes.io/docs/concepts/overview/working-with-objects/

[6] Workloads. Kubernetes. Accessed Sep. 22, 2024. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/

[7] Configuration. Kubernetes. Accessed Sep. 22, 2024. [Online]. Available: https://kubernetes.io/docs/concepts/configuration/

[8] Common Vulnerabilities and Exposures. MITRE. Accessed Sep. 22, 2024. [Online]. Available: https://www.cve.org/

[9] CVE Glossary. MITRE. Accessed Sep. 22, 2024. [Online]. Available: https://www.cve.org/ResourcesSupport/Glossary

[10] Common Vulnerability Scoring System. Forum of Incident Response and Security Teams (FIRST). Accessed Sep. 22, 2024. [Online]. Available: https://www.first.org/cvss/

[11] Trivy Operator. Aqua Security. Accessed Sep. 22, 2024. [Online]. Available: https://aquasecurity.github.io/trivy-operator/

[12] Trivy. Aqua Security. Accessed Sep. 22, 2024. [Online]. Available: https://aquasecurity.github.io/trivy/